Mike McQuaid

Git in Practice

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Git in Practice
Version 2

Welcome

Hi,

Thanks for purchasing the MEAP of *Git In Practice*. I hope that what you'll get access to will be of immediate use to you and, with your help, the final book will be great!

Git is a powerful distributed version control system but it can sometimes be difficult to understand what is going on behind the scenes or how you've found yourself in a particular state. In Git In Practice by the end of the book you should understand:

- The internals and confusing jargon Git will sometimes expose
- A wide array of commands for interacting with Git repositories
- Workflows for configuring, using Git and GitHub effectively for teams, products and open-source project

What you'll be getting immediately is Part 1 of the book with a chapter from Part 2. Next month, you'll get a chapter from 3.

Part 1 (Chapters 1-3) will fly through the basics of distributed version control using Git while teaching you the underlying concepts that are often misunderstood or omitted in beginners guides. You may know how to use Git already but I'd encourage you to persevere with this anyway; it's setting a good foundation which the rest of the book will build upon.

Chapter 4 (from Part 2) will show you various ways of interacting with the file system using Git in a problem/solution format.

Chapter 10 (from Part 3) discusses two approaches for team branch workflows using Git; rebasing and merging. These are compared by contrasting the approaches taken by two successful open-source projects: the CMake build system and the Homebrew OS X package manager.

Part 2 will continue to introduce more Git commands in a problem/solution format and Part 3 will introduce more high-level workflows for managing projects and teams when using Git.

Please let me know your thoughts on what's been written so far and what you'd like to see in the rest of the book. Your feedback will be invaluable in improving Git In Practice.

Thanks again for your interest and for purchasing the MEAP!

Mike McQuaid

brief contents

1 Introduction to local Git

In this chapter you will learn how and why to use Git as a local version control system by covering the following topics:

- Why Git was created
- How to create a new local Git repository
- How to commit files into a Git repository
- How to view the history of a Git repository
- How to use gitk/GitX to visualize the history of a Git repository
- How to view the differences between Git commits

Let’s start by learning why Git is so widely used by programmers.

1.1 Why do programmers use Git?

Git was created by a programmer to be used by programmers. Linus Torvalds, the creator of the Linux kernel, started writing Git in 2008 with the goal of having a distributed, open-source, high-performance and hard to corrupt version control system for the Linux kernel project to use. Within a week Git’s source code was hosted inside the Git version control system and within two and a half months the version 2.6.12 of the Linux kernel was released using Git.

From its initial creation for the Linux kernel Git is now used by many other open source projects, all sizes of companies and large "social coding" Git hosting sites such as GitHub.

Git is my preferred method of software source code control. I also use it for versioning plain text files such as the text for this book. Git has many strengths over other methods of source control. Git stores all of the history, branches and commits locally. This means adding new versions or querying history doesn’t require a network connection. Git’s history log viewing and branch creation is near-instant compared to e.g. Subversion's which is sluggish even on a fast network connection. As Git stores changes locally you are not constrained by the work of others when working in a team. For example, remote branch updates and merges can be done independently so you can continue your edit/commit workflow without interruptions whilst still downloading and examining any changes made by others. In Git you can modify the history of branches and even rewrite commit data across the entire repository. It's often useful to be able to make lots of small commits which are later turned into a single commit or make commit messages contain more information after the fact and Git's history rewriting enables this. Even with this flexibility in Git every commit has a unique reference that survives rewriting or changes and won't be purged until it's missing from all branches for at least 30 days. This means it's very hard to accidentally lose work if it has been committed.

Git's main downsides are that the command-line application's interface is often counterintuitive; it makes frequent use of jargon that can only be adequately explained by understanding Git's internals. Additionally Git's official documentation can be hard to follow; it also uses jargon and has to detail the large number of options for Git's commands. To the credit of the Git community both the UI and documentation around Git have improved hugely over the years. This book will help you understand Git's jargon and the Git's internal operations; these should help you to understand why Git does what it does when you run the various Git commands.

Despite these downsides the strengths of Git have proved too strong for many software projects to resist. Google, Microsoft, Twitter, LinkedIn and Netflix all use Git as well as open-source projects such as the Linux kernel (the first Git user), Perl, PostgreSQL, Android, Ruby on Rails, Qt, GNOME, KDE, Eclipse and X.org.

Many of the above projects and many users of Git have also been introduced to and use Git through a Git hosting provider. My favorite is GitHub but there are various other paid and free alternatives available.

1.2 Initial setup

Once you've installed Git (as detailed in Appendix B) the first thing you need to do is to tell Git your name and email (particularly before creating any commits). Rather than usernames Git uses a name and email address to identify the author of a commit.

My name is Mike McQuaid and email address is `mike@mikemcquaid.com` so I would run:

Listing 1.1 Set name and email address

```
# git config --global user.name "Mike McQuaid"
# git config --global user.email "mike@mikemcquaid.com"
```

❶ **a run command**

❷ **command output**

All command output listings in this book show:

- "run command (1)" prefixed with a #.
- "command output (2)" following the run command. In this case there was no output so this line has been left intentionally blank.

NOTE **How can I follow the listings?**
You can follow listings as you work through the book by running the listed commands in a Terminal/console/Git Bash and comparing the output to the listing.

Now Git is setup to use Git you need to initialize a Git *repository* on your local machine.

1.3 Creating a repository: git init

1.3.1 Background

A Git *repository* is the local collection of all the files related to a particular Git version control system and contains a `.git` subdirectory in its root. Git keeps track of the state of the files within the repository's directory on disk.

Git repositories store all their data on your local machine. Making commits, viewing history and requesting differences between commits are all local operations that do not require a network connection. This makes all these operations much faster in Git than with centralized version control systems such as Subversion.

Typically you create a new repository by downloading another repository that already exists (known as *cloning* by Git and introduced in Chapter 2) but let's start by initializing an empty, new local repository.

1.3.2 Problem

You wish to create a new local Git repository in a new subdirectory named `GitInPracticeRedux.

1.3.3 Solution

1. Change to the directory you wish to contain your new repository directory e.g. `cd /Users/mike/`.
2. Run `git init GitInPracticeRedux`.

Listing 1.2 Initializing a Git repository

```
# cd /Users/mike/
# git init GitInPracticeRedux

Initialized empty Git repository in
/Users/mike/GitInPracticeRedux/.git/
```

❶ **requested name**

❷ **repository path**

You have initialized a new local Git repository named `GitInPracticeRedux` accessible at e.g. `/Users/mike/GitInPracticeRedux`.

1.3.4 Discussion

NOTE **Where can I see the full syntax references for Git commands?** All `git` commands referenced in this book have complete references to all their possible syntax and arguments in Git's help. This can be accessed for a given command by running the command suffixed with `--help` e.g. `git init --help`. This book will cover only the most common and useful commands and arguments.

`git init` can be run without any arguments to create the local Git repository in the current directory.

.GIT SUBDIRECTORY

Under the new Git repository directory a `.git` subdirectory at e.g `/Users/mike/GitInPracticeRedux/.git/` is created with various files and directories under it.

NOTE **Why is the `.git` directory not visible?**
On some operating systems directories starting with a `.` such as `.git` will be hidden by default. They can still be accessed in the console using their full path (e.g. `/Users/mike/GitInPracticeRedux/.git/`) but will not show up in file listings in file browsers or by running e.g. `ls /Users/mike/GitInPracticeRedux/`.

Let’s view the contents of the new Git repository by changing to the directory containing the Git repository and running the `find` command.

Listing 1.3 Listing files created in a new repository

```
# cd /Users/mike/ && find GitInPracticeRedux

GitInPracticeRedux/.git/config                                    ❶ local configuration
GitInPracticeRedux/.git/description                               ❷ description file
GitInPracticeRedux/.git/HEAD                                      ❸ HEAD pointer
GitInPracticeRedux/.git/hooks/applypatch-msg.sample               ❹ event hooks
GitInPracticeRedux/.git/hooks/commit-msg.sample
GitInPracticeRedux/.git/hooks/post-update.sample
GitInPracticeRedux/.git/hooks/pre-applypatch.sample
GitInPracticeRedux/.git/hooks/pre-commit.sample
GitInPracticeRedux/.git/hooks/pre-push.sample
GitInPracticeRedux/.git/hooks/pre-rebase.sample
GitInPracticeRedux/.git/hooks/prepare-commit-msg.sample
GitInPracticeRedux/.git/hooks/update.sample
GitInPracticeRedux/.git/info/exclude                              ❺ excluded files
GitInPracticeRedux/.git/objects/info                              ❻ object information
GitInPracticeRedux/.git/objects/pack                              ❼ pack files
GitInPracticeRedux/.git/refs/heads                                ❽ branch pointers
GitInPracticeRedux/.git/refs/tags                                 ❾ tag pointers
```

Git has created files for:

- "local configuration (1)" of the local repository.
- "description file (2)" to describe the repository for those created for use on a server.
- "HEAD pointer (3)", "branch pointers (8)" and "tag pointers (9)" which point to commits.
- "*event hooks* (4)" samples; scripts that run on defined events e.g. pre-commit is run before every new commit is made.
- "excluded files (5)" which manages files which should be excluded from the repository.
- "object information (6)" and "pack files (7)" which are used for object storage and reference.

You shouldn’t edit any of these files directly until you have a more advanced understanding of Git (or, in my experience, never at all). You will instead modify these files and directories by interacting with the Git repository through Git's filesystem commands introduced in Chapter 3.

1.4 Creating a new commit: git add, git commit

To do anything useful in Git we first need one or more commits in our repository.

A *commit* is created from the changes to one or more files on disk. The typical workflow is that you will change the contents of files inside a repository, review the *diffs*, add them to the *index*, create a new commit from the contents of the index and repeat this cycle.

Git's index is a staging area used to build up new commits. Rather than requiring all changes in the working tree make up the next commit Git allows files to be added incrementally to the index. The add/commit/checkout workflow can be seen in Figure 1.1.

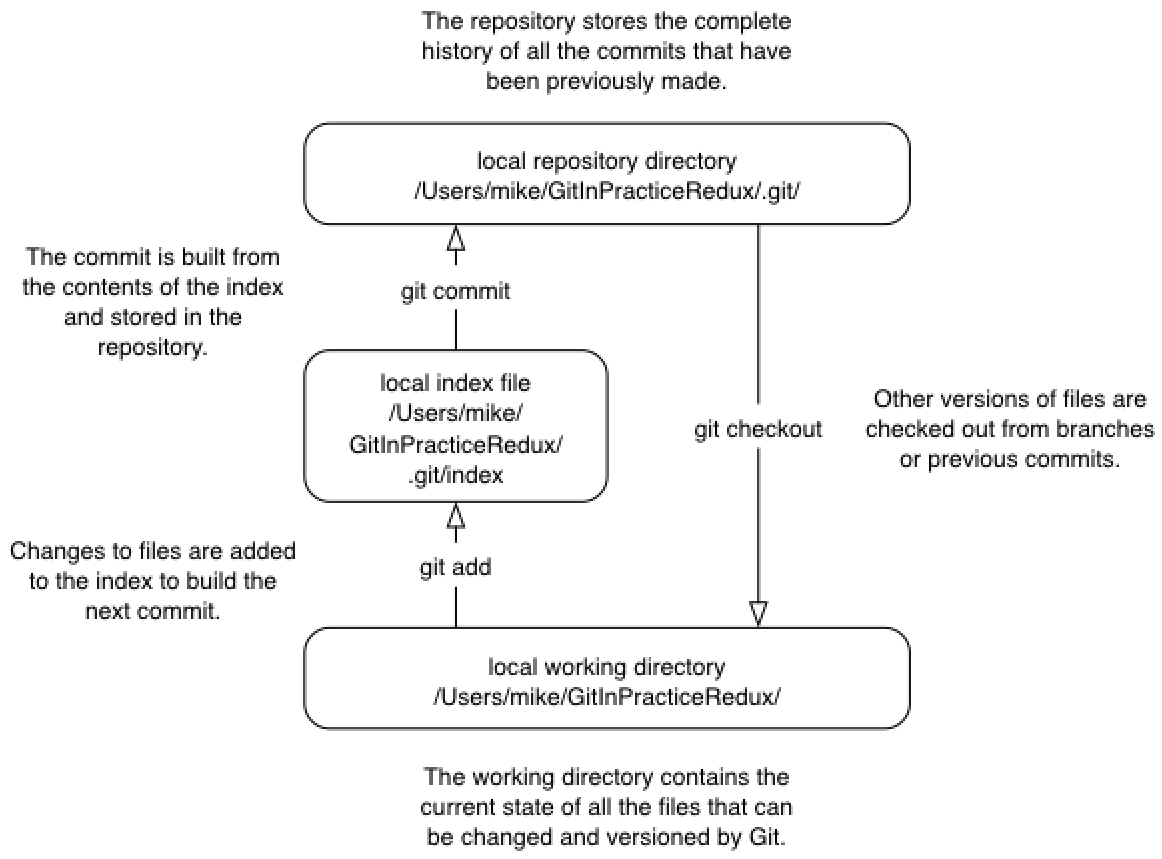

The repository stores the complete history of all the commits that have been previously made.
local repository directory /Users/mike/GitInPracticeRedux/.git/
The commit is built from the contents of the index and stored in the repository.
git commit
local index file /Users/mike/GitInPracticeRedux/.git/index
git checkout
Other versions of files are checked out from branches or previous commits.
Changes to files are added to the index to build the next commit.
git add
local working directory /Users/mike/GitInPracticeRedux/
The working directory contains the current state of all the files that can be changed and versioned by Git.

Figure 1.1 Git add/commit/checkout workflow

1.4.1 Building a new commit in the index staging area: git add

BACKGROUND

Git does not add anything to the index without your instruction. As a result, the first thing you have to do with a file you want to include in a Git repository is request Git add it to the index.

PROBLEM

You wish to add an existing file `GitInPractice.asciidoc` to the index staging area for inclusion in the next commit.

SOLUTION

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Ensure the file `GitInPractice.asciidoc` is in the current directory.
3. Run `git add GitInPractice.asciidoc`. There will be no output.

You have added the `GitInPractice.asciidoc` to the index. If this has been successful then the output of running `git status` should resemble:

Listing 1.4 Adding a file to the index

```
# git add GitInPractice.asciidoc
# git status

# On branch master                                    (1) default branch
#                                                          output
# Initial commit                                      (2) first commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   GitInPractice.asciidoc              (3) new file in index
#
```

In the status output:

- "default branch output (1)" is the first line of `git status` output (which unfortunately, like the run commands, is also always prefixed with a `#`). It shows the current *branch* which, by default, is always `master`. Do not worry about creating branches for now, this will be covered in Chapter 2.
- "first commit (2)". The "Initial commit" is shown to indicate that no commits have yet been made and the `git add` is being used to build the first commit.
- "new file in index (3)" shows the new file that you’ve just added to the index (the staging area for the next commit).

DISCUSSION

`git add` can also be passed directories as arguments instead of files. You can add everything in the current directory and its subdirectories by running `git add .`.

When a file is added to the index a file named `.git/index` is created (if it does not already exist). The added file contents and metadata are then added to the index file. You have requested two things of Git here:

1. for Git to track the contents of the file as it changes (this is not done without an explicit `git add`).
2. the contents of the file when `git add` was run should be added to the index, ready to create the next commit.

NOTE **Does `git add` need run more than once?**
Unlike some other version control systems `git add` may need to be run more than once for any particular file. It is not saying just for Git to add this file to the repository but to add the current contents of the file to the index staging area to build the next commit.

Now that the contents of the file have been added to the index you're ready to commit it.

1.4.2 Committing changes to files: git commit

BACKGROUND

Making *commit* stores the changes to one or more files. Each commit contains a message entered by the author, details of the author of the commit, a unique commit reference (in Git these are *SHA-1 hashes* e.g. `86bb0d659a39c98808439fadb8dbd594bec0004d`), a pointer to the preceding commit (known as the *parent commit*), the date the commit was created and a pointer to the contents of files when the commit was made. The file contents are typically displayed as the *diff* (the differences between the files before and the files after the commit).

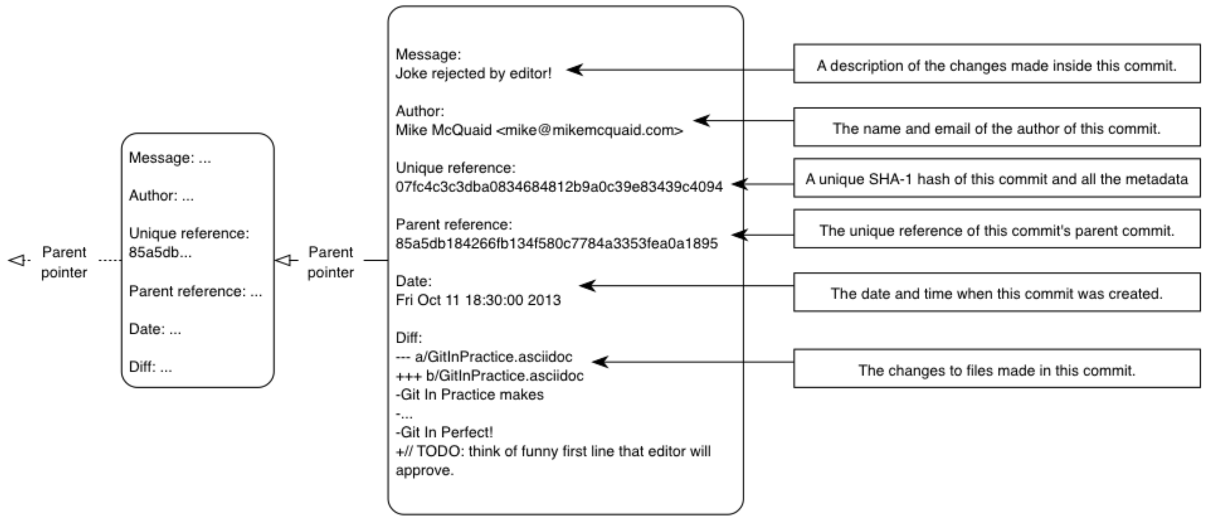

Parent pointer
Message: ...
Author: ...
Unique reference: 85a5db...
Parent reference: ...
Date: ...
Diff: ...
Parent pointer
Message: Joke rejected by editor!
Author: Mike McQuaid <mike@mikemcquaid.com>
Unique reference: 07fc4c3c3dba0834684812b9a0c39e83439c4094
Parent reference: 85a5db184266fb134f580c7784a3353fea0a1895
Date: Fri Oct 11 18:30:00 2013
Diff:
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
-Git In Practice makes
-...
-Git In Perfect!
+// TODO: think of funny first line that editor will approve.
A description of the changes made inside this commit.
The name and email of the author of this commit.
A unique SHA-1 hash of this commit and all the metadata
The unique reference of this commit's parent commit.
The date and time when this commit was created.
The changes to files made in this commit.

Figure 1.2 A typical commit broken down into its parts

NOTE **Why do the arrows point backwards?**
As you may have noticed Figure 1.2 uses arrows pointing from commits to their previous commit. The reason for this is that commits contain a pointer to the *parent commit* and not the other way round; when a commit is made it has no idea what the next commit will be yet.

PROBLEM

You wish to commit the contents of an existing file `GitInPractice.asciidoc` which has already been added to the index staging area. After this, you wish to make modifications to the file and commit them.

