Ferdinando Santacroce

Foreword by:

Arialdo Martini
Solution Architect with *Aduno Gruppe*

Giovanni Toraldo
Lead Software Developer, *Cloudesire.com*

Git Essentials

Second Edition

Create, merge, and distribute code with Git, the most powerful and flexible versioning system available

Packt

WOW! eBook
www.wowebook.org

Git Essentials

Second Edition

Create, merge, and distribute code with Git, the most powerful and flexible versioning system available

Ferdinando Santacroce

Packt

BIRMINGHAM - MUMBAI

WOW! eBook
www.wowebook.org

Git Essentials

Second Edition

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2015

Second edition: November 2017

Production reference: 1071117

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN 978-1-78712-072-3

www.packtpub.com

WOW! eBook
www.wowebook.org

Credits

Author
Ferdinando Santacroce

Reviewer
Giovanni Toraldo

Commissioning Editor
Aaron Lazar

Acquisition Editor
Denim Pinto

Content Development Editor
Vikas Tiwari

Technical Editor
Jijo Maliyekal

Copy Editor
Safis Editing

Project Coordinator
Ulhas Kambali

Proofreader
Safis Editing

Indexer
Francy Puthiry

Graphics
Kirk D'Penha

Production Coordinator
Shantanu Zagade

WOW! eBook
www.wowebook.org

Foreword

The book you are holding in your hands is not an ordinary one; drop it if you are not willing to start an exciting, lifelong, challenging, and, sometimes, demanding journey. Ferdinando's Git Essentials is definitely not in the *learn-git-in-2-days-without-effort* realm.

In fact, there are two approaches to Git.

Either you might be a typical, traditional--and, probably, slightly bored--developer, whose daily activities are more led by deadlines to meet and tasks to accomplish than by passion and challenges. Should this be the case, no worries; just ignore this book and rather ask your fellow to teach you the bare minimum of the Git commands (clone, checkout, commit, and merge) to let you survive in the future, and rely on Google or Stack Overflow in the case of needs.

Or chances are that you are a passionate, ardent, and pragmatic programmer, eagerly and humbly willing to question all your beliefs.

In this case, you will be excited to learn that Git is only outwardly just a very efficient and powerful tool for source code versioning; it is, in fact, a breaking point in the history of computer science, and this book is the perfect companion to discover this side of it.

I've got a few examples of similar breaking points in the history of IT. Besides the World Wide Web, as the most trivial example, I can think of Unix, with its "*Do one thing and do it well*" philosophy and infinitely composable commands, Kent Beck's TDD or Emacs, with its astonishing and endless extension possibilities, and Docker, which made infrastructure-as-code available to the masses and made the DevOps movement explode.

Meet one of the preceding technologies or ideas and be sure that nothing in your developer's life can be the same anymore.

I believe that the same is true for Git, which you are going to learn with the help of this book.

Differently from most other versioning systems, Git builds its design on the core idea that the development activity is inherently a communication and collaboration exercise. Of course, you might decide to use Git just as Subversion or TFS on steroids, but you would miss its beauty--Git builds on some very simple, yet powerful, concepts that enable developers to build scalable and collaborative networks.

WOW! eBook
www.wowebook.org

Git belongs to the core pillars of the open source movement: Richard Stallman, Eric S. Raymond, and other giants who provided the ideological foundation. The Free Software Foundation and licenses such as the GPL granted the needed legal authority. GNU and Linux are the very concrete and working implementations of those ideas and Git, with its social concepts in its DNA, is the fourth pillar--it is the very tool that enabled the community to build networks such as GitHub, which made the open source successful on a global scale.

So, if you are a passionate programmer and want to enter the social dimension of Git and the universe that revolves around it, learning the bare commands may be reductive. You may need to capture the deep meaning of what a Fork is, of how to manage a Pull Request, and of what a Rebase is, besides its technicalities.

And to achieve this, you may need the expert guide of someone as passionate as you are.

I believe that's exactly the value of the book you are reading--it is neither a replacement for the Git's man pages nor a quick shortcut for learning Git without effort. On the contrary, being a vast tutorial that forces you to get your hands dirty, it will carry you along a deep, not necessarily easy but always exciting, journey.

This second edition completes the only possible topic that was not covered in the first edition--it bravely takes you deep into the heart of Git's internals so that the moment you reemerge, you can reason about all the imaginable weird behaviors even while working with superficial commands.

As for the first edition, I must confirm that Git Essentials is a book that values code much more than words.

Crack open a shell, get ready, and enjoy your journey!

Arialdo Martini
Solution Architect with *Aduno Gruppe*

WOW! eBook
www.wowebook.org

Foreword

It's a pleasure for me to write the foreword to this book. It will guide beginners and accustomed users in the path of proficient usage of the tool that has nowadays become the de facto standard for source code versioning: Git.

Seven years ago, Git was emerging from the struggles that Linus Torvalds was facing while managing a global scale team working to the Linux kernel. A lot of competitors arose in the distributed version control systems scene, but finally, Git became widespread, thanks to its simplicity. A lot of companies adopted it--GitHub, for instance, revolutionized the way people contribute to open source projects.

If you are a professional software developer or an open source contributor, or both, you just can't ignore Git. You cannot hope that the first GUI on top of it will suffice to do your work. This book, carefully crafted by Ferdinando, will help you to understand the basics through the most advanced features that the Git toolbox has to offer. Your workflow will surely benefit from having everything tracked down; you can easily move through the history of your projects like in a time-travel machine.

The first edition readers will discover a refreshed experience in this book, lighting up the inner working gears of Git and the best workflows to adopt in every software project.

After this book, you will never again lose any precious hours of your work, I bet!

Giovanni Toraldo
Lead Software Developer, *Cloudesire.com*

WOW! eBook
www.wowebook.org

About the Author

Ferdinando Santacroce is a developer, author, and trainer who loves learning new things. As a software developer, Ferdinando has mainly worked on the .NET platform using C#, bridging the gap between old-style systems and new technologies. Over the span of his career, he has allowed some COBOL applications to talk to remote services, databases, and electronic devices such as cash handlers, scanners, and electronic shelf labels.

At the moment, he is committed to helping the largest energy player in Italy to face new challenges in the market by developing Java and JavaScript-based applications.

Other than this, he's focusing on continuous improvement and agile movement, which he follows with great care, in conjunction with XP foundations and lean manufacturing. He's one of the organizers of Italian Agile Days, the most famous and appreciated conference in the Italian Agile panorama.

Ferdinando loves to share ideas with other professionals and to speak at public conferences; every time he has, he has learned something new.

He enjoys writing as well. After a hiatus of a few years, he has started blogging again about his work and passions, which, according to him, are more or less the same thing.

Most of what he has learned over the years has been with the help of his friends and colleagues. Other than working within the same team or on the same code base, they have encouraged him to read books and attend inspiring conferences, such as XP Days, Italian Agile Days, and others that have helped his growth.

WOW! eBook
www.wowebook.org

About the Reviewer

Giovanni Toraldo is a Linux and open source enthusiast working as a lead software developer at cloudesire.com, a cloud marketplace start-up based in Pisa, Italy. He wrote the *OpenNebula 3 Cloud Computing* book, and reviewed *Mastering Redmine, Gitolite Essentials*, and the first edition of *Git Essentials*. His hobby is shooting with a heavy crossbow in a local medieval reenactment association.

WOW! eBook
www.wowebook.org

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.comand as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Mapt

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

WOW! eBook
www.wowebook.org

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1787120724`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

WOW! eBook
www.wowebook.org

Table of Contents

Preface 1

Chapter 1: Getting Started with Git 5

- **Foreword to the second edition** 6
- **Installing Git** 7
  - Installing Git on GNU-Linux 7
  - Installing Git on macOS 8
  - Installing Git on Windows 13
- **Running our first Git command** 21
  - Making presentations 23
  - Setting up a new repository 23
  - Adding a file 25
  - Committing the added file 26
  - Modifying a committed file 27
- **Summary** 31

Chapter 2: Git Fundamentals - Working Locally 32

- **Digging into Git internals** 32
- **Git objects** 32
  - Commits 36
    - The hash 36
    - The author and the commit creation date 36
    - The commit message 36
    - The committer and the committing date 37
    - Going deeper 37
    - Porcelain commands and plumbing commands 39
  - Trees 40
  - Blobs 40
- **Even deeper - the Git storage object model** 42
- **Git doesn't use deltas** 45
- **Wrapping up** 47
- **Git references** 48
  - It's all about labels 49
  - Branches are movable labels 50
  - How references work 50
    - Creating a new branch 52
    - HEAD, or you are here 53

WOW! eBook
www.wowebook.org

Reachability and undoing commits 56
Detached HEAD 61
The reflogs 64
Tags are fixed labels 66
Annotated tags 68
Staging area, working tree, and HEAD commit 70
The three areas of Git 77
Removing changes from the staging area 77
File status lifecycle 81
All you need to know about checkout and reset 83
Git checkout overwrites all the tree areas 84
Git reset can be hard, soft, or mixed 85
Rebasing 89
Reassembling commits 90
Rebasing branches 96
Merging branches 101
Fast forwarding 104
Cherry picking 108
Summary 112
Chapter 3: Git Fundamentals - Working Remotely 113
Working with remotes 113
Clone a local repository 114
The origin 115
Sharing local commits with git push 117
Getting remote commits with git pull 119
How Git keeps track of remotes 122
Working with a public server on GitHub 123
Setting up a new GitHub account 123
Cloning the repository 125
Uploading modifications to remotes 127
What do I send to the remote when I push? 129
Pushing a new branch to the remote 130
The origin 130
Tracking branches 131
Going backward – publishing a local repository to GitHub 132
Adding a remote to a local repository 134
Pushing a local branch to a remote repository 135
Social coding - collaborating using GitHub 135
Forking a repository 135
Submitting pull requests 138
Creating a pull request 139
Summary 144

WOW! eBook
www.wowebook.org

Chapter 4: Git Fundamentals - Niche Concepts, Configurations, and Commands 145

Dissecting Git configuration 145
Configuration architecture 145
Configuration levels 146
System level 147
Global level 147
Repository level 147
Listing configurations 148
Editing configuration files manually 148
Setting up some other environment configurations 148
Basic configurations 148
Typos autocorrection 149
Push default 149
Defining the default editor 150
Other configurations 151
Git aliases 151
Shortcuts to common commands 151
Creating commands 152
git unstage 152
git undo 152
git last 152
git difflast 153
Advanced aliases with external commands 153
Removing an alias 154
Aliasing the git command itself 154
Useful techniques 154
Git stash - putting changes temporally aside 154
Git commit amend - modify the last commit 159
Git blame - tracing changes in a file 159
Tricks 162
Bare repositories 162
Converting a regular repository to a bare one 162
Backup repositories 163
Archiving the repository 163
Bundling the repository 163
Summary 164

Chapter 5: Obtaining the Most - Good Commits and Workflows 165

The art of committing 165
Building the right commit 166
Making only one change per commit 167
Splitting up features and tasks 167

WOW! eBook
www.wowebook.org

Writing commit messages before starting to code 170
Including the whole change in one commit 170
Describing the change, not what have you done 171
Don't be afraid to commit 171
Isolating meaningless commits 172
The perfect commit message 172
Writing a meaningful subject 173
Adding bulleted details lines when needed 173
Tying other useful information 174
Special messages for releases 174
Conclusions 174
Adopting a workflow - a wise act 174
Centralized workflows 175
How they work 175
Feature branch workflow 176
Gitflow 176
Master branch 178
Hotfixes branches 178
The develop branch 178
The release branch 179
The feature branches 179
Conclusion 180
GitHub flow 180
Anything in the master branch is deployable 181
Creating descriptive branches off of master 181
Pushing to named branches constantly 181
Opening a pull request at any time 182
Merging only after pull request review 182
Deploying immediately after review 182
Conclusions 183
Trunk-based development 184
Other workflows 184
Linux kernel workflow 185
Summary 186
Chapter 6: Migrating to Git 187
Before starting 187
Installing a Subversion client 187
Working on a Subversion repository using Git 188
Creating a local Subversion repository 188
Checking out the Subversion repository with the svn client 188
Cloning a Subversion repository from Git 190
Adding a tag and a branch 190
Committing a file to Subversion using Git as a client 191
Retrieving new commits from the Subversion server 191

WOW! eBook
www.wowebook.org

Using Git with a Subversion repository 192
Migrating a Subversion repository 192
Retrieving the list of Subversion users 193
Cloning the Subversion repository 193
Preserving ignored files 194
Pushing to a local bare Git repository 194
Arrange branches and tags 194
Renaming trunk branch to master 195
Converting Subversion tags to Git tags 195
Pushing the local repository to a remote 195
Comparing Git and Subversion commands 195
Summary 197
Chapter 7: Git Resources 198
Git GUI clients 198
Windows 198
Git GUI 199
TortoiseGit 200
GitHub for Windows 200
Atlassian SourceTree 201
Cmder 203
macOS 203
Linux 204
Building up a personal Git server with web interface 205
SCM Manager 205
Learning Git in a visual manner 206
Git on the internet 208
Git for human beings Google Group 208
Git community on Google+ 209
Git cheat sheets 209
Online videos 209
Ferdinando Santacroce's blog 209
Summary 210
Index 211

WOW! eBook
www.wowebook.org

Preface

If you are reading this book, you are probably a software developer and a professional. What makes a professional a good one? His culture and his experience, sure, but there's more--a good professional is one who masters different tools, can choose the best tool for the job, and has the necessary discipline to develop good working habits.

Version control is one of the base skills for developers, and Git is one of the right tools for the job. However, Git is not a screwdriver, a simple tool with only a base function; Git provides a complete toolbox for managing your own code, within which there are also sharp tools that should be handled with caution.

The ultimate aim of this book is to help readers start using Git and its commands in the safest way, getting things done without injuries. Having said this, you will not get the most from Git commands if you do not acquire the right habits; as with other tools, in the end, it is the craftsman who makes the difference.

This is a book to be read in front of a computer; compared with the first edition, there are many more commands, examples, and exercises to test; in the first four chapters, we will learn by doing.

The book will cover all the basic Git topics, allowing readers to start using it even if they have little or no experience with versioning systems; they only need to know about versioning in general, so reading the related Wikipedia page will be sufficient.

What this book covers

`Chapter 1`, *Getting Started with Git*, shows the reader all the (simple) steps they need in order to install Git and do their first commit.

`Chapter 2`, *Git Fundamentals - Working Locally*, discusses how working locally reveals the essence of Git, how it takes care of your files, and how you can manage and organize your code.

`Chapter 3`, *Git Fundamentals - Working Remotely*, covers how working remotely moves your attention to the collaborating side of the tool, explaining the basic commands and options you use when working with remote repositories one by one.

WOW! eBook
www.wowebook.org

`Chapter 4`, *Git Fundamentals - Niche Concepts, Configurations, and Commands*, focuses on niche concepts and commands, completing the basic set of Git commands you need to know, giving the reader some more weapons to use in difficult situations.

`Chapter 5`, *Obtaining the Most - Good Commits and Workflows*, gives the reader some hints about common ways to organize source code within Git, helping them to develop good habits every developer should adopt.

`Chapter 6`, *Migrating to Git*, takes you through a way to give a hand to developers who use other versioning system, such as Subversion, to manage the transition phase into Git.

`Chapter 7`, *Git Resources*, offers some hints, based on the author's personal experience, which could be of interest to the reader.

What you need for this book

To follow the examples used in this book and get some practice using Git, you only need a computer and a valid Git installation. Git is available for free for every platform (Linux, Windows, and macOS).

The examples are based on the latest version of Git for Windows at the time of writing, which is 2.11.0.

Who this book is for

This book is mainly for developers. The book does not require experience in a particular programming language, nor does it require wide experience as a software developer. You will be able to read this book easily if you already use another versioning system, but this could even be your first time using a versioning tool if you at least know the basics of versioning.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

WOW! eBook
www.wowebook.org

Any command-line input or output is written as follows:

```
$ git log --oneline
```

Any command-line input or output is written as follows:

```
$ mkdir css
$ cd css
```

New terms and **important words** are shown in bold, while *important sentences* are shown in *italics*.

Warnings or important notes appear in a box like this.

Tips are rimmed:

Git log is your best friend: use it whenever you have to look at a repository history.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

WOW! eBook
www.wowebook.org

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/GitEssentialsSecondEdition_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

WOW! eBook
www.wowebook.org

1
Getting Started with Git

Whether you are a professional or an amateur developer, you've likely heard about the concept of version control. You may know that adding a new feature, fixing a broken one, or stepping back to a previous condition is a daily routine.

This requires the use of a powerful tool that can help you take care of your work, allowing you to move around your project quickly and without friction.

There are many tools for this job on the market, both proprietary and open source. Usually, you will find **Version Control Systems** (**VCS**) and **Distributed Version Control Systems** (**DVCS**). Some examples of centralized tools are **Concurrent Version System** (**CVS**), **Subversion** (**SVN**), **Team Foundation Server** (**TFS**), and **Perforce Helix**. While in DVCS, you can find **Bazaar**, **Mercurial**, and **Git**. The main difference between the two families is the constraint—in the centralized system—to have a remote server from which to get and in which to put your files; needless to say, if the network is down, you are in trouble. In DVCS, on the other hand, you can either have or not have a remote server (even more than one), but you can work offline, too. All your modifications are locally recorded so that you can sync them at some other time. Today, Git is the DVCS that has gained more public favor than others, growing quickly from a niche tool to mainstream.

Git has rapidly grown as the de facto source code versioning tool. It is the second famous child of **Linus Torvalds**, who, after creating the **Linux** kernel, forged this versioning software to keep track of his million lines of code.

In this first chapter, we will start at the very beginning, assuming that you do not have Git on your machine. This book is intended for developers who have never used Git or only used it a little bit, but who are scared to throw themselves headlong into it.

WOW! eBook
www.wowebook.org

If you have never installed Git, this is your chapter. If you already have a working Git box, you can quickly read through it to check whether everything is alright.

Foreword to the second edition

Welcome to the second edition of Git Essentials!

This paragraph is dedicated to those who have already read the first edition; here you will find an overview of the changes and new things inside this brand-new edition.

First of all, we listened to your feedback: in `Chapters 2`, *Git Fundamentals - Working Locally* and `Chapter 3`, *Git Fundamentals - Working Remotely* we will look at some technical details in more depth, describing more accurately the internals of Git; this entails extra effort on the part of the reader, but in return, he or she will obtain a more mindful understanding of Git architecture that will later help to grasp the commands of this powerful tool.

The Git ecosystem made some really giant steps forward since April 2015, but at its heart, Git is always the same. Here is an incomplete list of new features and improvements:

- Big improvements for Windows (for example, a fully working credential subsystem, performance enhancements, and so on—see `https://github.com/git-for-windows/git`).
- Git Large File Storage (LFS)-an additional tool from GitHub friends (see `https://git-lfs.github.com`).
- Git Virtual File System from Microsoft fellows (see `https://github.com/Microsoft/GVFS`).
- `git worktree` command and functionalities. Worktrees are a feature that was first included in Git 2.5; they let you check out and work on multiple repository branches in different directories simultaneously-see `https://git-scm.com/docs/git-worktree`.
- A lot of improvements and new options for common commands, too many to cite them all.

So the aim of this book is to get started with versioning and learn how to do it proficiently.

Let's start!

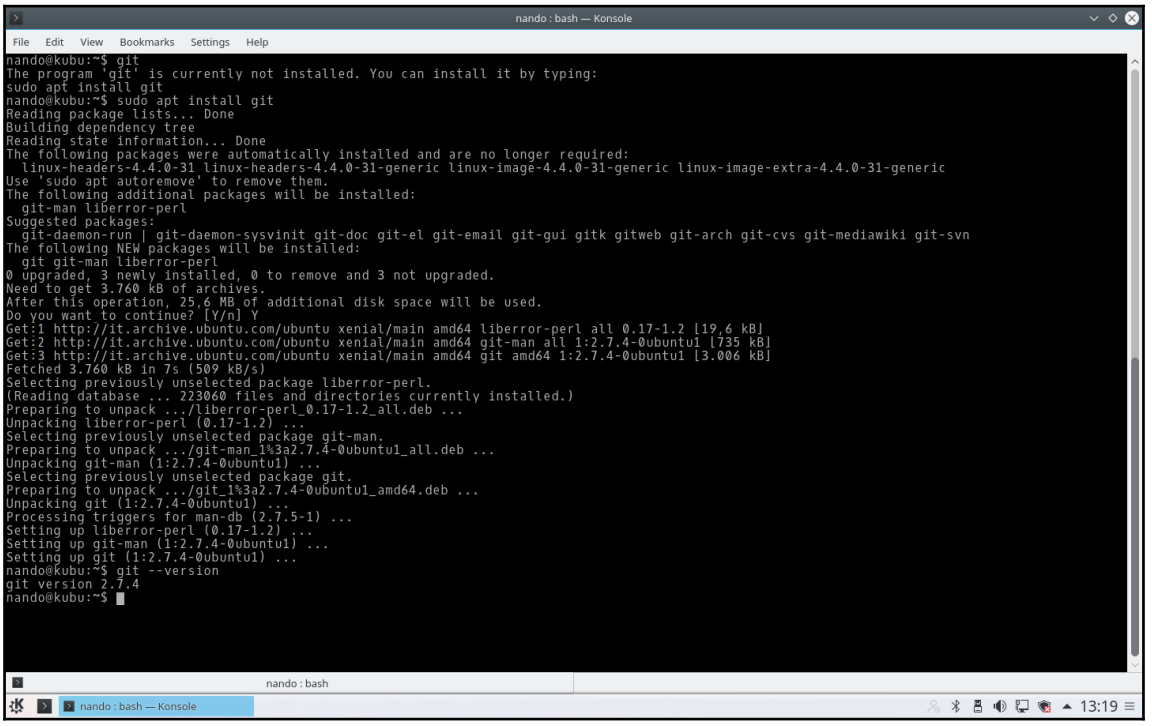
WOW! eBook
www.wowebook.org

Installing Git

Git is open source software. You can download it for free from `http://git-scm.com`, where you will find a package for all the most common environments (GNU-Linux, macOS and Windows). At the time of writing this book, the latest version of Git is 2.11.0.

Installing Git on GNU-Linux

If you are a Linux user, you may have Git out of the box.
If not, you can use the distribution package manager to download and install it; an `apt-get install git` command or equivalent will provide you Git and all the necessary dependencies in seconds, as shown in the following image:

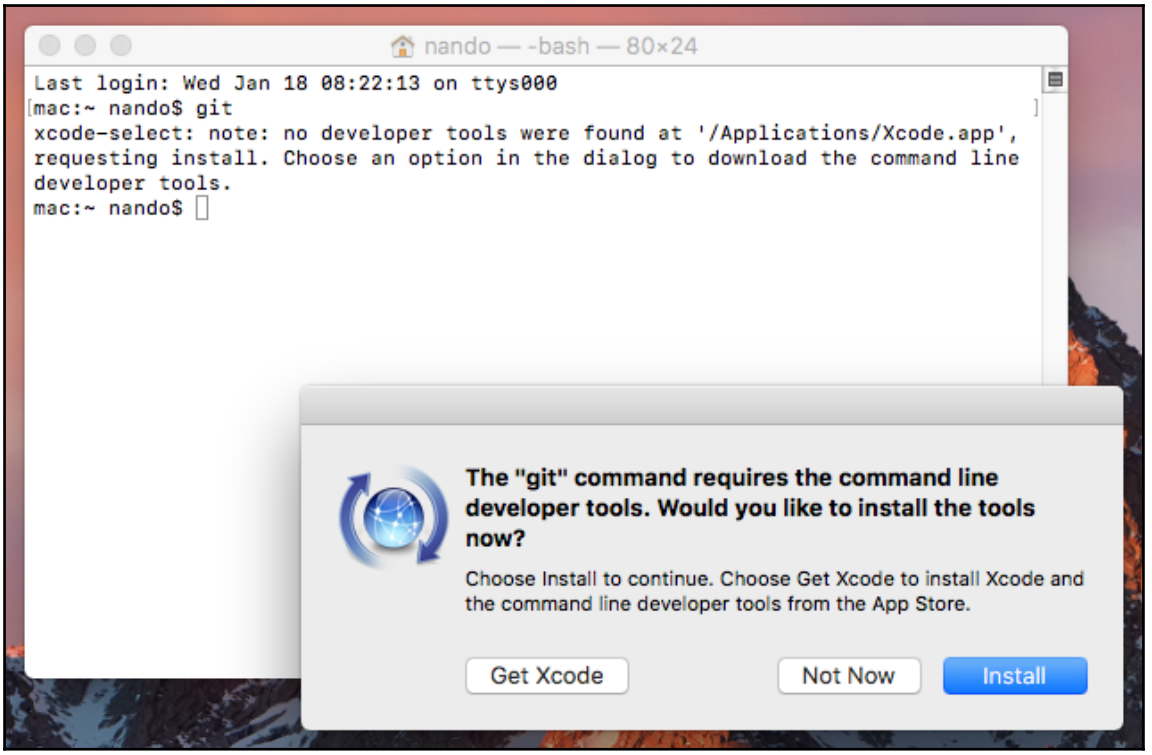


Installing Git on Kubuntu

WOW! eBook
www.wowebook.org

Installing Git on macOS

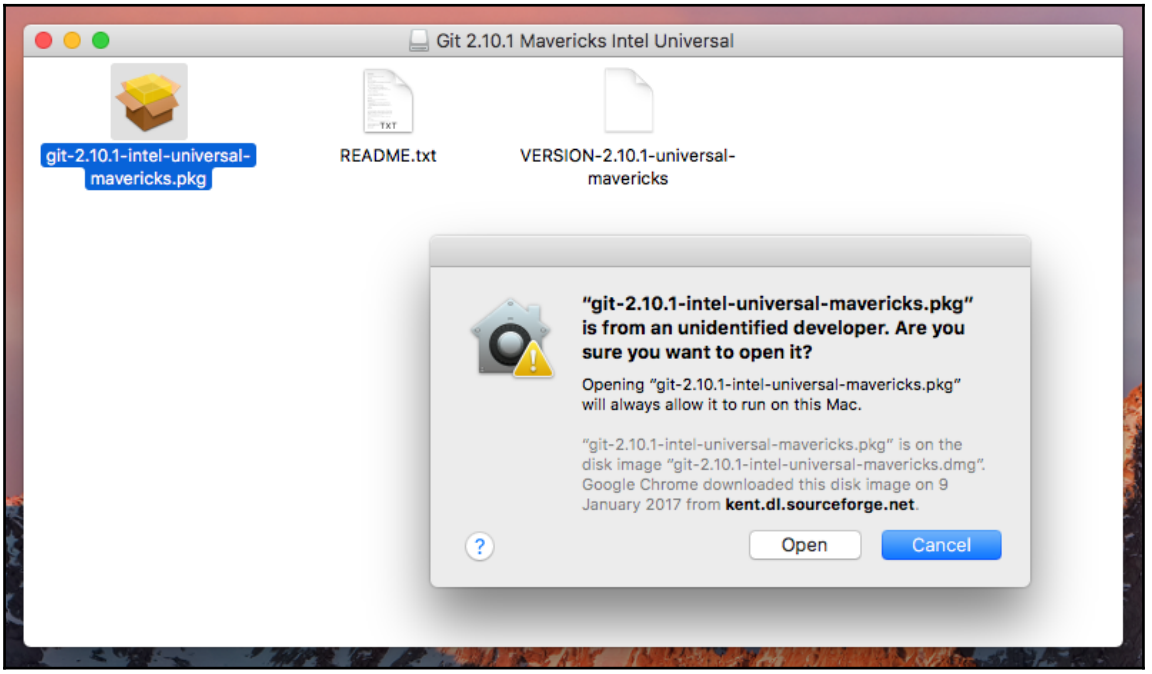
There are several ways to install Git on macOS. The easiest way is to install the *Xcode command line tools*. Since *Mavericks* (*10.9*), you can do this simply by trying to run `git` from the terminal for the very first time. If you don't have it installed already, it will prompt you to install it, as shown in the following image:



Clicking on the Install button will fire the installation process.

WOW! eBook
www.wowebook.org

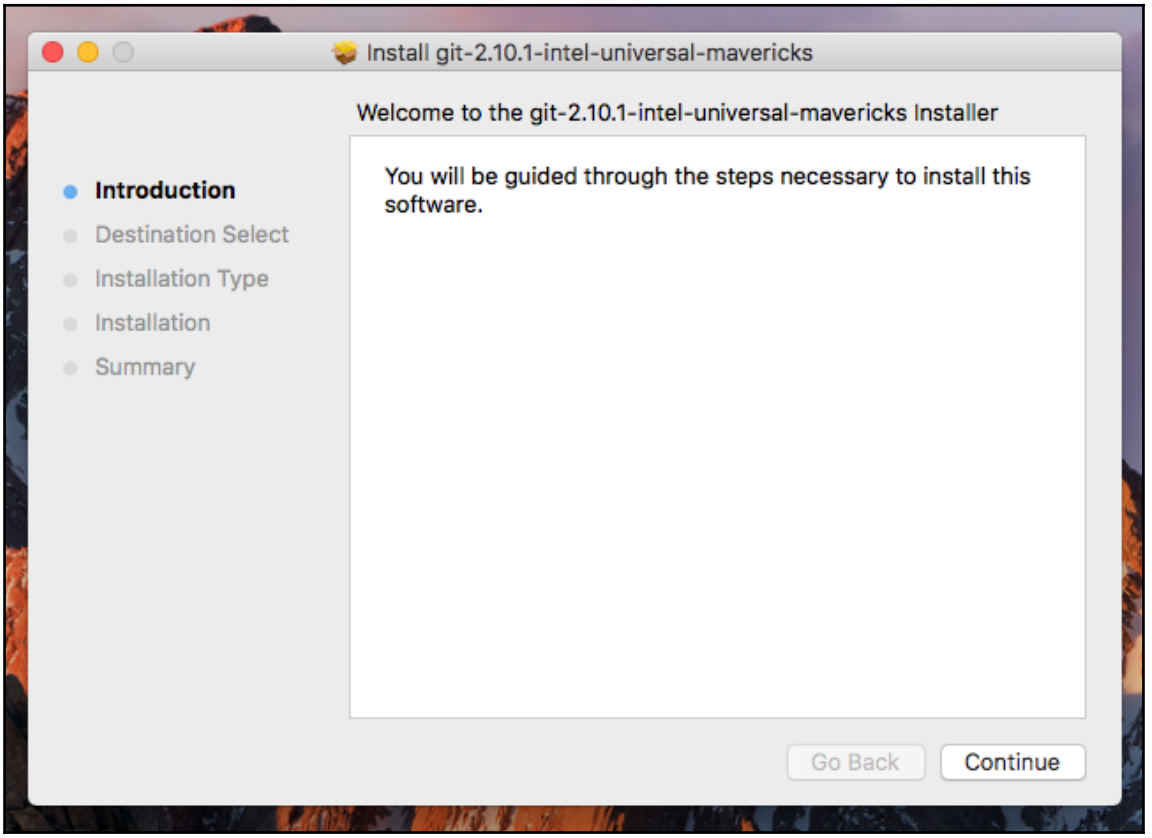
If you want a more up-to-date version, you can also install it via the `*.dmg` binary installer, downloaded from `git-scm.com` (it says *mavericks* in the file name, but just ignore that). Beware the macOS policies while installing packages downloaded from the Internet; to allow execution, you need to hold down *CTRL* and click on the package icon to open it:



Hold down CTRL and click to let macOS prompt you to open the package

WOW! eBook
www.wowebook.org

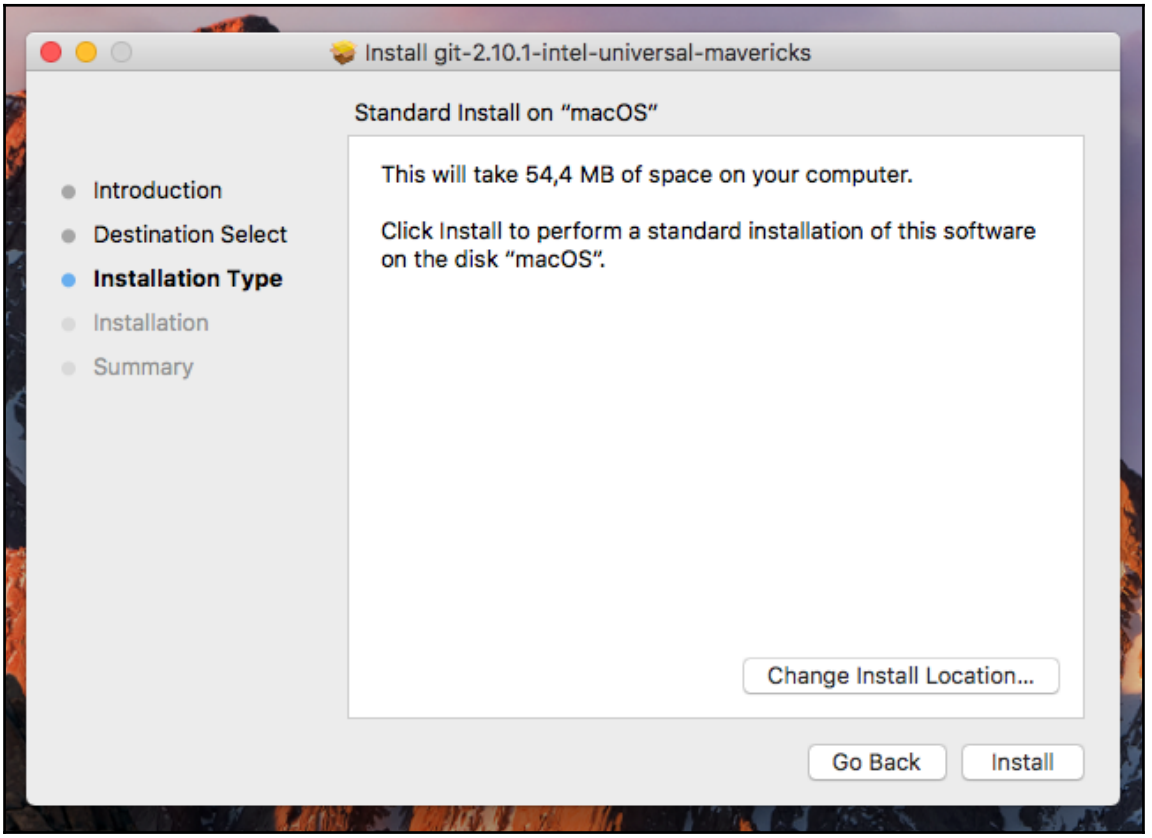
After this, the installation will be very easy-it's just a matter of clicking on the **Continue** button and following the steps represented in the following screenshot:



Let's start the installation process

WOW! eBook
www.wowebook.org

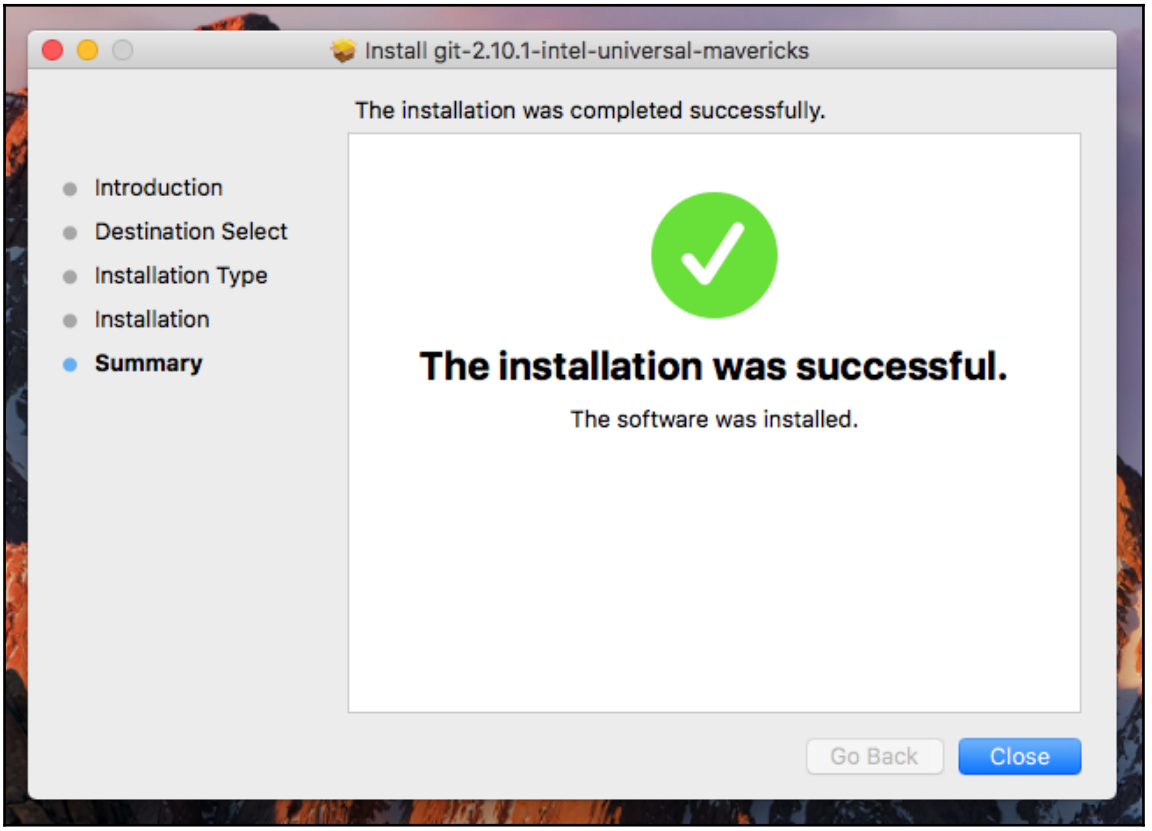
Click on the **Continue** button and then go on; a window like the one shown in the following image will appear:



Here you can change the installation location, if you need; if in doubt, simply click Install

WOW! eBook
www.wowebook.org

Click now on the **Install** button to start the installation. After a few seconds, the installation will be completed:

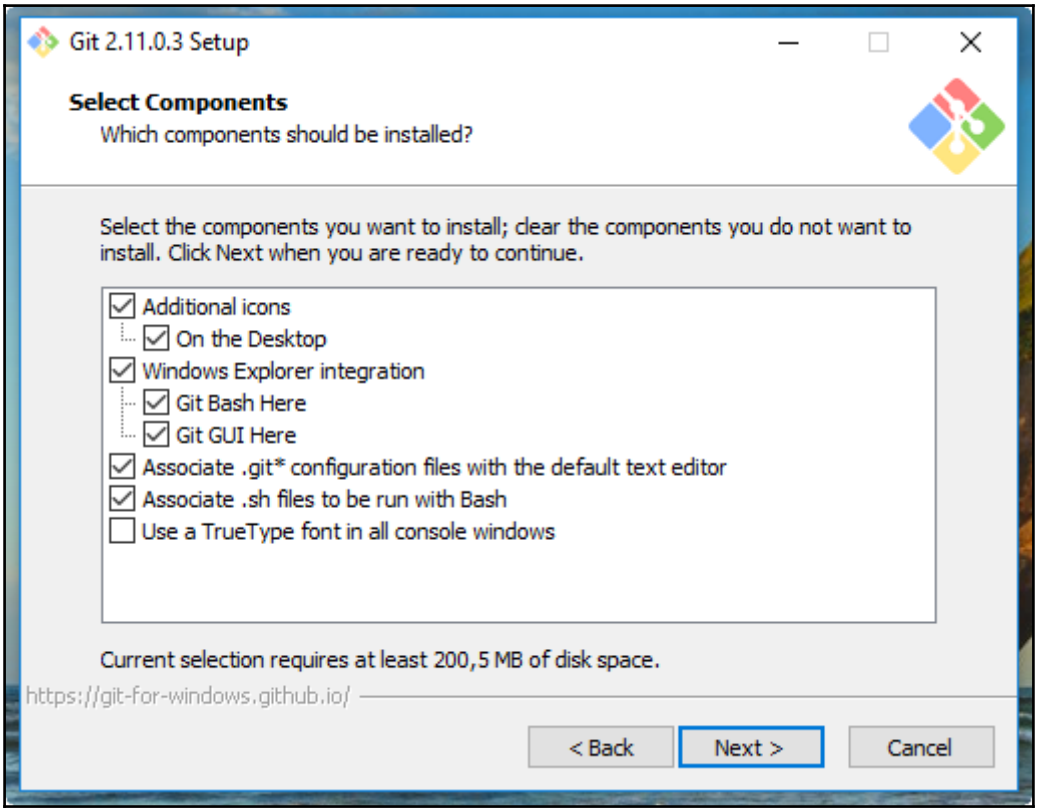


Installation complete

WOW! eBook
www.wowebook.org

Installing Git on Windows

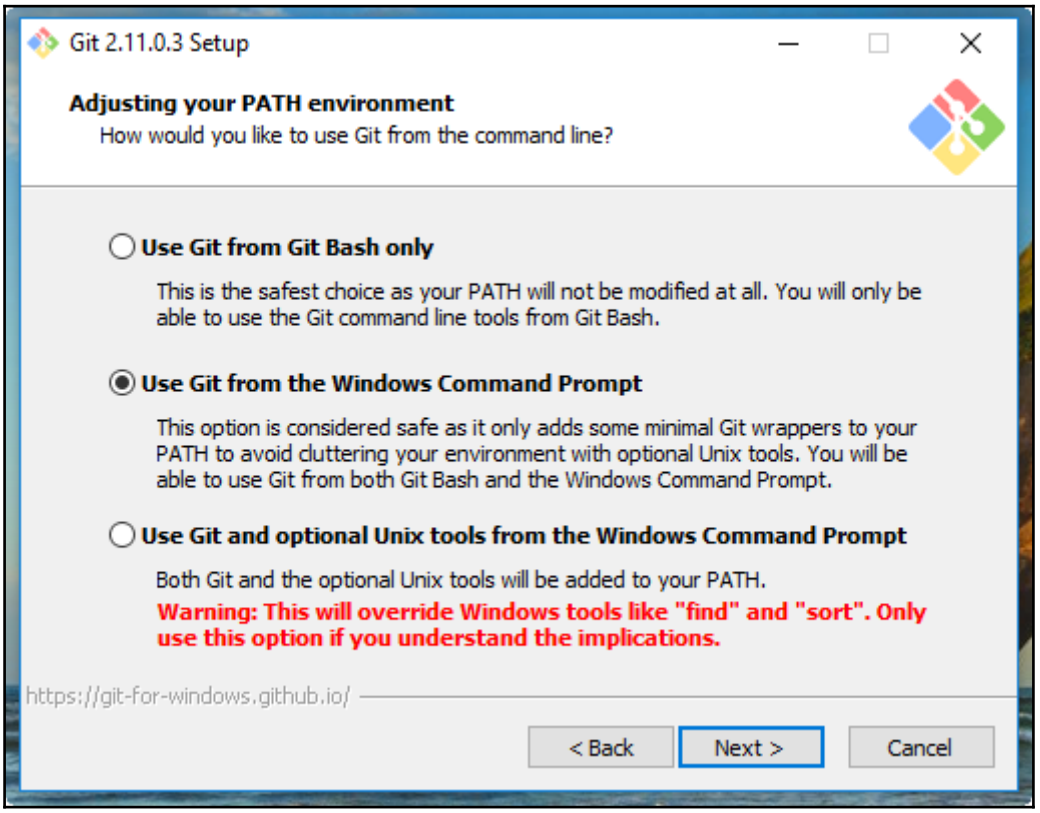
While clicking on the `Download` button on `http://git-scm.com`, you will automatically download Git in either the **x86** or the **x64** variant. I won't go into too much detail about the installation process itself, as it is trivial; I will only provide a few recommendations in the following screenshots:



Enabling **Windows Explorer integration** is generally useful; you will benefit from a convenient way to open a Git prompt in any folder by right-clicking on the contextual menu.

WOW! eBook
www.wowebook.org

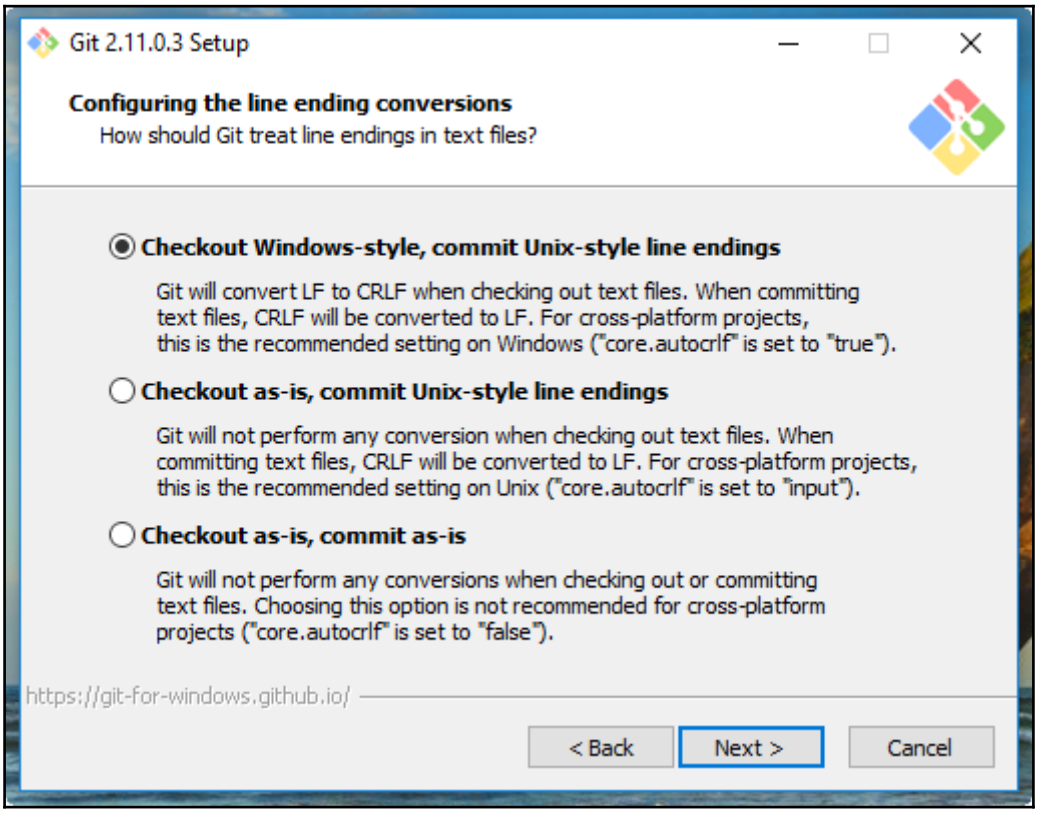
You should also enable Git to be used in the classic DOS command prompt, as shown in the following screenshot:



Git is provided with an embedded, Windows-compatible version of the famous **Bash shell** (see `https://en.wikipedia.org/wiki/Bash_(Unix_shell)`), which we will use extensively. By doing this, we will also make Git available to third-party applications, such as GUIs and so on. It will come in handy when we give some GUI tools a try.

