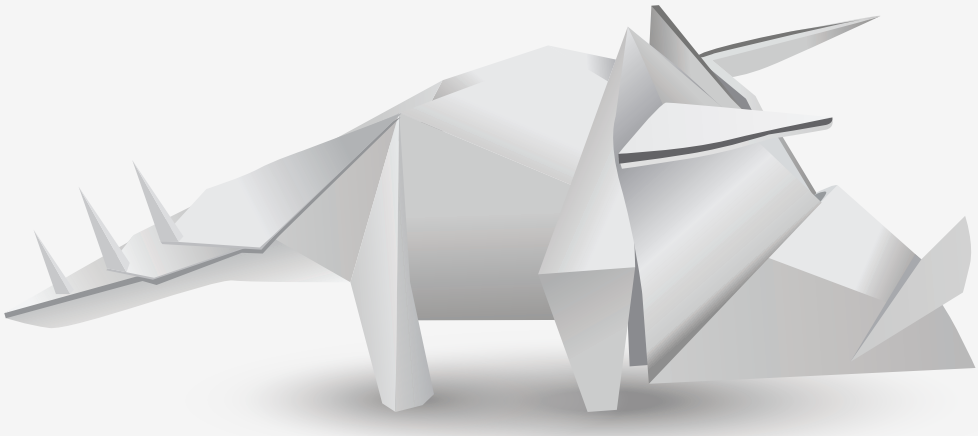
sitepoint

JUMP START GIT

BY **SHAUMIK DAITYARI**



TAKE CONTROL OF YOUR CODE AND ASSETS

Summary of Contents

sitepoint

JUMP START GIT

BY **SHAUMIK DAITYARI**

Jump Start Git

by Shaumik Daityari

Product Manager: Simon Mackie
Technical Editor: Craig Buckler
Technical Reviewer: Alexey Novak
English Editor: Ralph Mason
Cover Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9941826-5-4 (print)

ISBN 978-0-9943469-2-6 (ebook)
Printed and bound in the United States of America

About Shaumik Daityari

Shaumik is an optimist, but one who carries an umbrella. He is currently pursuing his MBA at IIM Lucknow, after completing his M.Tech at IIT Roorkee. Co-founder of The Blog Bowl, he loves writing, when he's not busy keeping the blue flag flying high.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

To my grandfather, Gagga, who
got me started with books.

Table of Contents

Preface

Most organizations involved with software development make use of version control. However, despite it being so useful, developers often think of version control as a separate skill, and only learn the bare minimum to get by, or put off learning version control until absolutely necessary. This is to miss out on some of the powerful utilities that version control provides.

This book is about Git—a free, open-source version control system. The aim of this book is to help beginners get up and running with version control quickly, and then to take a deeper dive into its mechanics if they so desire.

Who Should Read This Book

This book is suitable for anyone interested in managing multiple revisions of code, data and documents. It's ideal for beginners who plan to start working with Git, but it's also useful for seasoned developers who are looking to consolidate their understanding of Git.

What's Covered in This Book?

The book starts off by outlining the philosophy of version control and why Linus Torvalds decided to create Git for the Linux kernel.

It then proceeds to introduce the basics of Git and the various terms related to it. Most of the chapters in this book focus on using the command line to explore Git, as there's no better way to use all its features.

The focus next turns to using Git in a team environment, where version control is essential. This is where cloud services like GitHub, Bitbucket and GitLab come in. A general overview of the workings of GitHub is included to assist in getting started with that service.

This book also deals with workflows that are generally adopted by organizations. Considerable time is devoted to "branching", as this is one feature that makes Git arguably the best option for version control.

The focus then shifts to specific Git tools that assist with using Git more efficiently. A separate chapter is devoted to fixing errors while working with Git.

The bulk of the book discusses the usage of Git from the command line, but it ends by examining GUI tools, explaining their advantages and disadvantages over the command line interface.

Finally, we'll look at how people use Git for purposes other than code versioning, the problem of managing huge repositories through Git, and the future of Git.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all of it, ⋮ will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-design-real-user-
➥testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

https://www.sitepoint.com/premium/books/jsgit1

The book's website, containing links, updates, resources, and more.

http://community.sitepoint.com/

SitePoint's forums, for help on any tricky web problems.

books@sitepoint.com

Our email address, should you need to contact us for support, to report a problem, or for any other reason.

Acknowledgments

Writing this book has been my most challenging undertaking. The book would be incomplete without referring to the help of others. First and foremost, I'd like to thank my friends at IMG, IIT Roorkee, for helping me understand Git. Next, Louis Lazaris, who's had a significant impact on how I write since I started contributing to SitePoint. Without him, this book would never have been possible.

Thank you, Simon, for giving me the opportunity to write this book, and for patiently clearing my doubts about the complex process of getting published. Thank you also for reviewing this book with such precision. Thanks to Craig, for being the technical reviewer and challenging the fringes of my knowledge.

An extra special thanks to my GSoC mentor, Alexey Novak, for inspiring me to always explore new things and also for reviewing this book, even with a busy schedule. Credit also goes to my parents and family members for their support and encouragement, when I was going through a difficult phase of transition.

A special thanks to Alex Walker, for designing the cover. Nothing else could explain what version control stands for in such simple terms.

Last, and certainly the least, to the examiner of my English answer script during matriculation. That certainly got me going.

Want to take your learning further?

Thanks for choosing to buy a SitePoint book. Would you like to continue learning? You can now gain unlimited access to ALL SitePoint books and courses, plus high-quality books from our selected partners, at SitePoint Premium[1]. Enroll now and start learning today!

[1] https://www.sitepoint.com/premium/home

Chapter 1

Introduction

In my freshman year in college, I started work on my first intranet application. The files in the main directory of the partially functioning application looked something like Figure 1.1:

Figure 1.1. The directory structure of my first web application titled "Online Exams"

Looking at the file names in this directory, you can see that I used some very similar names, such as **exam.php**, **exam1.php** and **examfile.php**. The purpose of that naming convention was to create new versions of my application without losing the old, working logic—in case the new ideas failed! I assumed that, because I understood what each of those files did, it should be fine to have a bunch of similarly named files.

However, there were two flaws in that thinking. Firstly, anyone else examining this code wouldn't be able to make sense of this mess. Secondly, after a few months,

even I was struggling to recall what each version of these files was for. Clearly, I needed a better system for managing the various versions of my files.

If I had this much trouble working on a small, personal project, imagine how difficult it must have been for larger software projects, with thousands of files and contributors distributed all over the world! Developers once used emails to coordinate changes among team members. When they made changes to a project, they would each create a "diff" file with all their changes and email it to the lead developer, who would incorporate them into the project if everything worked properly.

When you're working on the same files as other developers, keeping track of what you've changed and trying to merge it with work done by your peers becomes very difficult. It can result in a lot of confusion and time wasting.

Imagine another situation, where you're working on an idea and your boss wants to see what you've already completed. Ideally, you'd want to be able to do the following:

- stash away the changes and revert to the last stable state
- show your boss the latest completed work
- resume your work with the current state once that's done.

All of the situations I've described above give rise to the need for what's known as "version control". So let's find out what that is.

Version Control

Version control (or **revision control**) is a system that records changes to a file or a group of files and directories over time, so that you can review or go back to specific versions later. Over the course of this book, I'll demonstrate how this works; but first, let's examine in more detail what version control is.

Quite literally, version control means maintaining versions of your work—perhaps most commonly in the form of source code, though it can be used for other kinds of work too. You may like to think of version control as a tool that takes snapshots of your work across time, creating checkpoints. You can return to those checkpoints any time you want. Not only are the changes recorded in these checkpoints, but

also information about who made the changes, when they made them, and the reasons behind the changes.

I've already mentioned the first objective of version control—to backup and restore. Version control eliminates the need to create backup files like I was doing in my college days (that is, endless duplicates with different names). Version control also gives you the ability to return to previous states of your work without losing the current state.

Version Control Doesn't Replace the Need for a Regular Backup Solution

The word "backup" above, as noted, refers to the process of creating multiple copies of the same file. Git removes the need for that. However, this is different from regularly backing up your files to an external source—such as a portable drive or cloud storage—to ensure you don't lose anything following a disk failure.

Next, version control lets you synchronize your work with peers who are working on the same projects. In other words, it enables you to collaborate with others without the possibility of someone's changes being lost.

Version control also tracks changes to a project and other data associated with the changes. It makes the process of debugging your code easy too, which we'll explore in some detail.

Conflicts in files can also be resolved through version control—such as when multiple people have made changes to a file that clash. A version control system highlights the conflicts and provides an opportunity to fix them.

Yet another feature of version control is that it enables work on multiple features of a project at the same time. This gives great scope for experimentation, trial and error. Each feature can be developed independently of the others, and can easily be removed if it doesn't work out.

Now that you've been introduced to the concept of version control, let's look at how we may already be using version control in our daily lives.

Examples of Version Control in Daily Life

You've probably visited the Wikipedia[1] site at some point. You may even have taken the opportunity to update its content, too—as we're all invited to do so. When editing a page, you may also have checked its history. That's where things get really interesting.

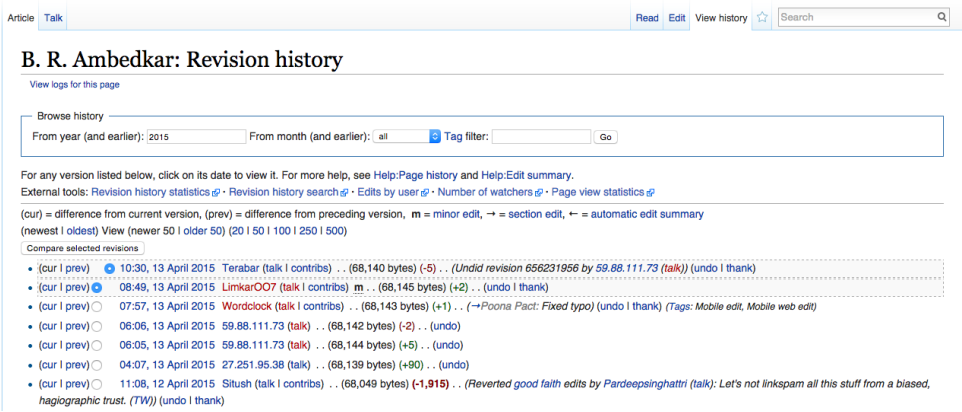


Figure 1.2. History of Wikipedia Page for B. R. Ambedkar

The history page shown in Figure 1.2 lists changes to that page. It also records the time of the change, the user who made it, and a message associated with the change. You can examine the complete details of each edit, and even revert back to an older version of the page. This is a good example of a simple form of version control.

[1] https://en.wikipedia.org/wiki/Main_Page

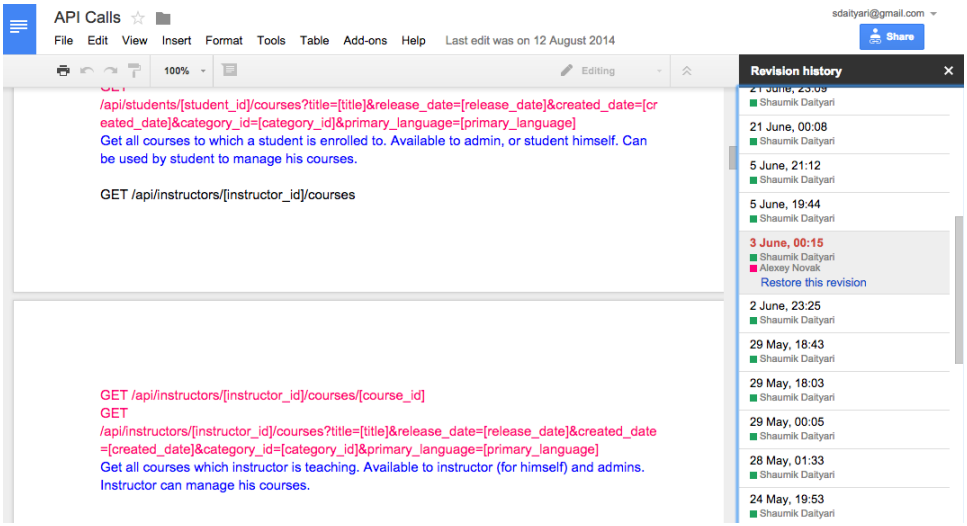


Figure 1.3. Revision history of Google Docs

Google Docs provides another example of version control that you might experience in daily life. If you check the revision history of a file in Google Docs, shown in Figure 1.3, you'll notice that Google saves the state of your file after every few changes. You can preview the status of the document in any of those previous states—and choose to revert back to it, if needed.

Version Control Systems: the Options

There are two types of version control systems (VCS), known as "centralized" and "distributed".

Centralized systems have a copy of the project hosted on a centralized server, to which everyone connects to in order to make changes. Here, the "first come, first served" principle is adopted: if you're the first to submit a change to a file, your code will be accepted.

In a **distributed** system, every developer has a copy of the entire project. Developers can make changes to their copy of the project without connecting to any centralized server, and without affecting the copies of other developers. Later, the changes can be synchronized between the various copies.

In the earliest version control systems, files were tracked only locally, and only one person could work on a file at a time. Examples of these include Source Code Control

System (SCCS) and Revision Control System (RCS), which were common in the 1970s and 1980s.

The next step forward was the introduction of client-server version control systems, which enabled multiple authors to work on the same file (although some still worked on the first come, first served basis). Examples of such systems include Concurrent Versions System (CVS) and Subversion, which are still in use today.

Since around 2005, distributed systems have gained widespread acceptance, with the emergence of systems such as Git[2], Mercurial[3] and Bazaar[4].

VCS Is Not CVS

Don't confuse the abbreviations VCS (Version Control System) and CVS (Concurrent Versions System). CVS is just one of the many kinds of VCS.

Back in my freshman year, version control systems were available. However, in the example of my small project, I didn't use one, simply because I was a beginner and didn't know they existed. Many people first get introduced to version control systems when they start working with a team.

Enter Git

This book is about Git, a distributed version control system. Git tracks your project history, enabling you to access any version of it back in time. It also allows multiple people to work on the same project, helping avoid confusion when more than one person tries to edit the same file.

Git was created by Linus Torvalds (who is also known for the Linux kernel), and Junio Hamano is its primary developer. Git, as described on the Git website, is a source code management (SCM) solution, but essentially it's just a type of version control system.

The primary objective behind Git was to implement and design a version control system that was distributed, reliable and fast. While working on Linux, Torvalds

[2] http://git-scm.com/
[3] https://mercurial.selenic.com/
[4] http://bazaar.canonical.com/en/

needed a version control system to manage the Linux code base. BitKeeper was a distributed system at that time, but Torvalds believed that, although BitKeeper was a good option, being a commercial product made it unsuitable for the development of an open-source project like Linux.

Torvalds had three criteria for a version control system: it had to be distributed, efficient and safe from corruption. There was no open-source, distributed version control system in the mid 2000s that could satisfy all these conditions. Hence, Git was developed out of necessity.

Git's Philosophy

Torvalds once explained in a Google Tech Talk[5] his reasons for creating Git. He has very strong views on the subject of version control, and I suggest you go through the talk once to understand the philosophy of Git. In this talk, Torvalds explains that he came up with the name Git because he believes the silliest names are our best creations. However, I recommend that you only watch the talk after you're comfortable with the basic Git operations, as it's not a tutorial: it's aimed at users who have some knowledge of Git or other version control systems.

Advantages of Distributed Version Control Systems

Torvalds insisted on a distributed system because of the independence it affords to developers. With a distributed system, you can work on your copy of the code without having to worry about ongoing work on the same code by others. What makes it even better is that any distributed copy of the project can contain all the history of the project. A distributed system also lets you work offline, meaning you can make changes without having access to the server that stores the central repository.

Another advantage of distributed systems is that you can sync your repositories among yourselves, bypassing the central location. Let's say the access to the main server goes down and you have to collaborate with a colleague. You can share changes with your colleague and continue to work on the project together, and then later push all your changes to the location everyone has access to.

[5] https://www.youtube.com/watch?v=4XpnKHJAok8

In a centralized system, anyone who makes a change needs to be given access to the central location. In contrast, in a distributed system, new developers can make changes to their own repositories without being granted write access, while more experienced contributors can be given write access and the ability to review other contributions before merging them into the repository. Managing access is easier in distributed systems.

Git and GitHub

Since its creation, Git has become immensely popular—not only due to its own merits and the fact that Torvalds created it, but also because of the popular code sharing site GitHub[6].

People often confuse Git and GitHub, but they are quite different things. GitHub provides services that are *related to* Git. It's a website that helps you manage Git-controlled projects.

GitHub allows users to put their Git repositories on the cloud, and to perform Git-based operations through a web interface. It also provides desktop and mobile apps that offer the same services. GitHub was launched a few years after Git, and remains very popular among open source enthusiasts.

There are many other websites like GitHub, such as Bitbucket[7] and GitLab[8]. GitHub and Bitbucket are cloud-based solutions, but GitLab allows you to set up this functionality on your own servers. Other, similar services have come and gone, but these options have remained popular over the last few years. We'll explore these code sharing websites in a later chapter, and discuss how you can make use of them.

Conclusion

What Have You Learned?

- What is version control?
- How do we unknowingly use version control in our lives?

[6] https://github.com/
[7] https://bitbucket.org/
[8] https://about.gitlab.com/gitlab-com/

- What are the types of VCS?
- What is Git? What are its capabilities?

What's Next?

In the next chapter, we'll look at how to install Git and use it in your projects.

Chapter 2

Getting Started with Git

Now that we have a basic concept of what a version control system does, let's get our feet wet with Git.

Installation

The first step is to install Git. Git's official website provides detailed instructions on installing Git on your local machine[1], depending on your operating system.

- If you're using Linux, you can install Git through the terminal using a package manager. For the popular Linux distro Ubuntu, Git can be installed using `apt-get`:

```
apt-get install git
```

- In OS X, if you have Homebrew[2], you can install Git using the command line through the following command:

[1] http://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[2] http://brew.sh/

```
brew install git
```

- If you're on Windows, the official build of Git[3] can be downloaded from the Git website.

GUI Tools

For Windows and OS X, you can also install Git as a part of a GUI tool such as the GitHub for Desktop[4] and SourceTree[5]. We'll cover GUI tools in detail in a later chapter. However, for most parts of the book, we'll stick to the command line interface to really understand how Git works.

If you're using an operating system other than these three, like Minix[6] or HelenOS[7], or if you want to get the latest development version of Git for testing and development, you can install Git from its source. Grab a tarball of the desired version of Git from GitHub[8], untar it and check the README file for instructions on how to install Git. However, I wouldn't recommend following this unless you know what you're doing, as this process can lead to errors, and development versions may be unstable.

The Git Workflow

Git doesn't track all of the files stored on your computer. You need to instruct Git to track certain files and directories. This process is called **initialization**. The parent directory containing your project—all the files and directories to be tracked by Git—is called a **repository**. This repository might contain many files and directories, or even just a single file.

There are three basic operations performed by Git on your project (shown in Figure 2.1 below): track, stage, and commit.

- **Track**. Once you've initialized your repository, you'll need to add files to your project. Any files you add are initially untracked by Git. You need to specify

[3] http://git-scm.com/download/win
[4] https://desktop.github.com/
[5] https://www.sourcetreeapp.com/download/
[6] http://www.minix3.org/
[7] http://www.helenos.org/
[8] https://github.com/git/git/releases

that you want Git to **track** them. Git monitors tracked files for changes and ignores untracked files.

- **Stage**. After making the required changes to your files, you need to **stage** them. Staging is a way of tagging certain (or all) changes that you want to keep a record of.

- **Commit**. The next step is to create a **commit**. A commit is like a photograph that records the current state of your code. You can go back to a certain commit at a later time, view the status of the repository with respect to that commit, and check the changes that were made in the commit. The commit records the changes in a repository since the last commit. You can revert back to any commit at any point of time. Each commit contains a **commit hash** that uniquely identifies the commit, the author details, a commit message, and the list of changes in that commit.

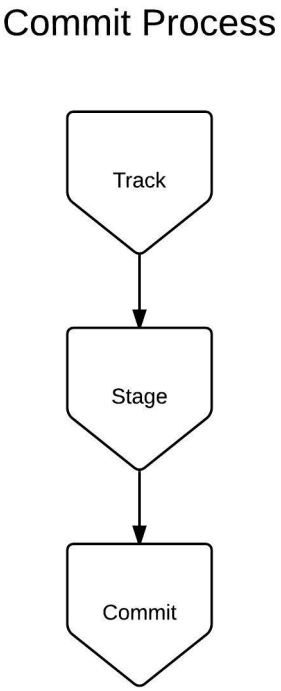


Figure 2.1. Commit workflow

Once you've committed your files, you may wish to **push** them to a remote location. A push refers to the process of sending the changes you've made in your local repository to a remote location. A remote location is a copy of your repository stored on a remote server. (We'll set up a remote repository later in this chapter.)

Essentially, the flow chart in Figure 2.2 below illustrates the steps that we'll follow in this chapter:

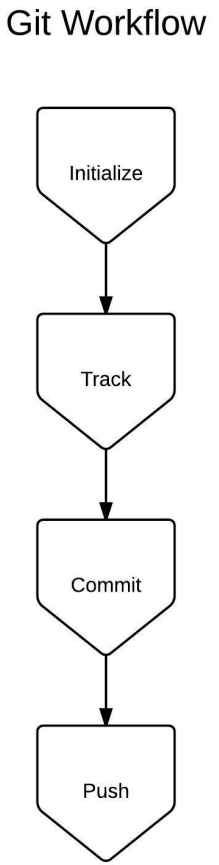


Figure 2.2. The Git workflow

Baby Steps with Git: First Commands

Set Configuration Settings

Before we proceed with using Git in a project, let's define a few global settings:

```
git config --global user.name "Shaumik"
git config --global user.email "sdaityari@gmail.com"
git config --global color.ui "auto"
```

The commands are fairly self-explanatory. We set the default name and email to be associated with our commits. We also set the `color.ui` to `"auto"`, to enable Git to color code the output of Git commands on the terminal. The `--global` setting allows these settings to be applied to any other repository that you work on locally.

If you don't set the values for name and email, they are left empty. When you make a commit, it takes different values depending on the OS or the GUI tool that you use. When you make a commit without setting these parameters, Git will automatically set them based on the username and hostname. For instance, the name is set to the name of the user that is logged in to the computer in OS X, whereas in Linux, the name is set to be the username of the active user account. In both cases, the email is set as `username@hostname`.

If you want to check all the configuration settings for your repository, you can run the following command:

```
git config --list
```

Also, if you want to edit any of your configuration settings, you can do so by editing the **~/.gitconfig** file in Linux and OS X, where ~ refers to your home directory. In Windows, it's located in your home directory: **C:/Users/<username>/.gitconfig**.

Create a Git Project

Let's first create a directory where we'll store the files for our project:

```
mkdir my_git_project
cd my_git_project
```

The first command creates a new directory, and the second changes the active directory to the newly created one. These two commands work on all operating systems (Windows, OS X, and Linux).

So, **my_git_project** is the parent directory that will contain all the files for this project. From now on, we'll refer to it as our project's repository.

Now that we're in the repository, we need to initiate Git for that directory using the following command:

```
git init
```

Issuing Git Commands

Just like `git init`, all Git commands start with the keyword `git`, followed by the command.

Git Autocomplete

When working in the terminal, developers often use the **Tab** key for autocompletion. However, this doesn't work on Git commands by default. You can install an autocomplete script for Git using the following commands. Note that this only works on Linux and OS X.

- Download the autocomplete script and place it in your home directory:

```
curl https://raw.githubusercontent.com/git/git/master/
➥contrib/completion/git-completion.bash -o
➥~/.git-completion.bash
```

- Add the following lines to the file **~/.bash_profile**:

```
if [ -f ~/.git-completion.bash ]; then
    . ~/.git-completion.bash
fi
```

If you're using Git Bash on Windows, autocompletion is preconfigured. If you're using Windows command prompt (`cmd.exe`), you'll need to install Clink[9].

Create Our First Commit

Let's look at the repository again. Notice the newly created **.git** directory, shown in Figure 2.3 (line 4). All information related to Git is stored in this repository. The **.git** directory, and its contents, are normally hidden from view.

```
SMA:~ donny$ mkdir my_git_project
SMA:~ donny$ cd my_git_project/
SMA:my_git_project donny$ git init
Initialized empty Git repository in /Users/donny/my_git_project/.git/
SMA:my_git_project donny$ ls -al
total 0
drwxr-xr-x   3 donny  staff   102 Apr 18 17:47 .
drwxrwxrwx+ 51 donny  staff  1734 Apr 18 17:47 ..
drwxr-xr-x  10 donny  staff   340 Apr 18 17:47 .git
SMA:my_git_project donny$
```

Figure 2.3. Initializing a Git repository

Don't Edit .git

Never edit any files in the **.git** directory. It can corrupt the whole repository. This book doesn't discuss the internals of Git, and thus doesn't include working on this hidden **.git** directory.

Now that we've initialized Git, let's add a few files to our repository. On your computer, navigate to the **my_git_project** directory and add three text files with the following names: **my_file**, **myfile2** and **myfile3**. Place some content in each one, such as a simple sentence.

After adding the files, let's return to the terminal and run the following command to see how Git reacts:

[9] http://mridgers.github.io/clink/

```
git status
```

You can see the output in Figure 2.4.

Demonstration Only

The file names **my_file**, **myfile2** and **myfile3** are used for demonstration purposes. They signify three different files and not the different versions of the same file.

Checking the Status

`git status` is perhaps the most used Git command—as you'll see over the course of this book. In simple terms, this command shows the status of your repository. It provides a lot of information, such as which files are untracked, which are tracked and what their changes are, which is the current "branch", and what the status of the current branch is with respect to a "remote" (we'll discuss branches and remotes later). You should frequently check the status of your repository.

```
SMA:my_git_project donny$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        my_file
        myfile2
        myfile3

nothing added to commit but untracked files present (use "git add" to track)
SMA:my_git_project donny$
```

Figure 2.4. Status of the repository

In a Git repository, any file that is added is either tracked or untracked. A file is said to be tracked when Git monitors the changes being made to that file. On the other hand, the changes to an untracked file are ignored by Git and do not form a part of any commits.

Checking the status of our repository, we can see that three files are currently marked in red. They're also grouped as untracked. Git does not track all files in a repository. You can explicitly tell Git which files to track and which to ignore.

In order to track these files, we run the following command:

```
git add my_file myfile2 myfile3
```

As an alternative, you can simply run the following:

```
git add .
```

The `.` (period) is an alias for the current directory. Running `git add .` tells Git to track the current directory, as well as any files or sub-directories within the current directory.

Beware of Adding Unwanted Files

Don't make a habit of using `git add .` as you may end up adding unnecessary files to the repository. You should add only those files that are a part of your package. Adding files like compiled files and configuration files just increases the size of your repository. Configuration files may also contain database passwords, which could lead to a security risk if committed to the repository.

Now that we've set our new files to be tracked by Git, let's check the status of the repository again, shown in Figure 2.5.

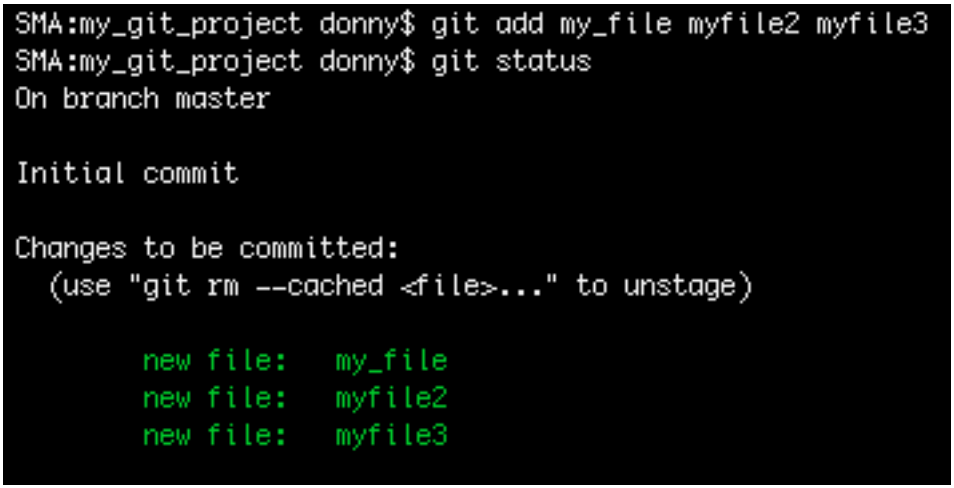


Figure 2.5. Status of the repository after tracking files

We're now ready to make a commit:

```
git commit -m "First Commit"
```

The `-m` option specifies that you are going to add a message within the command. (The message is the text in quotes after `-m`: "First Commit".) Alternatively, you can just run `git commit`, and a text editor will open up and ask you to enter a commit message.