SOLUTION

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Ensure the file `GitInPractice.asciidoc` is in the current directory and that its changes were staged in the index with `git add`.
3. Run `git commit --message 'Initial commit of book.'`. The output should resemble:

Listing 1.5 Committing changes staged in the index

```
# git commit --message 'Initial commit of book.'

[master (root-commit) 6576b68] Initial commit of book.   ❶
 1 file changed, 2 insertions(+)   ❷
 create mode 100644 GitInPractice.asciidoc   ❸
```

❶ branch, SHA-1, message
❷ changed files, lines
❸ new file created

From the commit output:

- "branch, SHA-1, message (1)" shows the name of the branch that the commit was made (the default, `master`), the shortened SHA-1 (`6576b68`) and the commit message. The `(root-commit)` means the same as the `Initial commit` you saw earlier. It is only shown for the first commit in a repository and means it has no parent commit.
- "changed files, lines (2)" shows the number of files changed and the number of lines inserted or deleted across all the files in this commit.
- "new file created (3)" shows that a new file was created and the Unix file mode (`100644`). The file mode is related to Unix file permissions and the `chmod` command but are not important in understanding how Git works so can be safely ignored.

You have made a new commit containing `GitInPractice.asciidoc`.

NOTE **What is a SHA-1 hash?**
A "SHA-1 hash" is a secure hash digest function that is used extensively inside of Git. It outputs a 160-bit (20-byte) hash value which is usually displayed as a 40 character hexadecimal string. The hash is used to uniquely identify commits by Git by their contents and metadata. They is used instead of incremental revision numbers (like in Subversion) due to the distributed nature of Git. When you commit locally Git cannot know whether your commit occurred before or after another commit on another machine so it cannot use ordered revision numbers. As the full 40 characters are rather unwieldy Git will often show shortened SHA-1s (as long as they are unique in the repository). Anywhere that Git accepts a SHA-1 unique commit reference it will also accept the shortened version (as long as the shortened version is still unique within the repository).

Let’s create another commit.

1. Modify `GitInPractice.asciidoc` and stage the changes in the index with `git add`.
2. Run `git commit --message 'Add opening joke. Funny?`. The output should resemble:

Listing 1.6 Making a second commit

```
# git add GitInPractice.asciidoc
# git commit --message 'Add opening joke. Funny?'

[master 6b437c7] Add opening joke. Funny?  ❶
 1 file changed, 3 insertions(+), 1 deletion(-)  ❷
```

❶ branch, SHA-1, message
❷ changed files, lines

From the second commit output:

- "branch, SHA-1, message (1)" has a different shortened SHA-1 as this is a new commit with different contents and metadata. No `(root-commit)` is shown as this second commit has the first as its parent.
- "changed files, lines (2)" shows three insertions and one deletion because Git treats the modification of a line as the deletion of an old line and insertion of a new one.

You have made modifications to `GitInPractice.asciidoc` and committed them.

DISCUSSION

The `--message` flag for `git commit` can be abbreviated to `-m`. If this flag is omitted then Git will open a text editor (specified by the `EDITOR` or `GIT_EDITOR` environment variables) to prompt you for the commit message. These variables will also be used by other commands later in the book (such as interactive rebase in Chapter 6) when requesting text input.

`git commit` can be called with `--author` and `--date` flags to override the auto-set metadata in the new commit.

`git commit` can be called with a path (like `git add`) to do the equivalent of an add followed immediately by a commit. It can also take the `--all` (or `-a`) flags to add all changes to files tracked in the repository into a new commit. Although these methods all save time they tend to result in larger (and therefore worse) commits so I recommend avoiding their use until you’ve got used to using them separately. Reasons small commits are better than large ones are covered in Section 1.6.1.

OBJECT STORE

Git is a version control system built on top of an *object store*. Git creates and stores a collection of objects when you commit. The object store is stored inside the Git *repository*.

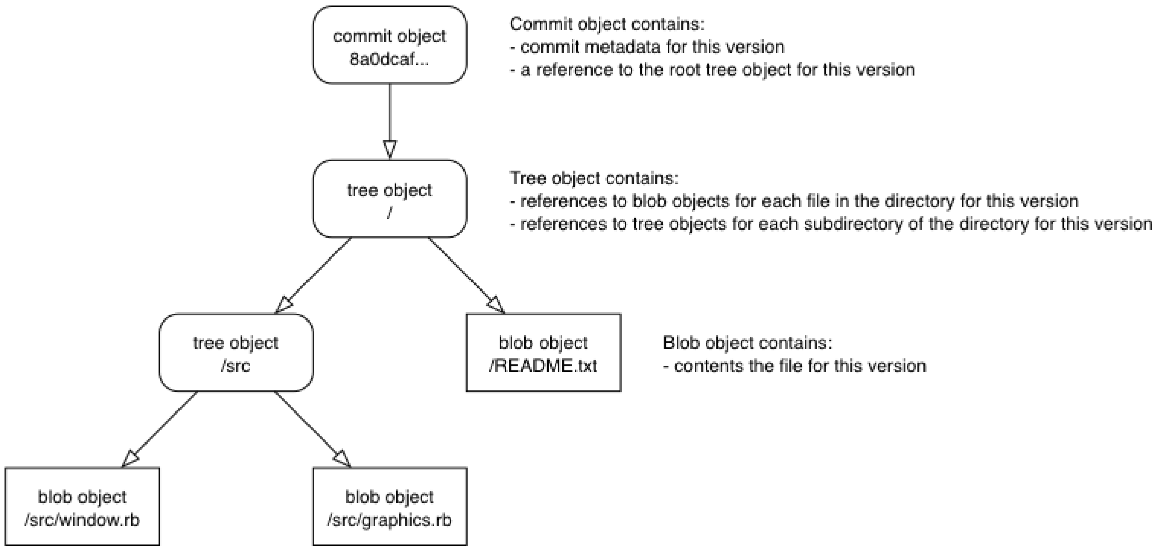

commit object
8a0dcaf...
Commit object contains:
- commit metadata for this version
- a reference to the root tree object for this version
tree object
/
Tree object contains:
- references to blob objects for each file in the directory for this version
- references to tree objects for each subdirectory of the directory for this version
tree object
/src
blob object
/README.txt
Blob object contains:
- contents the file for this version
blob object
/src/window.rb
blob object
/src/graphics.rb

Figure 1.3 Commit, blob and tree objects

In Figure 1.3 you can see the main Git objects we're concerned with: *commits*, *blobs* and *trees*. There is also a *tag* object but don't worry about tags until they are introduced in Chapter 2. Commit objects were covered in Figure 1.2 and you saw that they store metadata and referenced file contents. The file contents reference is actually a reference to a *tree object*. A tree object stores a reference to all the *blob objects* at a particular point in time and other tree objects if there are any subdirectories. A blob object stores the contents of a particular version of a particular single file in the Git repository.

NOTE **Should objects being interacted with directly?**
When using Git you should never need to interact with objects or object files directly. The terminology of *blobs* and *trees* are not used regularly in Git or in this book but it's useful to remember what these are so you can build a conceptual understanding of what Git is doing internally. When things go well this should be unnecessary but when we start to delve into more advanced Git functionality or Git spits out a baffling error message then remembering *blobs* and *trees* may help you work out what has happened.

PARENT COMMITS

Every commit object points to its *parent commit*. The parent commit in a linear, branch-less history will be the one that immediately preceded it. The only commit that lacks a parent commit is the *initial commit*; the first commit in the repository. By following the parent commit, its parent, its parent and so on you will always be able to get back from the current commit to the initial commit. You can see an example of parent commit pointers in Figure 1.4.

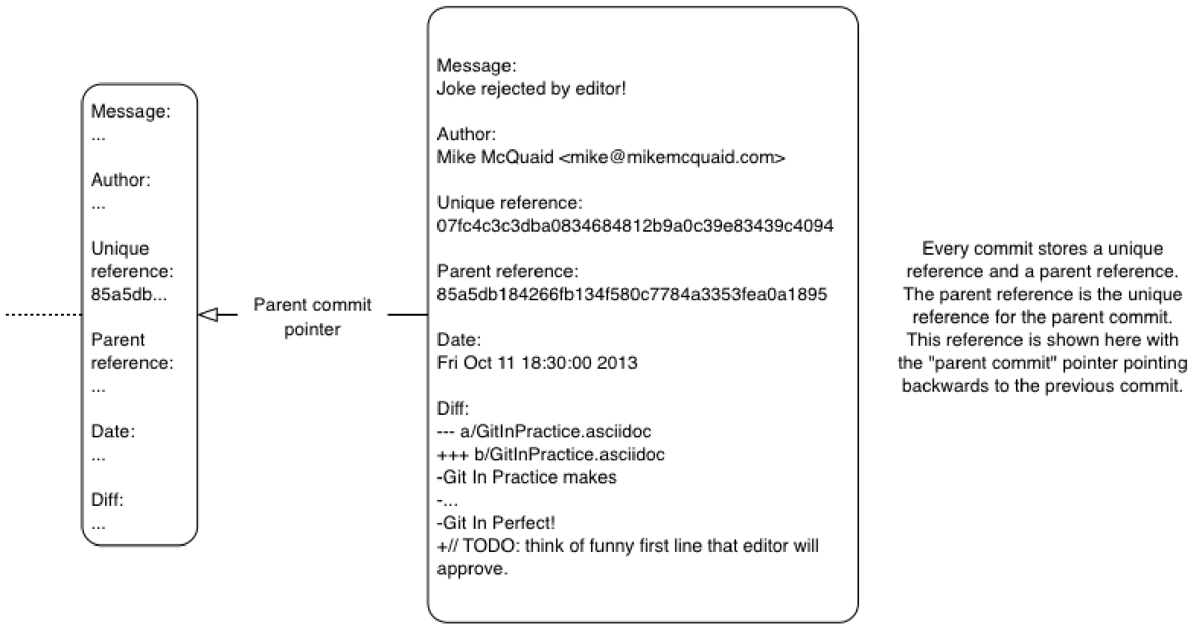

Message:
...
Author:
...
Unique reference:
85a5db...
Parent reference:
...
Date:
...
Diff:
...
Parent commit pointer
Message:
Joke rejected by editor!
Author:
Mike McQuaid <mike@mikemcquaid.com>
Unique reference:
07fc4c3c3dba0834684812b9a0c39e83439c4094
Parent reference:
85a5db184266fb134f580c7784a3353fea0a1895
Date:
Fri Oct 11 18:30:00 2013
Diff:
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
-Git In Practice makes
-...
-Git In Perfect!
+// TODO: think of funny first line that editor will approve.
Every commit stores a unique reference and a parent reference. The parent reference is the unique reference for the parent commit. This reference is shown here with the "parent commit" pointer pointing backwards to the previous commit.

Figure 1.4 Parent commit pointers

Now that we have two commits and have learned how they are stored we can start looking at Git's history.

1.5 Viewing history: git log, gitk, gitx

1.5.1 Background

The *history* in Git is the complete list of all commits made since the repository was created. The history also contains the references to any *branches*, *tags* and *merges* made within the repository. These three will be covered in Chapter 2.

When you are using Git you will find yourself regularly checking the history; sometimes to remind yourself of your own work, sometimes to see why other changes were made in the past and sometimes reading new changes than have been made by others. In different situations different pieces of data will be interesting but all pieces of data will always be available for every commit.

As you may have got a sense of already: how useful the history is relies very much on the quality of the data entered into it. If I made a commit once per year with huge numbers of changes and a commit message of "fixes" then it would be fairly hard to use the history effectively. Ideally commits are small and well-described; follow these two rules and having a complete history becomes a very useful tool.

NOTE **Why are small commits better?**

Sometimes, however, it is desirable to pick only some changed files (or even some changed lines within files) to include in a commit and leave the other changes for adding in a future commit. Commits should be kept as small as possible. This allows their message to describe a single change rather than multiple changes that are unrelated but were worked on at the same time. Small commits keep the history readable; it’s easier when looking at a small commit in future to understand exactly why the change was made. If a small commit was later found to be undesirable it can be easily reverted. This is much more difficult if many unrelated changes are clumped together into a single commit and you wish to revert a single change.

NOTE **How should commit messages be formatted?**
The commit message you entered is structured like an email. The first line of it is treated as the subject and the rest as the body. The commit subject will be used as a summary for that commit when only a single line of the commit message is shown and it should be 50 characters or less. The remaining lines should be wrapped at 72 characters or less and separated from the subject by a single, blank line. The commit message should describe what the commit does in as much detail as is useful in the present tense.

Let’s learn how to view the history of a repository.

1.5.2 Problem

You wish to view the commit history (also known as log) of a repository.

1.5.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git log`. The output should resemble:

Listing 1.7 History output

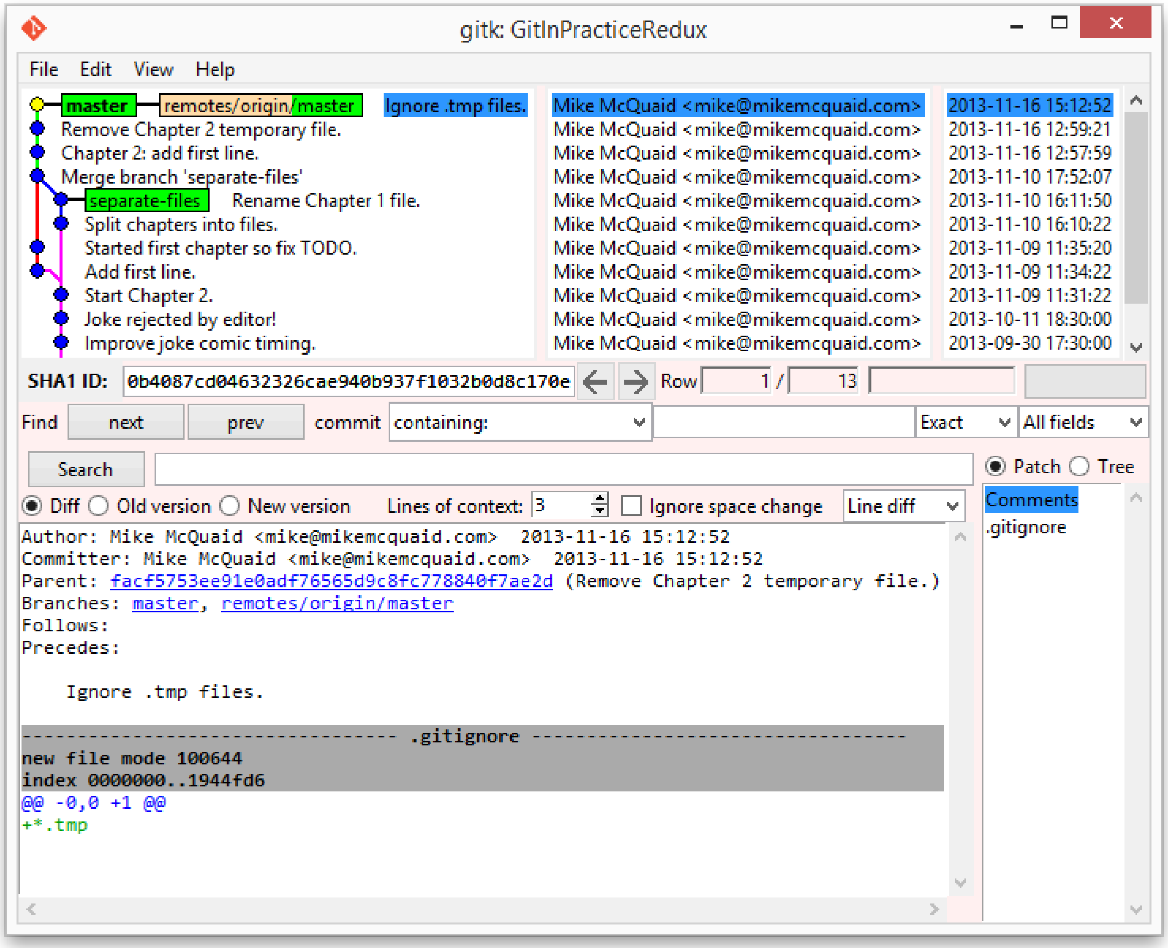
```
# git log

commit 6b437c7739d24e29c8ded318e683eca8f03a5260     (1) unique SHA-1
Author: Mike McQuaid <mike@mikemcquaid.com>        (2) commit author
Date:   Sun Sep 29 11:30:00 2013 +0100             (3) committed date

    Add opening joke. Funny?                       (4) full commit
                                                       message
commit 6576b6803e947b29e7d3b4870477ae283409ba71
Author: Mike McQuaid <mike@mikemcquaid.com>
Date:   Sun Sep 29 10:30:00 2013 +0100

    Initial commit of book.
```

The `git log` output lists all the commits that have been made on the current branch in reverse chronological order i.e. the most recent commit comes first.

- "unique SHA-1 (1)" shows the full 40 character commit reference.
- "commit author (2)" shows the name and email address set by the person who made the commit.
- "committed date (3)" shows the date and time when the commit was made.
- "full commit message (4)" first line is the commit message subject and remaining lines

are the commit message body.

It’s also useful to graphically visualize history.


gitk: GitInPracticeRedux
File Edit View Help
SHA1 ID: 0b4087cd04632326cae940b937f1032b0d8c170e
Row 1 / 13
Find next prev commit containing: Exact All fields
Search
Patch Tree
Diff Old version New version Lines of context: 3 Ignore space change Line diff
Comments
.gitignore

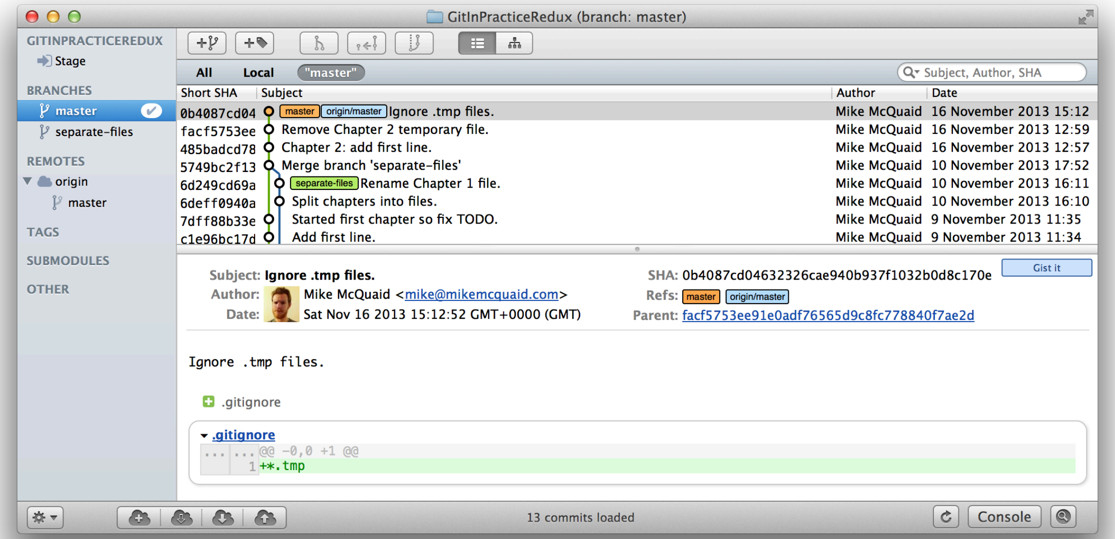
```
master  remotes/origin/master  Ignore .tmp files.   Mike McQuaid <mike@mikemcquaid.com>  2013-11-16 15:12:52
Remove Chapter 2 temporary file.                    Mike McQuaid <mike@mikemcquaid.com>  2013-11-16 12:59:21
Chapter 2: add first line.                          Mike McQuaid <mike@mikemcquaid.com>  2013-11-16 12:57:59
Merge branch 'separate-files'                       Mike McQuaid <mike@mikemcquaid.com>  2013-11-10 17:52:07
  separate-files  Rename Chapter 1 file.            Mike McQuaid <mike@mikemcquaid.com>  2013-11-10 16:11:50
  Split chapters into files.                        Mike McQuaid <mike@mikemcquaid.com>  2013-11-10 16:10:22
  Started first chapter so fix TODO.                Mike McQuaid <mike@mikemcquaid.com>  2013-11-09 11:35:20
  Add first line.                                   Mike McQuaid <mike@mikemcquaid.com>  2013-11-09 11:34:22
  Start Chapter 2.                                  Mike McQuaid <mike@mikemcquaid.com>  2013-11-09 11:31:22
  Joke rejected by editor!                          Mike McQuaid <mike@mikemcquaid.com>  2013-10-11 18:30:00
  Improve joke comic timing.                        Mike McQuaid <mike@mikemcquaid.com>  2013-09-30 17:30:00
```

```
Author: Mike McQuaid <mike@mikemcquaid.com>  2013-11-16 15:12:52
Committer: Mike McQuaid <mike@mikemcquaid.com>  2013-11-16 15:12:52
Parent: facf5753ee91e0adf76565d9c8fc778840f7ae2d (Remove Chapter 2 temporary file.)
Branches: master, remotes/origin/master
Follows:
Precedes:

    Ignore .tmp files.

---------------------------------- .gitignore ----------------------------------
new file mode 100644
index 0000000..1944fd6
@@ -0,0 +1 @@
+*.tmp
```

Figure 1.5 `gitk` on Windows 8.1

`gitk` is a tool for viewing the history of Git repositories. It is usually installed with Git but may need installed by your package manager or separately. It’s ability to graphically visualize Git’s history is particularly helpful when history becomes more complex (e.g. with merges and remote branches). It can be seen running on Windows 8.1 in Figure 1.5.