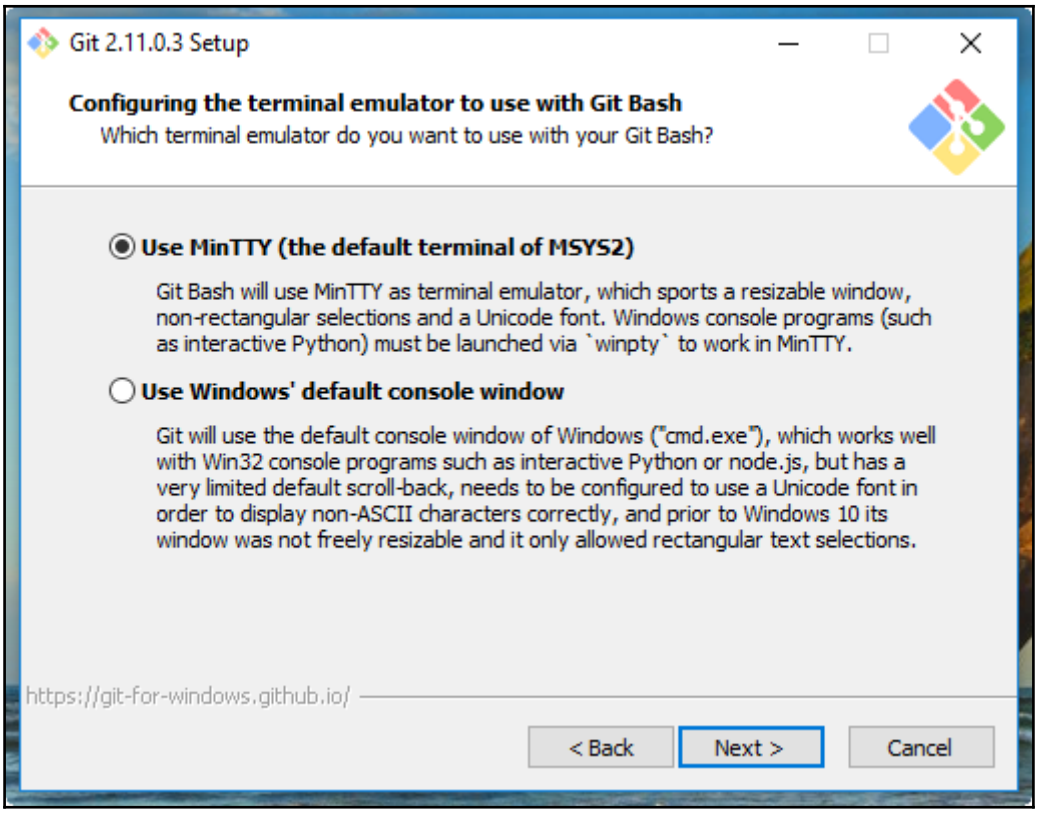
WOW! eBook
www.wowebook.org

Use defaults for line endings. This will protect you from future annoyances while working on multiplatform repositories:



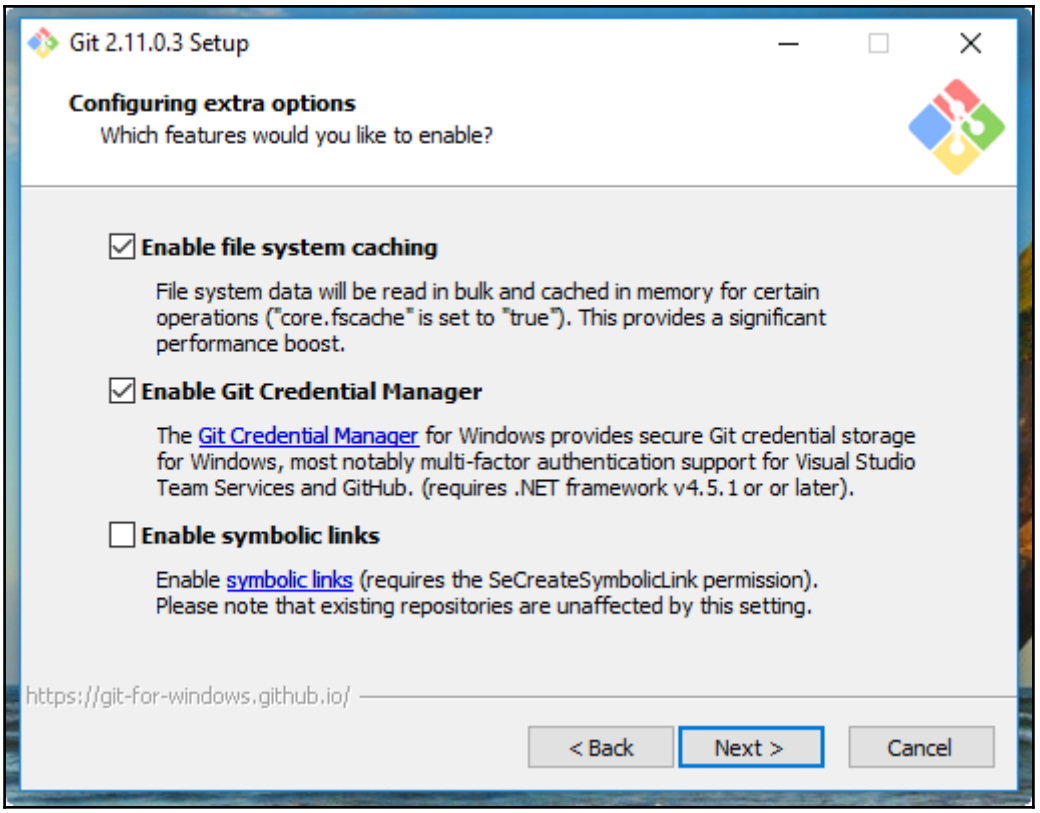
WOW! eBook
www.wowebook.org

Now it's time to choose a terminal emulator for Git; I recommend using **MinTTY** (see `https://mintty.github.io`), as it is a very good shell, fully customizable and user friendly:



Now, let's look at some new stuff from the latest Git releases-that is **file system caching**, the **Git Credential Manager**, and **symbolic links**:

WOW! eBook
www.wowebook.org



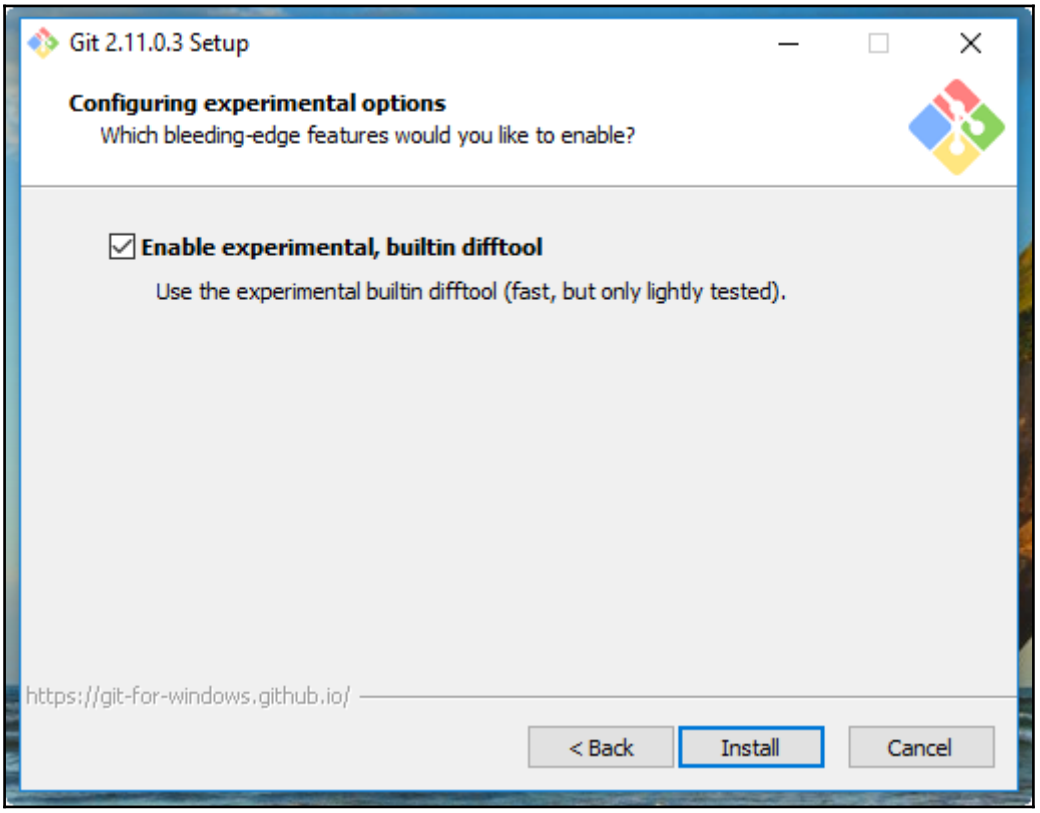
File system caching has been considered experimental until Git for Windows v2.7.4 (March 18, 2016), but now it is stable, and since Git 2.8 this feature is enabled by default. This is a Windows-only configuration option, and allows Git to be quicker when dealing with the underlying read/write operations. I recommend that you enable it for optimal performance.

Git Credential Manager (see `https://github.com/Microsoft/Git-Credential-Manager-for-Windows`) is included in the Git for Windows installer since v2.7.2 (February 23, 2016). Thanks to Microsoft, today, you can deal with Git users and passwords as easily as you can on other platforms. It requires .NET framework v4.5 or later, and perfectly integrates even with Visual Studio (see `https://www.visualstudio.com/`) and the GitHub for Windows (see `https://desktop.github.com`) GUI. I recommend that you enable it, as it saves some time while working.

WOW! eBook
www.wowebook.org

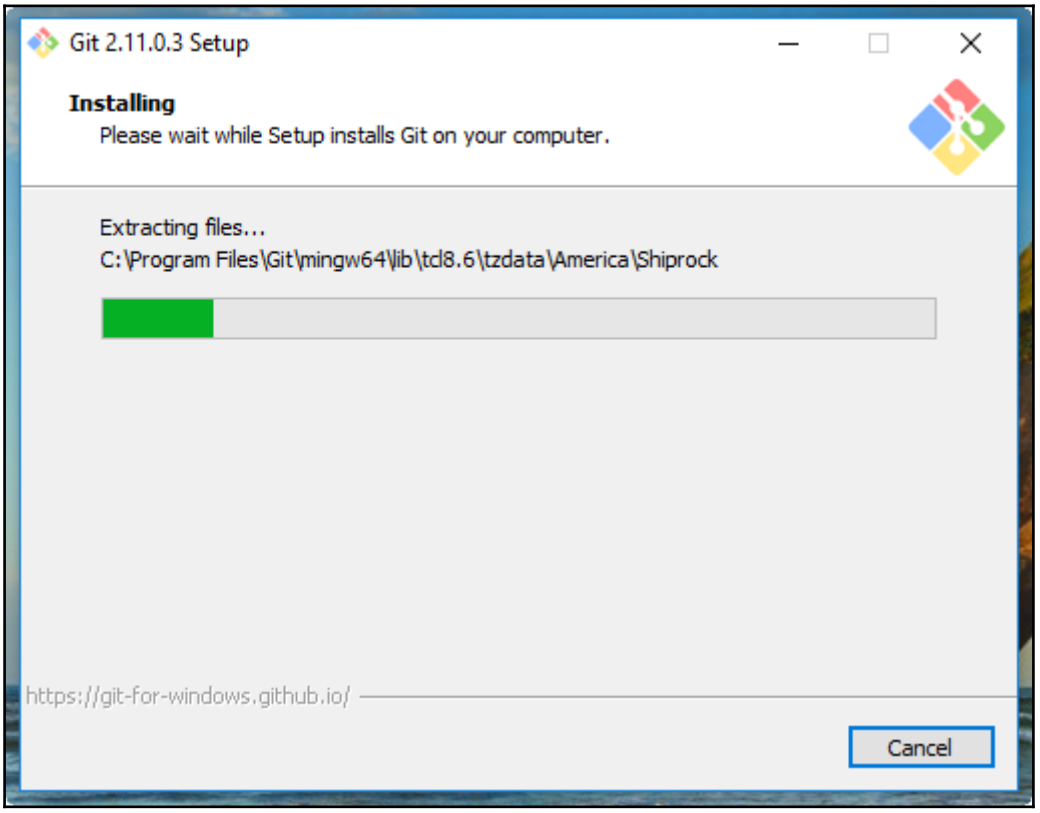
Symbolic links is a feature that Windows has lacked from the beginning, and even when they were introduced in Windows Vista, they have highlighted many incompatibilities with with Unix-like symlinks.

Anyway, Git and its Windows subsystem can handle them (with some limitations), so, if needed, you can try to install this feature and enable it in configuration options (they are disabled by default). For now, however, the best thing is to not use them at all in your repository if you need to work on the Windows platform. You can find more info at `https://github.com/git-for-windows/git/wiki/Symbolic-Links`.



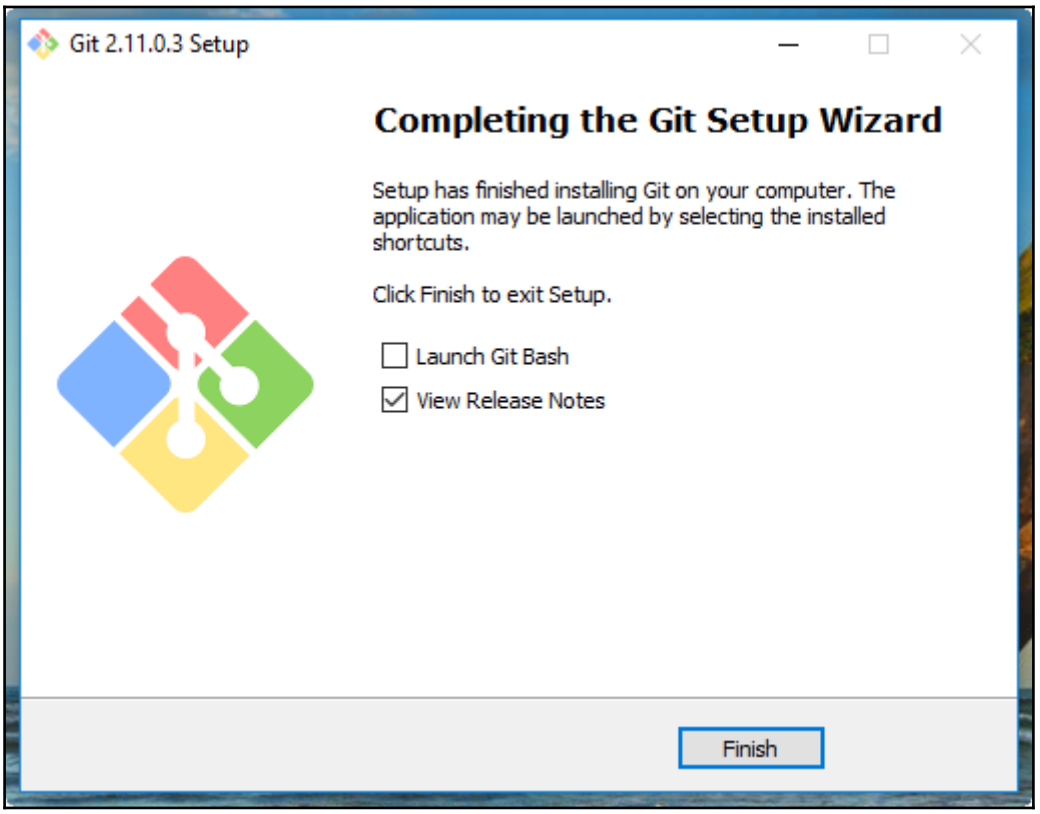
WOW! eBook
www.wowebook.org

Git for Windows v2.10.2 (November 2 2016) introduced **a new, built-in difftool** that promises quicker diffs. I use it on a daily basis, and I find it quite stable and fast. Enable it if you want give it a try, but this is not mandatory for the purposes of this book.



WOW! eBook
www.wowebook.org

Git for Windows will install it in the default `Program Files` folder, as all the Windows programs usually do.



WOW! eBook
www.wowebook.org

At the end of the process, we will have Git installed, and all its `*nix` friends will be ready to use it.

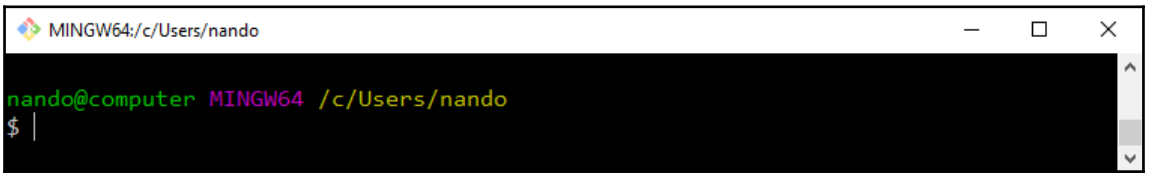
Please keep an eye on the **Release Notes** to see what's new in the latest release.

Running our first Git command

From now on, as a matter of convenience, we will use Windows as our platform of reference. Our screenshots will always refer to that platform. In any case, all the Git main commands we will use will work on the platforms we have previously mentioned anyway.

It's time to test our installation. Is Git ready to rock? Let's find out!

Using shell integration, right-click on an empty place on the desktop and choose the new menu item **Git Bash Here**. It will appear as a new MinTTY shell, providing you a Git-ready bash for Windows:



This is a typical Bash prompt: we can see the user, `nando`, and the host, `computer`. Then there's a `MINGW64` string, which refers to the actual platform we are using, called Minimalist GNU for Windows (see `http://www.mingw.org`), and at the end we find the actual path, in a more `*nix` fashion, `/c/Users/nando`. Later, we will look at this argument in more detail.

WOW! eBook
www.wowebook.org

Now that we have a new, shiny Bash prompt, simply type `git` (or the equivalent, `git --help`), as shown in the following screenshot:

```
MINGW64:/c/Users/nando

nando@computer MINGW64 /c/Users/nando
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local branch
   push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.

nando@computer MINGW64 /c/Users/nando
$
```


WOW! eBook
www.wowebook.org

If Git has been installed correctly, typing `git` without specifying anything else will result in a short help page, with a list of common commands (if not, try reinstalling Git).

So, we have Git up and running! Are you excited? Let's begin to type!

Making presentations

Git needs to know who you are. This is because in Git, every modification you make in a repository has to be signed with the name and email of the author. So, before doing anything else, we have to tell Git this information.

Type these two commands:

```
MINGW64:/home
nando@computer MINGW64 ~
$ git config --global user.name "Ferdinando Santacroce"

nando@computer MINGW64 ~
$ git config --global user.email "ferdinando.santacroce@gmail.com"

nando@computer MINGW64 ~
$
```

Using the `git config` command, we set up two configuration variables-`user.name` and `user.email`. Starting from now, Git will use them to sign your commits in all your repositories. Do not worry about it for now; in the next few chapters, we will explore the Git configuration system in more detail.

Setting up a new repository

The first step is to set up a new repository. A **repository** is a container for your entire project; every file or subfolder within it belongs to that repository, in a consistent manner. Physically, a repository is nothing other than a folder that contains a special `.git` folder, the folder where the magic happens.

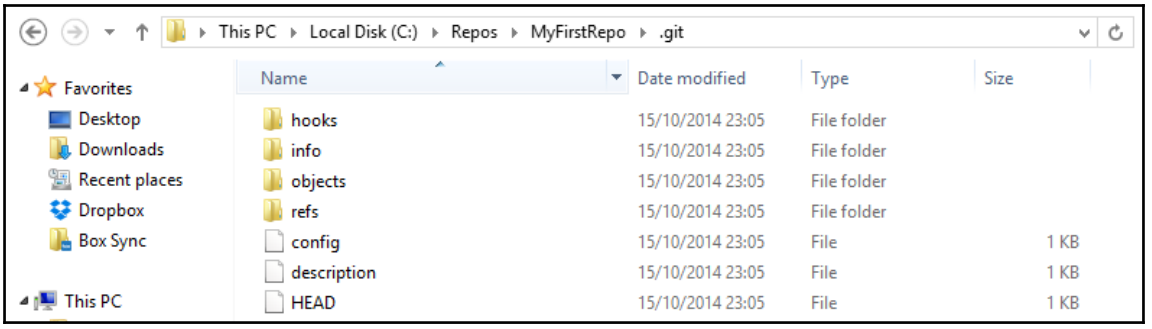
WOW! eBook
www.wowebook.org

Let's try to make our first repository. Choose a folder you like (for example, `C:\Repos\MyFirstRepo`), and type the `git init` command, as shown here:

```
MINGW64:/c/Repos/MyFirstRepo

[1] ~
$ cd /c/Repos/MyFirstRepo/

[2] /c/Repos/MyFirstRepo
$ git init
Initialized empty Git repository in C:/Repos/MyFirstRepo/.git/

[3] /c/Repos/MyFirstRepo (master)
$
```

As you can see, I slightly modified the default Git Bash prompt to better fit the need of the demoing commands; I removed the user and host, and added an incremental number to every command we type so that it will be simpler for me to refer to it while explaining, and for you to refer to it while reading.

Let's get back on topic. What just happened inside the `MyFirstRepo` folder? Git created a `.git` subfolder. The subfolder (normally hidden in Windows) contains some other files and folders, as shown in the next screenshot:



At this point in time, it is not important for us to understand what is inside this folder. The only thing you have to know is that you do not have to touch it, ever! If you delete it or if you modify the files inside by hand, you could get into trouble. Have I frightened you enough?

WOW! eBook
www.wowebook.org

Now that we have a repository, we can start to put files inside it. Git can trace the history of any type of file, text-based or binary, small or large, with the same efficiency (more or less; large files are always a problem).

Adding a file

Let's create a text file, just to give it a try:

```
MINGW64:/c/Repos/MyFirstRepo
[3] /c/Repos/MyFirstRepo (master)
$ echo "Git rocks!" >> file.txt

[4] /c/Repos/MyFirstRepo (master)
$
```

And now what? Is that all? No! We have to tell Git to put this file in your repository, *explicitly*. **Git doesn't do anything that you don't want it to do**. If you have some spare or temp files in your repository, Git will not take care of them, but will only remind you that there are some files in your repository that are not under version control (in the next chapter, we will see how to instruct Git to ignore them when necessary).

Okay, back to the topic. I want `file.txt` under the control of Git, so let's add it, as shown here:

```
MINGW64:/c/Repos/MyFirstRepo
[4] /c/Repos/MyFirstRepo (master)
$ git add file.txt

[5] /c/Repos/MyFirstRepo (master)
$
```

The `git add` command tells Git that we want it to take care of that file and check it for future modifications.

In response to this command, it could happen that you will see this response message from Git:

```
warning: LF will be replaced by CRLF in file.txt.
```


WOW! eBook
www.wowebook.org

The file will have its original line endings in your working directory.

This is because of the option that we selected when installing Git: *Checkout Windows-style, commit Unix-style line endings*. Don't worry about it for the moment; we will deal with it later.

Now, let us see if Git obeyed us.

Using the `git status` command, we can check the status of the repository, as shown in this screenshot:

```
MINGW64:/c/Repos/MyFirstRepo
[5] /c/Repos/MyFirstRepo (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   file.txt


[6] /c/Repos/MyFirstRepo (master)
$
```

As we can see, Git has accomplished its work as expected. In this image, we can read words such as `branch`, `master`, `commit`, and `unstage`. We will look at them briefly, but for the moment, let's ignore them: The purpose of this first experiment is overcome our fear and start playing with Git commands; after all, we have an entire book in which to learn the significant details.

Committing the added file

At this point, Git knows about `file.txt`, but we have to perform another step to fix the snapshot of its content. We have to commit it using the appropriate `git commit` command. This time, we will add some flavor to our command, using the `--message` (or `-m`) subcommand, as shown here:

WOW! eBook
www.wowebook.org

```
MINGW64:/c/Repos/MyFirstRepo
[6] /c/Repos/MyFirstRepo (master)
$ git commit --message "First commit"
[master (root-commit) 356a954] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt

[7] /c/Repos/MyFirstRepo (master)
$
```

With the commit of `file.txt`, we have finally fired up our repository. Having done the first commit (also known as the root-commit, as you can see in the screenshot), the repository now has a `master` branch with a commit inside it. We will play with branches in the forthcoming chapters. Right now, think of it as a path of our repository, and keep in mind that a repository can have multiple paths that often cross each other.

Modifying a committed file

Now, we can try to make some modifications to the file and see how to deal with it, as shown in the following screenshot:

```
MINGW64:/c/Repos/MyFirstRepo
[7] /c/Repos/MyFirstRepo (master)
$ echo "That's easy!" >> file.txt

[8] /c/Repos/MyFirstRepo (master)
$ 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:   file.txt

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

[9] /c/Repos/MyFirstRepo (master)
$
```


WOW! eBook
www.wowebook.org

As you can see, the Bash shell warns us that there are some modifications painting the name of the modified files in red. Here, the `git status` command informs us that there is a file with some modifications, and that we need to commit it if we want to save this modification step in the repository history.

However, what does **`no changes added to commit`** mean? It is simple. Git makes you take a second look at what you want to include in the next commit. If you have touched two files but you want to commit only one, you can add only that one.

If you try to commit by skipping the `add` step, nothing will happen (see the following screenshot). We will analyze this behavior in depth in the next chapter.

So, let's add the file again for the purpose of getting things ready for the next commit:

```
MINGW64:/c/Repos/MyFirstRepo
[9] /c/Repos/MyFirstRepo (master)
$ git commit
On branch master
Changes not staged for commit:
        modified:   file.txt

no changes added to commit

[10] /c/Repos/MyFirstRepo (master)
$ git add file.txt

[11] /c/Repos/MyFirstRepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   file.txt


[12] /c/Repos/MyFirstRepo (master)
$
```

Okay, let's make another commit, this time, avoiding the `--message` subcommand. Type `git commit` and hit the *Enter* key:

WOW! eBook
www.wowebook.org

```
MINGW64:/c/Repos/MyFirstRepo
[12] /c/Repos/MyFirstRepo (master)
$ git commit
```

Fasten your seatbelts! You are now entering in a piece of code history!

```
MINGW64:/c/Repos/MyFirstRepo

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   file.txt
#
~
~
~
/c/Repos/MyFirstRepo/.git/COMMIT_EDITMSG [unix] (16:56 21/01/2017)1,0-1 All
```

What is that? It's **Vim** (**Vi IMproved**), an ancient and powerful text editor, used even today by millions of people. You can configure Git to use your own preferred editor, but if you don't do it, this is what you have to deal with. Vim is powerful, but for newcomers, it can be a pain to use. It has a strange way of dealing with text. To start typing, you have to press *I* for inserting text, as shown in the following screenshot:

```
MINGW64:/c/Repos/MyFirstRepo
Thing are becoming serious :D
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   file.txt
#
~
~
~
<Repos/MyFirstRepo/.git/COMMIT_EDITMSG[+] [unix] (16:56 21/01/2017)1,30 All
-- INSERT --
```


WOW! eBook
www.wowebook.org

Once you have typed your commit message, you can press *Esc* to get out of editing mode. Then, you can type the `:w` command to write changes and the `:q` command to quit. You can also type the command in pairs as `:wq`, as we do in this screenshot, or use the equivalent `:x` command:

```
MINGW64:/c/Repos/MyFirstRepo
Thing are becoming serious :D
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       modified:   file.txt
#
~
~
~
<Repos/MyFirstRepo/.git/COMMIT_EDITMSG[+] [unix] (16:56 21/01/2017)1,29 All
:wq
```

After that, press *Enter* and another commit is done, as shown here:

```
MINGW64:/c/Repos/MyFirstRepo
[11] /c/Repos/MyFirstRepo (master)
$ git commit
[master 1a0bf2e] Thing are becoming serious :D
 1 file changed, 1 insertion(+)

[13] /c/Repos/MyFirstRepo (master)
$
```

Note that when you exit from Vim, Git automatically dispatches the commit, as you can see in the preceding screenshot.

Well done! Now, it's time to recap.

WOW! eBook
www.wowebook.org

Summary

In this chapter, you learned that Git is not so difficult to install, even on a non-Unix platform, such as Windows.

Once you have chosen a directory to include in a Git repository, you can see that initializing a new Git repository is as easy as executing a `git init` command. For now, don't worry about saving it on a remote server and so on. It's not mandatory to save it; you can do this when you need, preserving the entire history of your repository. This is a killer feature of Git and DVCS in general. You can comfortably work offline and push your work to a remote location when the network is available, without any hassle.

Lastly, we discovered one of the most important character traits of Git: it will do nothing if you don't mention it explicitly. You also learned a little bit about the `add` command. We were obliged to perform a `git add` command for a file when we committed it to Git for the very first time. Then, we used another command when we modified it. This is because if you modify a file, Git does not expect that you want it to be automatically added to the next commit (and it's right to assume this, I'd say).

In the next chapter, we will look at some fundamentals of Git.

WOW! eBook
www.wowebook.org

2

Git Fundamentals - Working Locally

In this chapter, we will dive deep into some of the fundamentals of Git; it is essential to understand well how Git thinks about files, its way of tracking the history of commits, and all the basic commands that we need to master, in order to become proficient.

Digging into Git internals

In this second edition of *Git Essentials*, I slightly changed my approach in explaining how Git works; instead of explaining with words and figures, this time I want to show you how Git works internally with only the help of the shell, allowing you to follow all the steps on your computer and hoping that these will be clear enough for you to understand.

Once you know the fundamentals of the Git working system, I think the rest of the commands and patterns will be clearer, allowing you to accomplish proficiently your daily work, getting out of trouble when needed.

So, it's time to start digging inside the true nature of Git; in this chapter, we will get in touch with the essence of this powerful tool.

Git objects

In `Chapter 1`, *Getting Started with Git*, we created an empty folder (in `C:\Repos\MyFirstRepo`) and then we initialized a new Git repository, using the `git init` command.

WOW! eBook
www.wowebook.org

Let's create a new repository to refresh our memory and then start learning a little bit more about Git.

In this example, we use Git to track our shopping list before going to the grocery; so, create a new grocery folder, and then initialize a new Git repository:

```
[1] ~
$ mkdir grocery

[2] ~
$ cd grocery/

[3] ~/grocery
$ git init
Initialized empty Git repository in C:/Users/san/Google
Drive/Packt/PortableGit/home/grocery/.git/
```

As we have already seen before, the result of the `git init` command is the creation of a `.git` folder, where Git stores all the files it needs to manage our repository:

```
[4] ~/grocery (master)
$ ll
total 8
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 ./
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 ../
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 .git/
```

So, we can move this `grocery` folder wherever we want, and no data will be lost. Another important thing to highlight is that we don't need any server: we can create a repository locally and work with it whenever we want, even with no LAN or internet connection. We only need them if we want to share our repository with someone else, directly or using a central server.
In fact, during this example, we won't use any remote server, as it is not necessary.

Go on and create a new `README.md` file to remember the purpose of this repository:

```
[5] ~/grocery (master)
$ echo "My shopping list repository" > README.md
```

Then add a banana to the shopping list:

```
[6] ~/grocery (master)
$ echo "banana" > shoppingList.txt
```


WOW! eBook
www.wowebook.org

At this point, as you already know, before doing a commit, we have to add files to the *staging area*; add both the files using the shortcut `git add .`:

```
[7] ~/grocery (master)
$ git add .
```

With this trick (the dot after the `git add` command), you can add all the new or modified files in one shot.

At this point, if you didn't set up a global username and email like we did in `Chapter 1`, *Getting Started with Git*, this is a thing that could happen:

```
[8] ~/grocery (master)
$ git commit -m "Add a banana to the shopping list"
[master (root-commit) c7a0883] Add a banana to the shopping list
Committer: Santacroce Ferdinando <san@intre.it>

Your name and email address were configured automatically based on your
username and hostname. Please check that they are accurate.

You can suppress this message by setting them explicitly:
git config --global user.name "Your Name"
git config --global user.email you@example.com
After doing this, you may fix the identity used for this commit with:
git commit --amend --reset-author

 2 files changed, 2 insertions(+)
 create mode 100644 README.md
 create mode 100644 shoppingList.txt
```

First of all, take a look at the second line, where Git says something like `root commit`; this means this is **the first commit** of your repository, and this is like a root in a tree (or a root on a disk partition; maybe you nerds will understand this better). Later we will come back to this concept.

Then, Git shows a message that says: "*You didn't set a global username and email; I used ones I found configured in your system, but if you don't like it, you can go back and remake your commit with another pair of data*".

WOW! eBook
www.wowebook.org

I prefer not to set up a global username and password in Git, as I usually work on different repositories using different usernames and emails; if I don't pay attention, I end up doing a job commit with my hobby profile or vice versa, and this is annoying. So, I prefer setting up usernames and emails per repository; in Git, you can set up your config variables at three levels: *repository* (with the `--local` option, the default one), *user* (with the `--global` option), and *system-wide* (with the `--system` option). Later we will learn something more about configuration, but this is what you need for now to go on with.

So, let's change these settings and *amend* our commit (amending a commit is a way to redo the last commit and fix up some little mistakes, such as adding a forgotten file, changing the message or the author, as we are going to do; later we will learn in detail what this means):

```
[9] ~/grocery (master)
$ git config user.name "Ferdinando Santacroce"

[10] ~/grocery (master)
$ git config user.email ferdinando.santacroce@gmail.com
```

As I didn't specify the config level, these parameters will be set at *repository level* (the same as `--local`); from now on, all the commits I will do in this repository will be signed by `"Ferdinando Santacroce"`, with the `ferdinando.santacroce@gmail.com` email (now you know how to get in touch with me, just in case).

Now it's time to type this command, `git commit --amend --reset-author`. When amending a commit this way, Git opens the default editor to let you change even the commit message, if you like; as we have seen in `Chapter 1`, *Getting Started with Git*, in Windows the default editor is *Vim*. For the purpose of this exercise, please leave the message as it is, press *Esc*, and then input the `:wq` (or `:x`) command and press *Enter* to save and exit:

```
[11] ~/grocery (master)
$ git commit --amend --reset-author #here Vim opens
[master a57d783] Add a banana to the shopping list
 2 files changed, 2 insertions(+)
 create mode 100644 README.md
 create mode 100644 shoppingList.txt
```

Okay, now I have a commit with the right author and email.

WOW! eBook
www.wowebook.org

Commits

Now it's time to start investigating commits.

To verify the commit we have just created, we can use the `git log` command:

```
[12] ~/grocery (master)
$ git log
commit a57d783905e6a35032d9b0583f052fb42d5a1308
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
Date:   Thu Aug 17 13:51:33 2017 +0200

Add a banana to the shopping list
```

As you can see, `git log` shows the commit we did in this repository; `git log` shows all the commits, in reverse chronological order; we have only a commit for now, but next we will see this behavior in action.

The hash

It is now time to analyze the information provided. The first line contains the commit's **SHA-1** (`https://en.wikipedia.org/wiki/SHA-1`), an alphanumeric sequence of 40 characters representing a hexadecimal number. This *code*, or **hash**, as it is usually called, uniquely identifies the commit within the repository, and it's thanks to it that from now on we can refer to it doing some actions.

The author and the commit creation date

We already talked about authors just a couple of paragraphs before; the **author** is who performed the commit, and the **date** is the full date when the commit was generated. Since that instance, this commit is part of the repository.

The commit message

Just under the author and date, after a blank line, we can see the message we attached to the commit we made; even the message is part of the commit itself.

WOW! eBook
www.wowebook.org

But there's something more under the hood; let's try to use the `git log` command with the `--format=fuller` option:

```
[13] ~/grocery (master)
$ git log --format=fuller
commit a57d783905e6a35032d9b0583f052fb42d5a1308
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
AuthorDate: Thu Aug 17 13:51:33 2017 +0200
Commit: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
CommitDate: Thu Aug 17 13:51:33 2017 +0200

Add a banana to the shopping list
```

The committer and the committing date

Other than the author, a commit preserves even the **committer**, and the **committing date**; what's the difference compared to author and author date? First of all, don't worry too much about this: 99% of commits in your repository will have the same values for the author and committer, and the same dates.

In some situations, such as the *cherry-pick*, you carry an existing commit on top of another branch, making a brand-new commit that applies the same changes of the previous. In this case, the author and author date will remain the same, while the committer and the committing date will be related to the person who performed this operation and the date they did it. Later we will get in touch with this useful Git command.

Going deeper

We analyzed a commit, and the information supplied by a simple `git log`; but we are not yet satisfied, so go deeper and see what's inside.

Using the `git log` command again, we can enable x-ray vision using the `--format=raw` option:

```
[14] ~/grocery (master)
$ git log --format=raw
commit a57d783905e6a35032d9b0583f052fb42d5a1308
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693
+0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
1502970693 +0200

Add a banana to the shopping list
```


WOW! eBook
www.wowebook.org

This time the output format is different; we can see the author and committer, as we saw before, but in a more compact form; then there is the commit message, but something new appears: it's a *tree*. Please be patient, we will talk about trees in a couple of paragraphs.

What I want to show now is another command, this time a little bit more obscure; it's `git cat-file -p`.

Let's try this command. To make it work, we need to specify the **object** we want to investigate; we can use the hash of the object, our first commit in this case. You don't need to specify the entire hash, but the first five-six characters are enough for small repositories. Git is smart enough to understand what's the object even with less than the 40 characters; the minimum is four characters, and the number increases as the total amount of Git objects in the repository increases. Just to give you an idea, the Linux kernel is currently 15 million lines of code, with millions of tracked files and folders; in that Git repository[1], you need to specify 12 characters to get the right object.

When in need, I usually try typing only the first five characters; if they are not sufficient to make Git aware of the object I need, it will warn me to input a character or two more.

Back on topic; type the command, specifying the first characters of the commit's hash (`a57d7` in my case):

```
[15] ~/grocery (master)
$ git cat-file -p a57d7
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693
+0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
1502970693 +0200

Add a banana to the shopping list
```

Okay, as you can see, the output is the same of `git log --format=raw`.

This is not unusual in Git: there are different commands and options that end up doing the same thing; this is a common *feature* of Git, and it's due to its organic growth across the years. Git changed (and changes) continuously, so the developers have to guarantee some backward compatibility when introducing new commands; this is one of the side effects.

I introduced this command only to have the chance of introducing another peculiarity of Git, the separation between *porcelain commands* and *plumbing commands*.

WOW! eBook
www.wowebook.org

Porcelain commands and plumbing commands

Git, as we know, has a myriad of commands, some of which are practically never used by the average user; as by example, the previous `git cat-file`. These commands are called **plumbing commands**, while those we have already learned about, such as `git add`, `git commit`, and so on, are among the so-called **porcelain commands**.

The metaphor originates directly from the fervent imagination of Linus Torvalds, the dad of Git, and has to do with plumbers. They, as is well known, also take care of the maintenance of the toilets; here Linus refers to the toilet bowl. The bowl is a porcelain artifact, which makes us sit comfortably; using then a series of pipes and devices, it allows us a correct discharge of what you know down into the sewerage system.

Linus has used this refined metaphor to divide the Git commands into two families, the highest-level ones, comfortable for a user interested in the most common operations (*porcelain*) and those used internally by the same (but usable at the discretion of the more experienced users) to perform lower-level operations (*plumbing*).

We can, therefore, consider porcelain commands as *interface* commands to the user, while the plumbing works at a *low level*. This also means that porcelain commands stay more *stable* over time (usage patterns and options vary with more caution and delayed time), as they are used directly, but are also implemented in numerous graphic tools, editors, and so on, while plumbing generally evolves with less restrictions.

There is no precise subdivision between these two categories of commands, as the border is often quite lively; we will still use them, in order to better understand the internal functioning of Git.

Let's go back to the topic now; we were talking about Git objects.

Git uses four different types of **objects**, and *commit* is one of these. Then there are *tree*, *blob*, and *annotated tag*. Let's leave the annotated tags aside for a moment (whoever already uses a versioning system knows what tags are) and focus on blobs and trees.

Here, for convenience, there is the output of the `git cat-file -p` command typed before:

```
[15] ~/grocery (master)
$ git cat-file -p a57d7
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693
+0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
1502970693 +0200
```


WOW! eBook
www.wowebook.org

As we can understand now, this plumbing command lets you peek into the Git objects; with the `-p` option (which means *pretty-print* here), we ask Git to show an easier way to read what the contents of the object are.

At this point, it's time to learn what a tree is in Git; in fact, in the command output, we can see this line: `tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40`.

What does it mean? Let's see it together.

Trees

The **tree** is a **container** for blobs and other trees. The easiest way to understand how it works is to think about folders in your operating system, which also collect files and other subfolders inside them.

Let's try to see what this additional Git object holds, using again the `git cat-file -p` command:

```
[16] ~/grocery (master)
$ git cat-file -p a31c3
100644 blob 907b75b54b7c70713a79cc6b7b172fb131d3027d README.md
100644 blob 637a09b86af61897fb72f26bfb874f2ae726db82 shoppingList.txt
```

This tree, which we said is something that Git uses to identify a folder, also contains some additional objects, called **blobs.**

Blobs

As you can see, at the right of the previous command output, we have `README.md` and `shoppinglist.txt`, which makes us guess that Git blobs represent the **files**. As before, we can verify its contents; let's see what's inside `637a0`:

```
[17] ~/grocery (master)
$ git cat-file -p 637a0
banana
```

Wow! Its content is exactly the content of our `shoppingFile.txt` file.

WOW! eBook
www.wowebook.org

To confirm, we can use the `cat` command, which on `*nix` systems allows you to see the contents of a file:

```
[18] ~/grocery (master)
$ cat shoppingList.txt
banana
```

As you can see, the result is the same.

Blobs are binary files, nothing more and nothing less. These byte sequences, which cannot be interpreted with the naked eye, retain inside information belonging to any file, whether binary or textual, images, source code, archives, and so on. Everything is compressed and transformed into a blob before archiving it into a Git repository.

As already mentioned previously, each file is marked with a *hash*; this hash uniquely identifies the file within our repository, and it is thanks to this ID that Git can then retrieve it when needed, and detect any changes when the same file is altered (files with different content will have different hashes).

We said SHA-1 hashes are unique; but what does it mean?

Let's try to understand it better with an example.

Open a shell and try to play a bit with another plumbing command, `git hash-object`:

```
[19] ~/grocery (master)
$ echo "banana" | git hash-object --stdin
637a09b86af61897fb72f26bfb874f2ae726db82
```

The `git hash-object` command is the plumbing command to calculate the hash of any object; in this example, we used the `--stdin` option to pass as a command argument the result of the preceding command, `echo "banana";` in a few words, we calculated the hash of the string `"banana"`, and it came out `637a09b86af61897fb72f26bfb874f2ae726db82`.

And on your computer, did you try it? What is the result?

A bit of suspense... That's incredible, it's the same!

You can try to rerun the command as many times as you want, the resulting hash will always be the same (if not, it can be due to different line endings in your operating system or shell).

WOW! eBook
www.wowebook.org

This makes us understand something very important: **an object**, whatever it is, **will always have the same hash in any repository**, in any computer, on the face of the Earth.

The experienced and the smart ones probably had "*smelt a rat*" for some time now, but I hope that in the rest of the readers I have pulled up the same amazement that caught me when I did this for the first time. This behavior has some interesting implications, as we will see soon.

Last, but not least, I want to highlight how **Git calculates the hash on the content of the file, not in the file itself**; in fact, the `637a09b86af61897fb72f26bfb874f2ae726db82` hash calculated using `git hash-object` is the same as the blob we inspect previously using `git cat-file -p`. This teaches us an important lesson: if you have two different files with the same content, even if they have different names and paths, in Git you will end up having only one blob.

Even deeper - the Git storage object model

Okay, now we know there are different Git objects, and we can inspect inside them using some plumbing commands. But how and where does Git store them?

Do you remember the `.git` folder? Let's put our nose inside it:

```
[20] ~/grocery (master)
$ ll .git/
total 13
drwxr-xr-x 1 san 1049089   0 Aug 18 17:22 ./
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 ../
-rw-r--r-- 1 san 1049089 294 Aug 17 13:52 COMMIT_EDITMSG
-rw-r--r-- 1 san 1049089 208 Aug 17 13:51 config
-rw-r--r-- 1 san 1049089  73 Aug 17 11:11 description
-rw-r--r-- 1 san 1049089  23 Aug 17 11:11 HEAD
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 hooks/
-rw-r--r-- 1 san 1049089 217 Aug 18 17:22 index
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 info/
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 logs/
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 objects/
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 refs/
```


WOW! eBook
www.wowebook.org

Within it, there is an `objects` subfolder; let's take a look:

```
[21] ~/grocery (master)
$ ll .git/objects/
total 4
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ./
drwxr-xr-x 1 san 1049089 0 Aug 18 17:22 ../
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 63/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 90/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 a3/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 a5/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 c7/
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 info/
drwxr-xr-x 1 san 1049089 0 Aug 18 17:12 pack/
```

Other than `info` and `pack` folders, which are not interesting for us right now, as you can see there are some other folders with a strange two-character name; let's go inside the `63` folder:

```
[22] ~/grocery (master)
$ ll .git/objects/63/
total 1
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ./
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ../
-r--r--r-- 1 san 1049089 20 Aug 17 13:34
7a09b86af61897fb72f26bfb874f2ae726db82
```

Hmmm...

Look at the file within it, and think: `63` + `7a09b86af61897fb72f26bfb874f2ae726db82` is actually the hash of our `shoppingList.txt` blob!

Git is amazingly smart and simple: to be quicker while searching through the filesystem, Git creates a set of folders where the name is two characters long, and those two characters represent the first two characters of a hash code; inside those folders, Git writes all the objects using as a name the other 38 characters of the hash, regardless of the kind of Git object.

So, the `a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40` tree is stored in the `a3` folder, and the `a57d783905e6a35032d9b0583f052fb42d5a1308` commit in the `a5` one.

WOW! eBook
www.wowebook.org

Isn't that the most clever and simple thing you have ever seen?

Now, if you try to inspect those files with a common `cat` command, you will be deluded: those files are plain text files, but Git compresses them using the `zlib` library to reserve space on your disk. This is why we use the `git cat-file -p` command, which decompresses them on the fly for us.

This highlights once again the simplicity of Git: no metadata, no internal databases, or useless complexity, but simple files and folders are enough to make it possible to manage any repository.

At this point, we know how Git stores objects, and where they are archived; we also know that there is no database, no central repository or stuff like that, so how is Git able to reconstruct the history of our repository? How can it define which commit precedes or follows another one?