Make Your Commit Messages Meaningful!

A meaningful commit message is an essential part of your commit. You can give a meaningless commit message like "Commit X", but in the future, it might be difficult for someone else (or even you) to understand why you created that commit.

```
SMA:my_git_project donny$ git commit -m "First Commit"
[master (root-commit) b6bd481] First Commit
 3 files changed, 3 insertions(+)
 create mode 100644 my_file
 create mode 100644 myfile2
 create mode 100644 myfile3
SMA:my_git_project donny$
```

Figure 2.6. First commit message

Notice the string `b6bd481` shown in Figure 2.6 (second line). It's the hash of the commit, or its identity. (A hash is a unique, identifying signature for each commit, generated automatically by Git.) What's shown here is a short version of a considerably longer string, which we'll look at further below.

Further Commits with Git

The first commit in a Git repository is a little different from subsequent commits. In subsequent commits, Git is already tracking the files you're working on (unless you're adding new files). So we'll need another important command, `git diff`, which shows you the changes in the tracked files since the last commit.

Let's make some changes to the files and see how Git reacts. For demonstration purposes, I've added a line to **my_file**, and some extra words to an existing line in **myfile2**. Let's check the status of the repository by running `git status`:

```
SMA:my_git_project donny$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   my_file
        modified:   myfile2

no changes added to commit (use "git add" and/or "git commit -a")
```

Figure 2.7. Status of the repository after making changes to files

As shown in Figure 2.7, Git shows that certain changes have been made to two files. We can also see exactly what was changed in the files, by running the following command:

```
git diff
```

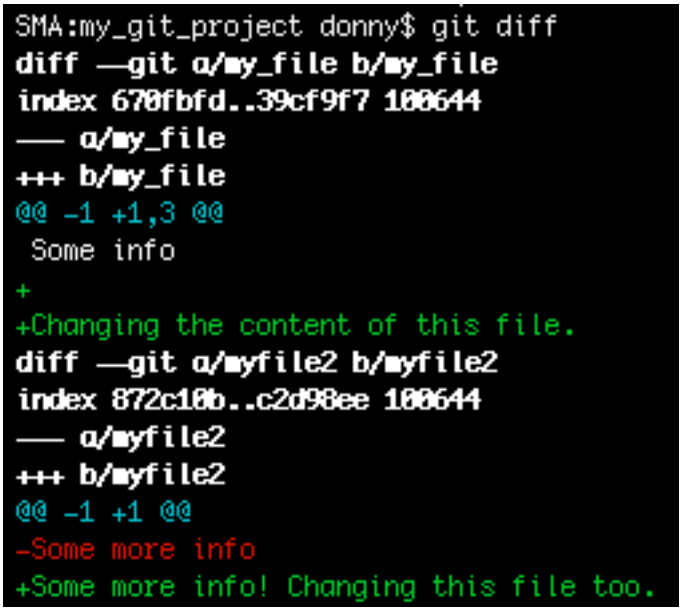


Figure 2.8. Changes in files tracked by Git

The `diff` command shows the changes that have been made to the tracked files in the repository since the last commit. In the output shown in Figure 2.8, green lines starting with a + sign show what's been added, and the red line starting with a - sign shows what's been removed. (When you edit a line of code, the same thing happens: the old line is shown in red with a - sign, and the new version of the line is shown in green with a +.)

If you want to check the changes in a single file, add the file name after the `diff` command. For instance:

```
git diff my_file
```

`Diff` Only Shows Changes In Tracked Files

As mentioned earlier, Git tracks only the files that you ask it to. The `git diff` command shows the changes only in tracked files.

After you've reviewed the changes you made, you need to "stage" the changes to be committed:

```
git add my_file myfile2
```

Alternately, you can add all tracked files like so:

```
git add -u
```

You can go one step further and add only parts of the changes to a file to the commit. This process is a bit complex, though, and we'll tackle it in a later chapter.

Now that you've staged the files, they're ready to be committed:

```
git commit -m "Made changes to two files"
```

```
SMA:my_git_project donny$ git add my_file myfile2
SMA:my_git_project donny$
SMA:my_git_project donny$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   my_file
        modified:   myfile2

SMA:my_git_project donny$ git commit -m "Made changes to two files"
[master e95a5ae] Made changes to two files
 2 files changed, 3 insertions(+), 1 deletion(-)
SMA:my_git_project donny$
```

Figure 2.9. Second commit

Be Careful of Shortcuts

You can skip the adding (staging) of a modified file by postfixing `-a` to the `git commit`, which performs the add operation. However, you should avoid doing this, because it can lead to mistakes. Firstly, postfixing `-a` only adds tracked files—so you'd miss any untracked files that you may have wanted in the commit. Secondly, it may be that you've modified two files but want them to appear in separate commits. A `git commit -a` would add both files to the same commit.

Always Review Your Changes

I mentioned earlier that `git status` is perhaps the most used command. However, the most important command is probably `git diff`. Never stage files for commit before reviewing the changes that you've made in them. Also, stage files for commit individually after carefully reviewing the changes that were made to them.

Why `git add` Again?

At this point, you may think—why add tracked files again? Well, before you commit, Git needs you to specify which files you want to commit. It may happen that you've make changes to two files, but only want to commit one of those files.

The process is like sending a package. `git add` is adding an item to the package. `git commit` is sealing the package and writing a note on it. `git push` (which I'll explain shortly) is sending the package to the recipient.

Commit History

Now that we have more than one commit, let's explore a new area of Git—the **history** of the project. The simplest way of reviewing the history of a project is running the following:

```
git log
```

This command shows the commits that we've made so far:

```
SMA:my_git_project donny$ git log
commit e95a5aedcb0c03344a23e16c5816731aef330067
Author: Shaumik <sdaityari@gmail.com>
Date:   Mon Apr 20 11:22:20 2015 +0530

    Made changes to two files

commit b6bd481ac209bbd8d0b663c19ac42d1ac1f193f4
Author: Shaumik <sdaityari@gmail.com>
Date:   Sat Apr 18 18:58:10 2015 +0530

    First Commit
```

Figure 2.10. Commit history of the project

The history (Figure 2.10) shows the list of commits, each with a unique hash, an author, a timestamp and a commit message.

Previously in this chapter (see Figure 2.6), we encountered a commit hash that was truncated. Although the long 40-character commit hash uniquely identifies each commit, usually five or six characters are enough to identify them in a repository:

```
git show b6bd481
```

The `git show` command lists information about a commit. Let's see how short we can go until Git fails to identify the hash:

```
git show b6bd481
git show b6bd48
git show b6bd4
git show b6bd
git show b6b
```

It's only once we're down to the first three characters, shown in Figure 2.12, that Git gives us a fatal error:

```
ambiguous argument 'b6b': unknown revision or path not in
the working tree.
```

Although it only failed at three characters in our repository with a very short history, it will probably need to be longer in repositories with a considerably longer history.

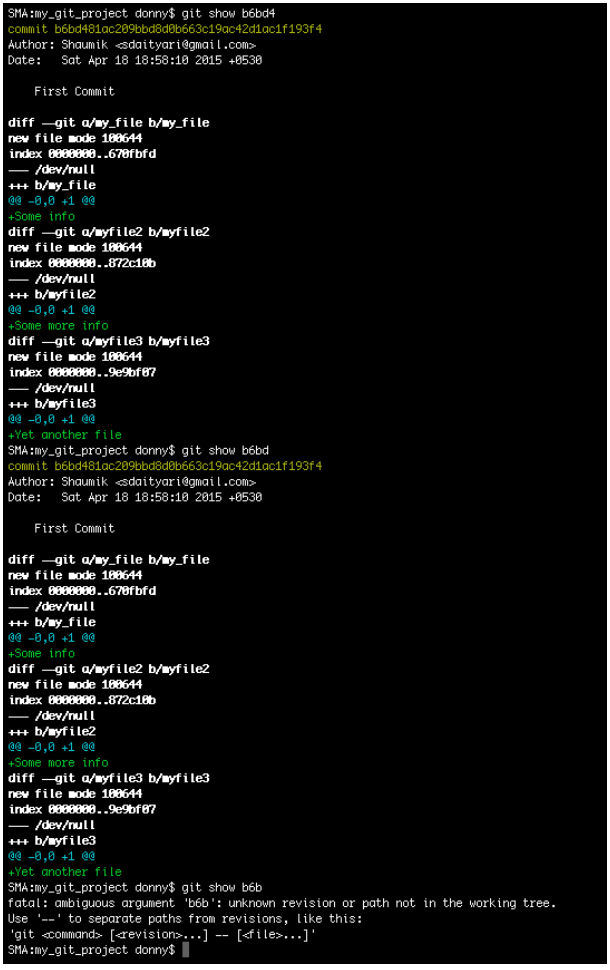


Figure 2.11. Checking how short we can go until Git fails to identify a commit hash

The .gitignore File

Although I've mentioned that Git only tracks files you explicitly ask it to, it could happen that you ask it to track some files by mistake. You need a way to hide certain files from Git that you know you'll never want it to track. This is exactly what a **.gitignore** file does.

A **.gitignore** file is added to the root directory of the repository, and it lists files you don't want Git to track or display as part of `git status`. You can add items to the **.gitignore** file and commit them.

Unintentionally Tracking a File Listed in .gitignore

Although a file listed in **.gitignore** is not meant to be tracked, it's possible that you could accidentally tell Git to track a file that's listed in there. If that happens, you won't get any error message. This is another reason you should avoid running `git add .` as it may cause files to be tracked by Git unintentionally.

Examples of files that you might want to add to **.gitignore** include compiled files with extensions like **.exe** and **.pyc**, local configuration files, OS X **.DS_Store** files, **Thumbs.db** on Windows, directories of modules in Node.js and build folders of Grunt or gulp.js.

Let's have a look at what a **.gitignore** file looks like:

```
configuration/
some_file.m
*.exe
```

The three lines in this sample file are used to tell Git to ignore a whole repository and its contents (the **configuration** directory), a single file (**some_file.m**), and all files with a **.exe** extension.

The screenshot in Figure 2.12 below shows the effect of a **.gitignore** file that tells Git to ignore ***.exe** files that has already been committed to the repository. I've created a new file called **b.exe** in our project directory, but Git is ignoring it. `git status` shows that there is nothing to commit.

```
SMA:my_git_project donny$ git status
On branch master
nothing to commit, working directory clean
SMA:my_git_project donny$ echo "something" > b.exe
SMA:my_git_project donny$ ls
b.exe           my_file         myfile2         myfile3         sample.exe
SMA:my_git_project donny$ git status
On branch master
nothing to commit, working directory clean
SMA:my_git_project donny$
```

Figure 2.12. Effect of **.gitignore** file

Hiding .gitignore from Git

Although it's advised to add the **.gitignore** file to your repository, you can even hide the **.gitignore** file from Git. Just add a line `.gitignore` to the file and Git will ignore the **.gitignore** file. However, in such a situation, the file will only reside in the local copy of the repository.

Nowadays, many **.gitignore** templates are available online, depending on the framework you're working on, such as Rails[10]. You may want to browse through this huge collection[11] of **.gitignore** files on GitHub. These **.gitignore** templates serve as handy starting points for new projects.

Set Up Your .gitignore Early

Beginners often have a tendency to add a **.gitignore** file at the late stages of a project. However, if a file is already committed and you add it to the `.gitignore` file, it will continue to be committed in your repository and tracked by Git. The only way out in this case is to explicitly untrack the file in Git—after which Git will ignore the file. We'll discuss how to untrack a tracked file in Git in a later chapter.

Remote Repositories

As we've seen so far, you can use Git on your local machine to manage versions of your work. However, because Git is a distributed version control system, many copies of the same repository can exist. So rather than just keep your repository

[10] https://github.com/github/gitignore/blob/master/Rails.gitignore
[11] https://github.com/github/gitignore/

locally, it's common to store another copy in a centralized location on a centralized server (or in the cloud).

This also enables you to work in a team, as others can access the repository from the centralized copy. Any such copy of your repository can be linked to your repository to enable synchronization. Such an external copy is called a **remote**. A remote is simply a copy of your repository. It can be on a remote server, on a peer's system or even on a different location within your local system. Interestingly, if you have access to your co-worker's repository (through SSH for instance), even that can be added as a remote.

For demonstration purposes, let's create such a copy on GitHub.

GitHub Isn't the Only Option

GitHub is not the only option for setting up a remote. A `remote` may also be on your own server. However, using cloud services like GitHub offers benefits like eliminating the need to run a separate server. You could also create remotes on GitLab or Bitbucket.

To set up a remote repository on GitHub, you first need to create an account on GitHub, or log into GitHub with your credentials if you already have an account. After login, click on the + arrow on the top right and select **New repository** to create a new repository in the cloud, shown in Figure 2.13.

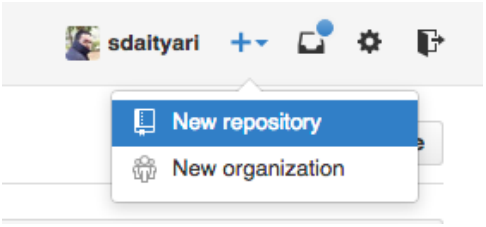


Figure 2.13. Create a new repository on GitHub

Choose a name for your repository. If you've chosen a paid or student account (see tip below), you can also choose whether to display your repository publicly or to keep it private.

Once the repository has been created, we have three options: create a new repository from the command line and push to GitHub; push the code from an existing repos-

itory from the command line; or import code from another GitHub repository. We'll take the second option here.

GitHub Offers Student Pricing

As of June 2015, GitHub doesn't provide free private repositories. Any repository you add is public if you are on the free plan. Micro plans start at $5 per month. However, if you're a student, you can apply for the GitHub Student Developer Pack[12] to get a free GitHub micro account, in addition to a lot of other services—which lasts as long as you are a student.

Returning to your local repository, run the following command to synchronize it with the remote repository:

```
git remote add origin https://github.com/sdaityari/my_git_project.git
git push -u origin master
```

The `push` command sends the commits from your local repository to the cloud repository. The `-u` option stands for "upstream". It links your repository to an upstream repository for future reference. When you add commits later, Git will show the status of your local copy in relation to the upstream repository. The `master` here signifies the files we want to synchronize.

Conclusion

What Have You Learned?

In this chapter, we've covered the basics of Git:

- the various ways to install Git on your system
- the three basic operations of track, stage, and commit
- the Git workflow of initialization, tracking, committing and pushing a repository
- starting a Git project from scratch
- the history of a repository

[12] https://education.github.com/pack

- the use of `.gitignore`
- setting up a remote on GitHub and pushing your code to the cloud.

What's Next?

In the next chapter, we'll explore a few more Git commands, focusing on the use of branches in Git.

You have encountered quite a few new things in this chapter, especially if you are new to version control. I think you may want to call it a day. Get a coffee and enjoy a well deserved break!

Chapter 3

Branching in Git

In Chapter 1, I talked about my one-time fear of trying out new things in a project. What if I tried something ambitious and it broke everything that was working earlier? This problem is solved by the use of branches in Git.

What Are Branches?

Creating a new **branch** in a project essentially means creating a new copy of that project. You can experiment with this copy without affecting the original. So if the experiment fails, you can just abandon it and return to the original—the `master` branch.

But if the experiment is successful, Git makes it easy to incorporate the experimental elements into the `master`. And if, at a later stage, you change your mind, you can easily revert back to the state of the project before this merger.

So a branch in Git is an independent path of development. You can create new commits in a branch while not affecting other branches. This ease of working with branches is one of the best features of Git. (Although other version control options like CVS had this branching option, the experience of merging branches on CVS[1]

[1] https://en.wikipedia.org/wiki/Concurrent_Versions_System

was a very tedious one. If you've had experience with branches in other version control systems, be assured that working with branches in Git is quite different.)

In Git, you find yourself in the `master` branch by default. The name “master” doesn't imply that it's superior in any way. It's just the convention to call it that.

Branch Conventions

Although you're free to use a different branch as your base branch in Git, people usually expect to find the latest, up-to-date code on a particular project in the `master` branch.

You might argue that, with the ability to go back to any commit, there's no need for branches. However, imagine a situation where you need to show your work to your superior, while also working on a new, cool feature which is not a part of your completed work. As branching is used to separate different ideas, it makes the code in your repository easy to understand. Further, branching enables you to keep only the important commits in the `master` branch or the main branch.

Yet another use of branches is that they give you the ability to work on multiple things at the same time, without them interfering with each other. Let's say you submit `feature 1` for review, but your supervisor needs some time before reviewing it. Meanwhile, you need to work on `feature 2`. In this scenario, branches come into play. If you work on your new idea on a separate branch, you can always switch back to your earlier branch to return the repository to its previous state, which does not contain any code related to your idea.

Let's now start working with branches in Git. To see the list of branches and the current branch you're working on, run the following command:

```
git branch
```

If you have cloned your repository or set a remote, you can see the remote branches too. Just postfix `-a` to the command above:

```
git branch -a
```

Figure 3.1. Command showing the branches the in local copy as well as the origin branch

As shown in Figure 3.1, the branches that colored red signify that they are on a remote. In our case, we can see the various branches that are present in the `origin` remote.

Create a Branch

There are various ways of creating a branch in Git. To create a new branch and stay in your current branch, run the following:

```
git branch test_branch
```

Here, `test_branch` is the name of the created branch. However, on running `git branch`, it seems that the active branch is still the `master` branch. To change the active branch, we can run the `checkout` command (shown in Figure 3.2):

```
git checkout test_branch
```

Figure 3.2. Creating a new branch and making it active

You can also combine the two commands above and thereby create and checkout to a new branch in a single command by postfixing `-b` to the `checkout` command:

```
git checkout -b new_test_branch
```

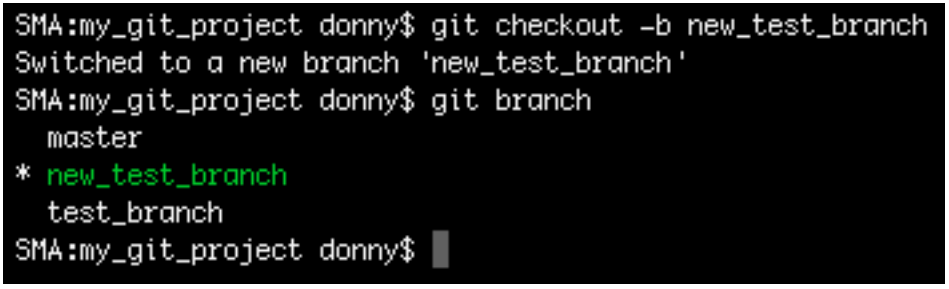


Figure 3.3. Create and checkout to a new branch in a single command

The branches we've just created are based on the latest commit of the current active branch—which in our case is `master`. If you want to create a branch (say `old_commit_branch`) based on a certain commit—such as `cafb55d`—you can run the following command:

```
git checkout -b old_commit_branch cafb55d
```

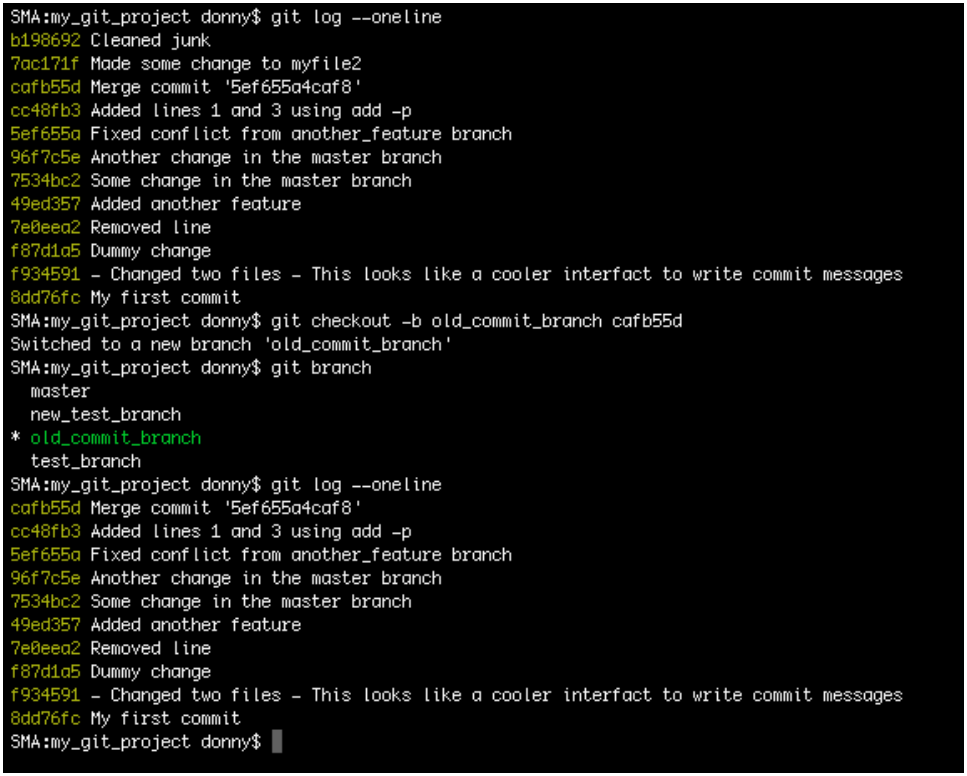


Figure 3.4. Creating a branch based on an old commit

To rename the current branch to `renamed_branch`, run the following command:

```
git branch -m renamed_branch
```

Delete a Branch

To delete a branch, run the following command:

```
git branch -D new_test_branch
```

```
SMA:my_git_project donny$ git branch
  master
  new_test_branch
* old_commit_branch
  test_branch
SMA:my_git_project donny$ git branch -D new_test_branch
Deleted branch new_test_branch (was b198692).
SMA:my_git_project donny$ git branch
  master
* old_commit_branch
  test_branch
SMA:my_git_project donny$
```

Figure 3.5. Deleting a branch in Git

Don't Delete Branches Unless You Have To

As there's not really any downside to keeping branches, as a precaution I'd suggest not deleting them unless the number of branches in the repository becomes too large to be manageable.

The `-D` option used above deletes a branch even if it hasn't been synchronized with a remote branch. This means that if you have commits in your current branch that have not been pushed yet, `-D` will still delete your branch without providing any warning. To ensure you don't lose data, you can postfix `-d` as an alternative to `-D`. `-d` only deletes a branch if it has been synchronized with a remote branch. Since our branches haven't been synced yet, let's see what happens if we postfix `-d`, shown in Figure 3.6:

```
SMA:my_git_project donny$ git branch
  master
* old_commit_branch
  test_branch
SMA:my_git_project donny$ git branch -d test_branch
error: The branch 'test_branch' is not fully merged.
If you are sure you want to delete it, run 'git branch -D test_branch'.
SMA:my_git_project donny$
```

Figure 3.6. Deleting a branch in Git using the `-d` option

As you can see, Git gives you a warning and aborts the operation, as the data hasn't been merged with a branch yet.

Branches and HEAD

Now that we've had a chance to experiment with the basics of branching, let's spend a little time discussing how branches work in Git, and also introduce an important concept: `HEAD`.

As mentioned above, a branch is just a link between different commits, or a pathway through the commits. An important thing to note is that, while working with branches, the `HEAD` of a branch points to the latest commit in the branch. I'll refer to `HEAD` a lot in upcoming chapters. In Git, the `HEAD` points to the latest commit in a branch. In other words, it refers to the tip of a branch.

A branch is essentially a pointer to a commit, which has a parent commit, a grand-parent commit, and so on. This chain of commits forms the pathway I mentioned above. How, then, do you link a branch and `HEAD`? Well, `HEAD` and the tip of the current branch point to the same commit. Let's look at a diagram to illustrate this idea (Figure 3.7):

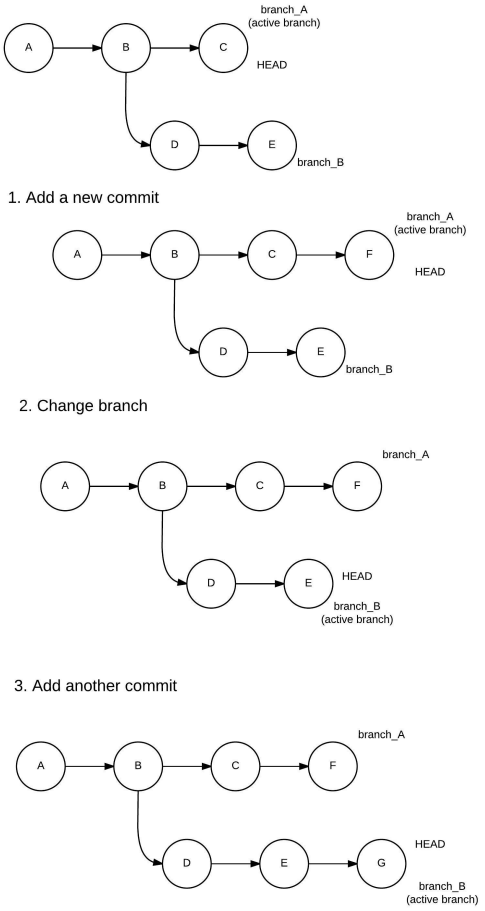


Figure 3.7. Branches and HEAD

As shown in Figure 3.7, `branch_A` initially is the active branch and `HEAD` points to commit C. Commit A is the base commit and doesn't have any parent commit, so the commits in `branch_A` in reverse chronological order (which also forms the pathway I've talked about) are C → B → A. The commits in `branch_B` are E → D → B → A. The `HEAD` points to the latest commit of the active `branch_A`, which is commit C. When we add a commit, it's added to the active branch. After the commit, `branch_A` points to F, and the branch follows F → C → B → A, whereas `branch_B` remains the same. `HEAD` now points to commit F. Similarly, the changes when we add yet another commit are demonstrated in the figure.

Advanced Branching: Merging Branches

As mentioned earlier, one of Git's biggest advantages is that merging branches is especially easy. Let's now look at how it's done.

We'll create two new branches—`new_feature` and `another_feature`—and add a few dummy commits. Checking the history in each branch shows us that the branch `another_feature` is ahead by one commit, as shown in Figure 3.8:

```
SMA:my_git_project donny$ git checkout another_feature
Branch another_feature set up to track remote branch another_feature from origin.
Switched to a new branch 'another_feature'
SMA:my_git_project donny$ git log --oneline
49ed357 Added another feature
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a cooler interfact to write commit messages
8dd76fc My first commit
SMA:my_git_project donny$ git checkout new_feature
Switched to branch 'new_feature'
Your branch is up-to-date with 'origin/new_feature'.
SMA:my_git_project donny$ git log --oneline
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a cooler interfact to write commit messages
8dd76fc My first commit
SMA:my_git_project donny$
```

Figure 3.8. Checking the history in each branch

This situation can be visualized as shown in Figure 3.9. Each circle represents a commit, and the branch name points to its `HEAD` (the tip of the branch).