There are more attractive, up-to-date and platform-native alternatives to `gitk`. On Linux/Unix I’d instead recommend using tools such as `gitg` for gtk+/GNOME integration and `QGit` for Qt/KDE integration. These can be installed using your package manager.



Short SHA	Subject	Author	Date
0b4087cd04	master origin/master Ignore .tmp files.	Mike McQuaid	16 November 2013 15:12
facf5753ee	Remove Chapter 2 temporary file.	Mike McQuaid	16 November 2013 12:59
485badcd78	Chapter 2: add first line.	Mike McQuaid	16 November 2013 12:57
5749bc2f13	Merge branch 'separate-files'	Mike McQuaid	10 November 2013 17:52
6d249cd69a	separate-files Rename Chapter 1 file.	Mike McQuaid	10 November 2013 16:11
6deff0940a	Split chapters into files.	Mike McQuaid	10 November 2013 16:10
7dff88b33e	Started first chapter so fix TODO.	Mike McQuaid	9 November 2013 11:35
c1e96bc17d	Add first line.	Mike McQuaid	9 November 2013 11:34

Subject: **Ignore .tmp files.**
Author: Mike McQuaid <mike@mikemcquaid.com>
Date: Sat Nov 16 2013 15:12:52 GMT+0000 (GMT)
SHA: 0b4087cd04632326cae940b937f1032b0d8c170e
Refs: master origin/master
Parent: facf5753ee91e0adf76565d9c8fc778840f7ae2d

```
Ignore .tmp files.
```

.gitignore

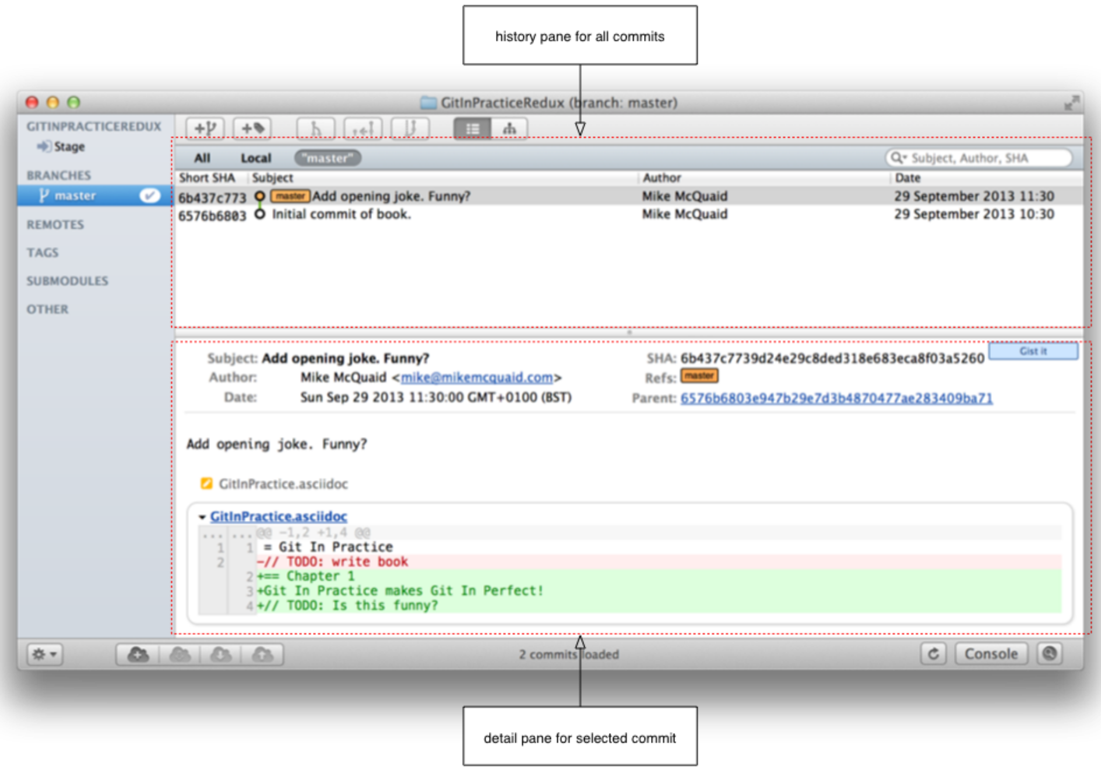
```
.gitignore
@@ -0,0 +1 @@
+*.tmp
```

Figure 1.6 GitX-dev on OS X Mavericks

On OS X there are tools such as `GitX` (and various forks of the project). As OS X is my platform of choice I'll be using screenshots of the `GitX-dev` fork of `GitX` to discuss history in this book and would recommend you use it too if you use OS X. `GitX-dev` is available at https://github.com/rowanj/gitx and can be seen in Figure 1.6.

To view the commit history with gitk or GitX:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `gitk` or `gitx`.

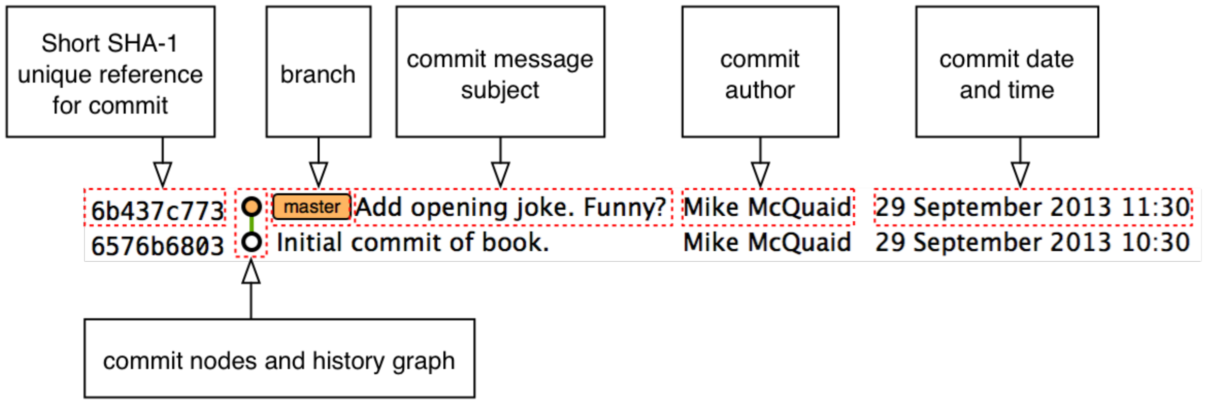


```
Short SHA   Subject                                  Author         Date
6b437c773   master Add opening joke. Funny?          Mike McQuaid   29 September 2013 11:30
6576b6803   Initial commit of book.                  Mike McQuaid   29 September 2013 10:30
```

```
Subject: Add opening joke. Funny?                    SHA: 6b437c7739d24e29c8ded318e683eca8f03a5260
Author:  Mike McQuaid <mike@mikemcquaid.com>         Refs: master
Date:    Sun Sep 29 2013 11:30:00 GMT+0100 (BST)     Parent: 6576b6803e947b29e7d3b4870477ae283409ba71

Add opening joke. Funny?

GitInPractice.asciidoc

GitInPractice.asciidoc
@@ -1,2 +1,4 @@
 = Git In Practice
-// TODO: write book
+== Chapter 1
+Git In Practice makes Git In Perfect!
+// TODO: Is this funny?
```

Figure 1.7 GitX history output

The GitX history (seen in Figure 1.7) shows similar output to `git log` but in a different format. You can also see the current branch and the contents of the current commit including the diff and parent SHA-1. There's a lot of information that doesn't differ between commits, however.


Short SHA-1 unique reference for commit
branch
commit message subject
commit author
commit date and time
commit nodes and history graph

```
6b437c773   master Add opening joke. Funny?   Mike McQuaid   29 September 2013 11:30
6576b6803   Initial commit of book.           Mike McQuaid   29 September 2013 10:30
```

Figure 1.8 GitX history graph output

In Figure 1.8 you can see the GitX history graph output. This format will be used throughout the book to show the current state of the repository and/or the previous few commits. It concisely shows the unique SHA-1, all branches (only `master` in this case), the current local branch (shown in the GUI with an orange label), the commit message subject (the first line of the commit message) and the commit's author, date and time.

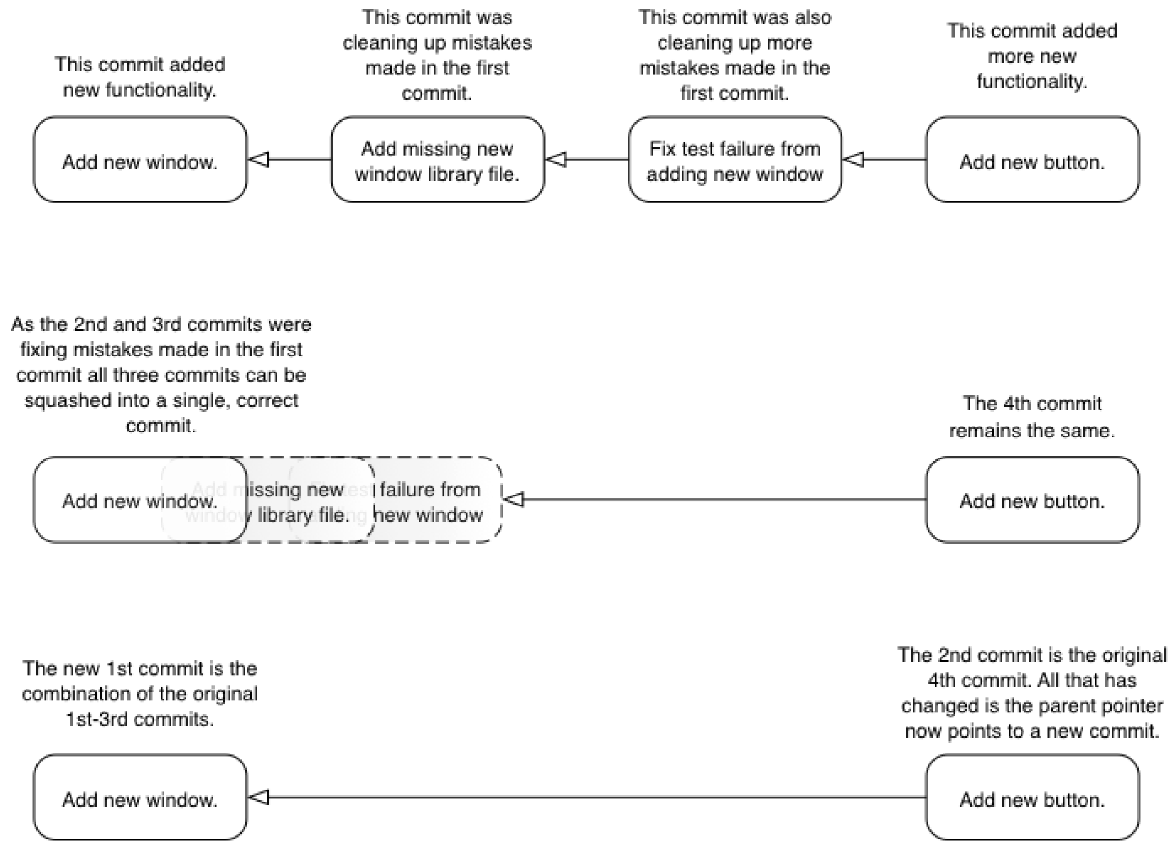
1.5.4 Discussion

`git log` can take revision or path arguments to specify the output history be shown starting at the given revision or only include changes to the requested paths.

`git log` can take a `--patch` (or `-p`) flag to show the *diff* for each commit output. It can also take `--stat` or `--word-diff` flag to show a *diffstat* or *word diff*. These terms will be explained in Section 1.7.4.

REWRITING HISTORY

Git is unusual compared to many other version control systems in that it allows history to be rewritten. This may seen surprising or worrying; after all did I not just tell you that the history contains the entire list of changes to the project over time? Sometimes you may want to highlight only broader changes to files in a version control system over a period of time instead of sharing ever single change that was made in reaching the final state.



This commit added new functionality.
This commit was cleaning up mistakes made in the first commit.
This commit was also cleaning up more mistakes made in the first commit.
This commit added more new functionality.
Add new window.
Add missing new window library file.
Fix test failure from adding new window
Add new button.
As the 2nd and 3rd commits were fixing mistakes made in the first commit all three commits can be squashed into a single, correct commit.
The 4th commit remains the same.
Add new window.
Add missing new window library file.
Fix test failure from adding new window
Add new button.
The new 1st commit is the combination of the original 1st-3rd commits.
The 2nd commit is the original 4th commit. All that has changed is the parent pointer now points to a new commit.
Add new window.
Add new button.

Figure 1.9 Squashing multiple commits into a single commit

In Figure 1.9 you see a fairly common use-case for rewriting history with Git. If you were working on some window code all morning and wanted your coworkers to see it later (or just include it in the project) then there’s no need for everyone to see the mistakes you made along the way. In Figure 1.9 the commits are *squashed* together so instead of three commits and the latter two fixing mistakes in the first commit we have squashed these together to create a single commit for the window feature. We’d only rewrite history like this if working on a separate branch that hadn’t had other work from other people relying on it yet as it has changed some parent commits (so, without intervention, other people’s commits may point to commits that no longer exist). Don’t worry too much about squashing work for now; just remember this as a situation where you may want to rewrite history. In Chapter 6 we’ll learn how to rewrite history and the cases where it is useful and safe to do so.

What we’re generally interested in when reading the history (and why we clean it up) is ensuring the changes between commits are relevant (for example don’t make changes only to revert then immediately in the next commit five minutes later), minimal and readable. These changes are known as *diffs*.

The history can give us a quick overview of all the previous commits. However, querying the differences between any two arbitrary commits can also sometimes be useful so let’s learn how to do that.

1.6 Viewing the differences between commits: git diff

1.6.1 Background

A *diff* (also known as a *change* or *delta*) is the difference between two commits. In a Git you can request a diff between any two commits, branches or tags. It’s often useful to be able to request the difference between two parts of the history for analysis. For example, if an unexpected part of the software has recently started misbehaving you may go back into the history to verify that it previously worked. If it did work previously then you may want to examine the diff between the the code in the different parts of the history to see what has changed. The various ways of displaying diffs in version control typically allow you to narrow them down per-file, directory and even committer.

1.6.2 Problem

You wish to view the differences between the previous commit and the latest.

1.6.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git diff master~1 master`. The output should resemble:

Listing 1.8 The differences between the previous commit and latest

```
# git diff master~1 master ❶

diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc ❷
index 48f7a8a..b14909f 100644 ❸
--- a/GitInPractice.asciidoc ❹
+++ b/GitInPractice.asciidoc ❺
@@ -1,2 +1,4 @@ ❻
 = Git In Practice
-// TODO: write book ❼
+== Chapter 1 ❽
+Git In Practice makes Git In Perfect! ❾
+// TODO: Is this funny?
```

❶ git diff command
❷ virtual diff command
❸ index SHA-1 changes
❹ old virtual path
❺ new virtual path
❻ diff offsets
❼ modified/deleted line
❽ modified/inserted line
❾ inserted line

The diff output contains:

- "git diff command (1)" requests Git to show the diff between the commit before the top of `master` (`master~1`) and the commit on top of `master`. Both `master~1` and `master` are *refs* and will be explained later in Section 1.7.4.
- "virtual diff command (2)" is the invocation of the Unix `diff` command that Git is simulating. Git pretends that it is actually diffing the contents two directories the "old virtual path (4)" and the "new virtual path (5)" and the "virtual diff command (2)" represents that. The `--git` flag can be ignored as it just shows this is the Git simulation and the Unix `diff` command is never run.
- "index SHA-1 changes (3)" show the difference in the contents of the working tree between these commits. This can be safely ignored other than noticing that these SHA-1s do not refer to the commits themselves.
- "old virtual path (4)" shows the simulated directory for the `master~1` commit.

- "new virtual path (5)" shows the simulated directory for the `master` commit.
- "diff offsets (6)" can be ignored; they are used by the Unix `diff` command to identify what lines the diff relates to for files that are too large to be shown in their entirety.
- "modified/deleted (7) line" shows the previous version of a line that differs between the commits. Recall that a modified line is shown as a deletion and insertion.
- "modified/inserted (8) line" shows the new version of a line that differs between the commits.
- "inserted line (9)" is a new line that was added in the latter commit.

1.6.4 Discussion

`git diff` can take path arguments after to specify the differences only requested paths.

If `git diff` is run with no arguments it shows the differences between the index staging area and the current state of the files tracked by Git i.e. any changes you've made but not yet added with `git add`.

DIFF FORMATS

Diffs are shown by default in Git (and in the above example) in a format that is known as a *unified format diff*. Diffs are used often by Git to indicate changes to files; for example when navigating through history or viewing what you are about to commit.

Sometimes it is desirable to display diffs in different formats. Two common alternatives to a typical unified format diff are a *diffstat* and *word diff*.

Listing 1.9 Diffstat format

```
# git diff --stat master~1 master

 GitInPractice.asciidoc | 4 +++-
 1 file changed, 3 insertion(+), 1 deletions(-)
```

1 one file's changes
2 all files' changes

The diffstat output contains:

- "one file's changes (1)" shows the filename that has been changed, the number of lines changed in that file and +/- characters summarizing the overall changes to the file. If multiple files were changed this would show multiple filenames and each would have the lines changed for that file and +/- characters.
- "all files' changes (2)" shows a summary of totals of the number of files changes and lines inserted/deleted across all files.

This diffstat shows the same changes as the unified format diff in the previous solution. Rather than showing the breakdown of exactly what has changed it indicates what files have changed and a brief overview of how many lines were involved in the changes. This can be useful when getting a quick overview of what has changed without needing all the detail of a normal unified format diff.

