Learn by doing: less theory, more results

Git: Version Control for Everyone

The non-coder's guide to everyday version control for increased efficiency and productivity

Beginner's Guide

Ravishankar Somasundaram

[PACKT] PUBLISHING open source community experience distilled

Git: Version Control for Everyone Beginner's Guide

The non-coder's guide to everyday version control for increased efficiency and productivity

Ravishankar Somasundaram

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Git: Version Control for Everyone Beginner's Guide

First published: January 2013

Production Reference: 1170113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-752-2

www.packtpub.com

Cover Image by Asher Wishkerman (a.wishkerman@mpic.de)

Credits

Author

Ravishankar Somasundaram

Reviewers

Giovanni Giorgi

Michael J. Smalley

Acquisition Editor

Robin de Jongh

Lead Technical Editor

Aaron Rosario

Technical Editors

Dipesh Panchal

Veronica Fernandes

Project Coordinator

Leena Purkait

Proofreaders

Maria Gould

Lawrence A. Herman

Indexer

Tejal Soni

Graphics

Sheetal Aute

Valentina D'silva

Aditi Gajjar

Production Coordinator

Arvindkumar Gupta

Cover Work

Arvindkumar Gupta

About the Author

Ravishankar Somasundaram has over 6 years of techno-functional experience in providing solutions to clients across multiple sectors and domains. Being passionate about learning and teaching, he also strongly believes that the sole purpose of learning is to make our minds think in different perspectives, and he facilitates this in his training sessions through a blended learning approach mainly focused on how to "learn to learn".

Junior Scientist: Apart from winning several prizes in science projects in his lower schoolings, he was awarded the title "Junior Scientist" by the committee consisting of people from the Indian Space Research Organization (ISRO) in an Inter school Science Fest for a model display on *Evolution of Airplanes through Aerodynamics*. This is one of his childhood achievements.

His final year college project, aimed at eliminating the scenario of English alone being the medium of programming in all programming languages, which restricts people who don't know English from getting into the IT field and implementing their ideas, was selected and funded by MIT NRCFOSS and considered as a landmark.

By early 2010 he was an official third-party developer of Moodle code, one of the seven people from India and the only one from Tamil Nadu. He shares his knowledge by helping people on the Moodle official forum and on IRC. He has also presented a paper in the 9th International Tamil Internet conference on Moodle: For Enhanced Learning which talks about leveraging Modle's capability to expand user base for one of the oldest language known to mankind - Tamil.

Ravi currently leads Thirdware technology solutions efforts on "Next Generation Mobility" by playing with evolving technologies through its trends - predominantly focusing on Enterprise mobility (MEAP segment) as a Senior Technical Analyst heading the R&D division.

Recently he represented his company at an international conference: "Yugma – Unleashing the Innovation Potential", with an idea that uses Artificial Intelligence to empower the next generation of enterprise mobile solutions.

Acknowledgement

I am thankful to all the people I have met, for they have contributed to my growth either by being an inspiration or personally guiding and pointing me to the right direction when facing challenging situations or throwing critiques continuously, making me recognize there is always an area for improvement in my career and personal life.

Thanks to my clients, employers, and colleagues for providing invaluable opportunities to expand my knowledge and shape my career.

Thanks to all the people who dwell in IRC. Special thanks to Ron for the Mac screenshots.

I dedicate all my accomplishments to my fun loving dad, my ever loving mom, my supportive sister, my understanding wife Madhu, my friends (particularly Sridhar, Ranjith, Ramya, Antano Solar, and Krishnan), and other relatives for all the guidance, faith, hope, love, and support.

Finally, thanks to Packt Publishing for giving me this wonderful opportunity to share my knowledge and thank you for reading!

About the Reviewers

Giovanni Giorgi is an IT professional with a strong cultural background, living in Milan, Italy.

Giovanni was born in 1974; he started playing with Commodore 8-bit computers when he was an 11-year-old child.

During college he studied Latin, Greek during school time, and Turbo Pascal and C programming language as a hobby.

He then attended university in September 1993. After one year he fell in love with open source philosophy.

Giovanni got a Masters degree in Information Technology from DSI of Milan, Italy in 2000. He currently works as an IT Consultant for NTT Data, and has 15 years of experience in banking and finance projects.

He worked with his co-worker on a big project, and he chose Git as the revision control system to coordinate the Pune-based team with the one based in Milan.

He currently write articles on his blog, `http://gioorgi.com`.

Michael J. Smalley is the founder of Smalley Creative LLC, a technology consulting, education, and development organization originating in Philadelphia, PA. He is a professional systems administrator and programmer, as well as the creator and maintainer of the popular Smalley Creative Blog, a regularly updated source of tutorials, news, and solutions of a technical nature. Michael is interested in entrepreneurship, teaching, creative startups, and financial independence, as well as vintage computers, gaming, road bicycling, and musicianship.

When he isn't hunched over a keyboard, he can be found leading the Bucks County Game Creators Meetup, promoting the fact that technology and an opportunity to learn should be accessible to everyone, singing the praises of being an extrovert in a proudly introverted field, and traveling with his wife Kali.