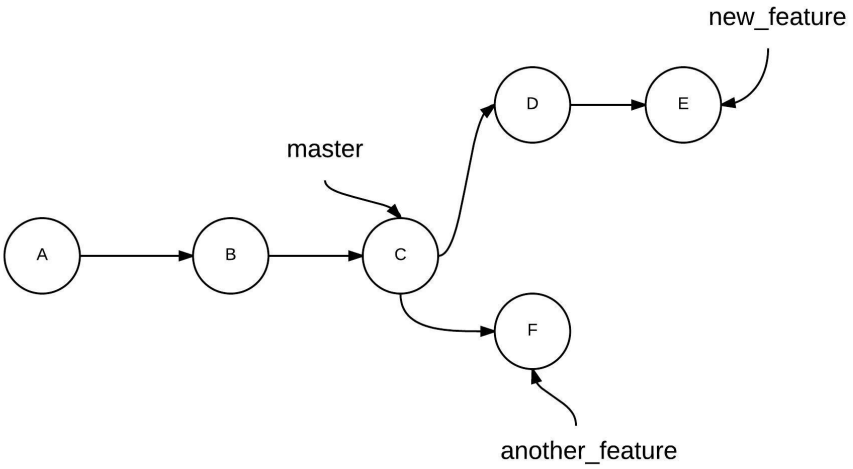


Figure 3.9. Visualizing our branches before the merge

To merge `new_feature` with `master`, run the following (after first making sure the `master` branch is active):

```
git checkout master
git merge new_feature
```

The result can be visualized as shown in Figure 3.10:

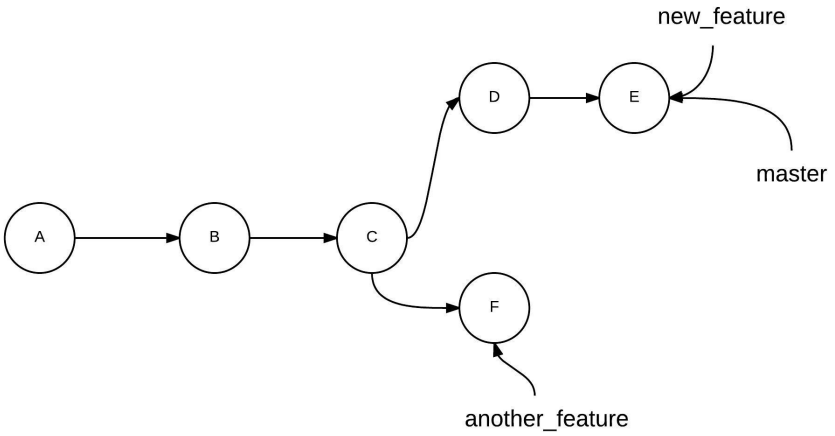


Figure 3.10. The status of the repository after merging `new_feature` into `master`

To merge `another_feature` with `new_feature`, just run the following (making sure that the branch `new_feature` is active):

```
git checkout new_feature
git merge another_feature
```

The result can be visualized as shown in Figure 3.11:

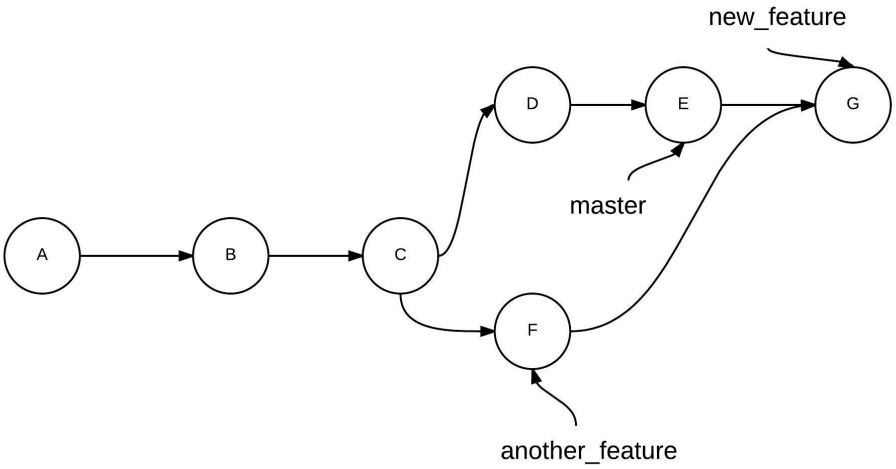


Figure 3.11. The status of the repository after merging `another_feature` into `new_feature`

Watch Out for Loops

The diagram above shows that this merge has created a loop in your project history across the two commits, where the workflows diverged and converged, respectively. While working individually or in small teams, such loops might not be an issue. However, in a larger team—where there might have been a lot of commits since the time you diverged from the main branch—such large loops make it difficult to navigate the history and understand the changes. We'll explore a way of merging branches without creating loops using the `rebase` command in Chapter 6.

```
SMA:my_git_project donny$ git merge another_feature
Updating 7e0eea2..49ed357
Fast-forward
 my_file | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
SMA:my_git_project donny$ git log --oneline
49ed357 Added another feature
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a cooler interfact to write commit messages
8dd76fc My first commit
SMA:my_git_project donny$
```

Figure 3.12. The status of branch `new_feature` after the merge

This merge happened without any "conflicts". The simple reason for that is that no new commits had been added to branch `new_feature` as compared to the branch `another_feature`. **Conflicts** in Git happen when the same file has been modified in non-common commits in both branches. Git raises a conflict to make sure you don't lose any data.

We'll discuss conflicts in detail in the next chapter. I mentioned earlier that branches can be visualized by just a simple pathway through commits. When we merge branches and there are no conflicts, such as above, only the branch pathway is changed and the `HEAD` of the branch is updated. This is called the **fast forward** type of merge.

The alternate way of merging branches is the **no fast forward** merge, by postfixing `--no-ff` to the `merge` command. In this way, a new commit is created on the base branch with the changes from the other branch. You are also asked to specify a commit message:

```
git merge --no-ff new_feature
```

In the example above, the former (merging `new_feature` with `master`) was a fast forward merge, whereas the latter was a no fast forward merge with a merge commit.

While the fast forward style of merges is default, it's generally a good idea to go for the no fast forward method for merges into the master branch. In the long run, a new commit that identifies a new feature merge might be beneficial, as it logically separates the part of the code that is responsible for the new feature into a commit.

Conclusion

What Have You Learned?

In this chapter, we discussed what branches are and how to manage them in Git. We looked at creating, modifying, deleting and merging branches.

What's Next?

I've already spoken about how Git is beneficial to developers working in teams. The next chapter will look at this in more detail, as well as specific Git actions and commands that are frequently used while working in a distributed team.

Chapter 4

Using Git in a Team

So far, we've looked at managing source code by starting a Git project, working with branches, and pushing code to a remote repository. In this chapter, we'll focus on the features of Git that help you contribute in a team.

We've seen how useful Git's version control tools can be for a sole coder. Git's power is even more evident when it comes to managing a project with many contributors. It enables members of a team to work independently on a project and stay in sync—even when they're located far apart from each other.

Getting Started in a Team: Cloning from a Remote

Earlier, we performed a push operation to GitHub, sending a copy of our local repository to the cloud. This is the process you follow when the repository has been created on your local system.

However, if you're working on a team, it's possible that some work has already been done on the repository when you join. In this scenario, you need to grab a copy of the code from a central repository and work on it. The process of grabbing this re-

pository is called **cloning**. Cloning is the process of creating a copy of a remote repository. The copy (or clone) that you create has its own project history, and any work done on it is independent of the development on the remote.

The Source is the `origin`

If you clone a repository, the source from which you cloned it from is designated as the `origin` remote by default. You may modify the remote using the `git remote` command.

Think of cloning as creating photocopies of a document. If you overwrite something in the photocopy, the original document remains untouched. Similarly, if you change the original document after making the photocopy, the photocopy retains the contents of the original document. Until you merge the clone with the original remote, they are separate entities.

To clone a remote repository, you need to know its location. This location usually takes the form of a URL. In GitHub, you can find the URL of a project on the bottom right corner of the home page of that project. Let's look at an example of a repository on my own GitHub account, as shown in Figure 4.1:

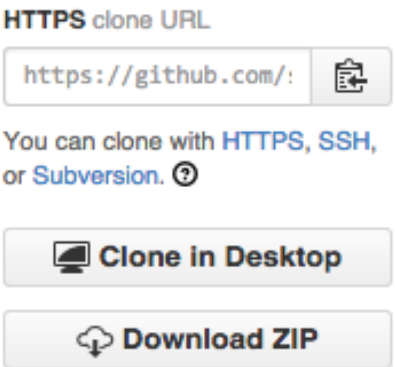


Figure 4.1. GitHub showing the location of the clone URL

To clone this project, we need to run the following command:

```
git clone https://github.com/sdaityari/my_git_project.git
```

When the repository is successfully cloned, a local directory is created with the same name as the project name (in our case, `my_git_project`), and all the files under the repository are present in that directory. It's not necessary to keep the directory name; you can change it any time. If you want to change the root directory name of the repository while cloning it—let's say to **my_project**—you'll need to provide the name to the clone command:

```
git clone https://github.com/sdaityari/my_git_project.git my_project
```

You may also rename the directory after you've cloned the repository.

Once you've cloned the repository, you can verify that the `origin` remote points to the URL that you just cloned from, shown in Figure 4.2:

```
git remote -v
```

Figure 4.2. Verifying the origin remote

The `-v` option is short for `--verbose` and tells Git to display the URLs of the remotes next to the names.

Optional: Different Protocols While Cloning

In the command we used to clone the repository, you may have noticed that the URL starts with `https`. You have the option of choosing a different protocol. The available protocols for any Git remote are as follows:

- Local protocol
- Git protocol

- HTTP/HTTPS protocol
- SSH protocol

The local protocol involves cloning in the same system. For instance, you may clone a repository like so:

```
git clone /Users/donny/my_git_project
```

The biggest disadvantage is the access this protocol provides, which is limited to the local computer.

If you clone over the Git protocol, your URL starts with `git` instead of `https`: `git://github.com/sdaityari/my_git_project.git`. This doesn't provide any security. You only get read-only access over the `git` protocol, and therefore you can't push changes.

With the `https` protocol, your connection is encrypted. GitHub allows you to clone or pull code anonymously over `https` if the repository is public. However, for pushing any code, your username and password are verified first. GitHub recommends using `https` over `ssh`, because the `https` option always works, even if you're behind a firewall or a proxy.

If you're using the `https` protocol, you need to type in your credentials every time you push code. However, if you push your code frequently, you can make Git remember your credentials for a given amount of time after you successfully enter them once. This is done with the `credential.helper` setting. Run the following to enable credential storage:

```
git config --global credential.helper cache
```

By default, Git stores your credentials for 15 minutes. You may also set the timeout limit in seconds:

```
git config --global credential.helper "cache --timeout=3600"
```

This command makes Git store your credentials for an hour.

Alternative Credential Storage

An alternative but less secure way of saving the username and password indefinitely would be to store them within the remote path itself. In such a case, your remote would look like this: `https://sdaityari:password@github.com/sdaityari/my_git_project.git`.

The `ssh` protocol, on the other hand, authenticates your requests using public key authentication[1]. You establish a connection with the remote server over `ssh` first, and then you request the resource. To set up authentication using `ssh`, you need to generate your public/private key pair.

In Linux or OS X, the following command generates a key pair:

```
ssh-keygen -t rsa -C "sdaityari@gmail.com"
```

In Windows, you need either PuTTY or Git Bash to generate the key. GitHub provides detailed instructions on the process of generating the key pair on Windows[2].

GitHub Desktop Can Generate Keys for You

If you use the GitHub desktop client, the process of generating a key pair and linking it with your GitHub account is done automatically by the client. We'll review clients in a later chapter.

Your public key is stored in the file **~/.ssh/id_rsa.pub**. You can view it using the `cat` command, shown in Figure 4.3:

[1] If you're interested in learning how the public key authentication works, you may check out this video on public key encryption [https://www.comodo.com/resources/small-business/digital-certificates2.php].
[2] https://help.github.com/articles/generating-ssh-keys/#platform-windows

```
cat ~/.ssh/id_rsa.pub
```

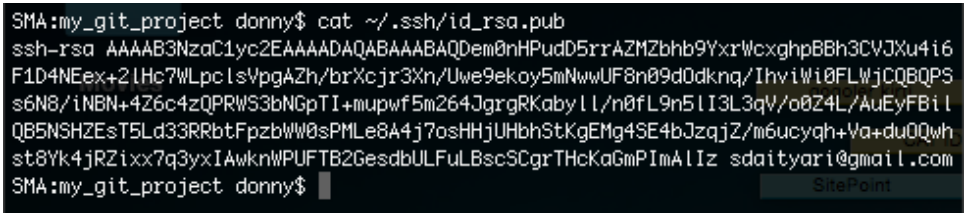


Figure 4.3. Viewing the contents of the public key

The `cat` command prints the contents of a file on the terminal. ~ stands for the home directory of the current active user. For instance, if your username is `donny`, ~ points to `/Users/donny/` on OS X and `/home/donny` on Linux.

You need to add the contents of the public key to your GitHub SSH settings[3] in order to establish `ssh` connections to GitHub, as shown in Figure 4.4:

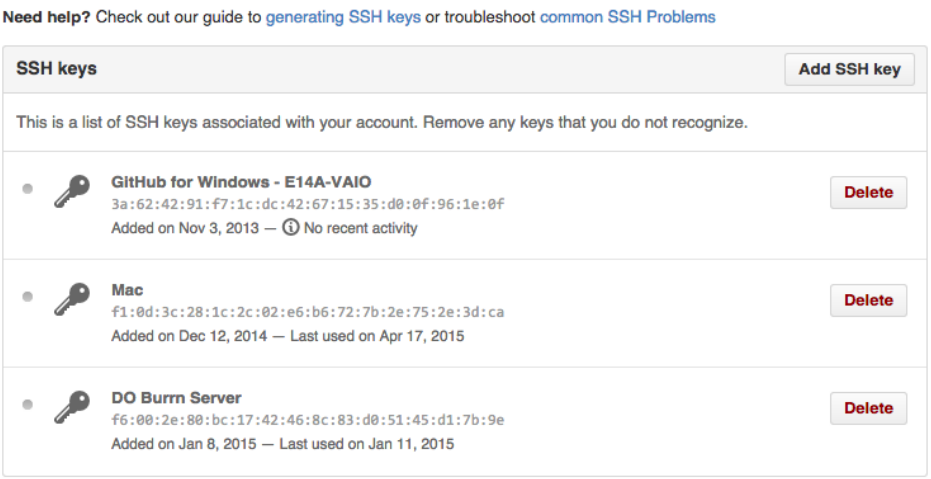


Figure 4.4. SSH Keys on a GitHub profile

[3] https://github.com/settings/ssh

Contributing to the Remote: Git Push Revisited

Earlier in this book, we created a repository in the cloud and pushed our local code to it. Once you've made changes to a repository, they need to be pushed to the remote if the central repository is to reflect them. `git push` is a simple command that does the trick:

```
git push
```

We'll now explore `push` a little further. There are various ways to push code to a remote.

A `git push` simply pushes the code in the current branch to the `origin` remote branch of the same name. A branch is created if the branch with the same name as the current local branch doesn't exist on the `origin`:

```
git push remote_name
```

This command pushes the code in the current branch to the `remote_name` remote branch. A branch is created on the remote if the branch with the same name as the current local branch doesn't exist on the `remote_name` remote.

```
git push remote_name branch_name
```

This command pushes the code on the `branch_name` branch (irrespective of your current branch) to the remote branch of the same name. If `branch_name` doesn't exist on the remote, it is created. If `branch_name` doesn't exist on the local repository, an error is shown.

```
git push remote_name local_branch:remote_branch
```

This command pushes the `local_branch` from the local repository to the `remote_branch` of the remote repository. Although it involves typing a longer command, I would always advise that you use this syntax for pushing your code, as it avoids mistakes.

Figure 4.5 gives a rough idea of how the states of the `master` and `origin/master` look before and after a `push` operation:

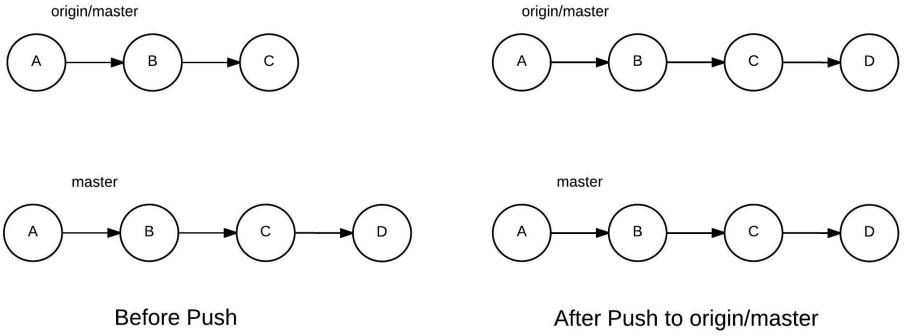


Figure 4.5. The status of a remote after a push operation

You Can Delete Branches Using `git push`

You can modify the syntax listed above to delete a branch on the remote:

```
git push remote_name :remote_branch
```

In this command, you are essentially sending an empty branch to the `remote_branch` branch of `remote_name`, which empties the `remote_branch`, or in other words, deletes it on the remote. You should therefore be careful while attempting this operation.

Keeping Yourself Updated with the Remote: Git Pull

Now that we've looked at how to push the changes to the remote, let's explore the situation where others are working on the same project and you need to update your local repository with the changes other contributors have made.

The ideal way to update your local repository with the commits others have made to the remote is, firstly, by downloading the new data, and then by merging it with the appropriate branches.

To download the changes that have appeared in the remote, we run the following command:

```
git fetch remote_name
```

This updates our local branches from the remote `remote_name`. (We can skip the name of the remote by running just `git fetch`, and the command will update the branches of the local repository from the remote `origin`.)

When you clone a repository or set an upstream, local versions of their branches are also maintained. The `fetch` command updates these local versions with the latest commits from the remote.

Following a `fetch`, to update your local branch you need to merge it with the appropriate branch from the remote. For instance, if you're planning to update the local master branch with the remote's master branch, run the following command:

```
git merge origin/master
```

This is basically merging the branch `origin/master` with your current active branch. Following the `fetch`, your `origin/master` is updated with the latest commits of the branch on the remote. You have therefore succeeded in updating a local branch with the latest commits from a remote branch.

To understand what's going on, let's explore further with the help of a diagram (Figure 4.6):

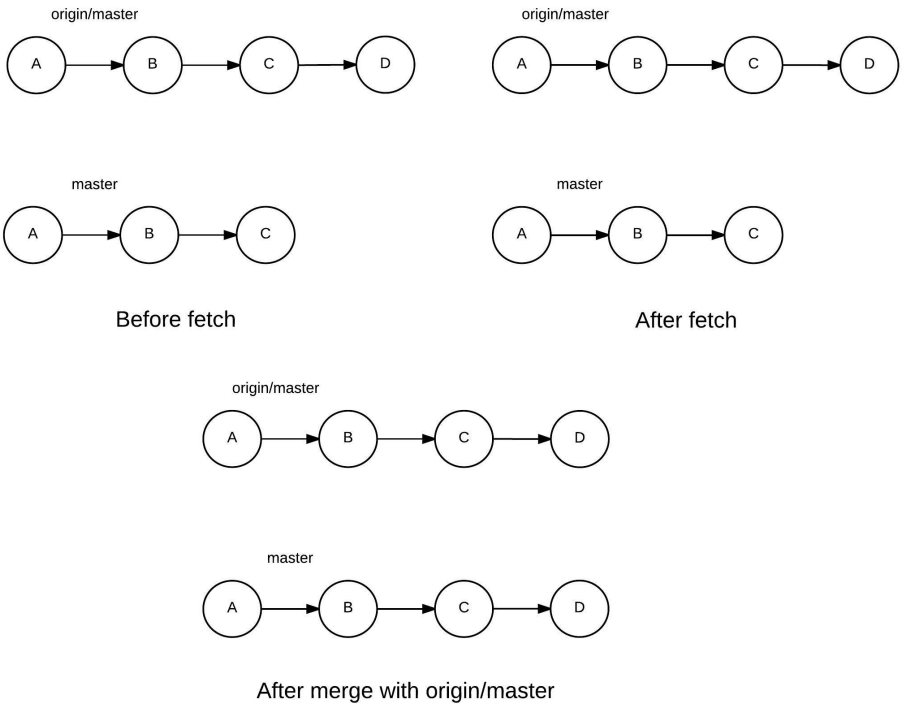


Figure 4.6. Status of the repositories before and after the fetch/merge process

Alternatively, a shorter way of updating the local branch by downloading and merging a remote branch is by using `pull`. The `git pull` command is essentially a `git fetch` followed by a `git merge`. To update the current active branch through `pull`, run the following:

```
git pull origin master
```

Pulls Are Fast Forward by Default

Just as with merging, you can specify whether or not a pull should be a fast-forward. It *is* by default, but this can be overridden with the `--no-ff` postfix.

As with `git push`, it's possible to specify different local and remote branches for `git pull` too:

```
git pull
```

A `git pull` simply downloads the code from the `master` branch of the `origin` remote branch. It then merges the code with the current active branch.

```
git pull remote_name
```

The command above first downloads the code from the `master` branch of the `remote_name` remote branch. It then merges the code with the current active branch.

```
git pull remote_name branch_name
```

The command above first downloads the code from the `branch_name` branch of the `remote_name` remote branch. It then merges the code with the current active branch.

```
git pull remote_name local_branch:remote_branch
```

This command first downloads the code from the `remote_branch` branch of the `remote_name` remote branch. It then merges the code with the `local_branch` in the local repository.

To help visualize the process of a `git pull`, the following diagram shows the status of the local repository before and after a pull (Figure 4.7):

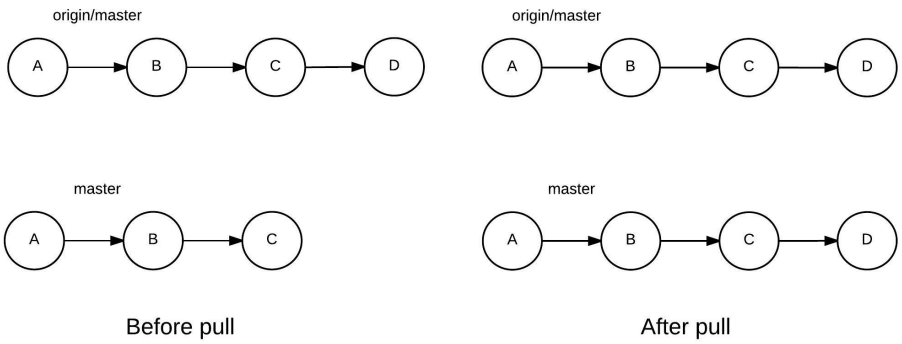


Figure 4.7. Illustration of the status of a local repository before and after a pull

Here Be Conflicts!

A `fetch-merge` or `pull` may result in conflicts, in which case you will need to resolve the conflicts before completing the `merge` or `pull`. We'll discuss conflicts later in this chapter.

Dealing With a Rejected Git Push

Now that you have the knowledge of both sending and receiving updates in your local repository, let's look at a special situation. It involves pushing new code to a remote branch that's been updated since your last synchronization. In this case, your push would be rejected—with the message that "it is `non-fast-forward`". This simply means that, since changes were made to both the remote and your local copy, Git is not able to determine how to merge them.

In such a situation, you last synced the `master` branch from `origin` (hence referred to as `origin/master`) when it was at commit `B` (as named in the diagram below). You've proceeded with two commits, `D` and `E`. Since your last sync, a new commit `C` has been added to `origin/master`. Git doesn't merge both these workflows, as they've taken different pathways. Therefore, you should first pull from `origin/master` and merge it with `master`, resolving any conflicts that appear. This would make commit `C` appear in your `master` branch. Git will then be able to accept the push.

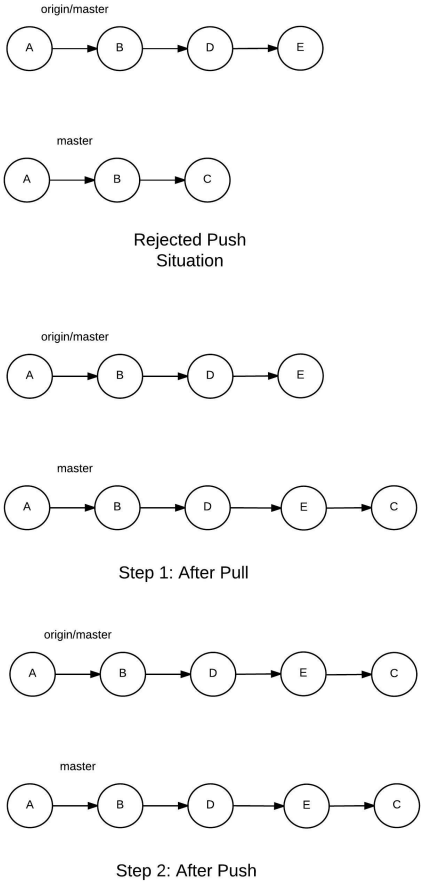


Figure 4.8. Example of a situation where a push is rejected

Rebase?

In this example, we demonstrate a `pull --rebase` in Figure 4.8 rather than just a pull. For now, just ignore this, as I'll explain `rebase` in Chapter 6.

Conflicts

Let's now address conflicts—the topic perhaps most dreaded by people working with Git.

Conflicts can occur when you're trying to merge two branches or to perform a pull. However, as a pull operation essentially involves merging, we'll address conflicts only during a merge. If you encounter a conflict during a pull, the process of resolving it remains the same.

A **conflict** arises when your current branch and the branch to be merged have diverged, and there are commits in your current branch that aren't present in the other branch, and vice versa. Git isn't able to determine which changes to keep, so it raises a conflict to ask the user to review the changes. The last common commit between the two branches—which is also the point where they diverged—is called the **base commit**.

When Git merges the two branches, it looks at the changes in each branch since the base commit. When there are unambiguous differences—like changes to different files, and sometimes different parts of the same file—the changes are applied. However, if there are changes to the same parts of the same file, and Git can't determine which changes to keep, it raises a conflict.

To understand conflicts properly, let's try to create an example conflict ourselves. We'll create a reference branch named `base_branch`. Let's also create a sample program in Python—**sample.py**—the contents of which are shown below:

```
CONSTANT = 5

def add_constant(number):
    return CONSTANT + number
```

It's a simple program that adds a constant to a provided number. Now imagine a scenario where you make a branch, `conflict_branch`, where you change the value of `CONSTANT` to 7. And suppose a friend has worked on the same line numbers of the same file on the branch `friend_branch`, and changed the `CONSTANT` to 9. We can visualize this with Figure 4.9:

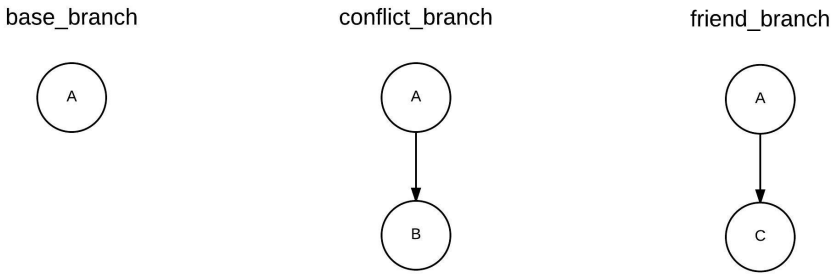


Figure 4.9. A situation where a merge raises a conflict

Now, let's see what happens when we try to merge the `friend_branch` with our `conflict_branch`:

```
git merge friend_branch
```

Git shows a message that the automatic merge failed, and that there are conflicts in **sample.py** that need to be resolved (Figure 4.10):

```
SMA:my_git_project donny$ git merge friend_branch
Auto-merging sample.py
CONFLICT (content): Merge conflict in sample.py
Automatic merge failed; fix conflicts and then commit the result.
SMA:my_git_project donny$
```

Figure 4.10. Failed merge due to conflicts

That doesn't sound so great! Let's do a `git status` to see what's wrong (Figure 4.11):

```
SMA:my_git_project donny$ git status
On branch conflict_branch
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   sample.py

no changes added to commit (use "git add" and/or "git commit -a")
SMA:my_git_project donny$
```

Figure 4.11. Status during a failed merge

Git shows that both files have been modified, and that we need to make a commit after fixing the conflicts. Naturally, this isn't a fast-forward commit, as Git has failed to automatically resolve the merge. A new commit will be created once you fix the conflicts and commit your changes.

Note that a conflict arises only when Git is unable to determine which lines to keep. To make sure no data is lost, you're asked which lines should be kept. Figure 4.12 shows the contents of the file in Sublime Text:

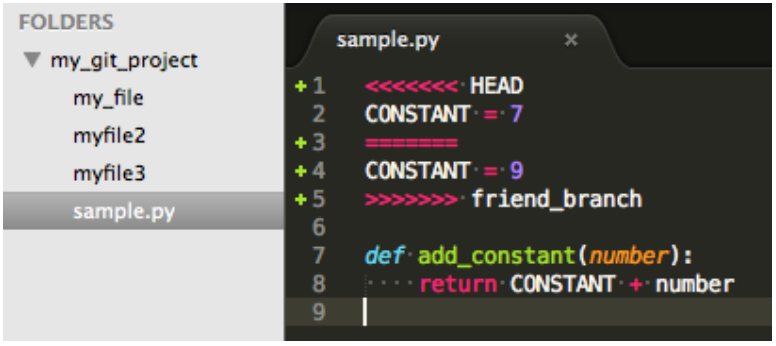


Figure 4.12. Contents of conflict file

Look at the contents of the file now. Since you initiated the merge, Git has modified the file to show you the changes in the two versions of the same file:

```
<<<<<<< HEAD
CONSTANT = 7
=======
CONSTANT = 9
>>>>>>> friend_branch

def add_constant(number):
    return CONSTANT + number
```

The lines between `<<<<<<< HEAD` and `=======` contain your version of the part of the file, whereas the lines between `=======` and `>>>>>>> friend_branch` contain the part of the file that is present in the `friend_branch`. You should review these lines and decide which lines to keep. You may need to take up the issue with your team before you decide which version to keep. In our case, let's keep the change we made.