Listing 1.10 Word diff format

```
# git diff --word-diff master~1 master

diff --git a/GitInPractice.asciidoc b/GitInPractice.asciidoc
index 48f7a8a..b14909f 100644
--- a/GitInPractice.asciidoc
+++ b/GitInPractice.asciidoc
@@ -1,2 +1,4 @@
= Git In Practice
{+== Chapter 1+}
{+Git In Practice makes Git In Perfect!+}                    ❶ added line
// TODO: [-write book-]{+Is this funny?+}                    ❷ modified line
```

The word diff output contains:

- "added line (1)" is surrounded by `{+}` and shows a completely new line that was inserted.
- The "modified line (2)" has some characters that were deleted surrounded by `[-]` and some lines that were inserted surrounded by `{+}`.

This word diff shows the same changes as the unified format diff in the previous solution. A word diff is similar to a unified format diff but shows modifications per-word rather than per-line. This is particularly useful when viewing changes that are not to code but plain text; in README files we probably care more about individual word choices than knowing that an entire line has changed and the special characters (`[-]{+}`) are not used as often in prose than in code.

REFS

In Git *refs* are the possible ways of addressing individual commits. They are an easier way to refer to a specific commit or branch when specifying an argument to a Git command.

The first ref you have already seen is a branch (which is `master` by default if you haven't created any other branches). Branches are actually pointers to a specific commit. Referencing the branch name `master` is the same as referencing the SHA-1 of commit at the top of the master branch e.g. the short SHA-1

6b437c7 in the last example. Whenever you might type 6b437c7 to a command you could instead type master and vice-versa. Using branch names is quicker and easier to remember for referencing commits than always using SHA-1s.

Refs can also have modifiers appended. Suffixing a ref with ~1 is the same as saying *one commit before that ref*. For example master~1 is the penultimate commit on the master branch e.g. the short SHA-1 6576b68 in the last example. Another equivalent syntax is master^ which is the same as master~1 (and master^^ equivalent to master~2).

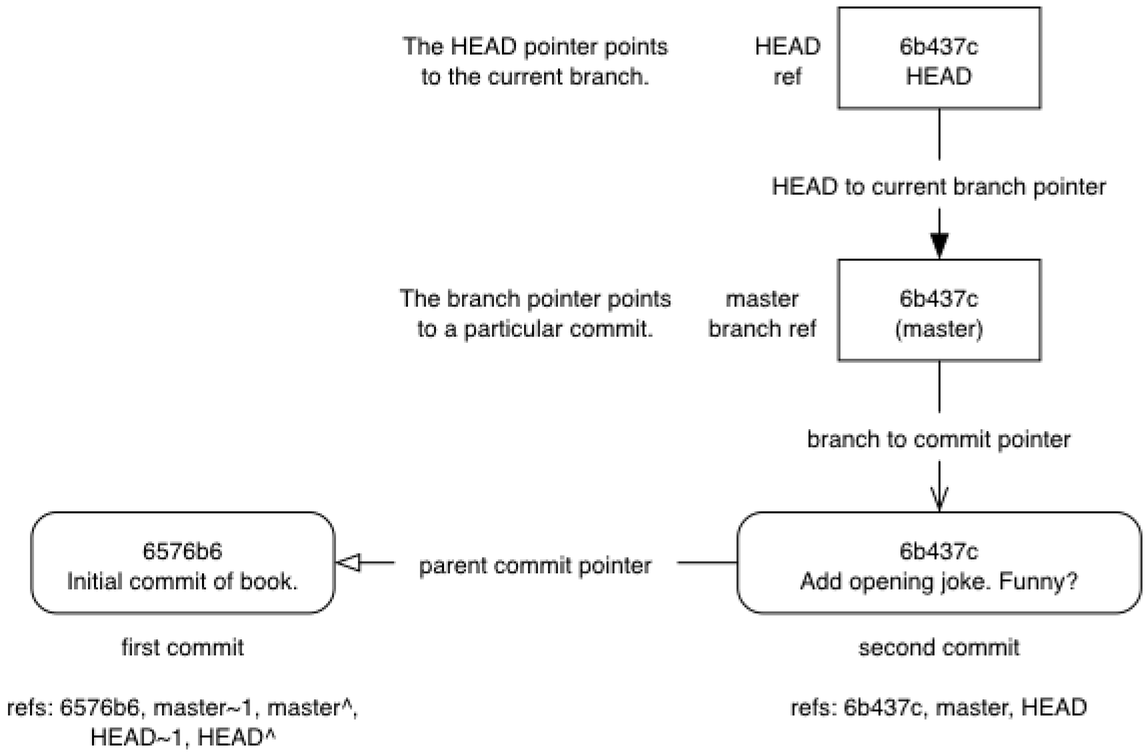

The HEAD pointer points to the current branch.
HEAD ref
6b437c HEAD
HEAD to current branch pointer
The branch pointer points to a particular commit.
master branch ref
6b437c (master)
branch to commit pointer
6576b6 Initial commit of book.
parent commit pointer
6b437c Add opening joke. Funny?
first commit
second commit
refs: 6576b6, master~1, master^, HEAD~1, HEAD^
refs: 6b437c, master, HEAD

Figure 1.10 HEAD, master and modified refs

The second ref is the string HEAD. The HEAD always points to the top of whatever you have currently checked out so almost always be the top commit of the current branch you are on. Therefore if you have the master branch checked out then master and HEAD (and 6b437c7 in the last example) are equivalent. See the master/HEAD pointers demonstrated in Figure 1.10.

These git diff invocations are all equivalent:

- `git diff master~1 master`
- `git diff master~1..master`
- `git diff master^ master`
- `git diff master^ master`

- `git diff master~1 HEAD`
- `git diff 6576b68 6b437c7`

You can also use the tool `git rev-parse` if you want to see what SHA-1 a given ref expands to:

Listing 1.11 Parsing refs

```
# git rev-parse master

6b437c7739d24e29c8ded318e683eca8f03a5260

# git rev-parse 6b437c7

6b437c7739d24e29c8ded318e683eca8f03a5260
```

There are more types of refs (such as tags and remote references) but you don't need to worry about them just now; they will be introduced in Chapter 2.

1.7 Summary

In this chapter you hopefully learned:

- Why Git is a good and high-performance version control system
- How to create a new local repository using `git init`
- How to add files to Git's index staging area using `git add`
- How to commit files to the Git repository using `git commit`
- How to view history using `git log` and `gitk`/`gitx`
- How to see the differences between commits using `git diff`
- How to use refs to reference commits

Now let's learn how to use these concepts to interact with repositories that are not stored on your local machine.

2 Introduction to remote Git

In this chapter you will learn how to retrieve and share changes with other users' Git repositories by learning the following topics:

- How to download a remote repository
- How to send/receive changes from a remote repository
- How to create and receive branches
- How to merge commits from one branch to another

As you learned in Chapter 1 it’s possible to work entirely with Git as a local version control system and never share changes with others. Usually, however, if you are using a version control system you will want to share changes with others; from simply sending files to a remote server for backup to collaborating as part of a large development team. Team collaboration will also require knowledge of how to create and interact with branches for working on different features in parallel. Let’s start by adding a remote repository.

2.1 Adding a remote repository: git remote add

2.1.1 Background

Typically when using version control you will want to share your commits with other people using other computers. With a traditional, *centralized version control system* (such as Subversion or CVS) the repository is usually stored on another machine. As you make a commit it is sent over the network, checked that it can apply (there may be other changes since you last checked) and then committed to the version control system where others can see it.

With a *distributed version control system* like Git every user has a complete repository on their own computer. While there may be a centralized repository that people send their commits to it will not be accessed unless specifically requested. All commits, branches and history are stored offline unless users choose to send or receive commits from another repository.

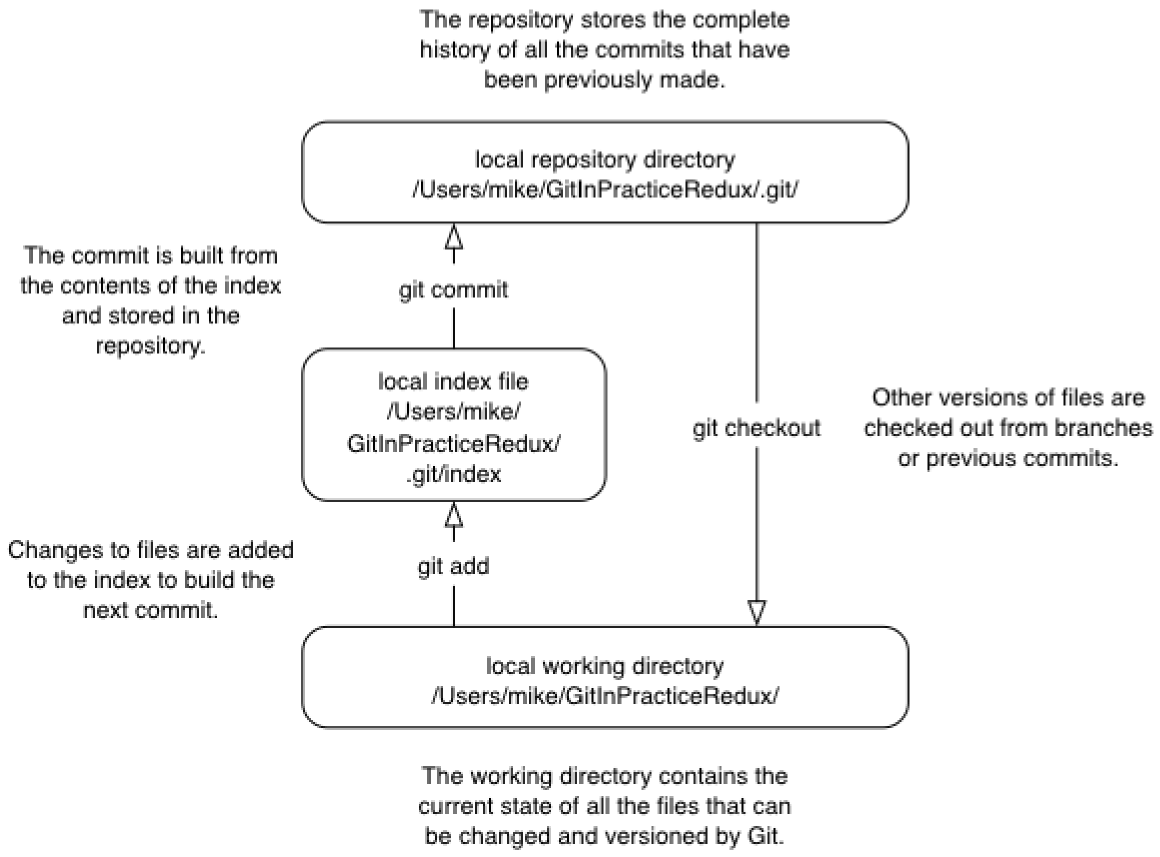

The repository stores the complete history of all the commits that have been previously made.
local repository directory /Users/mike/GitInPracticeRedux/.git/
The commit is built from the contents of the index and stored in the repository.
git commit
local index file /Users/mike/ GitInPracticeRedux/ .git/index
git checkout
Other versions of files are checked out from branches or previous commits.
Changes to files are added to the index to build the next commit.
git add
local working directory /Users/mike/GitInPracticeRedux/
The working directory contains the current state of all the files that can be changed and versioned by Git.

Figure 2.1 Git add/commit/checkout cycle

Figure 2.1 shows the local Git cycle we used in Chapter 1. Files in the local working directory are modified and added with `git add` to the index staging area. The contents of the index staging area is committed with `git commit` to form a new commit which is stored in the local repository directory. Later, this repository can be queried to view the differences between versions of files using `git diff`. In Section 2.7 you will also see how to use `git checkout` to change to different local branches' versions of files.

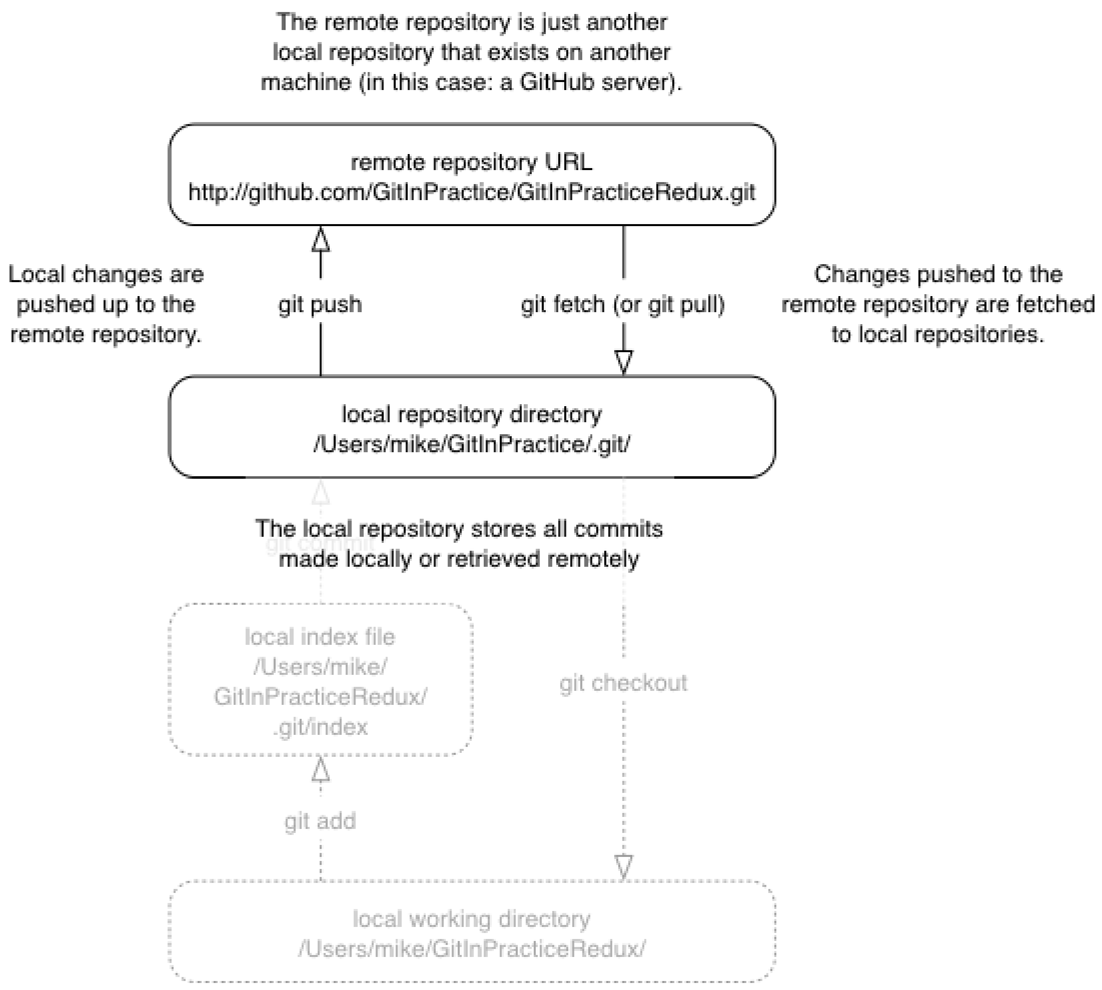

The remote repository is just another local repository that exists on another machine (in this case: a GitHub server).
remote repository URL
http://github.com/GitInPractice/GitInPracticeRedux.git
Local changes are pushed up to the remote repository.
git push
git fetch (or git pull)
Changes pushed to the remote repository are fetched to local repositories.
local repository directory
/Users/mike/GitInPractice/.git/
The local repository stores all commits made locally or retrieved remotely
git commit
local index file
/Users/mike/GitInPracticeRedux/.git/index
git checkout
git add
local working directory
/Users/mike/GitInPracticeRedux/

Figure 2.2 Git add/commit/push/pull/checkout cycle

Figure 2.2 shows the remote Git cycle we will look at in this chapter. As in the local workflow files are modified, added, committed and can be checked out. However there are now two repositories: a local repository and *remote repository*.

If your local repository needs to send or receive data to a repository on another machine it will need to add a *remote repository*. A remote repository is one that is typically stored on another computer. `git push` sends your new commits to it and `git fetch` retrieves any new commits made by others from it.

In Chapter 1 you created a local repository on your machine. Please sign up for a GitHub account and create a *remote repository* on GitHub (detailed in Appendix D). You can use another Git hosting provider but this book will assume the use of GitHub (as it is the most widely used).

The first action you’re concerned with is adding a reference for your newly-created remote repository (also known as a *remote*) on GitHub to your previous local repository so you can push and fetch commits.

2.1.2 Problem

You wish to add the new GitInPractice remote repository to your current repository.

2.1.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git remote add origin` with your repository URL appended. e.g. if your username is `GitInPractice` and repository is named `GitInPracticeRedux` run `git remote add origin https://github.com/GitInPractice/GitInPracticeRedux.git`. There will be no output.

You can verify this remote has been created successfully by running `git remote --verbose`. The output should resemble:

Listing 2.1 Remote repositories

```
# git remote --verbose

origin  https://github.com/GitInPractice/GitInPracticeRedux.git (fetch)
origin  https://github.com/GitInPractice/GitInPracticeRedux.git (push)
```

❶ **fetch URL**
❷ **push URL**

In the remote listing:

- "fetch URL (1)" specifies the URL that `git fetch` uses to fetch new remote commits.
- "push URL (2)" specifies the URL that `git push` uses to send new local commits.

NOTE **When the fetch and push URLs differ?**
These will not differ unless they have been set to do so by the `git remote` command or by Git configuration. It's almost never necessary to do this so I will not cover it in this book.

You have added a remote named `origin` that points to the remote `GitInPracticeRedux` repository belonging to the `GitInPractice` user on GitHub. You can now send and receive changes from this remote. Nothing has been sent or received yet; the new remote is effectively just a named URL pointing to the remote repository location.

2.1.4 Discussion

`git remote` can also be called with the `rename` and `remove` (or `rm`) subcommands to alter remotes accordingly.

`git remote show` will query and show verbose information about the given remote.

`git remote prune` will delete any remote references to branches that have been deleted from the remote repository by other users. Don't worry about this for now; remote branches will be covered in Section 2.8.

NOTE **What is the default name for a remote?**
You can have multiple remote repositories connected to your local repository so the remote repositories are named. Typically if you have a single remote repository it will be named `origin`.

AUTHORITATIVE VERSION STORAGE

With centralized version control systems the central server always stores the authoritative version of the code. Clients to this repository will typically only store a small proportion of the history and require access to the server to perform most tasks. With a distributed version control system like Git every local repository has a complete copy of the data. Which repository stores the authoritative version in this case? It turns out that this is merely a matter of convention; Git itself does not deem any particular repository to have any higher priority than another. Typically in organizations there will be a central location (like with a centralized version control) which is treated as the authoritative version and people are encouraging to push their commits and branches to.

The lack of authority for a particular repository with distributed version control systems is sometimes seen as a liability but can actually be a strength. The Linux kernel project (for which Git was original created) makes use of this to provide a network of trust and a more manageable way of merging changes. When Linus Torvalds, the self-named "benevolent dictator" of the project, tags a new release this is generally considered a new release of Linux. What is in his repository (well, his publicly accessible one; he will have multiple repositories between various personal machines that he does not make publicly accessible) is generally considered to be what is in Linux. Linus has trusted lieutenants from who he can pull and merge commits and branches. Rather than every single merge to Linux needing to be done by Linus he can leave some of it to his lieutenants (who leave some to their sub-lieutenants and so on) so everyone can needs only worry about verifying and including the work of a small number of others. This particular workflow may not make sense in many organizations but it demonstrates how distributed version control systems can allow different ways of managing merges to centralized version control.