My gratitude goes out to my intelligent, beautiful, and creative wife Kali M. Whyte-Smalley for always believing in my bold, arguably crazy interests and endeavors, and for encouraging my enthusiasm in bringing these ideas to life and goals to fruition. Thank you to my elegant and selfless mother, Lisa A. Smalley, for encouraging me to stay well-rounded, for showing me the true meaning of fortitude, and for encouraging me to surround myself with people who exemplify these qualities. Thank you to my father Michael G. Smalley for his dedication to our family unit through hard work and commitment, and for recognizing my constant questioning of the world around me not as naivety, but as a desire for truth and knowledge. Thank you to my brother David P. Smalley for continuing to stand as an unyielding ally. Thank you to my charming Persian cat, Desmond, for unknowingly demonstrating that even lifelong dog people can unwillingly become cat people. Finally, thank you to Packt Publishing for giving me this opportunity to share my knowledge, and thank you for reading!

"All men dream: but not equally. Those who dream by night in the dusty recesses of their minds wake in the day to find that it was vanity: but the dreamers of the day are dangerous men, for they may act their dreams with open eyes, to make it possible." - T.E. Lawrence, *Seven Pillars of Wisdom*.

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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.com` and 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.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

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

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

This book is a non-coder's guide to get a kick start in using the Git version control system on a daily basis to improve their efficiency and productivity when dealing with all forms of electronic data.

With step-by-step examples and illustrative screenshots, you will be guided through the process of installing, configuring, and mastering the concepts needed to version control your data with the help of the best in class tool, Git.

Concepts in every chapter are explained through simple, day-to-day examples and interesting analogies which makes the learning itself an experience to cherish.

Specifically catered to address the needs of an audience from diverse backgrounds using multiple operating systems such as Microsoft Windows, Linux, and Mac OS, all illustrations are explained using both Graphical User Interface (GUI) and Command-Line Interface (CLI) modes.

The final chapter is dedicated to readers who want to understand the behind the scenes operations of Git's functions which they performed in all other chapters, in simple terms. This will also interest people who have been using Git prior to this book.

By the end of the book, you will not only gain theoretical knowledge but also a hands-on practical understanding and experience about the concepts which are needed to make a difference in the way you deal with digital files.

This book can also be used as a reference or to relearn the concepts that have been discussed in each chapter. It has illustrative examples, wherever necessary, to make sure it is easy to follow.

What this book covers

Chapter 1, *Breathe Easy – Introduction to Version Control System*, introduces the concept of version controlled systems, its necessity along with its evolution, and more importantly why Git is considered to be the best in class.

Chapter 2, *Welcome Aboard – Installing Git*, guides you through the installation of Git, specific to your operating system.

Chapter 3, *Polishing Your Routine – How to Increase Everyday Productivity*, teaches you five basic, important concepts (initiate your repository, add your files to it, start versioning by committing them, travel back using checkout, and reset whenever it is needed) which is all you need to get started with versioning your files using Git.

Chapter 4, *Split the Load – Distributed Working with Git*, teaches you the essentials of collaborative development by sharing content with others over multiple mediums such as the Internet and intranet and explores various methods to continue the work from different locations with different people.

Chapter 5, *Be a Puppet Master – Learn Fancy Features to Control Git's Functions*, teaches you a few tips and tricks which can be implemented in various situations to change Git's usual behavior pertaining to the functions which we have come across in earlier chapters.

Chapter 6, *Unleash the Beast – Git on Text-based Files*, exposes to you a new feature called merging that is considered to be one of the hallmarks of Git. You will learn how to merge content and solve conflicts caused by such merges.

Chapter 7, *Parallel Dimensions – Branching with Git*, introduces one of Git's most applauded features, the concept called branching, its importance, and the ways it can be implemented to transform your mode of work.

Chapter 8, *Behind the Scenes – Basis of Git Basics*, takes a deep dive into Git's internals and puts it in simple terms. You will get to know the underlying operations which Git performed when you executed the various Git commands in all of the earlier chapters.

What you need for this book

The basic requirement for learning the concepts in this book will be an administrative (or at least installation) access to a machine running a Windows, Linux, or Mac operating system. And occasionally, Internet connectivity for the said machine.

Apart from this, it's good to have your favorite text editor along with a zip utility (your machine will have one by default) and an office package such as MS Office, OpenOffice, and LibreOffice to create word documents.

Who this book is for

This book is for any one who is computer literate and wants to maintain multiple versions of his/her files in an efficient manner and travel back in time to visit such different versions without juggling numerous files along with their confusing names stored at different locations.

This book is even for people who have prior experience with Git or any other version control system, as they will pick up interesting points from the final chapter which is focused on Git's internals put in simple terms.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Have a go hero – heading

These set practical challenges and give you ideas for experimenting with what you have learned.

You will also 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.

Code words in text are shown as follows: "Create a directory called BCT on your desktop."

A block of code is set as follows:

```
[remote "capsource"]
url = https://github.com/cappuccino/cappuccino
fetch = +refs/heads/*:refs/remotes/capsource/*
```