Multiple Conflicts

In our simple example, there was just one conflict in a single file. If there are conflicts in multiple files, they'll appear when you run `git status`. You need to edit them individually to check which version to keep. If there are multiple conflicts in the same file, you should search for the word `HEAD` or `<<<<<` (multiple "less than" signs together are rarely used in your source code) to find out the instances within a file where conflicts have arisen, and then work on them individually.

After you've resolved the conflicts, you should stage the changed files for commit. In our case, there's only a single file:

```
git add sample.py
```

You should then proceed to making a commit, as shown in the line of code below and in Figure 4.13:

```
git commit -m "Concluded merge with friend_branch"
```

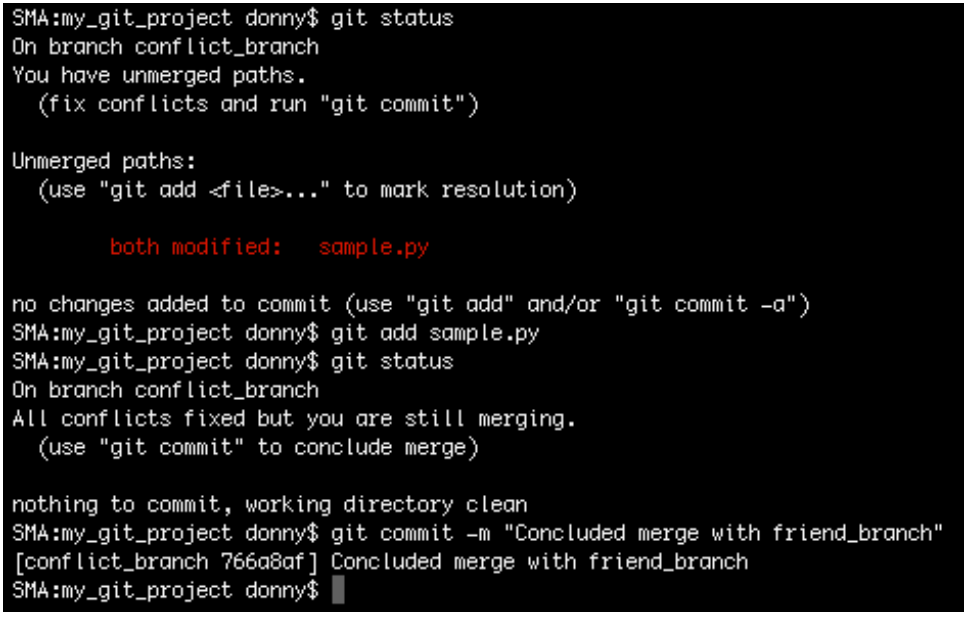


Figure 4.13. Successful commit after resolving conflicts

Aborting a Merge with Conflicts

After initiating a merge that's resulted in conflicts, if you're overwhelmed and want to go back to the pre-merge state, you can do so by aborting the merge:

```
git merge --abort
```

```
SMA:my_git_project donny$ git merge friend_branch
Auto-merging sample.py
CONFLICT (content): Merge conflict in sample.py
Automatic merge failed; fix conflicts and then commit the result.
SMA:my_git_project donny$ git status
On branch tb
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   sample.py

no changes added to commit (use "git add" and/or "git commit -a")
SMA:my_git_project donny$ git merge --abort
SMA:my_git_project donny$ git status
On branch tb
nothing to commit, working directory clean
SMA:my_git_project donny$
```

Figure 4.14. Aborting a merge with conflicts

Git Workflows

With the knowledge of branches, merges and conflicts, I believe we're now in a position to discuss the best practices of using Git in a team.

Ideally, if you're working in an organization, you should clone the repository, but you should never change anything in your `master` branch. Any new addition—be it a bug fix or a new feature—should be started in a new branch. Once you've completed your work on the new branch, merge it with your updated `master` branch and ask your organization to pull from your branch. If your code is accepted, it will appear in your `master` branch when you pull from the main repository next time. If your code is still under review or not accepted, you can start work on a new feature by creating a new branch from the `master` branch.

Let's now look at a few workflows.

Centralized Workflow

In the centralized workflow, a centralized repository is created and every contributor has a clone of the repository. Contributors work on their own copy independently and push new commits to the centralized repository when necessary. If the push fails, the local branch is updated; conflicts, if any, are resolved; and a new push is initiated.

This workflow suits organizations migrating to Git from a centralized version control system like Subversion. The workflow remains the same, but every developer works on a local copy of the code.

Feature Branch Workflow

The feature branch workflow is an extension of the centralized workflow. However, instead of working on the `master` branch, the development of each new feature or bug fix is initiated in a new branch.

Essentially, you should avoid committing directly to your `master` branch, but only keep it updated with the central repository. You work on your feature branch, and push the feature branch to the central repository once you complete your work. If your feature is accepted to the `master` branch of the central repository, this will be reflected in your local `master` once you pull changes from `origin/master`.

Forking and Pull Requests: The Open-source Workflow

The next workflow is followed generally in open-source projects. For projects that use code sharing websites like GitHub, there's the concept of forking. When you **fork** a project, you're creating your own copy of the repository on the cloud. This is required for two reasons: it's difficult for the organization to pull directly from your local machine, and it's not practical to give write access for their main repository to every would-be contributor. A fork is a personal copy of a repository on a code sharing website like GitHub, BitBucket or GitLab. You have full write access to your fork, although you may not have write access to the main repository which was the source of the fork.

Although open-source organizations follow this workflow strictly, every organization with a good project delivery line and team organization needs code reviews and

merges through pull requests. This open-source workflow helps organizations to performing such code reviews before merging any new code into their repository.

Once you've created a fork and cloned it to your local machine, you can experiment with it as you please. You can create branches, push them to your fork, and submit pull requests to the organization that maintains the original repository. If the organization chooses to merge your changes, those changes will become a part of the central repository.

Regarding the use of the `master` branch, ideally, the feature branch workflow idea applies here. You never make changes to the `master` branch of your fork. You pull the changes from the main repository and keep your `master` branch updated.

In the example above, your fork is assigned the remote `origin` (since you clone your local repository from the fork), and the organization's main repository is assigned the remote `upstream`. You usually pull from the `upstream` to get the latest commits, whereas you push to your `origin` before creating a pull request.

In general, the feature branch and forking workflows are supersets of the centralized workflow. You may or may not follow the feature branch workflow in the forking workflow strictly. However, the general advice is to use a combination of the forking and feature branch workflows, because the development process of each developer stays in the fork, and only the code that needs to be merged gets into the centralized repository.

Conclusion

With this, we come to the end of another fairly lengthy chapter. Let's briefly review the things that we learned.

What Have You Learned?

In this chapter, we've covered how to:

- clone from a remote repository
- create, update, merge and delete branches
- keep a local repository updated

- send the changes from a local repository to a remote
- manage conflicts during merges.

We've also looked at general workflows while working with organizations.

What's Next?

In the next chapter, we'll explore common mistakes in Git. First, we'll focus on amending errors while working with Git. Then, we'll move on to debugging in Git with two useful commands—`blame` and `bisect`.

Chapter 5

Correcting Errors While Working With Git

In the last few chapters, we've built a good foundation in Git basics. We've gone through the basic Git commands, followed by some more advanced processes that help you contribute to an organization. Up to this point, we haven't discussed how to fix mistakes you might make while working with Git.

Alexander Pope once said "To err is human"—and it's only human to commit mistakes during the Git workflow. Git makes it possible to correct mistakes at each stage of a project—which is yet another reason why it's so popular with developers.

In this chapter, we'll look first at how you can correct your own mistakes. Then we'll look at how to weed out bugs introduced at various points into repository either by you or by others.

Amending Errors in the Git Workflow

With Git, it's fairly easy to undo changes you've made. In this section, we'll look at three examples: undoing a stage operation; undoing a commit, by reverting back to an older commit; and undoing a push, by rewriting the history of a remote repository.

Undo Git Add

The `git add` command either tells Git to track an untracked file, or to stage the changes in a tracked file for a commit.

If you've just asked Git to track a new file that you've created but not yet committed—let's call it **mistake_file**—you can undo the operation by running the following command:

```
git rm --cached mistake_file
```

Here, `rm` stands for remove (just like the regular terminal command `rm`). When we postfix `--cached`, we ask Git to untrack the file, but let it remain in the file system.

Why Can't I Just Delete the File?

If we simply delete the file, Git will show that a tracked file has been deleted—a change that needs to be staged and committed to appear in the history.

You can check the status of the repository to confirm that the file is untracked again (Figure 5.1):

```
SMA:my_git_project donny$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        mistake_file

nothing added to commit but untracked files present (use "git add" to track)
SMA:my_git_project donny$ git add mistake_file
SMA:my_git_project donny$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   mistake_file

SMA:my_git_project donny$ git rm --cached mistake_file
rm 'mistake_file'
SMA:my_git_project donny$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        mistake_file

nothing added to commit but untracked files present (use "git add" to track)
SMA:my_git_project donny$
```

Figure 5.1. Undoing `git add`

The command `git rm --cached` can also be used to remove a file from the repository. Once a file has been removed, you need to commit the changes to take effect. Figure 5.2 shows this in action:

```
SMA:my_git_project donny$ git rm --cached myfile2
rm 'myfile2'
SMA:my_git_project donny$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    myfile2

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        myfile2

SMA:my_git_project donny$
```

Figure 5.2. Removing a tracked file from the Git repository

Forced Removal

If you run just `git rm` without the `--cached` option, it will lead to an error. The other option that can be postfixed with `git rm` is `-f` for forced removal. The `-f` option untracks the file and then removes it from your local system altogether. Therefore, you should be careful when you're removing tracked files if you use this option. All the same, there is way to backtrack from `rm -f` too. Even if you commit after using `rm -f` on a file, you can still get the file back by reverting to an old commit. We'll discuss the process of reset and reverting to an old commit shortly.

Let's say you make changes to a tracked file (**myfile2**), and then run `git add` to stage it for commit. Then you realize you made a mistake before committing it. You can run the following command to unstage the changes:

```
git reset HEAD myfile2
```

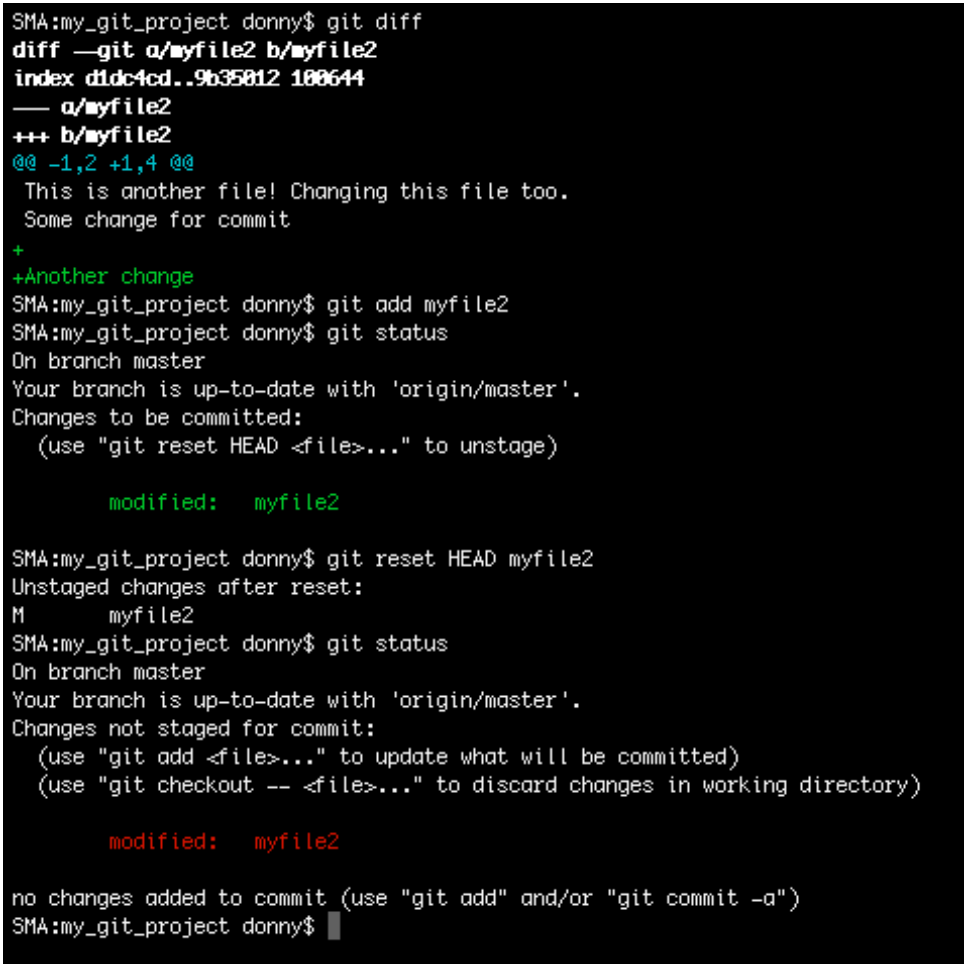


Figure 5.3. Unstaging changes

This command resets a file to the state where the `HEAD`, or the last commit, points to. This is the same as "unstaging" the changes in a file.

Once you've unstaged the changes in a file, you can undo the changes you made in the file as well, reverting it back to the state during the last commit. This is where the following command comes in:

```
git checkout myfile2
```

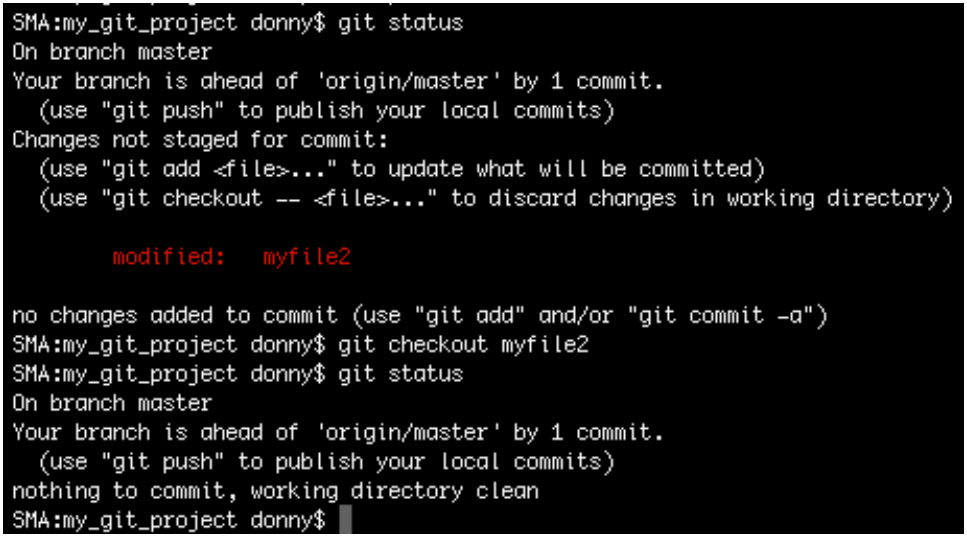


Figure 5.4. Undo changes in a tracked file

We've seen the `checkout` command used previously during the process of branching. It's also used to restore any unstaged changes in a file, as seen in Figure 5.4.

So What Does `checkout` Really Do?

Basically, `checkout` updates the file(s) in the current status of the repository to an earlier version.

When we were changing branches, `checkout` changed the status of files to a different branch. In this case, `checkout` restores the file to its version at the time of the last commit.

Undo Git Commit

If you've already committed your changes and then realize your mistake, there's a way to undo that too. Let's do an unnecessary commit and try to revert back to the original. Run the following command to see Git do some magic:

```
git reset --soft HEAD~1
```

We can see the result in Figure 5.5:

```
SMA:my_git_project donny$ git log --oneline
960fa28 Commit with error
b198692 Cleaned junk
7ac171f Made some change to myfile2
cafb55d Merge commit '5ef655a4caf8'
cc48fb3 Added lines 1 and 3 using add -p
5ef655a Fixed conflict from another_feature branch
96f7c5e Another change in the master branch
7534bc2 Some change in the master branch
49ed357 Added another feature
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files
8dd76fc My first commit
SMA:my_git_project donny$ git reset --soft HEAD~1
SMA:my_git_project donny$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   myfile2

SMA:my_git_project donny$
```

Figure 5.5. The result of undoing a Git commit

The `--soft` option undoes a commit, but lets the changes you made in that commit remain staged for you to review. The `HEAD~1` means that you want to go back one commit from where your current `HEAD` points (which is the last commit).

What's with HEAD~1?

We encountered `HEAD` earlier, and we know that it points to the last commit in the current branch. I've added ~ to `HEAD` in the example above. This refers to the parent of the last commit in the current branch. You can also use ^. Using either ~ or ^ refers to the parent of the last commit in the current branch, while ~~ and ^^ both refer to the grandparent of the last commit in the current branch. You can also add numbers to move back a specific number of commites the hierarchy. However, adding numbers after either ~ or ^ can mean different things:

- ~2 goes up two levels in the hierarchy of commits, via the first parent if a commit has more than one parent.
- ^2 refers to the second parent where a commit has more than one parent (which could be the result of a merge).

> You can also combine these postfixes. For instance, `HEAD~3^2` refers to the second parent of the great-grandparent commit, which you reached through the first parent and grandparent.

The second option here is postfixing the `--hard` option to permanently undo commits. It's generally advised that you avoid using the `--hard` option—unless you're absolutely sure you want to do away with the commits.

A third option of `reset` is `--mixed`, which is also the default option. In this option, the commit is reverted, and the changes are unstaged.

The process of committing involves three steps: making changes in a file, staging it for a commit, and performing a commit operation. The `--soft` option takes us back to just before the commit, when the changes are staged. The `--mixed` option takes us back to just before the staging of the files, where the files have just been changed. The `--hard` option takes us to a state even before you changed the files.

There's yet another Git command that could help you in case you've committed changes by mistake. This is a the `revert` command. The `reset` command changes the history of the project, but `revert` undoes the changes made by the faulty commit by creating a new commit that reverses the changes. Figure 5.6 shows the difference between revert and reset:

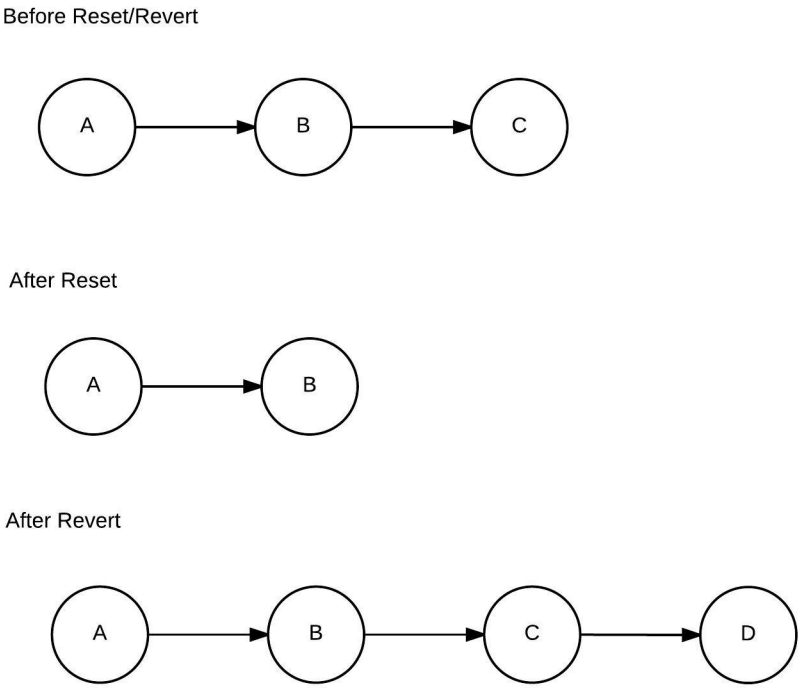


Figure 5.6. The difference between a `revert` and a `reset`

Here's how to go back one commit using `revert`:

```
git revert HEAD~1
```

It also asks you whether you want to modify the commit message for the commit that reverses the changes of the unwanted commits:

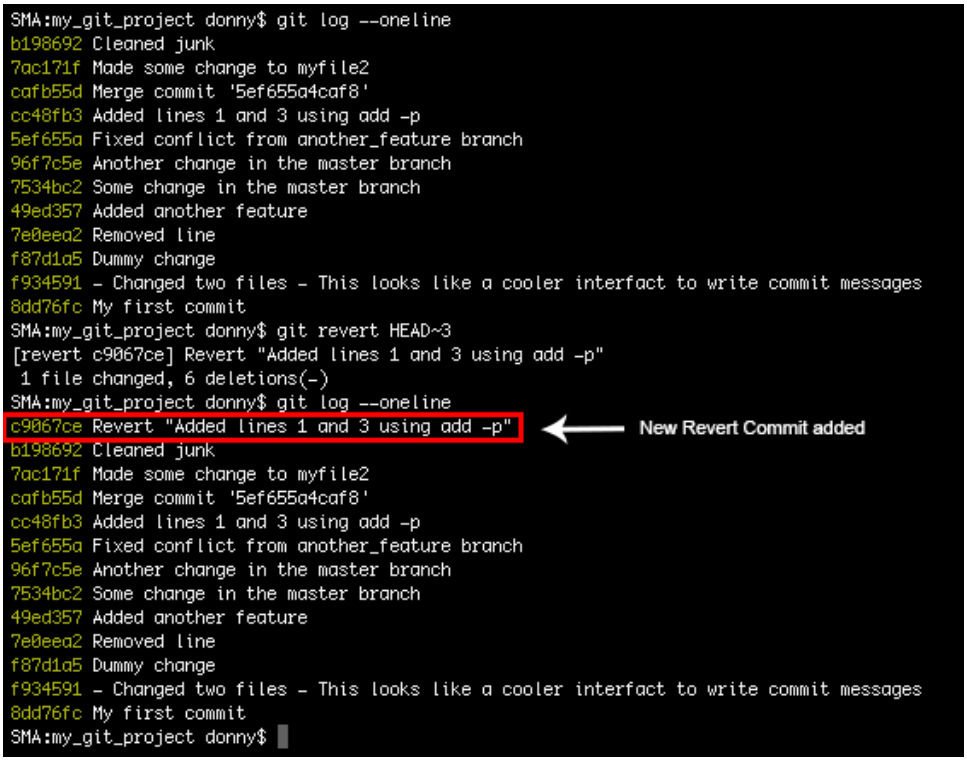


Figure 5.7. Example of `revert`

You can change the commit message of the last commit by running the following command:

```
git commit --amend -m "New Message"
```

```
SMA:my_git_project donny$ git log --oneline -3
953477d Dummy Commit
f36d753 Squashed last two commits
083e7ee Added yet another test
SMA:my_git_project donny$ git commit --amend -m "Changing Commit Message"
[master 1273598] Changing Commit Message
 Date: Wed Jul 15 16:07:27 2015 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 x
SMA:my_git_project donny$ git log --oneline -3
1273598 Changing Commit Message
f36d753 Squashed last two commits
083e7ee Added yet another test
SMA:my_git_project donny$
```

Figure 5.8. Changing a commit message

The `--amend -m` option changes the commit message of the last commit. Notice in Figure 5.8 that the hash changes too, effectively rewriting the history.

Undo Git Push

In case you've also pushed your changes to a central repository, it's possible to revert changes in the push too.

The simplest way is to go for a `revert` and push the new commit that undoes the changes:

```
git revert HEAD~1
git push origin master
```

However, if you also want the other commit(s) to vanish from the remote repository, you first need to go for a `reset` command—deleting the unwanted commit—and then push the changes to the remote. If you perform a normal `git push`, the push will be rejected—because the origin `HEAD` is at a more advanced position than your local branch. Therefore, you need to force the change with a postfix, `-f`, which forces the push on the remote `origin`:

```
git reset --hard HEAD~2
git push -f origin master
```

Use `-f` With Caution

Postfixing `-f` is a dangerous move, as it rewrites the remote without confirming it. Make sure you double check your local changes before going for an `-f` push.

Debugging Tools

The scenarios we've discussed so far help you to undo changes in Git. They've dealt with mistakes you've committed in the near past and want to correct. Now we'll look at dealing with bugs introduced by you or others in the past. This will involve exploring tools in Git that help in the process of debugging. These tools are required when you're working on a relatively large code base with a large number of contributors.

You may or may not know the location of the bug. If you know which file or set of files is the source of the bug, you can debug with `git blame`. If you don't know the source of the bug, you can debug with `git bisect`. If you've written unit tests, you can also automate the process of debugging. So let's explore the different ways of debugging your code in Git.

Git Blame

Running the `git blame` command on a file gives you detailed information about each line in the file. `git blame` lists the commits that introduced changes in a file, along with basic information about the commit, like the commit hash, author and date[1].

`git blame` is usually used when you know which file is causing a bug. Let's see how it works:

[1] Some people may feel that "blame" is a harsh way to put it. Perhaps a better name for the command would have been "attribute".

```
git blame my_file
```

```
SMA:my_git_project donny$ git blame my_file
^8dd76fc (Shaumik 2014-05-06 15:28:03 +0530  1) This is some information!
f934591c (Shaumik 2014-05-06 15:31:00 +0530  2)
cc48fb3c (Shaumik 2014-06-11 22:38:21 +0530  3) Adding Line 1.
cc48fb3c (Shaumik 2014-06-11 22:38:21 +0530  4)
f934591c (Shaumik 2014-05-06 15:31:00 +0530  5) I am changing the content of this file.
7534bc23 (Shaumik 2014-05-15 03:16:48 +0530  6)
cc48fb3c (Shaumik 2014-06-11 22:38:21 +0530  7) Adding Line 2.
cc48fb3c (Shaumik 2014-06-11 22:38:21 +0530  8)
7534bc23 (Shaumik 2014-05-15 03:16:48 +0530  9) This change is in the master branch!
96f7c5e6 (Shaumik 2014-05-15 03:17:18 +0530 10) Another line in the master branch.
cc48fb3c (Shaumik 2014-06-11 22:38:21 +0530 11)
cc48fb3c (Shaumik 2014-06-11 22:38:21 +0530 12) Adding Line 3.
SMA:my_git_project donny$
```

Figure 5.9. Results of `git blame` on **my_file**

As you can see in Figure 5.9, the command `git blame` displays each line of the file. These lines are prepended with information in the following order: the hash of the commit that added the line, and the commit author, date, time and time zone.

In this scenario, as you already know where the faulty code is, you can just display the details of the required commit to find out more about the bug that was created. Let's assume it was commit `f934591c` that introduced the bug. You should therefore run the following:

```
git show f934591c
```

Once you've figured out what caused the error, you can go ahead and fix it in your repository and then commit the changes.

Normally, though, you'll most likely have no idea what caused the bug. So we need to explore some more debugging tools.

Git Bisect

There's probably no better way to search for a bug than with `bisect`. Even if you have a thousand commits to check, `bisect` can help you do it in just a few steps.

Let's assume you have no idea what's causing an error. However, you *do* know that at a certain point in time—after a particular commit—the bug wasn't present in your code. Git's `bisect` helps you quickly traverse between these stages to identify the

commit that introduced the bug. `bisect` essentially performs a binary search through these commits.

To start the process, you select a "good" commit from the history, where you know the bug wasn't present, and a "bad" commit (which is usually the latest commit). Git then changes the state of your repository to an intermediate commit and asks you if the bug is present there. You search for the bug and assign that commit as "good" or "bad". This process continues until Git finds the faulty commit. Since a binary search algorithm is used, the number of steps required is a logarithmic value of the number of commits in between the initial "good" and "bad" commits.