2.2 Pushing changes to a remote repository: git push

2.2.1 Background

You will eventually wish to send commits made in the local repository to a remote. To do this always requires an explicit action. Only changes specifically requested will be sent and the Git (which can operate over HTTP, SSH or it's own protocol (`git://`)) will ensure that only the differences between the repositories are sent. As a result you can push small changes from a large local repository to a large remote repository very quickly as long as they have most commits in common.

Let's push the changes you made in our repository in Chapter 1 to the newly created remote you made in Section 2.1.3.

2.2.2 Problem

You wish to push the changes from the local `GitInPracticeRedux` repository to the `origin` remote on GitHub.

2.2.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git push --set-upstream origin master` and enter your GitHub username and password when requested. The output should resemble:

Listing 2.2 Push and set upstream branch

```
# git push --set-upstream origin master

Username for 'https://github.com': GitInPractice   ❶
Password for 'https://GitInPractice@github.com':   ❷
Counting objects: 6, done.   ❸
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 602 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux.git   ❹
 * [new branch]      master -> master   ❺
Branch master set up to track remote branch master from origin.   ❻
```

❶ username entry
❷ password entry
❸

object preparation/transmission
❹ remote URL
❺ local/remote branch
❻ set tracking branch

From the push output you can see:

- "username entry (1)" and "password entry (2)" are those for your GitHub account. They may only be asked for the first time you push to a repository depending on your operating system of choice (which may decide to save the password for you). They are always required to `push` to repositories but are only required for `fetch` when fetching from private repositories.
- "object preparation/transmission (3)" can be safely ignored in this or future figures; it is simply Git communicating details on how the files are being sent to the remote repository and isn't worth understanding beyond basic progress feedback.
- "remote URL (4)" matches the push URL from the `git remote --verbose` output earlier. It is where Git has sent the local commits to.
- "local/remote branch (5)" indicates that this was a new branch on the remote. This is because the remote repository on GitHub was empty until we pushed this; it had no commits and thus no `master` branch yet. This was created by the `git push`. The `master -> master` indicates the local master branch (the first of the two) has been pushed to the remote `master` branch (the second of the two). This may seem redundant but it is shown as it is possible (but ill-advised due to the obvious confusion it causes) to have local and remote branches with different names. Don't worry about local or remote branches for now as these will be covered in Section 2.6.
- "set tracking branch (6)" is shown because the `--set-upstream` option was passed to `git push`. By passing this option you have is told Git that you want the local `master` branch you have just pushed to *track* the `origin` remote's branch `master`. The `master` branch on the `origin` remote (which is often abbreviated as `origin/master`) is now known as the *tracking branch* (or *upstream*) for your local `master` branch.

You have pushed your `master` branch's changes to the `origin` remote's `master` branch.

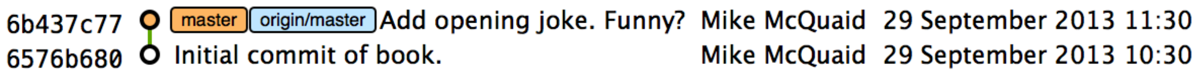
2.2.4 Discussion

The `git push --set-upstream` (or `-u`) flag and explicit specification of `origin` and `master` are only required the first time you push a branch. After that a `git push` with no arguments will default to running the equivalent of `git push origin master`.

`git push` can take an `--all` flag which will push all branches and tags at once. Be careful when doing this; you may push some branches with work in-progress.

`git push` can take a `--force` flag which will disable some checks on the remote repository to allow rewriting of history. *This is very dangerous. Do not use* *this flag until after reading (and rereading) Chapter 6.*

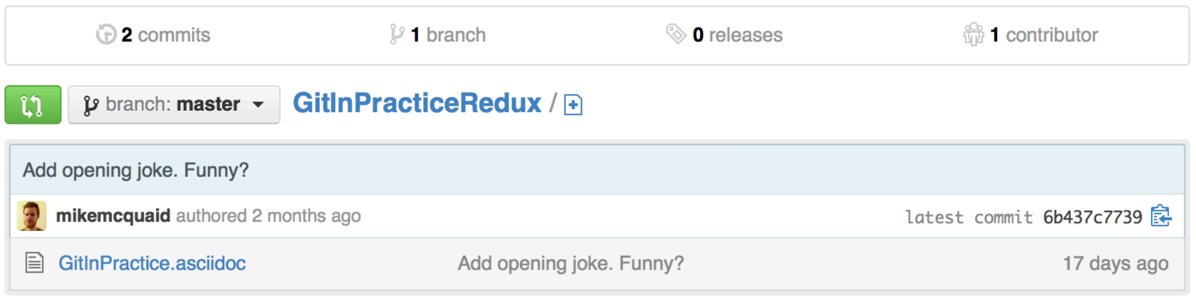
A *tracking branch* is the default push or fetch location for a branch. This means in future you could run `git push` with no arguments on this branch and it will do the same thing as running `git push origin master` i.e. push the current branch to the `origin` remote's `master` branch.



```
6b437c77  master origin/master Add opening joke. Funny?  Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                        Mike McQuaid  29 September 2013 10:30
```

Figure 2.3 Local repository after `git push`

Figure 2.3 shows the state of the repository after the `git push`. There is one addition since we last looked at it in Figure 2.10: the `origin/master` label. This is attached to the commit which matches the currently known state of the `origin` remote's `master` branch.



2 commits | 1 branch | 0 releases | 1 contributor

branch: master | GitInPracticeRedux /

Add opening joke. Funny?

mikemcquaid authored 2 months ago — latest commit 6b437c7739

GitInPractice.asciidoc	Add opening joke. Funny?	17 days ago

Figure 2.4 GitHub repository after `git push`

Figure 2.4 shows the remote repository on GitHub after the `git push`. The latest commit SHA-1 there matches your current latest commit on the `master` branch seen in Figure 2.3 (although they are different lengths; remember SHA-1s can always be shortened as long as they remain unique). To update this in future you would run `git push` again to push any local changes to GitHub.

2.3 Cloning a remote/GitHub repository onto your local machine: git clone

2.3.1 Background

It is useful to learn how to create a new Git repository locally and push it to GitHub. However, you will usually be downloading an existing repository to use as your local repository. This process of creating a new local repository from an existing remote repository is known as *cloning* a repository.

Some other version control systems (such as Subversion) will use the terminology of *checking out* a repository. The reasoning for this is that Subversion is a centralized version control system so when you download a repository locally you are only actually downloading the latest revision from the repository. With Git it is known as *cloning* because you are making a complete copy of that repository by downloading all commits, branches, tags; the complete history of the repository onto your local machine.

As you just pushed the entire contents of the local repository to GitHub let's remove the local repository and recreate it by cloning the repository on GitHub.

2.3.2 Problem

You wish to remove the existing `GitInPracticeRedux` local repository and recreate it by cloning from GitHub:

1. Change to the directory where you want the new `GitInPracticeRedux` repository to be created e.g. `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeRedux`.
2. Run `rm -rf GitInPracticeRedux` to remove the existing `GitInPracticeRedux` repository.
3. Run `git clone https://github.com/GitInPractice/GitInPracticeRedux.git`. The output should resemble:

Listing 2.3 Cloning a remote repository

```
# git clone https://github.com/GitInPractice/GitInPracticeRedux.git

Cloning into 'GitInPracticeRedux'...   ❶
remote: Counting objects: 6, done.   ❷
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 6 (delta 0)
Unpacking objects: 100% (6/6), done.
Checking connectivity... done
```

❶ destination directory
❷ object preparation/transmission

From the clone output you can see:

- "destination directory (1)" is the directory in which the new `GitInPracticeRedux` local repository was created.
- "object preparation/transmission (2)" can be safely ignored again (although if you're wondering why there were 6 objects remember the different objects in the object store in Chapter 1).

You have cloned the `GitInPracticeRedux` remote repository and created a new local repository containing all its commits in `/Users/mike/GitInPracticeRedux`.

You can verify this remote has been created successfully by running `git remote --verbose`. The output should resemble:

Listing 2.4 Remote repositories

```
# git remote --verbose

origin  https://github.com/GitInPractice/GitInPracticeRedux.git (fetch)
origin  https://github.com/GitInPractice/GitInPracticeRedux.git (push)
```

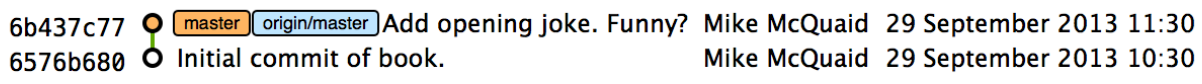
❶ fetch URL
❷ push URL

2.3.3 Discussion

`git clone` can take `--bare` or `--mirror` flags which will create a repository suitable for hosting on a server. This will be covered more in Chapter 13.

`git clone` can take a `--depth` flag followed by an integer which will create a *shallow clone*. A shallow clone is one where only the specified number of revisions are downloaded from the remote repository but it is limited as it cannot be cloned/fetched/pushed from or pushed to.

`git clone` can take a `--recurse-submodules` (or `--recursive`) flag which will initialize all the Git submodules in the repository. Submodules will be covered in Chapter 12.



```
6b437c77  master origin/master Add opening joke. Funny?  Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                        Mike McQuaid  29 September 2013 10:30
```

Figure 2.5 Local repository after `git clone`

Figure 2.5 shows the state of the repository after the `git clone`. It is identical to the state after the `git push` in Figure 2.3. This shows that the clone was successful and the newly created local repository has the same contents as the deleted old local repository.

Cloning a repository has also created a new remote called `origin`. `origin` is the default remote and references the repository that the clone originated from (which is https://github.com/GitInPractice/GitInPracticeRedux.git in this case).

Now let's learn how to pull new commits from the remote repository.

2.4 Pulling changes from another repository: git pull

2.4.1 Background

`git pull` downloads the new commits from another repository and merges the remote branch into the current branch.

If you run `git pull` on the local repository you just see a message stating `Already up-to-date.`. `git pull` in this case contacted the remote repository, saw that there were no changes to be downloaded and let us know that it was up to date. This is expected as this repository has been pushed to but not updated since.

To test `git pull` let's create another clone of the same repository, make a new commit and `git push` it. This will allow downloading new changes with `git pull` on the original remote repository.

To create another cloned, local repository and push a commit from it:

1. Change to the directory where you want the new `GitInPracticeRedux` repository to be created e.g. `cd /Users/mike/` to create the new local repository in `/Users/mike/GitInPracticeReduxPushTest`.
2. Run `git clone https://github.com/GitInPractice/GitInPracticeRedux.git GitInPracticeReduxPushTest` to clone into the `GitInPracticeReduxPushTest` directory.
3. Change directory to the new Git repository e.g. `cd /Users/mike/GitInPracticeReduxPushTest/`.
4. Modify the `GitInPractice.asciidoc` file.
5. Run `git add GitInPractice.asciidoc`.
6. Run `git commit --message 'Improve joke comic timing.'`.
7. Run `git push`.

Now that you've pushed a commit to the `GitInPracticeRedux` remote on GitHub you can change back to your original repository and `git pull` from it. Keep the `GitInPracticeReduxPushTest` directory around as we'll use it later.

2.4.2 Problem

You wish to pull new commits into the current branch on the local `GitInPracticeRedux` repository from the remote repository on GitHub.

2.4.3 Solution

1. Change directory to the original Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git pull`. The output should resemble:

Listing 2.5 Pulling new changes

```
# git pull

remote: Counting objects: 5, done.   ❶
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/GitInPractice/GitInPracticeRedux   ❷
   6b437c7..85a5db1  master     -> origin/master   ❸
Updating 6b437c7..85a5db1   ❹
Fast-forward   ❺
 GitInPractice.asciidoc | 5 +++--   ❻
 1 file changed, 3 insertions(+), 2 deletions(-)   ❼
```

❶ object preparation/transmission
❷ remote URL
❸ remote branch update
❹ local branch update
❺ merge type
❻ lines changed in file
❼ diff summary

You can see from the pull output:

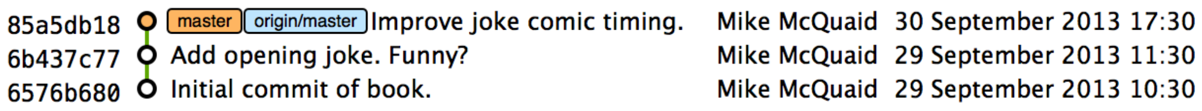
- "object preparation/transmission (1)" can be safely ignored again.
- "remote URL (2)" matches the remote repository URL we saw used for `git push`.
- "remote branch update (3)" shows how the state of the `origin` remote’s `master` branch was updated and that this can be seen in `origin/master`. `origin/master` is a valid ref that can be used with tools such as `git diff` so `git diff origin/master` will show the differences between the current working tree state and the `origin` remote’s `master` branch.
- "local branch update (4)" shows that after `git pull` downloaded the changes from the other repository it merged the changes from the tracking branch into the current branch. In this case your `master` branch had the changes from the `master` branch on the remote `origin` merged in. You can see in this case the SHA-1s match those in the "remote branch update (3)". It has been updated to include the new commit (`85a5db1`).
- "merge type (5)" was a *fast-forward merge* which means that no merge commit was made. Fast-forward merges will be explained in Section 2.9.3.

- "lines changed in file <6>" is the same as the lines changed from `git commit` or `git diff` in Chapter 1. It is showing a summary of the changes that have been pulled into your `master` branch.
- "diff summary <7>" is the same as the diff summary from `git commit` or `git diff` in Chapter 1.

2.4.4 Discussion

`git pull` can take a `--rebase` flag which will perform a rebase rather than a merge. This will be covered in Chapter 6.

NOTE **Why did a merge happen?**
It may be confusing that a merge has happened here. Didn't you just ask for the updates from that branch? You haven't created any other branches so why did a merge happen? In Git all remote branches (which includes the default `master` branch) are only linked to your local branches if the local branch is tracking the remote branch. As a result when you are pulling in changes from a remote branch into your current branch you may sometimes result in a situation where you have made local changes and the remote branch has also received changes. In this case a merge must be made to reconcile the differing local and remote branch.



```
85a5db18  master origin/master Improve joke comic timing.   Mike McQuaid  30 September 2013 17:30
6b437c77  Add opening joke. Funny?                          Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                           Mike McQuaid  29 September 2013 10:30
```

Figure 2.6 Local repository after `git pull`

You can see from Figure 2.6 that a new commit has been added to the repository and that both `master` and `origin/master` have been updated.

You have pulled the new commits from the `GitInPracticeRedux` remote repository into your local repository and Git has merged them into your `master` branch. Now let's learn how to download changes without applying them onto your master branch.

2.5 Fetching changes from a remote without modifying local branches: git fetch

2.5.1 Background

Remember that `git pull` performs two actions: fetching the changes from a remote repository and merging them into the current branch. Sometimes you may wish to download the new commits from the remote repository without merging them into your current branch (or without merging them yet). To do this you can use the `git fetch` command. `git fetch` performs the fetching action of downloading the new commits but skips the merge step (which you can manually perform later).

To test `git fetch` let's use the `GitInPracticeReduxPushTest` local repository again to make another new commit and `git push` it. This will allow downloading new changes with `git fetch` on the original remote repository.

To push another commit from the `GitInPracticeReduxPushTest` repository:

1. Change directory to the `GitInPracticeReduxPushTest repository e.g. `cd /Users/mike/GitInPracticeReduxPushTest/`.
2. Modify the `GitInPractice.asciidoc` file.
3. Run `git add GitInPractice.asciidoc`.
4. Run `git commit --message 'Joke rejected by editor!'`.
5. Run `git push`.

Now that you've pushed another commit to the `GitInPracticeRedux` remote on GitHub you can change back to your original repository and `git fetch` from it. If you wish you can now delete the `GitInPracticeReduxPushTest` repository by running e.g. `rm -rf /Users/mike/GitInPracticeReduxPushTest/`

2.5.2 Problem

You wish to fetch new commits to the local `GitInPracticeRedux` repository from the `GitInPracticeRedux` remote repository on GitHub without merging into your `master` branch.

2.5.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git fetch`. The output should resemble:

Listing 2.6 Fetching new changes

```
# git fetch

remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/GitInPractice/GitInPracticeRedux
   85a5db1..07fc4c3  master     -> origin/master
```

❶ object preparation/transmission

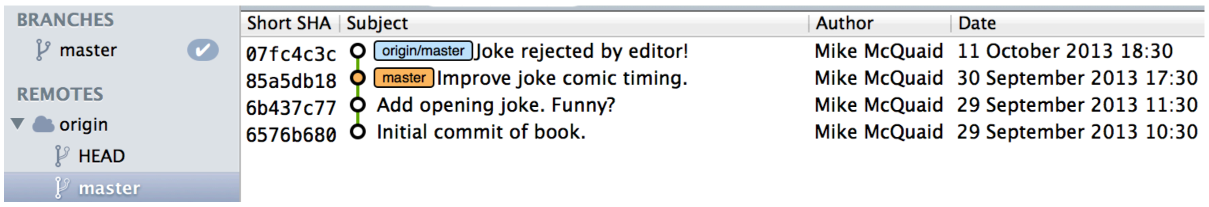
❷ remote URL

❸ remote branch update

The `git fetch` output is the same as the first part of the `git pull` output. However the SHA-1s are different again as a new commit was downloaded. This is because `git fetch` is effectively half of what `git pull` is doing. If your `master` branch is tracking the `master` branch on the remote `origin` then `git pull` is directly equivalent to running `git fetch && git merge origin/master`.

You've fetched the new commits from the remote repository into your local repository without not merging them into your `master` branch.

2.5.4 Discussion



Short SHA	Subject	Author	Date
07fc4c3c	origin/master Joke rejected by editor!	Mike McQuaid	11 October 2013 18:30
85a5db18	master Improve joke comic timing.	Mike McQuaid	30 September 2013 17:30
6b437c77	Add opening joke. Funny?	Mike McQuaid	29 September 2013 11:30
6576b680	Initial commit of book.	Mike McQuaid	29 September 2013 10:30

Figure 2.7 Remote repository after `git fetch`

You can see from Figure 2.7 that another new commit has been added to the repository but this time only `origin/master` has been updated but `master` has not. To see this you may need to select the `origin` remote and `master` remote branch in the GitX sidebar. Selecting commits by remote branches is a feature sadly not available in `gitk`

To clean up our local repository let's do another quick `git pull` to update the state of the `master` branch based on the (already fetched) `origin/master`.

To pull new commits into the current branch on the local `GitInPracticeRedux` repository from the remote repository on GitHub:

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.

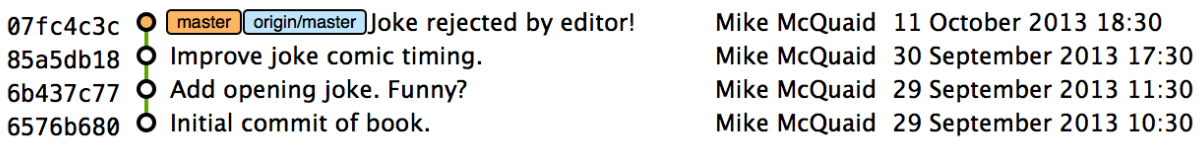
2. Run `git pull`. The output should resemble:

Listing 2.7 Pull after fetch

```
# git pull

Updating 85a5db1..07fc4c3 ❶
Fast-forward ❷
 GitInPractice.asciidoc | 4 +--- ❸
 1 file changed, 1 insertion(+), 3 deletions(-) ❹
```

❶ local branch update
❷ merge type
❸ lines changed in file
❹ diff summary

This shows the latter part of the first `git pull` output we saw. There were no more changes fetched from the `origin` remote and the local `master` branch had not been updated. As a result this `git pull` behaved the same as running `git merge origin/master`.



```
07fc4c3c  master origin/master Joke rejected by editor!   Mike McQuaid  11 October 2013 18:30
85a5db18  Improve joke comic timing.                      Mike McQuaid  30 September 2013 17:30
6b437c77  Add opening joke. Funny?                        Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                         Mike McQuaid  29 September 2013 10:30
```

Figure 2.8 Local repository after `git fetch` then `git pull`

Figure 2.8 shows that the `master` branch has now been updated to match the `origin/master` latest commit once more.