To become aware of this, we need a new commit. So, let's now proceed modifying the `shoppingList.txt` file:

```
[23] ~/grocery (master)
$ echo "apple" >> shoppingList.txt

[24] ~/grocery (master)
$ git add shoppingList.txt

[25] ~/grocery (master)
$ git commit -m "Add an apple"
[master e4a5e7b] Add an apple
 1 file changed, 1 insertion(+)
```

Use the `git log` command to check the new commit; the `--oneline` option allows us to see the log in a more compact way:

```
[26] ~/grocery (master)
$ git log --oneline
e4a5e7b Add an apple
a57d783 Add a banana to the shopping list
```

Okay, we have a new commit, with its hash. Time to see what's inside it:

```
[27] ~/grocery (master)
$ git cat-file -p e4a5e7b
tree 4c931e9fd8ca4581ddd5de9efd45daf0e5c300a0
parent a57d783905e6a35032d9b0583f052fb42d5a1308
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503586854
+0200
```


WOW! eBook
www.wowebook.org

```
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
1503586854 +0200

Add an apple
```

There's something new!

I'm talking about the `parent a57d783905e6a35032d9b0583f052fb42d5a1308` row; did you see? A **parent** of a commit is simply the commit that precedes it. In fact, the `a57d783` hash is actually the hash of the first commit we made. So, every commit has a parent, and following these relations between commits, we can always navigate from a random one down to the first one, the already mentioned **root commit**.

If you remember, the first commit did not have a parent, and this is the main (and only) difference between all commits and the first one. Git, while navigating and reconstructing our repository, simply knows it is done when it finds a commit without a parent.

Git doesn't use deltas

Now it's time to investigate another well-known difference between Git and other versioning systems. Take Subversion as an example: when you do a new commit, Subversion creates a new numbered revision that only contains deltas between the previous one; this is a smart way to archive changes to files, especially among big text files, because if only a line of text changes, the size of the new commit will be much smaller.

Instead, in Git even if you change only a char in a big text file, it always stores a new version of the file: **Git doesn't do deltas** (at least not in this case), and **every commit is actually a snapshot of the entire repository**.

At this point, people usually exclaim: *"Gosh, Git waste a large amount of disk space in vain!"*. Well, this is simply untrue.

In a common source code repository, with a certain amount of commit, Git usually won't need more space than other versioning systems. As an example, when Mozilla went from Subversion to Git, the exact same repository went from 12GB to 420MB disk space required; look at this comparison page to learn more:
`https://git.wiki.kernel.org/index.php/GitSvnComparsion`

WOW! eBook
www.wowebook.org

Furthermore, Git has a clever way to deal with files; let's take a look again at the last commit:

```
[28] ~/grocery (master)
$ git cat-file -p e4a5e7b
tree 4c931e9fd8ca4581ddd5de9efd45daf0e5c300a0
parent a57d783905e6a35032d9b0583f052fb42d5a1308
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503586854
+0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
1503586854 +0200

Add an apple
```

Okay, now to the tree:

```
[29] ~/grocery (master)
$ git cat-file -p 4c931e9
100644 blob 907b75b54b7c70713a79cc6b7b172fb131d3027d README.md
100644 blob e4ceb844d94edba245ba12246d3eb6d9d3aba504 shoppingList.txt
```

Annotate the two hashes on a notepad; now we have to look at the tree of the first commit; cat-file the commit:

```
[30] ~/grocery (master)
$ git cat-file -p a57d783
tree a31c31cb8d7cc16eeae1d2c15e61ed7382cebf40
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1502970693
+0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
1502970693 +0200
Add a banana to the shopping list
```

Then cat-file the tree:

```
[31] ~/grocery (master)
$ git cat-file -p a31c31c
100644 blob 907b75b54b7c70713a79cc6b7b172fb131d3027d README.md
100644 blob 637a09b86af61897fb72f26bfb874f2ae726db82 shoppingList.txt
```

Guess what! The hash of the `README.md` file is the same in the two trees of the first and second commit; this allows us to understand another simple but clever strategy that Git adopts to manage files; when a file is untouched, while committing Git creates a tree where the blob for the file points to the already existing one, *recycling* it and avoiding waste of disk space.

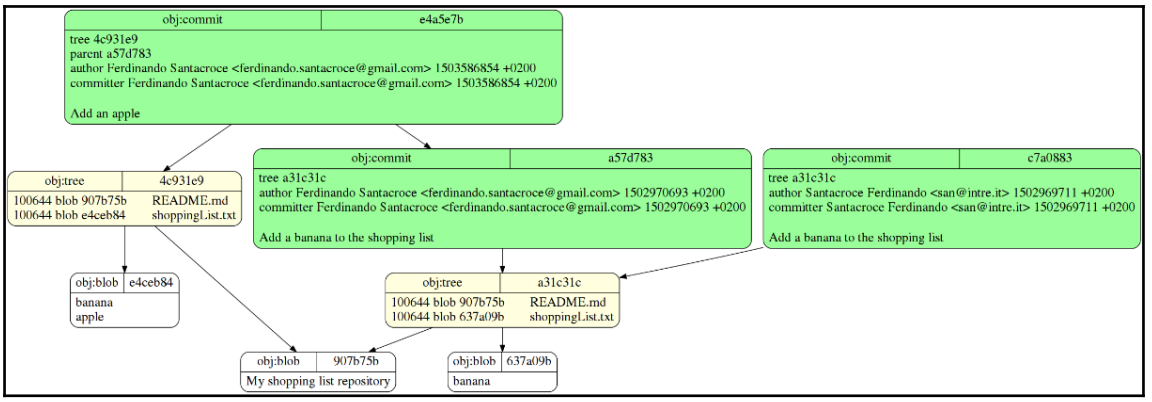
WOW! eBook
www.wowebook.org

The same applies to the trees: if my working directory has some folders and files within them that will remain untouched, when we do a new commit Git recycles the same trees.

Wrapping up

It's time to summarize all the concepts illustrated since now.

An image, as they say, is worth a thousand words, so here you can find a picture representing the actual state of our repository, thanks to the **git-draw** tool (`https://github.com/sensorflo/git-draw`):



In this graphic representation, you will find a detailed diagram that represents the current structure of the newly created repository; you can see trees (yellow), blobs (white), commits (green), and all relationships between them, represented by oriented arrows.

Note how the direction of the arrow joining the commit comes from the second commit and goes to the first, or from descendant to its ancestor; it may seem a detail, but it is important that graphic representations such as these are properly indicated in order to correctly highlight the relationship that binds the commits between them (it is always the child who depends on the parent).

WOW! eBook
www.wowebook.org

I just want to highlight some other things; such as:

- The two different trees refer to the same `README.md` blob
- There are two different blobs for the `shoppingList.txt` files, one containing only the `banana` text line and one containing `banana` and `apple`
- The second commit refers to the first
- The first commit has no parent
- There are three commits!

What the heck?!

Okay, don't panic. Look at the commit at the right of the picture, and read author and email: that was the first commit we did using the *wrong* user and email; after that, we amended the commit, changing the author, remember?

Well, but why is it already there? And why do we see it in this picture, but we don't see it in `git log`?

It's about **reachability** of the commit, a topic that we will talk about in the following sections.

Git references

In the previous section, we have seen that a Git repository can be imagined as a tree that, starting from a root (the root-commit), grows upward through one or more branches.

These branches are generally distinguished by a name. In this Git is no exception; if you remember, the experiments conducted so far led us to commit to the `master` branch of our test repository. *Master* is precisely the name of the *default branch* of a Git repository, somewhat like `trunk` is for Subversion.

But Subversion analogies end here: we will now see how Git handles branches, and for Subversion users it will be a little surprising.

WOW! eBook
www.wowebook.org

It's all about labels

In Git, **a branch is nothing more than a label**, a *mobile label* placed on a commit.

In fact, every leaf on a Git branch has to be labeled with a meaningful name to allow us to reach it and then move around, go back, merge, rebase, or discard some commits when needed.

Let's start exploring this topic by checking the current status of our `grocery` repository; we do it using the well-known `git log` command, this time adding some new options:

```
[1] ~/grocery (master)
$ git log --oneline --graph --decorate
* e4a5e7b (HEAD -> master) Add an apple
* a57d783 Add a banana to the shopping list
```

Let's look at those options in detail:

- `--graph`: In this case it just adds an asterisk to the left, before the commit hash, but when you have more branches, this option will draw them for us giving a simple but effective graphical representation of the repository
- `--decorate`: This option prints out the labels attached to any commits that are shown; in this case, it prints `(HEAD ->master)` on the `e4a5e7b` commit
- `--oneline`: This is easy to understand: it reports every commit using one line, shortening things when necessary

We'll now do a new commit and see what happens:

```
[2] ~/grocery (master)
$ echo "orange" >> shoppingList.txt

[3] ~/grocery (master)
$ git commit -am "Add an orange"
[master 0e8b5cf] Add an orange
 1 file changed, 1 insertion(+)
```

Have you noticed? After adding an orange to the `shopingList.txt`, I made a commit without first making `git add`; the *trick* is in the `-a` (`--add`) option added to the `git commit` command, which means *add to this commit all the modified files that I have already committed at least one time before*. In our case, this option allowed us to go faster and skip the `git add` command.

WOW! eBook
www.wowebook.org

Anyway, use it carefully, especially while learning and using Git at the beginning: you easily end up doing commit with more files than you want.

Okay, go on now and take a look at the current repository situation:

```
[4] ~/grocery (master)
$ git log --oneline --graph --decorate
* 0e8b5cf (HEAD -> master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Interesting! Both `HEAD` and `master` have now moved on the last commit, the third one; what does it mean?

Branches are movable labels

We have seen in the previous sections how the commits are linked to each other by a parent-and-son relationship: each commit contains a reference to the previous commit.

This means that, for example, to *navigate* within a repository I cannot start from the first commit and try to go to the next, because a commit has no reference to who comes next, but to who comes first. By staying in our *arboreal* metaphor, this means that our tree is only navigable from *leaves*, from the extreme *top* of a branch, and then down to root-commit.

So, branches are nothing but labels that are on the tip commit, the last one. This commit, our leaf, must always be identified by a label so that its ancestors commits can be reached while browsing within a repository. Otherwise, we should remember for every branch of our repository the hash code of the tip commit, and you can imagine how easy it would be for humans.

How references work

So, every time we make a commit to a branch, the **reference** that identifies that branch will move accordingly to always stay associated with the tip commit.

WOW! eBook
www.wowebook.org

But how will Git handle this feature? Let's go back to putting the nose again in the `.git` folder:

```
[5] ~/grocery (master)
$ ll .git/
total 21
drwxr-xr-x 1 san 1049089   0 Aug 25 11:20 ./
drwxr-xr-x 1 san 1049089   0 Aug 25 11:19 ../
-rw-r--r-- 1 san 1049089  14 Aug 25 11:20 COMMIT_EDITMSG
-rw-r--r-- 1 san 1049089 208 Aug 17 13:51 config
-rw-r--r-- 1 san 1049089  73 Aug 17 11:11 description
-rw-r--r-- 1 san 1049089  23 Aug 17 11:11 HEAD
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 hooks/
-rw-r--r-- 1 san 1049089 217 Aug 25 11:20 index
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 info/
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 logs/
drwxr-xr-x 1 san 1049089   0 Aug 25 11:20 objects/
drwxr-xr-x 1 san 1049089   0 Aug 18 17:15 refs/
```

There's a `refs` folder: let's take a look inside:

```
[6] ~/grocery (master)
$ ll .git/refs/
total 4
drwxr-xr-x 1 san 1049089 0 Aug 18 17:15 ./
drwxr-xr-x 1 san 1049089 0 Aug 25 11:20 ../
drwxr-xr-x 1 san 1049089 0 Aug 25 11:20 heads/
drwxr-xr-x 1 san 1049089 0 Aug 17 11:11 tags/
```

Now go to `heads`:

```
[7] ~/grocery (master)
$ ll .git/refs/heads/
total 1
drwxr-xr-x 1 san 1049089  0 Aug 25 11:20 ./
drwxr-xr-x 1 san 1049089  0 Aug 18 17:15 ../
-rw-r--r-- 1 san 1049089 41 Aug 25 11:20 master
```

There's a `master` file inside! Let's see what's the content:

```
[8] ~/grocery (master)
$ cat .git/refs/heads/master
0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8
```


WOW! eBook
www.wowebook.org

As you could imagine, Git manages all this articulated reference system... with a trivial text file! It contains the hash of the last commit made on the branch; in fact, if you look at the previous `git log` output, you can see the hash of the last commit is `0e8b5cf`.

Nowadays it has been time since the first time, but I continue to be amazed by how essential and effective the internal structure of Git is.

Creating a new branch

Now that we have warmed up, the fun begins. Let's see what happens when you ask Git to create a new branch. Since we are going to serve a delicious fruit salad, it's time to set a branch apart for a *berries-flavored* variant recipe:

```
[9] ~/grocery (master)
$ git branch berries
```

That's all! To create a new branch, all you need to do is call the `git branch` followed by the name of the branch you'd like to use. And this is super-fast; always working locally, Git does this kind of work in a blink of an eye.

To be true, there are some (complicated) rules to be respected and things to know about the possible name of a branch (all you need to know is here: `https://git-scm.com/docs/git-check-ref-format`), but for now it is not relevant.

So, `git log` again:

```
[10] ~/grocery (master)
$ git log --oneline --graph --decorate
* 0e8b5cf (HEAD -> master, berries) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Wonderful! Now Git tells us there's a new branch, `berries`, and it refers to the same commit as a `master` branch.

Anyway, at the moment we continue to be located in the `master` branch; in fact, as you can see in the shell output prompt, it continues to appear `(master)` between the round parenthesis:

```
[10] ~/grocery (master)
```


WOW! eBook
www.wowebook.org

How can I switch branch? By using the `git checkout` command:

```
[11] ~/grocery (master)
$ git checkout berries
Switched to branch 'berries'
```

Do a `git log` to see:

```
[12] ~/grocery (berries)
$ git log --oneline --graph --decorate
* 0e8b5cf (HEAD -> berries, master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Mmm, interesting! Now there's a `(berries)` sign into the shell prompt, and more, something happened to that `HEAD` thing: now the arrows points to `berries`, not more to `master`. What does it mean?

HEAD, or you are here

During previous exercises we continued to see that `HEAD` thing while using `git log`, and now it's time to investigate a little bit.

First of all, what is `HEAD`? As branches are, `HEAD` is a **reference**. It represents a pointer to the place on where we are right now, nothing more, nothing less. In practice instead, it is just another plain text file:

```
[13] ~/grocery (berries)
$ cat .git/HEAD
ref: refs/heads/berries
```

The difference between the `HEAD` file and branches text file is that the `HEAD` file usually refers to a branch, and not directly to a commit as branches do. The `ref:` part is the convention Git uses internally to declare a pointer to another branch, while `refs/heads/berries` is of course the relative path to the `berries` branch text file.

So, having checked out the `berries` branch, in fact we moved that pointer from the `master` branch to the `berries` one; from now on, every commit we do will be part of the `berries` branch. Let's give it a try.

WOW! eBook
www.wowebook.org

Add a blackberry to the shopping list:

```
[14] ~/grocery (berries)
$ echo "blackberry" >> shoppingList.txt
```

Then perform a commit:

```
[15] ~/grocery (berries)
$ git commit -am "Add a blackberry"
[berries ef6c382] Add a blackberry
 1 file changed, 1 insertion(+)
```

Take a look on what happened with the usual `git log` command:

```
[16] ~/grocery (berries)
$ git log --oneline --graph --decorate
* ef6c382 (HEAD -> berries) Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Nice! Something happened here:

- The `berries` branch moved to the last commit we performed, confirming what we said before: a branch is just a label that follows you while doing new commits, getting stuck to the last one
- The `HEAD` pointer moved too, following the branch it is actually pointing to, the `berries` one
- The `master` branch remains where it was, stuck to the penultimate commit, the last one we did before switching to the `berries` branch

Okay, so now our `shoppingList.txt` file appears to contain these text lines:

```
[17] ~/grocery (berries)
$ cat shoppingList.txt
banana
apple
orange
blackberry
```

What happens if we move back to the `master` branch? Let's see.

WOW! eBook
www.wowebook.org

Check out the master branch:

```
[18] ~/grocery (berries)
$ git checkout master
Switched to branch 'master'
```

Look at the `shoppingFile.txt` content:

```
[19] ~/grocery (master)
$ cat shoppingList.txt
banana
apple
orange
```

We actually moved back to where we were before adding the blackberry; as it is being added in the `berries` branch, here in the `master` branch it does not exist: sounds good, doesn't it?

Even the `HEAD` file has been updated accordingly:

```
[20] ~/grocery (master)
$ cat .git/HEAD
ref: refs/heads/master
```

But at this point someone could raise their hand and say: *"That's weird! In Subversion, we usually have different folders for each different branch; here Git seems to always overwrite the content of the same folder, isn't it?"*.

Of course, it is. This is how Git works. When you switch a branch, Git goes to the commit the branch is pointing to, and following the parent relationship and analyzing trees and blobs, rebuilds the content on the **working directory** accordingly, getting hold of that files and folders (that is the same Subversion can do with the *switch branch* feature, actually).

This is a big difference between Git and Subversion (and other similar versioning systems); people used to Subversion often argue that in this manner you cannot easily compare branches file by file, or open in your favorite IDE two different *versions* of your in-development software. Yes, this is true, in Git you cannot do the same, but there are some tricks to work around this issue (if it is an issue for you).

Another important thing to say is that in Git you cannot check out only a folder of the repository, as you can do in Subversion; when you check out a branch, you get all its content.

WOW! eBook
www.wowebook.org

Go back to the repository now; let's do the usual `git log`:

```
[21] ~/grocery (master)
$ git log --oneline --graph --decorate
* 0e8b5cf (HEAD -> master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Uh-oh: where is the `berries` branch? Don't worry: `git log` usually displays only the branch you are on, and the commit that belongs to it. To see all branches, you only need to add the `--all` option:

```
[22] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* ef6c382 (berries) Add a blackberry
* 0e8b5cf (HEAD -> master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Okay, let's see: we are on the `master` branch, as the shell prompts and as `HEAD` remembers us, with that arrow that points to `master`; then there is a `berries` branch, with a commit more than `master`.

Reachability and undoing commits

Now let's imagine this scenario: we have a new commit on the `berries` branch, but we realized it is a wrong one, so we want the `berries` branch to go back where `master` is. We actually want to discard the last commit on the `berries` branch.

First, check out the `berries` branch:

```
[23] ~/grocery (master)
$ git checkout -
Switched to branch 'berries'
```

New trick: using the dash (-), you actually are saying to Git: "*Move me to the branch I was before switching*"; and Git obeys, moving us to the `berries` branch.

Now a new command, `git reset` (please don't care about the `--hard` option for now):

```
[24] ~/grocery (berries)
$ git reset --hard master
HEAD is now at 0e8b5cf Add an orange
```


WOW! eBook
www.wowebook.org

In Git, this is simple as this. The `git reset` actually **moves a branch from the current position to a new one**; here we said Git to move the current `berries` branch to where `master` is, and the result is that now we have all the two branches pointing to the same commit:

```
[25] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
* 0e8b5cf (HEAD -> berries, master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

You can double-check this looking at `refs` files; this is the `berries` one:

```
[26] ~/grocery (berries)
$ cat .git/refs/heads/berries
0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8
```

And this is the `master` one:

```
[27] ~/grocery (berries)
$ cat .git/refs/heads/master
0e8b5cf1c1b44110dd36dea5ce0ae29ce22ad4b8
```

Same hash, same commit.

A *side effect* of this operation is losing the last commit we did in `berries`, as we already said: but why? And how?

This is due to the **reachability** of commits. A commit is not more reachable when no branches points to it directly, nor it figures as a parent of another commit in a branch. Our *blackberry commit* was the last commit on the `berries` branch, so moving the `berries` branch away from it, made it unreachable, and it *disappears* from our repository.

But are you sure it is gone? Want to make a bet?

Let's try another trick: we can use `git reset` to move the actual branch directly to a commit. And to make things more interesting, let's try to point the *blackberry commit* (if you scroll your shell window backwards, you can see its hash, which for me is `ef6c382`) so, `git reset` to the `ef6c382` commit:

```
[28] ~/grocery (berries)
$ git reset --hard ef6c382
HEAD is now at ef6c382 Add a blackberry
```


WOW! eBook
www.wowebook.org

And then do the usual `git log`:

```
[29] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
* ef6c382 (HEAD -> berries) Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

That's magic! We actually recovered the lost commit!

Okay, jokes aside, there's no magic in Git; it simply won't delete unreachable commits, at least not immediately. It makes some housekeeping automatically at a given time, as it has some powerful **garbage collection** features (look at the `git gc` command help page if you are curious; I want you to remember that any Git command, followed by the `--help` option, will open for you the internal man page for it).

So, we have seen what reachability of commits means, and then learnt how to undo a commit using the `git reset` command, that is a thing to know to take advantage of Git features while working on a repository.

But let's continue experimenting with branches.

Assume you want to add a watermelon to the shopping list, but later you realize you added it to the wrong `berries` branch; so, add `"watermelon"` to the `shoppingList.txt` file:

```
[30] ~/grocery (berries)
$ echo "watermelon" >> shoppingList.txt
```

Then do the commit:

```
[31] ~/grocery (berries)
$ git commit -am "Add a watermelon"
[berries a8c6219] Add a watermelon
 1 file changed, 1 insertion(+)
```

And do a `git log` to check the result:

```
[32] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
* a8c6219 (HEAD -> berries) Add a watermelon
* ef6c382 Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```


WOW! eBook
www.wowebook.org

Now our aim here is: have a new `melons` branch, which the *watermelon commit* have to belong to, then set the house in order and move the `berries` branch back to the *blackberry commit*. To keep the *watermelon commit*, first create a `melon` branch that points to it with the well-known `git branch` command:

```
[33] ~/grocery (berries)
$ git branch melons
```

Let's check:

```
[34] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
* a8c6219 (HEAD -> berries, melons) Add a watermelon
* ef6c382 Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Okay, so now we have both `berries` and `melons` branches pointing to the watermelon commit.

Now we can move the `berries` branch back to the previous commit; let's get advantage of the opportunity to learn something new.

In Git, you often have the need to point to a preceding commit, like in this case, the one before; for this scope, we can use `HEAD` reference, followed by one of two different special characters, the *tilde*~ and the *caret*^. A **caret** basically means *a back step*, while two carets means two steps back, and so on. As you probably don't want to type dozens of carets, when you need to step back a lot, you can use **tilde**: similarly, `~1` means *a back step*, while `~25` means 25 steps back, and so on.

There's more to know about this mechanism, but it is enough for now; for all the details check `http://www.paulboxley.com/blog/2011/06/git-caret-and-tilde`.

So, let's step back our `berries` branch using caret; do a `git reset --hard HEAD^`:

```
[35] ~/grocery (berries)
$ git reset --hard HEAD^
HEAD is now at ef6c382 Add a blackberry
```


WOW! eBook
www.wowebook.org

Let's see the result:

```
[36] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
* a8c6219 (melons) Add a watermelon
* ef6c382 (HEAD -> berries) Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Well done! We successfully recovered the mistake, and learnt how to use the `HEAD` reference and `git reset` command to move branches from here to there.

Just to remark concepts, let's take a look at the `shoppingList.txt` file here in the `berries` branch:

```
[37] ~/grocery (berries)
$ cat shoppingList.txt
banana
apple
orange
blackberry
```

Okay, here we have blackberry, other than the other previously added fruits.

Switch to `master` and check again; check out the `master` branch:

```
[38] ~/grocery (berries)
$ git checkout master
Switched to branch 'master'
```

Then `cat` the file:

```
[39] ~/grocery (master)
$ cat shoppingList.txt
banana
apple
orange
```

Okay, no blackberry here, but only fruits added before the `berries` branch creation.

And now a last check on the `melons` branch; check out the branch:

```
[40] ~/grocery (master)
$ git checkout melons
Switched to branch 'melons'
```


WOW! eBook
www.wowebook.org

And `cat` the `shoppingList.txt` file:

```
[41] ~/grocery (melons)
$ cat shoppingList.txt
banana
apple
orange
blackberry
watermelon
```

Fantastic! Here there is the watermelon, other than fruits previously added while in the `berries` and `master` branches.

Quick tip: while writing the branch name, use *Tab* to autocomplete: Git will write the complete branch name for you.

Detached HEAD

Now it's time to explore another important concept about Git and its references, the `detached HEAD` state.

For the sake of the explanation, go back to the `master` branch and see what happens when we check out the previous commit, moving `HEAD` backward; perform a `git checkout HEAD^`:

```
[42] ~/grocery (master)
$ git checkout HEAD^
Note: checking out 'HEAD^'.


You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at e4a5e7b... Add an apple
```

Wow, a lot of new things to see here. But don't be scared, it's not that complicated: let's take some baby steps into the long message Git showed us.

WOW! eBook
www.wowebook.org

First, think about it: Git is very kind and often tells us loads of useful information in its output messages. Don't under evaluate this behavior: especially at the beginning, reading Git messages allows you to learn a lot, so read them carefully.

Here, Git says we are in a `detached HEAD` state. Being in this state basically means that `HEAD` does not reference a branch, but directly a commit, the `e4a5e7b` one in this case; do a `git log` and see:

```
[43] ~/grocery ((e4a5e7b...))
$ git log --oneline --graph --decorate --all
* a8c6219 (melons) Add a watermelon
* ef6c382 (berries) Add a blackberry
* 0e8b5cf (master) Add an orange
* e4a5e7b (HEAD) Add an apple
* a57d783 Add a banana to the shopping list
```

First of all, in the shell prompt you can see that between rounds, that now are doubled, there is not a branch name, but the first seven characters of the commit, `((e4a5e7b...))`.

Then, `HEAD` is now stuck to that commit, while branches, especially the `master` one, remains at their own place. As a result, the `HEAD` file now contains the hash of that commit, not a ref to a branch as before:

```
[44] ~/grocery ((e4a5e7b...))
$ cat .git/HEAD
e4a5e7b3c64bee8b60e23760626e2278aa322f05
```

Going on, Git says that in this state we can look around, make experiments, doing new commits if we like, and then discard them simply by checking out an existing branch, or save them if you like creating a new branch. Can you say why this is true?

Due to reachability of commits, of course. If we do some commits, then move `HEAD` to an existing branch, those commits become unreachable. They stay in a reachable state until `HEAD` is on top of the last of them, but when you move `HEAD` with a `git checkout`, they are gone. At the same time, if you create a new branch before moving `HEAD`, there will be a label, a pointer Git can use to reach those commits, so they are safe.

Want to try?

Okay, let's have some fun; modify the `shoppingList.txt` file, adding a `bug`:

```
[45] ~/grocery ((e4a5e7b...))
$ echo "bug" > shoppingList.txt
```


WOW! eBook
www.wowebook.org

Then `commit` this voluntary mistake:

```
[46] ~/grocery ((e4a5e7b...))
$ git commit -am "Bug eats all the fruits!"
[detached HEAD 07b1858] Bug eats all the fruits!
 1 file changed, 1 insertion(+), 2 deletions(-)
```

Let's `cat` the file:

```
[47] ~/grocery ((07b1858...))
$ cat shoppingList.txt
bug
```

Ouch, we actually erased all your shopping list files!

What happened in the repository then?

```
[48] ~/grocery ((07b1858...))
$ git log --oneline --graph --decorate --all

  * 07b1858 (HEAD) Bug eats all the fruits!
  | * a8c6219 (melons) Add a watermelon
  | * ef6c382 (berries) Add a blackberry
  | * 0e8b5cf (master) Add an orange
  |/

* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Nice! We have a new commit, the bug one, and we can see as `HEAD` followed us, so now it points to it. Then, the console drew two different `paths`, because starting from `apple commit`, we traced two routes: one that goes to the `master` branch (then to `berries` and `melons`), and one that goes to the `bug commit` we just made.

Okay, so if we now check out `master` again, what happens? Give it a try:

```
[49] ~/grocery ((07b1858...))
$ git checkout master

Warning: you are leaving 1 commit behind, not connected to
any of your branches:

  07b1858 Bug eats all the fruits!

If you want to keep it by creating a new branch, this may be a good time to
do so with:
```


WOW! eBook
www.wowebook.org

```
 git branch <new-branch-name> 07b1858

Switched to branch 'master'
```

Okay, we have already seen this message: Git is aware that we are leaving a commit behind; but in this case, it's not a problem for us, indeed it's actually what we really want.

Let's check the situation:

```
[50] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* a8c6219 (melons) Add a watermelon
* ef6c382 (berries) Add a blackberry
* 0e8b5cf (HEAD -> master) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Yay! The bug commit is gone, so nothing is compromised. In the previous message, Git was kind enough to remind us how to recover that commit, just in case; the trick is to directly create a branch that points to that commit, and Git pinned us even the complete command. Let's try it, creating a `bug` branch:

```
[51] ~/grocery (master)
$ git branch bug 07b1858
```

Let's see what happened:

```
[52] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 07b1858 (bug) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (HEAD -> master) Add an orange
|/
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Wow, that's amazingly simple! The commit is here again, and now we have even a branch to check out if we like.

The reflogs

Okay, but what if we ignore the Git message the first time, then time goes and at the end, we can't remember the hash of the commit we want to retrieve?

WOW! eBook
www.wowebook.org

Git never forgets you. It has another powerful tool in its wrench box, and that is called the **reference log**, or reflog for short. Basically, the reflog (or better the reflogs, as there is one for every reference) records what happens in the repository while you commit, reset, check out, and so on. To be more precise, every reflog records all the times that tips of the branches and other references (such as `HEAD`) where updated.

We can take a look at it with a convenient Git command, `git reflog show`:

```
[53] ~/grocery (master)
$ git reflog show
0e8b5cf HEAD@{0}: checkout: moving from
07b18581801f9c2c08c25cad3b43aeee7420ffdd to master
07b1858 HEAD@{1}: commit: Bug eats all the fruits!
e4a5e7b HEAD@{2}: checkout: moving from master to HEAD^
0e8b5cf HEAD@{3}: reset: moving to 0e8b5cf
e4a5e7b HEAD@{4}: reset: moving to HEAD^
0e8b5cf HEAD@{5}: checkout: moving from melons to master
a8c6219 HEAD@{6}: checkout: moving from master to melons
0e8b5cf HEAD@{7}: checkout: moving from berries to master
ef6c382 HEAD@{8}: reset: moving to HEAD^
a8c6219 HEAD@{9}: commit: Add a watermelon
ef6c382 HEAD@{10}: reset: moving to ef6c382
ef6c382 HEAD@{11}: reset: moving to ef6c382
0e8b5cf HEAD@{12}: reset: moving to master
ef6c382 HEAD@{13}: checkout: moving from master to berries
0e8b5cf HEAD@{14}: checkout: moving from berries to master
ef6c382 HEAD@{15}: commit: Add a blackberry
0e8b5cf HEAD@{16}: checkout: moving from master to berries
0e8b5cf HEAD@{17}: commit: Add an orange
e4a5e7b HEAD@{18}: commit: Add an apple
a57d783 HEAD@{19}: commit (amend): Add a banana to the shopping list
c7a0883 HEAD@{20}: commit (initial): Add a banana to the shopping list
```

Actually, here there are all the movements the `HEAD` reference made in my repository since the beginning, in reverse order, as you may have already noticed.
In fact, the last one (`HEAD@{0}`) says:

```
checkout: moving from 07b18581801f9c2c08c25cad3b43aeee7420ffdd to master
```

Actually, this is the very last thing we did, apart from the creation on the `bug` branch. As we never moved into it, the `HEAD` reflog doesn't log anything about `bug` branch creation.

WOW! eBook
www.wowebook.org

The reflog is a quite complex topic to be discussed in depth, so here we only learn how to open and read it, and how to interpret information from it.

The only things I want you to know are that this log will be cleared at some point; the default retention is 90 days. Then, there is a reflog for every reference; what we are seeing now is the `HEAD` reflog (`HEAD@` is a hint about this), but if you type `git reflog show berries` you will see the movements `berries` branch did in the past:

```
[54] ~/grocery (master)
$ git reflog berries

ef6c382 berries@{0}: reset: moving to HEAD^
a8c6219 berries@{1}: commit: Add a watermelon
ef6c382 berries@{2}: reset: moving to ef6c382
0e8b5cf berries@{3}: reset: moving to master
ef6c382 berries@{4}: commit: Add a blackberry
0e8b5cf berries@{5}: branch: Created from master
```

To go back to our problem, if we want to check out a currently unreachable commit, we can go to the `HEAD` reflog and look for a line where we did the commit (in this example, I would look for a `commit:` logline, searching the one where the commit message says something that helps me to remind, something like `bug` in this case).

Well done, that's enough for now; later we will use reflog again.

Tags are fixed labels

Tags are labels you can pin to a commit, but unlike branches, they will stay there.

Creating a tag is simple: you only need the `git tag` command, followed by a tag name; we can create one in the tip commit of `bug` branch to give it a try; check out the `bug` branch:

```
[1] ~/grocery (master)
$ git checkout bug
Switched to branch 'bug'
```

Then use the `git tag` command followed by the funny `bugTag` name:

```
[2] ~/grocery (bug)
$ git tag bugTag
```


WOW! eBook
www.wowebook.org

Let's see what `git log` says:

```
[3] ~/grocery (bug)
$ git log --oneline --graph --decorate --all
* 07b1858 (HEAD -> bug, tag: bugTag) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (master) Add an orange
|/

* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

As you can see in the log, now on the tip of the `bug` branch there is even a tag named `bugTag`.

If you do a commit in this branch, you will see the `bugTag` will remain at its place; add a new line to the same old shopping list file:

```
[4] ~/grocery (bug)
$ echo "another bug" >> shoppingList.txt
```

Perform a `commit`:

```
[5] ~/grocery (bug)
$ git commit -am "Another bug!"
[bug 5d605c6] Another bug!
 1 file changed, 1 insertion(+)
```

Then look at the current situation:

```
[6] ~/grocery (bug)
$ git log --oneline --graph --decorate --all
* 5d605c6 (HEAD -> bug) Another bug!
* 07b1858 (tag: bugTag) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (master) Add an orange
|/
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

That's exactly what we predict.

WOW! eBook
www.wowebook.org

Tags are useful to give a particular meaning to some particular commits; for instance, as a developer, you maybe want to tag every release of your software: in that case, this is all you need to know to do that job.

Even tags are references, and they are stored, as branches, as simple text files in the tags subfolder within the `.git` folder; take a look under the `.git/refs/tags` folder, you will see a `bugTag` file; look at the content:

```
[7] ~/grocery (bug)
$ cat .git/refs/tags/bugTag
07b18581801f9c2c08c25cad3b43aeee7420ffdd
```

As you maybe have already predicted, it contains the hash of the commit it refers to.

To delete a tag, you have to simply append the `-d` option: `git tag -d <tag name>`.

As you can't move a tag, if you need to move it you have to delete the previous tag and then create a new one with the same name that points to the commit you want; you can create a tag that points a commit wherever you want, appending the hash of the commit as an argument, for example, `git tag myTag 07b1858`.

Annotated tags

Git has two kinds of tags; this is because in some situations you may want to add a message to the tag, or because you like to have the author stick to it.

We already have seen the first type, the simpler one; tags containing this extra information load belong to the second type, the **annotated tag**.

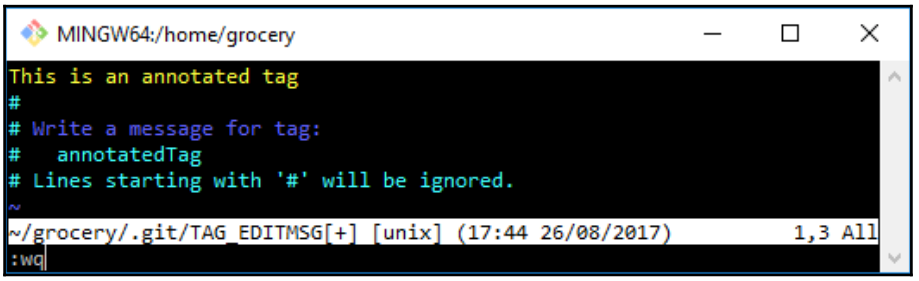
An annotated tag is both a *reference* and a *git object* such as commits, trees, and blobs.

To create one, simply append `-a` to the command; let's create another one to give this a try:

```
[8] ~/grocery (bug)
$ git tag -a annotatedTag 07b1858
```


WOW! eBook
www.wowebook.org

At this point Git opens the default editor, to allow you to write the tag message, as in the following screenshot:



Save and exit, and then see the log:

```
[9] ~/grocery (bug)
$ git log --oneline --graph --decorate --all
* 5d605c6 (HEAD -> bug) Another bug!
* 07b1858 (tag: bugTag, tag: annotatedTag) Bug eats all the fruits!
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
| * 0e8b5cf (master) Add an orange
|/

* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Okay, there are two tags now on the same commit.

A new ref has been created:

```
[10] ~/grocery (bug)
$ cat .git/refs/tags/annotatedTag
17c289ddf23798de6eee8fe6c2e908cf0c3a6747
```

But even a new object: try to `cat-file` the hash you see in the reference:

```
[11] ~/grocery (bug)
$ git cat-file -p 17c289
object 07b18581801f9c2c08c25cad3b43aeee7420ffdd
type commit
tag annotatedTag
tagger Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 150376226 4
+0200

This is an annotated tag
```


WOW! eBook
www.wowebook.org

This is how an annotated tag looks like.

Obviously, the `git tag` command has many other options, but I only highlighted the ones I think are worth knowing at the moment.

If you want to look at all the options of a command, remember you can always do a `git <command> --help` to see the complete guide.

Time to spend some words on the staging area, as we have only scratched the surface.

Staging area, working tree, and HEAD commit

Until now, we have barely named the **staging area** (also known as an **index**), while preparing files to make a new commit with the `git add` command.

Well, the staging area purpose is actually this. When you change the content of a file, when you add a new one or delete an existing one, you have to tell Git what of these modifications will be part of the next commit: the staging area is the container for this kind of data.

Let's focus on this right now; move to the `master` branch, if not already there, then type the `git status` command; it allows us to see the actual status of the staging area:

```
[1] ~/grocery (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Git says there's nothing to commit, our working tree is clean. But what's a **working tree**? Is it the same as the working directory we talked about? Well, yes and no, and it's confusing, I know.

Git had (and still have) some troubles with names; in fact, as we said a couple of lines before, even for the staging area we have two names (the other one is index). Git uses both in its messages and commands output, and the same often does people, blogs, and books like this one while talking about Git. Having two names for the same thing is not always a good idea, especially when they represent exactly the same thing, but being aware of this is enough (time will give us a less confusing Git, I'm sure).

WOW! eBook
www.wowebook.org

For the working tree and working directory, the story is this. At some point, someone argued: *If I'm in the root of the repository I'm in a working directory, but if I walk through a subfolder, I'm in another working directory*. This is technically true by a filesystem perspective, but while in Git, doing some operations such as checkout or reset does not affect the current working directory, but the entire... working tree. So, to avoid confusion, Git stopped talking about working directory in its messages and "renamed" it as working tree. This is the commit on Git repository that made this change: `https://github.com/git/git/commit/2a0e6cdedab306eccbd297c051035c13d0266343`, if you want to go in deep. Hope I've clarified a little bit.

Back on topic now.

Add a `peach` to the `shoppingList.txt` file:

```
[2] ~/grocery (master)
$ echo "peach" >> shoppingList.txt
```

Then make use of this new learnt command again, `git status`:

```
[3] ~/grocery (master)
$ 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:   shoppingList.txt

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

Okay, now it's time to learn about **staged** changes; with the word `staged`, Git means modifications we already added to the staging area, so they will be part of the next commit. In the current situation, we modified the `shoppingList.txt` file, but we have not added it yet to the staging area (using the good old `git add` command).

So, Git informs us: it tells that there is a modified file (in red color), and then offers two possibilities: *stage* it (add it to the staging area), or *discard* the modification, using the `git checkout -- <file> command`.

WOW! eBook
www.wowebook.org

Let's try to add it; we will see the second option later.

So, try a `git add` command, with nothing more:

```
[4] ~/grocery (master)
$ git add
Nothing specified, nothing added.
Maybe you wanted to say 'git add .'?
```

Okay, new thing learnt: `git add` wants you to specify something to add. A common thing is to use the dot `.` as a wildcard, and this by default means, *add all the files in this folder and subfolders to the staging area*. This is the same as `git add -A` (or `--all`), and by "all" I mean:

- **Files in this folder and sub-folders I added in the past at least one time**: This set of files is also known as the **tracked files**
- **New files**: These are called **untracked files**
- Files marked for deletion

Be aware that this behavior changed over time: before Git 2.x, `git add .` and `git add -A` had different effects. Here is a table for quickly understanding the differences.

Git version 1.x:

	New files	**Modified files**	**Deleted files**	
`git add -A`	yes	yes	yes	Stage all (new, modified, deleted) files
`git add .`	yes	yes	no	Stage new and modified files only
`git add -u`	no	yes	yes	Stage modified and deleted files only

Git version 2.x:

	New files	**Modified files**	**Deleted files**	
`git add -A`	yes	yes	yes	Stage all (new, modified, deleted) files
`git add .`	yes	yes	**yes**	Stage all (new, modified, deleted) files
`git add --ignore-removal .`	**yes**	**yes**	**no**	Stage new and modified files only
`git add -u`	no	yes	yes	Stage modified and deleted files only

WOW! eBook
www.wowebook.org

As you can see, in Git 2.x there's a new way to stage new and modified files only, the `git add --ignore-removal .` way, and then `git add .` became the same as `git add -A`. If you are wondering, the `-u` option is the equivalent of `--update`.

Another basic usage is to specify the file we want to add; let's give it a try:

```
[5] ~/grocery (master)
$ git add shoppingList.txt
```

As you can see, when `git add` goes right, Git says nothing, no messages: let's consider it a tacit approval.

Other ways to add files is specifying a directory to add all the changed files within it, using wildcards such as the star `*` with or without something else (for example, `*.txt` for adding all `txt` files, `foo*` for adding all files starting with `foo` and so on).
Please refer to `https://git-scm.com/docs/git-add#git-add-ltpathspecgt82308203` for all the information.

Okay, time to look back at our repository; go with a `git status` now:

```
[6] ~/grocery (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   shoppingList.txt
```

Nice! Our file has been added to the staging area, and now it is one of the changes that will be part of the next commit, the only one actually.

Now take a look at what Git says then: if you want to `unstage` the change, you can use the `git reset HEAD` command: what does it mean? **Unstage** is a word to say *remove a change from the staging area*, for example, because we realized we want to add that change not in the next commit, but later.

WOW! eBook
www.wowebook.org

For now, leave things how they are, and do a `commit`:

```
[7] ~/grocery (master)
$ git commit -m "Add a peach"
[master 603b9d1] Add a peach
 1 file changed, 1 insertion(+)
```

Check the status:

```
[8] ~/grocery (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Okay, now we have a new `commit` and our working tree is clean again; yes, because the effect of `git commit` is to create a new commit with the content of the staging area, and then empty it.

Now we can make some experiments and see how to deal with the staging area and working tree, undoing changes when in need.

So, follow me and make things more interesting; add an onion to the shopping list and then add it to the staging area, and then add a garlic and see what happens:

```
[9] ~/grocery (master)
$ echo "onion" >> shoppingList.txt

[10] ~/grocery (master)
$ git add shoppingList.txt

[11] ~/grocery (master)
$ echo "garlic" >> shoppingList.txt

[12] ~/grocery (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   shoppingList.txt

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:   shoppingList.txt
```


WOW! eBook
www.wowebook.org

Okay, good! We are in a very interesting state now. Our `shoppingList.txt` file has been modified two times, and only the first modification has been added to the staging area. This means that at this point if we would commit the file, only the `onion` modification would be part of the commit, but not the `garlic` one. This is a thing to underline, as in other versioning systems it is not so simple to do this kind of work.

To highlight the modification we did, and take a brief look, we can use the `git diff` command; for example, if you want to see the difference between the working tree version and the staging area one, try to input only the `git diff` command without any option or argument:

```
[13] ~/grocery (master)
$ git diff
diff --git a/shoppingList.txt b/shoppingList.txt
index f961a4c..20238b5 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -3,3 +3,4 @@ apple
 orange
 peach
 onion
+garlic
```

As you can see, Git highlights the fact that in the working tree we have a `garlic` more than the staging area version.

The last part of the output of the `git diff` command is not difficult to understand: green lines starting with a plus + symbol are new lines added (there would be red lines starting with a minus – for deleted lines). A modified line will be usually highlighted by Git with a minus red deleted line and a plus green added line; to be `true`, Git can be instructed to use different `diff` algorithms, but this is out of the scope of this book.

Other than this, the first part of the `git diff` output is a little bit too difficult to explain in a few words; please refer to `https://git-scm.com/docs/git-diff` for all the details.

But what if you want to see the differences between the last committed version of the `shoppingList.txt` file and the one added into the staging area?

We have to use the `git diff --cached HEAD` command:

```
[14] ~/grocery (master)
$ git diff --cached HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..f961a4c 100644
--- a/shoppingList.txt
```


WOW! eBook
www.wowebook.org

```
+++ b/shoppingList.txt
@@ -2,3 +2,4 @@ banana
 apple
 orange
 peach
+onion
```

We have to dissect this command to better understand what's the purpose; appending the `HEAD` argument, we are asking to use the last commit we did as a subject of the compare. To be `true`, in this case, the `HEAD` reference is optional, as it is the default: `git diff --cached` would return the same result.
Instead, the `--cached` option says, *compare the argument (HEAD in this case) with the version in the staging area*.

Yes, dear friends: the staging area, also known as an `index`, sometimes is called `cache`, hence the `--cached` option.

The last experiment that we can do is compare the `HEAD` version with the working tree one; let's do it with a `git diff HEAD`:

```
[15] ~/grocery (master)
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..20238b5 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -2,3 +2,5 @@ banana
 apple
 orange
 peach
+onion
+garlic
```

Okay, it works as expected.

Now it's time to take a break from the console and spend a couple of words to talk about these three `locations` we compared.

WOW! eBook
www.wowebook.org

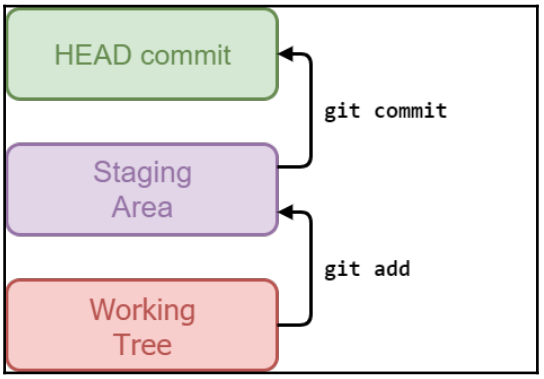
The three areas of Git

In Git, we work at three different levels:

- The **working tree** (or working directory)
- The **staging area** (or index, or cache)
- The **HEAD commit** (or the last commit or tip commit on the current branch)

When we modify a file, we are doing it at working tree level; when we do a `git add`, we are actually copying the changes from the working tree to the staging area. At the end, when we do a `git commit`, we finally move the changes from the staging area to a brand new commit, referenced by `HEAD`, which will become part of our repository history: this is what I mean with the HEAD commit.