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

```
git add .
git commit –m 'Unfinished list of marketing team'
git checkout master
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Add** button."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through 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 on `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.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

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 would 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/support`, 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.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Breathe Easy – Introduction to Version Control System

We might have wondered numerous times if there is a way we could travel back in time to change what had happened in life. Well, as fictional as it sounds, you are going to learn a method to perform this travel when it comes to digitized files! Yes, you read that right; this chapter is going to introduce you to a system that makes this possible. We will begin by giving you a solid conceptual understanding about version control with Git.

This chapter will answer the following questions:

- What is a version control system?
- Where do you need one?
- How did they evolve?
- Why is Git your best bet?

By the end of this chapter you would be able to visualize how you can better handle situations where frequent changes happen on different parts of your digitized files. So let's get started right away!

Do you need help

I learned to play computer games even before learning to switch a computer on or off, for which I sought an adult's help. The early computer games, which put us in awe even at that time, had a few frustrating moments when they wouldn't allow us to save our progress. Even if they had a save option it was a single save slot at a time, which meant you could only save your progress at the cost of your earlier save. This was a shame, because your previous save might have been at an exceptionally fun part of the game that you would like to preserve now and revisit later some day, or even worse, your present save might have been at an unwinnable situation that you want to undo.

Computer games have evolved from this state while our way of working with digitized files remains the same. Options like *undo* and *redo* help us momentarily when the file that we are working with is still open, but fail to go beyond that. You cannot just open a file and start undoing the changes that you have made before your last save to get back to an older state.

There are also several situations where we would like to maintain multiple versions of the same file. Even the most widely used way of maintaining multiple versions of a file by naming the new files sequentially, for example, `Inventory_product_2011-09-21.doc`, `System_requirement_specification_v6.xls`, and so on, become a pain as the number of versions increases because of the huge volume of the number of files that has to be maintained.

Now if you have experienced or thought about any of these situations and wondered whether there is a way to handle this better, you will be rejoicing at the end of this chapter. This is where a **version control system** (**VCS**) comes into play.

What is a version control system

A system capable of recording the changes made to a file or a set of files over a time period in such a way that it allows us to get back in time from the future to recall a specific version of that file, is called a version control system.

To give you a more formal explanation, a version control system is a software package that when initiated will monitor your files for changes and allow you to tag the changes at different levels so that you can revisit those tagged stages whenever needed.

When installed and initiated, this version control system creates a local directory at the same place where your files reside, which it uses to manage the entire history of the changes made to your files.

Why you need a version control system

Try answering the following questions with regards to your present system setup:

- Can you maintain multiple versions of the same file under the same name, thus avoiding cluttering of files with small differences in their names mentioning their versions?
- Do you have any means of marking a specific portion of your content in the file/files that you might need in future before changing them for present needs?
- Are you satisfied with the existing scenario where your only failsafe plan for getting back your content is copying and pasting the file or group of files in a separate folder that contains the word "backup" in its name? And updating it regularly?

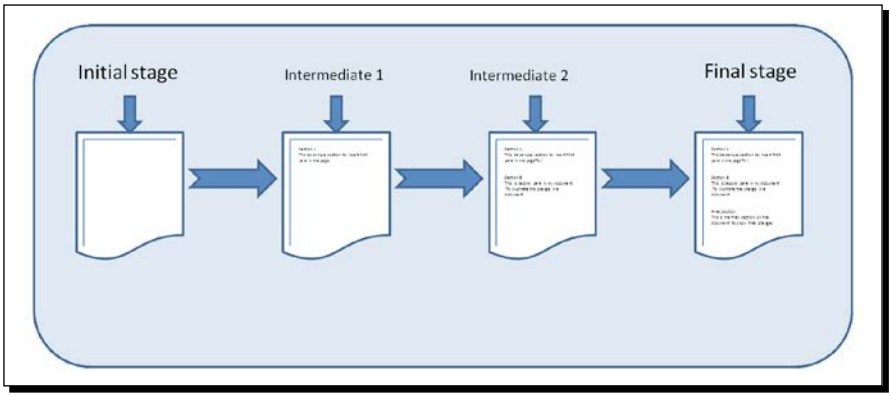
If your answer to any of these questions is a big *no*, then that's exactly the reason you would probably need a version control system and this book.

If your answers to these questions are *yes*, it means that you have probably managed to find roundabout ways to solve these issues. Simple measures could include creating a restoration point in latest releases of Windows, which internally stores versions of all documents such as your Word, Excel, or PowerPoint files present at that point in time as a part of creating your restoration point.

As varied as the potential solutions may be, allow me tell you that version control systems will amaze you with their power, simplicity, and ease of use. They will allow you to achieve much better results with less than half the time and effort that you would normally put into your present solutions.

By using a version control system you have the power to play with the flow of changes happening to your documents. Whenever you have to make considerable amount of changes to the existing content you can mark those changes as a stage (with a tag) that you can revisit later; this serves as a failsafe mechanism just in case things didn't go as per your plan and you want to revert the content of the document back to a particular older state.

The following figures demonstrate the flow of content creation with and without a version control system:

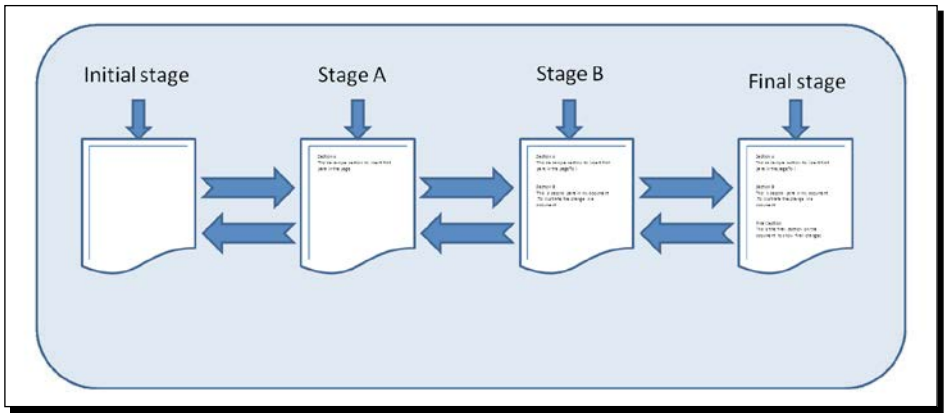


The previous figure shows you the flow of a content creation matrix at different times spread across sessions. As you can observe in a regular constructive context the flow is from *left to right*, meaning you progress with content creation one way when it comes to building content on different time periods. In this flow you cannot go back to a previous phase from where you can take a different direction altogether when compared to what you have already taken.

To explain it using our flow diagram, you cannot go back to any of the intermediate stages from the final stage to write an entirely different third paragraph to serve a new purpose without any data loss. (You cannot use the undo feature as the content was built across time periods, and you cannot undo something once you have saved and closed your file.)

Presently we achieve this using the "save as" option, giving the file a different name, deleting the third paragraph, and starting to write a new one.

In contrast, when you use a version control system, it's a **multi-directional free flow context**. You mark each and every change you consider important as a new stage and proceed with your content creation. This allows you to get back to any earlier stages that you have created, without any data loss.



And the best part is you are not limited by the following:

- Number of hops
- Number of stages between the hops
- Direction of the hop

This means we can, without any concern, jump to and fro between and across stages in any direction without any data loss. Now doesn't that sound like the need of the hour?

Types of version control systems

There are three types of version control systems available. These are classified based on their **mode of operation**:

- Local version control system
- Centralized version control system
- Distributed version control system

Let's quickly go through the history in brief.

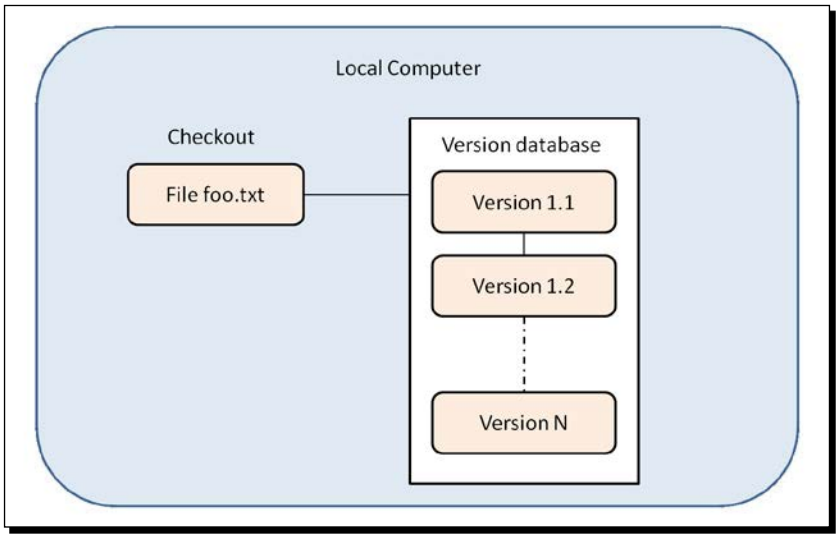
Local version control system

After understanding that maintaining multiple versions of files by just following a file naming convention is highly error prone, a **local version control system** was the first successful attempt to solve this issue.