NOTE **Should I use pull or fetch?**
I prefer to use `git fetch` over `git pull`. It means I can continue to fetch regularly in the background and only include these changes in my local branches when it is convenient and in the method I find most appropriate which may be merging or rebasing (or resetting which you will see in Chapter 3). Additionally, I sometimes work in situations where I have no internet connection (such as on planes) and using `git fetch` is superior in these situations; it can fetch changes without requiring any human interaction in the case of e.g. a merge conflict.

We've talked about local branches and remote branches but haven't actually created any ourselves yet. Let's learn about how branches work and how to create them.

2.6 Creating a new local branch from the current branch: git branch

2.6.1 Background

When committing in Git the history continues linearly; what was the most recent commit becomes the parent commit for the new commit. This parenting continues back to the initial commit in the repository. You saw an example of this in Figure 2.9.

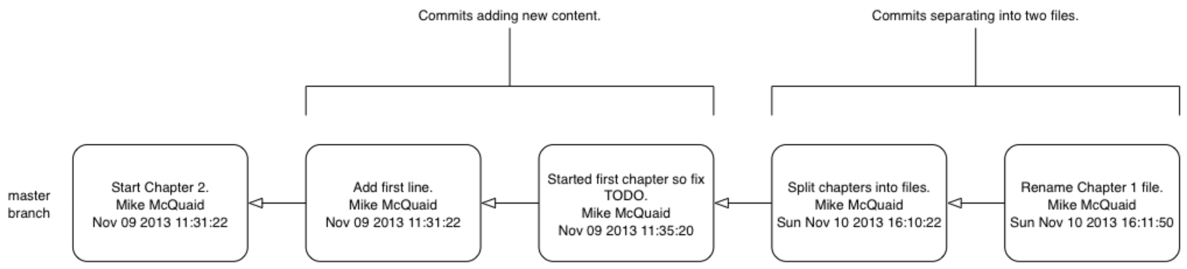

Commits adding new content.
Commits separating into two files.
master branch
Start Chapter 2. Mike McQuaid Nov 09 2013 11:31:22
Add first line. Mike McQuaid Nov 09 2013 11:31:22
Started first chapter so fix TODO. Mike McQuaid Nov 09 2013 11:35:20
Split chapters into files. Mike McQuaid Sun Nov 10 2013 16:10:22
Rename Chapter 1 file. Mike McQuaid Sun Nov 10 2013 16:11:50

Figure 2.9 Committing without using branches

Sometimes this linear approach is not enough for software projects. Sometimes you may need to make new commits which are not yet ready for public consumption. This requires *branches*.

Branching allows two independent tracks through history to be created and committed to without either modifying the other. Programmers can happily commit to their independent branch without the fear of disrupting the work of another branch. This means that they can, for example, commit broken or incomplete features rather than having to wait for others to be ready for their commits. It also means they can be isolated from changes made by others until they are ready to integrate them into their branch. Figure 2.10 shows the same commits as Figure 2.9 if they were split between two branches instead for isolation.

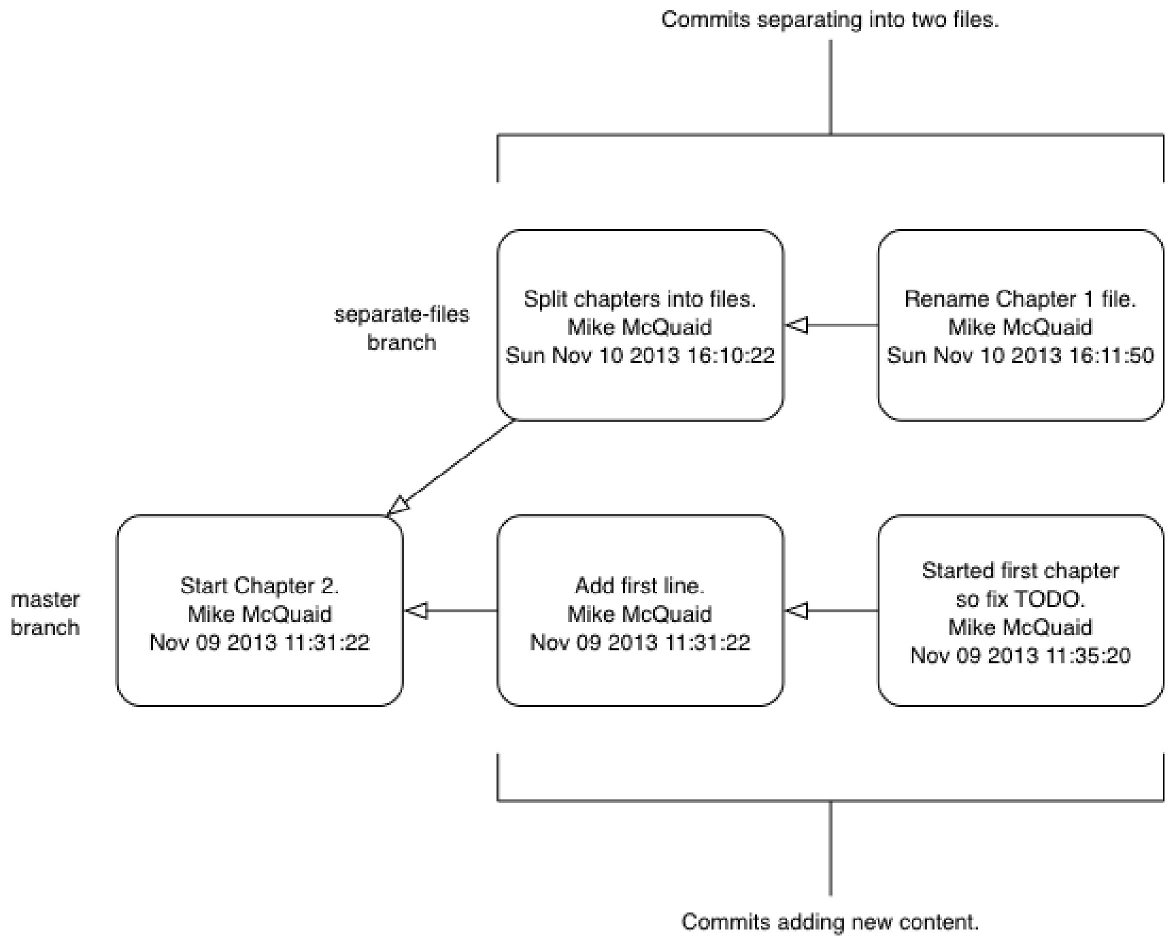

Commits separating into two files.
separate-files branch
Split chapters into files. Mike McQuaid Sun Nov 10 2013 16:10:22
Rename Chapter 1 file. Mike McQuaid Sun Nov 10 2013 16:11:50
master branch
Start Chapter 2. Mike McQuaid Nov 09 2013 11:31:22
Add first line. Mike McQuaid Nov 09 2013 11:31:22
Started first chapter so fix TODO. Mike McQuaid Nov 09 2013 11:35:20
Commits adding new content.

Figure 2.10 Committing to multiple branches

When a branch is created and new commits are made that branch advances forward to include the new commits. In Git a branch is actually no more than a pointer to a particular commit. This is unlike other version control systems such as Subversion in which branches are just a subdirectory of the repository.

The branch is pointed to a new commit when a new commit is made on that branch. A *tag* is quite similar to a branch but points to a single commit and remains pointing to the same commit even when new commits are made. Typically tags are used for annotating commits; for example, when you release version 1.0 of your software you may tag the commit used to built the 1.0 release with a "1.0" tag. This means you can come back to it in future, rebuild that release or check how certain things worked without fear that it will be somehow changed automatically.

Branching allows two independent tracks of development to occur at once. In Figure 2.10, the `separate-files branch` was used to separate the content from a single file and split it into two new files. This allowed refactoring of the book structure to be done in the `separate-files` branch while the default branch (known as `master` in Git) could be used to create more content. In version control systems like Git where creating a branch is a quick, local operation branches may be used for every independent change.

Some programmers will create new branches whenever they work on a new bug fix or feature and then integrate these branches at a later point; perhaps after requesting review of their changes from others. This means even for programmers working without a team it can be useful to have multiple branches in use at any one point. For example, you may be working on a new feature but realize that a critical error in your application needs fixed immediately. You could quickly create a new branch based off the version used by customers, fix the error and switch branch back to the branch you had been committing the new feature to.

2.6.2 Problem

You wish to create a new local branch named `chapter-two` from the current (`master`) branch.

2.6.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git branch chapter-two`. There will be no output.

You can verify the branch was created by running `git branch` which should have the following output:

Listing 2.8 List branches

```
# git branch

  chapter-two                                             ❶ new branch
* master                                                  ❷ current branch
```

From the branch output:

- "new branch (1)" was created with the expected name.
- "current branch <2>" is indicated by the `*` prefix which shows you are still on the master branch as before. `git branch` creates a new branch but does not change to it.

You have created a new local branch named `chapter-two` which currently points to the same commit as `master`.

2.6.4 Discussion

`git branch` can take a second argument with the *start point* for the branch. This defaults to the current branch you are on e.g. `git branch chapter-two` is the equivalent of `git branch chapter-two master` if you're already on the master branch. This can be used to create branches from previous commits which is sometimes useful if e.g. the current `master` branch state has broken unit tests that you need to be working.

`git branch` can take a `--set-upstream` flag which, combined with a start point, will set the upstream for the branch (similarly to `git push --set-upstream` but without pushing anything remotely yet).

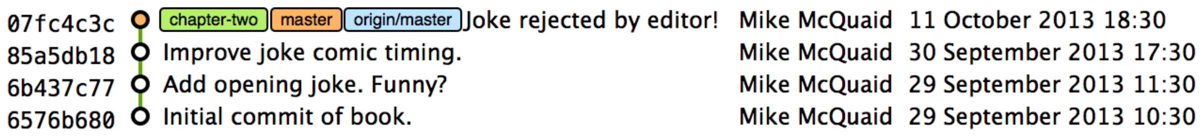

```
07fc4c3c  chapter-two master origin/master Joke rejected by editor!  Mike McQuaid  11 October 2013 18:30
85a5db18  Improve joke comic timing.                                 Mike McQuaid  30 September 2013 17:30
6b437c77  Add opening joke. Funny?                                   Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                                    Mike McQuaid  29 September 2013 10:30
```

Figure 2.11 Local repository after `git branch chapter-two`

You can see from Figure 2.11 that there is a new branch label for the `chapter-two` branch. In the GitX GUI the label colors indicate:

- orange: the currently checked-out local branch
- green: a non-checked-out local branch
- blue: a remote branch

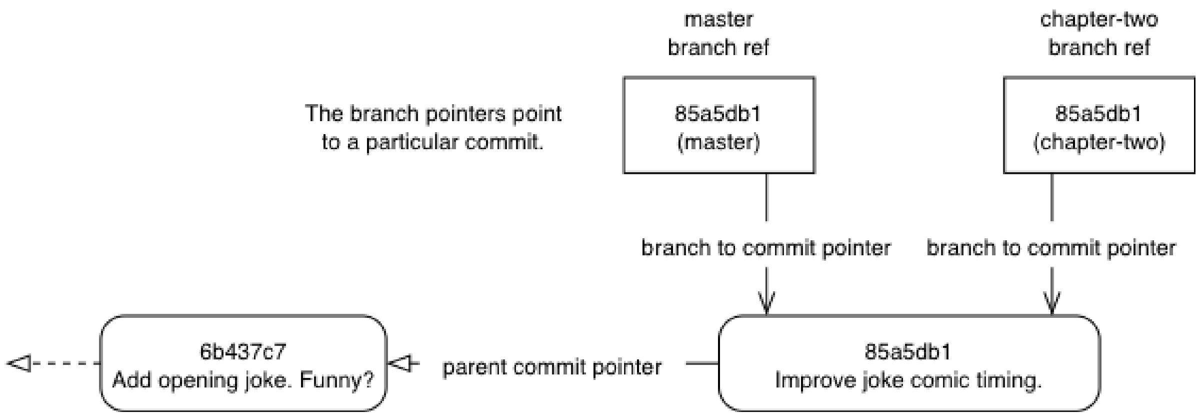


Figure 2.12 Branch pointers

Figure 2.12 shows how these two branch pointers point to the same commit.

You've seen `git branch` creates a local branch it does not change to it. To do that requires using `git checkout`.

NOTE **Can branches be named anything?**
Branches cannot have two consecutive dots (`..`) anywhere in their name so `chapter..two` would be an invalid branch name and `git branch` will refuse to create it. This particular case is due to the special meaning of `..` for a commit range for the `git diff` command (which we saw used in Chapter 1).

NOTE **What names should I use for branches?**
Name branches according to their contents. For example, the `chapter-two` branch we've created here describes that the commits in this branch will be referencing the second chapter. I recommend a format of describing the branch's purpose in multiple words separated by hyphens. For example, a branch that is performing cleanup on the test suite should be named `test-suite-cleanup`.

2.7 Checking out a local branch: git checkout

2.7.1 Background

Once you've created a local branch you will want to check out the contents of another branch into Git's working directory. The state of all the current files in the working directory will be replaced with the new state based on the revision that the new branch is currently pointing to.

2.7.2 Problem

You wish to change to a local branch named `chapter-two` from the current (`master`) branch.

2.7.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout chapter-two`. The output should be `Switched to branch 'chapter-two'`.

You've checked out the local branch named `chapter-two` and moved from the `master` branch.

2.7.4 Discussion

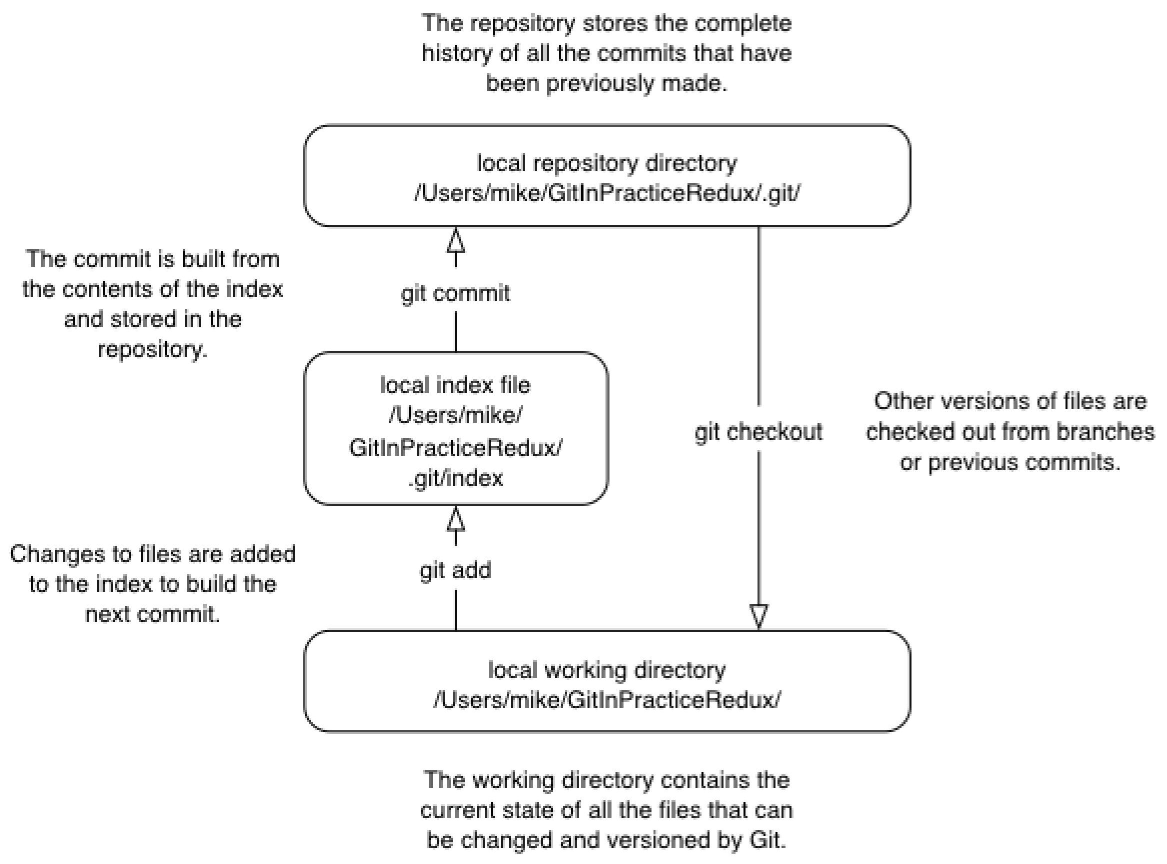

The repository stores the complete history of all the commits that have been previously made.
local repository directory /Users/mike/GitInPracticeRedux/.git/
The commit is built from the contents of the index and stored in the repository.
git commit
local index file /Users/mike/ GitInPracticeRedux/ .git/index
git checkout
Other versions of files are checked out from branches or previous commits.
Changes to files are added to the index to build the next commit.
git add
local working directory /Users/mike/GitInPracticeRedux/
The working directory contains the current state of all the files that can be changed and versioned by Git.

Figure 2.13 Git add/commit/checkout workflow

NOTE **Why do Subversion and Git use `checkout` to mean different things?**

As mentioned earlier some other version control systems (e.g. Subversion) use `checkout` to refer to the initial download from a remote repository but `git checkout` is used here to change branches. This may be slightly confusing until we look at Git's full remote workflow. Figure 2.13 shows Git's local workflow again. Under closer examination `git checkout` and `svn checkout` behave similarly; both check out the contents of a version control repository into the working directory but Subversion's repository is remote and Git's repository is local. In this case `git checkout` is requesting the checkout of a particular branch so the current state of that branch is checked out into the working directory.

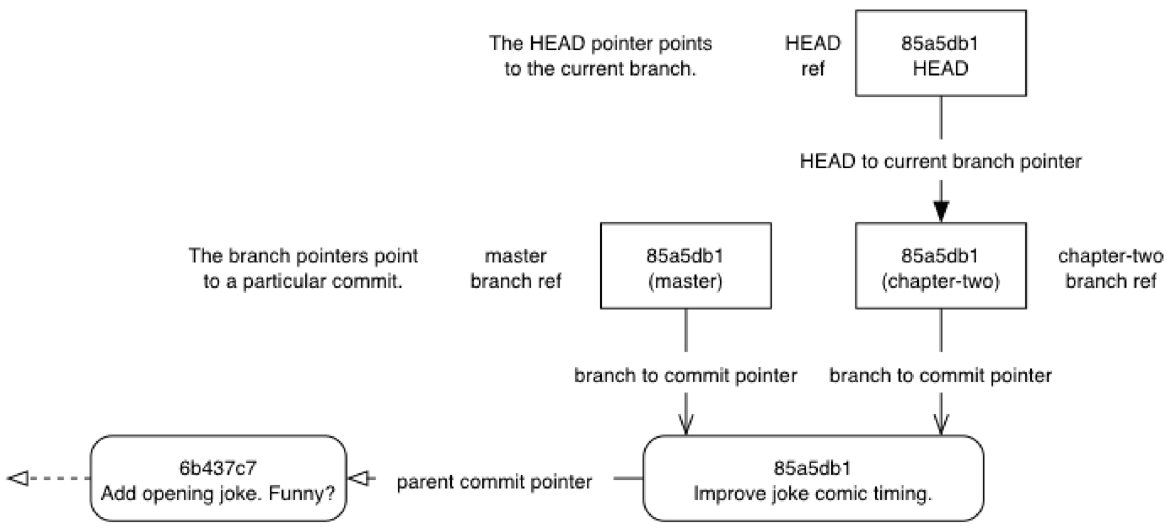

The HEAD pointer points to the current branch.
HEAD ref
85a5db1 HEAD
HEAD to current branch pointer
The branch pointers point to a particular commit.
master branch ref
85a5db1 (master)
85a5db1 (chapter-two)
chapter-two branch ref
branch to commit pointer
branch to commit pointer
6b437c7 Add opening joke. Funny?
parent commit pointer
85a5db1 Improve joke comic timing.