The following figure draws those three areas:



We can move changes between these areas forward, from the working tree to the `HEAD` commit, but we can even go backward, undoing changes if we like.

We already know how to go forward using `git add` then `git commit`; let's take a look at the commands to go backward.

Removing changes from the staging area

It happens that you add changes to the staging area, then you realize they fit better in a future commit, not in the commit you are composing right now.

WOW! eBook
www.wowebook.org

To remove those changes to one or more files from the staging area, you can use the `git reset HEAD <file>` command; get back the shell and follow me.

Check the repository current status:

```
[16] ~/grocery (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   shoppingList.txt

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:   shoppingList.txt
```

This is the actual situation, remember? We have an `onion` in the staging area and a `garlic` more in the working tree.

Now go with a `git reset HEAD`:

```
[17] ~/grocery (master)
$ git reset HEAD shoppingList.txt
Unstaged changes after reset:
M   shoppingList.txt
```

Okay, Git confirms we unstaged changes. The `M` on the left side means `Modified`; here Git is telling us we have unstaged a modification to a file. If you create a new file and you add it to the staging area, Git knows this is a new file; if you try to unstage it, Git would put an `A` for `Added` in the left, to remember that you just unstaged the addition of a new file. Same if you unstage the deletion of an existing file: on the left would appear a `D` for `Deleted`.

Well, time to verify what happened:

```
[18] ~/grocery (master)$ 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:
shoppingList.txtno changes added to commit (use "git add" and/or "git
commit -a")
```


WOW! eBook
www.wowebook.org

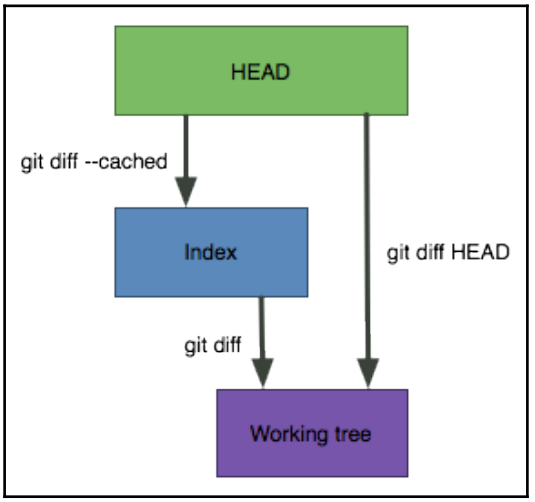
Okay, by using `git status` we see that now the staging area is empty, there's nothing staged. We only have some unstaged modification, but what modification? Did the `git reset HEAD` actually delete the onion?

Let's verify this using the `git diff` command:

```
[19] ~/grocery (master)
$ git diff
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..20238b5 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -2,3 +2,5 @@ banana
 apple
 orange
 peach
+onion
+garlic
```

No, fortunately! The `git reset HEAD` command won't destroy your modification; it only moves away them from the staging area, so they will not be part of the next commit.

The following figure shows a quick summary of `git diff` different behaviors:



Now imagine that we completely messed up: the modification we did to the `shoppingList.txt` file is wrong (in fact they are, no tasty fruit salad with onion and garlic), so we need to undo them.

WOW! eBook
www.wowebook.org

The command for that is `git checkout -- <file>`, as Git gently reminds in `the git status` output message. Give it a try:

```
[20] ~/grocery (master)
$ git checkout -- shoppingList.txt
```

Check the status:

```
[21] ~/grocery (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Check the content of the file:

```
[22] ~/grocery (master)
$ cat shoppingList.txt
banana
apple
orange
peach
```

That's it! We actually removed onion and garlic from the shopping list file. But be aware: we lost them! As those modifications were only in the working tree, there's no way to reclaim them, so be careful: `git checkout --` is a `destructive` command, use it carefully.

Other than this, we need to remember that `git checkout` overwrites even the staging area; as per the preceding figure, working tree and `HEAD` commit are in direct relationship: changes always go through the staging area. Later we will grasp this concept better while delving into `git reset` options.

At this point you maybe have noticed here we used `git reset` and `git checkout` commands in a different way than we did in the preceding sections, and this is true.

At the beginning, this can be a little confusing for the newcomers, because mentally you cannot associate a single command to a single operation as it can be used to do more than one. For example, you cannot say, *git checkout is for branch switching* (or for commit inspection, going to a detaching `HEAD` state), as it can be used even to discard changes in a working tree, as we just did.

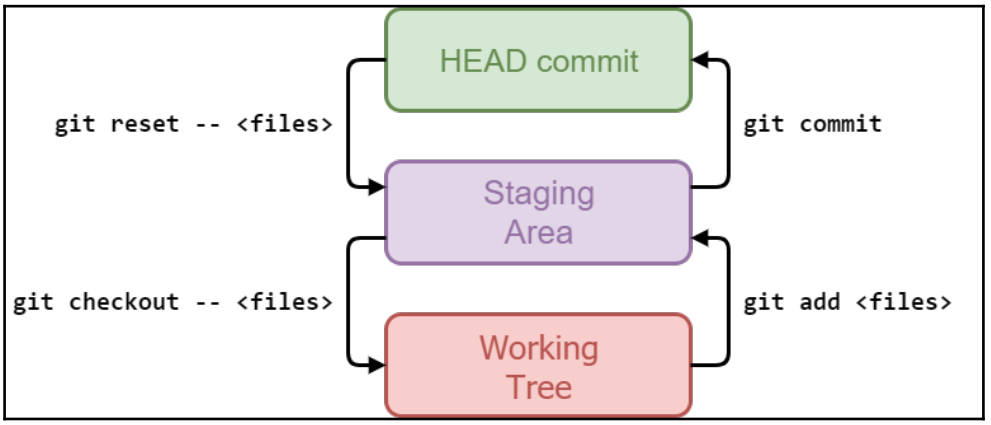
The trick you can use to differentiate the two variations for these commands is to take into account that double-dash -- notation. So, you can remember *git checkout is for switching branches* and *git checkout -- is for discarding local changes*.

WOW! eBook
www.wowebook.org

This is `true` even for the `git reset` command; in fact, do a `git reset -- <file>` is actually the same as doing a `git reset HEAD <file>`.

To be true, the double-dash `--` notation is not mandatory; if you do a `git checkout <file>` or `git reset <file>` without `--`, in 99% of cases Git does what you expect. The double-dash is needed when, due to a coincidence, there is a file and a branch with the same name: in that case, Git has to know if you want to deal with branches, for example switching to another one with `git checkout`, or if you want to manage files. In this situation, the double-dash is the way to tell Git *I want to handle files, not branches*.

The following figure summarizes the commands to move changes between those three areas:



Now it's time to complete our cultural baggage about file status lifecycle within a repository.

File status lifecycle

In a Git repository, files pass through some different states. When you first create a file in the working tree, Git notices it and tells you there's a new untracked file; let's try to create a new `file.txt` in our `grocery` repository and see the output of `git status`:

```
[23] ~/grocery (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
```


WOW! eBook
www.wowebook.org

```
file.txt

nothing added to commit but untracked files present (use "git add" to
track)
```

As you can see, Git explicitly says that there's an **untracked** file; an untracked file is basically a new file Git has never seen before.

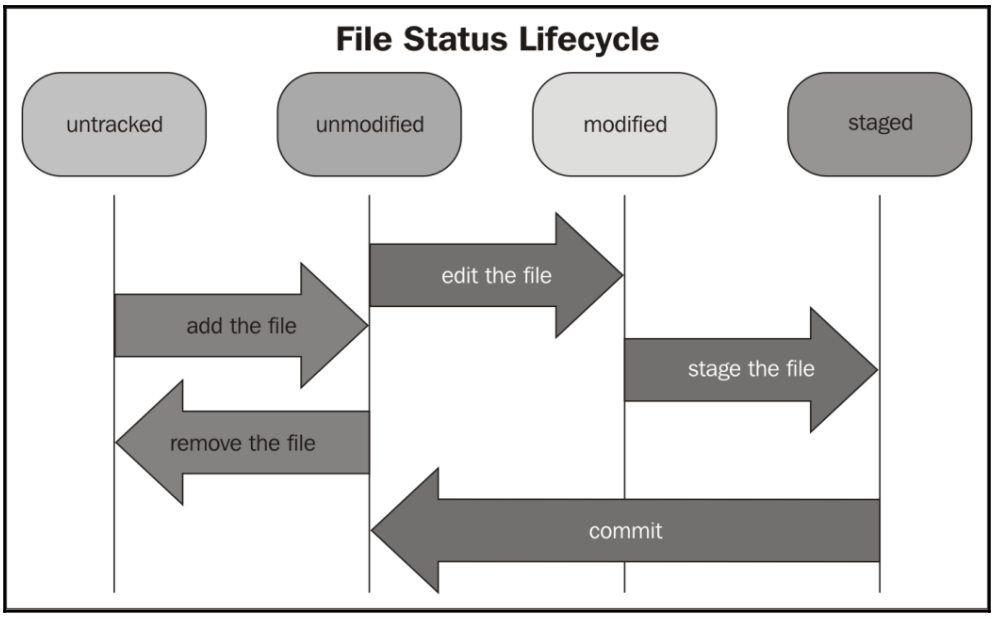
When you add it, it becomes a **tracked** file.

If you commit the file, then it goes in an **unmodified** state; this means Git knows it, and the current version of the file in the working tree is the same as the one in the `HEAD` commit.

If you make some changes, the file goes to a **modified** state.

Adding a modified file to the staging area makes it a **staged** file.

The following figure summarizes these states:



Knowing this terminology is important to better understand Git messages, and it helps me and you to go smoothly while talking about files in a Git repository.

Now it's time to go a little bit deep with `git reset` and `git checkout` commands.

WOW! eBook
www.wowebook.org

All you need to know about checkout and reset

First of all, we need to do some housekeeping. Go back to the `grocery` repository and clean up the working tree; double-check that you are in the `master` branch, and then do a `git reset --hard master`:

```
[24] ~/grocery (master)
$ git reset --hard master
HEAD is now at 603b9d1 Add a peach
```

This allows us to discard all the latest changes and go back to the latest commit on `master`, cleaning up even the staging area.

Then, delete the `bug` branch we created some time ago; the command to delete a branch is again the `git branch` command, this time followed by a `-d` option and then the branch name:

```
[25] ~/grocery (master)
$ git branch -d bug
error: The branch 'bug' is not fully merged.
If you are sure you want to delete it, run 'git branch -D bug'.
```

Okay, Git has an objection. It says the branch is not fully merged, in other words, *if you delete it, the commit within it will be lost*. No problem, we don't need that commit; so, use the capital `-D` option to force the deletion:

```
[26] ~/grocery (master)
$ git branch -D bug
Deleted branch bug (was 07b1858).
```

Okay, now we are done, and the repository is in good shape, as the `git log` command shows:

```
[27] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 603b9d1 (HEAD -> master) Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```


WOW! eBook
www.wowebook.org

Git checkout overwrites all the tree areas

Now switch to the `melons` branch using the `git checkout` command:

```
[28] ~/grocery (master)
$ git checkout melons
Switched to branch 'melons'
```

Check the log:

```
[29] ~/grocery (melons)
$ git log --oneline --graph --decorate --all
* 603b9d1 (master) Add a peach
| * a8c6219 (HEAD -> melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Okay, what happened?

Git took the tip commit in the `melons` branch, analyzed it, and then rebuilt the snapshot the commit represents into our working tree. It basically copied all those files and folders into the staging area and then into the working tree.

Remember that `git checkout` can destroy changes in your working tree; in fact, if you have some local modification Git will block you.

We can try it; add a `potato` to the shopping list file:

```
[30] ~/grocery (melons)
$ echo "potato" >> shoppingList.txt
```

Then checkout `master`:

```
[31] ~/grocery (melons)
$ git checkout master
error: Your local changes to the following files would be overwritten by
checkout:
shoppingList.txt
Please commit your changes or stash them before you switch branches.
Aborting
```


WOW! eBook
www.wowebook.org

As you can see, you cannot switch branch if you are not in a clean state.

Now please remove the potato from the shopping list file, by an editor or by Git (I leave this to you as an exercise).

Git reset can be hard, soft, or mixed

Finally, you will see what `git reset --hard` means, and what are the other reset options that we have.

To avoid messing up our repo again, go into a detached `HEAD` state, so at the end it will be easier to throw all the things away. To do this, checkout directly the penultimate commit on the `master` branch:

```
[32] ~/grocery (master)
$ git checkout HEAD~1
Note: checking out 'HEAD~1'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 0e8b5cf... Add an orange
```

Okay, this is the content of the `shoppingList.txt` file in this commit:

```
[33] ~/grocery ((0e8b5cf...))
$ cat shoppingList.txt
banana
apple
orange
```

Now just replicate the `onion` and `garlic` situation we used before: append an `onion` to the file and add it to the staging area, and then add a `garlic`:

```
[34] ~/grocery ((0e8b5cf...))
$ echo "onion" >> shoppingList.txt

[35] ~/grocery ((0e8b5cf...))
$ git add shoppingList.txt
```


WOW! eBook
www.wowebook.org

```
[36] ~/grocery ((0e8b5cf...))
$ echo "garlic" >> shoppingList.txt

[37] ~/grocery ((0e8b5cf...))
$ git status
HEAD detached at 0e8b5cf
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

modified:   shoppingList.txt

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:   shoppingList.txt
```

Now use the `git diff` command to be sure we are in the situation we desire; check the differences with the staging area:

```
[38] ~/grocery ((0e8b5cf...))
$ git diff --cached HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index edc9072..063aa2f 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,3 +1,4 @@
 banana
 apple
 orange
+onion
```

Check the differences between the working tree and `HEAD` commit:

```
[39] ~/grocery ((0e8b5cf...))
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index edc9072..93dcf0e 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,3 +1,5 @@
 banana
 apple
 orange
+onion
+garlic
```


WOW! eBook
www.wowebook.org

Okay, we have a `HEAD` commit with only fruits, and then a staging area with an onion more and working tree with a garlic more.

Now try to do a **soft reset** to the `master` branch, with the `git reset --soft master` command:

```
[40] ~/grocery ((0e8b5cf...))
$ git reset --soft master
```

Diff to the staging area:

```
[41] ~/grocery ((603b9d1...))
$ git diff --cached HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..063aa2f 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,4 +1,4 @@
 banana
 apple
 orange
-peach
+onion
```

What did Git do? It basically moved the `HEAD` reference to the last commit in the `master` branch, the `603b9d1`. Small break: note as when in detached `HEAD` state, even if you reset to a commit with a branch label, Git continues to reference directly the commit, not the branch.

Okay, having done this, now the differences between the `HEAD` commit and staging area are those we see in the output: the peach that was part of the `shoppingList.txt` file in the `HEAD` commit is not part of the currently staged `shoppingList.txt` file, so Git marks a `peach` text line in red with a preceding minus, to tell you *actually this line has been deleted*, while the `onion` one has been added.

The same is if you compare the `HEAD` commit with a working tree:

```
[42] ~/grocery ((603b9d1...))
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..93dcf0e 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,4 +1,5 @@
 banana
 apple
 orange
```


WOW! eBook
www.wowebook.org

```
-peach
+onion
+garlic
```

In this case, Git even notes that two new lines have been added, `onion` and `garlic`.

This soft-reset technique can help you quickly compare changes between two commits, as it only overwrites the `HEAD` commit area.

Another option is the **mixed reset**; you can do it using the `--mixed` option (or simply using no options, as this is the default):

```
[43] ~/grocery ((603b9d1...))
$ git reset --mixed master
Unstaged changes after reset:
M   shoppingList.txt
```

Okay, there's something different here: Git tells us about unstaged changes. In fact, the `--mixed` option makes Git overwrite even the staging area, not only the `HEAD` commit. If you check differences between the `HEAD` commit and staging area with `git diff`, you will see that there are no differences:

```
[44] ~/grocery ((603b9d1...))
$ git diff --cached HEAD
```

Instead, differences arise between the `HEAD` commit and working tree:

```
[45] ~/grocery ((603b9d1...))
$ git diff HEAD
diff --git a/shoppingList.txt b/shoppingList.txt
index 175eeef..93dcf0e 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -1,4 +1,5 @@
 banana
 apple
 orange
-peach
+onion
+garlic
```

This mixed-reset technique can be useful, for example, to clean up all the staged changes in one shot, with a simple `git reset HEAD`.

WOW! eBook
www.wowebook.org

At this point, you can presume what is the purpose of the `--hard` option: it overwrites all the three areas:

```
[46] ~/grocery ((603b9d1...))
$ git reset --hard master
HEAD is now at 603b9d1 Add a peach

[47] ~/grocery ((603b9d1...))
$ git diff --cached HEAD

[48] ~/grocery ((603b9d1...))
$ git diff HEAD
```

In fact, now there are no differences at any level.

This hard-reset technique is used to completely discard all the changes we did, with a `git reset --hard HEAD` command, as we did in our previous experiments.

We are done. Now we know a little more about both the `git checkout` and `git reset` command; but before leaving, go back in a non-detached `HEAD` state, checking out the `master` branch:

```
[49] ~/grocery ((603b9d1...))
$ git checkout master
Switched to branch 'master'
```

Rebasing

Now I want to tell you something about the `git rebase` command; a **rebase** is a common term while using a versioning system, and even in Git this is a hot topic.

Basically, with `git rebase` you **rewrite history**; with this statement, I mean you can use rebase command to achieve the following:

- Combine two or more commits into a new one
- Discard a previous commit you did
- Change the starting point of a branch, split it, and much more

WOW! eBook
www.wowebook.org

Reassembling commits

One of the widest uses of the `git rebase` command is for reordering or combining commits. For this first approach, imagine you have to combine two different commits.

Suppose we erroneously added half a grape in the `shoppingList.txt` file, then the other half, but at the end we want to have only one commit for the entire grape; follow me with these steps.

Add a `gr` to the shopping list file:

```
[1] ~/grocery (master)
$ echo -n "gr" >> shoppingList.txt
```

The `-n` option is for not adding a new line.

Cat the file to be sure:

```
[2] ~/grocery (master)
$ cat shoppingList.txt
banana
apple
orange
peach
gr
```

Now perform the first commit:

```
[3] ~/grocery (master)
$ git commit -am "Add half a grape"
[master edac12c] Add half a grape
 1 file changed, 1 insertion(+)
```

Okay, we have a commit with half a grape. Go on and add the other half, `ape`:

```
[4] ~/grocery (master)
$ echo -n "ape" >> shoppingList.txt
```


WOW! eBook
www.wowebook.org

Check the file:

```
[5] ~/grocery (master)
$ cat shoppingList.txt
banana
apple
orange
peach
grape
```

Perform the second commit:

```
[6] ~/grocery (master)
$ git commit -am "Add the other half of the grape"
[master 4142ad9] Add the other half of the grape
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Check the log:

```
[7] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 4142ad9 (HEAD -> master) Add the other half of the grape
* edac12c Add half a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Well, this is inconvenient: I'd like to have only a commit with the entire `grape`.

We can repair the mistake with an **interactive rebase**. To do this, we have to rebase the last two commits, creating a new one that is, in fact, the sum of the two.

So, type `git rebase -i HEAD~2` and see what happen; `-i` means *interactive*, while the `HEAD~2` argument means *I want to rebase the last two commits*.

WOW! eBook
www.wowebook.org

This is a screenshot of the console:

```
MINGW64:/home/grocery
pick edac12c Add half a grape
pick 4142ad9 Add the other half of the grape

# Rebase 603b9d1..4142ad9 onto 603b9d1 (2 commands)
#
# 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
# d, drop = remove commit
#
# 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
~
~
~/grocery/.git/rebase-merge/git-rebase-todo [unix] (14:10 26/08/2017)  1,1 All
```

As you can see in the preceding screenshot, Git opens the default editor, Vim. Then it tells us how to edit this temporary file (you can see the location at the bottom of the screenshot) using some commented lines (those starting with #).

Let's read this message carefully.

Here we can reorder the commit lines; doing only this, we basically change the order of commits in our repository. Maybe this can seem a not-so-useful feature, but it can be so if you plan to create new branches after this rebase and you want to clear the ground before.

Then you can delete lines: if you delete a line, basically you drop the corresponding commit.

WOW! eBook
www.wowebook.org

Finally, for every line (every commit), you can use one of the following commands, as per the comments showed in the Vi editor:

- # **p, pick = use commit**: If you pick a commit, the commit will continue to be part of your repo. Think at it as, *Okay, I want to preserve this commit as is*.
- # **r, reword = use commit, but edit the commit message**: Reword allows you to change the commit message, useful if you realized you wrote something wrong in it. It's kind of, *Okay, I want to preserve this commit, but I want to change the message*.
- # **e, edit = use commit, but stop for amending**: When you amend a commit you basically want to reassemble it. For example, you forgot to include a file in it, or you added one too many. If you mark a commit to be edited, Git will stop the subsequent rebase operations to let you do what you need. So, the commit will be preserved, but it will be altered.
- # **s, squash = use commit, but meld into previous commit**: Squash is a term we will see again; it basically means put together two commits or more. In this case, if you squash a commit, it will be removed, but the changes within it will be part of the preceding commit. This is maybe the command we need?
- # **f, fixup = like "squash", but discard this commit's log message**: Fixup is like squash, but let's provide you with a new commit message. This is definitely what I need; as I want the new `grape` commit to have a new message.
- # **x, exec = run command (the rest of the line) using shell**: Exec is, well, advanced stuff. You basically tell Git to run a particular command when it will manipulate this commit during the following rebase actions. This can be useful to do something you forgot between two commits, rerun some tests, or whatever.
- # **d, drop = remove commit**: Drop simply removes the commit, the same as deleting the entire line.

Okay, now we can proceed. We have to modify this file using those commands, and then save it and exit: Git will then continue the rebase process executing every command in order, from top to bottom.

To resolve our issue, I will *reword* the first commit and then *fixup* the second; the following is a screenshot of my console:

WOW! eBook
www.wowebook.org

```
MINGW64:/home/grocery
reword edac12c Add half a grape
f 4142ad9 Add the other half of the grape

# Rebase 603b9d1..4142ad9 onto 603b9d1 (2 commands)
#
# 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
# d, drop = remove commit
#
# 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
~
~
<grocery/.git/rebase-merge/git-rebase-todo[+] [unix] (14:10 26/08/2017)2,6 All
:wq
```

Note that you can use the long format of the command or the short one (for example, `f` –> short, fixup –> long).

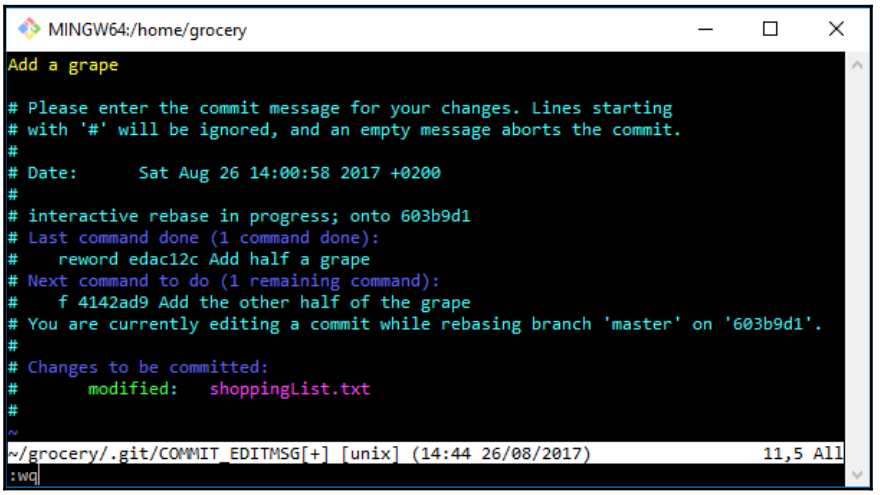
Okay, now Git does the work and then opens a new temporary file to allow us to write the new message for the commit we decided to reword, that is, the first one. The following is the screenshot:

```
MINGW64:/home/grocery
Add half a grape

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sat Aug 26 14:00:58 2017 +0200
#
# interactive rebase in progress; onto 603b9d1
# Last command done (1 command done):
#    reword edac12c Add half a grape
# Next command to do (1 remaining command):
#    f 4142ad9 Add the other half of the grape
# You are currently editing a commit while rebasing branch 'master' on '603b9d1'.
#
# Changes to be committed:
#       modified:   shoppingList.txt
#
~
~/grocery/.git/COMMIT_EDITMSG [unix] (14:44 26/08/2017)                1,1 All
```


WOW! eBook
www.wowebook.org

Note as Git tell us word for word what it is going to do.

Now edit the message, and then save and exit, like in the following screenshot:



Press `ENTER` and we're done.

This is the final message from Git:

```
[8] ~/grocery (master)
$ git rebase -i HEAD~2
unix2dos: converting file C:/Users/san/Google
Drive/Packt/PortableGit/home/grocery/[detached HEAD 53c73dd] Add a grape
 Date: Sat Aug 26 14:00:58 2017 +0200
 1 file changed, 1 insertion(+)
Successfully rebased and updated refs/heads/master.
```

Take a look at the log:

```
[9] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```


WOW! eBook
www.wowebook.org

Wonderful! We just accomplished our mission.

Now let's make a little experiment on rebasing branches.

Rebasing branches

With the `git rebase` command you can also modify the story of branches; one of the things you do more often inside a repository is to change - or better to say - move the point where a branch started, bringing it to another point of the tree. This operation makes it possible to keep low the level of ramifications that would instead be generated using the command `git merge`, which we will see later.

In order to better understand this, let me give you an example.

Let's imagine that to the commit where the orange was added, a branch `nuts` was created in the past, to which a walnut was added.

At this point, let's imagine that we want to move this branch above, to the point where it is now `master` as if this branch had been created starting from there and not from the orange commit.

Let's see how this can be achieved using the `git rebase` command.

Let's start by creating a new branch that points to commit `0e8b5cf`, the orange one:

```
[1] ~/grocery (master)
$ git branch nuts 0e8b5cf
```

This time I used the `git branch` command followed by two arguments, the name of the branch and the commit where to stick the label. As a result, a new `nuts` branch has been created:

```
[2] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf (nuts) Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```


WOW! eBook
www.wowebook.org

Move `HEAD` to the new branch with the `git checkout` command:

```
[3] ~/grocery (master)
$ git checkout nuts
Switched to branch 'nuts'
```

Okay, now it's time to add a `walnut`; add it to the `shoppingList.txt` file:

```
[4] ~/grocery (nuts)
$ echo "walnut" >> shoppingList.txt
```

Then do the commit:

```
[5] ~/grocery (nuts)
$ git commit -am "Add a walnut"
[master 3d3ae9c] Add a walnut
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Check the log:

```
[6] ~/grocery (nuts)
$ git log --oneline --graph --decorate --all
* 9a52383 (HEAD -> nuts) Add a walnut
| * 6409527 (master) Add a grape
| * 603b9d1 Add a peach
|/
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

As you can see, the graph is now a little bit more complicated; starting from the `orange` commit, there are three branches: the `berries`, the `master`, and the `nuts` ones.

Now we want to move the `nuts` branch starting point, form an orange commit to a `grape` commit, as if the `nuts` branch is just one commit next to the `master`.

WOW! eBook
www.wowebook.org

Let's do it, rebasing the `nuts` branch on top of `master`; double-check that you actually are in the `nuts` branch, as a rebase command basically rebases the current branch (`nuts`) to the target one, `master`; so:

```
[7] ~/grocery (nuts)
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add a walnut
Using index info to reconstruct a base tree...
M   shoppingList.txt
Falling back to patching base and 3-way merge...
Auto-merging shoppingList.txt
CONFLICT (content): Merge conflict in shoppingList.txt
Patch failed at 0001 Add a walnut
The copy of the patch that failed is found in: .git/rebase-apply/patch

When you have resolved this problem, run "git rebase --continue".
If you prefer to skip this patch, run "git rebase --skip" instead.
To check out the original branch and stop rebasing, run "git rebase --
abort".

error: Failed to merge in the changes.
```

Okay, don't be scared: the rebase failed, but that's not a problem. In fact, it failed because Git cannot merge differences between `shoppingList.txt` file versions automatically.

Read the message: Now you have three choices:

1. Fix the merge conflicts and then continue, with `git rebase -continue.`
2. Skip this step, and discard the modification using `git rebase -skip.`
3. Abort the rebase, using `git rebase -abort.`

We will choose the first option, but I want to tell you something about the second and third ones.

While rebasing, Git internally creates patches and applies them to the commits we are moving; in fact, while rebasing a branch you actually move all its commits on top of another commit of choice, in this case the last commit on the `master` branch.

In this case, the `nuts` branch has only one commit, so Git compared the destination commit (the grape commit on `master`) with a walnut commit on the `nuts` branch. At the end, only one comparing and patching step will be necessary (this is the meaning of the `REBASE 1/1` message on the console: you are rebasing commit 1 of 1 total commits to rebase).

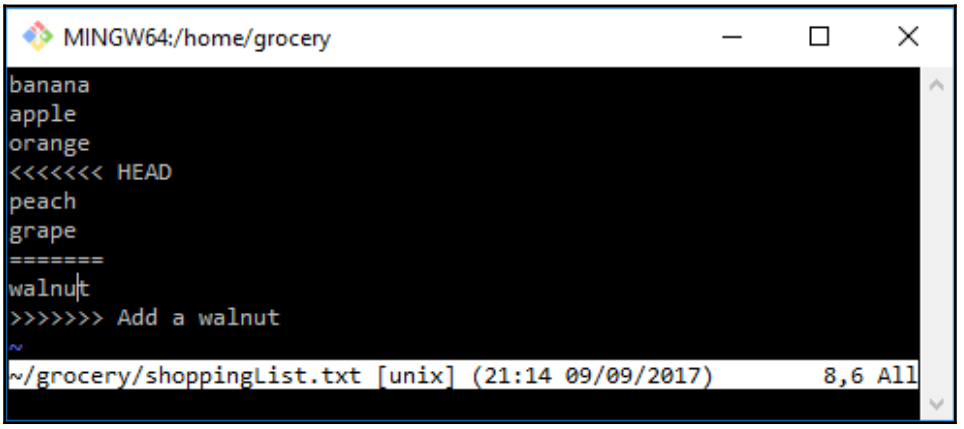
WOW! eBook
www.wowebook.org

That being said, you can now understand what `git rebase --skip` means: if you find the current patching step not useful nor necessary, you can skip it and move on to the next one.

Finally, with `git rebase --abort` you simply stop the current rebase operation, backing to the previous pre-rebase situation.

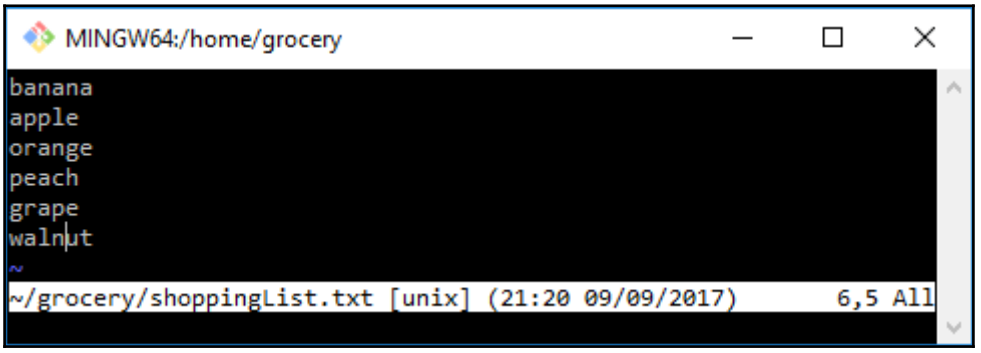
Now, back to our repository; if you open the file with Vim, you can see the generated conflict:

```
[8] ~/grocery (nuts|REBASE 1/1)
$ vi shoppingList.txt
```



Walnut has been added at line 4, but in the `master` branch, that line is occupied by the peach, and then there's a `grape`.

I will fix it adding the `walnut` at the end of the file:



WOW! eBook
www.wowebook.org

Now, the next step is to `git add` the `shoppingList.txt` file to the staging area, and then go on with the `git rebase --continue` command, as the previous message suggested:

```
[9] ~/grocery (nuts|REBASE 1/1)
$ git add shoppingList.txt

[10] ~/grocery (nuts|REBASE 1/1)
$ git rebase --continue
Applying: Add a walnut

[11] ~/grocery (nuts)
$
```

As you can see, after the `git rebase --continue` command, the rebase ends successfully (no errors and no more `REBASE` message in shell prompt at step `[11]`).

Now take a look at the repo using `git log` as usual:

```
[12] ~/grocery (nuts)
$ git log --oneline --graph --decorate --all
* 383d95d (HEAD -> nuts) Add a walnut
* 6409527 (master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Well done! The `nuts` branch is now just a commit beyond the `master` one.

Okay, now to keep the simplest and most compact repository, we cancel the `walnut` commit and put everything back in place as it was before this little experiment, even removing the `nuts` branch:

```
[13] ~/grocery (nuts)
$ git reset --hard HEAD^
HEAD is now at 6409527 Add a grape

[14] ~/grocery (nuts)
$ git checkout master
Switched to branch 'master'

[15] ~/grocery (master)
$ git branch -d nuts
Deleted branch nuts (was 6409527).
```


WOW! eBook
www.wowebook.org

Well done.

Rebasing is a wide and fairly complex topic; we would need another entire chapter (or book) to tell everything about it, but this is basically what we need to know about rewriting history.

Merging branches

Yes, I know, probably there's a thought on your mind since we start playing with branches: *why he doesn't talk about merging?*.

Now the moment has arrived.

In Git, merging two (or more!) branches is the act of making their personal history meet each other. When they meet, two things can happen:

- Files in their tip commit are different, so some conflict will rise
- Files do not conflict
- Commits of the target branch are directly behind commits of the branch we are merging, so a fast-forward will happen

In the first two cases, Git will guide us assembling a new commit, a so-called **merge commit**; in the fast-forward case instead, no new commit is needed: Git will simply move the target branch label to the tip commit of the branch we are merging.

Let's give it a try.

We can try to merge the `melons` branch into the `master` one; to do so, you have to check out the target branch, `master` in this case, and then fire a `git merge <branch name>` command; as I'm already on the `master` branch, I go straight with the `merge` command:

```
[1] ~/grocery (master)
$ git merge melons
Auto-merging shoppingList.txt
CONFLICT (content): Merge conflict in shoppingList.txt
Automatic merge failed; fix conflicts and then commit the result.
```


WOW! eBook
www.wowebook.org

Uh-oh, conflicts here. Git always tries to auto-merge a file (it uses complex algorithms to reduce your manual work on files), but if you're in doubt, pretend you fix the issues by hand.

See the conflict with `git diff`:

```
[2] ~/grocery (master|MERGING)
$ git diff
diff --cc shoppingList.txt
index 862debc,7786024..0000000
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@@ -1,5 -1,5 +1,10 @@@
  banana
  apple
  orange
++<<<<<<< HEAD
 +peach
- grape

++grape
++=======
+ blackberry
+ watermelon
++>>>>>>> melons
```

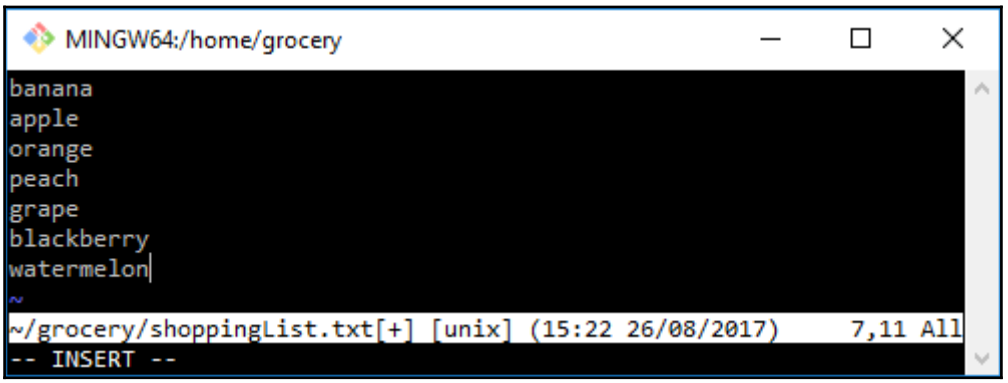
Okay, it's clear that the fourth and fifth line of our `shoppingList.txt` diverged in the two branches: in `master`, they are occupied respectively by `peach` and `grape`, in the `melons` branch the place is taken by `blackberry` and `watermelon`.

Note the shell prompt: it has that `MERGING` word after the branch name to remind us that we are in the middle of a merge. Instead, don't mind the `- grape ++grape` part: it is a due line ending mismatch between my Windows computer and the GNU/Linux Git subsystem.

To resolve the merge, you have to edit the file accordingly, and then add and commit it; let's go.

WOW! eBook
www.wowebook.org

I will edit the file enqueuing `blackberry` and `watermelon` after `peach` and `grape`, as per the following screenshot:



After saving the file, add it to the staging area and then commit:

```
[3] ~/grocery (master|MERGING)
$ git add shoppingList.txt

[4] ~/grocery (master|MERGING)
$ git commit -m "Merged melons branch into master"
[master e18a921] Merged melons branch into master
```

The commit is done, and the merge is finished. Perfect!

Now take a look at the log:

```
[5] ~/grocery (master)
$ git log --oneline --graph --decorate --all
*   e18a921 (HEAD -> master) Merged melons branch into master
|\
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Wow, that's cool!

WOW! eBook
www.wowebook.org

Look at the green path, the one on the right: this is now the new history of the `master` branch. It starts from the beginning, the banana commit, goes to `apple`, `orange`, and then to `peach`, `grape`, `blackberry`, and the `watermelon` commit.

The tip commit on the `master` branch is the merge commit, the result of the merge. Can you tell how Git is able to draw this graph?

Suggestion: look at the merge commit with `git cat-file -p`:

```
[6] ~/grocery (master)
$ git cat-file -p HEAD
tree 2916dd995ee356351c9b49a5071051575c070e5f
parent 6409527a1f06d0bbe680d461666ef8b137ac7135
parent a8c62190fb1c54d1034db78a87562733a6e3629c
author Ferdinando Santacroce <ferdinando.santacroce@gmail.com> 1503754221
+0200
committer Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
1503754221 +0200

Merged melons branch into master
```

A-ha! This commit has **two parents**! In fact, this is the result of the merge of two previous commits, and this is how Git handles merges. Storing the two parents inside the commit, Git can keep track of the merge, and use the information to draw the graph and let you remember, even after years, when and how you merged two branches.

Fast forwarding

A merge not always generates a new commit; to test this case, try to merge the `melons` branch into a `berries` one:

```
[7] ~/grocery (master)
$ git checkout berries
Switched to branch 'berries'

[8] ~/grocery (berries)
$ git merge melons
Updating ef6c382..a8c6219
Fast-forward
 shoppingList.txt | 1 +
 1 file changed, 1 insertion(+)

[9] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
```


WOW! eBook
www.wowebook.org

```
|\
| * a8c6219 (HEAD -> berries, melons) Add a watermelon
| * ef6c382 Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

As `melons` contained only a commit more than the `berries` branch, and as it changes between the two are not in conflict, doing a merge here is just a matter of a second: Git only needs to move the `berries` label to the same tip commit of the `melons` branch.

This is called **fast-forwarding**.

This time there's no merge commit, as it is not necessary; someone will argue that in this manner you lose the information that tells you when two branches have been merged. If you want to force Git always create a new merge commit, you can use the `--no-ff` (no fast-forward) option.

Wanna try? Okay, good chance to make another exercise.

Move back the `berries` branch where it was using `git reset`:

```
[10] ~/grocery (berries)
$ git reset --hard HEAD^
HEAD is now at ef6c382 Add a blackberry

[11] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (HEAD -> berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

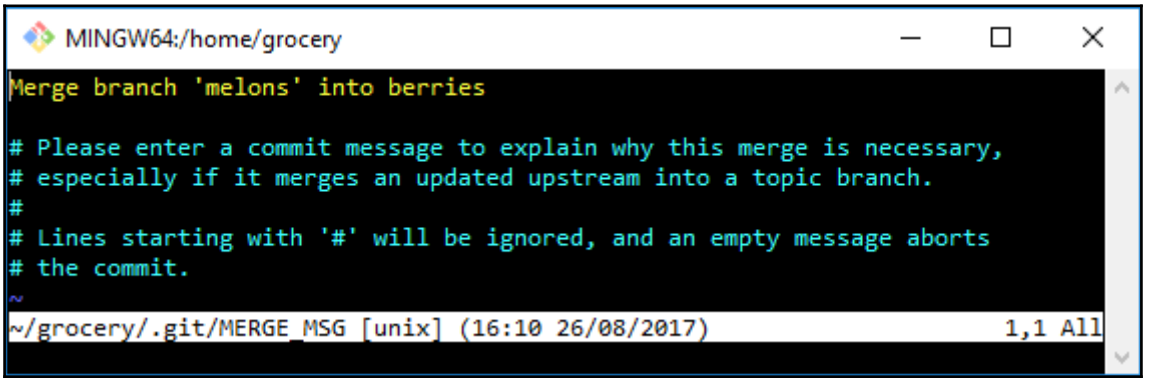

WOW! eBook
www.wowebook.org

We have just undone a merge, did you realize it?

Okay, now do the merge again with the `--no-ff` option:

```
[12] ~/grocery (berries)
$ git merge --no-ff melons
```

Git will now open your default editor to allow you to specify a commit message, as shown in the following screenshot:



As you can see, when Git can automatically merge the changes, it does; it then asks you for a commit message, suggesting a default one.

Accept the default message, save and exit:

```
[13] ~/grocery (berries)
Merge made by the 'recursive' strategy.--all
 shoppingList.txt | 1 +
 1 file changed, 1 insertion(+)
```

Merge done.

Git tells us what merging strategy is adopted for the automatic merge, and then what changed in terms of files and changes to them (insertions or deletions).

WOW! eBook
www.wowebook.org

Now a `git log`:

```
[14] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   cb912b2 (HEAD -> berries) Merge branch 'melons' into berries
|\
| | *   e18a921 (master) Merged melons branch into master
| | |\
| | |/
| |/|
| * | a8c6219 (melons) Add a watermelon
|/ /
* | ef6c382 Add a blackberry
| * 6409527 Add a grape
| * 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Okay, now the graph highlights the merge between the two branches. As you can see, the graph is a little bit more complicated now, and this is why doing a fast-forward merge is normally preferable: it ends with a more compact and simple repository structure.

We are done with these experiments; anyway, I want to undo this merge, because I want to keep the repository as simple as possible to allow you to better understand the exercise we do together; go with a `git reset --hard HEAD^`:

```
[15] ~/grocery (berries)
$ git reset --hard HEAD^
HEAD is now at ef6c382 Add a blackberry


[16] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   e18a921 (master) Merged melons branch into master
|\
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (HEAD -> berries) Add a blackberry
* | 6409527 Add a grape
* | 603b9d1 Add a peach
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```


WOW! eBook
www.wowebook.org

Okay, now undo even the past merge we did on the `master` branch:

```
[17] ~/grocery (master)
$ git reset --hard HEAD^
HEAD is now at 6409527 Add a grape

[18] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

I'm sure you get the point now: undoing a merge in Git is easy. I wanted to show you this more and more because sometimes merging branches is scaring; after doing it, sometimes you realize you messed up your project, and you go out of mind. Instead, don't worry about it: recovering from this situation is simple.

Cherry picking

Sometimes you don't want to merge two branches, but simply your desire is to apply the same changes in a commit on top to another branch. This situation is very common when working on bugs: you fix a bug in a branch, and then you want to apply the same fix on top of another branch.

Git has a convenient way to do it; this is the `git cherry-pick` command.

Let's play with it a little bit.

Assume you want to pick the `blackberry` from the `berries` branch, and then apply it into the `master` branch; this is the way:

```
[1] ~/grocery (master)
$ git cherry-pick ef6c382
error: could not apply ef6c382... Add a blackberry
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'
```


WOW! eBook
www.wowebook.org

For the argument, you usually specify the hash of the commit you want to pick; in this case, as that commit is referenced even by the `berries` branch label, doing a `git cherry-pick berries` would have been the same.

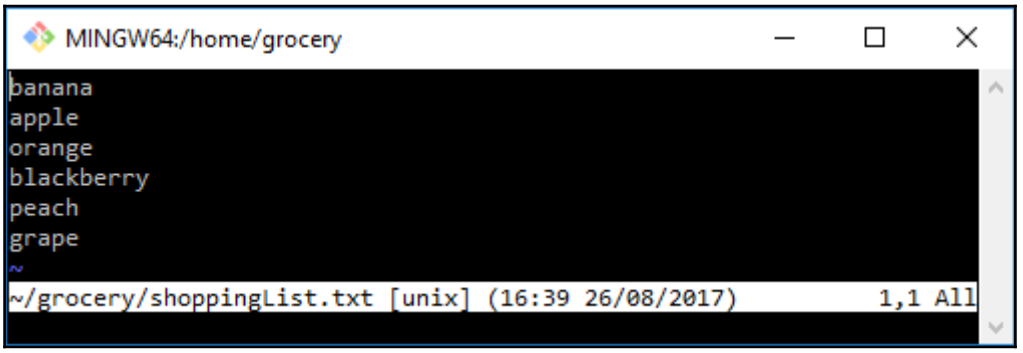
Okay, the cherry pick raised a conflict, of course:

```
[2] ~/grocery (master|CHERRY-PICKING)
$ git diff
diff --cc shoppingList.txt
index 862debc,b05b25f..0000000
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@@ -1,5 -1,4 +1,9 @@@
  banana
  apple
  orange
++<<<<<<< HEAD
 +peach
- grape
++grape
++=======
+ blackberry
++>>>>>>> ef6c382... Add a blackberry
```

The fourth line of both the `shoppingList.txt` file versions has been modified with different fruits. Resolve the conflict and then add a commit:

```
[3] ~/grocery (master|CHERRY-PICKING)
$ vi shoppingList.txt
```

The following is a screenshot of my Vim console, and the files are arranged as I like:



WOW! eBook
www.wowebook.org

```
[4] ~/grocery (master|CHERRY-PICKING)
$ git add shoppingList.txt

[5] ~/grocery (master|CHERRY-PICKING)
$ git status
On branch master
You are currently cherry-picking commit ef6c382.
  (all conflicts fixed: run "git cherry-pick --continue")
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Changes to be committed:

modified:   shoppingList.txt
```

Note the `git status` output: you always have some suggestions; in this case, to abort a `cherry-pick` and undo all you did, you can do a `git cherry-pick --abort` (you can do the same even while rebasing or merging).