An example will help explain how `git bisect` works. Let's create a file in our repository, **sum.py**, containing a function that adds two numbers in Python. The contents of the file are as follows:

```
#sum.py
def add_two_numbers(a, b):
    '''
        Function to add two numbers
    '''
    addition = a + b
    return addition

if __name__ == '__main__':
    a = 5
    b = 7
    print add_two_numbers(a, b)
```

I've intentionally added the second block of code to print the response of the function to two dummy values. To run the program, just run the following:

```
python sum.py
```

After adding a few more commits, let's change the file **sum.py** to introduce an error:

```
#sum.py
def add_two_numbers(a, b):
    '''
        Function to add two numbers
    '''
    addition = 0 + b
```

```
    return addition

if __name__ == '__main__':
    a = 5
    b = 7
    print add_two_numbers(a, b)
```

Running the program now, we can see that the result is not `12`, but `7`. Let's now demonstrate the use of `git bisect`. To decide the good and bad commits, we need to have a look at the commit history:

```
git log
```

```
commit 083e7eef5cde1625bcc78417e7b1f82c691875d2
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:55:12 2015 +0530

    Added yet another test

commit 49a6bec7c629e5a84e07c55301f2447f890bad4c
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:51:36 2015 +0530

    Added more tests

commit 5199b4e10ba04b63ed1e76118259913123fbf72d
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:50:09 2015 +0530

    ERROR COMMIT: Introduced error in sum.py

commit b00caea53381979ec1732d919d6f76e3baaf80fc
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:47:32 2015 +0530

    Added tests.py

commit b117516301acd6b08a1c62e3a3f5be48b0b46091
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:44:48 2015 +0530

    Dummy Commit after adding sum.py

commit 7d1b1ec580fbd648b80917d136c60787f3d9038b
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:43:52 2015 +0530

    Added sum.py
```

Figure 5.10. Project history after adding our sample program to add two numbers

As is evident from the history in Figure 5.10, the latest commit `083e7eef5cd` (at the top) is “bad”, whereas the commit two positions before we introduced the bug `7d1b1ec580` is “good”. To better identify the bug, I’ve mentioned in the commit message which commit introduced the error. We must now undertake the following steps to find out the bug:

- start the Git Bisect wizard
- select a good commit

- select a bad commit
- assign commits as good or bad as the wizard takes you through the commits
- end the Git Bisect wizard.

Let's go ahead and start the Git bisect wizard:

```
git bisect start
```

This takes Git into a binary search mode. Next, we need to tell Git the last known commit where the bug was absent, which in our case is `7d1b1ec580`:

```
git bisect good 7d1b1ec580
```

Now assign the latest commit as the bad one:

```
git bisect bad 083e7eef5cd
```

```
SMA:my_git_project donny$ git bisect start
SMA:my_git_project donny$ git bisect good 7d1b1ec580
SMA:my_git_project donny$ git bisect bad 083e7eef5cd
Bisecting: 2 revisions left to test after this (roughly 1 step)
[b00caea53381979ec1732d919d6f76e3baaf80fc] Added tests.py
SMA:my_git_project donny$
```

Figure 5.11. Start of the Git bisect wizard

Why is `git bisect` So Fast?

Notice that in Figure 5.11 the bisect wizard tells you that there are two revisions left for us to perform in this process until it ends. Because bisect essentially performs a binary search, at each step it tries to cut the number of revisions to check by half. In our case, there are six commits to check, which will take about two steps. But 100 commits would require roughly 7 steps, and 1000 commits would require about 10 steps.

To combine the last three commands (`start`, `good`, and `bad`) into one, you may instead start the wizard with the following command:

```
git bisect start 083e7eef5cd 7d1b1ec580
```

As soon as you assign the good and bad commits, `git bisect` starts its work and takes the state of your repository to an intermediate commit. At this point, you're shown the commit hash and commit message, and you're asked whether or not the bug is present in that commit.

Learn More About Each Commit

If you want to know more about a commit during the time the bisect wizard is running, you can run `git show` for the commit.

In our situation, we just run the file **sum.py** to find out if the bug is present. For the commit `b00caea5`, we see that the output is `12`. So the bug is absent. We mark it as good, as shown in Figure 5.12:

```
git bisect good
```

```
SMA:my_git_project donny$ python sum.py
12
SMA:my_git_project donny$ git bisect good
Bisecting: 0 revisions left to test after this (roughly 1 step)
[49a6bec7c629e5a84e07c55301f2447f890bad4c] Added more tests
SMA:my_git_project donny$
```

Figure 5.12. Assigning a commit as good during the bisect process

In the next step, we're asked whether commit `49a6bec7c6` is good. We check the commit by running **sum.py** again and assign it as bad:

```
git bisect bad
```

Once we're done with this, Git shows us the faulty commit as `7a3d629df`, which is also evident from the commit message that I added when I introduced the error:

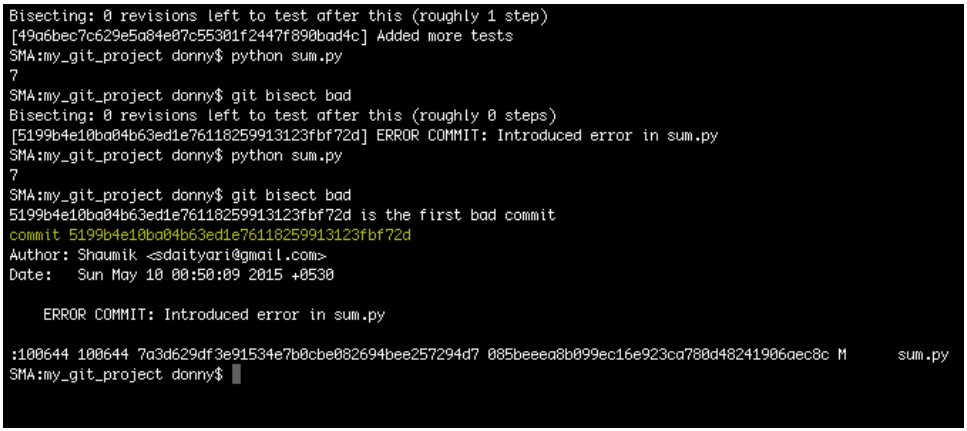


Figure 5.13. Bisect results

Once you've found your faulty commit, you can exit the wizard by running the following:

```
git bisect reset
```

In this case, the use of `git bisect` was overkill and not necessary (as we knew the source of the bug already). However, in real life there are often bugs that are difficult to trace back to a file, but the bug is visible only in the way your code functions. For instance, you have a complex algorithm to find out the popularity of a person in social media and you find out that the results are not right. In such cases, you employ the bisect tool to find out which commit first introduced the error to rectify it.

Automated Bisect with Unit Tests

We've just seen how bisect helps you find the commit that introduced a bug. However, this process is tedious, as you need to check for the bug at every single step of the wizard.

The easiest way to automate the process is to write unit tests. You can also write custom scripts that test the required functionalities. In our case, we'll write a custom file, **test_sum.py**, that tests the functionality of the function in `sum.py`. This file is just for demonstration of the functionality of `bisect`. (You don't need to understand

the code here. To learn more about testing in Python, you can read about Python's `unittest` module[2].)

Exit Codes in Custom Shell Scripts

If you create a custom shell script to perform your tests, make sure it has custom exit codes, in addition to printing messages on the terminal about the status of the tests. In general, the `0` exit code is considered a success, whereas everything else is a failure.

```
#test_sum.py
import unittest
from sum import add_two_numbers

class TestsForAddFunction(unittest.TestCase):

    def test_zeros(self):
        result = add_two_numbers(0, 0)
        self.assertEqual(0, result)

    def test_both_positive(self):
        result = add_two_numbers(5, 7)
        self.assertEqual(12, result)

    def test_both_negative(self):
        result = add_two_numbers(-5, -7)
        self.assertEqual(-12, result)

    def test_one_negative(self):
        result = add_two_numbers(5, -7)
        self.assertEqual(-2, result)

if __name__ == '__main__':
    unittest.main()
```

Running the file **test_sum.py** runs the tests specified in it. Running it on our current code shows errors, as seen in Figure 5.14:

[2] https://docs.python.org/2/library/unittest.html

```
python test_sum.py
```

```
SMA:my_git_project donny$ python test_sum.py
FFF.
======================================================================
FAIL: test_both_negative (__main__.TestsForAddFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum.py", line 17, in test_both_negative
    self.assertEqual(-12, result)
AssertionError: -12 != -7

======================================================================
FAIL: test_both_positive (__main__.TestsForAddFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum.py", line 13, in test_both_positive
    self.assertEqual(12, result)
AssertionError: 12 != 7

======================================================================
FAIL: test_one_negative (__main__.TestsForAddFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum.py", line 21, in test_one_negative
    self.assertEqual(-2, result)
AssertionError: -2 != -7

----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=3)
SMA:my_git_project donny$
```

Figure 5.14. Running tests on the current code

Let's start the bisect process again:

```
git bisect start 083e7eef5cd 7d1b1ec580
```

We next inform Git about the command that runs the tests:

```
git bisect run python test_sum.py
```

If you have a custom command to run your tests, replace **python test_sum.py** with your command.

On informing Git about the command that tests our code, the wizard runs it against the remaining commits and figures out which commit introduced the error, as shown in Figure 5.15:

```
SMA:my_git_project donny$ git bisect start 083e7eef5cd 7d1b1ec580
Bisecting: 2 revisions left to test after this (roughly 1 step)
[b00caea53381979ec1732d919d6f75e3baaf80fc] Added tests.py
SMA:my_git_project donny$ git bisect run python test_sum.py
running python test_sum.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
Bisecting: 0 revisions left to test after this (roughly 1 step)
[49a6bec7c629e5a84e07c55301f2447f890bad4c] Added more tests
running python test_sum.py
FF.
======================================================================
FAIL: test_both_positive (__main__.TestsForAddFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum.py", line 13, in test_both_positive
    self.assertEqual(12, result)
AssertionError: 12 != 7

======================================================================
FAIL: test_one_negative (__main__.TestsForAddFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum.py", line 17, in test_one_negative
    self.assertEqual(-2, result)
AssertionError: -2 != -7

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=2)
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[5199b4e10ba04b63ed1e76118259913123fbf72d] ERROR COMMIT: Introduced error in sum.py
running python test_sum.py
FF.
======================================================================
FAIL: test_both_positive (__main__.TestsForAddFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum.py", line 13, in test_both_positive
    self.assertEqual(12, result)
AssertionError: 12 != 7

======================================================================
FAIL: test_one_negative (__main__.TestsForAddFunction)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum.py", line 17, in test_one_negative
    self.assertEqual(-2, result)
AssertionError: -2 != -7

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=2)
5199b4e10ba04b63ed1e76118259913123fbf72d is the first bad commit
commit 5199b4e10ba04b63ed1e76118259913123fbf72d
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:50:09 2015 +0530

    ERROR COMMIT: Introduced error in sum.py

:100644 100644 7a3d629df3e91534e7b0cbe082694bee257294d7 085beeea8b099ec16e923ca780d48241905aec8c M      sum.py
bisect run success
SMA:my_git_project donny$
```

Figure 5.15. Automating the process of git bisect

Once the bug has been identified, reset the wizard:

```
git bisect reset
```

Beware of Using Old Test Files

If you're using a testing script for the process of running bisect, be aware that when Git is testing an old commit, it's checking against the old version of the testing script, too.

You can instead provide new test files, which are not a part of the repository. Even when old commits are being tested, your latest script would be used for the process.

Once you've found out which commit introduced the error, you can look carefully into it to see the faulty code. Once you identify that, you can fix it and commit it to the repository.

Conclusion

What Have You Learned?

In this chapter, we looked at how Git lets you undo mistakes:

- Undo Git Add
- Undo Git Commit
- Undo Git Push

We also looked at two debugging tools, which help you find bugs in your Git workflow:

- Blame
- Bisect

What's Next?

In the next chapter, we'll look at a list of useful commands that help you use Git to its fullest.

Chapter 6

Unlocking Git's Full Potential

So far in this book, we've covered the fundamentals of Git and some of its advanced commands. In this chapter, we'll look at more of these advanced commands.

Advanced Use of `log`

We've seen earlier that you can view the history of your project in Git using the `log` command. However, in busy repositories that handle hundreds to thousands of commits each day, a long list of commits is not going to be useful unless you know how to navigate through them. The manual entry for the `log` command[1] shows the different options that can be postfixed to this command to get a desired output. We'll look at a few tweaks to the `log` command, which could prove useful in such situations.

Since our dummy project[2] doesn't have a considerable number of commits, we're going to use the open-source repository of an e-learning management system, ATutor[3], to explore the different capabilities of the `log` command.

[1] http://git-scm.com/docs/git-log
[2] https://github.com/sdaityari/my_git_project
[3] https://github.com/atutor/atutor/

Short Version

In general, the `log` command shows a list of commits in the active branch, each with the commit hash, author, date and commit message. Depending on your screen size and text size of the output, you get around five to ten commit details in a screen. Each commit occupies four to five lines on the screen, or even more if the size of the commit message is large.

In case you want to have a quick glance at the list of commits, you can format the output to show only the commit hashes and single line messages. A single commit is displayed on each line, and thus many more commits fit onto the screen at once:

```
git log --oneline
```

The screenshots in Figure 6.1 illustrate the effect of this command:

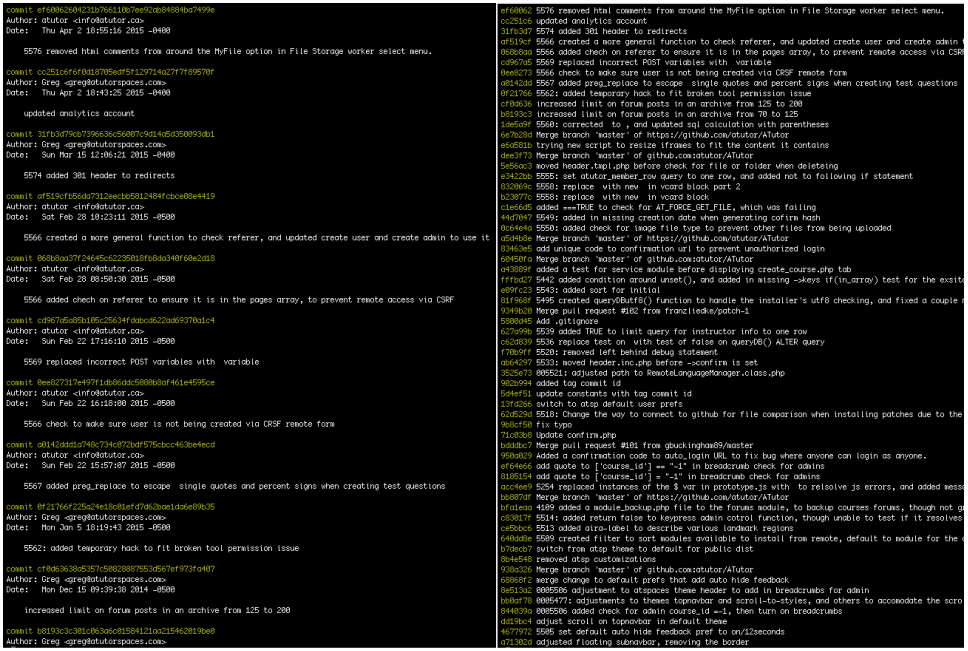


Figure 6.1. Comparison of the `log` command without (left) and with (right) the use of `--oneline`

Branches and History

The `log` command can also be used to view the workflow and commits in branches other than the current active branch. If you want to view the commits in all branches, just postfix `--all` to the command:

```
git log --all
```

As our dummy repository contains only a few commits for every branch, let's go back and see the effect of postfixing `--all` to the `log` command (Figure 6.2):

```
commit 083e7eef5cde1625bcc78417e7b1f82c691875d2
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:55:12 2015 +0530

    Added yet another test

commit 49a6bec7c629e5a84e07c55301f2447f890bad4c
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:51:36 2015 +0530

    Added more tests

commit 5199b4e10ba04b63ed1e76118259913123fbf72d
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:50:09 2015 +0530

    ERROR COMMIT: Introduced error in sum.py

commit b00caea53381979ec1732d919d6f76e3baaf80fc
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:47:32 2015 +0530

    Added tests.py

commit b117516301acd6b08a1c62e3a3f5be48b0b46091
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:44:48 2015 +0530

    Dummy Commit after adding sum.py
```

Figure 6.2. Showing commits in all branches

This doesn't look very appealing, as you have no idea which commit came from which branch. You can add the `--decorate` option to view which branch each commit belongs to. It also shows the remote branches. Note that I have used `--oneline` to accommodate more commits in the screenshot in Figure 6.3:

```
git log --all --decorate --oneline
```

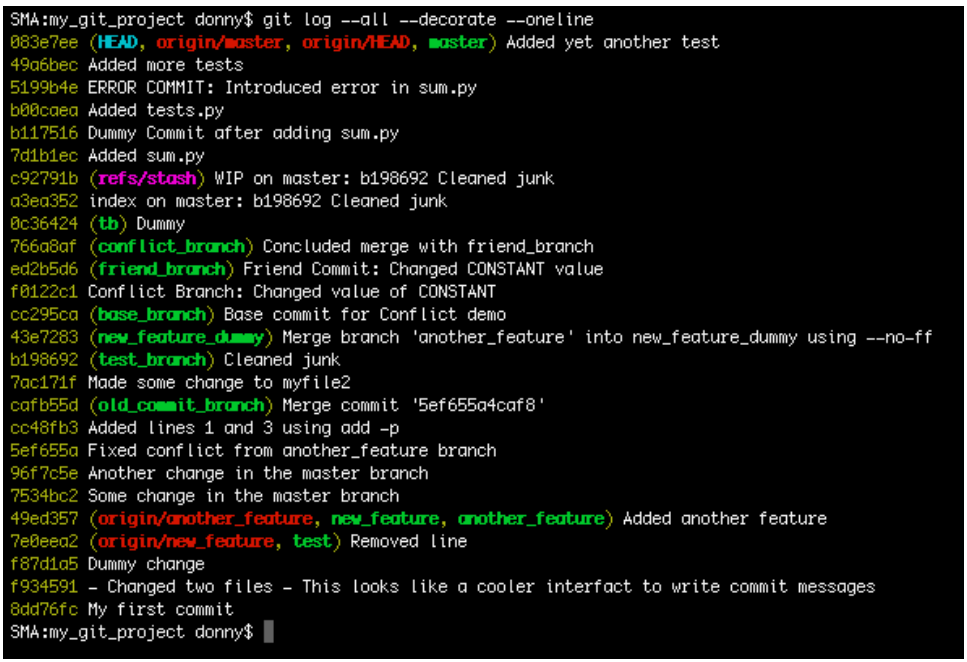


Figure 6.3. Showing commits with the branches they belong to

The `--graph` option shows you the commit history, with a graphical representation interconnecting the links between commits of different branches (if any). Combining it with `--all` shows you how the different branches in your repository have progressed:

```
git log --all --decorate --oneline --graph
```

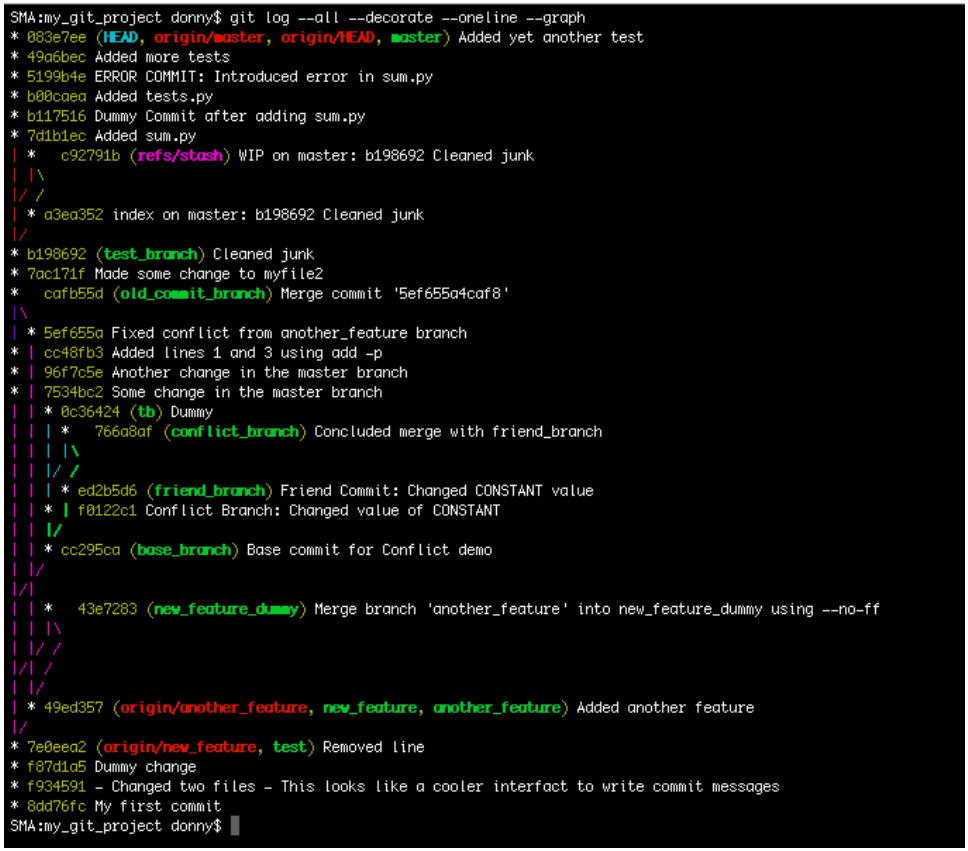


Figure 6.4. Graphical representation of commits of all branches

To understand this concept better, let's take a look at Figure 6.4. `8dd76fc` is the first commit of this repository, which appears at the bottom of the output. As you traverse upwards from the bottom of the figure, notice that commit `49ed357` diverges from the master branch into a new branch, `another_feature`. Following the path of `another_feature` shows us that commit `53f655a` is the last commit in the branch, before it merges back with master at commit `cafb55d`.

Filter Commits

When you view the history, you're shown all the commits in the history's branch. However, if you wish to view only a few of the latest commits, postfix `-n`, followed by the number of commits you wish to see:

```
git log -n 2
```

Alternatively, you can use the following command as well, which serves as a shortcut for the previous command:

```
git log -2
```

Figure 6.5. Showing a specified number of commits in history

You can also view the commits in a specified time range. This can be achieved by postfixing `--after` and `--before` to the `log` command:

```
git log --after='2015-3-1' --before='2015-5-1'
```

`--after` and `--before` can be replaced by `--since` and `--until`. For instance, the following pairs of commands will produce the same results:

```
git log --after='2015-3-1'
git log --since='2015-3-1'

git log --before='2015-3-1'
git log --until='2015-3-1'
```

```
git log --after='2015-3-1' --before='2015-6-1'
git log --since='2015-3-1' --until='2015-6-1'
```

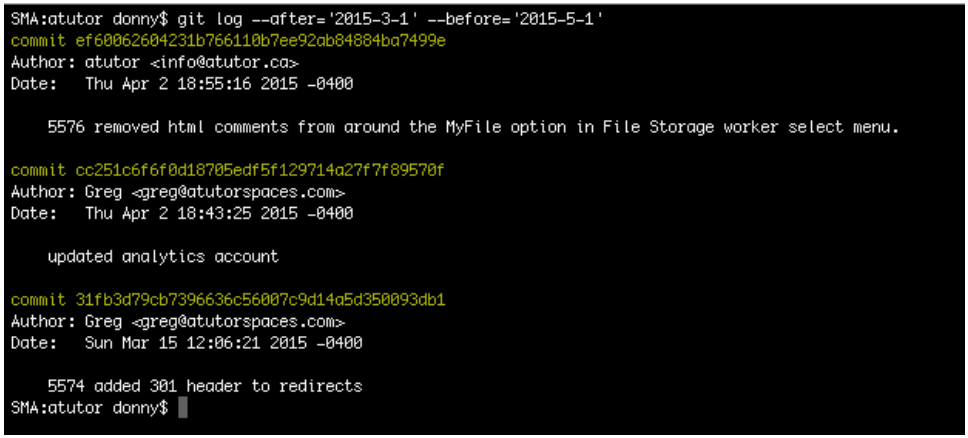


Figure 6.6. Showing commits in a time range

You Must Specify a Range

The specified dates have to signify a date range, as it doesn't make sense for Git to search for a commit at a point in time. If you want to find the commits on a particular day, you need to specify the whole day in the range.

You can also use date references such as "Yesterday" or "1 week ago", as explained by Alex Peattie on his blog[4].

Trace Changes in a Single File

If you want to check the commits that resulted in changes in a single file, you can use the `--follow` option:

[4] http://alexpeattie.com/blog/working-with-dates-in-git/

```
git log --follow index.php
```

```
SMA:atutor donny$ git log --follow --oneline index.php
0ecd220 undo grabhandle commit
6ad3c0a add descriptive labels for movable grid reorderer elements
69b9ed6 setting up  for tintructor admin toolbar
be304e5 add slow to various jQuery hide/shows
8c2c233 replaced mysql with queryDB
982269c generate the description metadata for a course home page using the course decription
5f5c072 Committing intial multisite - this should not beused in production. Please edit to remove aboslute paths
70cf818 move code up one directory
b8db04e http://atutor.ca/atutor/mantis/view.php?id=4821
e5894c6 fix the script header copyright/license block
a5cb88f moved infusion/framework/fss/css/fss-layout.css into default header.tmpl
698c212 http://www.atutor.ca/atutor/mantis/view.php?id=4105
f65a41a fix the location of test_result_functions.inc.php: point to new location
ec66fab http://www.atutor.ca/atutor/mantis/view.php?id=3907
bd0a394 http://www.atutor.ca/atutor/mantis/view.php?id=3978
732510e fix the bug that the "close" button on detail-viewed module does not work.
f3ae5c4 fix html validation errors due to '<' used in javascript. The solution is to wrap javascript into html c
c9b3fea http://www.atutor.ca/atutor/mantis/view.php?id=3937
2feef54 http://www.atutor.ca/atutor/mantis/view.php?id=3934
ec6d56f http://www.atutor.ca/atutor/mantis/view.php?id=3925
daf1618 http://www.atutor.ca/atutor/mantis/view.php?id=3890 http://www.atutor.ca/atutor/mantis/view.php?id=3894
 javascript error.
91c2abc use ajax request to remove modules from course index page and student tools page, at "detail view"
a26235e implement fluid re-orderer for module "detail view" on course index page and student tools index page
99dbc2b 1. on "student tools" index page, allow tool of moving icons around 2. in "constants.inc.php", fix the c
ee6ce5b 1. in switch_view.php and move_module.php, redirect back to http_referer, the referer can be "course ind
t/right, as they are not available to student tools.
d98ba64 1. move query to access courses.home_type out of template 2. use absolute URL instead of hardcoded path
alling them from different path depth.
7c8e027 1. wrap timezone handling inside AT_DATE() function. 2. remove calls on at_timezone() from scripts.
d64ae7e fixed type in head of file
70348b4 1. In "edit content" page, revert form name from "contentForm" back to "form" in order to fix the proble
176d2e3 add at_timezone to announcement dates
e939869 Add author and date to announcements displayed on course home page
60fd24b update copyright date in code header
a34b57c
21d6eae http://atutor.ca/atutor/mantis/view.php?id=3074 http://atutor.ca/atutor/mantis/view.php?id=3075
873aafa http://www.atutor.ca/atutor/mantis/view.php?id=2883 http://www.atutor.ca/atutor/mantis/view.php?id=2895
```

Figure 6.7. Tracking a single file

Tracing the changes in a file may be useful while debugging, especially if you want to see if anyone has changed a particular file since a certain time. It also helps you to check if parts of a file were removed in previous commits.

How Is Tracing Different From `git blame`?

We used the `blame` command earlier to get more information about each line in a file, and which commit it is associated with. `blame` enables you to check only the current contents of a file. The `log --follow` command, on the other hand, lists the changes the file has gone through since Git started tracking the file.

Therefore, any part of the file that was removed in an earlier commit would show up on the output of `log --follow`, but not on `blame`.

Track Your Peers

The `shortlog` is a command that shows the authors who've contributed to the repository, their commits and commit messages. You may use this command if you're interested in knowing the contributions of different developers.