Figure 2.14 HEAD pointer with multiple branches

Afterwards the HEAD pointer (seen in Figure 2.14) is updated to point to the current, `chapter-two` branch pointer which in turn points to the top commit of that branch. The HEAD pointer moved from the `master` to the `chapter-two` branch when you ran `git checkout chapter-two`; setting `chapter-two` to be the current branch.

NOTE **Will `git checkout` overwrite any uncommitted changes?**
Make sure you've committed any changes on the current branch before checking out a new branch. If you do not do this `git checkout` will refuse to check out the new branch if there are changes in that branch to a file with uncommitted changes. If you wish to overwrite these uncommitted changes anyway you can force this with `git checkout --force`.

2.8 Pushing a local branch remotely

2.8.1 Background

Now that you've created a new branch and checked it out it would be useful to push any new commits made to the remote repository. To do this requires using `git push` again.

2.8.2 Problem

You wish to push the changes from the local `chapter-two` branch to create the remote branch `chapter-two` on GitHub.

2.8.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout chapter-two` to ensure you are on the `chapter-two` branch.
3. Run `git push --set-upstream origin chapter-two`. The output should resemble:

Listing 2.9 Push and set upstream branch

```
git push --set-upstream origin chapter-two

Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/GitInPractice/GitInPracticeRedux.git
 * [new branch]      chapter-two -> chapter-two
Branch chapter-two set up to track remote branch
chapter-two from origin.
```

1 object preparation/transmission
2 local/remote branch
3 set tracking branch

The push output is much the same as the previous `git push` run:

- "object preparation/transmission (1)" (although still ignorable) shows that no new objects were sent. The reason for this is that the `chapter-two` branch still points to the same commit as the `master` branch; it's effectively a different name (or, more accurately, ref) pointing to the same commit. As a result there have been no more commit objects created and therefore no more were sent.
- "local/remote branch (2)" has `chapter-two` as the branch name.
- "set tracking branch (3)" has `chapter-two` as the branch name.

You have pushed your local `chapter-two` branch and created a new remote branch named `chapter-two` on the remote repository.

2.8.4 Discussion

Remember that now the local `chapter-two` branch is tracking the remote `chapter-two` branch so any future `git pull` or `git push` on the `chapter-two` branch will use the `origin` remote's `chapter-two` branch.

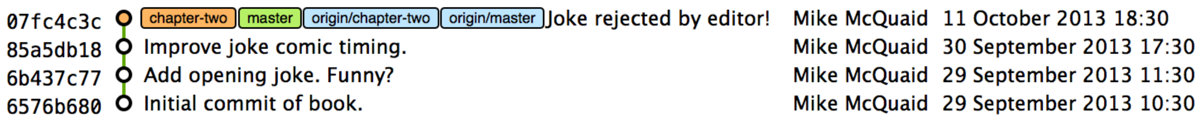

```
07fc4c3c  chapter-two master origin/chapter-two origin/master Joke rejected by editor!  Mike McQuaid  11 October 2013 18:30
85a5db18  Improve joke comic timing.                                                    Mike McQuaid  30 September 2013 17:30
6b437c77  Add opening joke. Funny?                                                      Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                                                       Mike McQuaid  29 September 2013 10:30
```

Figure 2.15 Local repository after `git push --set-upstream origin chapter-two`

As you'll hopefully have anticipated Figure 2.15 shows the addition of another remote branch named `origin/chapter-two`.

2.9 Merging an existing branch into the current branch: git merge

2.9.1 Background

At some point we have a branch that we’re done with and we want to bring all the commits made on it into another branch. This process is known as a `merge`.

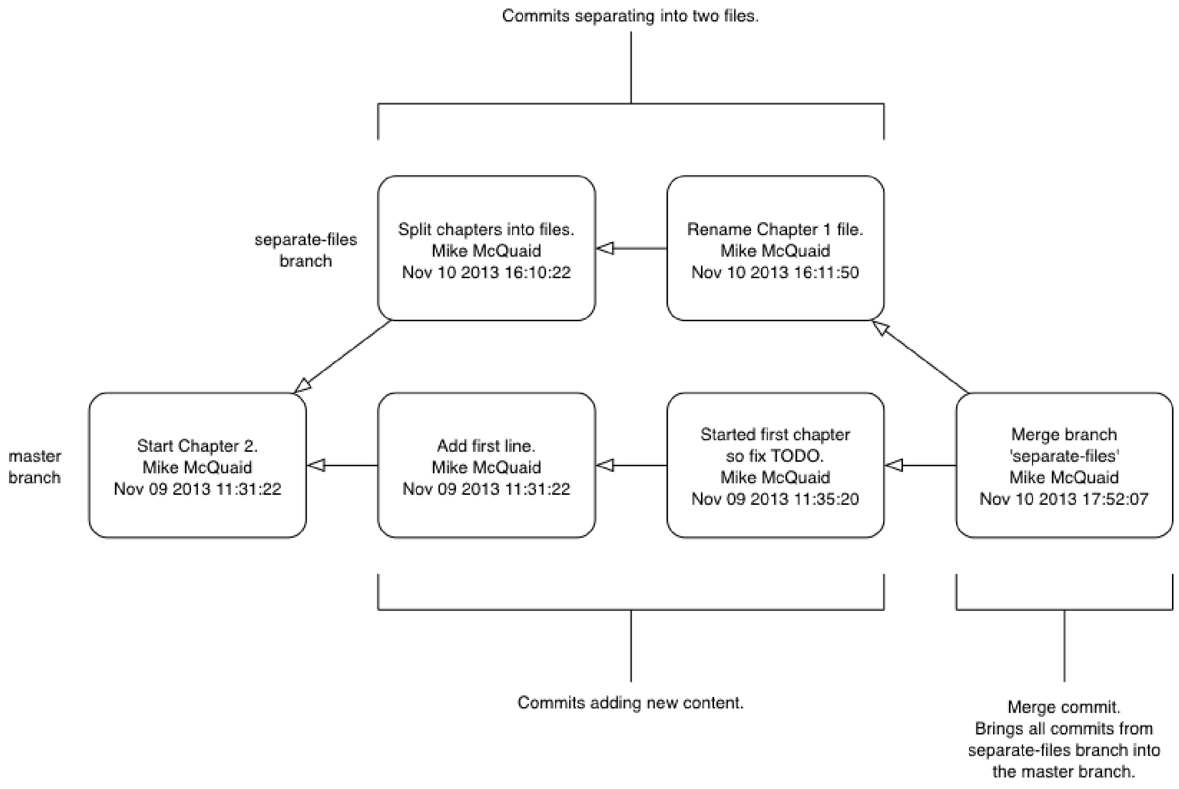

Commits separating into two files.
separate-files branch
Split chapters into files. Mike McQuaid Nov 10 2013 16:10:22
Rename Chapter 1 file. Mike McQuaid Nov 10 2013 16:11:50
master branch
Start Chapter 2. Mike McQuaid Nov 09 2013 11:31:22
Add first line. Mike McQuaid Nov 09 2013 11:31:22
Started first chapter so fix TODO. Mike McQuaid Nov 09 2013 11:35:20
Merge branch 'separate-files' Mike McQuaid Nov 10 2013 17:52:07
Commits adding new content.
Merge commit. Brings all commits from separate-files branch into the master branch.

Figure 2.16 Merging a branch into master

When a merge is requested all the commits from another branch are pulled into the current branch. Those commits then become part of the history of the branch. Please note from Figure 2.16 the commit in which the merge is made has two parents commits rather than one; it is joining together two separate paths through the history back into a single one. After a merge you may decide to keep the existing branch around to add more commits to it and perhaps merge again at a later point (only the new commits will need to be merged next time). Alternatively, you may delete the branch and make future commits on the Git’s default `master` branch and create another branch when needed in the future.

2.9.2 Problem

You wish to make a commit on the local branch named `chapter-two` and merge this into into the `master` branch.

2.9.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout chapter-two` to ensure you are on the `chapter-two` branch.
3. Modify the contents of `GitInPractice.asciidoc` and run `git add GitInPractice.asciidoc`.
4. Run `git commit --message 'Start Chapter 2.'`.
5. Run `git checkout master`.
6. Run `git merge chapter-two`. The output should resemble:

Listing 2.10 Merge branch

```
# git merge chapter-two

Updating 07fc4c3..ac14a50   ❶
Fast-forward   ❷
 GitInPractice.asciidoc | 2 ++
 1 file changed, 2 insertions(+)   ❸
```

❶ local branch update
❷ merge type
❸ diff summary

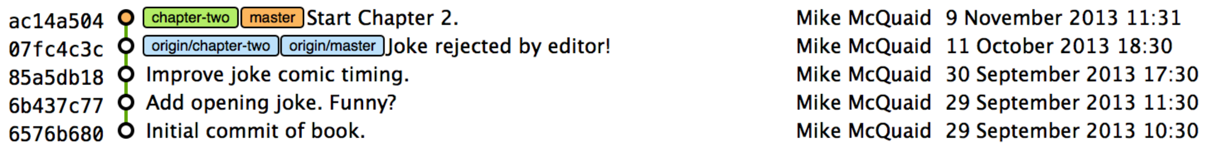
The output may seem familiar from the `git pull` output. Remember this is because `git pull` actually does a `git fetch && git merge`.

- "local branch update (1)" shows the changes that have been merged into the local `master` branch. Note that the SHA-1 has been updated from the previous `master` SHA-1 (`07fc4c3`) to the current `chapter-two` SHA-1 (`ac14a50`).
- "merge type (2)" was a *fast-forward merge*. This means that no merge commit (a commit with multiple parents) was needed so none was made. The `chapter-two` commits were made on top of the `master` branch but no more commits had been added to the `master` branch before the merge was made. In Git's typical language: the merged commit (tip of the `chapter-two` branch) is a descendent of the current commit (tip of the `master` branch). If there had been another commit on the `master` branch before merging then this merge would have created a merge commit. If there had been conflicts between the changes made in both branches that could not automatically be resolved then a merge conflict would be created and need to be resolved.
- "diff summary <3>" shows a summary of the changes that have been merged into your `master` branch from the `chapter-two` branch.

You have merged the `chapter-two` branch into the `master` branch.

2.9.4 Discussion

This brings the commit that was made in the `chapter-two` branch into the `master` branch.



```
ac14a504  chapter-two master Start Chapter 2.                          Mike McQuaid  9 November 2013 11:31
07fc4c3c  origin/chapter-two origin/master Joke rejected by editor!   Mike McQuaid  11 October 2013 18:30
85a5db18  Improve joke comic timing.                                   Mike McQuaid  30 September 2013 17:30
6b437c77  Add opening joke. Funny?                                     Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                                      Mike McQuaid  29 September 2013 10:30
```

Figure 2.17 Local repository after `git merge chapter-two`

You can see from Figure 2.17 that now the `chapter-two` and `master` branches point to the same commit once more.

MERGE CONFLICTS

So far merges may have sounded too good to be true; you can work on multiple things in parallel and combine them at any later point in any order. Not so fast my merge-happy friend; I haven't told you about merge conflicts yet.

A *merge conflict* occurs when both branches involved in the merge have changed the same part of the same file. Git will try and automatically resolve these conflicts but sometimes is unable to do so without human intervention. This case produces a merge conflict.

Listing 2.11 Merge conflict resolution with Git

```
= Git In Practice    ❶
<<<<<<< HEAD    ❷
== Chapter 1    ❸
It is a truth universally acknowledged, that a single person in
possession of good source code, must be in want of a version control
system.

== Chapter 2
// TODO: write second chapter.
=======    ❹
>>>>>>> separate-files    ❺
```

❶ unchanged line
❷ previous changes marker
❸ previous line version
❹ changes separator
❺ new changes marker

When a merge conflict occurs the version control system will go through any files that have conflicts and insert something similar to the above markers. These markers indicate the versions of the file on each branch.

- "unchanged line (1)" is provided only for context in this example
- "previous changes marker (2)" starts the section containing the lines from the previous commit (referenced by `HEAD` here).
- "previous line version (3)" shows a line from the previous commit.
- "changes separator (4)" starts the section containing the lines from the new branch.
- "new changes (5)" marker ends the section containing the lines from the new branch (referenced by `separate-files`; the name of the branch being merged in).

NOTE **How can conflict markers be found quickly?**
When searching a large file for the merge conflict markers you should enter <<<< into your text editor's find tool to quickly locate them.

The person performing the merge will need to manually edit the file to produce the correctly merged output, save it and mark the commit as resolved. Sometimes resolving the conflict will involve picking all the lines of a single version; either the previous version's lines or the new branch's lines. Other times resolving the conflict will involve combining some lines from the previous version and some lines from the new branch.In cases where other files have been edited (like this example) it may also involve putting some of these lines into other files.

When conflicts have been resolved a *merge commit* can be made. This will store the two parent commits and the conflicts that were resolved so they can be inspected in the future. Unfortunately sometimes people will pick the wrong option or merge incorrectly so it's good to be able to later see what conflicts they had to resolve.

REBASING

A *rebase* is a method of history rewriting in Git that is similar to a merge. A rebase involves changing the parent of a commit to point to another.

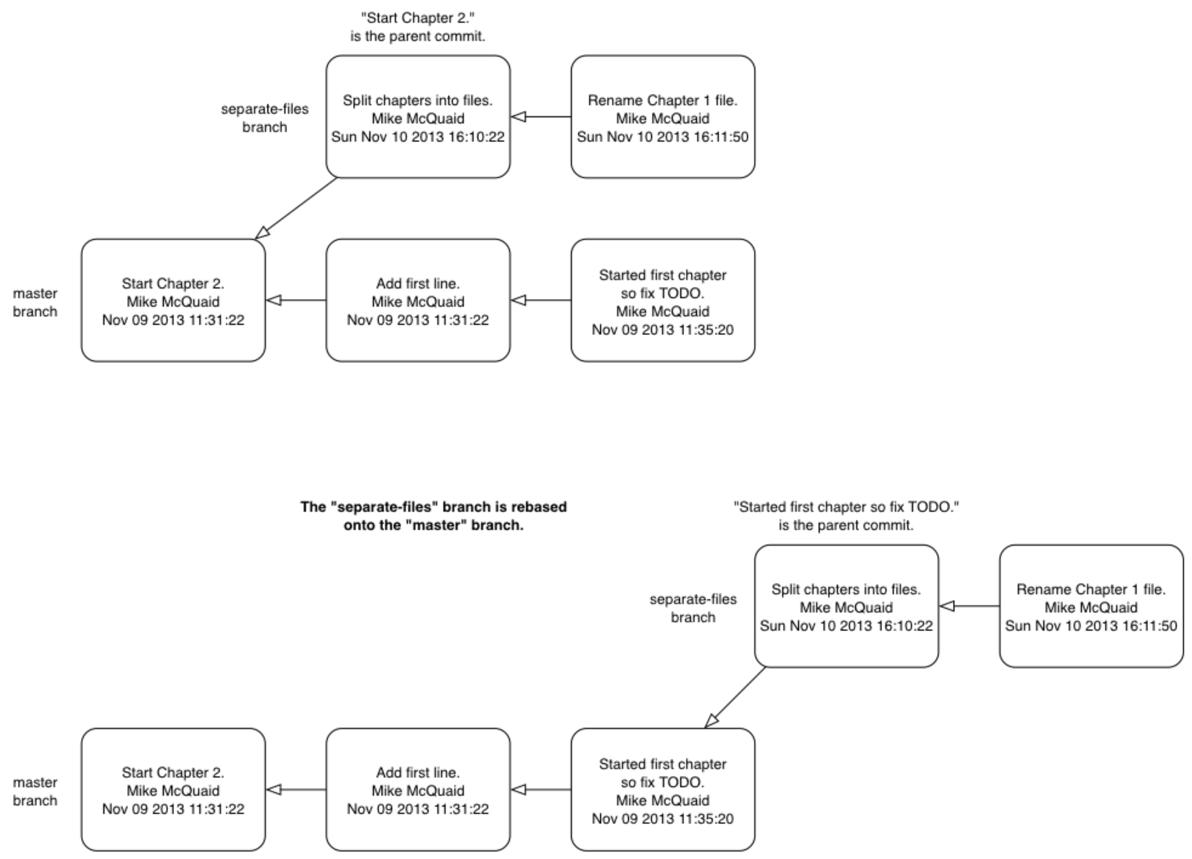

"Start Chapter 2."
is the parent commit.
separate-files branch
Split chapters into files. Mike McQuaid Sun Nov 10 2013 16:10:22
Rename Chapter 1 file. Mike McQuaid Sun Nov 10 2013 16:11:50
master branch
Start Chapter 2. Mike McQuaid Nov 09 2013 11:31:22
Add first line. Mike McQuaid Nov 09 2013 11:31:22
Started first chapter so fix TODO. Mike McQuaid Nov 09 2013 11:35:20
The "separate-files" branch is rebased onto the "master" branch.
"Started first chapter so fix TODO."
is the parent commit.
separate-files branch
Split chapters into files. Mike McQuaid Sun Nov 10 2013 16:10:22
Rename Chapter 1 file. Mike McQuaid Sun Nov 10 2013 16:11:50
master branch
Start Chapter 2. Mike McQuaid Nov 09 2013 11:31:22
Add first line. Mike McQuaid Nov 09 2013 11:31:22
Started first chapter so fix TODO. Mike McQuaid Nov 09 2013 11:35:20

Figure 2.18 Rebasing a branch on top of master

Figure 2.20 shows a rebase of the `seperate-files` branch onto the `master` branch. The rebase operation has changed the parent of the first commit in the `separate-files` branch to be the last commit in the `master` branch. This means all the content changes from the `master` branch are now included in the `separate-files branch` and any conflicts were manually resolved but were not stored (as they would be in a merge conflict).

We'll cover rebasing in more detail later in Chapter 6. All that's necessary to remember for now is that it's a different approach to a merge that can be used for a similar outcome (pulling changes from one branch into another).

2.10 Deleting a remote branch

2.10.1 Background

Now that the `chapter-two` branch has been merged into the `master` branch the new commit that made in the `chapter-two` branch is now in the `master` branch. This means that we can push the `master` branch to push all the `chapter-two` changes to `origin/master`. Once this is done (and assuming we don't want to make any more commits to the `chapter-two` branch) then `origin/chapter-two` can be safely deleted.

NOTE **Why delete the branches?**
Sometimes branches in version control systems are kept around for a long time and sometimes they are very temporary. A long-running branch may be one that represents the version deployed to a particular server. A short-running branch may be a single bug fix or feature which has been completed. In Git once a branch has been merged the history of the branch is still visible in the history and the branch can be safely deleted as a merged branch is, at that point, just a ref to an existing commit in the history of the branch it was merged into.