Revision control system (**RCS**) was one of the most popular version control systems in this cadre.

This tool basically works by keeping patch sets (that is, the difference between the file's content at progressive stages) using a special format in the version tracker that is stored in your local hard disk.

It can then recreate the file's contents exactly at any given point in time by adding up all the relevant patches in order and "checking it out" (reproducing the content to the user's workplace).



Tidbits

Version tracker is nothing but a file with its own file format holding structured content format through which it can perform its functionalities.

When a file is put under RCS it creates a version tracker entry that will hold details such as RCS configuration for that particular file at the top followed by version number, date, time, author, state, branch, and a link to the next stage followed by contents of the file in a specially formatted manner. After this process your file is deleted!

Retrieval of the file, as stated previously, is done through reconstruction of patches.

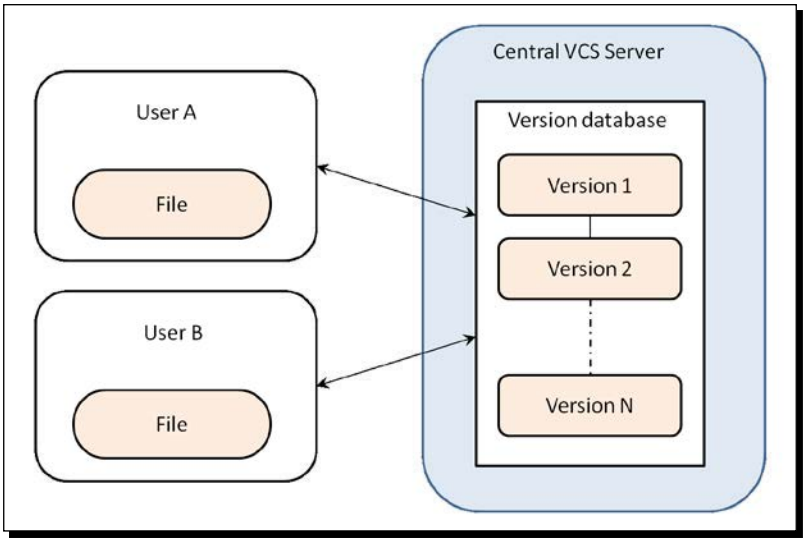
Centralized version control system

As with any other software package or concept, as the requirements kept evolving, users felt that local version control systems limited their activities.

People were not able to work collaboratively on the same project, as the files with their versions are stored in somebody's local computer and were not accessible to other people working on the same files.

So how do you solve this problem?

It is solved by keeping the files in a common place (server) that everybody has access to from their local machines (clients). Hence, the birth of a **centralized version control system**.



Whenever people want to edit single or multiple files only the last version of the files are retrieved.

This setup not only provides access to the files for people who require them but also offers visibility on what other people are working towards.

As the files are stored in one single location from which everybody needs to share the files, any changes made to the files are automatically shared with other individuals as well.

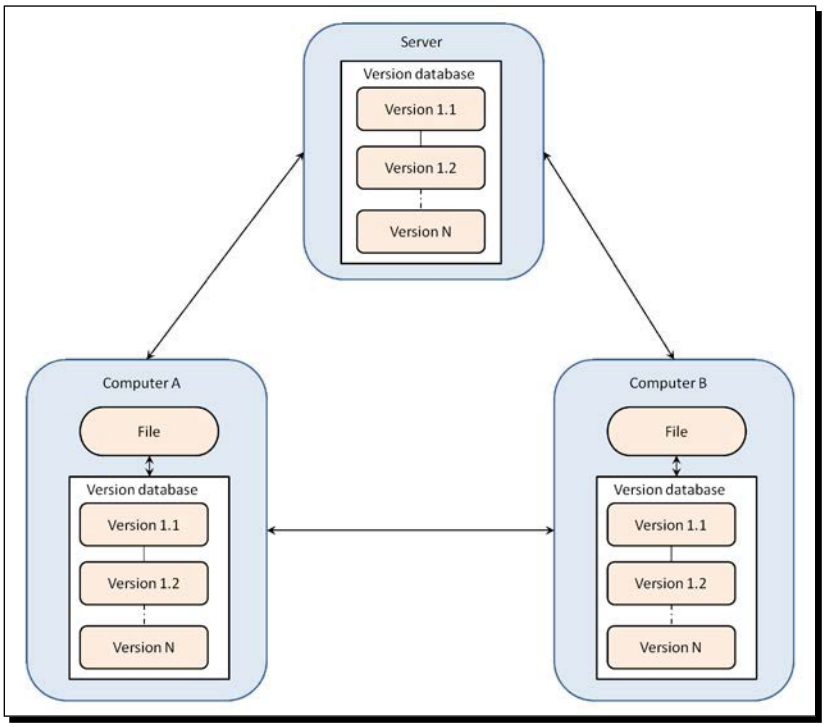
Distributed version control system

Whenever you bet big time on one single unit, the probability of losing is also high. Yes, there is a high degree of risk involved in using a centralized version control system because the users only have the last version of files in their system for working purposes; there is a chance you might ultimately lose the entire history of your files if the server gets corrupt and if you don't have fail-safe procedures implemented.

Now people get confused. You risk everything when you store your entire history in one single place using a centralized version control concept; on the contrary you lose the power to work collaboratively when you use local version control.

So what do you do?

Exactly! You combine the advantages of both and make a **hybrid system**. And that's one of the key reasons why **distributed version control systems** came into the picture.



Distributed version control systems have the advantages of local version control systems, such as the following:

- Making local changes without any concern of full time connectivity to the server
- Not relying on a single copy of files stored in the server

These are combined with the advantages of centralized version control systems, such as the following:

- Reusability of work
- Collaborative working, not relying on history stored on individual machines

A distributed version control system is designed to act both ways. It stores the entire history of the file/files on each and every machine locally and also syncs the local changes made by the user back to the server whenever required so that the changes can be shared with others providing a collaborative working environment.

There are several other advantages in terms of performance, ease of use, and administration. It's a general saying that "you name anything that a centralized version control system can perform; a distributed version control system can handle the same thing and perform much better".