The output of this command is sorted by name, and you can postfix `-n` to sort it by the number of commits[5]:

[5] We used `-n` earlier too. Note that `-n` is often postfixed to a command when you want to limit the number of outputs.

```
git shortlog
```

```
SMA:atutor donny$ git shortlog
ATutor (25):
      Merge pull request #77 from mancoolgunda/get_group_activties
      Merge pull request #82 from mancoolgunda/auto_enroll_feedback
      Merge pull request #84 from mancoolgunda/pref_auto_hide_feedback
      Merge pull request #83 from mancoolgunda/unenroll_group_forum
      Merge pull request #81 from mancoolgunda/disabled_theme
      Merge pull request #80 from mancoolgunda/edit_expired_test
      Merge pull request #85 from mancoolgunda/autoenroll_registered
      Merge pull request #87 from mancoolgunda/admin_custom_logo
      Merge pull request #88 from ayushgupta2209/sessionbox
      Merge pull request #90 from mancoolgunda/custom_logo_update
      Merge pull request #91 from ayushgupta2209/content
      Merge pull request #92 from mancoolgunda/admin_custom_logo_update
      Merge pull request #96 from kashyap7/master
      Merge pull request #97 from easycron/patch-1
      Merge pull request #67 from mancoolgunda/groups_detailed_view
      Merge pull request #98 from mancoolgunda/tests_no_questions
      Merge pull request #101 from gbuckingham89/master
      Update confirm.php
      Merge pull request #102 from franzliedke/patch-1
      added ===TRUE to check for AT_FORCE_GET_FILE, which was failing
      5558: replace  with new  in vcard block
      5558: replace  with new  in vcard block part 2
      5555: set atutor_member_row query to one row, and added not to following if statement
      trying new script to resize iframes to fit the content it contains
      Merge branch 'master' of https://github.com/atutor/ATutor

ATutor Dev (1):
      add copy config_multisite.php if exists

ATutorSpace Root User (2):
      replaced % with %% in user query for compatibility with queryDB()
      5472 check for valid user instead of memember_id, removed returning user drop down for admins

AYUSH GUPTA (3):
      ISSUE 5439: Fixed default button in timeout dialog
      Indentation fix
      ISSUE 5438: Content Navigation improvement

Abhinav Koppula (4):
      Groups Detailed View fixed
      Resolved mixing of js and php; Removed code duplication; Fixed indentation
      Issue 4097: Indentation fixed on lines 59 and 61.
      Issue 4097: Removed unwanted comments
```

Figure 6.8. Displaying all authors and their contributions

You can view the commits by a single author too, by using the `--author` option:

```
git log --author='Alexey'
```

```
SMA:atutor donny$ git log --author='Alexey' -3
commit baa46007549ffc14063839498adf0f1d2d4a56f2
Author: Alexey Novak <abovebluesky@gmail.com>
Date:   Fri Feb 15 16:11:18 2013 -0500

    Creating a separate function for session logging out. Renaming a function.

commit 8692916c5de14d22ea630b6c13e1fa9cd9c67fad
Merge: 78c4654 1ac9d14
Author: Alexey Novak <abovebluesky@gmail.com>
Date:   Fri Feb 15 15:50:31 2013 -0500

    Merge branch 'umaster' into cookieTimeout

commit 78c4654a43281e22ae2bda3c4a792a80f8c2659c
Author: Alexey Novak <abovebluesky@gmail.com>
Date:   Fri Feb 15 15:50:06 2013 -0500

    Making all functions private and exposing only autoLogout to the user
SMA:atutor donny$
```

Figure 6.9. Display commits by a single author

You only need to type just enough of the name for Git to identify the author. If there are two authors matching the string you've provided, both their commits will be displayed. If there are two authors with the same name committing to the same repository, Git differentiates them through other details—such as their email address, or the system from which the commit was generated.

Search in Commit Messages

Imagine a situation where you'd like to know when a certain feature was introduced. Searching for a commit through its commit message would be useful. Git enables searching in the commit messages by using the `--grep` option. For instance, if you want to search for the word "redirect" in your commit history, you should use the following command:

```
git log --grep='redirect'
```

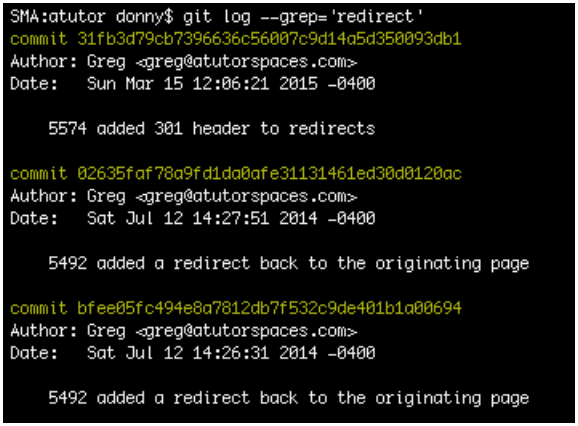


Figure 6.10. Search within commit messages

On the Importance of Meaningful Commit Messages

When I introduced commits in this book, I mentioned the importance of writing meaningful commit messages, even though it's not mandatory. Imagine how difficult it would be to search through commits if your commit messages weren't meaningful!

You can also use regular expressions while using the `grep` command.

Using the grep Terminal Command

You can use the terminal command `grep` (not to be confused with Git's `grep` option!) to search commit messages too. The command for that is:

```
git log --oneline | grep 'redirect'
```

The pipe (`|`) passes on the output of the command `git log --oneline` to the second part, which searches for the word "redirect" in it.

The terminal `grep` command works on Linux and OS X, but has no native command substitute in Windows, although there's a Findstr[6] command that performs

[6] https://technet.microsoft.com/en-us/library/bb490907.aspx?f=255&MSPPError=-2147217396

a similar task. You can, however, install third party utilities like Cygwin[7] and UnxUtils[8], which enable the use of the `grep` command on Windows.

Tagging in Git

You've most likely noticed that software updates normally come with a version number. For instance, as of August 5th, 2015, the version number of Google's Chrome browser is 44.0.2403.130.

Git allows you to associate these version numbers with specific milestone commits in your repositories, by attaching labels to these commits. The labels are called **tags**. Let's again visit the ATutor repository to check its use of tags.

Tagging can be used to easily find any commit that's important to a developer. Tags can also be used to mark a breakthrough after debugging, or a milestone in development. They can also be used to mark changes being made without creating an extra branch. Tags provide an easy way to go back in branch history if something didn't work out right.

To list the tags in alphabetical order, run the following command:

```
git tag
```

Figure 6.11. List of tags in the ATutor project

There are two types of tags—**lightweight** and **annotated**. Lightweight tags contain only the tag name and point to a commit. Annotated tags contain the tag name, information about the tagger, and a message associated with the tag.

[7] http://www.cygwin.com/
[8] http://unxutils.sourceforge.net/

Annotated tags are generally preferred in organizations, because they contain information about the tagger, when the tag was created, and why. Lightweight tags are handy for tagging special commits when you're working on your personal projects.

To view the details of a tag—say `Atutor_1.4.1`—run the following command:

```
git show Atutor_1.4.1
```

You can create a lightweight tag `latest_commit`, associated with the latest commit, by running the following:

```
git tag latest_commit
```

To create an annotated tag, you need to postfix `-a` for annotated and `-m` for an associated message:

```
git tag -a latest_commit -m "this is the latest commit"
```

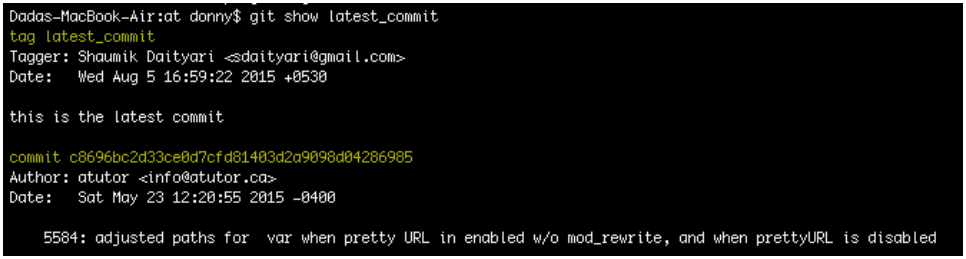


Figure 6.12. Details of an annotated tag

You can also checkout to a tag `Atutor_1.4.1` by creating a new branch `version_1_4_1` (just like you checkout to a commit):

```
git checkout -b version_1_4_1 Atutor_1.4.1
```

When you push your code, your tags aren't pushed to the remote. If you specifically want to push newly created tags to the remote `origin`, you can run the following:

```
git push origin --tags
```

If you specifically want to push a tag to a remote, run the following:

```
git push origin Atutor_1.4.1
```

Refs and `reflog`

Now that we've explored the `log` command in detail, let's now have a look at something new—refs. You already know that a commit is identified by its hash, a long string unique to a commit. A **ref**, short for a reference, is a way of referencing a commit. In other words, the hash is a name, whereas a ref is a pointer.

Refs are stored internally in Git, and we won't go into how Git treats refs. We will, however, use the `reflog` command to utilize refs.

We've discussed what a `HEAD` in Git points to. At this point, it's important to note that `HEAD` is also a ref. There are other such special refs like `ORIG_HEAD`, `MERGE_HEAD` and `FETCH_HEAD`.

This brings us to the `reflog`. It's a "log of refs". That is, any change you make in Git is recorded and accessible via the `reflog` command. For instance, if you create a commit, checkout to a new branch, merge two branches, pull, push or even make a failed merge, `reflog` records them all:

```
git reflog
```

```
SMA:my_git_project donny$ git reflog
cafb55d HEAD@{0}: reset: moving to HEAD~2
b198692 HEAD@{1}: checkout: moving from test to test_branch
f934591 HEAD@{2}: reset: moving to HEAD~2
7e0eea2 HEAD@{3}: checkout: moving from squash to test
dd43634 HEAD@{4}: rebase -i (finish): returning to refs/heads/squash
dd43634 HEAD@{5}: rebase -i (squash): Squash: After First Code Review
085ba54 HEAD@{6}: rebase -i (start): checkout HEAD~3
7e05633 HEAD@{7}: rebase -i (finish): returning to refs/heads/squash
7e05633 HEAD@{8}: rebase -i (start): checkout HEAD~2
7e05633 HEAD@{9}: rebase: aborting
3c1df4f HEAD@{10}: rebase -i (start): checkout HEAD~2
7e05633 HEAD@{11}: rebase -i (finish): returning to refs/heads/squash
7e05633 HEAD@{12}: rebase -i (start): checkout HEAD~2
7e05633 HEAD@{13}: commit: Squash: After Second Code Review
```

Figure 6.13. Changes in a repository visible through `reflog`

The `reflog` command stores the records for each action you perform in your repository. When you push the changes, this data isn't synced with the server. Using the `reflog` command is necessary if you want to review changes to your local repository. It could also be used to recover lost commits.

`reflog` Can Act as Insurance

If you make a hard reset and lose a commit or two, you can safely go back to any commit you made earlier. For instance, you can run the `reflog` command, which would have a record corresponding to the time when the commit was created, mentioning the commit hash. When you know the hash, you can start a new branch based on that commit to go back to the state of that commit.

The `reflog` is like an insurance policy in Git.

`reflog` Only Tracks Commits for a Certain Period of Time

The `reflog` command only tracks back changes for a certain amount of time. Git is responsible for cleaning up the reflog data periodically, which by default is 90 days. You may modify this value by specifying the `expire` option of the command. If you want `reflog` never to forget any action, run the following command:

```
git reflog expire --expire=never
```

Checking for Lost Commits

We've just seen how `reflog` can help you search for commits that might be lost because of the use of a hard reset. However, it's difficult to search specifically for lost commits in a repository with a huge history.

A commit is *lost* when it's not a part of any branch. The `log` command fails to search and show lost commits. One way of losing commits from your branch history is to do a hard reset, but deleting a branch without merging it with a different one can also lead to commits that are recorded by Git, but not present anywhere in any of your branches.

You can search for commits that aren't a part of any branch by using the `fsck` (file system check) command:

```
git fsck --lost-found
```

```
SMA:my_git_project donny$ git fsck --lost-found
Checking object directories: 100% (256/256), done.
dangling blob 0f2416ebfdd47d1ace2728ed53b22fecae2c9fb3
dangling commit 136273f292b22bc09663af1ff1dfd073458c93cc
dangling commit 18410c487439c2774bf7bb0f6aa9e85279bde943
dangling commit 5824b6253a2d23dcd94978737327896af3b3840e
dangling commit 7e0563324211c5910450b7316bf031db35b1c7ee
dangling commit 960fa2861bdac2145b11265799ec5dc420e2504f
dangling commit c9067cebe0333e0c5b48eb6ab3c052dc4ef52012
dangling commit c92791bd1c7381295985131d1ad48d84d9d0f831
dangling commit dd43634d74e5e0745c7abf7aedd526e828de5de2
dangling blob e539649594c846b5e880e4a22932a0f961d249ad
SMA:my_git_project donny$
```

Figure 6.14. Finding lost commits through `git fsck`

Not to be Confused with the Unix Command

`fsck` is also a Unix command to check for and repair inconsistencies in your file system. Don't confuse it with the `git fsck` command, which checks for inconsistencies in your commits.

If you want to recover a lost commit `c9067` from the list to your current branch, you can run the following:

```
git merge c9067
```

`fsck` Versus `reflog`

There's an advantage of `fsck` over `reflog`. Imagine you cloned a remote branch and deleted it. The commits present there would never show up on `reflog`, because they were never done on your local system. However, `fsck` will list all the lost commits from that branch.

Rebase

We saw earlier how merge works: it creates loops in the commit history of a project. These loops don't really cause any problems for Git, though over time, they can make project histories difficult to understand and navigate. For the central repository of a project, it's preferable to have a linear history, rather than a bunch of interconnected loops.

In this section, we'll discuss a merging mechanism—`rebase`—that avoids loops in the project history. I mentioned `rebase` earlier, when I used it with the `git pull` command. Quite literally, the process of "rebasing" is a way of rewriting the history of a branch by moving it to a new "base" commit.

If you're rebasing a `master` into `new_feature`, the new commits in `master` are put before the new commits in `new_feature` that are not common to `master`. To do so, run the following command from the `new_feature` branch:

```
git rebase master
```

Working in a Team

If you're working in a team, you should first checkout to `master`, pull from the upstream branch to update your `master` with the latest commits, and then switch back to `new_feature` before running the above command.

This can also be accomplished by the following:

```
git merge --rebase master
```

Let me illustrate the above command in Figure 6.15:

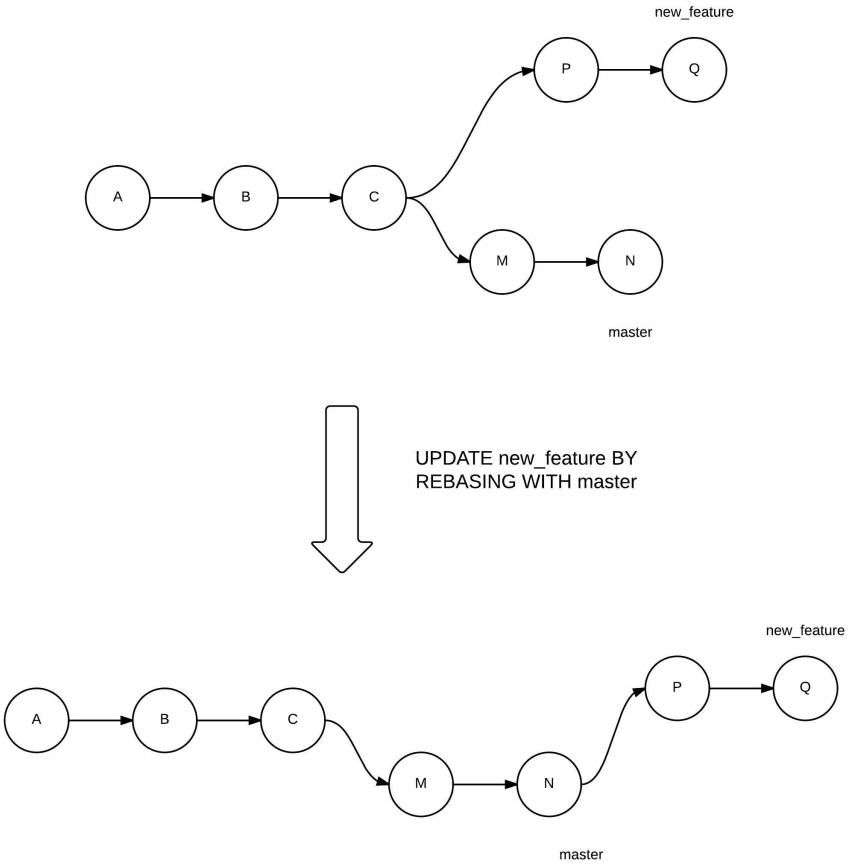


Figure 6.15. Rebase `master` into `new_feature`

One important observation from the diagram is the presence of a linear commit history, which is not present in a merge. Let's see the difference between a merge and a rebase for two branches (Figure 6.16):

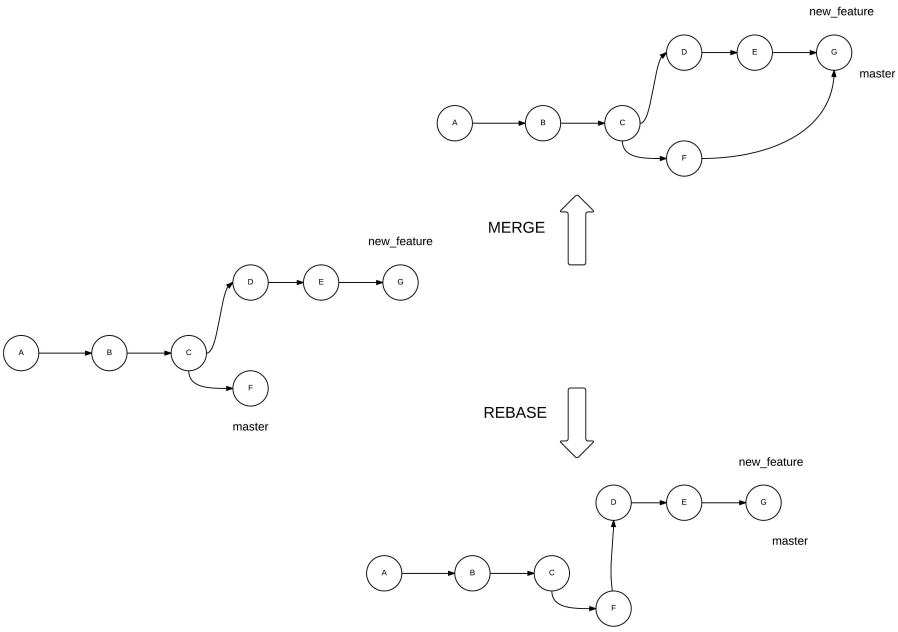


Figure 6.16. Illustrating the difference between merge and rebase

A rebase operation may lead to conflicts, just like a merge operation. The process of resolving a conflict is exactly the same as we discussed earlier.

When you're pulling changes, you can use `rebase` too. It essentially puts the new commits in the master of the remote in your history, and then superimposes your commits on them. Any conflicts that arise can be fixed easily, because they've been raised by your code. You can rebase with a pull using the following command:

```
git pull --rebase origin master
```

Just for Illustration

The last command assumes that you added commits to your `master` branch and then updated it from the central repository. This is just for the sake of argument, and not the best way to work in Git. Ideally, when you work in your own branch and keep it updated using pull operations, no conflicts would arise in the `master` branch.

Squash Commits Together

When you're contributing to a codebase by working on a different branch, the code may not be accepted at the first go. Once changes in your code have been suggested, you create a new commit with the changes. You may, however, be asked to make more changes and, before you know it, you may have added multiple commits to the pull request. Since you created the pull request asking for your code to be merged, all of the commits would also get merged.

In such a situation, you might have a list of commits, the first of which was an attempt to resolve a bug, whereas the latter were attempts at refactoring the code to follow best coding practices. The group of commits as a whole signifies a single task that has been accomplished, and hence, it makes logical sense to package them together as a single commit (rather than merging these multiple commits into the main project history).

This can also be done through the `rebase` command (essentially rebasing your current branch). If you want to squash the last two commits, run the following command:

```
git rebase -i HEAD~2
```

The `HEAD~2` refers to the last two commits in the current branch, and the `-i` option stands for interactive (which can be replaced by `--interactive`). You're then taken to an interactive screen, where you need to `pick` the old commit and `squash` the latest commit (Figure 6.17):

```
pick 3c1df4f Squash: Base Commit
squash dd43634 Squash: After First Code Review

# Rebase 083e7ee..dd43634 onto 083e7ee (       2 TODO item(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Figure 6.17. Git showing a list of commits to squash

You then proceed to provide a commit message (Figure 6.18):

```
# This is a combination of 2 commits.
# The first commit's message is:


# This is the 2nd commit message:

Squashed last two commits

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun May 17 00:35:33 2015 +0530
#
# rebase in progress; onto 083e7ee
# You are currently editing a commit while rebasing branch 'squash' on '083e7ee'.
#
# Changes to be committed:
#   modified:   my_file
#   modified:   myfile2
#   modified:   myfile3
#
```

Figure 6.18. Git asking for a commit message for the squashed commit

Let's look at the repository after the squash operation, to make sure the last two commits have been converted into one (Figure 6.18):

```
SMA:my_git_project donny$ git log --oneline
dd43634 Squash: After First Code Review
3c1df4f Squash: Base Commit
083e7ee Added yet another test
49a6bec Added more tests
5199b4e ERROR COMMIT: Introduced error in sum.py
b00caea Added tests.py
b117516 Dummy Commit after adding sum.py
7d1b1ec Added sum.py
b198692 Cleaned junk
7ac171f Made some change to myfile2
cafb55d Merge commit '5ef655a4caf8'
cc48fb3 Added lines 1 and 3 using add -p
5ef655a Fixed conflict from another_feature branch
96f7c5e Another change in the master branch
7534bc2 Some change in the master branch
49ed357 Added another feature
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a cooler interfact to write commit messages
8dd76fc My first commit
SMA:my_git_project donny$ git rebase -i HEAD~2
[detached HEAD 30dc1fa] Squashed last two commits
 Date: Sun May 17 00:35:33 2015 +0530
 3 files changed, 2 insertions(+), 4 deletions(-)
Successfully rebased and updated refs/heads/squash.
SMA:my_git_project donny$ git log --oneline
30dc1fa Squashed last two commits
083e7ee Added yet another test
49a6bec Added more tests
5199b4e ERROR COMMIT: Introduced error in sum.py
b00caea Added tests.py
b117516 Dummy Commit after adding sum.py
7d1b1ec Added sum.py
b198692 Cleaned junk
7ac171f Made some change to myfile2
cafb55d Merge commit '5ef655a4caf8'
cc48fb3 Added lines 1 and 3 using add -p
5ef655a Fixed conflict from another_feature branch
96f7c5e Another change in the master branch
7534bc2 Some change in the master branch
49ed357 Added another feature
7e0eea2 Removed line
f87d1a5 Dummy change
f934591 - Changed two files - This looks like a cooler interfact to write commit messages
8dd76fc My first commit
SMA:my_git_project donny$
```

Figure 6.19. Status of repository before and after squash

Aborting a Squash

If a squash operation gets overwhelming, you can safely run `git rebase --abort` to get back to the pre-squash state.

Squash Modifies the Branch History

A squash operation changes the history of your branch. If you need to push your changes after a squash operation, you need to use the `-f` option, or your push will be rejected.

Stash Changes

Imagine a situation where you're working on a bug or a feature, and many files have been edited since the last commit. However, you need to switch branches to work on something else, or you need to demonstrate the state of your last commit. You can't commit your current changes, as they're not complete yet. How do you solve this problem? `stash` allows you to save the changes you've made in your repository and revert back to the state of the last commit. At a later stage, you can get back your changes if you wish. To stash uncommitted changes, run the following command:

```
git stash
```

You can check the list of stashes in your Git repository by running the following:

```
git stash list
```

```
SMA:my_git_project donny$ git stash list
stash@{0}: WIP on master: 083e7ee Added yet another test
stash@{1}: WIP on master: b198692 Cleaned junk
SMA:my_git_project donny$
```

Figure 6.20. List of stashes in a repository

Do note in Figure 6.20 the serial numbers associated with each stash, which Git uses to identify it. The commit hash and message refer to the last commit of the active branch when you stashed the changes.

To apply the changes that were stored in the last stash, you can use the following command:

```
git stash apply
```

To restore an old stash, you need to mention the serial number next to the stash in the list of stashes:

```
git stash apply stash@{1}
```

You can apply multiple stashes too.

`stash` Only Stashes Tracked Files

The `stash` command stashes the changes that have been made to tracked files only. If you want to add an untracked file to the stash as well, just start tracking it with `git add` before running the `stash` command.

Don't Use the `-u` Option Just Yet

In newer versions of Git (1.7.7+), you can add the `-u` option to stash untracked files without tracking them. However, you should avoid its usage, as many users have reported bugs where ignored files from the **.gitignore** file have been deleted on using the `-u` option with `stash`.

Advanced Use of `add`

We saw earlier that we can instruct Git to track a new file, or stage changes to a modified tracked file using the `add` command. In this section, we go a step further and see how we can stage only a part of our modifications to the same file.

It's generally a good idea to associate a commit with a single bug fix or feature, as commits can then be used to separate different logical ideas. If you solved two bugs by changing parts of the same file and want those changes to appear in different commits, you can do so as follows.

To simplify the process, I'll add three lines at three different positions in the same file and view the changes that I've just added (Figure 6.21):

```
git diff
```

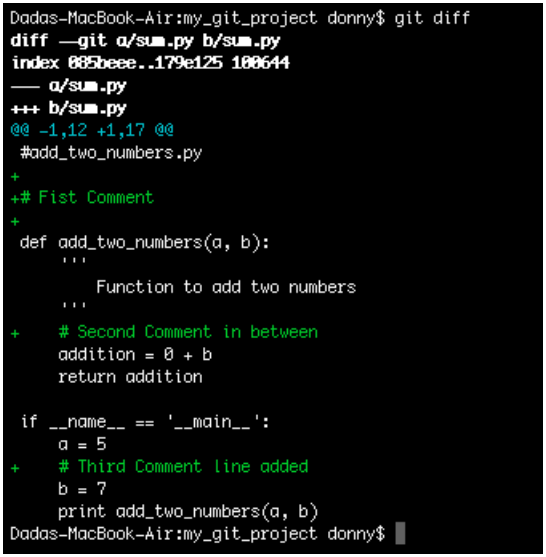


Figure 6.21. Adding three lines at different positions in a file

Let's say I want to add the second line among the three added lines to my commit. We can start the process with `git add`. Note the `-p` postfix to the `add` command to initiate this process (Figure 6.22):

```
git add -p
```

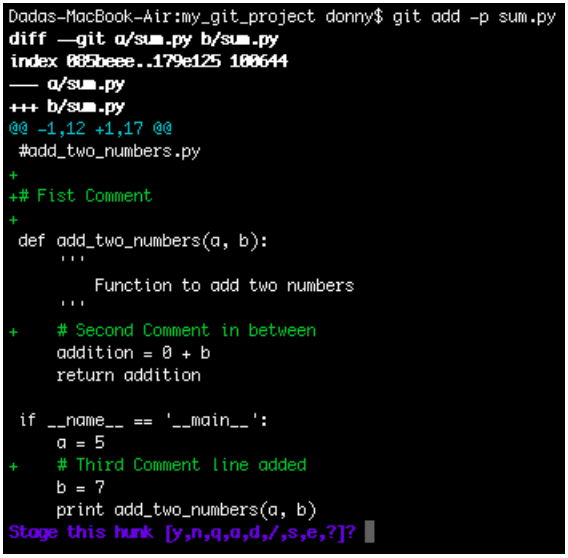


Figure 6.22. Initiate the process of adding part of a modified file

Git has clubbed all the changes together into a hunk. Notice that Git now asks us to enter an option. These are the options and their uses:

- y — stage the hunk
- n — don't stage the hunk
- e — edit the hunk
- d — exit the process
- s — split the hunk

In this case, we want to add only the second line, but because all three lines are a part of the same hunk, we need to split it (Figure 6.23):

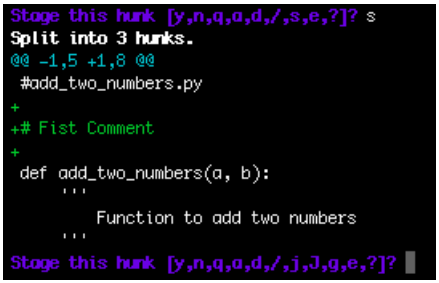


Figure 6.23. Smaller hunk after splitting the larger hunk

After splitting the larger hunk, we're provided the first of the three smaller hunks. We wish to add only the second one. Therefore, we go to the next one by selecting option `n` (Figure 6.24):

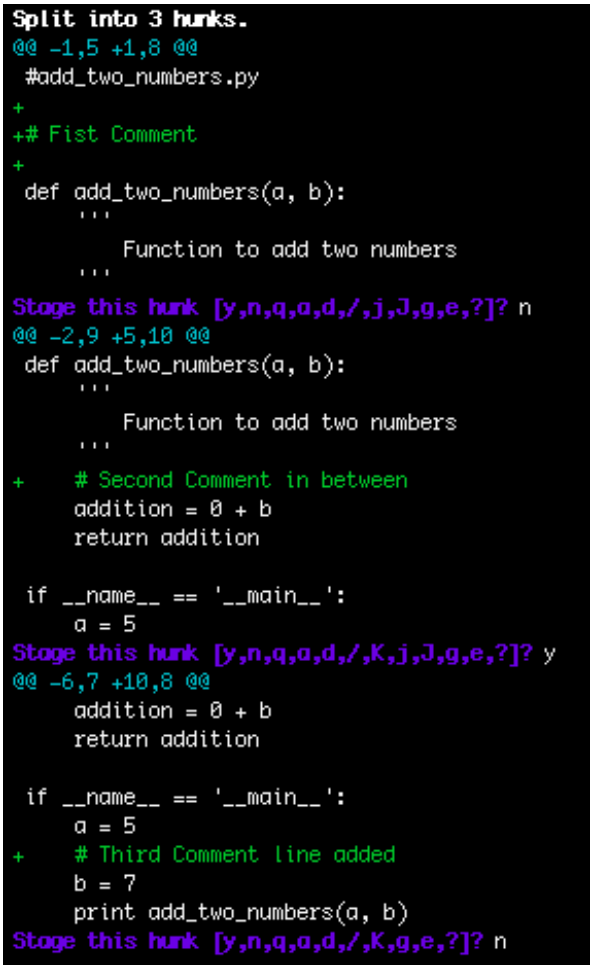


Figure 6.24. Adding the desired hunk

Next, we're asked if we want to stage the second line. Therefore, we select option `y`, followed by option `n` for the third line. Let's see how the status of the repository looks (Figure 6.25):