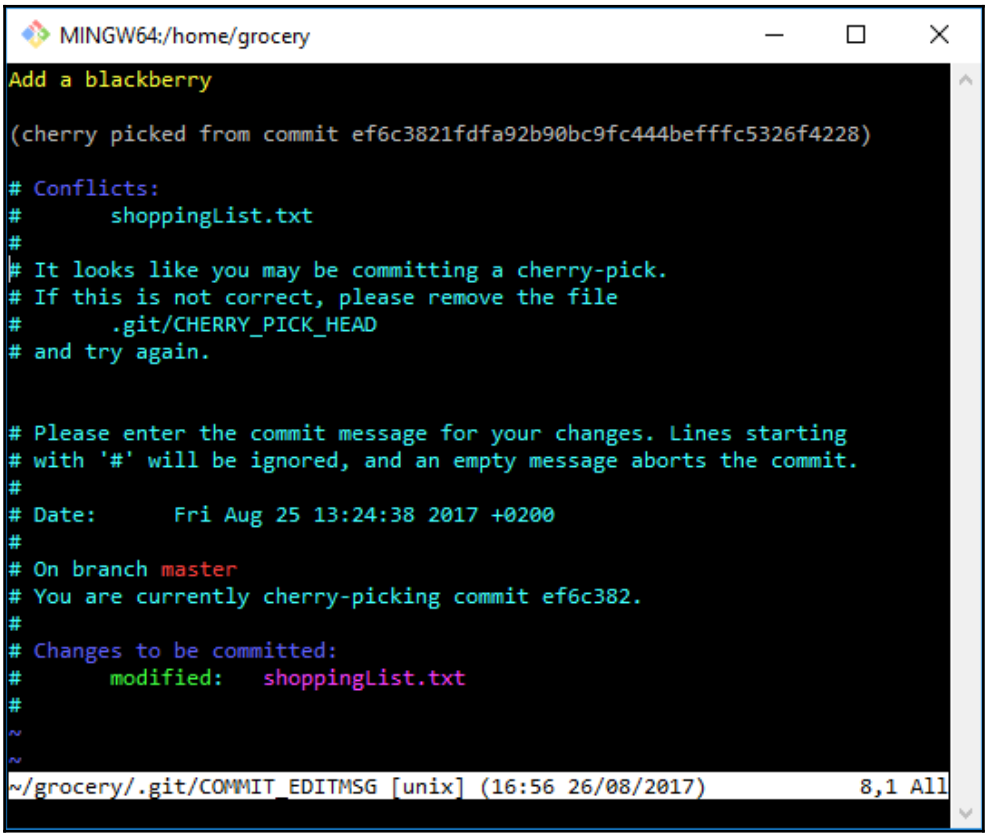
Now go on and commit:

```
[6] ~/grocery (master)
$ git commit -m "Add a cherry-picked blackberry"
On branch master
nothing to commit, working tree clean

[7] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* 99dd471 (HEAD -> master) Add a cherry-picked blackberry
* 6409527 Add a grape
* 603b9d1 Add a peach
| * a8c6219 (melons) Add a watermelon
| * ef6c382 (berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Okay, as you can see a new commit appeared, but there are no new paths in the graph. Unlike the merging feature, with cherry-picking you only pick changes made inside the specified commit, and no relationship will be stored between the cherry-picked commit and the new one created.

WOW! eBook
www.wowebook.org

If you want to track what was the commit you cherry-picked, you can append the `-x` option to the `git cherry-pick` command; then, while committing, don't append the message in the `git commit` command using the `-m` option, but type `git commit` and then press `ENTER` to allow Git to open the editor: it will suggest you a message that contains the hash of the cherry-picked commit, as you can see in the following screenshot:



This is the only way to track a cherry pick, if you want.

WOW! eBook
www.wowebook.org

Summary

This has been a very long chapter, I know.

But now I think you know all you need to work proficiently with Git, at least in your own local repository. You know about working tree, staging area, and `HEAD` commit; you know about references as branches and `HEAD`; you know how to merge rebase, and cherry pick; and finally, you know how Git works under the hood, and this will help you from here on out.

In the next chapter, we will learn how to deal with **remotes**, and pushing and pulling changes from a server such as GitHub.

WOW! eBook
www.wowebook.org

3

Git Fundamentals - Working Remotely

In the previous chapter, we learned a lot about Git; we learned how it works internally and how to manage a local repository, but now it's time to learn how to share our goodies.

In this chapter, we finally start to work in a distributed manner, using remote servers as a contact point for different developers. These are the main topics we will focus on:

- Dealing with remotes
- Cloning a remote repository
- Working with online hosting services, such as GitHub

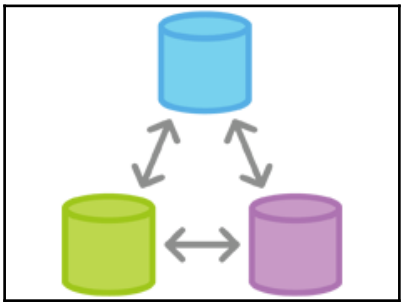
As we said before, Git is a distributed version control system: this chapter concerns the *distributed* part.

Working with remotes

Git is a tool for versioning files, as you know, but it has been built with collaboration in mind. In 2005, Linus Torvalds had the need for a light and efficient tool to share the Linux kernel code, allowing him and hundreds of other people to work on it without going crazy; the pragmatism that guided its development gave us a very robust layer for sharing data among computers, without the need of a central server.

WOW! eBook
www.wowebook.org

Basically, a Git **remote** is another "place" that has the same repository you have on your computer. As shown in the following image, you can think of it as different copies of the same repository that can exchange data between themselves:



So, a remote Git repository is just a remote copy of the same Git repository we created locally; if you have access to that remote via common protocols such as SSH, the custom `git://` protocol, or other protocols, you can keep in sync your modification with it.

Even another folder in your computer can act as a remote: for Git, the filesystem is a *communication protocol* like any other, and you are allowed to use it if you like.

This is what we will do to grasp the basic concepts about remotes.

Clone a local repository

Create a new folder on your disk to clone our `grocery` repository:

```
[1] ~
$ mkdir grocery-cloned
```

Then clone the `grocery` repository using the git clone command:

```
[2] ~ $ cd grocery-cloned [3] ~/grocery-cloned $ git clone ~/grocery .
Cloning into '.'...
```

Done.

WOW! eBook
www.wowebook.org

The dot `.` argument at the end of the command means *clone the repository in the current folder*, while the `~/grocery` argument is actually the path where Git has to look for the repository.

Now, go directly to the point with a `git log` command:

```
[4] ~/grocery-cloned (master)
$ git log --oneline --graph --decorate --all
* 6409527 (HEAD -> master, origin/master, origin/HEAD) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (origin/melons) Add a watermelon
| * ef6c382 (origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

As you can see, other than the green `master` branch label, we have some red `origin/<branch>` labels on our log output.

The origin

What is the *origin*?

Git uses origin as the default name of a remote. Like with `master` for branches, `origin` is just a convention: you can call remotes whatever you want.

The interesting thing to note here is that Git, thanks to the `--all` option in the `git log` command, shows us that there are some more branches in the remote repository, but as you can see, they do not appear in the locally cloned one. In the cloned repository, there is only `master`.

But don't worry: a local branch in which to work locally can be created by simply checking it out:

```
[5] ~/grocery-cloned (master)
$ git checkout berries
Branch berries set up to track remote branch berries from origin.
Switched to a new branch 'berries'
```

Look at the message, Git says that a *local branch has been set up to track the remote* one; this means that, from now on, Git will actively track differences between the local branch and the remote one, notifying you of differences while giving you output messages (for example, while using the `git status` command).

WOW! eBook
www.wowebook.org

Having said that, if you do a commit in this branch, you can send it to the remote and it will be part of the remote `origin/berries`.

This seems obvious, but, in Git, you can pair branches as you want; for example, you can track a remote `origin/foo` branch by a local `bar` branch, if you like. Alternatively, you can have local branches that simply don't exist on the remote. Later, we will look at how to work with remote branches.

Now, look at the log again:

```
[6] ~/grocery-cloned (berries)
$ git log --oneline --graph --decorate --all
* 6409527 (origin/master, origin/HEAD, master) Add a grape
* 603b9d1 Add a peach
| * a8c6219 (origin/melons) Add a watermelon
| * ef6c382 (HEAD -> berries, origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Now a green `berries` label appears, just near the red `origin/berries` one; this makes us aware that the local `berries` branch and remote `origin/berries` branch point to the same commit.

What happens if I do a new commit?

Let's try:

```
[7] ~/grocery-cloned (berries)
$ echo "blueberry" >> shoppingList.txt
[8] ~/grocery-cloned (berries)
$ git commit -am "Add a blueberry"
[berries ab9f231] Add a blueberry
Committer: Santacroce Ferdinando <san@intre.it>
```

Your name and email address were configured automatically based on your *username* and *hostname*. Please check that they are accurate.

WOW! eBook
www.wowebook.org

You can suppress this message by setting them explicitly:

```
git config --global user.name "Your Name"
git config --global user.email you@example.com
```

After doing this, you may fix the identity used for this commit with the following code:

```
git commit --amend --reset-author

1 file changed, 1 insertion(+)
```

As in the previous chapter, Git warns me about author and email; this time I will go with the suggested ones.

OK, let's see what happened:

```
[9] ~/grocery-cloned (berries)
$ git log --oneline --graph --decorate --all
* ab9f231 (HEAD -> berries) Add a blueberry
| * 6409527 (origin/master, origin/HEAD, master) Add a grape
| * 603b9d1 Add a peach
| | * a8c6219 (origin/melons) Add a watermelon
| |/
|/|
* | ef6c382 (origin/berries) Add a blackberry
|/
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Nice! The local `berries` branch moved forward, while `origin/berries` is still in the same place.

Sharing local commits with git push

As you may already know, Git works locally; there's no need for a remote server.

So, when you do a commit, it is available only locally; if you want to share it with a remote counterpart, you have to send it in some manner.

In Git, this is called **push**.

WOW! eBook
www.wowebook.org

Now, we will try to push the modifications in the `berries` branch to the `origin`; the command is `git push`, followed by the name of the remote and the target branch:

```
[10] ~/grocery-cloned (berries)
$ git push origin berries
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 323 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
   ef6c382..ab9f231  berries -> berries
```

Wow! There is a lot of things in this output message. Basically, Git informed us about the operations it does before and during the sending of commits to the remote.

Note that, as Git will send only the objects it knows are not present in the remote to the remote (three, in this case: a commit, a tree, and a blob), it will not send unreachable commits, nor other related unreachable objects (such as trees, blobs, or annotated that tighten only with unreachable commits).

Finally, Git tells us where it is sending objects, the destination, which in this case is just another folder on my computer: `To C:/Users/san/Google Drive/Packt/PortableGit/home/grocery`. It then tells the commit hash remote where the `origin/berries` originally was and the hash of the new tip commit, that is the same as the one in the local `berries` branch, `ef6c382..ab9f231`. Lastly, it gives the name of the two branches, the local and the remote branches, `berries -> berries`.

Now, we obviously want to see if, in the remote repository, there is a new commit in the `berries` branch; so, open the `grocery` folder in a new console and do `git log`:

```
[11] ~
$ cd grocery

[12] ~/grocery (master)
$ git log --oneline --graph --decorate --all
* ab9f231 (berries) Add a blueberry
| * 6409527 (HEAD -> master) Add a grape
| * 603b9d1 Add a peach
| | * a8c6219 (melons) Add a watermelon
| |/
|/|
* | ef6c382 Add a blackberry
|/
```


WOW! eBook
www.wowebook.org

```
* 0e8b5cf Add an orange
* e4a5e7b Add an apple
* a57d783 Add a banana to the shopping list
```

Yes, wonderful!

Just a little warning: usually, a remote repository copy is managed as a **bare repository**; in `Chapter 4`, *Git Fundamentals - Niche Concepts, Configurations, and Commands*, we will spend some words on it. As you normally won't work directly on it, a bare repository contains only the `.git` folder; it doesn't have a checked out working tree nor a HEAD reference. Instead, we use a normal repository as a remote. This is not a problem; you have just to remember one thing: you cannot push changes to the actual checked out remote branch.

In fact, in that `grocery` repository, we are actually on the `master` branch, and in the `grocery-cloned` repository, we push the `berries` branch.
The reason for this is quite simple to understand: by pushing to a remote checked out branch, you will affect the work in progress on that repository, maybe destroying ongoing changes, and this is not fair.

Getting remote commits with git pull

Now, it's time to experiment the inverse: retrieving updates from the remote repository and applying them to our local copy.

So, make a new commit in the `grocery` repository, and then, we will download it into the `grocery-cloned` one:

```
[13] ~/grocery (master)
$ printf "\r\n" >> shoppingList.txt
```

I firstly need to create a new line, because due to the previous *grape rebase*, we ended having the `shoppinList.txt` file with no new line at the end, as `echo "" >> <file>` usually does:

```
[14] ~/grocery (master)
$ echo "apricot" >> shoppingList.txt

[15] ~/grocery (master)
$ git commit -am "Add an apricot"
[master 741ed56] Add an apricot
 1 file changed, 2 insertions(+), 1 deletion(-)
```

OK, now back to the `grocery-cloned` repository.

WOW! eBook
www.wowebook.org

We can retrieve objects from a remote with `git pull`.

In truth, `git pull` is a *super command*; in fact, it is basically the sum of two other Git commands, `git fetch` and `git merge`. While obtaining objects from a remote, Git won't force you to merge them into a local branch; this can seem a little bit confusing at first, as in other versioning systems, such as Subversion, this is the default behavior.

Instead, Git is more conservative: it could happen that someone pushed a commit or more on top of a branch, but you realized those commits are not good for you, or simply they are just wrong. So, using `git fetch`, you can get and inspect them before applying them on your local branch with a `git merge`.

Let's try `git pull` for now, then we will try to use `git fetch` and `git merge` separately.

Go back to the `grocery-cloned` repository, switch to the `master` branch, and do a `git pull`:

```
[16] ~/grocery-cloned (berries)
$ git checkout master
Your branch is up-to-date with 'origin/master'.
Switched to branch 'master'
```

Git says that our branch is up-to-date with `'origin/master'`, but this is not true, as we just did a new commit there. This is because, for Git, the only way to know if we are updated in respect a remote repository is to perform a `git fetch`, and we didn't. Later we will see this more clearly.

For now, go with `git pull`: the command wants you to specify the name of the remote you want to pull from, which is `origin` in this case, and then the branch you want to merge into your local one, which is `master`, of course:

```
[17] ~/grocery-cloned (master)
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
 * branch    master     -> FETCH_HEAD
   6409527..741ed56  master     -> origin/master
Updating 6409527..741ed56
Fast-forward
 shoppingList.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```


WOW! eBook
www.wowebook.org

Good! Git tells us that there are three new objects to fetch; after it has obtained them, it performs a merge on top of the local `master` branch, and in this case, it performs a fast-forward merge.

OK, now I want you to try doing these steps in a separate manner; create the umpteenth new commit in the `grocery` repository, the `master` branch:

```
[18] ~/grocery (master)
$ echo "plum" >> shoppingList.txt

[19] ~/grocery (master)
$ git commit -am "Add a plum"
[master 50851d2] Add a plum
1 file changed, 1 insertion(+)
```

Now perform a `git fetch` on `grocery-cloned` repository:

```
[20] ~/grocery-cloned (master)
$ git fetch
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From C:/Users/san/Google Drive/Packt/PortableGit/home/grocery
   741ed56..50851d2  master     -> origin/master
```

As you can see, Git found new objects on the remote, and it downloaded them.

Note that you can do a `git fetch` in whatever branch you are in, as it simply downloads remote objects; it won't merge them. Instead, while doing a `git pull`, you have to be sure to be in the right local target branch.

Do a `git status` now:

```
[21] ~/grocery-cloned (master)
$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-
forwarded.
  (use "git pull" to update your local branch)
nothing to commit, working tree clean
```

OK, as you can see, when there is a remote, `git status` informs you even on the state of the *synchronization* between your local repository and the remote one; here it tells us we are behind the remote because it has because it has one commit more than us in the `master` branch, and that commit can be fast-forwarded.

WOW! eBook
www.wowebook.org

Now, let's sync with a `git merge`; to merge a remote branch, we have to specify, other than the branch name, even the remote one, as we did in the `git pull` command previously:

```
[22] ~/grocery-cloned (master)
$ git merge origin master
Updating 741ed56..50851d2
Fast-forward
 shoppingList.txt | 1 +
 1 file changed, 1 insertion(+)
```

That's all!

This is basically what you need to know to work with remotes. Note that, if some changes on the remote are in conflict with the local ones, you will have to solve them as we did in the previous merge examples.

How Git keeps track of remotes

Git stores remote branch labels in a similar way to how it stores the local branches ones; it uses a subfolder in `refs` for the scope, with the symbolic name we used for the remote, in this case `origin`, the default one:

```
[23] ~/grocery-cloned (master)
$ ll .git/refs/remotes/origin/
total 3
drwxr-xr-x 1 san 1049089  0 Aug 27 11:25 ./
drwxr-xr-x 1 san 1049089  0 Aug 26 18:19 ../
-rw-r--r-- 1 san 1049089 41 Aug 26 18:56 berries
-rw-r--r-- 1 san 1049089 32 Aug 26 18:19 HEAD
-rw-r--r-- 1 san 1049089 41 Aug 27 11:25 master
```

The command to deal with remotes is `git remote`; you can add, remove, rename, list, and do a lot of other things with them; there's no room here to see all the options. Please refer to the Git guide if you need to know more about the `git remote` command.

Now, we will play a little bit with a remote on a public server; we will use free GitHub hosting for this purpose.

WOW! eBook
www.wowebook.org

Working with a public server on GitHub

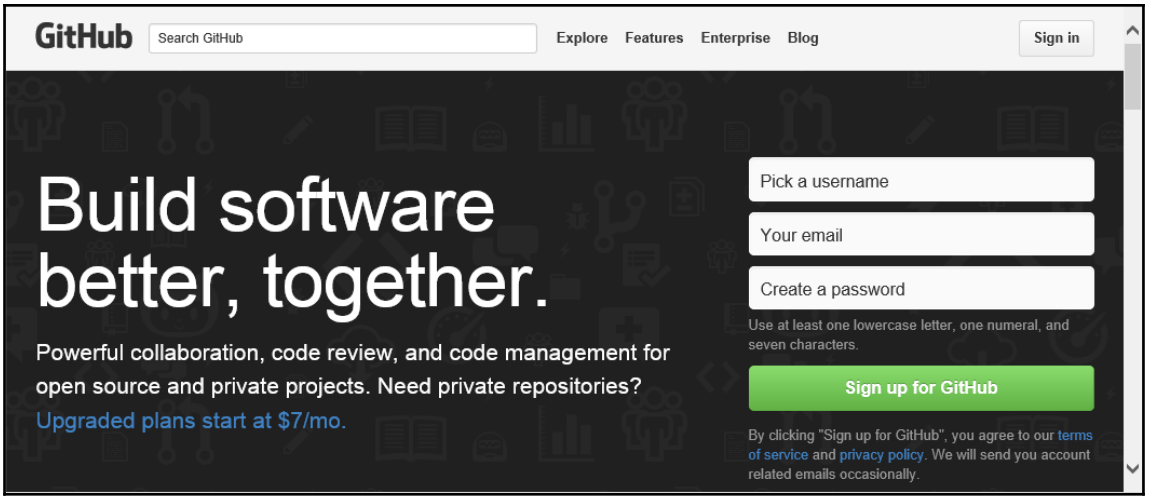
To start working with a public hosted remote, we have to get one. Today, it is not difficult to achieve; the world has plenty of free online services offering room for Git repositories. One of the most commonly used is **GitHub**.

Setting up a new GitHub account

GitHub offers unlimited free public repositories, so we can make use of it without investing a penny. In GitHub, you have to pay only if you need private repositories; for example, to store closed source code you base your business on.

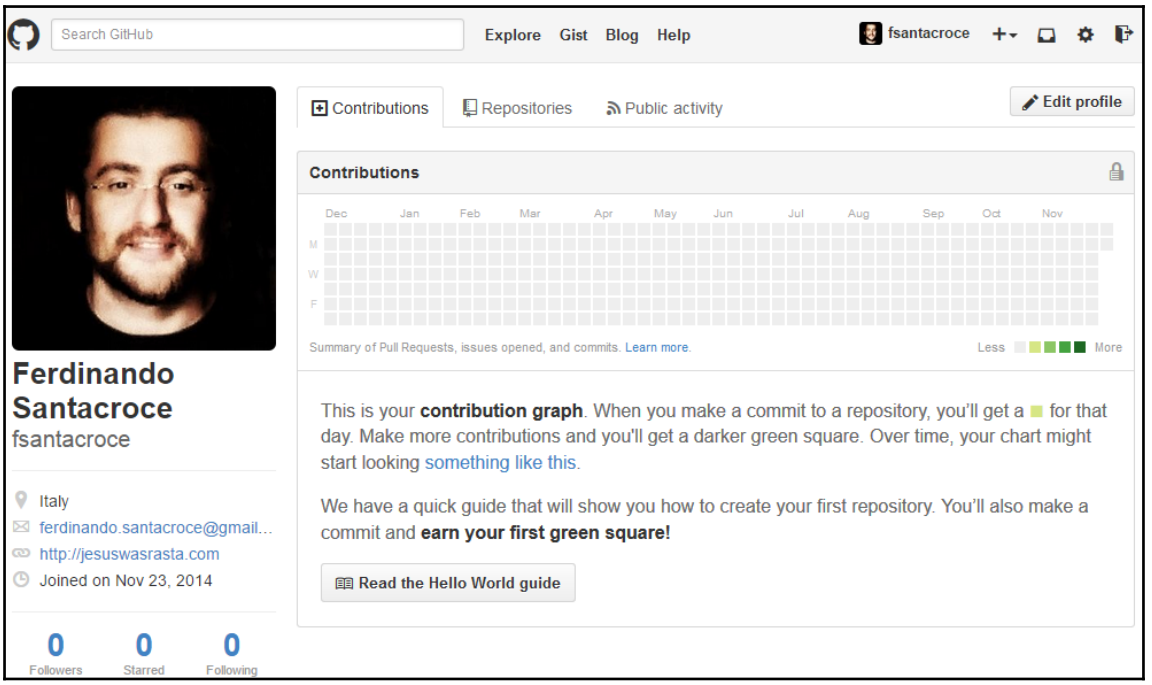
Creating a new account is simple:

1. Go to `https://github.com`.
2. Sign up, filling the textboxes, as per the following image, and provide a username, a password, and your email:

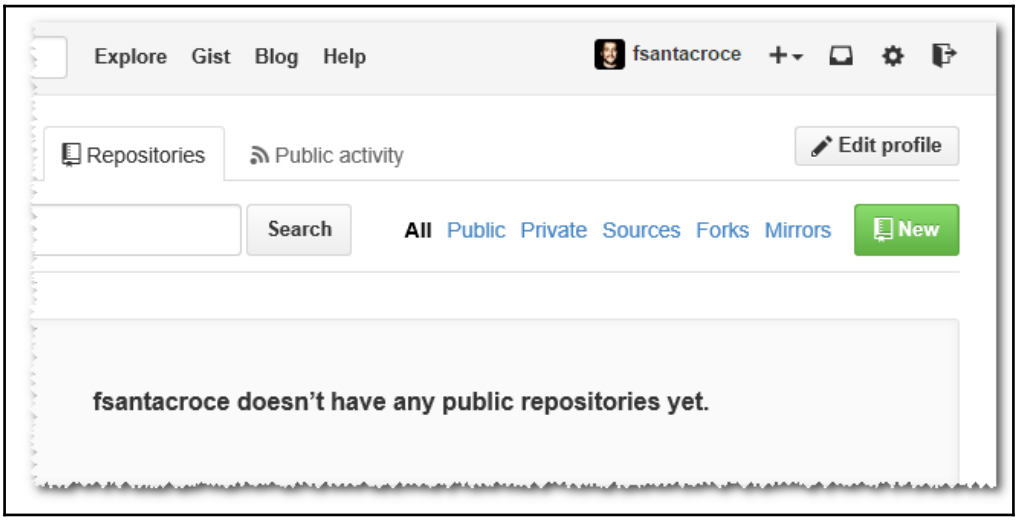


WOW! eBook
www.wowebook.org

When done, we are ready to create a brand new repository in which to push our work:

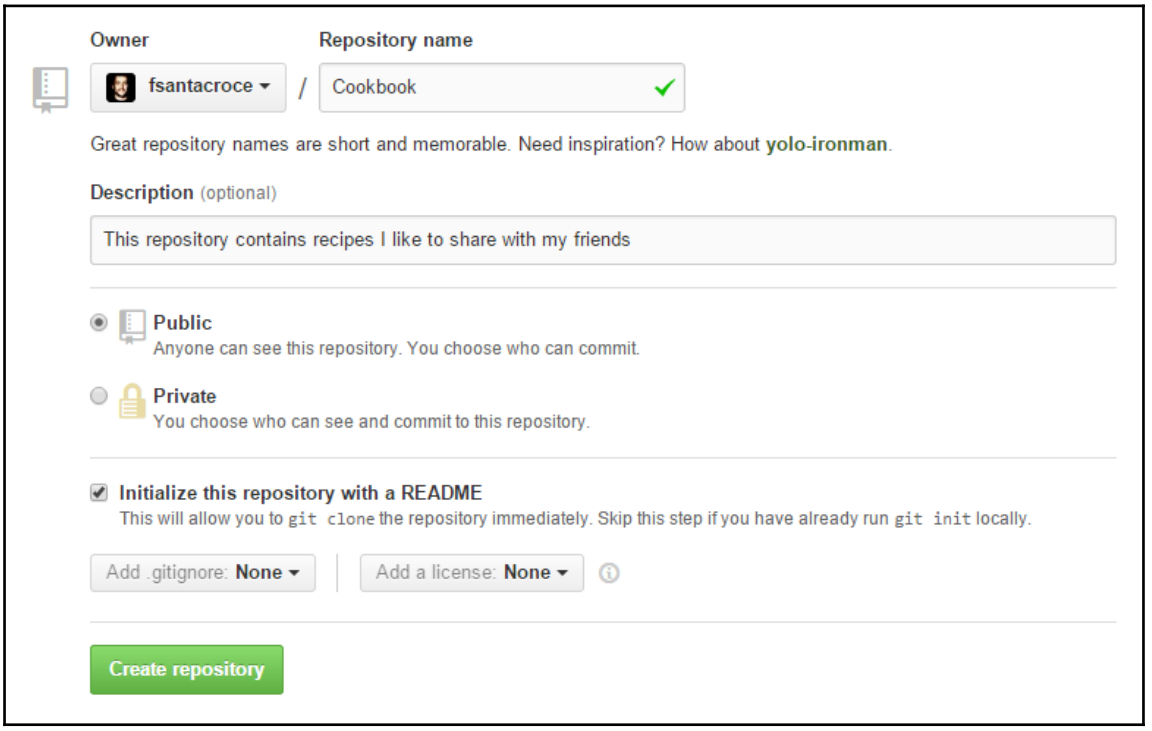


Go to the **Repositories** tab, click the green **New** button, and choose a name for your repository, as you can see in the following screenshot:



WOW! eBook
www.wowebook.org

For the purpose of learning, I will create a simple repository for my personal recipes, written using the **markdown markup language** (`http://daringfireball.net/projects/markdown/`):



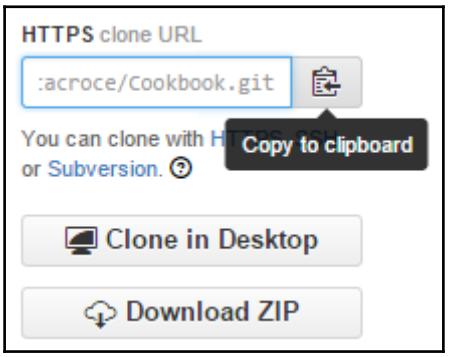
Then, you can write a description for your repository; this is useful for allowing people that come to visit your profile to better understand what your project is intended for. We create our repository as public because private repositories have a cost, as we said before, and then we initialize it with a `README` file; choosing this, GitHub makes a first commit for us, initializing the repository that now is ready to be cloned.

Cloning the repository

Now that we have a remote repository, it's time to *hook* it locally. For this, Git provides the `git clone` command, as we have already seen.

WOW! eBook
www.wowebook.org

Using this command is quite simple; in this case, all we need to know is the URL of the repository to clone. The URL is provided by GitHub on the right down part of the repository home page:



To copy the URL, you can simply click the clipboard button at the right of the textbox.

So, let's try to follow these steps together:

1. Go to a local root folder for the repositories.
2. Open a Bash shell within it.
3. Type `git clone https://github.com/fsantacroce/Cookbook.git`.

Obviously, the URL of your repository will be different; as you can see, GitHub URLs are composed as follows: `https://github.com/<Username>/<RepositoryName>.git`:

```
[1] ~
$ git clone https://github.com/fsantacroce/Cookbook.git
Cloning into 'Cookbook'...
remote: Counting objects: 15, done.
remote: Total 15 (delta 0), reused 0 (delta 0), pack-reused 15
Unpacking objects: 100% (15/15), done.

[2] ~
$ cd Cookbook/

[3] ~/Cookbook (master)
$ ll
total 13
drwxr-xr-x 1 san 1049089   0 Aug 27 14:16 ./
drwxr-xr-x 1 san 1049089   0 Aug 27 14:16 ../
drwxr-xr-x 1 san 1049089   0 Aug 27 14:16 .git/
-rw-r--r-- 1 san 1049089 150 Aug 27 14:16 README.md
```


WOW! eBook
www.wowebook.org

At this point, Git creates a new `Cookbook` folder containing the downloaded copy of our repository; inside, we will find a `README.md` file, a classical one for a GitHub repository. In that file, you can describe your repository using the common markdown markup language to users who will chance upon it.

Uploading modifications to remotes

So, let's try to edit the `README.md` file and upload modifications to GitHub:

1. Edit the `README.md` file using your preferred editor, adding, for example, a new sentence.
2. Add it to the index and then commit.
3. Put your commit on the remote repository using the `git push` command.

But firstly, set the user and email this time, so Git will not output the message we have seen in the previous chapters:

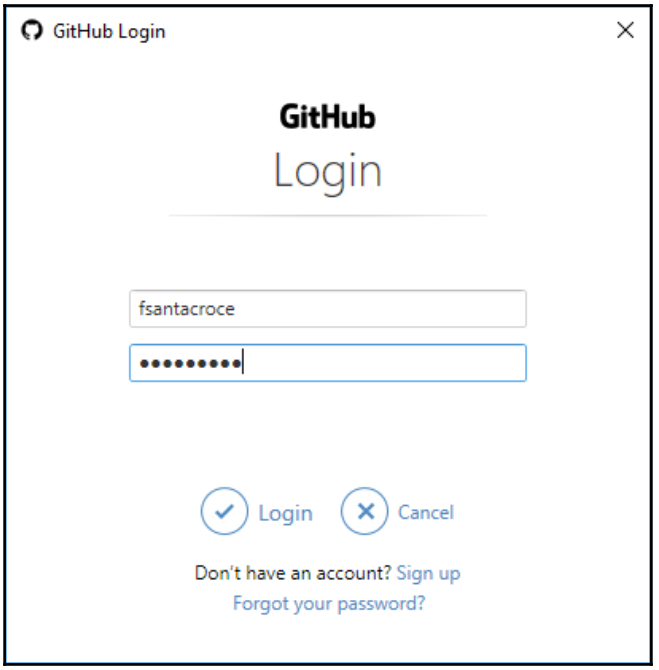
```
[4] ~/Cookbook (master)
$ git config user.name "Ferdinando Santacroce"

[5] ~/Cookbook (master)
$ git config user.email "ferdinando.santacroce@gmail.com"

[6] ~/Cookbook (master)
$ vim README.md

Add a sentence then save and close the editor.

[7] ~/Cookbook (master)
$ git add README.md

[8] ~/Cookbook (master)
$ git commit -m "Add a sentence to readme"
[master 41bdbe6] Add a sentence to readme
 1 file changed, 2 insertions(+)
```

Now, try to type `git push` and press `ENTER`, without specifying anything else:

```
[9] ~/Cookbook (master)
$ git push
```


WOW! eBook
www.wowebook.org

Here, in my Windows 10 workstation, this window appears:



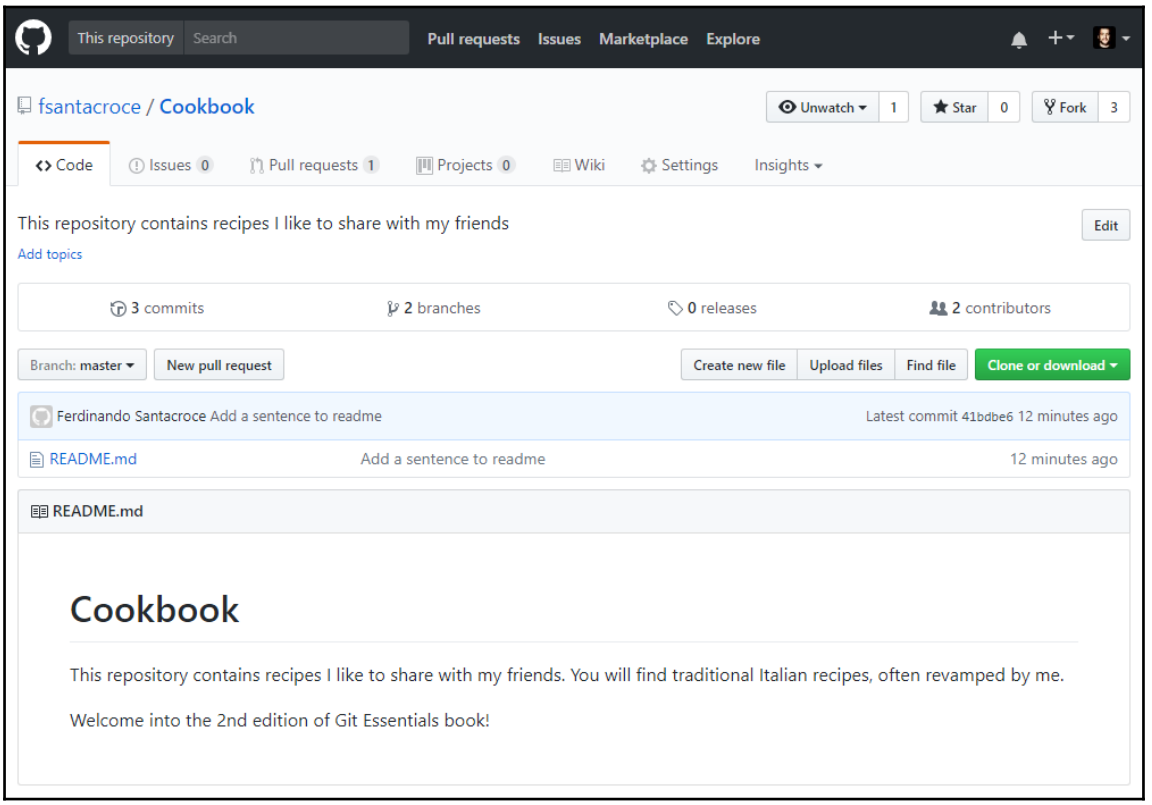
This is the **Git Credential Manager**; it allows you to set credentials on your Windows machine. If you are on Linux or macOS, the situation may be different, but the underlying concept is the same: we have to give Git the credentials in order to access the remote GitHub repository; they will then be stored to our system.

Input your credentials, and then press the **Login** button; after that, Git continues with:

```
[10] ~/Cookbook (master)
$ git push
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/fsantacroce/Cookbook.git
   e1e7236..41bdbe6  master -> master
```


WOW! eBook
www.wowebook.org

The `git push` command allows you to *upload* local work to a configured remote location; in this case, a remote GitHub repository, as you can see in the following screenshot:



There are a few more things we need know about pushing; we can begin to understand the message Git gave us just after we ran the `git push` command.

What do I send to the remote when I push?

When you give a `git push` without specifying anything else, Git sends all the **new commits** and all the related objects you did locally in your current branch to the remote; for *new commits*, it mean that we will send only local commits that have not been uploaded yet.

WOW! eBook
www.wowebook.org

Pushing a new branch to the remote

Obviously, we can create and push a new branch to the remote to make our work public and visible to other collaborators; for instance, I will create a new branch for a new recipe, then I will push to the remote GitHub server. Follow these simple steps:

1. Create a new branch, for instance `Risotti`.
2. Add to it a new file, for example, `Risotto-alla-Milanese.md`, and commit it.
3. Push the branch to the remote using `git push -u origin Risotti`:

```
[11] ~/Cookbook (master)
$ git branch Risotti

[12] ~/Cookbook (master)
$ git checkout Risotti

[13] ~/Cookbook (Risotti)
$ notepad Risotto-alla-Milanese.md

[14] ~/Cookbook (Risotti)
$ git add Risotto-alla-Milanese.md

[15] ~/Cookbook (Risotti)
$ git commit -m "Add risotto alla milanese recipe ingredients"
[Risotti b62bc1f] Add risotto alla milanese recipe ingredients
 1 file changed, 15 insertions(+)
 create mode 100644 Risotto-alla-Milanese.md

[16] ~/Cookbook (Risotti)
$ git push -u origin Risotti
Total 0 (delta 0), reused 0 (delta 0)
Branch Risotti set up to track remote branch Risotti from origin.
To https://github.com/fsantacroce/Cookbook.git
 * [new branch]      Risotti -> Risotti
```

Before continuing, we have to examine in depth some things that happened after this `git push` command.

The origin

With the `git push -u origin Risotti` command, we told Git to upload our `Risotti` branch (and the commits within it) to the `origin`; with the `-u` option, we set the local branches to track the remote one.

WOW! eBook
www.wowebook.org

We know that `origin` is the default remote of a repository, just as `master` is the default branch name; when you clone a repository from a remote, that remote becomes your `origin` alias. When you tell Git to push or pull something, you often have to tell it the remote you want to use; using the alias `origin`, you say to Git you want to use your default remote.

If you want to see remotes actually configured in your repository, you can type a simple `git remote` command, followed by `-v` (`--verbose`) to get some more details:

```
[17] ~/Cookbook (master)
$ git remote -v
origin  https://github.com/fsantacroce/Cookbook.git (fetch)
origin  https://github.com/fsantacroce/Cookbook.git (push)
```

In the details, you will see the full URL of the remote and discover that Git stores two different URLs:

- The Fetch URL, which is where we take updates from others
- The Push URL, which is where we send out updates to others

This allows us to push and pull changes from different remotes, if you like, and underlines how Git can be considered a peer-to-peer versioning system.

You can add, update, and delete remotes using the `git remote` command.

Tracking branches

Using the `-u` option, we told Git to **track** the remote branch. Tracking a remote branch is *the way to tie your local branch with the remote one*; please note that this behavior is not automatic, you have to set it if you want it. When a local branch tracks a remote branch, you actually have a local and a remote branch that can be kept easily in sync (please note that a local branch can track only one remote branch). This is very useful when you need to collaborate with some remote coworkers at the same branch, allowing all of them to keep their work in sync with other people's changes.

WOW! eBook
www.wowebook.org

To better understand the way our repository is now configured, try to type `git remote show origin`:

```
[18] ~/Cookbook (master)
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/fsantacroce/Cookbook.git
  Push  URL: https://github.com/fsantacroce/Cookbook.git
  HEAD branch: master
  Remote branches:
    Pasta   tracked
    Risotti tracked
    master  tracked
  Local branches configured for 'git pull':
    Risotti merges with remote Risotti
    master  merges with remote master
  Local refs configured for 'git push':
    Risotti pushes to Risotti (up to date)
    master  pushes to master  (fast-forwardable)
```

As you can see, the `Pasta`,`Risotti`, and `master` branches are all tracked.

You see also that your local branches are configured to push and pull to remote branches with the same name, but remember: it is not mandatory to have local and remote branches with the same name; the local branch, `foo`, can track the remote branch, `bar`, and vice versa; there's no restrictions.

Going backward – publishing a local repository to GitHub

Commonly, you will find yourself needing to put your local repository on a shared place where someone else can look at your work; in this section, we will learn how to achieve this purpose.

Create a new local repository to publish, following these simple steps:

1. Go to our local repositories folder.
2. Create a new `HelloWorld` folder.
3. In it place a new repository, as we did in first chapter.

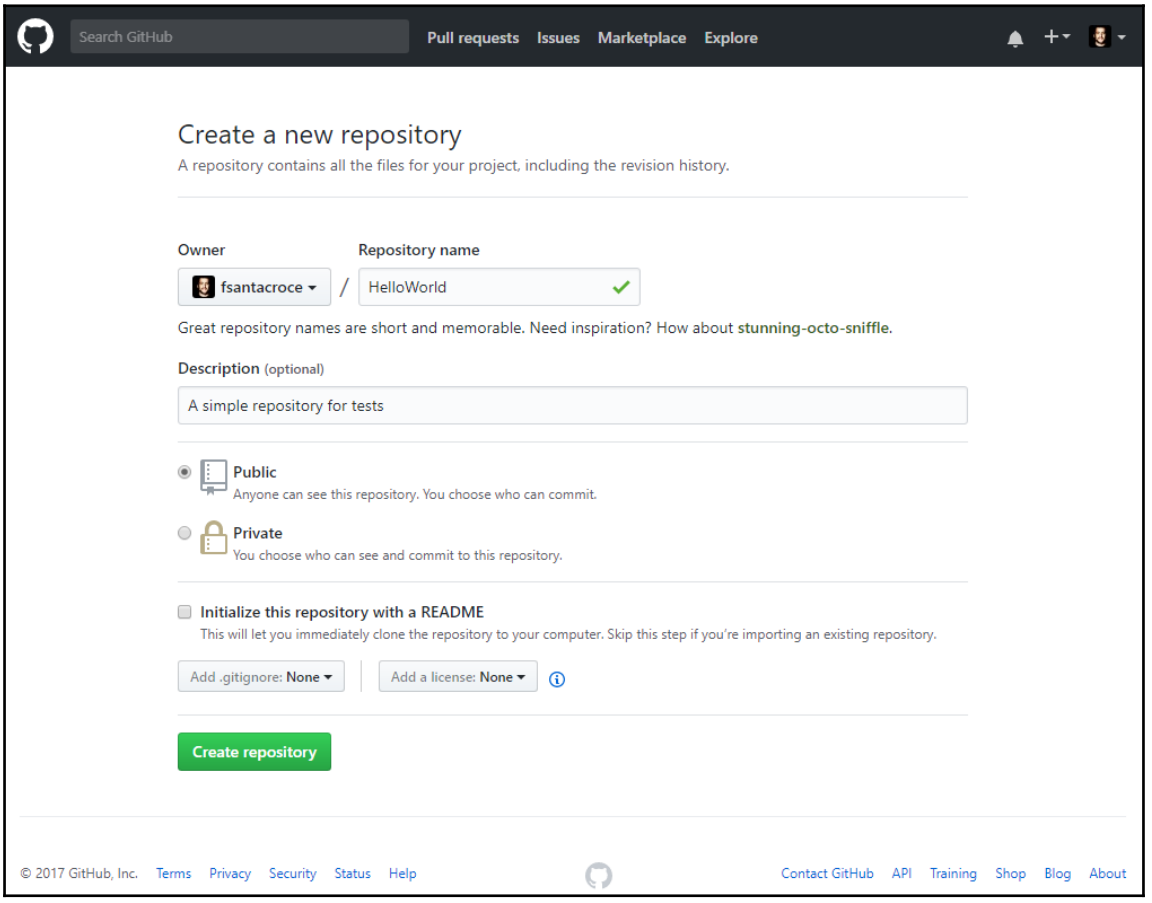
WOW! eBook
www.wowebook.org

4. Add a new `README.md` file and commit it:

```
[19] ~
$ mkdir HelloWorld

[20] ~
$ cd HelloWorld/

[21] ~/HelloWorld
$ git init
Initialized empty Git repository in C:/Users/san/Google
Drive/Packt/PortableGit/home/HelloWorld/.git/

[22] ~/HelloWorld (master)
$ echo "Hello World!" >> README.md

[23] ~/HelloWorld (master)
$ git add README.md

[24] ~/HelloWorld (master)
$ git config user.name "Ferdinando Santacroce"

[25] ~/HelloWorld (master)
$ git config user.email "ferdinando.santacroce@gmail.com"

[26] ~/HelloWorld (master)
$ git commit -m "First commit"
[master (root-commit) 5b41441] First commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md

[27] ~/HelloWorld (master)
```


WOW! eBook
www.wowebook.org

Now, create the GitHub repository as we did previously; this time leave it empty. Don't initialize it with a readme file; we already have one in our local repository. The following is a screenshot taken directly from the GitHub repository creation page:



At this point, we are ready to publish our local repository on GitHub or, even better, to add a remote to it.

Adding a remote to a local repository

To publish our `HelloWorld` repository, we simply have to add its first remote; adding a remote is quite simple: `git remote add origin <remote-repository-url>`

WOW! eBook
www.wowebook.org

So, this is the full command we have to type in the Bash shell:

```
[27] ~/HelloWorld (master)
$ git remote add origin https://github.com/fsantacroce/HelloWorld.git
```

Adding a remote, like adding or modifying other configuration parameters, simply consists of editing some text files in the `.git` folder, so it is blazing fast.

Pushing a local branch to a remote repository

After that, push your local changes to the remote using `git push -u origin master`:

```
[28] ~/HelloWorld (master)
$ git push -u origin master
Counting objects: 3, done.
Writing objects: 100% (3/3), 231 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
Branch master set up to track remote branch master from origin.
To https://github.com/fsantacroce/HelloWorld.git
 * [new branch]      master -> master
```

That's all!

Social coding - collaborating using GitHub

GitHub's trademark is the concept of so-called **social coding**; from its first steps, GitHub made it easy to share code, track other people's work, and collaborate using two basic concepts: **forks** and **pull requests**. In this section, I will illustrate them in brief.

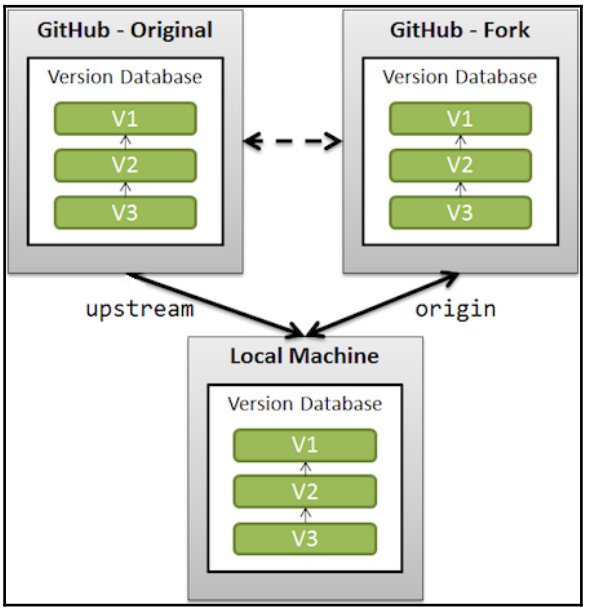
Forking a repository

Forking is a common concept for a developer; if you have already used a GNU-Linux based distribution, you know that there are some forefathers, such as Debian, and some derived distro, such as Ubuntu, also commonly called *forks* of the original distribution.

In GitHub things are similar. At some point, you find an interesting open-source project you want to modify slight to perfectly fit your needs; at the same time, you want to benefit from bug fixes and new features from the original project, keeping your work in touch. The right thing to do in this situation is to *fork* the project. But first, remember: **fork is not a Git term**, but GitHub terminology.

WOW! eBook
www.wowebook.org

When you fork on GitHub, what you get *is essentially a server-side clone of the repository* on your GitHub account; if you clone your forked repository locally, in the remote list, you will find an `origin` that points to your account repository, while the original repository will assume the *upstream* alias (you have to add it manually anyway):



To better understand this feature, go to your GitHub account and try to fork a common GitHub repository called `Spoon-Knife`, made by GitHub mascot user `octocat`; so:

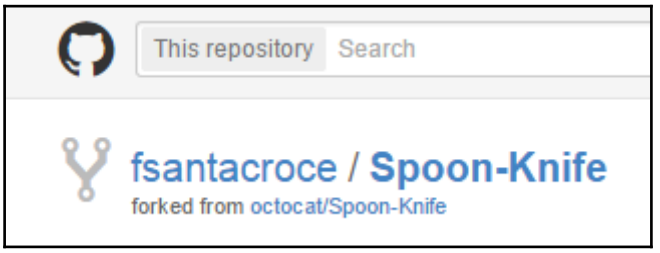
1. Log in to your GitHub account
2. Look for `spoon-knife` using the search textbox located up on the left of the page:

3. Click on the first result, **octocat/Spoon-Knife** repository

WOW! eBook
www.wowebook.org

4. Fork the repository using the **Fork** button at the right of the page:

5. After a funny photocopy animation, you will get a brand-new Spoon-Knife repository in your GitHub account:



Now, you can clone that repository locally, as we did before:

```
[1] ~
$ git clone https://github.com/fsantacroce/Spoon-Knife.git
Cloning into 'Spoon-Knife'...
remote: Counting objects: 19, done.
remote: Total 19 (delta 0), reused 0 (delta 0), pack-reused 19
Unpacking objects: 100% (19/19), done.