Falling for Git

We came across different types of version control systems in the previous section, from which we clearly understood that a distributed version control system is what will make our lives easy, safe, and secure.

Now, there are lots of distributed systems available in the market, so which one to choose?

Git is a relatively new software package (April 7, 2005 with its first prototype) that was designed from the ground up to avoid flaws that existed in many other version control systems.

Linus Torvalds, the man who gave us the Linux kernel, is the proud initiator of this project as well. The very architecture of GIT is tailored for better speed, performance, flexibility, and usability. When I first heard the previous sentence I had the same thought that you have in mind right now: "It talks the talk; can it walk the walk?"

As a matter of fact there are several live case studies; I got convinced when I saw Git handling the complex Linux kernel source code so gracefully.

For those of you who don't have any idea about Linux kernel or why it's tagged complex, just think about approximately 9 million lines of content spread across 25,000 files subjected to all kinds of content manipulation, travelling back and forth, numerous times every day by several hundred developers across the world. And still the response time of Git's operations are in seconds.

Why they trust Git for such challenging tasks and how Git meets their expectations is through the following:

- Atomicity
- Performance
- Security

Atomicity

Atomicity is nothing but a property of an operation that appears to occur at a single instant between its invocation and its response.

As an example let's take a banking system. When you transfer money from your account to another account, the operation is either completed fully or rejected meaning either the money gets debited from your account and gets credited to the recipient's account or the entire operation gets dropped and no money is debited from your account in the first place.

These systems avoid partial completions such as the amount getting debited from your account but not getting credited to recipient's account.

Another example would be a seat reservation system in which the following are the possible states:

- Both pay and reserve a seat
- Neither pay nor reserve a seat

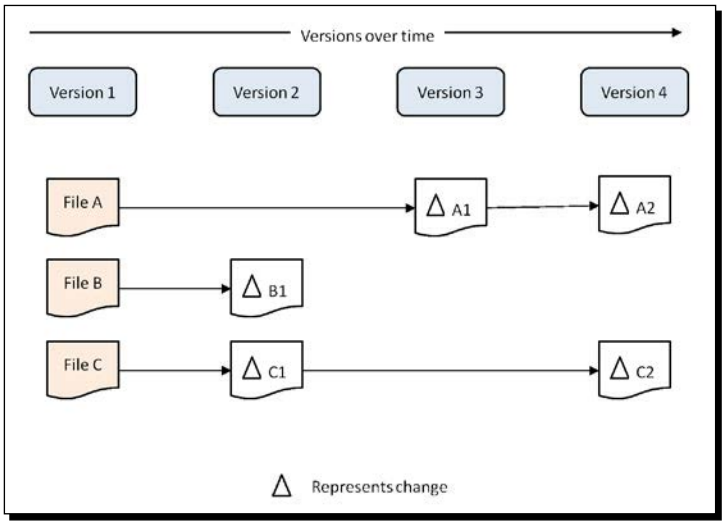
Git creators understood the value of our data, and implemented the same when handling content with Git. It ensures there is no data loss or version mismatch happening due to partial operations, which increases reliability.

Performance

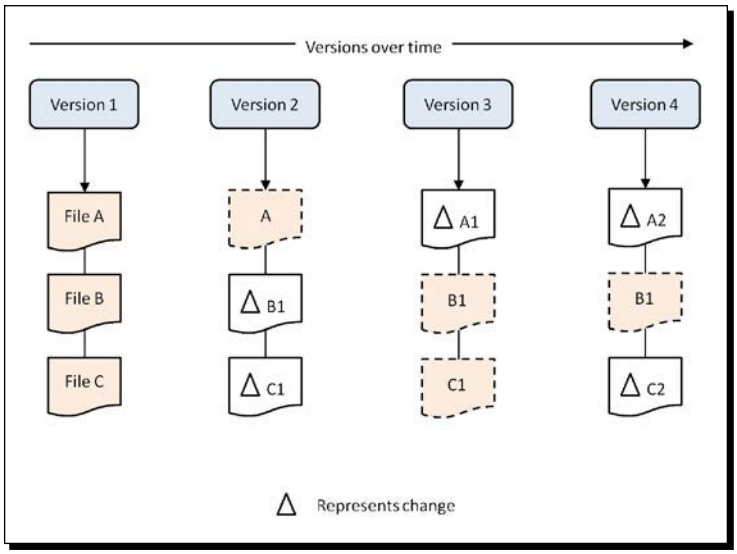
No matter how good a car's interiors are, if it isn't quick enough, it isn't fit enough for racing against time. Git is proven to be manyfold faster than its competitors.

Even when handling several million files, an operation performed using Git takes only seconds to complete. One of the main reasons for this would be the way Git handles your files. Conceptually most other systems (CVS, Subversion, Perforce, Bazaar, and so on) look at your data as a set of files and changes made to each of them as the version proceeds.

The following is a pictorial representation of how other systems handle files and their versions:



In contrast, Git sees a relation between your files and works upon it. It takes a **snapshot** of the entire set of files instead of storing the difference between versions of each file; this contributes to the lightning speed of Git in certain operations like reverting your file's contents to earlier versions (which we will see in later chapters). Each time a version is created, a snapshot is taken. This doesn't mean that Git stores multiple replicas of your files; if Git finds that there is no change in any of your files' content, just a reference to that file that points to the previous snapshot is stored in the new snapshot, as shown in the following figure:



The best part is that Git tries to occupy as little space (again, several times less when compared to other version control systems) as possible to maintain version histories of your files. A live case study in handling the source code of Mozilla Firefox published by Keith P. (`http://keithp.com/blogs/Repository_Formats_Matter/`) showed how effectively version control systems utilize space when it comes to maintaining the history of your files.

Mozilla's CVS repository was 2.7 GB in size; when imported to Subversion the size grew to 8.2 GB, and when put under Git the size got shrunk to 450 MB. For a source code of size 350 MB it's fairly nice to have the whole project history (from 1998) with just 100 MB more space.

Security

When you use Git, you can be sure that no one is tampering with your files' content. Everything that goes into Git is check-summed using an SHA-1 hash before it's stored, and after that it is referred to using that checksum.

This means it's impossible to change the contents of any file or directory without Git knowing about it. The SHA-1 hash used here is a collection of 40 hexadecimal characters (a-f and 0-9) which is generated based on the contents of a file or directory structure. The following is an example of a hash:

9e79e3e9dd9672b37ac9412e9a926714306551fe

For those of you who would like to know more about it, you can hear from the very creator, Linus Torvalds, who gives a presentation at Google's tech talk event.

Summary

In this chapter we discussed problems faced in our daily lives when it comes to digitized files, followed by exactly addressing those issues and assuring a solution to those challenging problems with the help of a version control system.

We also quickly went through the evolution of version control systems and obtained a solid understanding of how a distributed version control system can make our lives easy.

Then we got introduced to the best-in-class distributed version control system, Git, and discussed a few reasons for such a claim with some interesting statistics and case studies. This was followed by a view on a few of its internals such as atomicity, performance, and security.

Now that we've done enough ground work, we're ready to get our copy of Git and get it running in our system, which is the topic of the next chapter.

2
Welcome Aboard – Installing Git

In the previous chapter we got enlightened on how a version control system can change the way we face everyday situations when it comes to digitized files, followed by the evolution of version control systems.

We also understood why Git is considered the best in class and how it can serve our purposes.

In this chapter we will get to know how you can install and configure Git. We shall cover the following:

- Choosing the Git type that suits your environment
- Installing Git

Choosing your type – download your OS specific package

Like many other tools, Git can be downloaded from the Internet and the best part is it's free, thanks to the open source community. Git can be installed on a variety of operating systems such as Mac OS X, Windows, Linux, and Solaris. We will consider the top three operating systems in terms of user base in discussing our concepts:

- Windows
- Linux
- Mac OS X

To start, you need to download your operating system specific Git package; we can get the list of downloadable packages from `http://git-scm.com/downloads`. Go ahead and download the latest stable release relevant to your OS, which you can find on the website, as shown in the following screenshot (Version 1.8.0.2 was the latest stable release at the time of writing this chapter):

For those of you Linux and Mac users who enjoy using a **command-line interface** (**CLI**) mode to get things done, there's no need to go through the following procedures of downloading and installing via GUI. You can use your OS specific installer to get things done. For example, if you are using a Debian based Linux OS, `apt-get install git-core` is enough for installing Git on your system, whereas if you are using a Mac you can either use Apple's Xcode IDE, which is an Apple approved method to install things, or Macports or Fink to install Git for you.

Windows