2.10.2 Problem

You wish to push the current `master` branch and delete the branch named `chapter-two` on the remote `origin`.

2.10.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout master` to ensure you are on the `master` branch.
3. Run `git push`.
4. Run `git push --delete origin chapter-two`. The output should resemble:

Listing 2.12 Delete remote branch

```
# git push origin :chapter-two

To https://github.com/GitInPractice/GitInPracticeRedux.git
 - [deleted]         chapter-two
```

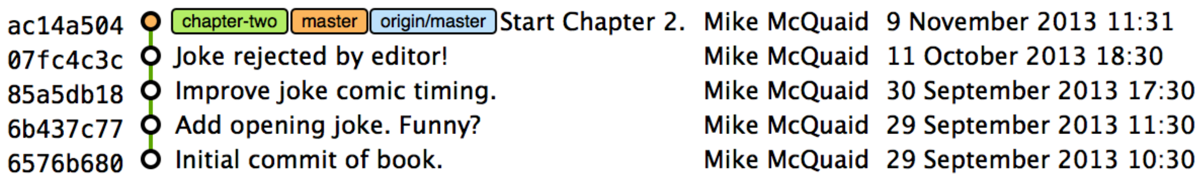
❶ **remote URL**
❷ **deleted branch**

From the deletion output:

- "remote URL (1)" shows the remote repository that the branch was deleted from.
- "deleted branch (2)" shows the name of the branch (`chapter-two`) that has been deleted from the remote repository.

You have deleted the `chapter-two` branch from the remote repository.

2.10.4 Discussion



```
ac14a504  chapter-two master origin/master Start Chapter 2.  Mike McQuaid  9 November 2013 11:31
07fc4c3c  Joke rejected by editor!                           Mike McQuaid  11 October 2013 18:30
85a5db18  Improve joke comic timing.                         Mike McQuaid  30 September 2013 17:30
6b437c77  Add opening joke. Funny?                           Mike McQuaid  29 September 2013 11:30
6576b680  Initial commit of book.                            Mike McQuaid  29 September 2013 10:30
```

Figure 2.19 Local repository after `git push origin :chapter-two`

In Figure 2.19 you can see that the `origin/master` has been updated to the same commit as `master` and that `origin/chapter-two` has now been removed.

2.11 Deleting the current local branch after merging

2.11.1 Background

The `chapter-two` branch has all its commits merged into the `master` branch and the remote branch deleted so the local branch can now be deleted too.

2.11.2 Problem

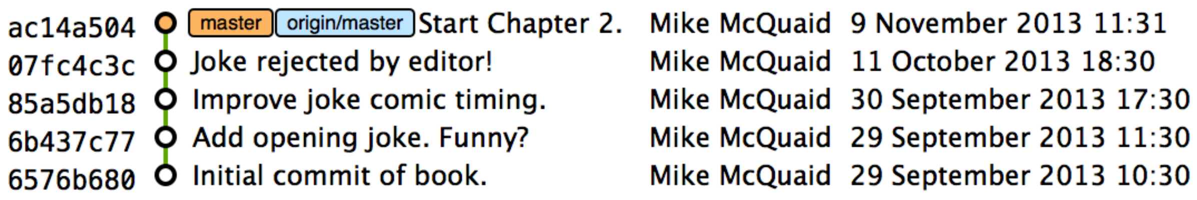
You wish to delete the local branch named `chapter-two`.

2.11.3 Solution

1. Change directory to the Git repository e.g. `cd /Users/mike/GitInPracticeRedux/`.
2. Run `git checkout master` to ensure you are on the `master` branch.
3. Run `git branch --delete chapter-two`. The output should be `Deleted branch chapter-two (was ac14a50).`

You've deleted the `chapter-two` branch from the local repository.

2.11.4 Discussion



```
ac14a504 master origin/master Start Chapter 2.  Mike McQuaid  9 November 2013 11:31
07fc4c3c Joke rejected by editor!               Mike McQuaid  11 October 2013 18:30
85a5db18 Improve joke comic timing.             Mike McQuaid  30 September 2013 17:30
6b437c77 Add opening joke. Funny?               Mike McQuaid  29 September 2013 11:30
6576b680 Initial commit of book.                Mike McQuaid  29 September 2013 10:30
```

Figure 2.20 Local repository after `git branch --delete chapter-two`

Figure 2.18 shows the final state with all evidence of the `chapter-two` branch now removed (other than the commit message).

NOTE **Why delete the remote branch before the local branch?**
We had merged all the `chapter-two` changes into the `master` branch and pushed this to `origin/master`. As a result the `chapter-two` and `origin/chapter-two` branches are no longer needed. However, Git will refuse to delete a local branch with `git branch --delete` if it has not been merged into the current branch or its changes have not been pushed to its tracking branch (`origin/chapter-two` in this case). Deleting `origin/chapter-two` first means that the local `chapter-two` branch can be deleted by `git branch --delete` without Git complaining that `chapter-two` has changes that need pushed to `origin/chapter-two`.

2.12 Summary

In this chapter you hopefully learned:

- How to push your local repository to a remote repository
- How to clone an existing remote repository
- How to push and pull changes to/from a remote repository
- That fetching allows obtaining changes without modifying local branches
- That pulling is the equivalent to fetching then merging
- How to checkout local and remote branches
- How to merge branches and then delete from the local and remote repository

Now let's learn how to perform some more advanced interactions with files inside the Git working directory.

3 Filesystem Interactions

In this chapter you will learn about interacting with files and directories in your Git working directory by learning the following topics:

- How to rename, move or remove versioned files or directories
- How to tell Git to ignore certain files or changes
- How to delete all untracked or ignored files or directories
- How to reset all files to their previously committed state
- How to temporarily stash and reapply changes to files

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

Understanding the filesystem commands around Git 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.

3.1 Rename or move a file: git mv

3.1.1 Background

Git keeps track of the changes to files in the working directory of a repository by their name. When you move or rename a file, Git does not see that a file was moved but that there is a file with a new filename and the file with the old filename was deleted (even if the contents remains the same). As a result of this renaming or moving a file in Git is essentially the same operation; both are telling Git to look for an existing file in a new location.

This may happen if you are working with tools (e.g. IDEs) which move files for you and aren't aware of Git (so don't give Git the correct move instruction).

Sometimes you will still need to manually rename or move files in your Git repository and you wish to preserve the history of the files after the rename or move operation. As you learnt in Chapter 1, 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.

3.1.2 Problem

You wish to rename a previously committed file in your Git working directory named `GitInPractice.asciidoc` to `01-IntroducingGitInPractice.asciidoc` and commit the newly renamed file.

3.1.3 Solution

1. Change to the directory containing your repository e.g. `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:

Listing 3.1 Renamed commit output

```
# git commit --message 'Rename Chapter 1 file.'

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

❶ **commit message**
❷ **no insertions/deletions**
❸ **old new filename**

You have renamed `Chapter1.asciidoc` to `01-FirstChapter.asciidoc` and committed it.

3.1.4 Discussion

Moving and renaming files in version control systems rather than deleting and recreating them is done to preserve their history. For example, when a file has been moved into a new directory you will still be interested in the previous versions of the file before it was moved. In Git's case it will try and autodetect renames or moves on `git add` or `git commit`; if a file is deleted and new file is created which has a majority of lines in common then Git will automatically detect the file was moved and `git mv` is not 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 to have consistent behavior across different versions of Git (which may have differing move autodetection behaviour).

After running `git mv` the move or rename will be added to Git's index staging area which, if you remember from Chapter 2, means that 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 inside the Git repository using the `git mv` command and the same syntax as above. If you wish to move files into or out of a repository you must use a different, non-Git command (such as a Unix `mv` command).

NOTE **What if the new filename already exists?**
If the filename that you move to already exists you will need to use the `git mv -f` (or `--force`) option to request Git to overwrite whatever file is at the destination. If the destination file has not already been added or committed to Git then it will not be possible to retrieve the contents if you erroneously asked Git to overwrite it.

3.2 Remove a file: git rm

3.2.1 Background

Removing files from version control systems requires, like moving/renaming, not just performing the filesystem operation as usual but notifying Git and committing the file. Almost any version-controlled project will see you wanting to remove some files at some point so it's essential to know how to do so. Removing version-controlled files is also more safe than removing non-version-controlled files as, even after removal, the files will 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 e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `echo Git Sandwich > GitInPracticeReviews.tmp`. This will create 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.'`

3.2.2 Problem

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

3.2.3 Solution

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

Listing 3.2 Removed commit output

```
# git rm GitInPracticeReviews.tmp

rm 'GitInPracticeReviews.tmp'

# git commit --message 'Remove Chapter 2 temporary file.'

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

❶ **commit message**
❷ **1 line deleted**
❸ **deleted filename**

You have removed `GitInPracticeReviews.tmp` and committed it.

3.2.4 Discussion

Git will only interact with the Git repository when you explicitly give it commands which is why when you remove a file Git does not automatically run `git rm` command. The `git rm` command is not just indicating to Git 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 wish to see a simulated run of `git rm` without actually removing the requested file then 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 actually removing the file.

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

NOTE **What if a file has uncommitted changes?**
If a file has uncommited changes but you still wish to remove it you will need to use the `git rm -f` (or `--force`) option to indicate to Git you wish to remove it before committing the changes.

3.3 Resetting files to the last commit: git reset

3.3.1 Background

There are times when you have made some changes to files in the working directory but you do not wish to commit these changes.

Perhaps you added debugging statements to files and have now committed a fix so want to reset all of the files that have not been committed to their last committed state (on the current branch).

3.3.2 Problem

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

3.3.3 Solution

1. Change to the directory containing your repository e.g. `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:

Listing 3.3 Hard reset output

```
# git reset --hard

HEAD is now at 06b5eb5 Remove unfavourable review file.       ❶ Reset commit
```

You have reset the Git working directory to the last committed state.

3.3.4 Discussion

The `--hard` argument signals to `git reset` that you wish 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 will reset 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) in Chapter 7.

WARNING **Dangers of using `git reset --hard`**
Care should be used with `git reset --hard`; it will immediately and without prompting remove all your uncommitted changes to any file in your working directory. This is probably the command which has caused me more regret than any other; I've typed it accidentally and removed work I hadn't intended to. Remember in Chapter 1 we learnt 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.

3.4 Delete untracked files: git clean

3.4.1 Background

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 do not wish to commit them to your version control system and instead wish to remove them to build clean versions (although this is generally better handled by *ignoring* these files as in Section 4.5).

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

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

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `echo Needs more cowbell > GitInPracticeIdeas.tmp`. This will create a new file named `GitInPracticeIdeas.tmp` with the contents "Needs more cowbell".

3.4.2 Problem

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

3.4.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git clean --force`. The output should resemble:

Listing 3.4 Force cleaned files output

```
# git clean --force

Removing GitInPracticeIdeas.tmp                    ❶ removed file
```

You have removed `GitInPracticeIdeas.tmp` from the Git working directory.

3.4.4 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 in Chapter 1 we learnt that accidentally losing any file or change committed to a version control system is very hard (and in Git, nearly impossible). This is the opposite situation; `git clean` will happily remove thousands of files very quickly which cannot be easily recovered (unless backed 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 the `git rm --dry-run` in that it prints the output of the removals that would be performed but does not actually do so.

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

3.5 Ignore files: .gitignore

3.5.1 Background

As discussed in the previous section, sometimes working directories will contain files which are *untracked* by Git and you do not wish to add them to the repository.

Sometimes these files are one-off occurrences; you accidentally copy a file to the wrong directory and wish to delete it. Usually, however, they are the product of some software (e.g. 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 just `git clean` these files each time but that would rapidly become tedious. Instead we could tell Git to ignore them so it never complains about these files being untracked and you do not accidentally add them to commits.

3.5.2 Problem

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

3.5.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `echo \*.tmp > .gitignore`. This will create 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:

Listing 3.5 Ignore file commit output

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

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

❶ commit message
❷ 1 line added
❸ created filename

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

3.5.4 Discussion

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 within the version control repository.

For example, I may have a `hello.c` file which is compiled into `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 that they can be shared between all the users of a repository and not accidentally committed.

Let’s try and add an ignored file.

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `touch GitInPractiseGoodIdeas.tmp`. This will create a new, empty file named `GitInPractiseGoodIdeas.tmp`.
3. Run `git add GitInPractiseGoodIdeas.tmp`. The output should resemble:

Listing 3.6 Trying to add an ignored file

```
# git add GitInPractiseGoodIdeas.tmp

The following paths are ignored by one of your .gitignore files:
GitInPractiseGoodIdeas.tmp
Use -f if you really want to add them.
fatal: no files added
```

❶ **ignored file**

❷ **error message**

From the add output:

- "ignored file (1)" `GitInPractiseGoodIdeas.tmp` was not added as its addition would contradict your `.gitignore` rules.
- "error message (2)" was printed as no files were added.

This interaction between `.gitignore` and `git add` is particularly useful when adding subdirectories of files and directories which may contain files that should to be ignored. `git add` will not add these files but will still successfully add all other that should not be ignored.

3.6 Delete ignored files

3.6.1 Background

When files have been successfully ignored by the addition of a `.gitignore` file you will sometimes with to delete them all.

For example, you may have a project in a Git repository which compiles input files (e.g. `.c` files) into output files (e.g. `.o` files) and wish to remove all of 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 e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `touch GitInPractiseFunnyJokes.tmp GitInPractiseWittyBanter.tmp`.

3.6.2 Problem

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

3.6.3 Solution

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

Listing 3.7 Force clean of ignored files output

```
# git clean --force -X

Removing GitInPractiseFunnyJokes.tmp
Removing GitInPractiseWittyBanter.tmp
```

❶ **removed file**

You have removed all ignored files from the Git working directory.

3.6.4 Discussion

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

The specified arguments can be combined with the others discussed in Section 4.4.4. For example, `git clean -xdf` would remove all untracked or ignored files (`-x`) and directories (`-d`) from a working directory. This will remove all files and directories for a Git repository that were not previously committed. Please take care when running this; there will be no prompt and all the files will be quickly deleted.

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

3.7 Temporarily stash some changes: git stash

3.7.1 Background

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 could *stash* your uncommitted changes to store them temporarily away and then be able to e.g. change branches, pull changes etc. without needing to worry about these changes getting in the way.

3.7.2 Problem

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

3.7.3 Solution

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

Listing 3.8 Stashing uncommitted changes output

```
# git stash save

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

You have stashed your uncommitted changes.

3.7.4 Discussion

`git stash save` actually creates a temporary commit with a pre-populated commit message and then returns your current branch to the state before the temporary commit was made. It's possibly 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:

Listing 3.9 List of stashes

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

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

If you save another stash then 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 will take it from the top. Similarly when you run `git stash` the new stash will be added will be added to the top (i.e. become `stash@{0}`) and the previous stash will no longer be at the top (i.e. become `stash@{1}`).

NOTE **Do you need to use `git add` before `git stash`**
No, `git add` is not needed. `git stash` will stash your changes whether or not they have been added to the index staging area by `git add` or not.

NOTE **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 is not needed. I've used it in the examples as it's more explicit and easier to remember.

3.8 Reapply stashed changes: git stash pop

3.8.1 Background

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

3.8.2 Problem

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

3.8.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git stash pop`. The output should resemble:

Listing 3.10 Reapply stashed changes output

```
# git stash pop

# On branch master   ❶
# Changes not staged for commit:   ❷
#   (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")   ❸
Dropped refs/stash@{0} (f7e39e2590067510be1a540b073e74704395e881)   ❹
```

❶ current branch output
❷ begin status output
❸ end status output
❹ stashed commit

You have reapplied the changes from the last `git stash save`.

3.8.4 Discussion

When running `git stash pop` the top stash on the stack (i.e. `stash@{0}`) will be applied to the working directory and removed from the stack. If there is a second stash in the stack (`stash@{1}`) then it will now be at the top (i.e. become `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 and it outputs `No stash found.`.

If you wish to apply an item from the stack multiple times (e.g. perhaps on multiple branches) then 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.

3.9 Clear stashed changes: git stash clear

3.9.1 Background

You may have stashed changes with the intent of popping them later but then realize that you no longer wish to do so. You know that the changes in the stack are now unnecessary so wish 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 good if there was a command that allowed you to do this in a single step. Thankfully, `git stash clear` allows you to do just this.

3.9.2 Problem

You wish to clear all previously stashed changes.

3.9.3 Solution

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

You have cleared all the previously stashed changes.

3.9.4 Discussion

WARNING **No prompt for `git stash clear`**
Clearing the stash will be done without a prompt and will remove every previous item from the stash so be careful when doing so. Cleared stashes cannot be recovered.

3.10 Assume files are unchanged

3.10.1 Background

Sometimes you may wish to make changes to files but have Git ignore the specific changes you have made so that operations such as `git stash` and `git diff` ignore these changes. In these cases you could just 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'm wanting to test a Rails configuration file change for a week or two while continuing to do my normal work. I don’t want to commit it because I don’t want it to apply to servers or my coworkers but I do want to continue testing it while I make other commits rather than changing to a particular branch each time.

3.10.2 Problem

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

3.10.3 Solution

1. Change to the directory containing your repository e.g. `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`.

3.10.4 Discussion

When you run `git update-index --assume-unchanged` Git sets a special flag on the file to indicate that it should not be checked for any changes that have been made. 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 particular huge and/or slow to read. This is not normally a problem on normal filesystems on which Git can quickly query if a file is modified by checking the "file modified" timestamp (rather than having to read the entire file and compare it).

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

3.11 List assumed unchanged files

3.11.1 Background

When you have told Git to assume there are no changes 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 does not seem to want to show you these changes. Additionally, you could forget that you had made these changes at all and be very confused as to why the state in your text editor does not seem to match the state that Git is seeing.

3.11.2 Problem

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

3.11.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git ls-files -v`. The output should resemble:

Listing 3.11 Assumed unchanged files listing output

```
# git ls-files -v

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

❶ committed file
❷ assumed unchanged file

From the listed files:

- "committed files (1)" are indicated by an uppercase `H` tag at the beginning of the line.
- "assumed unchanged file (2)" is indicated by a lowercase `h` tag.

3.11.4 Discussion

Like `git update-index`, `git ls-files -v` is a low level command that you will 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 that it is followed by tags which indicate file state.

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

`grep '^[hsmrck?]'` filters the output filenames to only show those that begin with any of the lowercase `hsmrck?` characters.

`cut -c 3-` filters the first two characters of each of the output lines so e.g. `h` followed by a space in the above example.

With these combined the output should resemble:

Listing 3.12 Assumed unchanged files output

```
# git ls-files -v | grep '^[hsmrck?]' | cut -c 3-

01-IntroducingGitInPractice.asciidoc
```

❶ assumed unchanged file

NOTE **How do pipes, `grep` and `cut` work?**
Do not worry if you don't understand quite how Unix pipes, `grep` or `cut` work; this book is about Git rather than shell scripting after all! Feel free to just use the above command as-is as a quick way of listing files that are assumed to be unchanged.

3.12 Stop assuming files are unchanged

3.12.1 Background

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 long-term they should probably be committed. At some point you will wish to tell Git to monitor any changes that are made to these files once more.

With the example I gave previously in Section 4.10 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 would not show up so I had to stop Git ignoring this particular change before I could make a new commit.

3.12.2 Problem

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

3.12.3 Solution

1. Change to the directory containing your repository e.g. `cd /Users/mike/GitInPracticeRedux/`
2. Run `git update-index --no-assume-unchanged 01-IntroducingGitInPractice.asciidoc`. There will be no output.

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

3.12.4 Discussion

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

3.13 Summary

In this chapter you hopefully learned:

- 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

Now let’s learn how to visualize history in a Git repository in different formats.