```
git status
```

```
Dadas-MacBook-Air:my_git_project donny$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   sum.py

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   sum.py
```

Figure 6.25. Status of the repository after staging part of a modified file

As you can see, the same file shows up in the list of modified files and in files staged for commit. This means that you successfully staged a part of a modified file. You can proceed to commit your changes now.

Don't Commit With the `-a` Option

After staging a part of a modified file, you shouldn't commit the changes by postfixing `-a`. This would add the rest of the modified file too!

Cherry Pick

Let's say our work is progressing in two branches. If you wanted to merge a single commit from one branch into another, `merge` or `rebase` won't suffice. The `cherry-pick` command allows you to pick a certain commit from a different branch and merge it into your current branch. Just like in merging and rebasing, `cherry-pick` can also result in conflicts, which should be resolved as discussed earlier.

How Does `cherry-pick` Differ From `merge` or `rebase`?

In `merge` or `rebase`, you join your current branch with a different branch. All the commits of the other branch—that have happened since it diverged from your branch—appear in your branch after the `merge`. However, as the name suggests,

> you can pick a single commit from a different branch and make it appear in your branch using `cherry-pick`.

The idea of a cherry-pick is illustrated in Figure 6.26:

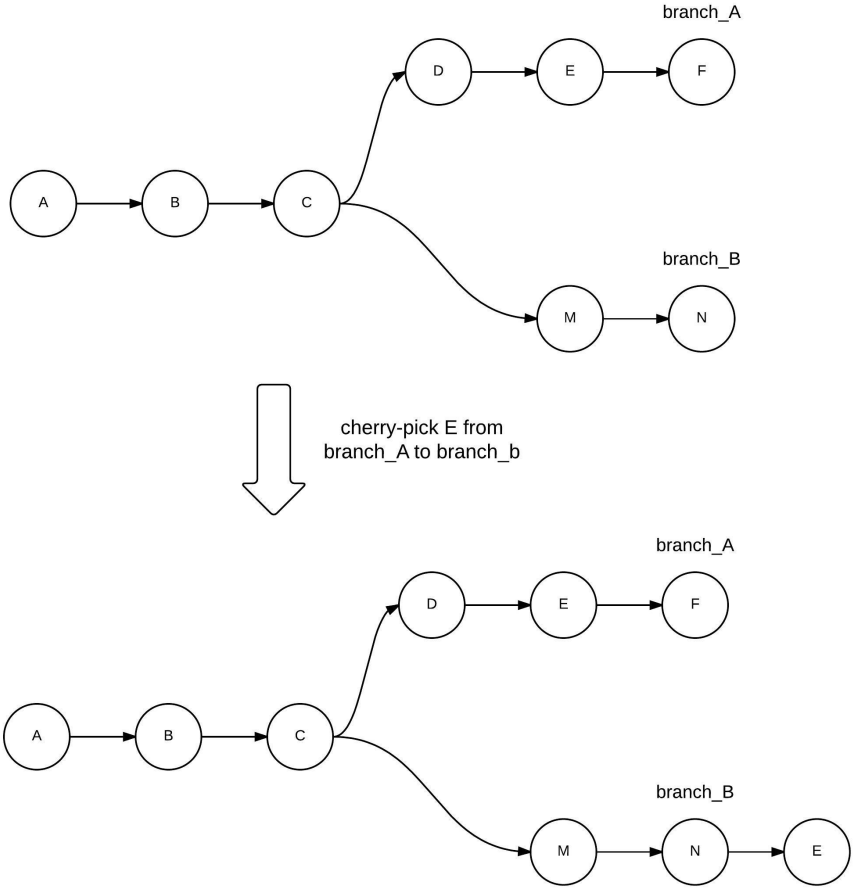


Figure 6.26. Illustration of `cherry-pick`

To merge a commit `30dc1fa2d` from a different branch to your current branch, run the following (Figure 6.27):

```
git cherry-pick 30dc1fa2d
```

```
SMA:my_git_project donny$ git log -2
commit 083e7eef5cde1625bcc78417e7b1f82c691875d2
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:55:12 2015 +0530

    Added yet another test

commit 49a6bec7c629e5a84e07c55301f2447f890bad4c
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:51:36 2015 +0530

    Added more tests
SMA:my_git_project donny$ git cherry-pick 30dc1fa2d
[master f36d753] Squashed last two commits
 Date: Sun May 17 00:35:33 2015 +0530
 3 files changed, 2 insertions(+), 4 deletions(-)
SMA:my_git_project donny$ git log -2
commit f36d753e80a769b6fc7735c5cd5d521146714e94
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 17 00:35:33 2015 +0530

    Squashed last two commits

commit 083e7eef5cde1625bcc78417e7b1f82c691875d2
Author: Shaumik <sdaityari@gmail.com>
Date:   Sun May 10 00:55:12 2015 +0530