[2] ~
$ cd Spoon-Knife

[3] ~/Spoon-Knife (master)
$ git remote -v
origin  https://github.com/fsantacroce/Spoon-Knife.git (fetch)
origin  https://github.com/fsantacroce/Spoon-Knife.git (push)
```

As you can see, the `upstream` remote is not present, you have to add it manually; to add it, use the `git remote add` command:

```
[4] ~/Spoon-Knife (master)
$ git remote add upstream https://github.com/octocat/Spoon-Knife.git

[5] ~/Spoon-Knife (master)
```


WOW! eBook
www.wowebook.org

```
$ git remote -v
origin https://github.com/fsantacroce/Spoon-Knife.git (fetch)
origin https://github.com/fsantacroce/Spoon-Knife.git (push)
upstream https://github.com/octocat/Spoon-Knife.git (fetch)
upstream https://github.com/octocat/Spoon-Knife.git (push)
```

Now, you can keep your local repository in sync with the changes in your remote, the `origin`, and you can also get those ones coming from the `upstream` remote, the original repository you forked. At this point. you are probably wondering how to deal with two different remotes; well, it is easy: simply pull from the `upstream` remote and merge those modifications in your local repository, then push them into your `origin` remote in a bundle with your changes. If someone else clones your repository, he or she will receive your work merged with the work done by someone else on the original repository.

Submitting pull requests

If you created a fork of a repository, it is because you are not a direct contributor of the original project, or simply you don't want to make a mess in other people's work before becoming familiar with the code.

However, at a certain point, you realize your work can be useful even for the original project: you realize a better implementation of a previous piece of code, you add a missing feature, and so on.

So, you find yourself needing to notify the original author that you did something interesting, asking him if he wants to take a look and, maybe, integrate your work. This is the moment when **pull requests** come in handy.

A pull request is a way to tell the original author *Hey! I did something interesting using your original code, do you want to take a look and integrate my work, if you find it good enough?* This is not only a technical way to achieve the purpose of integrating work, but it is even a powerful practice for promoting **code reviews** (and then the so-called *social coding*) as recommended by *eXtreme Programming* fellows (for more information, visit: `http://en.wikipedia.org/wiki/Extreme_programming`).

Another reason to use a pull request is that **you cannot push directly** to the `upstream` remote **if you are not a contributor** of the original project: pull requests are the only way. In small scenarios (such as a team of two or three developers that works in the same room) probably the *fork and pull* model represents an overhead, so it is more common to directly share the original repository with all the contributors, skipping the fork and pull ceremony.

WOW! eBook
www.wowebook.org

Creating a pull request

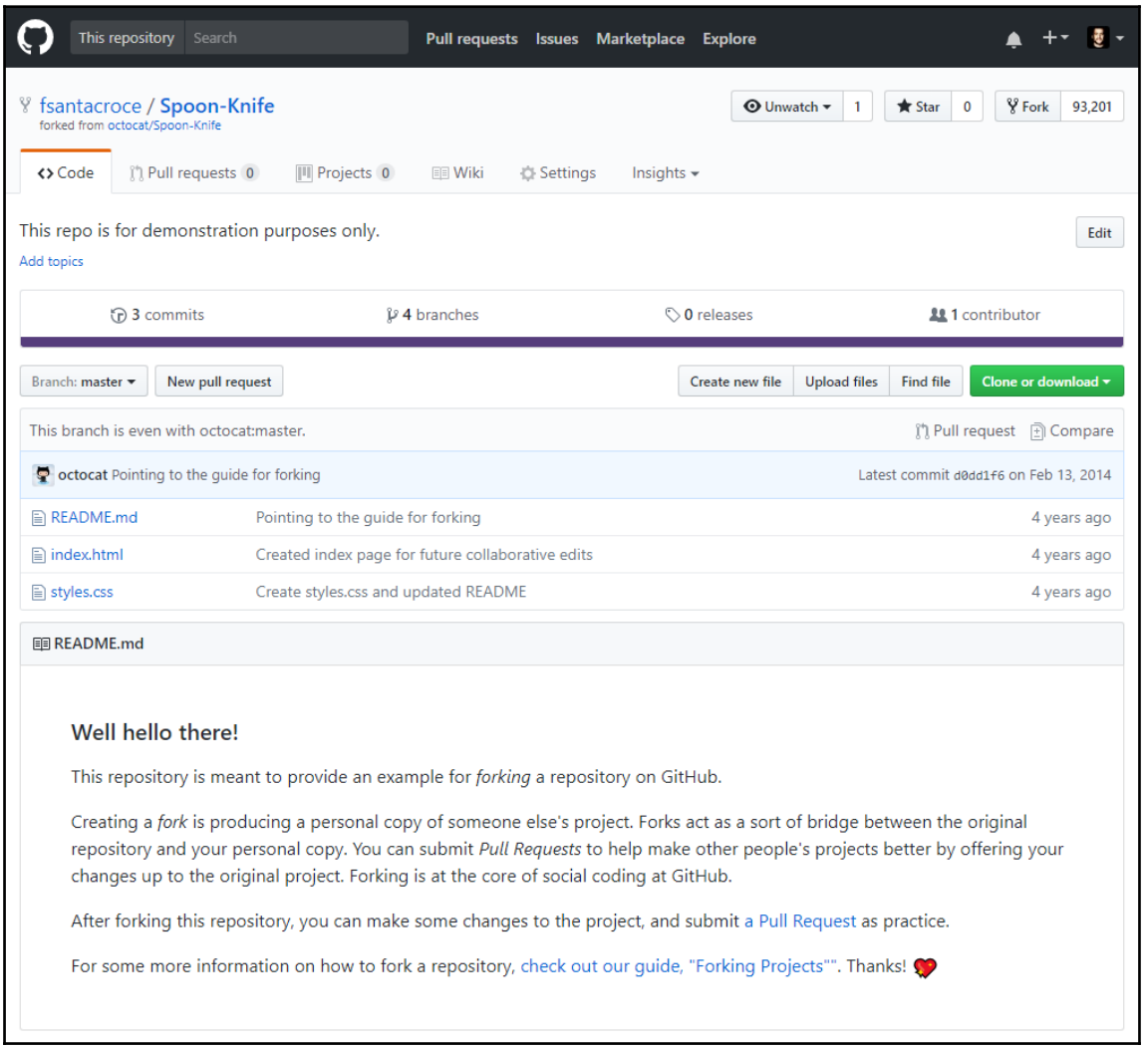
To create a pull request, you have to go on your GitHub account and make it directly from your forked account; but first, you have to know that **pull requests can be made only from separated branches**. At this point of the book, you are probably used to creating a new branch for a new feature or refactor purpose, so this is nothing new, isn't it?

To make an attempt, let's create a local `TeaSpoon` branch in our repository, commit a new file, and push it to our GitHub account:

```
[6] ~/Spoon-Knife (master)
$ git branch TeaSpoon

[7] ~/Spoon-Knife (master)
$ git checkout TeaSpoon
Switched to branch 'TeaSpoon'

[8] ~/Spoon-Knife (TeaSpoon)
$ vi TeaSpoon.md

[9] ~/Spoon-Knife (TeaSpoon)
$ git add TeaSpoon.md

[10] ~/Spoon-Knife (TeaSpoon)
$ git commit -m "Add a TeaSpoon to the cutlery"
[TeaSpoon 62a99c9] Add a TeaSpoon to the cutlery
1 file changed, 2 insertions(+)
 create mode 100644 TeaSpoon.md

[11] ~/Spoon-Knife (TeaSpoon)
$ git push origin TeaSpoon
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 417 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/fsantacroce/Spoon-Knife.git
   d0dd1f6..62a99c9  TeaSpoon -> TeaSpoon
```

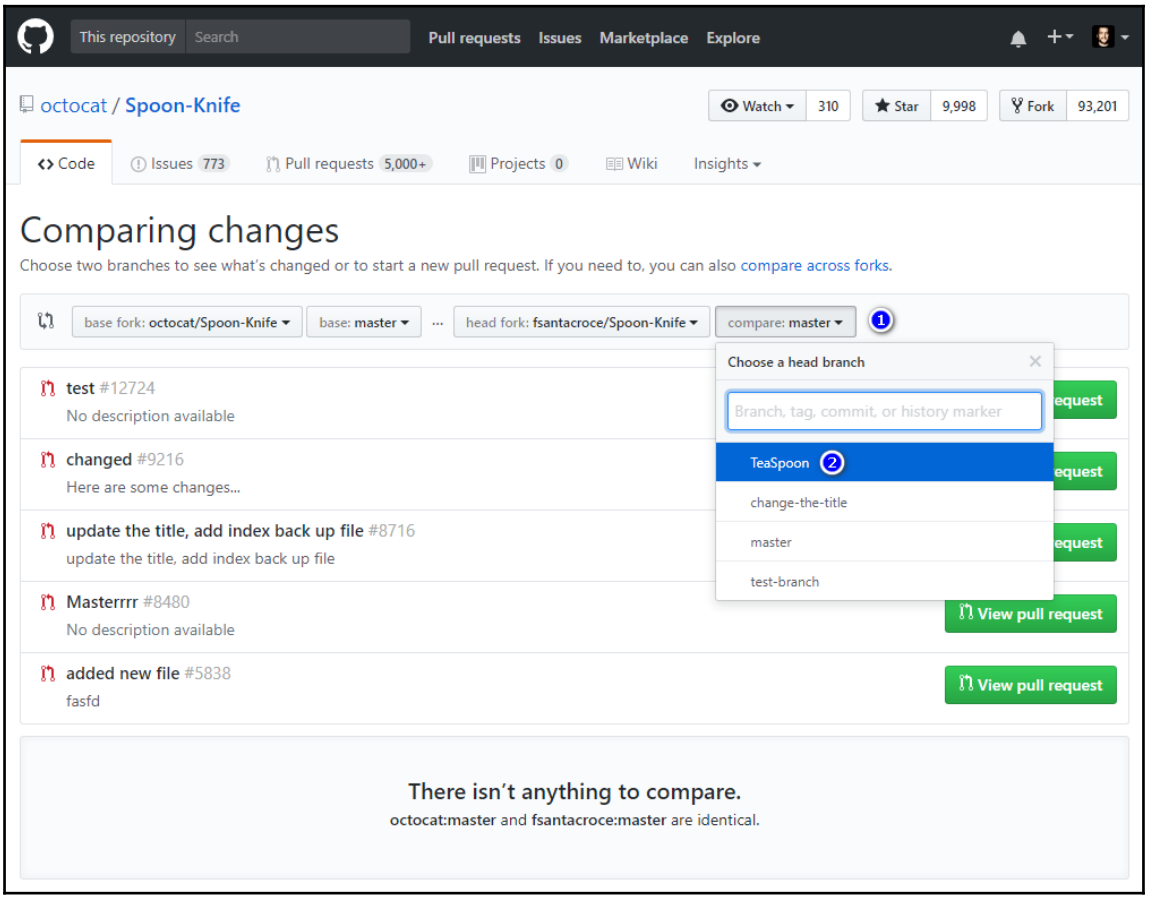

WOW! eBook
www.wowebook.org

If you take a look at your account, you will find a surprise: in your `Spoon-Knife` repository, there is now a **New pull request** button made for the purpose of starting a pull request, as you can see in the following screenshot:



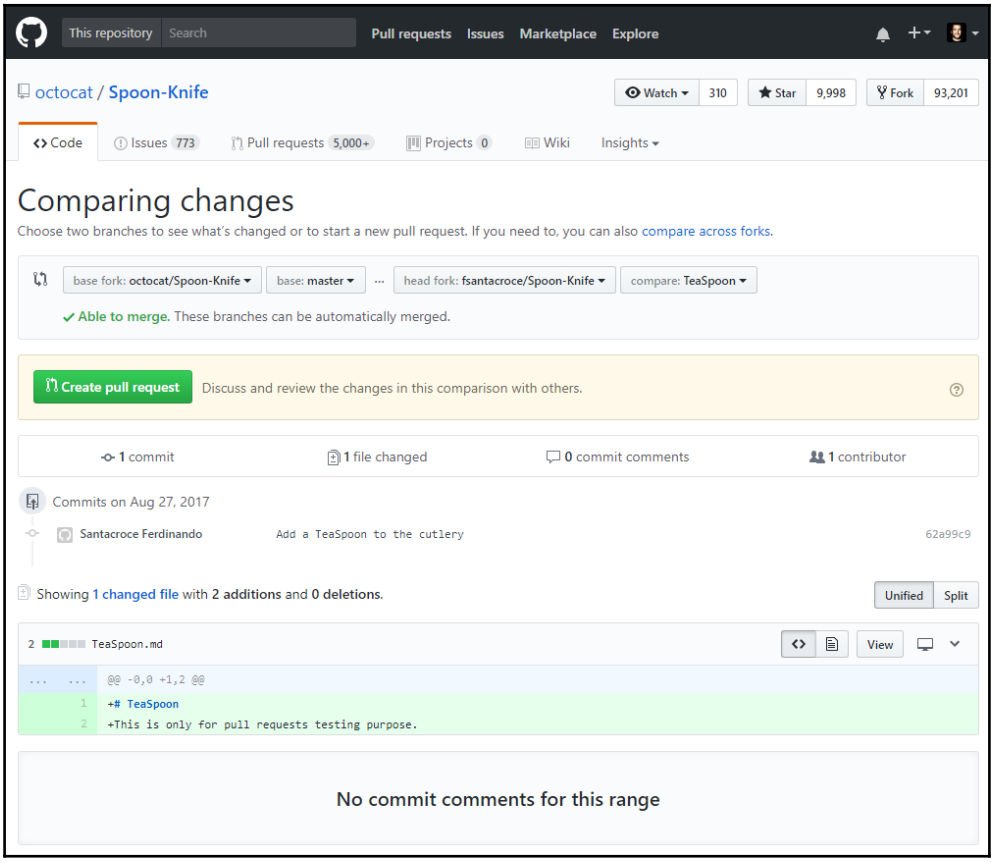
WOW! eBook
www.wowebook.org

Clicking that button makes GitHub open a new page; you now have to select the branch you want to compare to the original repository; look at the following screenshot:



WOW! eBook
www.wowebook.org

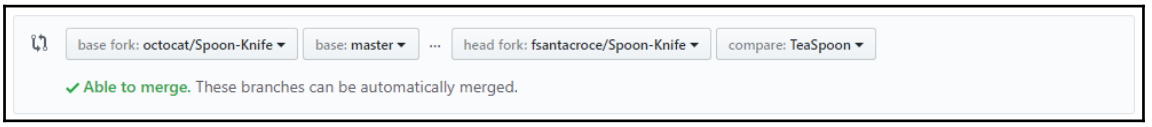
Go to the branches combo (1), select **TeaSpoon** branch (2), and then GitHub will show you something similar to the following screenshot:



This is a report, where you can see what you are going to propose: a commit containing a new file.

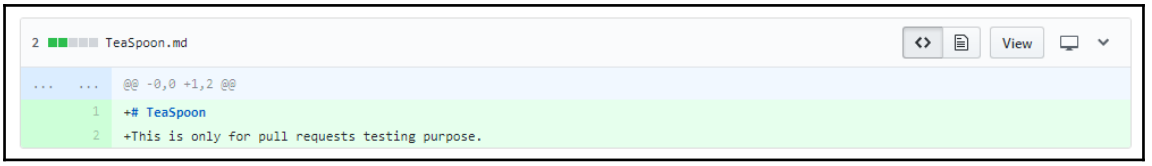
But let me analyze the page.

In the top left corner of the preceding screenshot, you will find what branches GitHub is about to compare for you; take a look at details in the following image:

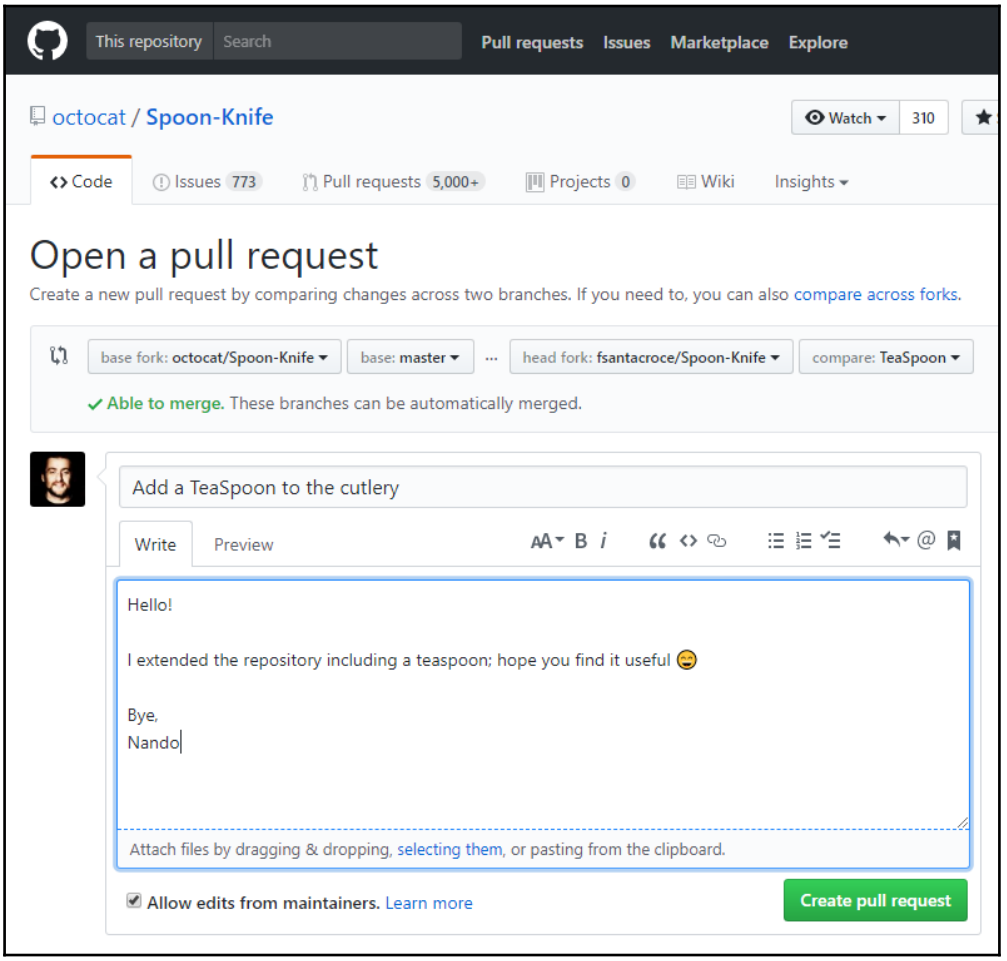


WOW! eBook
www.wowebook.org

This means that you are about to compare your local `TeaSpoon` branch with the original `master` branch of the `octocat` user. At the end of the page, you can see all the different details (files added, removed, changed, and so on):



Now, you can click on the green **Create pull request** button; the window in the following screenshot will appear:



WOW! eBook
www.wowebook.org

In the central part of the page, you can describe the work you did in your branch. A green **Able to merge** text on the top left informs you that these two branches can be automatically merged (there are no unresolved conflicts; that is always good if you want to see your work considered).

And now the last step: click the **Create pull request** button to send your request to the original author, letting him get your work and analyze it before accepting the pull request.

At this point, a new conversation begins, where you and the project collaborators can start to discuss your work; during this period, you and other collaborators can change the code to better fit common needs, until an original repository collaborator decides to accept your request or discard it, closing the pull request.

Summary

In this chapter, we finally got in touch with the Git ability to manage multiple remote copies of repositories. This gives you a wide range of possibilities to better organize your collaboration workflow inside your team. In the next chapter, you will learn some advanced techniques using well-known and niche commands. This will make you a more secure and proficient Git user, allowing you to resolve some common issues that occur in a developer's life with ease.

WOW! eBook
www.wowebook.org

4

Git Fundamentals - Niche Concepts, Configurations, and Commands

This chapter is a collection of short but useful tricks to make our Git experience more comfortable. In the first three chapters, we learned all the concepts we need to take the first steps into versioning systems using the Git tool; now it's time to go a little bit in depth to discover some other powerful weapons in the Git arsenal, and how to use them (without shooting yourself in the foot, preferably).

Dissecting Git configuration

In the first part of this chapter, you will learn how to enhance our Git configuration to better fit your needs and speed up the daily work: it's time to become familiar with configuration internals.

Configuration architecture

Configuration options are stored in plain text files. The `git config` command is just a convenient tool to edit these files without the hassle of remembering where they are stored and opening them in a text editor.

WOW! eBook
www.wowebook.org

Configuration levels

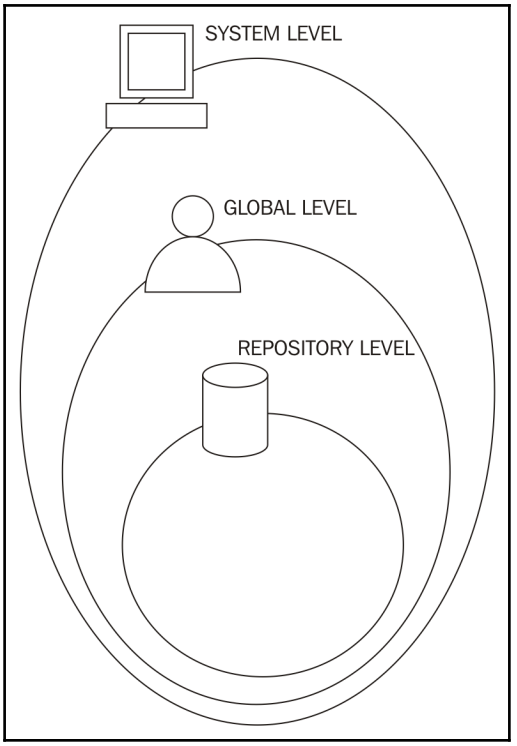
In Git, we have *three configuration levels*:

- System
- Global (user-wide)
- Repository

There are different configuration files for every different configuration level.

You can basically set every parameter at every level, according to your needs. If you set the same parameters at different levels, the lowest-level parameter hides the top-level ones; so, for example, if you set `user.name` at the global level, it will hide the one eventually set up at the system level, and if you set it at the repository level, it will hide the one specified at the global level and the one eventually set up at the system level.

The following figure will help you to better understand these levels:



WOW! eBook
www.wowebook.org

System level

The system level contains **system-wide configurations**; if you edit the configuration at this level, *every user and every user's repository will be affected*.

This configuration is stored in the `gitconfig` file usually located in:

- Windows: `C:\Program Files\Git\etc\gitconfig`
- Linux: `/etc/gitconfig`
- macOS: `/usr/local/git/etc/gitconfig`

To edit parameters at this level, you have to use the `--system` option; please note that it requires administrative privileges (for example, root permission on Linux and macOS). Anyway, as a rule of thumb, *editing the configuration at system level is discouraged*, in favor of per user configuration modification.

Global level

The global level contains **user-wide configurations**; if you edit the configuration at this level, *every user's repository will be affected*.

This configuration is stored in the `.gitconfig` file usually located in:

- Windows: `C:\Users\<UserName>\.gitconfig`
- Linux: `~/.gitconfig`
- macOS: `~/.gitconfig`

To edit parameters at this level, you have to use the `--global` option.

Repository level

The repository level contains **repository only configurations**; if you edit the configuration at this level, *only the repository in use will be affected*.

This configuration is stored in the `config` file located in the `.git` repository subfolder:

- Windows: `C:\<MyRepoFolder>\.git\config`
- Linux: `~/<MyRepoFolder>/.git/config`
- macOS: `~/<MyRepoFolder>/.git/config`

To edit parameters at this level, you can use the `--local` option or simply avoid using any option, as this is the default one.

WOW! eBook
www.wowebook.org

Listing configurations

To get a list of all the configurations currently in use, you can run the `git config --list` command; if you are inside a repository, it will show all the configurations, from repository to system level. To filter the list, append optionally `--system`, `--global` or `--local` options to obtain only the desired level configurations:

```
[1] ~/grocery (master)
$ git config --list --local
core.repositoryformatversion=0
core.filemode=false
core.bare=false
core.logallrefupdates=true
core.symlinks=false
core.ignorecase=true
user.name=Ferdinando Santacroce
user.email=ferdinando.santacroce@gmail.com
```

Editing configuration files manually

Even if it is generally discouraged, you can modify Git configurations directly by editing the files. Git configuration files are quite easy to understand, so when you look on the internet for a particular configuration you want to set, it is not unusual to find just the right corresponding text lines; the only little foresight in such cases is to back up files before editing them, just in case you mess with them. In the following paragraphs, we will try to make some changes in this manner.

Setting up some other environment configurations

Using Git can be a painful experience if you are not able to place it conveniently inside your work environment. Let's start to shape some rough edges using a bunch of custom configurations.

Basic configurations

In previous chapters, we have seen that we can change a Git variable value using the `git config` with the `<variable.name> <value>` syntax. In this section, we will make use of the `config` command to vary some Git behaviors.

WOW! eBook
www.wowebook.org

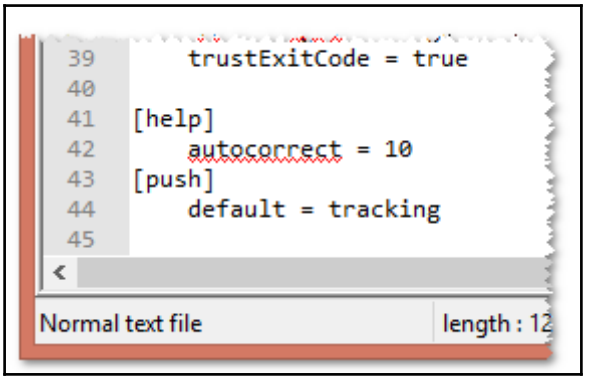
Typos autocorrection

So, let's try to fix an annoying question about typing command: *typos*. I often find myself re-typing the same command two or more times; Git can help us with embedded *autocorrection*, but we first have to enable it. To enable it, you have to modify the `help.autocorrection` parameter, defining how many tenths of a second Git will wait before running the assumed command; so giving a `help.autocorrect 10`, Git will wait for a second:

```
[2] ~/grocery (master)
$ git config --global help.autocorrect 10

[3] ~/grocery (master)
$ git chekcout
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 1.0 seconds automatically...
```

To abort the auto-correction, simply type *Ctrl+C*.

Now that we know about configuration files, you can note that the parameters we set by the command line are in this form: `section.parameter_name`. You can see the section names within `[]` if you look in the configuration file; for example, in `C:\Users\<UserName>\.gitconfig`:



Push default

We already talked about the `git push` command and its default behavior. To avoid annoying issues, it is good practice to set a more convenient default behavior for this command.

WOW! eBook
www.wowebook.org

There are two ways we can do this. First one: set Git to ask us the name of the branch we want to push every time, so a simple `git push` will have no effect. To obtain this, set `push.default` to `nothing`:

```
[1] ~/grocery-cloned (master)
$ git config --global push.default nothing

[2] ~/grocery-cloned (master)
$ git push
fatal: You didn't specify any refspecs to push, and push.default is
"nothing".
```

As you can see, now Git pretends that you specify the target branch at every push.

This is maybe too restrictive, but at least you can avoid common mistakes such as pushing some personal local branches to the remote, thus generating confusion in the team.

Another way to save yourself from this kind of mistake is to set the `push.default` parameter to `simple`, allowing Git to push only when there is a remote branch with the same name as the local one:

```
[3] ~/grocery-cloned (master)
$ git config --global push.default simple

[4] ~/grocery-cloned (master)
$ git push
Everything up-to-date
```

This will push the local tracked branch to the remote.

Defining the default editor

Some people really don't like `vim`, even only for writing commit messages; if you are one of them, there is good news: you can change it by setting the `core.default` config parameter:

```
[1] ~/grocery (master)
$ git config --global core.editor notepad
```

Obviously, you can set nearly all available text editors on the market. If you are a Windows user, remember that the full path of the editor has to be in the `PATH` environment variable; basically, if you can run your preferred editor by typing its executable name in a DOS shell, you can use it even in a Bash shell with Git.

WOW! eBook
www.wowebook.org

Other configurations

You can browse a wide list of other configuration variables at `git-scm.com/docs/git-config`.

Git aliases

We already mentioned Git aliases and their purpose; in this section, I will suggest only a few more, to help make things easier.

Shortcuts to common commands

One thing you may find useful is to *shorten common commands* such as `git checkout` and so on; therefore, useful aliases can include the following:

```
[1] ~/grocery (master)
$ git config --global alias.co checkout

[2] ~/grocery (master)
$ git config --global alias.br branch

[3] ~/grocery (master)
$ git config --global alias.ci commit

[4] ~/grocery (master)
$ git config --global alias.st status
```

Another common practice is to shorten a command, adding one or more options you use all the time; for example, set a `git cm <commit message>` command shortcut to the alias `git commit -m <commit message>`:

```
[5] ~/grocery (master)
$ git config --global alias.cm "commit -m"

[6] ~/grocery (master)
$ git cm "My commit message"
On branch master
nothing to commit, working tree clean
```


WOW! eBook
www.wowebook.org

Creating commands

Another common way to customize your Git experience is to *create commands* you think should exist.

git unstage

The classic example is the `git unstage` alias:

```
[1] ~/grocery (master)
$ git config --global alias.unstage 'reset HEAD --'
```

With this alias, you can remove a file from the index in a more meaningful way, compared to the equivalent `git reset HEAD -- <file>` syntax:

```
[2] ~/grocery (master)
$ git unstage myFile.txt
```

Now behaves the same as:

```
[3] ~/grocery (master)
$ git reset HEAD -- myFile.txt
```

git undo

Want a fast way to revert the last ongoing commit? Create a `git undo` alias:

```
[1] ~/grocery (master)
$ git config --global alias.undo 'reset --soft HEAD~1'
```

You can obviously use `--hard` instead of `--soft`, or go with the default `--mixed` option.

git last

A `git last` alias is useful to read about your last commit:

```
[1] ~/grocery (master)
$ git config --global alias.last 'log -1 HEAD'

[2] ~/grocery (master)
$ git last
```


WOW! eBook
www.wowebook.org

```
commit b25ffa60f44f6fc50e81181cab87ed3dbf3b172c
Author: Ferdinando Santacroce <ferdinando.santacroce@gmail.com>
Date:   Thu Jul 27 15:12:48 2017 +0200

    Add an apricot
```

git difflast

With the `git difflast` alias, you can see a `diff` against your last commit:

```
[1] ~/grocery (master)
$ git config --global alias.difflast 'diff --cached HEAD^'

[2] ~/grocery (master)
$ git difflast
diff --git a/shoppingList.txt b/shoppingList.txt
index d362b98..08e7361 100644
--- a/shoppingList.txt
+++ b/shoppingList.txt
@@ -4,3 +4,4 @@ orange
 peach
 grape
 blackberry
+apricot
```

Advanced aliases with external commands

If you want the alias to run external shell commands, instead of a Git sub-command, you have to prefix the alias with a `!`:

```
$ git config --global alias.echo !echo
```

Suppose you are annoyed by the canonical `git add <file>` plus `git commit <file>` sequence of commands, and you want to do it in a single shot; you can call the `git` command twice in sequence by creating this alias:

```
$ git config --global alias.cm '!git add -A && git commit -m'
```

With this alias you commit a file, adding it before if necessary.

Have you noted that I set the `cm` alias again? If you set an already configured alias, the previous alias will be overwritten.

WOW! eBook
www.wowebook.org

There are also aliases that define and use complex functions or scripts, but I'll leave it to the curiosity of the reader to explore these aliases. If you are looking for inspiration, take a look at this GitHub repository at `https://github.com/GitAlias/gitalias`.

Removing an alias

Removing an alias is quite easy; you have to use the `--unset` option, specifying the alias to remove. For example, if you want to remove the `cm` alias, you have to run:

```
$ git config --global --unset alias.cm
```

Note that you have to specify the configuration level with the appropriate option; in this case, we are removing the alias from the user (`--global`) level.

Aliasing the git command itself

I've already said I'm a bad typist; if you are too, you can alias the git command itself (using the default `alias` command in Bash):

```
$ alias gti='git'
```

In this manner, you will save some other keyboard strokes. Note that this is not a Git alias, but a Bash shell alias.

Useful techniques

In this section, we will improve our skills, learning some techniques that will come in handy in different situations.

Git stash - putting changes temporally aside

It sometimes happens that you need to switch branches for a moment, but some changes are in progress in the current branch. To put aside those changes for a while, we can use the `git stash` command: let's give it a try in our `grocery` repository.

WOW! eBook
www.wowebook.org

Append a new fruit to the shopping list, then try to switch branch; Git won't allow you to do so, because with the checkout you would lose your local (not yet committed) changes to the `shoppingList.txt` file. So, type the `git stash` command; your changes will be set apart and removed from your current branch, letting you switch to another one (`berries`, in this case):

```
[1] ~/grocery (master)
$ echo "plum" >> shoppingList.txt

[2] ~/grocery (master)
$ 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:   shoppingList.txt

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

[3] ~/grocery (master)
$ git checkout berries
error: Your local changes to the following files would be overwritten by
checkout:
        shoppingList.txt
Please commit your changes or stash them before you switch branches.
Aborting

[4] ~/grocery (master)
$ git stash
Saved working directory and index state WIP on master: b25ffa6 Add an
apricot
HEAD is now at b25ffa6 Add an apricot

[5] ~/grocery (master)
$ git checkout berries
Switched to branch 'berries'
```


WOW! eBook
www.wowebook.org

How does `git stash` work? Actually, `git stash` is a fairly complex command. It basically saves from two up to three different commits:

- A new *WIP commit* containing the actual state of the working copy; it contains all the tracked files, and their modifications.
- An *index commit*, as a parent of the WIP commit. This contains stuff added to the staging area.
- An optional third commit, let's call it an *untracked files commit*, which contains untracked files (using the `--include-untracked` option) or untracked plus previously ignored files (using the `--all` option).

Let's take a look at the actual situation in our repository using the `git log` command:

```
[6] ~/grocery (berries)
$ git log --oneline --graph --decorate --all
*   fedc4cf (refs/stash) WIP on master: b25ffa6 Add an apricot
|\
| * 7312ff0 index on master: b25ffa6 Add an apricot
|/
* b25ffa6 (master) Add an apricot
* 280e7a8 Cherry picked the blackberry
* 5dc3352 Add a grape
* de8bcb9 Add a peach
| * 362f8ec (HEAD -> berries) Add a strawberry
| * f037469 (melons) Add a watermelon
| * af9b640 Add a blackberry
|/
* 00404b4 Add an orange
* f583fdc Add an apple
* 40d865b Add a banana to the shopping list
```

As you can see, in this case there are only two commits. The *WIP commit*, `fedc4cf`, is the one with the message starting with *WIP on master*, where `master` is of course the branch where `HEAD` was at the time of the `git stash` command run. The *index commit*, `7312ff0`, is the one with the message starting with *index on master*.

The *WIP commit* contains the unstaged changes made to tracked files; as you can see, the *WIP commit* has two parents: one is the *index commit*, containing staged changes, the other is the last commit on the `master` branch, where `HEAD` was and where we run the `git stash` command.

WOW! eBook
www.wowebook.org

With all this shelved information, Git can then re-apply your work on top of the `master` branch when you finish your job on the `berries` branch; a stash can be applied wherever you want, and more than once if you like.

Using the `git stash` command, we actually used the `git stash save` subcommand, the default option. The `save` subcommand saves changes to tracked files only, using a default set of messages for these *special commits* we see.

To retrieve a stash, the command is `git stash apply <stash>`; it applies changes within the two commits, eventually modifying your working copy and staging area. The stash will not be deleted after the apply; you can do it manually using the `git stash drop <stash>` subcommand. Another way implies the `git stash pop <stash>` subcommand: it applies the stash and then deletes it.

While using these subcommands, you can refer to the various stashes you did in the past using different notations; the most common is `stash@{0}`, where `0` means *the last stash you did*. To retrieve the penultimate, you can use `stash@{1}` and so on.

To make a complete example, let's drop the actual stash without applying it, and then do a new one following these steps:

1. Drop the last stash created using `git stash drop` (`git stash clear` drops all the stashes).
2. Append a new fruit to the shopping list (for example, a plum) and add it to the staging area.
3. Then add another one (for example, a pear) but avoid adding it to the staging area.
4. Now create a new untracked file (for example, `notes.txt`).
5. Finally, create a new stash using `-u` (the `--include-untracked` option).

Here is the complete list of commands:

```
[1] ~/grocery (master)
$ echo "plum" >> shoppingList.txt

[2] ~/grocery (master)
$ git add .

[3] ~/grocery (master)
$ echo "pear" >> shoppingList.txt

[4] ~/grocery (master)
$ echo "Reserve some tropical fruit for next weekend" > notes.txt
```


WOW! eBook
www.wowebook.org

```
[5] ~/grocery (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   shoppingList.txt

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:   shoppingList.txt

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

        notes.txt

[6] ~/grocery (master)
$ git stash -u
Saved working directory and index state WIP on master: b25ffa6 Add an
apricot
HEAD is now at b25ffa6 Add an apricot
```

OK, let's see what happened using the `git log` command:

```
[7] ~/grocery (master)
$ git log --oneline --graph --decorate --all
*-.   87b1d8b (refs/stash) WIP on master: b25ffa6 Add an apricot
|\ \
| | * b07c304 untracked files on master: b25ffa6 Add an apricot
| * ad2efef index on master: b25ffa6 Add an apricot
|/
* b25ffa6 (HEAD -> master) Add an apricot
* 280e7a8 Cherry picked the blackberry
* 5dc3352 Add a grape
* de8bcb9 Add a peach
| * 362f8ec (berries) Add a strawberry
| * f037469 (melons) Add a watermelon
| * af9b640 Add a blackberry
|/
* 00404b4 Add an orange
* f583fdc Add an apple
* 40d865b Add a banana to the shopping list
```


WOW! eBook
www.wowebook.org

As you can see, this time there is one more commit, the *untracked files* one: this commit contains the `notes.txt` file, and figures as an additional parent for the *WIP commit*.

Summarizing, you basically use the `git stash save` command (with the `-u` or `--all` option if needed) to shelve your modification and then `git stash apply` to retrieve them; I suggest using `git stash apply` and then `git stash drop` instead of `git pop` to have a chance to redo your stash application when needed, or when your stash is not as trivial as usual.

To take a look at all the options for this command, please refer to the `git stash --help` output.

Git commit amend - modify the last commit

This trick is for people that don't double-check what they're doing. If you have pressed the enter key too early, there's a way to modify the last commit message or add that file you forgot, using the `git commit` command with the `--amend` option:

```
$ git commit --amend -m "New commit message"
```

Please note that with the `--amend` option, you are actually re-doing the commit, which will have a new hash; if you already pushed the previous commit, changing the last commit is not recommended - rather, it is deplorable.

If you amend an already pushed commit, then push the new one, you are basically discarding the latest commit on a branch, replacing it with the newly amended one: for those who will pull the branch, this can lead to some confusion, as they will see their local branch losing the last commit, replaced by a new one.

Git blame - tracing changes in a file

Working on source code in a team, it is not uncommon to have the need to look at the last modifications made to a particular file to better understand how it evolved over time. To achieve this result, we can use the `git blame <filename>` command.

Let's try it inside the `Spoon-Knife` repository to see changes made to the `README.md` file during a specific time:

```
[1] ~/Spoon-Knife (master)
$ git blame README.md
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 1) ### Well hello there!
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 2)
```


WOW! eBook
www.wowebook.org

```
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 3) This repository is meant
to provide an example for *forking* a repository on GitHub.
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 4)
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 5) Creating a *fork* is
producing a personal copy of someone else's project. Forks act as a sort of
bridge between the original repository and your personal copy. You can
submit *Pull Requests* to help make other people's projects better by
offering your changes up to the original project. Forking is at the core of
social coding at GitHub.
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 6)
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 7) After forking this
repository, you can make some changes to the project, and submit [a Pull
Request](https://github.com/octocat/Spoon-Knife/pulls) as practice.
bb4cc8d3 (The Octocat 2014-02-04 14:38:36 -0800 8)
d0dd1f61 (The Octocat 2014-02-12 15:20:44 -0800 9) For some more
information on how to fork a repository, [check out our guide, "Forking
Projects""](http://guides.github.com/overviews/forking/). Thanks!
:sparkling_heart:
```

As you can see, the result reports all the affected lines of the `README.md` file; for every line, you can see the commit hash, the author, the date, and the row number of the text file lines.

Suppose now you found that the modification you are looking for is the one made in the `d0dd1f61` commit; to see what happened there, type the `git show d0dd1f61` command:

```
[2] ~/Spoon-Knife (master)
$ git show d0dd1f61
commit d0dd1f61b33d64e29d8bc1372a94ef6a2fee76a9
Author: The Octocat <octocat@nowhere.com>
Date:   Wed Feb 12 15:20:44 2014 -0800

    Pointing to the guide for forking

diff --git a/README.md b/README.md
index 0350da3..f479026 100644
--- a/README.md
+++ b/README.md
@@ -6,4 +6,4 @@ Creating a *fork* is producing a personal copy of someone
else's project. Forks

 After forking this repository, you can make some changes to the project,
and submit [a Pull Request](https://github.com/octocat/Spoon-Knife/pulls)
as practice.