    Added yet another test
SMA:my_git_project donny$
```

Figure 6.27. Illustrating the use of `cherry-pick` in a repository

Conclusion

What Have You Learned?

We've reached the end of another chapter! In this chapter, we discussed various commands and their uses to make your Git experience easier. Here's a list of the commands we covered:

- `log`
- `shortlog`
- `reflog`
- `fsck`
- `rebase`
- `stash`
- `add`
- `cherry-pick`

You should try to incorporate these into your daily workflow to gain the most out of them.

What's Next?

In the next chapter, we'll look at some GUI tools for Git, examining how they handle the commands we've already discussed.

Chapter 7

Git GUI Tools

Up till now, we've performed all our Git-related actions through the terminal, looking in detail at what each command does. The advantage of terminal commands is that they work across all platforms. In Chapter 1, I mentioned that there are various GUI tools that can be used instead of the terminal. Although GUI tools can appear to make life simpler, they work differently across the three major operating systems, which is why I've avoided their use so far.

In this chapter, we'll look at the GUI tools that serve as Git clients. First, we'll review GitHub Desktop[1], GitHub's own GUI tool, and then SourceTree[2], by Atlassian. Both of these applications have Mac and Windows versions, but neither supports Linux. Other popular GUI clients are Tower (Mac)[3], GitBox (Mac)[4], SmartGit (Windows, Mac, Linux)[5] and GitEye (Windows, Mac, Linux)[6]. All of these applications are either free or have free trial versions.

[1] https://desktop.github.com/
[2] https://www.sourcetreeapp.com/
[3] http://www.git-tower.com/
[4] http://www.gitboxapp.com/
[5] http://www.syntevo.com/smartgit/index.html
[6] http://www.giteyeapp.com/

GUI tools are an attractive option to many developers, as they provide an easy interface for managing a project with Git. Though we arguably gain a deeper understanding of Git by learning it through the command line, GUI tools have their place, especially in simple situations. One issue with using GUI tools is that it's easy to forget proper Git commands. This is problematic if you find yourself in an environment without GUI software, or if you need to run emergency commands from the command line, like working on a remote server. I suggest using a combination of GUI tools and the command line, utilizing the advantages of each.

I'll now look at GitHub Desktop and SourceTree in turn, evaluating their features and ease of use.

GitHub Desktop

Let's first take a look at the GUI client of GitHub itself. It supports both Windows and Mac. Previously, the Windows and Mac versions were different, but in August 2015, GitHub launched a new unified client, GitHub Desktop[7], for both platforms.[8]

After installation, you should add your GitHub account details.

Not Just for GitHub

You can manage other local Git repositories with GitHub Desktop too, but it's tailor made for GitHub repositories. Although a bit confusing, you can even manage Bitbucket repositories through the GitHub GUI tool[9]!

When you successfully log in to your account, all your repositories are linked to your GUI tool. On clicking the + button on the top left, you can see a list of your repositories under the **Clone** option (Figure 7.1):

[7] https://desktop.github.com/

[8] If you want to know about more about this tool, you can check its online documentation [https://help.github.com/desktop/].

[9] http://www.binarymoon.co.uk/2013/10/use-bitbucket-github-mac/

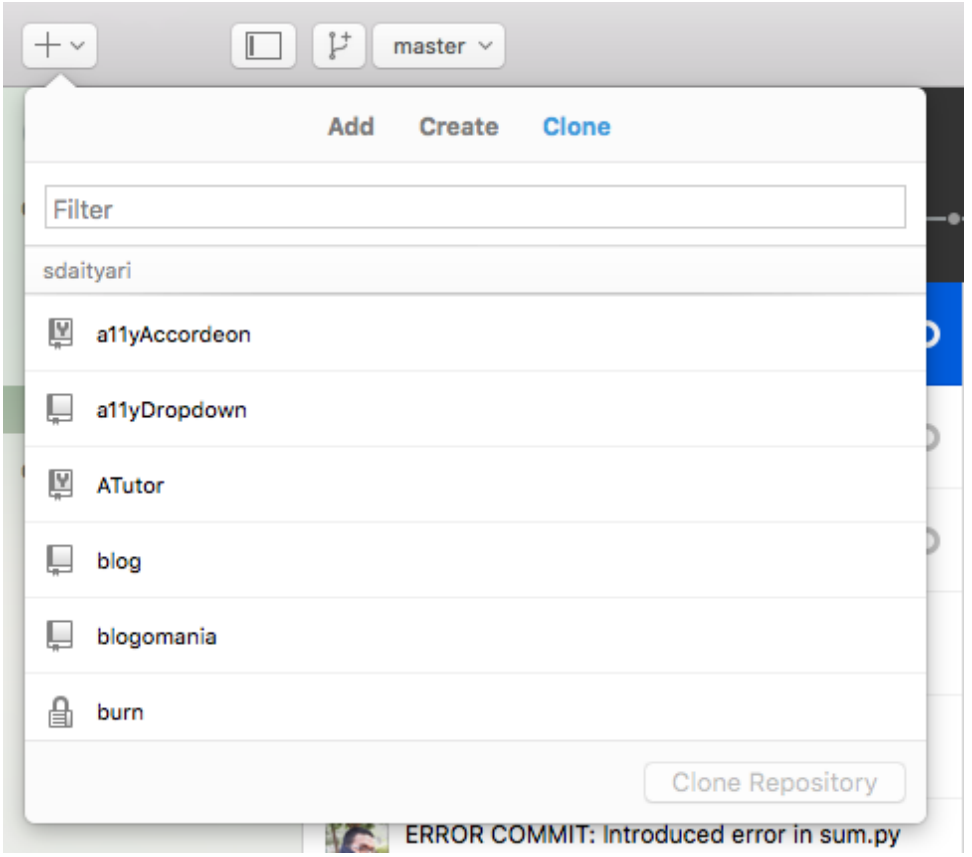


Figure 7.1. List of repositories to be cloned

Select the repository you want to clone, and click on **Clone Repository** to clone it.

Alternatively, you can add a local Git repository by choosing the **Add** instead of the **Clone** option. You're then asked to select the path to an existing Git repository on your local system (Figure 7.2):

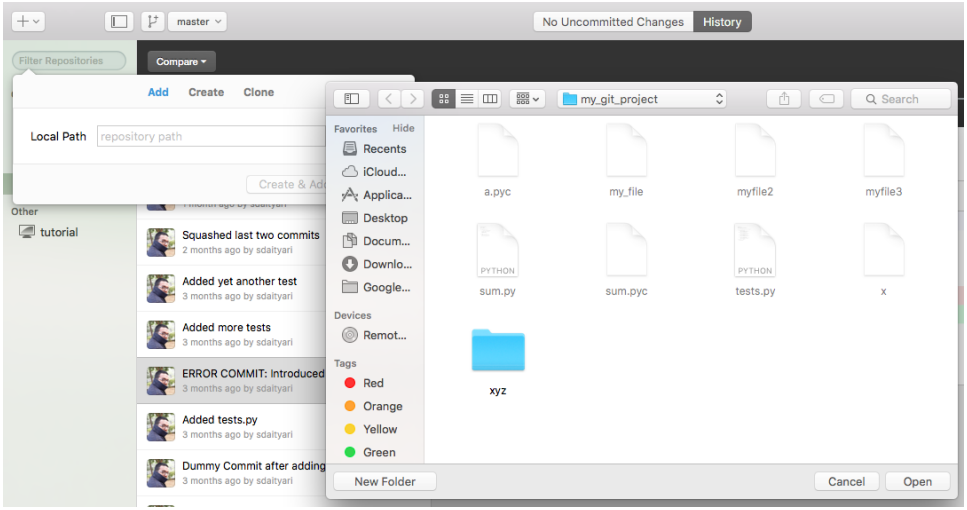


Figure 7.2. Add a local repository to be managed by GitHub Desktop

Once you've added your repository, you'll notice that it's now listed among the tracked repositories on the left of the window. If you added a GitHub repository, it will be listed under **GitHub**, whereas if you added a local repository, it will be listed under **Other**.

Tutorial Repository

There's a repository named "tutorial" listed under **Other**, which helps you get used to the features of the new GitHub Desktop. This is helpful if you were a user of the old GitHub GUI tools for Mac or Windows before the release of the unified desktop client.

Once a repository is selected, the commits in the current branch are listed. The UI resembles the GitHub website. If any commit is selected, the commit details are shown too. The workflow in the current branch is shown at the top.

On selecting the **History** tab at the top, you're shown the commits in the active branch (Figure 7.3). On selecting a specific commit, you're shown the changes that were made in that commit:

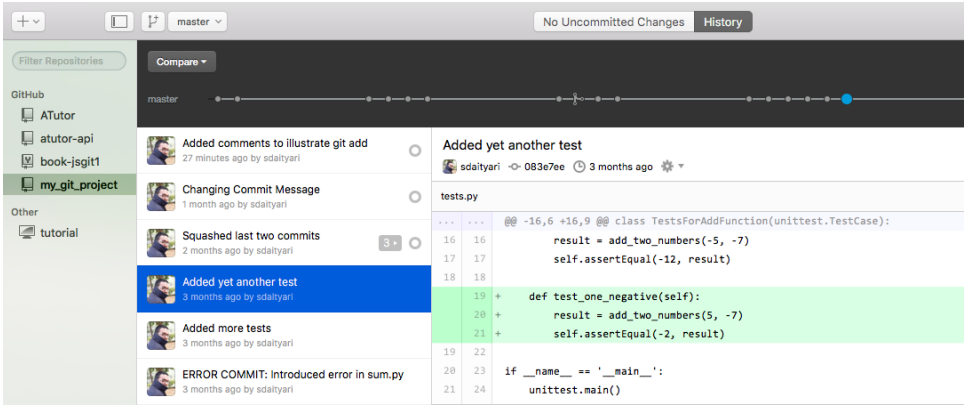


Figure 7.3. Listing the commits in the active branch

If you look at the workflow of the current branch at the top of the window, you can see that there's a **Compare** option, which enables you to compare your current branch with another. For instance, if we select `friend_branch`, we're shown the development of both branches with respect to each other. You can merge the branches by using the **Update from friend_branch** button.

Let's move on from comparing branches to creating or changing a branch. To create a branch, click on the Create New Branch button (an icon resembling a diverged workflow with a plus sign on top, as shown in Figure 7.4), and enter the name of the branch and the existing branch you want it to branch from:

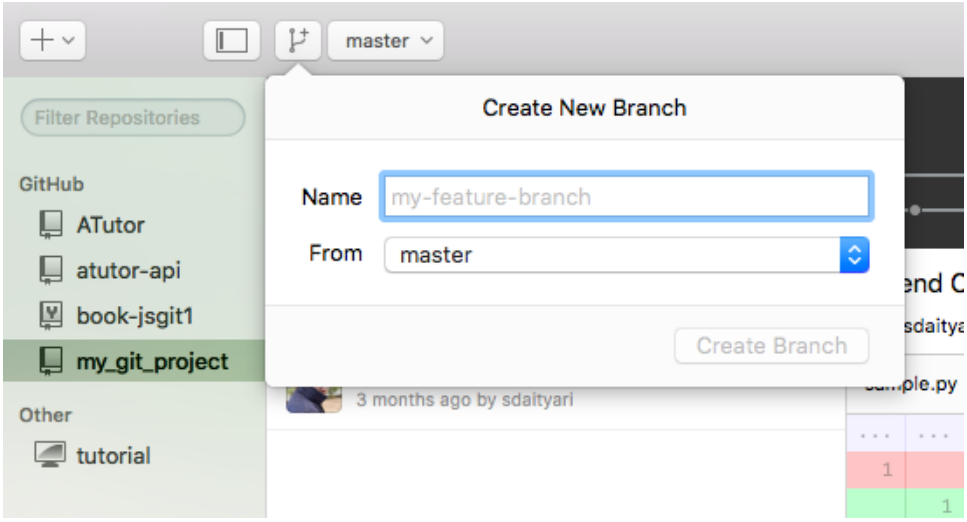


Figure 7.4. Creating a new branch

To change your current active branch to a different one, click on the button to the right of the new branch button, which also displays the name of your current active branch (Figure 7.5):

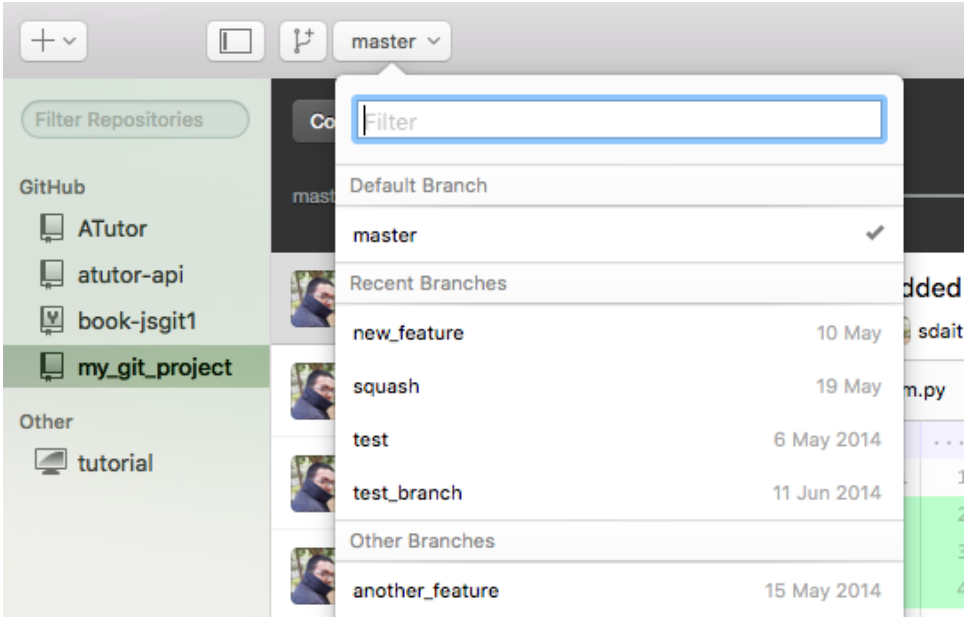


Figure 7.5. Changing the active branch

On the top right of the window there are **Pull Request** and **Sync** options. You can create a pull request from within the GUI client by first comparing two branches and then creating the pull request, just like you do on the GitHub website.

The Sync option, on the other hand, is interesting. It performs a pull and a push together, as it effectively "syncs" the commits in the local and remote.

When you select a commit, you can perform commit-level operations by selecting the gear option an the top right, as shown in Figure 7.6. However, as you'll see later, you have more options in the GUI tool we'll look at next, SourceTree.

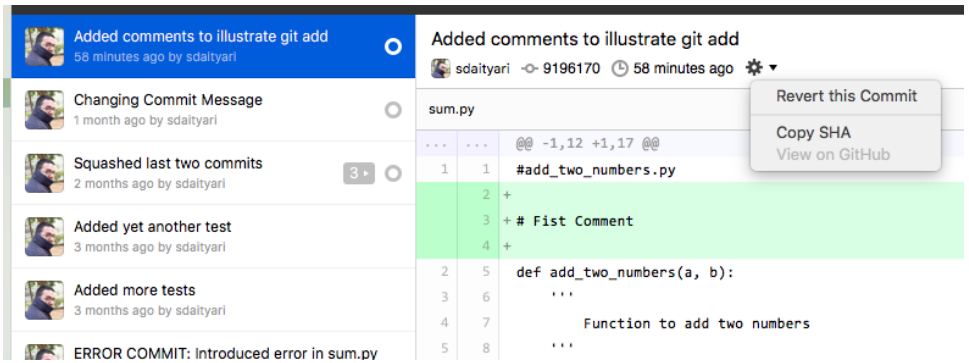


Figure 7.6. Commit options

On making any changes to the repository, the changes are visible by selecting the **Uncommitted Changes** tab at the top, shown in Figure 7.7. It lists the changes in the files, but note that there's no mention of the term "staging". You simply select the files you want to include in the commit and add a commit message before committing the changes. It makes the process simpler for beginners:

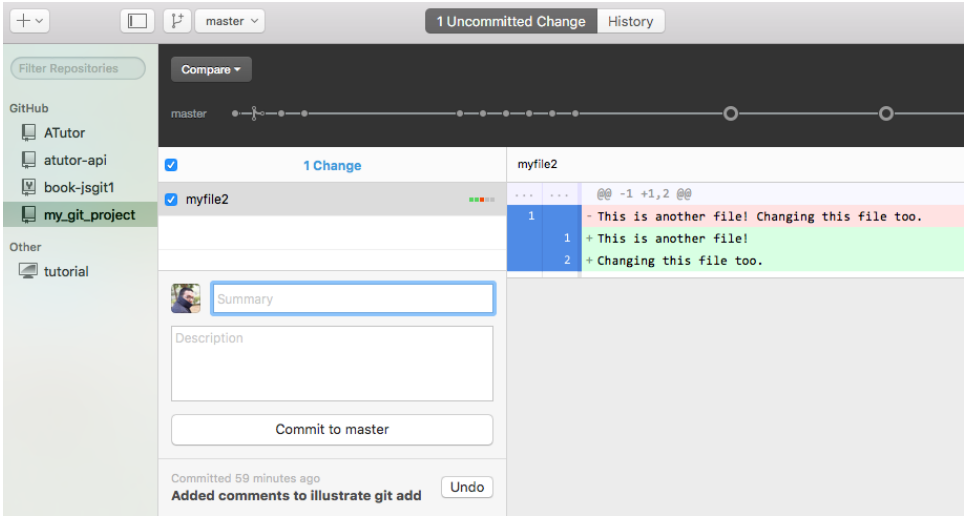


Figure 7.7. Committing changes in GitHub Desktop

GitHub Desktop tries to simplify the process of source code management, which is good for a beginner who's trying to learn Git. Let's now explore SourceTree, which has a wider range of functions.

SourceTree

SourceTree is a GUI client developed by Atlassian. It's compatible with repositories managed by both Git and Mercurial, another distributed VCS. SourceTree can use the version of Git already installed on your local system, or a version that's bundled with SourceTree itself. You can download and install the application from the SourceTree website[10].

SourceTree offers a wider range of features than GitHub's tool, and gives you more control over your repositories. Its various options also better match the corresponding terminal commands.

During installation, you're invited to add details of any accounts you hold at code sharing websites like GitHub and Bitbucket. If you skip this step, you can add accounts later by selecting **Settings** → **Add/Edit Account** (Figure 7.8):

[10] https://www.sourcetreeapp.com/

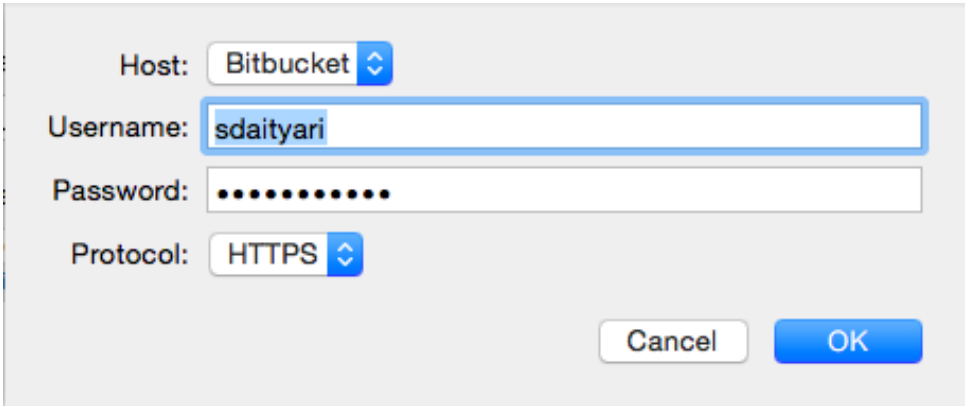


Figure 7.8. Adding a Bitbucket account in SourceTree

When adding a Bitbucket account, you're shown the list of repositories in your account, and likewise when adding a GitHub account. These repositories are listed under your **Remote** section, which you can see highlighted on the top left (Figure 7.9):

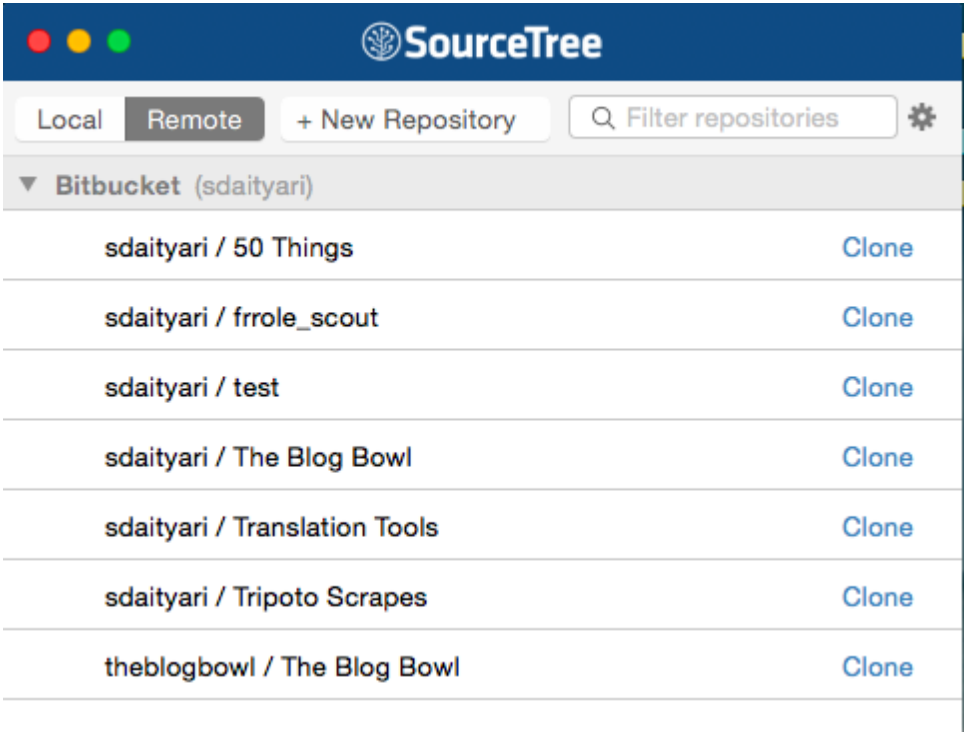


Figure 7.9. Repositories in Bitbucket

The repositories listed here are present only on the cloud, so they need to be cloned before you can start working on them locally. Click on the **Clone** link on the right of any repository to clone it (Figure 7.10):

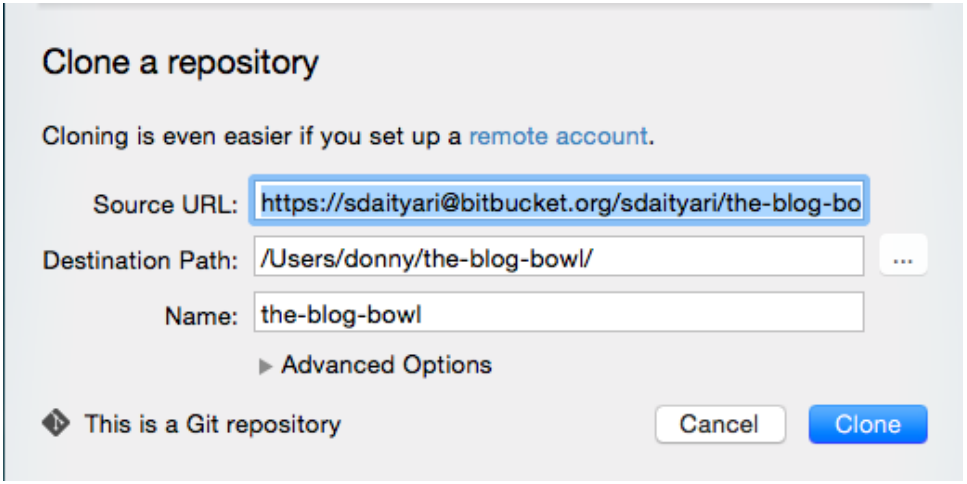


Figure 7.10. Confirming details of the repository to be cloned

After confirming the details, the remote repository is cloned to the location you specified in the last step.

Alternatively, you can add a local repository to SourceTree by clicking on the **+New Repository** button. Once you've added a repository, a new window opens with the details of the repository (Figure 7.11):

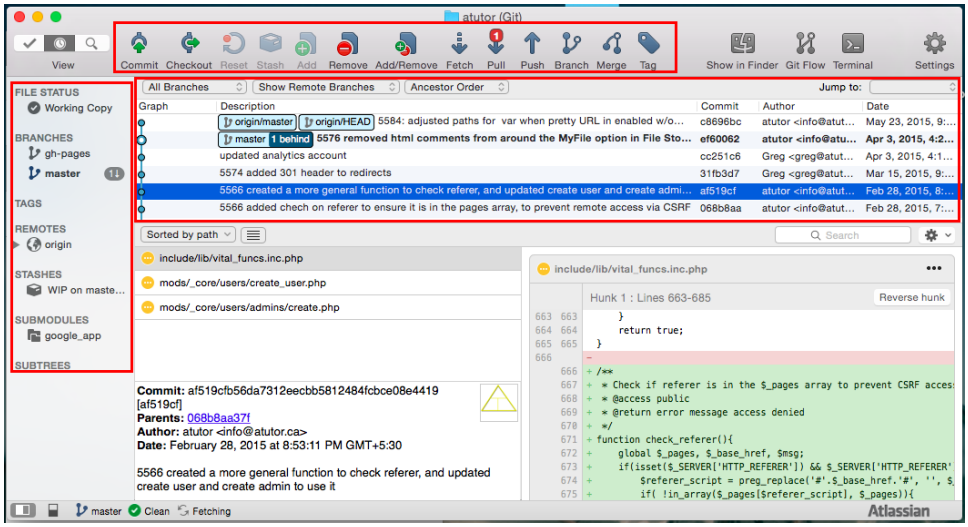


Figure 7.11. Repository details

As highlighted in the image above, the window has three parts—the top menu, the left menu and the central part. The top menu contains buttons that perform important actions in Git. The left menu lists the branches, remotes, stashes and submodules. The central part contains the list of commits in the active branch and the details of each commit.

If you look at the top menu (Figure 7.12), you'll notice that it contains the buttons to perform basic Git actions like commit, pull and push. There's also an option to open up a terminal in case you want to run a custom command.

Figure 7.12. SourceTree's top menu

The **Checkout** button helps you checkout to a new or an existing branch (Figure 7.13):

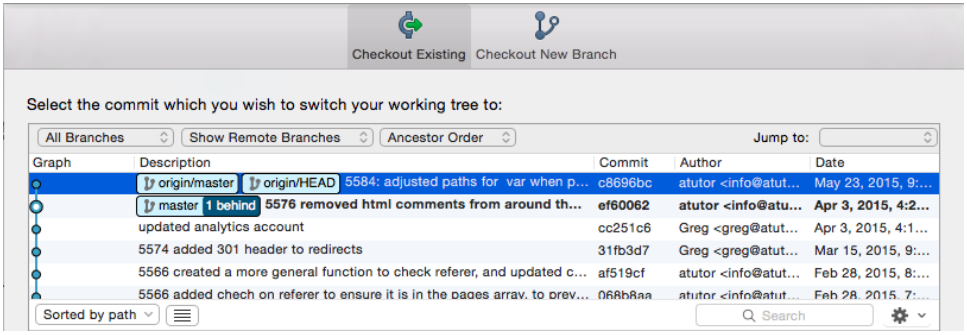


Figure 7.13. Checkout to new or existing branch

When you make changes to any file, the list of changed files pops up in the space for unstaged files (Figure 7.14). You can stage them by clicking the **Add** button on the top—after which they appear in the staged list. You can also removed staged files using the **Remove** button at the top.

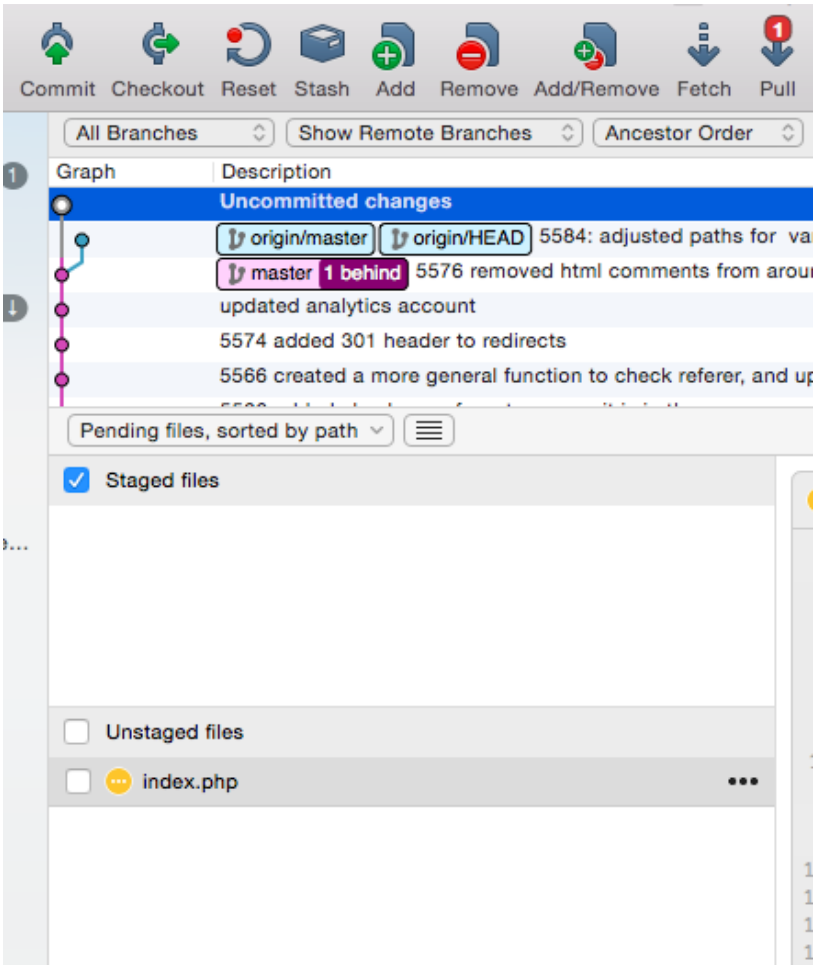


Figure 7.14. Staged and unstaged files following changes to your repository

Once you're ready to make a commit, click on the **Commit** button. For your first commit, you're asked to nominate a name and email address to be associated with your commits (Figure 7.15). This is similar to setting the global configuration settings through the terminal. From now on, your email address and name will be associated with this commit, as well as any future commits:

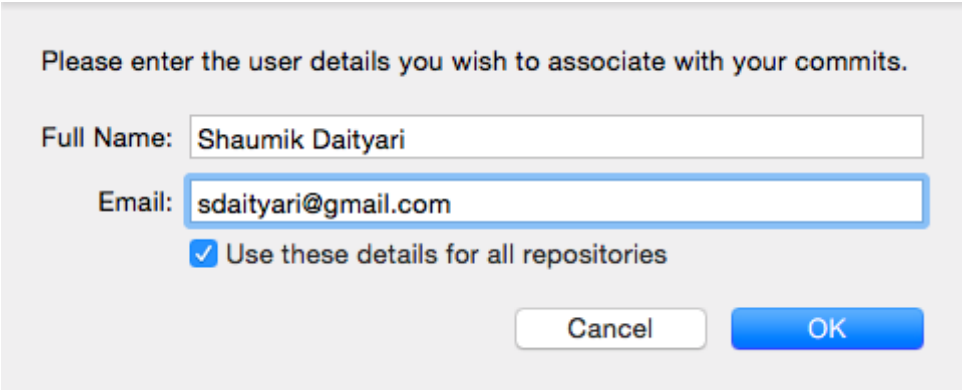


Figure 7.15. Adding name and email

After adding your name and email, you're asked to add a message describing your commit (Figure 7.16):

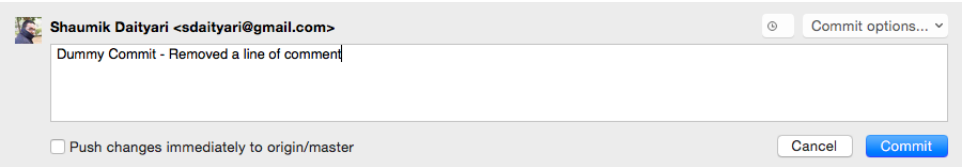


Figure 7.16. Adding a commit message

After a successful commit, notice the state of the repository and the change in the branch workflows: the blue color shows the current commit—which hasn't been merged with `origin/master`, denoted by purple (Figure 7.17):

Figure 7.17. Status of the repository after a commit

You can add or remove branches by clicking the **Branch** button in the top menu. You can force delete a branch even if it hasn't been merged yet, as shown in Figure 7.18 (which is analogous to the `-D` option in the terminal). You can merge branches through the **Merge** button in the top menu. If you want to merge `branch_A` into `branch_B`, make sure `branch_B` is active when you perform the merge operation.

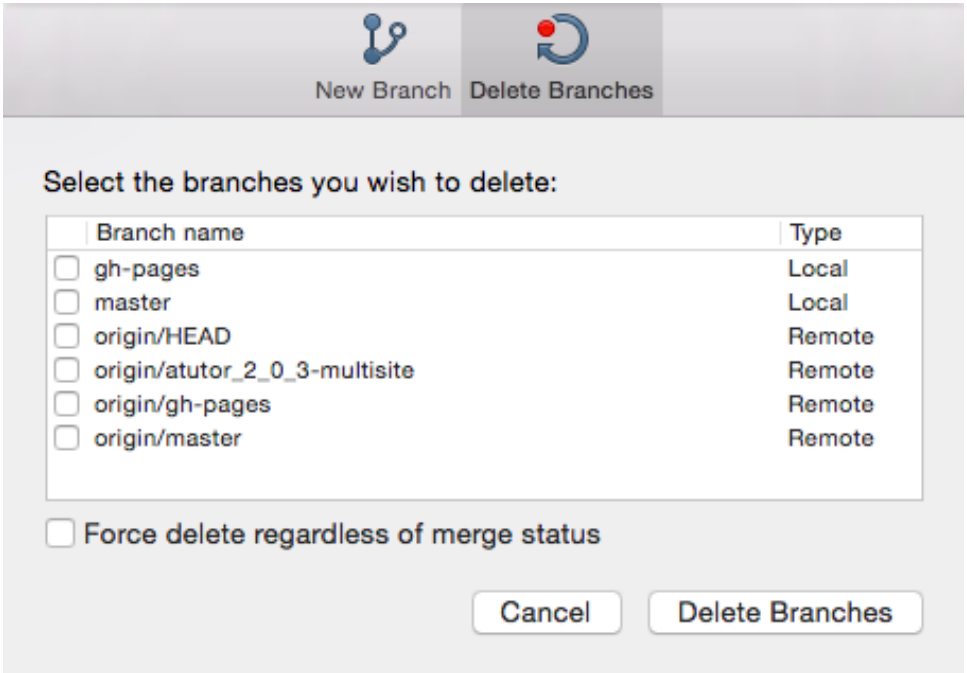


Figure 7.18. Branching in SourceTree: add and delete branches

Let's now have a look at the left menu, shown in Figure 7.19. It shows a list of branches, tags, remotes, stashes and submodules.

In this case, `master` and `gh-pages` are the two branches, and `origin` is the only remote. We also have one stash created on the `master` branch, which is shown in the screenshot below. SourceTree's stash option is a powerful, easy-to-use feature. You can apply any stash to your `HEAD`, with the option of keeping or removing the stash. Submodules are Git repositories within a parent repository. We haven't covered submodules in this book. This repository uses a `google_app` submodule.

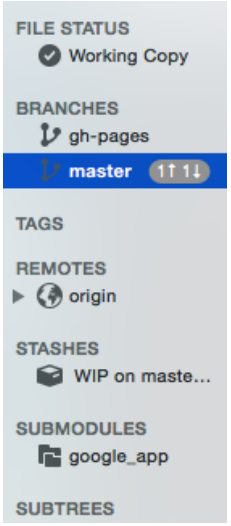


Figure 7.19. Left menu buttons

In addition, commit-based actions like checking out to the commit, cherry picking or creating a patch can be performed by right-clicking on a commit, as shown in Figure 7.20:

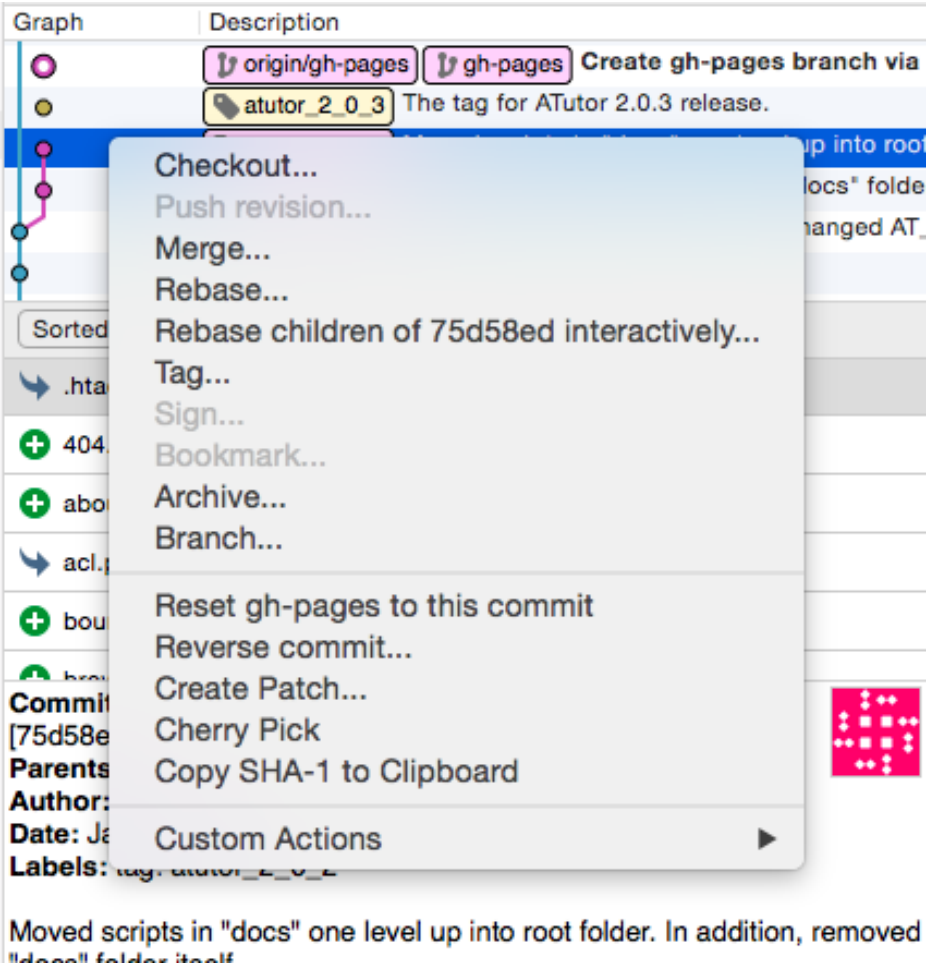


Figure 7.20. Specific operations for a commit

SourceTree Versus GitHub Desktop

Both SourceTree and GitHub Desktop are free to use.

SourceTree has a lot of features, with an information-rich display that directly relates to Git's terminal commands. Desktop, on the other hand, focuses more on bridging the gap between a local GitHub repository and the GitHub website, often substituting standard Git terms and processes with easier terms for beginners. It eases the process of hosting your repositories on GitHub, but makes it difficult—though not impossible—to host your repository elsewhere.

Finally, Desktop simplifies the whole process by cutting down on certain features, whereas SourceTree offers a fully-featured dashboard that might be overwhelming for beginners. I encourage you to try both GUI tools, in addition to a few more listed above, to work out which best suits your needs.

Conclusion

In this chapter, I reviewed two GUI tools for Git—SourceTree and GitHub Desktop.

GUI tools are definitely useful. The history of a project, with respect to the different branches, is easily visualized. Even when you're working on a project, it's useful to graphically analyze the changes you've made before committing them into the project history. Even when you're reviewing the work of others, it's a good idea of use a GUI tool to quickly review the changes.

Even though I find GUI tools to be great, if you're a beginner, I'd still recommend you learn the terminal commands first. As I mentioned above, GUI tools aren't cross platform, whereas terminal commands are. There's no single tool that works in Linux, Windows and OS X. Also, if you're working on a remote server (which is often a virtual machine), only command line tools can help you work with Git.

So for beginners and experienced users alike, I recommend using a combination of GUI tools and the terminal. Each has its pros and cons, which you'll discover through practice.

Chapter 8

Conclusion

As this book has guided you through the uses of Git, the focus has been on using it to manage a codebase. This is the most common use for Git, but certainly not the only one. In this concluding chapter, I'd like to discuss Git's meteoric rise, and then give you a glimpse of other innovative uses of Git, as well as its limitations, alternatives to Git, and what the future holds.

Git's Meteoric Rise

Back in 2009, over 57.5% of repositories used Subversion, whereas Git only had a 2.4% share of the SCM market, according to the Eclipse Community Survey[1]. In 2014, the same Eclipse Community Survey[2] showed that Git (at 33.3% market share) had surpassed Subversion (30.7%), as shown in (Figure 8.1). It's true that the figures are just an indication of global usage. But the Eclipse community represents a good sample, since it consists not just of open-source enthusiasts, but also of developers in the industry.

[1] http://www.eclipse.org/org/press-release/Eclipse_Survey_2009_final.pdf
[2] http://www.slideshare.net/IanSkerrett/eclipse-community-survey-2014

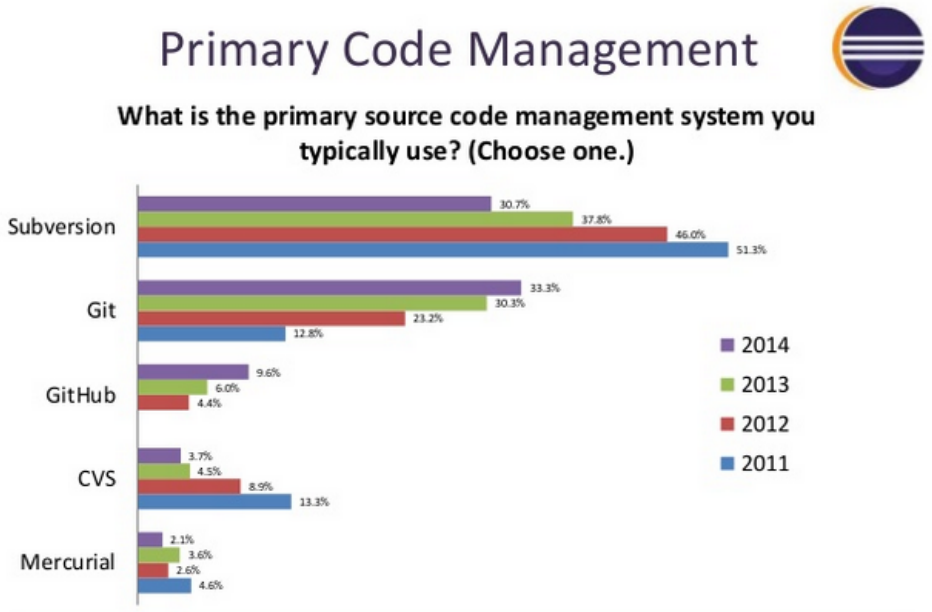


Figure 8.1. Trend of VCS usage over the years (Eclipse Community Surveys)

Google Search trends[3] indicate that Git and Subversion had roughly the same interest around the year 2011 (Figure 8.2). However, since then, the popularity of Subversion has declined, whereas that of Git has grown steadily.

[3] http://www.google.com/trends/explore#q=git,svn

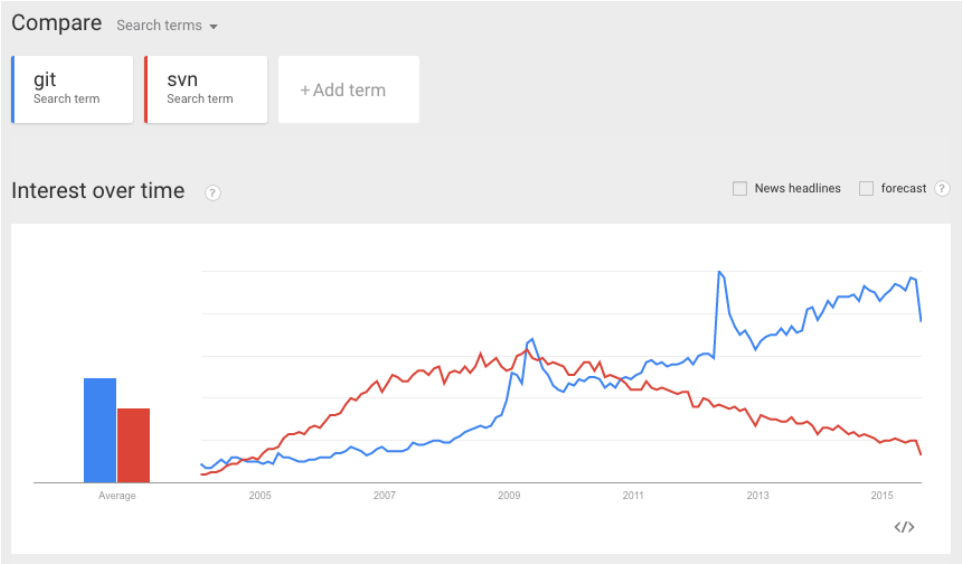


Figure 8.2. Google Search trends for Git and Subversion

Finally, if you look at job trends on indeed.com[4] (Figure 8.3), Git overtook subversion in early 2013, and there's been no turning back:

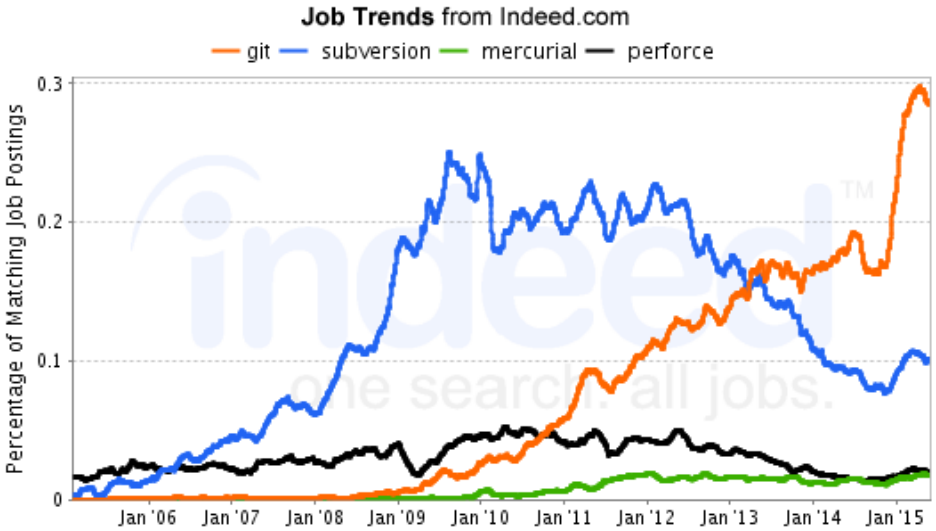


Figure 8.3. Git, Subversion, Mercurial and Perforce job trends

[4] http://www.indeed.com/jobtrends?q=git%2C+subversion%2C+mercurial%2C+perforce&l=

From all of these data sources, one thing is obvious: Git's meteoric rise proves it's doing the right thing. The future is definitely Git.

Could Git Fail?

Git is a great tool, and is often the first choice among developers. Naturally, organizations big and small choose Git to manage their projects, which is evident from the "Companies & Projects Using Git" section of the Git website[5]. But with widespread adoption, and being used in some very large projects, one of Git's major failings becomes evident: it doesn't manage very large repositories in the most efficient way. How large are we talking about here? Facebook large[6]. Facebook eventually shifted from Git to Mercurial, another distributed VCS. Let's look at why.

When engineers at Facebook extrapolated their growth in the near future, they found that file status operations in Git would become a major bottleneck, as Git examines each file for changes. With thousands of commits every day, it would have taken a few seconds to run even a `git status`. Integrating their own file monitor with Mercurial made for a much more efficient process, which is why Facebook shifted to Mercurial, and why their developers still contribute significantly to its development. However, even though Facebook shifted its main codebase to Mercurial, it's interesting that many of their important side projects like React[7] and RocksDB[8] are still managed through Git.

Prasoon Shukla, a Mercurial contributor, has described the differences between how Git and Mercurial work[9], and why Mercurial is more efficient when you scale to the size of Facebook.

In recent times, however, Git has made progress in the management of large repositories—both in terms of history, and the size of files in a repository. If you have a codebase with a very large history, you can perform a shallow clone, which enables you to clone only a specified number of latest commits. For instance, if you want to clone only the ten latest commits from our dummy project, you can specify `10` using the `--depth` option:

[5] http://git-scm.com/#companies-projects
[6] https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/
[7] http://facebook.github.io/react/
[8] http://rocksdb.org/
[9] http://blog.prasoonshukla.com/mercurial-vs-git-scaling

```
git clone --depth 10 https://github.com/sdaityari/my_git_project
```

Previously, Git had only limited support for shallow clones, especially if your shallow history wasn't long enough. You would often not be able to push from your shallow clone. However, recent versions (Git 1.9+) give you a greater ability to push and pull.

Another way of managing a large repository is to clone only a single branch. You can do so using the `--single-branch` option. To clone only the master branch of your dummy project, run the following:

```
git clone https://github.com/sdaityari/my_git_project --branch
➥ master --single-branch
```

Beyond Source Code Management

After reading this book, you hopefully feel very safe with Git. Once you create a commit, there's no way you can lose it (unless, of course, someone messes with the **.git** directory). You've seen the potential of Git. So isn't it natural that people are starting to use Git for tasks other than just managing code?

One very common example of using Git for a new purpose is to manage databases. There's no Git-like technology that tracks changes in a database. What developers have come up with is to take database dumps (which contain all the data in a database in the form of queries) and add them to a Git repository with regular commits. This enables one to track the changes in a database through the changes made in the dumps. With the algorithms used by Git to compress the data, this task doesn't use up as much space as you might imagine.

Git and MySQL

If you plan to back up your MySQL database, use the `mysqldump` command, with the options `-u` for username and `-D` for database. The `-p` prompts for a password for the user `my_user`.

```
mysqldump -u my_user -p -D my_database > dump_file
```

The `dump_file` that's created contains all the SQL queries needed to restore the database. Commit this file to Git and use the same command to update the contents of this file.

Git can also be useful for designers. Even though Photoshop or CorelDRAW files aren't comparable to source code, they can be tracked by Git. Design files are binary files, rather than meaningful text files like database dumps. Git doesn't easily recognize differences in two versions of these types of files, so the whole file is committed—which increases the size of repositories, making them more difficult to manage. While Git might remain practical for this purpose, the benefits are less significant. However, this hasn't discouraged enthusiastic designers from trying out Git[10].

Git's also being used in publishing. In fact, the team behind this book worked using Git too! I would create pull requests on GitHub, where the editors would suggest changes.

I mentioned in the first chapter that Google Docs is a good example of version control in action. There are applications built using Git on a similar premise of enabling change tracking. An example would be the WordPress plugin VersionPress[11], which tracks changes in a WordPress site using Git in the background.

The End

With this, we come to the end of the book. Although the book is ending, it's just the start of your journey. Get out there and do some amazing things with version control! I hope you've enjoyed reading the book as much as I've enjoyed writing it.

[10] http://courtnycotten.com/git-for-design/
[11] http://versionpress.net