-For some more information on how to fork a repository, [check out our
guide, "Fork a Repo"](https://help.github.com/articles/fork-a-repo).
Thanks! :sparkling_heart:
+For some more information on how to fork a repository, [check out our
```


WOW! eBook
www.wowebook.org

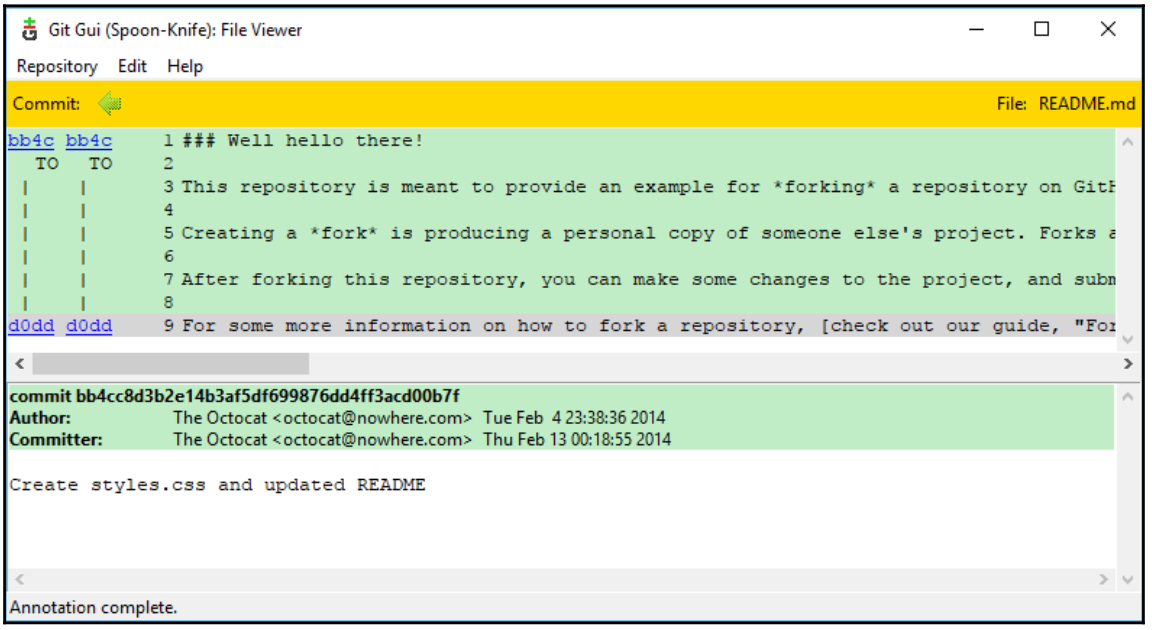
```
guide, "Forking Projects""](http://guides.github.com/overviews/forking/).
Thanks! :sparkling_heart:
```

The `git show` command is a multi-purpose command that can show to you one or more objects; in this case, we have used it to show the modification made in a particular commit using the `git show <commit-hash>` format.

The `git blame` and `git show` commands have a quite long list of options; the purpose of this section is to only point the reader to the way they can trace changes on a file; you can inspect other possibilities using the ever-useful `git <command> --help` command.

The last tip I want to suggest is to use the Git GUI:

```
[3] ~/Spoon-Knife (master)
$ git gui blame README.md
```



WOW! eBook
www.wowebook.org

With the help of the GUI, things are even easier to understand.

Tricks

In this section, I would like to suggest just a bunch of tips and tricks I have found useful in the past.

Bare repositories

Bare repositories are repositories that do not contain working copy files, but only the `.git` folder. A bare repository is essentially *for sharing*: if you use Git in a centralized way, pushing and pulling to a common remote (a local server, a GitHub repository, and so on), you will agree that the remote has no interest in checking out files you work on; the scope of that remote is only to be a central point of contact for the team, so having working copy files in it is only a waste of space as no one will edit them directly on the remote.

If you want to set up a bare repository, you only have to use the `--bare` option:

```
$ git init --bare NewRepository.git
```

As you may have noticed, I called it `NewRepository.git`, using a `.git` extension; this is not mandatory, but is a common way to identify bare repositories. If you pay attention, you will note that even in GitHub every repository ends with a `.git` extension.

Converting a regular repository to a bare one

It can happen that you start working on a project in a local repository, and then you feel the need to move it to a centralized server to make it available for other people or from other locations.

You can easily convert a regular repository to a bare one using the `git clone` command with the same `--bare` option:

```
$ git clone --bare my_project my_project.git
```

In this manner, you have a 1:1 copy of your repository in another folder, but in a bare version, ready to be pushed.

WOW! eBook
www.wowebook.org

Backup repositories

If you need a backup, there are two commands you can use: one for archiving only files and one for backing up the entire bundle, including versioning information.

Archiving the repository

To archive the repository without including versioning information, you can use the `git archive` command; there are many output formats but the classic one is the `.zip` one:

```
$ git archive master --format=zip --output=../repoBackup.zip
```

Please note that using this command is not the same as backing up folders in a filesystem; as you will have noticed, the `git archive` command can produce archives in a smarter way, including only files in a branch or even in a single commit; for example, by doing this you are archiving only the last commit:

```
$ git archive HEAD --format=zip --output=../headBackup.zip
```

Archiving files in this way can be useful if you have to share your code with people that don't have Git installed.

Bundling the repository

Another interesting command is the `git bundle` command. With `git bundle`, you can export a snapshot from your repository and then restore it wherever you want.
Suppose you want to clone your repository on another computer, and the network is down or absent; with this command, you can create a `repo.bundle` file of the `master` branch:

```
$ git bundle create ../repo.bundle master
```

With this other command, we can restore the bundle in the other computer using the `git clone` command:

```
$ cd /OtherComputer/Folder
$ git clone repo.bundle repo -b master
```


WOW! eBook
www.wowebook.org

Summary

In this chapter, we enhanced our knowledge about Git and its wide set of commands. We discovered how configuration levels work, and how to set our preferences using Git by, for example, adding useful command aliases to the shell. Then we looked at how Git deals with stashes, providing the way to shelve then and reapply changes.

Furthermore, we added some other techniques to our skill set, learning some things we will use as soon as we start to use Git extensively. Some simple tricks provide a way to stimulate the curiosity of the reader: Git has a lot more commands to explore.

In the next chapter, we will leave the console for a while, and talk about strategies to better organize our repositories. We will try to learn how to make significant commits, and we will get to know some of the adoptable flows to reconcile Git with our way of working.

WOW! eBook
www.wowebook.org

5

Obtaining the Most - Good Commits and Workflows

Now that we have acquired some familiarity with Git and versioning systems, it's time to look at the whole thing from a much higher perspective, to become aware of common patterns and procedures.

In this chapter, we will walk through some of the most common ways to organize and build meaningful commits and repositories, obtaining not only a well-organized code stack, but even a meaningful source of information.

The art of committing

While working with Git, committing seems the easiest part of the job: you add files, write a short comment, and then you're done. But it is just for its simplicity that often, especially at the very beginning of your experience, you acquire the bad habit of doing terrible commits: too late, too big, too short, or simply equipped with bad messages.

Now we will take some time to identify possible issues, like unmeaning or too large commits, drawing attention to tips and hints to get rid of those bad habits.

WOW! eBook
www.wowebook.org

Building the right commit

One of the harder skills to acquire while programming in general is to **split the work into small and meaningful tasks**.

Too often, I have experienced this scenario: you start to fix a small issue in a file; then you see another piece of code that can be easily improved, even if it's not related to what you are working on now - you can't resist, and you fix it. At the end, and after some time, you find yourself with a ton of **concurrent** files and **changes** to commit.

At this point, *things get worse*, because usually *programmers are lazy people*, so they don't write all the important things to describe changes in the commit message. In commit messages, you start to write sentences like "*Some fixes to this and that*", "*Removed old stuff*", "*Tweaks*" and so on, without anything that helps other programmers to understand what you have done:

Courtesy of http://xkcd.com/1296/

At the end, you realize *your repository is only a dump*, where you empty your index only now and then. I have seen some people committing only at the end of the day (and not every day), only to keep a backup of the data or because someone else needed the changes reflected on their computer.

Another side effect is that the resulting *repository history becomes useless* for anything other than retrieving the contents at a given point in time.

WOW! eBook
www.wowebook.org

The following tips can help you turn your VCS from a backup system into *a valuable tool for communication and documentation*.

Making only one change per commit

After the routine morning coffee, you open your editor and then you start to work on a bug: `BUG42`. Working around fixing the bug in the code, you realize that fixing `BUG79` will require tweaking just a single line of code, so you fix it, but you not only change that awful class name, but also add a good-looking label to the form and make a few more changes. *The damage is done now*.

How can you now wrap up all that work in a *meaningful commit*? Maybe in the meantime you went home for lunch, talked to your boss about another project, and even you can't remember exactly all the little things you did.

In this scenario, there is only one way to **limit the damage**: *split files to commit in more than one commit*. Sometimes this helps to reduce the pain, but it is only a *palliative*: too often you modify the same file for *different reasons*, so doing that is quite difficult, if not impossible.

The only way to solve this problem completely is to **only make one change per commit**. It seems easy, I know, but is quite difficult to acquire this ability. There are no tools for it; no one but you can help, as it requires **discipline**, *the most lacking virtue in creative people* (like programmers).

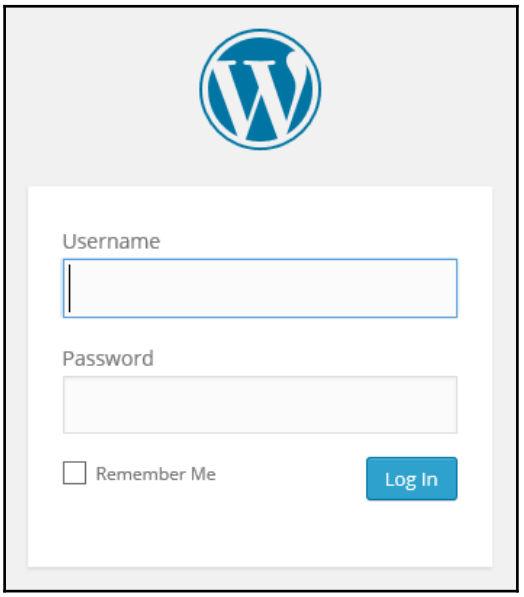
There are some tips to pursue this aim; let's have a look at them together.

Splitting up features and tasks

As said before, breaking up things to do is a fine art. If you know and adopt some **Agile Movement** techniques, you have probably faced these problems already, so you have an advantage; otherwise you will need to make a little more effort, but it isn't anything you can't achieve.

WOW! eBook
www.wowebook.org

Consider you have been assigned to add the **Remember Me** check in the login page of a web application, like the following one:



This feature is quite small, but implies changes at different levels. To accomplish this, you'll have to:

1. Modify the UI to add the check control.
2. Pass the *is checked* information through different layers.
3. Store this information somewhere.
4. Retrieve this information when needed.
5. Invalidate (set it to false) following some kind of policy (after 15 days, after 10 logins, and so on).

Do you think you can do all these things in a shot? Yes? You are wrong! Even if you estimate a couple of hours for an ordinary task, remember that Murphy's Law applies: you will receive four calls, your boss will look for you for three different meetings and your computer will go up in flames.

WOW! eBook
www.wowebook.org

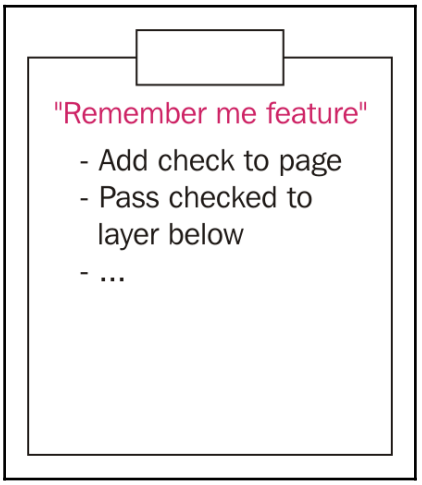
This is one of the first things to learn: **break up every piece of work into small tasks**. No matter if you use time-boxing techniques like the *Pomodoro Technique* or not, small things are easier to handle. I'm not talking about splitting hairs, but try to organize your tasks into things you can do in a defined amount of time, hopefully a bunch of half-hours, not days.

For more information on the Pomodoro Technique you can visit `https://cirillocompany.de/pages/pomodoro-technique` or Wikipedia `https://en.wikipedia.org/wiki/Pomodoro_Technique`

So, *take paper and a pen and write down all the tasks*, as we did before with the login page example. Do you think now you can do all those things in a small amount of time? Maybe yes, maybe not: some tasks are bigger than others. That's okay, this is not a scientific method, *it's a matter of experience*; can you split a task, creating two other meaningful tasks? Do it.

Can you? No problem, *don't try to split tasks if they lose meaning*.

Make a little notebook, like the one in the following picture - it will become one of your most precious tools:



WOW! eBook
www.wowebook.org

Writing commit messages before starting to code

Now you have a list of tasks to do; pick the first and... Start to code? No! Take another piece of paper and **describe every task's step with a sentence**; magically you will realize that every sentence can be the message of a single commit, where you describe features you deleted, added, or changed in your software.

This kind of prior preparation helps you in *defining modifications to implement* (letting *better software design* emerge), *focusing on what matters,* and *lowering the stress* to think about the versioning part of the work during the coding session. While you are facing a programming problem, your brain floods with little implementation details related to the code you are working on, so the fewer distractions, the better.

This is one of the best versioning related hints I ever received: if you have just quarter of an hour of spare time, I recommend reading the *Preemptive commit comments* blog post at `https://arialdomartini.wordpress.com/2012/09/03/pre-emptive-commit-comments/` by *Arialdo Martini*, which is where I learnt this trick.

Including the whole change in one commit

Making more than one change per commit is a bad thing, but even splitting a single change into more than one commit is considered harmful. As you may already know, in some trained teams you do not simply push your code to production; first you have to pass *code quality reviews*, where someone else tries to understand what you did to decide if your code is good or not (that is why there are *pull requests*, indeed). You can be the best developer in the world, but if the person at the other end can't get a sense of your commits, your work will probably be refused.

To avoid these unpleasant situations, you have to follow a simple rule: **don't do partial commits**. If time's up, if you have to go to that damn meeting (programmers hate meetings) or whatever, remember that you can save your work at any moment without committing, using the `git stash` command. If you want to close the commit, because you want to push it to the remote branch for backup purposes, remember that *Git is not a backup tool*: backup your stash on another disk, put it in the cloud, or simply end your work before leaving, but don't do commits like they are episodes of a TV series.

One more time, Git is a software tool like any other, and even it can fail: don't think that by using Git or other versioning systems you don't need backup strategies - backup local and remote repositories just the same as you backup all the other important things.

WOW! eBook
www.wowebook.org

Describing the change, not what have you done

Too often I read (and more often I wrote) commit messages like "*Removed this*", "*Changed that*", "*Added that one*" and so on.

Imagine you are going to work on the common "*lost password*" feature on your website; you'll probably find a message like this adequate: "*Added the lost password retrieval link to the login page*". This kind of commit message does not describe what modifications the feature brings to you, but what you did (and not everything you did). Try to answer sincerely: reading a repository history, do you want to read what every developer did? Or maybe it's better to read the feature implemented in every single commit?

Try to make the effort, and **start writing sentences where the change itself is the subject**, not what you did to implement it. *Use the imperative present tense* (for example, *fix*, *add*, or *implement*), describing the change in a small subject sentence, and then add some details (when needed) in other lines of text; "*Implement the password retrieval mechanism*" is a good commit message subject; if you find it useful then you can add some other information to get a well-formed message, like this:

```
"Implement the password retrieval mechanism

 - Add the "Lost password?" link into the login page
 - Send an email to the user with a link to renew the password"
```

Have you ever written a *changelog* for software by hand? I did, and it's one of the most boring things to do. If you don't like writing changelogs, like me, think of the repository history as your changelog: if you take the right care of your commit messages, you will get a beautiful changelog for free!

In the next paragraph, I will cover some other useful hints about good commit messages.

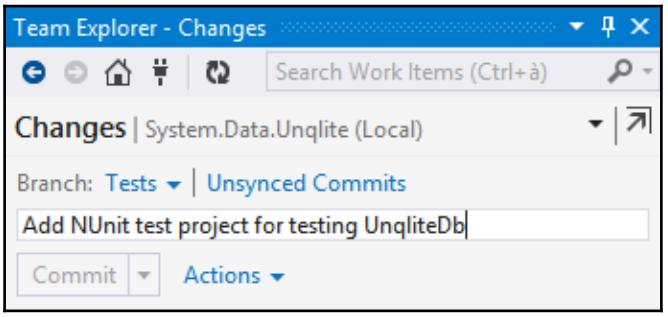
Don't be afraid to commit

Fear is one of the most powerful of emotions; it can drive a person to do the craziest things on Earth. One of the most common reactions to fear is the **breakdown**: *you don't know what to do, so you end up doing nothing*.

This is a common reaction even when you begin to use a new tool like Git, where gaining confidence can be difficult; because of the fear of making a mistake, you don't commit until you are obligated. **This is the real mistake: being scared**. In Git, you don't have to be scared; maybe the solution is not obvious, and maybe you have to dig on the internet to find the right way, but you can get away with small or no consequences, ever (well, unless you are a hard user of the `--hard` option).

WOW! eBook
www.wowebook.org

On the contrary, you have to make the effort to **commit often**, as soon as possible. The more frequently you commit, the smaller your commits; the smaller your commits, the easier it is to read and understand the changelog, and the easier it is to cherry-pick commits, and do code reviews. To help myself get used to committing this way, I followed this simple trick: write the commit message in Visual Studio before starting to write any code:



Try to do the same in your IDE or directly in the Bash shell, it helps a lot.

Isolating meaningless commits

The golden rule is to avoid them, but sometimes you need to commit *something that is not a real implementation*, but only a clean-up, like old comments deletion, formatting rearrangement, and so on.

In these cases, it is better to isolate these kind of code changes in separated commits. By doing this you prevent another team member from running towards you with a knife in his hand, frothing at the mouth. Don't commit meaningless changes, mixing them up with real ones, otherwise other developers (and you, after a couple of weeks) will not understand them while diffing.

The perfect commit message

Let me be honest: the perfect message does not exist. If you work alone, you probably find the best way for you, but when in a team there are different minds and different sensibilities, so what is good for me might not be as good for somebody else.

In this case you have to sit down around a table and make a retrospective, trying to end up with a shared standard; it may not be the one you prefer, but at least it's a way to find a common path.

WOW! eBook
www.wowebook.org

Rules for a good commit message really depend on the way you and your team work day by day, but some common hints can be applied by everyone; here they are.

Writing a meaningful subject

The subject of a commit is the most important part: its role is to make clear what the commit contains. Avoid technical details of other things - a common developer can understand opening the code, and focus on the big picture: remember that every commit is a sentence on the repository history. So, wear the hat of the changelog reader and try to write the most convenient sentence for him, not for you: use present tense, and write a 50 chars max sentence.

A good subject is one like this: "*Add the newsletter signup in homepage*".

As you can see, I used the *imperative past tense* and, more importantly, *I didn't say what I had done, but what the feature does*: it adds a newsletter signup box to my website.

The 50-char rule is due to the way you use Git from the shell or GUI tools; if you start to write long sentences, reviewing logs and so on can become a nightmare. So, don't try to be the *Stephen King* of commit messages: avoid adjectives and get straight to the point, you can then go more in-depth in the additional details lines.

One more thing: start with capital letters, and do not end sentences with periods - they are useless, and even dangerous.

Adding bulleted details lines when needed

Often you can say all that you want in 50 chars; in that case, use details lines. In this situation, the common rule is to *leave a blank line after the subject*, use a dash and go no longer than 72 chars:

```
"
Add the newsletter signup in homepage

- Add textbox and button on homepage
- Implement email address validation
- Save email in database"
```

In these lines go a little bit in depth, but not too much; try to describe the original problem (if you fixed it) or the original need, why this functionality has been implemented (what problem has been solved) and any possible limitations or known issues.

WOW! eBook
www.wowebook.org

Tying other useful information

If you use issue and project tracking systems, write down the issue number, bug id's, or anything else helpful:

```
"
Add the newsletter signup in homepage

- Add textbox and button on homepage
- Implement email address validation
- Save email in database

#FEAT-123: closed"
```

Special messages for releases

Another useful thing is to write *special format* commit messages for releases, so it will be easier to find them. I usually decorate subjects with some special characters, but nothing more; for highlighting a particular commit, like a release one, there is the `git tag` command, remember?

Conclusions

At the end, my suggestion is to try to compose your personal commit message standard, following previous hints and looking at message strategies adopted by great projects and teams around the web, but especially by doing it. Your standard will change for sure, as you evolve as a software developer and Git user, so start as soon as possible and let the time help you find the perfect way to write a commit message.

At least, don't imitate this link: `http://www.commitlogsfromlastnight.com`.

Adopting a workflow - a wise act

Now that we learnt how to perform good commits, it's time to fly higher and think of **workflows**. Git is a tool for versioning, but as with other powerful tools like knives, you can cut tasty sashimi or get hurt.

The thing that separates a great repository from a junkyard is the way you manage releases, the way you react when there is a bug to fix in particular version of your software, and the way you act when you have to make users able to beta-test incoming features.

These kinds of actions belong to ordinary administration for a modern software project, but too often I see teams out of breath because of the poor versioning workflows.

WOW! eBook
www.wowebook.org

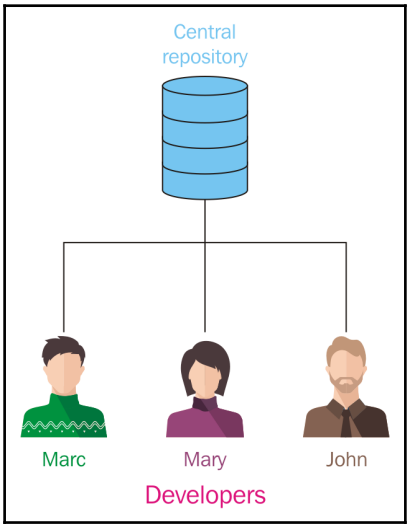
In this second part of the chapter, we will take a quick look at some of the most used workflows together with the Git versioning system.

Centralized workflows

As we used to do in other VCS like Subversion or so, even in Git it is not uncommon to adopt a *centralized way of working*. If you work in a team, it is often necessary to share repositories with others, so a *common point of contact* becomes indispensable.

We can assume that if you are not alone in your office, you will adopt one of the variations of this workflow. As we know, we can configure to get all the computers of our co-workers as remote, in a sort of *peer-to-peer* configuration, but you usually don't do this, because it quickly becomes too difficult to keep every branch in every remote in sync.

The scenario is represented in the following picture:



How they work

In this scenario, you usually follow these simple steps:

1. Someone initializes the remote repository (in a local Git server, on GitHub, BitBucket, or so on)

WOW! eBook
www.wowebook.org

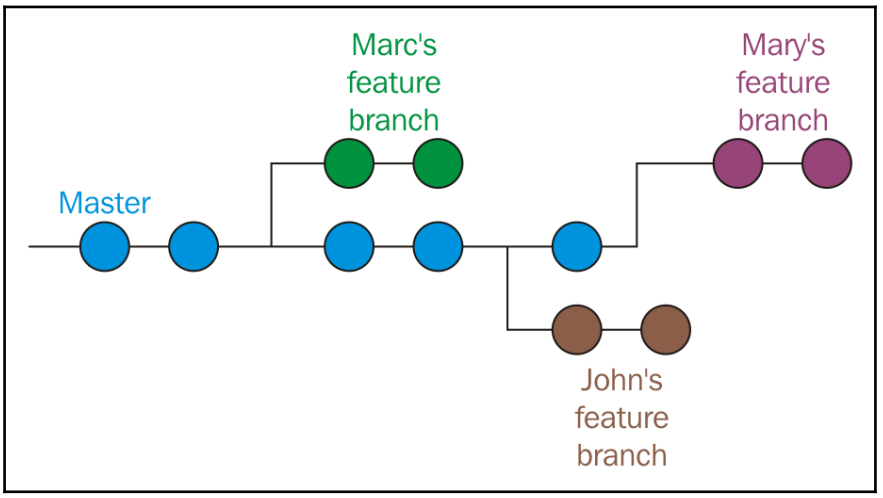
2. Other team members clone the original repository on their computer and start working
3. When the work is done, you push it to the remote to make it available to other colleagues

At this point, it is only a matter of internal rules and patterns.

Feature branch workflow

At this point, you will probably choose at least a *feature branch* approach, where every single developer works on his branch. When the work is done, the feature branch is ready to be merged onto the master branch; you will probably have to merge back from the `master` before, because one of your other colleagues has merged a feature branch after you started your one, but after that you basically have finished.

The following picture represents the branches evolution within the repository:



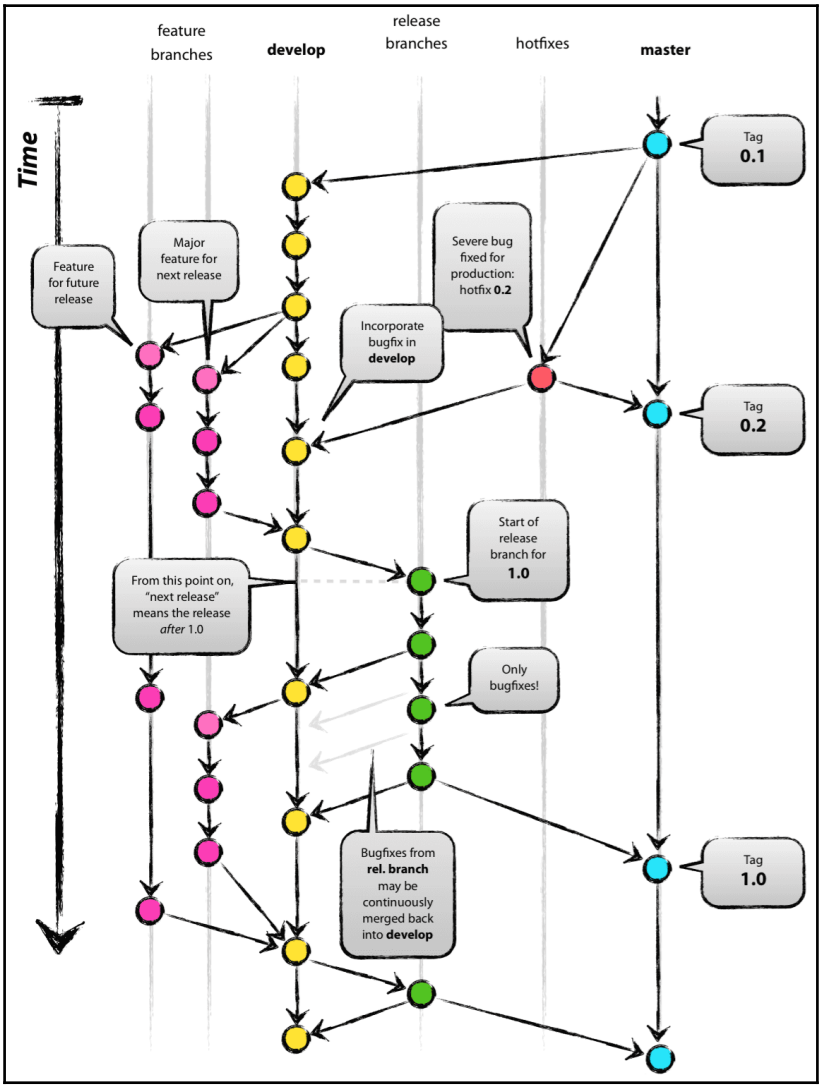
Gitflow

The **Gitflow** workflow comes from the mind of **Vincent Driessen**, a passionate software developer from the Netherlands; you can find his original blog post about it at `http://nvie.com/posts/a-successful-git-branching-model`.

WOW! eBook
www.wowebook.org

His workflow has gained success over the years, to the point that many other developers (including me!), teams, and companies are starting to use it. *Atlassian*, a well-known company that offers Git related services like *BitBucket*, integrates the Gitflow directly in their GUI tool, the nice *SourceTree*.

Even the Gitflow work flow is a centralized one, and it is well described by the following image:



WOW! eBook
www.wowebook.org

This workflow is based on the use of some **main branches**; what makes these branches special is nothing other than the significance we attribute to them: there are no *special branches* with *special characteristics* in Git, but we can certainly use them for different purposes.

Master branch

In Gitflow the `master` branch represents the final stage; merging your work in it is equal to making a *new release* of your software. You usually don't start new branches from the `master`; you do it only if there are severe bugs you have to fix instantly, even if that bug has been found and fixed in another evolving branch. This way of operating is superfast when you need to react to a painful situation. Other than that, the `master` branch is where you tag your release.

Hotfixes branches

Hotfixes branches are branches derived only from the `master`, as we said before; once you have fixed a bug, you merge the `hotfix` branch onto the `master`, to enable you to get a new release to ship. If the bug has not been resolved anywhere else in your repository, the strategy is to merge the `hotfix` branch into the `develop` branch. After that, you can delete the `hotfix` branch, as it has hit the mark.

In Git, there is *a trick to grouping similar branches*: you have to name them using a common prefix followed by a slash `/`; for the hotfix branches, the author recommends the `hotfix/<branchName>` prefix (for example `hotfix/LoginBug` or `hotfix/#123` for those using bug tracking systems, where `#123` is the bug ID).

These branches are usually not pushed to remote; you push them only if you need the help of other team members.

The develop branch

The `develop` branch is a sort of *staging* branch. When you start to implement a new feature, you have to create a new branch starting from `develop`; you will continue to work in that branch until you complete your task.

After the task completion, you can merge back to `develop` and delete your `feature` branch: as hotfix branches, these are only temporary branches.

WOW! eBook
www.wowebook.org

Like the master one, the `develop` branch is a **long living branch**: you will never close nor delete it.

This branch is pushed and shared to a remote Git repository.

The release branch

At some point, you need to wrap up the next release, including some of the features you implemented in the last few weeks. To prepare an incoming release you have to branch from `develop`, assigning the branch a name composed by the `release` prefix, followed by the numeric form of choice for your release (for example `release/1.0`).

Pay attention: **at this stage, no new feature is allowed**! You can no longer merge `develop` onto the `release` branch; you can create new branches from that branch only for bug-fixing; the purpose of this intermediate branch is to give the software to beta testers, allowing them to try it and send you feedback and bug tickets.

In case you have fixed some bug onto the `release` branch, the only thing to remember is to merge them into the `develop` branch, just to avoid the loss of the bug fix - the `release` branch will not be merged back to `develop`.

You can keep this branch alive as long as you want, until you decide the software is both mature and tested sufficiently to go into production: at this point you merge the `release` branch onto the `master` branch, making a new release.

After the merge to the `master` you have a choice: keep the `release` branch open, if you need to keep different releases alive, otherwise you can delete it. Personally, I always delete the `release` branch (as Vincent suggests), because I generally do frequent, small, and incremental releases (so I rarely need to fix an already shipped release) and because, as you certainly remember, you can open a brand-new branch from a commit (a tagged one in this case) whenever you want so, at the most, I will open it from that point only when necessary.

This branch is pushed and shared to a common remote repository.

The feature branches

When you have to start the implementation of a new feature, you have to create a new branch from the `develop` branch. Feature branches start with the `feature/` prefix (for example `feature/NewAuthenitcation` or `feature/#987` if you use some features tracking software, as `#987` is the feature ID).

WOW! eBook
www.wowebook.org

You will work on the feature release until you finish your work; I suggest you *frequently merge back* from `develop`: in case of concurrent modifications to the same files, you will resolve conflicts faster if you will resolve them earlier; then it is easier to resolve one or two conflicts at a time, rather than dozens at the end of the feature work.
Once your work is done, you merge the feature onto `develop` and you are done; you can now delete the `feature` branch.

Feature branches are mainly private branches, but you can push them to the remote repository in case you have to collaborate on it with some other team mates.

Conclusion

I really recommend taking a look at this workflow, as I can assure you there were no situations I have failed to solve using solve using it.
You can find a deeper explanation, with Git commands ready to use, on **Vincent Driessen's** already cited blog. You can even use the *gitflow commands* Vincent made to customize his Git experience; check these out on his GitHub account at `https://github.com/nvie/gitflow`.

GitHub flow

The previously described *GitFlow* has tons of followers, but it's always a matter of taste; someone else found it too complex and rigid for their situation, and in fact there are other ways to manage software repositories that have gained consensus during the last few years.

One of these is the workflow used at GitHub for internal projects and repositories; this workflow takes the name of **GitHub flow** and it has been firstly described by the well-known **Scott Chacon**, former GitHubber and *ProGit* book author, on his blog at `http://scottchacon.com/2011/08/31/github-flow.html`.

This workflow, compared to Gitflow, is better tailored for frequent releases, and when I say frequent, I mean very frequently, even twice a day. Obviously, this kind of flow works better on web projects, because to deploy you have to *only* put the new release on the production server; if you develop desktop solutions, you need a perfectly oiled update mechanism to do the same.

GitHub software basically doesn't have releases, because they deploy to production regularly, even more than once a day. This is possible due to a robust *Continuous Delivery* structure, which is not so easy to obtain; it requires some effort.

The GitHub flow is based on these simple rules.

WOW! eBook
www.wowebook.org

Anything in the master branch is deployable

Similar to GitFlow, even here in GitHub flow, deploy is done from the `master` branch. This is the only *main* branch in this flow; in Gitflow there are no `hotfix`, `develop`, or other particular branches. Bug fixes, new implementations and so on are constantly merged onto the `master` branch.

Other than this, code in the `master` branch is always in a *deployable* state; when you fix or add something new in a branch and then you merge it onto the `master`, you don't deploy automatically, but you can assume you changes will be up and running in a matter of hours.

Branching and merging constantly to the `master`, the production-ready branch, can be dangerous: you can easily introduce regressions or bugs, as no one other than you can check you have done a good job. This problem is avoided by a *social contract* commonly adopted by GitHub developers; in this contract, you promise to test your code before merging it to the `master`, assuring you that all automated tests have been successfully completed.

Creating descriptive branches off of master

In GitFlow you always branch from the `master`, so it's easy to get a forest of branches to look at when you have to pull one. To better identify them, in GitHub flow you have to use descriptive names to get meaningful *topic branches*. Even here it is a matter of good manners; if you start to create branches named *stuff-to-do* you will probably fail in adopting this flow. Some examples are `new-user-creation`, `most-starred-repositories`, and so on (note the use of the **Kebab Case**, `http://wiki.c2.com/?KebabCase`); using a common way to define topics, you will easily find branches you are interested in by looking for topics' keywords.

Pushing to named branches constantly

Another great difference when comparing GitHub flow to Gitflow is that in GitHub flow you push feature branches to the remote regularly, even if you are the only developer involved and interested. This is done for continuous integration and testing, or maybe also for backup purposes; regarding the backup part, even if I already exposed my opinion in merit, I can't say this is a bad thing.

WOW! eBook
www.wowebook.org

A thing I appreciate about Gitflow is that this habit of push every branch to the remote gives you the ability to see, with a simple `git fetch`, all the branches currently active, and so all the work in progress, even that of your team mates.

Opening a pull request at any time

In `Chapter 3`, *Git Fundamentals - Working Remotely*, we talked about GitHub and made a quick try with *Pull Requests*. We have seen that basically they are for *contributing*: you fork someone else's repository, create a new branch, make some modifications and then ask for a pull request from the original author.

In GitHub flow you use pull requests massively, even for asking another developer on your team to have a look at your work and help you, give you a hint, or review the work done. At this point you can start a discussion, using the GitHub pull request to chat and involving other people by putting in /CCing their username. In addition, the pull request feature lets you comment on even a single line of code in the diff view, making users involved able to discuss the work under revision.

Merging only after pull request review

You can understand now that the *pull requested branch stage* we have seen above becomes a sort of *review stage*, where other users can take a look at the code and even simply leave a positive comment, just a `+1` to let other users know that they are confident about the job, and that they approve its merge into master.

After this step, when the CI server says the branch still passes all the automated tests, you are ready to merge the branch in `master`.

Deploying immediately after review

At this stage, you merge your branch into `master` and the work is done. The deploy is not instantly fired, but at GitHub they have a very straight and robust deployment procedure, so they can do that easily. They deploy big branches with 50 commits but even branches with a single commit and a single line of code change, because deployment is very quick and cheap for them.

WOW! eBook
www.wowebook.org

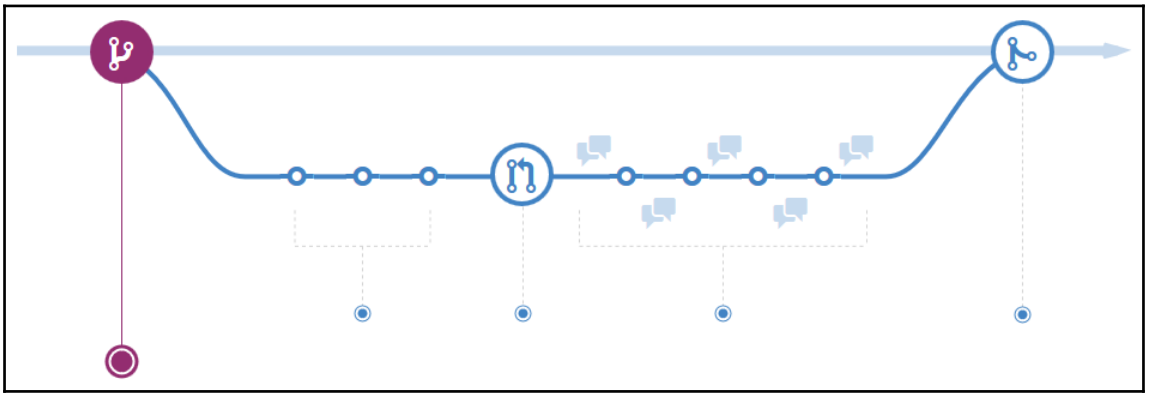
This is the reason why they can afford such a simple branching strategy, where you put on `master` and then you deploy without the need for passing through `develop` or `release` stage branches like in GitFlow.

Conclusions

I consider this flow very responsive and effective for web based projects, where basically you deploy to production without much regard for the versions of your software. Using only the `master` branch to derive and integrate branches is faster than light, but this strategy can be applied only if you have these prerequisites:

- A centralized remote ready to manage pull requests (as GitHub does)
- A good shared agreement about branch names and pull requests usage
- A very robust deploy system

This is a big picture of this flow, graphically represented in the following image; for more details, I recommend visiting the GitHub related page at `https://guides.github.com/introduction/flow/index.html`:

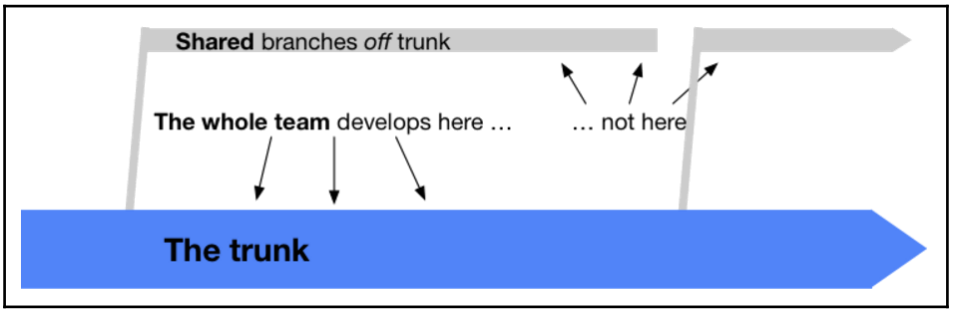


WOW! eBook
www.wowebook.org

Trunk-based development

These days, another strategy has regained a certain popularity among developers all around the world; its name says it all: *stop using branches*, use only the *master branch*!

In the picture below there's the essence of this flow:



This trend aims to fight the so called *merge hell*; this happens when branches diverge for too long, so merging them is a pain. Similar to the GitHub flow, here there aren't long living branches, but even feature branches are discouraged.

Continuous Integration and Continuous Delivery are under the light here, and this way of working really enforces these good practices we already know thanks to the **eXtreme Programming** (`http://www.extremeprogramming.org/`) mindset and practices.

This movement is too wide and deep to be discussed here in a couple of sentences, but it's worth reading its principles, as it makes you reflect on the topics a developer faces during his day by day work. So please take a minute and read more about it at `trunkbaseddevelopment.com`.

Other workflows

Obviously, there are many other workflows; I will spend just a moment on the one that (fortunately!) convinced Linus Torvalds to realize the Git VCS.

WOW! eBook
www.wowebook.org

Linux kernel workflow

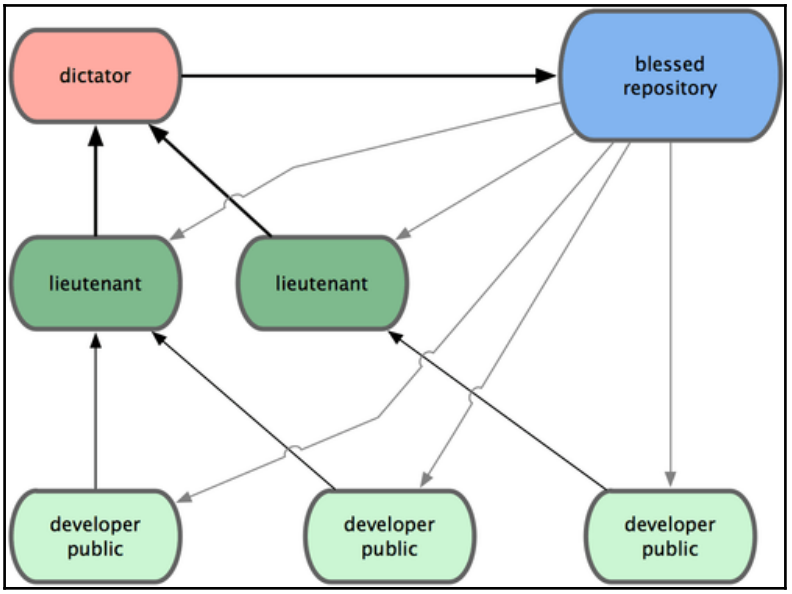
The **Linux kernel** uses a workflow that refers to the traditional way **Linus Torvalds** has driven its evolution during these years, based on a *military like hierarchy*.

Simple kernel developers work on their personal branches, rebasing the master branch on the reference repository, then push their branches to the *lieutenant developers* `master` branch. Lieutenants are developers that Linus assigned to particular topics and areas of the kernel because of their experience. When a lieutenant has his work done, he pushes it to the *benevolent dictator* `master` branch (Linus branch) and then if things are okay (it is not simple to cheat him), Linus pushes his master branch onto the *blessed repository*, the one the developers use to rebase from before starting their work.

This kind of workflow is not usual; Linus and the Linux kernel bandwagon crafted it as it exactly reflected the way they used to work on projects since the beginning, when developers used patches and email to forward their work to Linus Torvalds.

Having millions of lines of code to manage, and thousands of contributors, I find this hierarchy model to be a good compromise in terms of working scope, responsibility, and patches skimming.

The following picture helps you better understand this flow:



WOW! eBook
www.wowebook.org

Summary

In this chapter, we became aware of effective ways to use Git; I personally consider this chapter the most important for the *new Git user*, because it is only by applying some rules and discipline that you will obtain the most from this tool. So please pick up a good workflow (make your own, if necessary!), and pay attention to your commits: this is the only way to become a good versioning tool user, not only in Git.

In the next chapter, we will see some tips and tricks for using Git even if you have to deal with Subversion servers, and then we will take a quick look at migrating definitely from Subversion to Git.

WOW! eBook
www.wowebook.org

6
Migrating to Git

You often come to Git after using other versioning systems; there are many different VCS in the world but one of the most popular is **Subversion**, for sure.

Git and Subversion can coexist, as Git has some dedicated commands for exchanging data with Subversion.

The purpose of this chapter is to help developers who actually use Subversion to start using Git instantly, even if the rest of the team will continue to use Subversion.

In addition, the chapter covers definitive migration and preserving the changes' history for people who decide to abandon Subversion in favor of Git.

Before starting

In this first part of the chapter, we will take a look at best practices to maintain safety and work on actual SVN repository with no hassles.

Bear in mind that the purpose of this chapter is only to give readers some hints; dealing with big and complex repositories deserves a more prudent and articulated approach.

Installing a Subversion client

To be able to do these experiments, you need a Subversion tool; on Windows, the most widely used is the well-known **TortoiseSVN** (`http://tortoisesvn.net`), which provides command-line tools to both GUI and shell integration.

WOW! eBook
www.wowebook.org

I recommend a full installation of TortoiseSVN, including command-line tools, as we'll need some of them to make experiments.

Working on a Subversion repository using Git

In the first part, we will see the most cautious approach when starting to move away from Subversion, which is to keep the original repository, using Git to fetch and push changes. For the purpose of learning, we will create a local Subversion repository, using both Subversion and Git to access to its contents.

Creating a local Subversion repository

Without the hassle of remote servers, let's create a local Subversion repository as a container for our experiments:

```
$ cd C:\Repos
$ svnadmin create MySvnRepo
```

Nothing more, nothing less; the repository is now ready to be filled with folders and files.

Checking out the Subversion repository with the svn client

At this point, we have a working Subversion repository; we can now check it out in a folder of our choice, which will become our *working copy*; in my case, I will use the `C:\Sources` folder:

```
$ cd C:\Sources\svn
$ svn checkout file:///Repos/MySvnRepo
```

You now have a `MySvnRepo` folder under your `Sources` folder, ready to be filled with your project files; but first, let me remind you of a couple of things.

WOW! eBook
www.wowebook.org

As you may know, a Subversion repository generally has the following subfolders structure:

- `/trunk`, the main folder, where, generally, you have the code under development
- `/tags`, the root folder of the snapshots you usually freeze and leave untouched, for example, `/tags/v1.0`
- `/branches`, the root folder of all the repository branches you will create for features development, for example, `/branches/NewDesign`

Subversion does not provide a command to initialize a repository with this layout (commonly known as *standard layout*), so we have to build it up by hand.
At this point, we can import a skeleton folder that already contains the three subfolders (`/trunk`, `/branches` and `/tags`), with a command like this:

```
$ cd \Sources\svn\MySvnRepo
$ svn import /path/to/some/skeleton/dir
```

Otherwise, we can create folders by hand using the `svn mkdir` command:

```
$ cd \Sources\svn\MySvnRepo
$ svn mkdir trunk
$ svn mkdir tags
$ svn mkdir branches
```

Commit the folders we just created and the repository is ready:

```
svn commit -m "Initial layout"
```

Now, add and commit the first file:

```
$ cd trunk
$ echo "This is a Subversion repo" > readme.txt
$ svn add readme.txt
$ svn commit -m "Readme file"
```

Feel free to add even more files, or import an existing project if you want to replicate a more real situation; to import files in a Subversion repository, you can use the `svn import` command, as we already have seen before:

```
$ svn import \MyProject\Folder
```

Later, we will add a *tag* and a *branch* to verify how Git interacts with them.

WOW! eBook
www.wowebook.org

Cloning a Subversion repository from Git

Git provides a set of tools for cooperating with Subversion; the base command is actually `git svn`; with `git svn`, you can clone Subversion repositories, retrieve and upload changes, and more.

So, wear the Git hat and clone the Subversion repository using the `git svn clone` command:

```
$ cd \Sources\git
$ git svn clone file:///c/Repos/MySvnRepo
```

As you can see, this time I added the root drive letter `c` to the `file:///` path; in Windows, Git pretends that you provide paths starting from the drive letter.

Adding a tag and a branch

Just to have a more realistic situation, I will add a tag and a branch to the Subversion repository; in this manner, we will see how to deal with them while in Git.

So, let's get back to adding a new file:

```
$ cd \Sources\svn\MySvnRepo\trunk
$ echo "This is the first file" > svnFile01.txt
$ svn add svnFile01.txt
$ svn commit -m "Add first file"
```

Then, tag this snapshot of the repository as `v1.0`; as you may know, in Subversion, a tag or a branch are copy of a snapshot:

```
$ svn copy file:///Repos/MySvnRepo/trunk file:///Repos/MySvnRepo/tags/v1.0
-m "Release 1.0"
```

Once we have a tag, we can even create a branch, supposing we want a place to add bug fixes for the release, `v1.0`:

```
$ svn copy file:///Repos/MySvnRepo/trunk
file:///Repos/MySvnRepo/branches/v1.x -m "Maintenance branch for release
1.0"
```


WOW! eBook
www.wowebook.org

Committing a file to Subversion using Git as a client

Now that we have a running clone of our original Subversion repository, we can use Git as it was a Subversion client. So, add a new file and commit it using Git:

```
$ echo "This file comes from Git" >> gitFile01.txt
$ git add gitFile01.txt
$ git commit -m "Add a file using Git"
```

Now, we have to *push* this file to the Subversion server:

```
$ git svn dcommit
```

Well done!

Retrieving new commits from the Subversion server

We can even use Git to fetch changes with the `git svn fetch` command, or directly update the local working copy using `git svn rebase` as a Git counterpart for the `svn update` command:

```
$ git svn rebase
```

Git will fetch new commits from the remote Subversion server, as per a `git pull` command; then, it will rebase them on the top of the branch you are in at the moment. Maybe you are wondering why we are using rebasing instead of merging, like the `git pull` command does by default while dealing with a Subversion remote. Using the `merge` command instead of `rebase` while applying remote commits can be harmful; in the past, Git had some troubles dealing with Subversion `svn:mergeinfo` properties (`http://svnbook.red-bean.com/en/1.6/svn.ref.svn.c.mergeinfo.html`), and even if it supports them (`https://www.git-scm.com/docs/git-svn/2.11.1#git-svn---mergeinfoltmergeinfogt`) rebasing is considered a safer option.

Git and Subversion integration is a wide topic; for other commands and options, I recommend to read the main page `git svn --help`.

Using Git as a Subversion client is not the best we can obtain, but at least it is a way to start using Git even if you cannot abandon Subversion instantly.

WOW! eBook
www.wowebook.org

Using Git with a Subversion repository

Using Git as a client of Subversion can raise some confusions, due to the flexibility of Git as compared to the more rigid way Subversion organizes files.
To be sure to maintain a Subversion-friendly way of work, I recommend you follow some simple rules.

First of all, ensure that your Git `master` branch is related to the `trunk` one in Subversion; as we already said, Subversion users usually organize a repository in this way:

- A `/trunk` folder, the main one
- A `/branches` root folder, where you put all the branches, each one located in a separated subfolder (for example, `/branches/feat-branch`)
- A `/tags` root folder, where you collect all the tags you made (for example, `/tags/v1.0.0`)

To adhere to this layout, you can use the `--stdlayout` option when cloning a Subversion repository:

```
$ git svn clone <url> --stdlayout
```

In this manner, Git will hook the `/trunk` Subversion branch to the Git `master` one, then replicate all the `/branches` and `/tags` branches in your local Git repository, allowing you to work with them in a 1:1 synchronized context.

Migrating a Subversion repository

When possible, it is recommended to completely migrate a Subversion repository to Git; this is quite simple to do and mostly depends on the size of the Subversion repository and the organization.

If the repository follows the standard layout described earlier, a migration takes only a matter of minutes.

WOW! eBook
www.wowebook.org

Retrieving the list of Subversion users

If your Subversion repository has been used by different people, you are probably interested in preserving commit author's names as is even in the new Git repository.
If the `awk` command is available (maybe using Git Bash shell or Cygwin while in Windows), there's a script here that fetches all the users from Subversion logs and appends them to a text file we can use in Git while cloning to perfectly match Subversion users even in Git-converted commits:

```
$ svn log -q | awk -F '|' '/^r/ {sub("^ ", "", $2); sub(" $", "", $2);
print $2" = "$2" <"$2">"}' | sort -u > authors.txt
```

Now, we will use the `authors.txt` file in the next cloning step.

Cloning the Subversion repository

To begin the migration, we have to clone the Subversion repository locally, as we did earlier; I again recommend adding the `--stdlayout` option, to preserve branches and tags, and then to adding the `-A` option to let Git *convert* commit authors while cloning:

```
$ git svn clone <repo-url> --stdlayout --prefix svn/ -A authors.txt
```

In case the Subversion repository has trunk, branches, and tags located in other paths (so not a standard layout), Git provides you with a way to specify them with the `--trunk`, `--branches` and `--tags` options:

```
$ git svn clone <repo-url> --trunk=<trunk-folder> --branches=<branches-
folder> --tags=<tags-folder>
```

When you fire the `clone` command, remember that **this operation can be time consuming**; in a repository with a thousand commits, it is not unusual to wait for one or two quarters of an hour.

WOW! eBook
www.wowebook.org

Preserving ignored files

To preserve the previous ignored files in Subversion, we can append the `svn:ignore` settings to the `.gitignore` file:

```
$ git svn show-ignore >> .gitignore
$ git add .gitignore
$ git commit -m "Convert svn:ignore properties to .gitignore"
```

Pushing to a local bare Git repository

Now that we have a local copy of our repository, we can move it to a brand new Git repository. Here, you can already use a remote repository on your server of choice, even GitHub or BitBucket, but I recommend that you use a local bare repository. We may as well do some other little adjustments (such as renaming tags and branches) before pushing files to a blessed repository. So, first initialize a bare repository in a folder of choice:

```
$ mkdir \Repos\MyGitRepo.git
$ cd \Repos\MyGitRepo.git
$ git init --bare
```

Now, make the default branch to match the Subversion `trunk` branch name:

<pre>**$ git symbolic-ref HEAD refs/heads/trunk**

Then, add a `bare` remote pointing to the bare repository we just created:

```
$ cd \Sources\MySvnRepo
$ git remote add bare file:///C/Repos/MyGitRepo.git
```

Finally, push the local cloned repository to the new bare one:

```
$ git push --all bare
```

We have now a brand new bare repository that is a perfect copy of the original Subversion one. We can now adjust branches and tags to fit the Git usual layout better.

Arrange branches and tags

Now, we can rename branches and tags to obtain a more Git-friendly scenario.

WOW! eBook
www.wowebook.org

Renaming trunk branch to master

Subversion main development branch is `/trunk`, but in Git, you know, we prefer to call the main branch `master`; here's a way to rename it:

```
$ git branch -m trunk master
```

Converting Subversion tags to Git tags

Subversion treats tags as branches; they are all copies of a certain trunk snapshot. In Git, on the contrary, branches, and tags have different significance.
To convert Subversion tags' branches into Git tags, here's a simple script that does the work:

```
$ git for-each-ref --format='%(refname)' refs/heads/tags |
cut -d / -f 4 |
while read ref
do
  git tag "$ref" "refs/heads/tags/$ref";
  git branch -D "tags/$ref";
done
```

Pushing the local repository to a remote

You now have a local bare Git repository ready to be pushed to a remote server; the result of the conversion is a full Git repository, where branches, tags, and commit history have been preserved. The only thing you have to do by hand is to eventually accommodate Git users.

Comparing Git and Subversion commands

In the page that follows, you can find a short and partial recap table, where I try to pair the most common Subversion and Git commands to help Subversion users shift their minds from Subversion to Git quickly.

WOW! eBook
www.wowebook.org

Remember that Subversion and Git behave differently, so maybe comparing commands is not the best thing to do, but for Git newcomers coming from Subversion, this can help match basic Subversion to the Git counterpart while learning:

Subversion	**Git**
Create a repository	

`svnadmin create <repo-name>`
`svnadmin import <project-folder>`
	`git init <repo-name>`
	`git add .`
	`git commit -m "Initial commit"`

Get the whole repository for the first time	
`svn checkout <url>`	`git clone <url>`
Inspecting local changes	
`svn status` `svn diff | less`	`git status` `git diff`
Dealing with files (adding, removing, moving)	

`svn add <file>`
`svn rm <file>`
`svn mv <file>`
	`git add <file>`
	`git rm <file>`
	`git mv <file>`

Committing local changes	
`svn commit -m "<message>"`	`git commit -a -m "<message>"`
Reviewing history	
`svn log | less` `svn blame <file>`	`git log` `git blame <file>`
Branching and Tagging	
`svn copy <source> <branch-name>` `svn copy <source> <tag-name>`	`git branch <branch-name>` `git tag <tag-name>`
Remember: In Subversion, tags and branches represent physical copies of a source branch (the trunk, another branch, or another tag), while a tag is only a "pointer" to a particular commit in Git.	
Merging	
(assuming that the branch was created in revision 42, and you are inside a working copy of trunk) `svn merge -r 42:HEAD <branch>`	`git merge <branch>`

WOW! eBook
www.wowebook.org

Summary

This chapter barely scratches the surface, but I think it can be useful to get a sense of the topic. If you have wide Subversion repositories, you will probably need a better training before beginning to convert them to Git, but for small to medium ones, now you know the fundamentals to start moving away.

The only suggestion I want to share with you is to not be in a hurry; start letting Git cooperate with your Subversion server, reorganize the repository when it is messy, take a lot of backups, and finally, try to convert it; you will probably convert it more than once, as I did, but you will get satisfaction in the end.

In the next chapter, I will share with you some useful resources I found during my career as a Git user.

WOW! eBook
www.wowebook.org

7
Git Resources

This chapter is a collection of resources I built during my experience with Git. I will share some thoughts about GUI tools, web interfaces to Git repositories, and learning resources, hoping they will act as a springboard for a successful Git career.

Git GUI clients

When beginning to learn a new tool, especially such a wide and complex one as Git, it can be useful to take advantage of some GUI tools, to be able to picture commands and patterns in a simpler way to understand.

Git benefits from a wide range of GUI tools, so it's only a matter of choice; I want to tell you right away that there is not one perfect tool, as frequently happens, but there are enough of them to pick the one that best fits your needs.

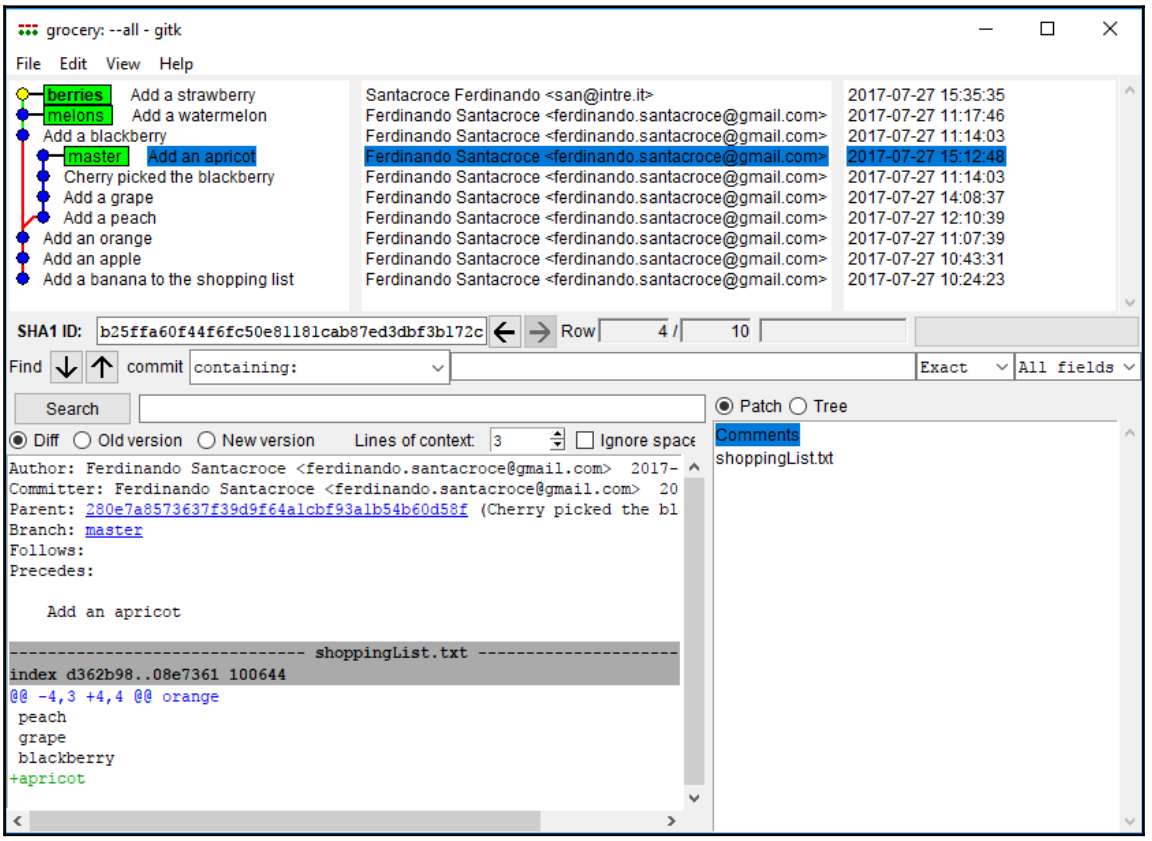
Windows

As a Microsoft .NET developer, I use Windows 99% of the time; in my spare time, I play a little bit with Linux, but in that case, I prefer to use the command line. In this section, you will find tools I use or have used in the past, while in the other platform section I will provide some hints based on the words of other people.

WOW! eBook
www.wowebook.org

Git GUI

Git has an integrated GUI, as we learnt in the previous chapters. It is probably not one of the most eye-catching solutions you will find, but it can be enough. The reason for using it is that it is already installed when you install Git, and that it is well integrated even with the command prompt; so, blaming files, viewing history, or interactive merging can be fired easily (just type `git gui <command>` into your shell):

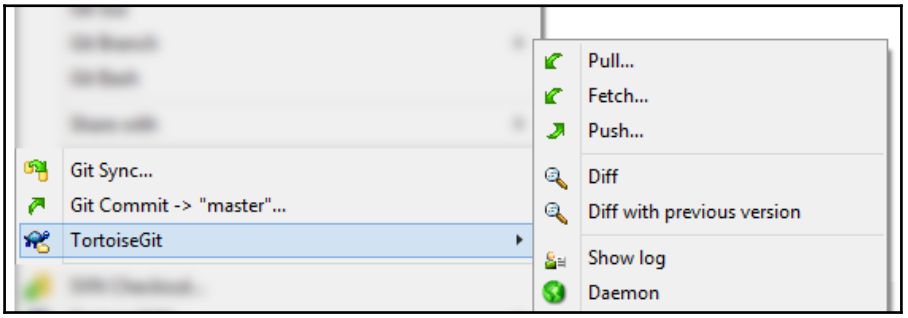


WOW! eBook
www.wowebook.org

TortoiseGit

If you have migrated from Subversion to Git, you probably already heard about *TortoiseSVN*, a well-crafted tool for dealing with Subversion commands directly from Explorer through the right-click shell integration.

TortoiseGit brings Git, instead of Subversion, to the same place; by installing TortoiseGit you will benefit from the same Explorer integration, leaving most used Git commands only a step away from you. Even if I discourage the use of localized versions, TortoiseGit is available in different languages; bear in mind that you need to install Git first as it is not included in the TortoiseGit setup package:

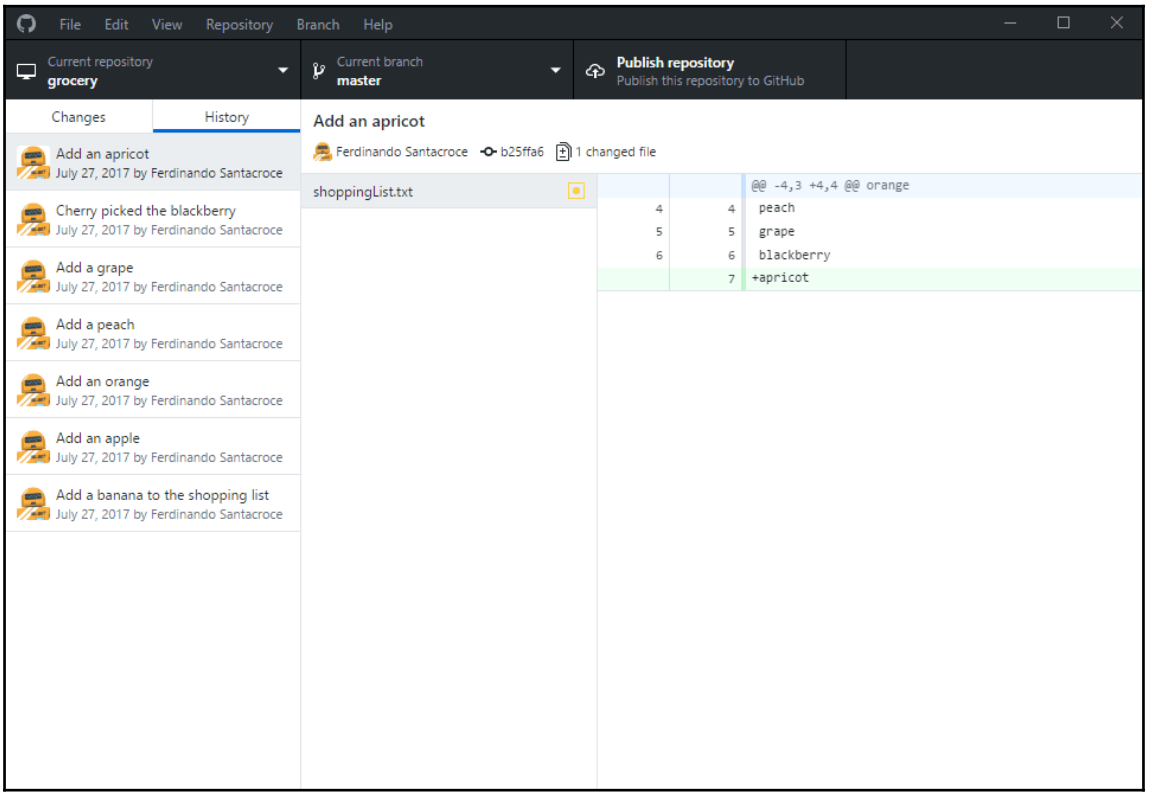


GitHub for Windows

GitHub offers a stylish Modern UI based client. I have to admit that I snubbed it at first, mostly because I was sure that I could only use it for GitHub repositories; instead, you can use it even with other remotes, but it's clear that the client is tailored for GitHub - to use other remotes, you have to edit the `config` file by hand, substituting the GitHub remote with the one you want.

WOW! eBook
www.wowebook.org

If you want a general-purpose client, this is probably not the best tool for you, but if you work mostly on GitHub, chances are it is the best GUI for your needs:

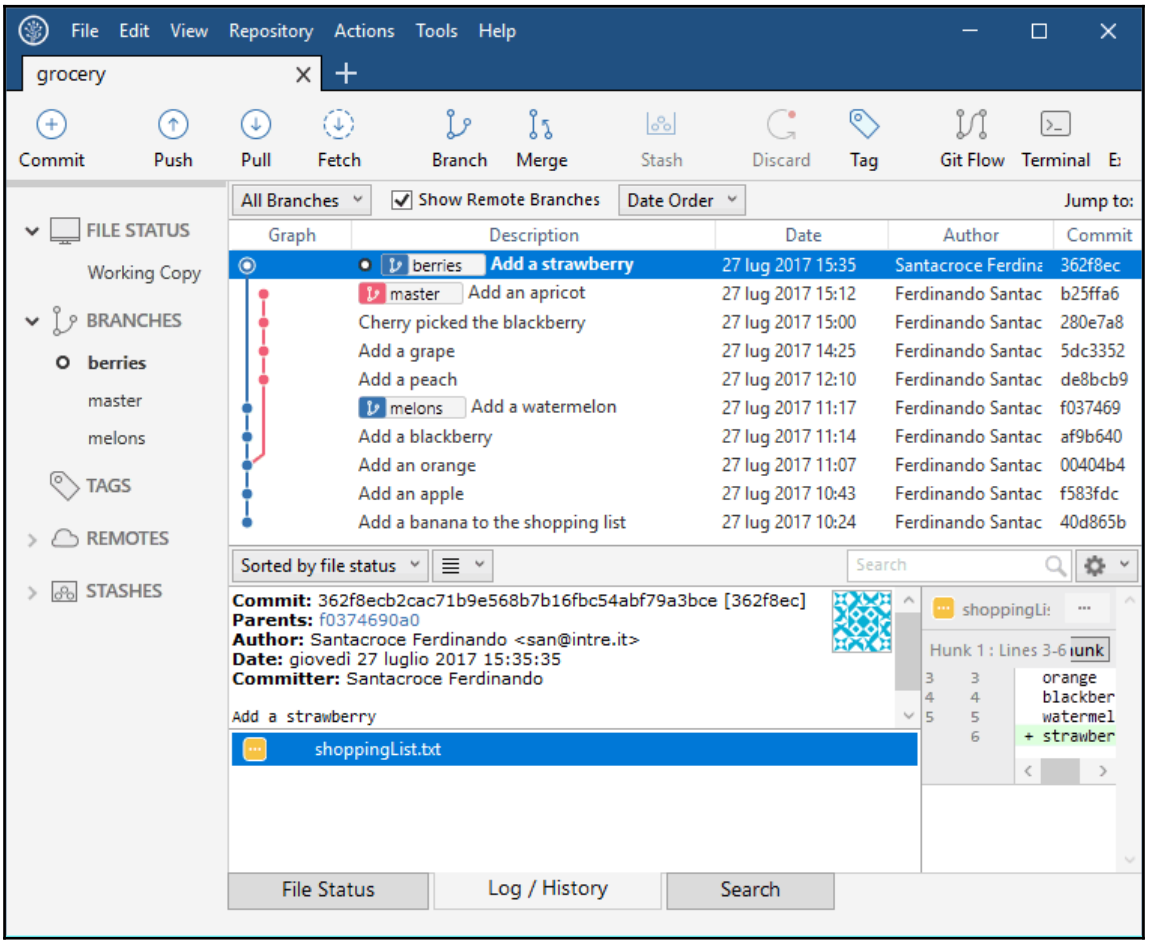


Atlassian SourceTree

This is my favorite client. SourceTree is free like all the other tools; it comes from the mind of **Atlassian**, the well-known company behind *BitBucket* and other popular services like *Jira*, and *Confluence*. SourceTree can handle all kind of remotes, offering facilities (such as remembering passwords) to access most popular services like BitBucket and GitHub.

WOW! eBook
www.wowebook.org

It embeds the GitFlow way of organizing repositories by design, offering a convenient button to initialize a repository with Gitflow branches, and integrating GitFlow commands provided by the author. The most interesting thing I found at first was that you can enable a window where SourceTree shows the equivalent Git command when you use some of the Git commands through the user interface; in this manner, when you are in doubt and you don't remember the right command for the job, you can use SourceTree to accomplish your task and see what commands it uses to get the work done:

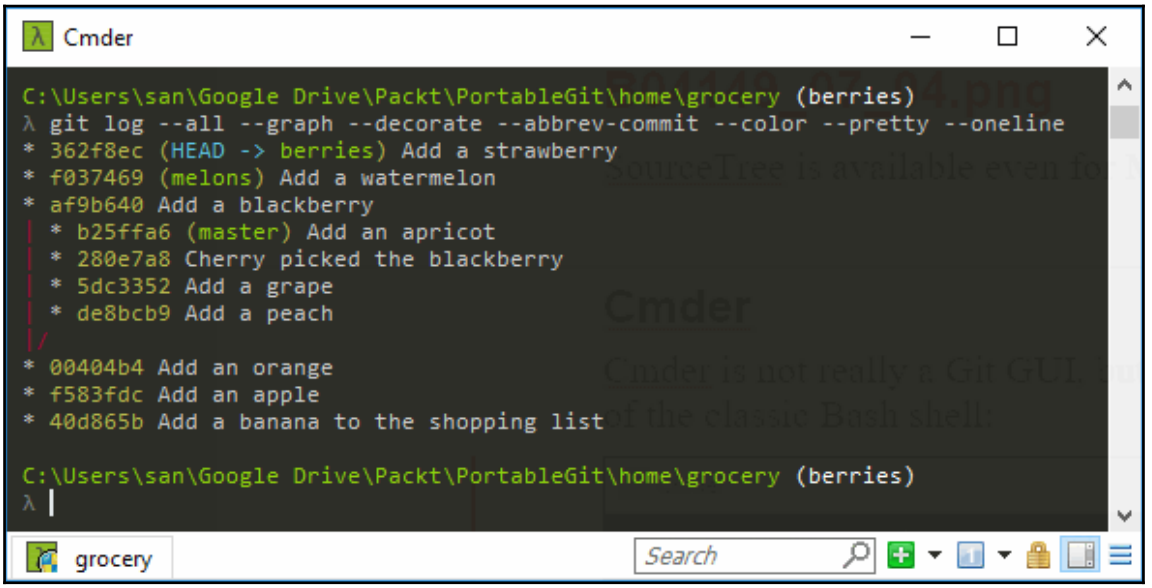


SourceTree is available even for macOS.

WOW! eBook
www.wowebook.org

Cmder

Cmder is not really a Git GUI, but a nicer portable console emulator you can use instead of the classic Bash shell:

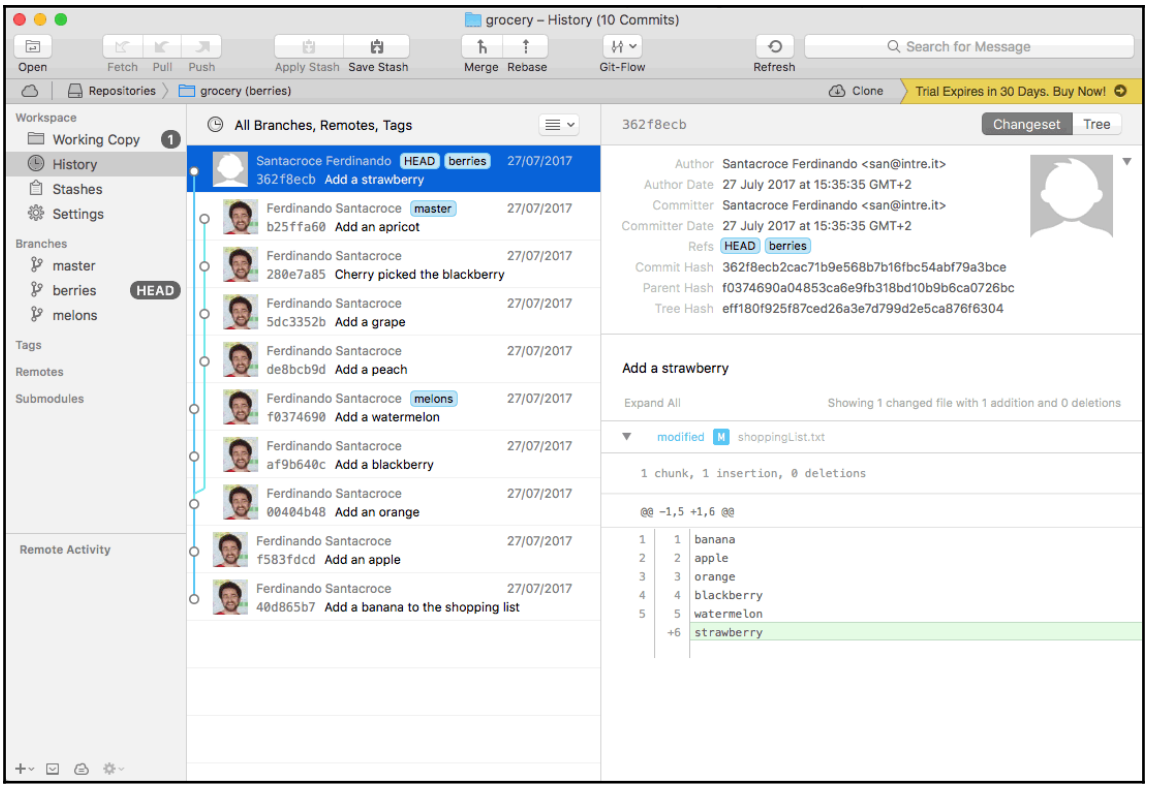


It looks nicer than the original shell; it has multi-tab support and a wide set of configuration options to let you customize it as you prefer, thanks to *ConEmu* and *Clink* projects. Finally, yet importantly, it comes with Git embedded. You can download it from GitHub at `https://github.com/bliker/cmder`.

macOS

As I already said, I have no experience with macOS Git clients; the only information I can share with you is that GitHub offers its client for free even for this operating system, like Atlassian with SourceTree. There is no TortoiseGit for Mac, but I heard about a cool app called Git Tower, please consider giving it a try, as it seems very well crafted:

WOW! eBook
www.wowebook.org



Linux

Linux is the reason for Git, so I think that it is the best place to work with Git. I play with Linux now and then, and I usually use the Bash shell for Git.

For ZSH shell lovers, I suggest looking at `http://ohmyz.sh/`, an interesting open-source project where you can find tons of plugins and themes. When it comes to plugins, there are some that let you enhance your Git experience with this famous alternate console.

At the end, take a look at some Git GUI for Linux by visiting
`http://git-scm.com/download/gui/linux`

WOW! eBook
www.wowebook.org

Building up a personal Git server with web interface

At a company I worked for, I was the first person to use Git for production code; at some point, after months of little trials in my spare time, I took courage and converted all the Subversion repositories where I usually worked alone into Git ones.

Unfortunately, firm IT policies stopped me from using external source code repositories, so no *GitHub* or *BitBucket*; to make things even worse, I also could not obtain a Linux server, and take advantage of great web interfaces like *Gitosis*, *Gitlab*, and so on. So, I started Googling for a solution, and I finally found one that can be useful even for people in the same situation.

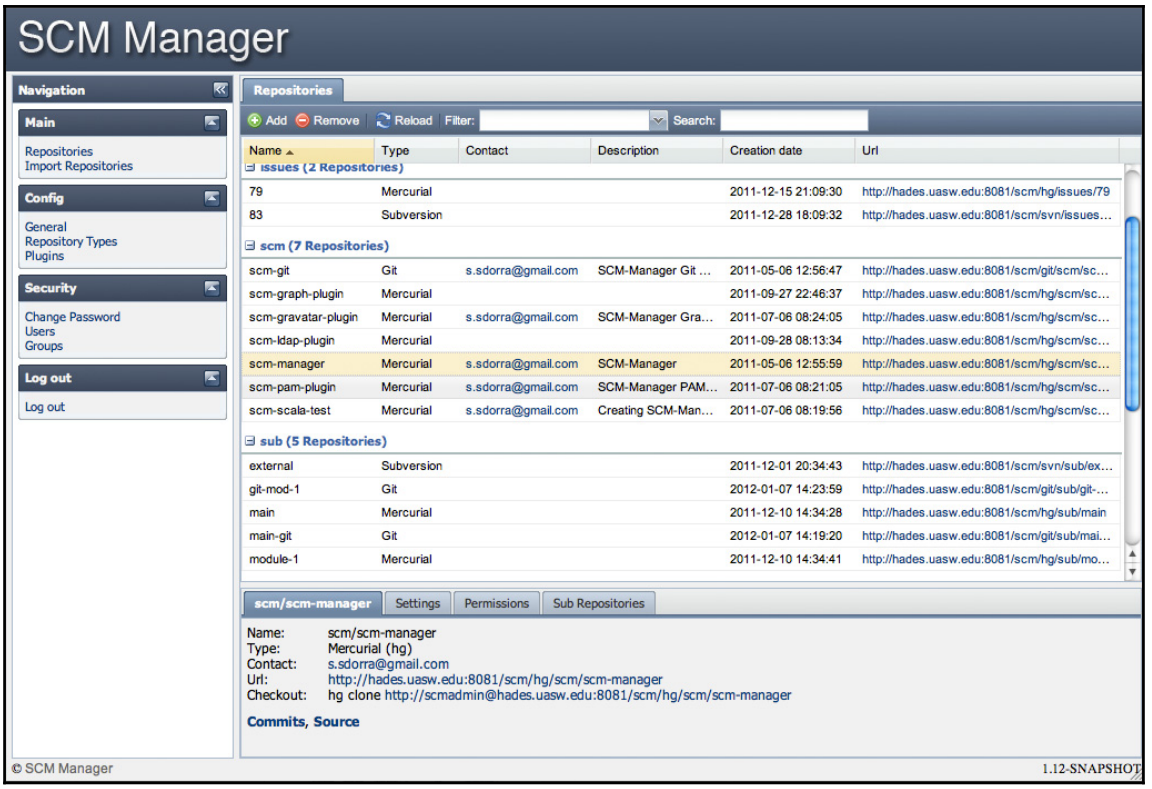
SCM Manager

SCM Manager (`https://www.scm-manager.org/`) is a very easy solution to share your Git repositories in a local Windows network; it offers a standalone solution to install and make it work on top of Apache Web Server directly in Windows. Although it is built in Java, you can make it work even in Linux or Mac.

It can manage Subversion, Git, and Mercurial repositories, allowing you to define users, groups, and so on; it has a good list of plugins too, for other version control systems and other development related tools like Jenkins, Bamboo, and so on. There's also a *Gravatar* plugin and an Active Directory one, to let you and your colleagues use default domain credentials to access your internal repositories.

I have been using this solution for about two years without a hitch, with the exception of some configuration related annoyances during updates, due to my custom path personalization:

WOW! eBook
www.wowebook.org

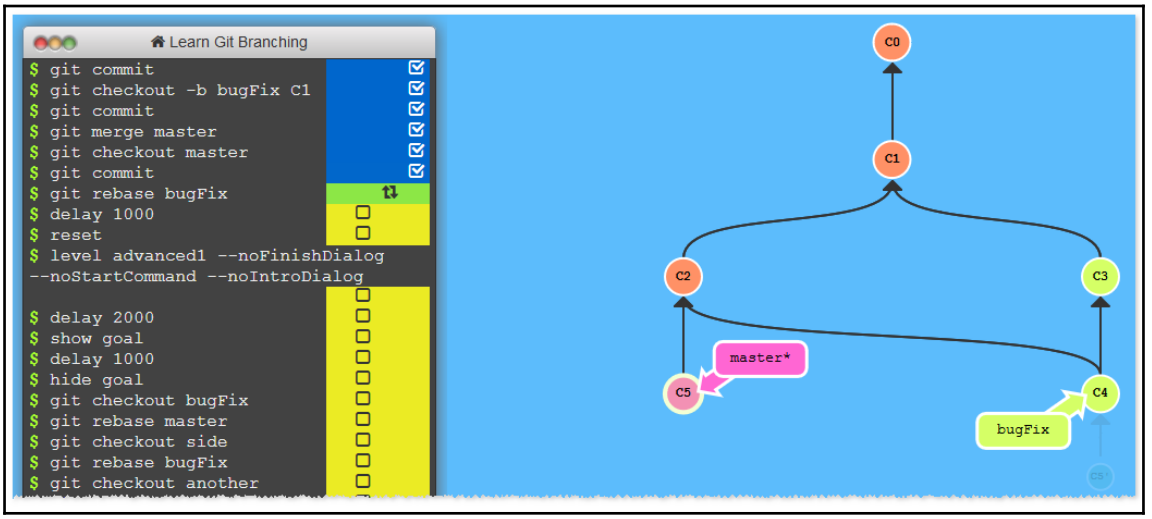


Learning Git in a visual manner

The last thing I'd like to share with readers is a web app I found useful at the very beginning for better understanding the way Git works.

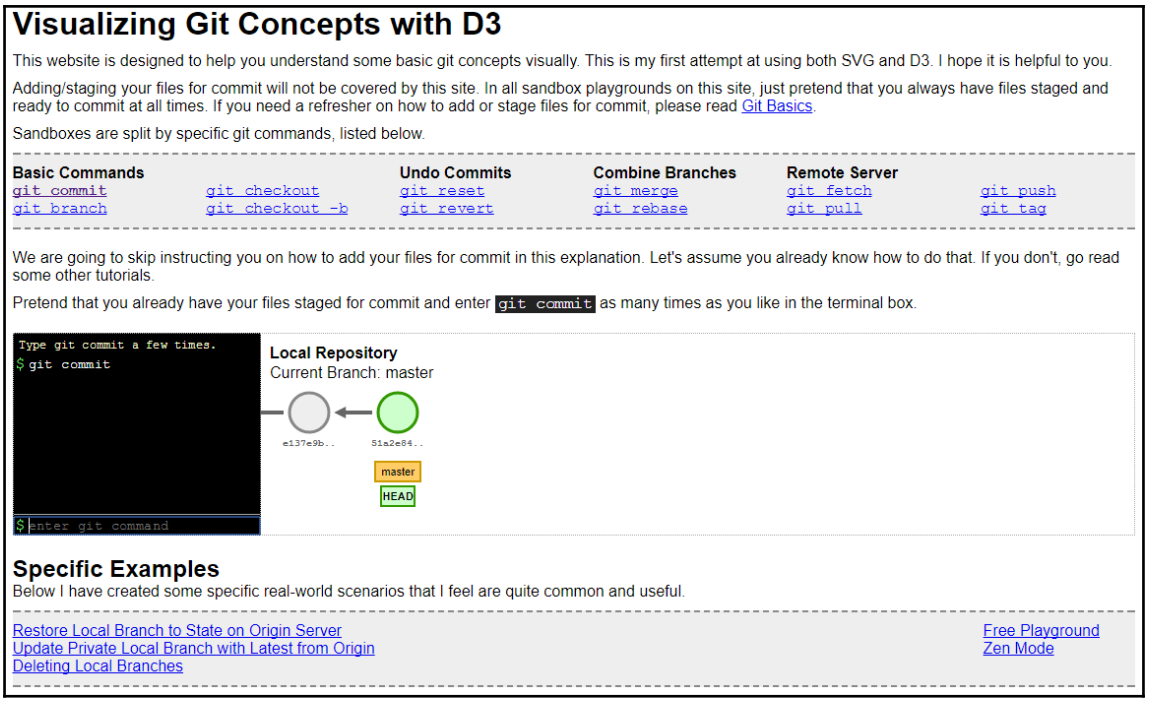
WOW! eBook
www.wowebook.org

Learn Git Branching (`https://learngitbranching.js.org/`) is a tremendously helpful web app that offers you some exercises to help you grow your Git culture. Starting from a basic commit exercise, you learn how to branch, rebase, and so on, but the really cool thing is that on the right of the page, you will see a funny repository graph evolving in real time, following the commands you type in the emulated shell:



Another good resource of this kind is **Visualizing Git Concepts with D3**, where you can grasp all the most important commands visually. Find it at `https://onlywei.github.io/explain-git-with-d3/`:

WOW! eBook
www.wowebook.org



Git on the internet

Finally, I suggest following some resources I usually follow in order to learn new things and get in touch with other smart and funny Git users over the internet.

Git for human beings Google Group

This group is frequented by Git pro users; if you need some help in getting out of difficult situations, the best place to ask for it is at ;`https://groups.google.com/forum/#!forum/git-users`.

WOW! eBook
www.wowebook.org

Git community on Google+

This community is full of people who are happy to share their knowledge with you; most of the coolest things I know about Git have been discovered at `https://plus.google.com/u/0/communities/112688280189071733518`.

Git cheat sheets

The internet has plenty of good cheat sheets about Git; here are my preferred ones:

Git pretty: `http://justinhileman.info/article/git-pretty/`

Hylke Bons Git cheat sheet: `https://github.com/hbons/git-cheat-sheet`

Git Minutes and Thomas Ferris Nicolaisen blog

Thomas is a skilled Git user, and a very kind person. On his blog, you will find many interesting resources, including videos where he talks about Git at local German programming events. More than this, Thomas runs the *Git Minutes* podcast series, where he talks about Git with other people, discussing general purpose topics, tools, opinions, and so on.

Take a look at `www.tfnico.com` and `www.gitminutes.com`.

Online videos

YouTube and other video sharing online platforms have plenty of good and free videos about Git; don't underrate this opportunity while learning Git. Furthermore, *Packt* has a very rich catalog about Git in video format, take a look for more information at `https://www.packtpub.com/video?search=git`

Ferdinando Santacroce's blog

On my personal blog, `jesuswasrasta.com`, I recently started a *Git Pills* series, where I share with readers some things I discovered using Git, quick techniques to get the job done and ways to recover from weird situations.

WOW! eBook
www.wowebook.org

Summary

In this chapter, we looked at some Git GUI clients. Even if I encourage people to understand Git by using shell commands, I have to admit that for most common tasks using a GUI-based tool or the IDE integration facilities makes me feel more comfortable, especially when diffing or reviewing history.

Then we discovered we could obtain a personal Git server with a fancy web interface: the internet has plenty of good pieces of software to achieve this target.

At the end, as my last suggestion, I mentioned some good resources to enhance your Git comprehension; listening to experts and asking them questions is the most effective way to get your work done.

WOW! eBook
www.wowebook.org

Index

A

Agile Movement 167
annotated tag 68, 69
Atlassian SourceTree 201, 202

B

backup repositories
  about 163
  archiving 163
  bundling 163
bare repository
  about 119, 162
  regular repository, converting to 162
Bazaar 5
blobs 40, 41, 42
branch naming
  reference link 52
branches
  fast forwarding 104, 106, 107, 108
  merging 101, 102, 103, 104
  rebasing 96, 100

C

caret
  about 59
  reference link 59
centralized workflows
  about 175
  working with 175, 176
cherry picking 108, 109, 110, 111
cmder
  about 203
  reference link 203
commands
  creating 152
  git difflast 153
  git last 152
  git undo 152
  git unstage 152
  shortcuts 151
commit
  about 36
  author 36
  building 166
  change, description 171
  commands, using 93
  committer, preserving 37
  creation date 36
  date, preserving 37
  exploring 37
  features and tasks, splitting 167, 168
  frequent use 171, 172
  hash 36
  meaningless commits, isolating 172
  message 36, 37
  message, writing prior to code initialization 170
  modifying once 167
  perfect message 172
  plumbing commands 39, 40
  porcelain commands 39, 40
  reassembling 90
  reference link 71
  whole change, including 170
committing 165
Concurrent Version System (CVS) 5
container 40

D

develop branch 178
Distributed Version Control Systems (DVCS) 5

WOW! eBook
www.wowebook.org

E

extreme Programming
  about 184
  URL 184

F

fast forwarding 104, 106, 107, 108
feature branch 179, 180
feature branch workflow 176
file status lifecycle 81
file system caching 17
file
  adding 25, 26
forking 135
forks 135

G

garbage collection 58
Git aliases
  about 151
  commands, creating 152
  commands, shortcut 151
  enhancement, with external commands 153, 154
  removing 154
git blame 159
Git branch 49
Git cheat sheets 209
Git command
  added file, committing 26
  aliasing 154
  committed file, modifying 27, 28, 29, 30
  executing 21, 23
  file, adding 25, 26
  presentations, making 23
  repository, setting up 23, 24, 25
  versus Subversion command 195, 196
git commit 159
Git community, on Google+
  about 209
  reference link 209
Git configuration
  architecture 145
  basic configuration 148
  default editor, defining 150
  dissecting 145
  environment, setting up 148
  files, editing manually 148
  levels 146
  listing 148
  push default 149, 150
  typos autocorrection 149
Git configurations
  reference link 151
Git Credential Manager
  about 17, 128
  reference link 17
git diff output
  reference link 75
Git GUI clients
  about 198
  Linux 204
  Mac OS X 203
  Windows 198
Git objects
  about 32, 33, 34, 35
  blobs 40, 41, 42
  commits 36
  summarizing 47, 48
  trees 40
Git pretty
  reference link 209
Git reference
  about 48
  branch, creating 52, 53
  branches, movable labels 50
  commits, reachability 56, 57, 58, 61
  commits, undoing 56, 57, 58, 61
  detached HEAD 61, 62, 64
  HEAD 53, 54, 55, 56
  labels 49, 50
  reflogs 64, 65, 66
  tags, fixed labels 66, 67, 68
  working 50, 51, 52
Git stash 154
Git storage object model 42, 43, 44
git worktree command
  reference link 6
Git, for Google Group
  about 208

WOW! eBook
www.wowebook.org

reference link 208
git-draw tool
reference link 47
Git
about 5
author blog 209
blogs 209
deltas, avoiding 45, 46
HEAD commit 77
installing 7
installing, on GNU-Linux 7
installing, on macOS 8, 9, 10, 11
installing, on Windows 13, 14, 15, 16
learning, in visual manner 206, 207
on internet 208
online videos 209
reference link 209
Subversion repository, from 190
URL, for downloading 7
used, for working on Subversion repository 188
using, with Subversion repository 192
working internally 32
working tree 77
Gitflow workflow
about 176, 177
conclusion 180
develop branch 178, 179
feature branch 179, 180
hotfixes branches 178
master branch 178
reference link 176
release branch 179
GitHub flow
about 180
branch, deploying after review 182
conclusion 183
descriptive branches off, creating of master 181
master branch, deployable 181
named branches, pushing constantly 181, 182
pull request review, merging after 182
pull request, opening at any time 182
GitHub
about 123
account, setting up 123, 124, 125
local repository, publishing 132, 134
modifications, uploading to remotes 127, 128
origin 130
public server, working with 123
pull request, creating 139, 140, 141, 143, 144
pull request, submitting 138
repository, cloning 125, 126
repository, forking 135, 136, 137
used, for collaboration 135
GNU-Linux
Git, installing on 7

H

hash 36
HEAD commit 70, 77
hotfixes branch 178
Hylke Bons Git cheat sheet
reference link 209

I

interactive rebase 91

K

Kebab Case
about 181
URL 181

L

Large File Storage (LFS)
URL 6
Learn Git Branching
about 207
URL 207
levels, Git configuration
global level 147
repository level 147
system level 147
Linus Torvalds 5
Linux kernel workflow 185
Linux
about 5, 204
reference link 204
local repository
cloning 114, 115
local branch, pushing to remote repository 135

WOW! eBook
www.wowebook.org

local commits, sharing with git push 117
origin 115, 116, 117
publishing, to GitHub 132, 134
remote commits, obtaining with git pull 119, 120, 121, 122
remote, adding 134, 135
local Subversion repository
creating 188

M

Mac OS X 203
macOS
Git, installing 8, 9, 10, 11
markdown markup language
about 125
reference link 125
master branch 178
Mercurial 5
mergeinfo properties
reference link 191
messaging rules, commit
bulleted details lines, adding 173
meaningful subject, writing 173
reference link 174
special messages, for release 174
suggestion 174
useful information, combining 174
Minimalist GNU, for Windows
URL 21
MinTTY
URL 16

O

origin
about 116, 117, 130
branches, tracking 131, 132

P

parent 45
Perforce Helix 5
personal Git server
building, with web interface 205
SCM Manager 205
plumbing commands 39, 40
porcelain commands 39, 40
public server
working, on GitHub 123
pull requests
about 135, 138
creating 139, 141, 143, 144
submitting 138
push 117

R

rebasing
about 89
branches 96, 100
commits, reassembling 90
reference log 65
release branch 179
remotes
about 114
branch, pushing 130
local repository, cloning 114, 115
modifications, uploading 127
working with 113
repository only configurations 147
repository
about 23
cloning 125, 126
setting up 23, 24, 25
right commit
building 166
root commit 45

S

SCM Manager
about 205
URL 205
SHA-1
reference link 36
social coding 135
staged file 82
staging area
about 70
changes, removing 77, 78, 81
file status lifecycle 81
Subversion (SVN) 5
Subversion client
installing 187, 188

WOW! eBook
www.wowebook.org

Subversion command
  version Git command 195, 196
Subversion repository
  branch, adding 190
  branches, and tags arranging 194
  cloning 193
  cloning, from Git 190
  commits, retrieving from server 191
  file, committing with Git as client 191
  Git, using with 192
  ignored files, preserving 194
  local bare Git repository, pushing to 194
  local repository, pushing to remote 195
  local Subversion repository, creating 188
  migrating 192
  Subversion tags, converting to Git tags 195
  tag, adding 190
  trunk branch, renaming to master 195
  users list, retrieving 193
  verifying, with svn client 188, 189
  working, with Git 188
Subversion, versus Git
  reference link 45
symbolic links
  about 18
  reference link 18
system-wide configurations 147

T

tags
  about 66
  annotated tag 68, 69, 70
  creating 66
  fixed labels 66
Team Foundation Server (TFS) 5
tilde
  about 59
  reference link 59
TortoiseGit 200
tracked file 82
trees 40
trunk based development 184

U

unstage 73
untracked file 82
user-wide configurations 147

V

Version Control Systems (VCS) 5
Vim (Vi IMproved) 29
Visualizing Git Concepts with D3
  reference link 207

W

Windows
  about 198
  Atlassian SourceTree 201, 202
  cmder 203
  Git GUI 199
  Git, installing 13, 14, 15, 16
  GitHub 200, 201
  TortoiseGit 200
workflows
  about 174
  adopting 174
  centralized workflows 175
  feature branch workflow 176
  Gitflow workflow 176, 177
  GitHub flow 180
  Linux kernel workflow 185
  trunk based development 184
working directory 55
working tree
  about 70, 77
  checking 83
  Git, checking for overwrites 84
  Git, reset option 85
  resetting 83

Z

ZSH shell
  URL 204

WOW! eBook
www.wowebook.org