If you are trying to download from a Windows machine, the website detects that and automatically provides the **Download for Windows** button. Upon clicking that, you will be prompted to save the setup file. Choose your preferred location to save the file and you are good to go.

Linux

There are always multiple ways of doing things in Unix operating systems. We will take the easiest possible route which everybody can follow. So unless you are a person who loves installing packages through compiling, there is no need for downloading them from this website. You can move to the *Installation* section directly from here.

For people who want to install Git by getting the source code and compiling it, perform the following steps:

- Click on the **Git source repository** link; you will be taken to a page that lists the contents of the source code package. Click on the button called **Zip**, which will prompt you with the file download. Save it in your preferred location.
- You can then go through the regular `unzip, configure`, `make`, and `make install` commands, which we won't discuss here.

Mac

If you are trying to download from a Mac machine, the website detects that and automatically provides the **Download for Mac** button. Upon clicking that, you will be prompted to save the setup file. Choose your preferred location to save the file and you are good to go.

The download file usually follows a naming convention like Git-latest.stable_release_version_here-min-required-os-info.dmg, for example, **git-1.8.0.2-3-intel-universal-snow-leopard.dmg**.

You can use the same 1.8.0.2 installer for any version of Mac OS on or above Snow Leopard. For Leopard users there are lower versions of Git available, which you can get from `http://code.google.com/p/git-osx-installer/downloads/list`.

Installation

Now that you have got your own copy of Git, let's proceed to the installation phase. Let's go over the installation process for different operating systems one at a time.

As with any other installation, you need administrative rights to install this software.

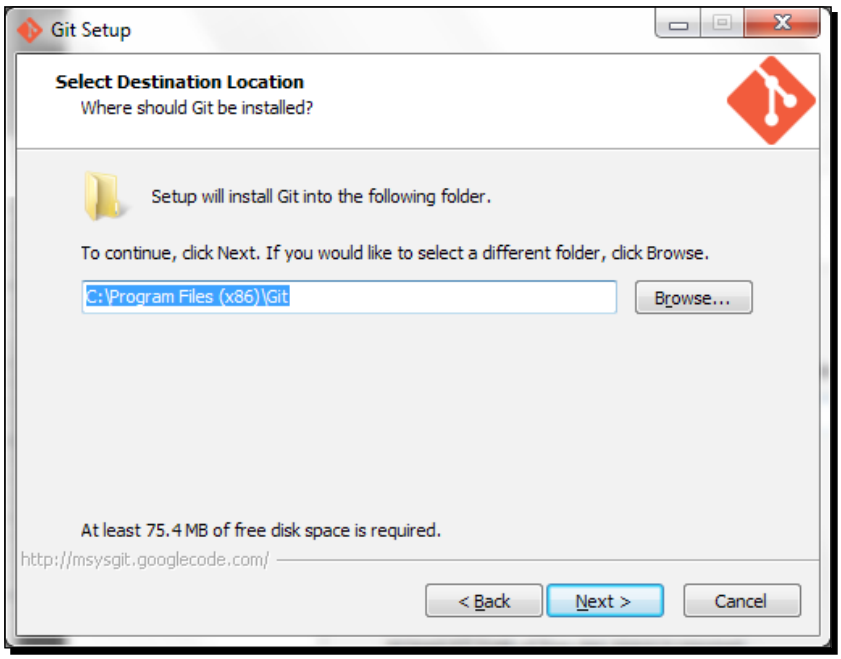
Time for action – installing Git on Windows

Perform the following steps:

1. Double-click on the downloaded setup file to get started with the installation.
2. The first and second steps of the installation process are self-explanatory. It first greets you with a welcome message and informs you about the "safe to follow" procedures before installation, that is, to close all other opened applications before continuing (just to avoid the remote chance of any shared dll/exe being overridden or a simple case of your system running out of memory, which is required in large amounts whenever you perform an installation). Then it shows you the details about the GNU public license Version 2 by which our Git is governed.

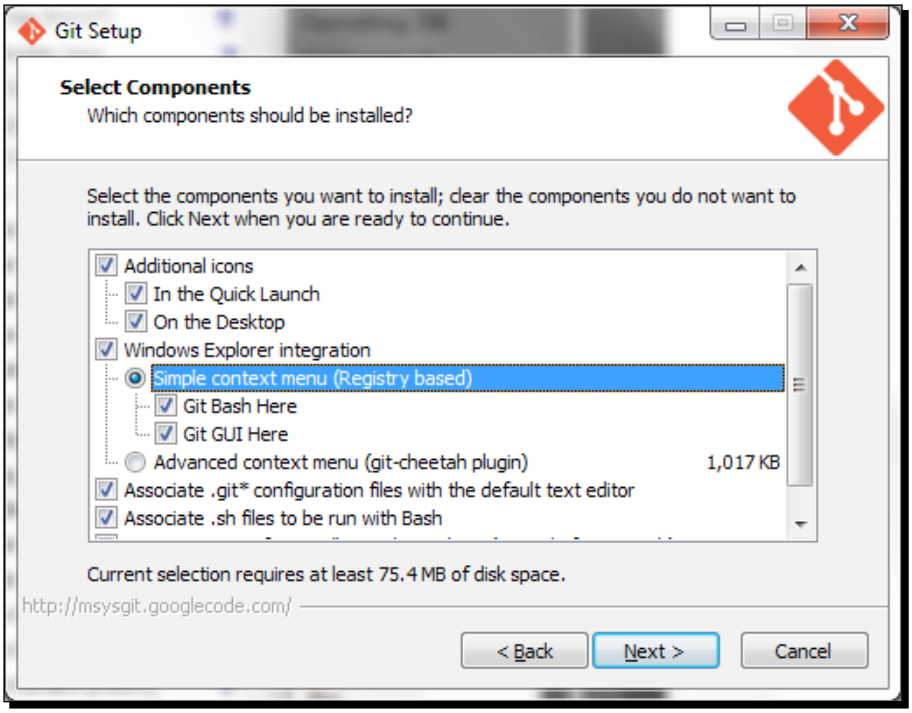
To get to know more about what you can and cannot do with the software package, go ahead and read it.

3. Clicking on the **Next** button on the screen brings us to the following screen, which gets your preferred location for the installation:



The default location is inside the **Program Files** directory of your Windows installation. If you group all custom software installations in a separate partition to safeguard your data in case of an OS crash, you can go ahead and select your preferred path by clicking on the **Browse…** button.

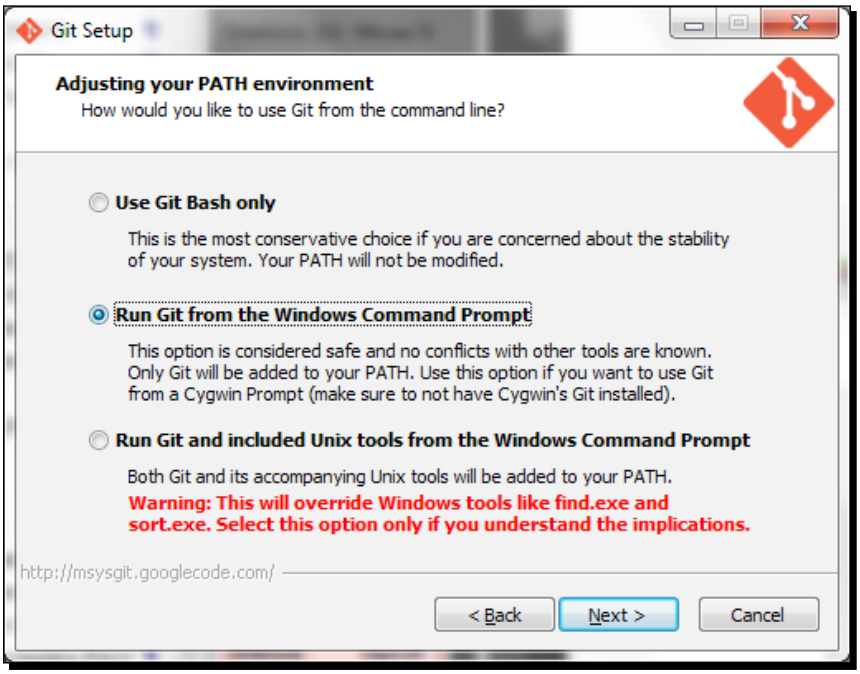
4. After clicking on **Next** you will be facing a component configuration screen in which you need to select **Git Bash Here** and **Git GUI Here** options under the **Context menu entries** section, as shown in the following screenshot:



These options are going to provide us quick access to interfaces that we can use to command Git. We will see more about them in the oncoming chapters.

5. Next we are prompted to select a group name under which shortcuts are placed in the start menu for easy and quick access. Let us leave it to the default value `Git` and click on **Next**.

6. This brings us to the following screen, in which we select the second option, which says **Run Git from the Windows Command Prompt**:

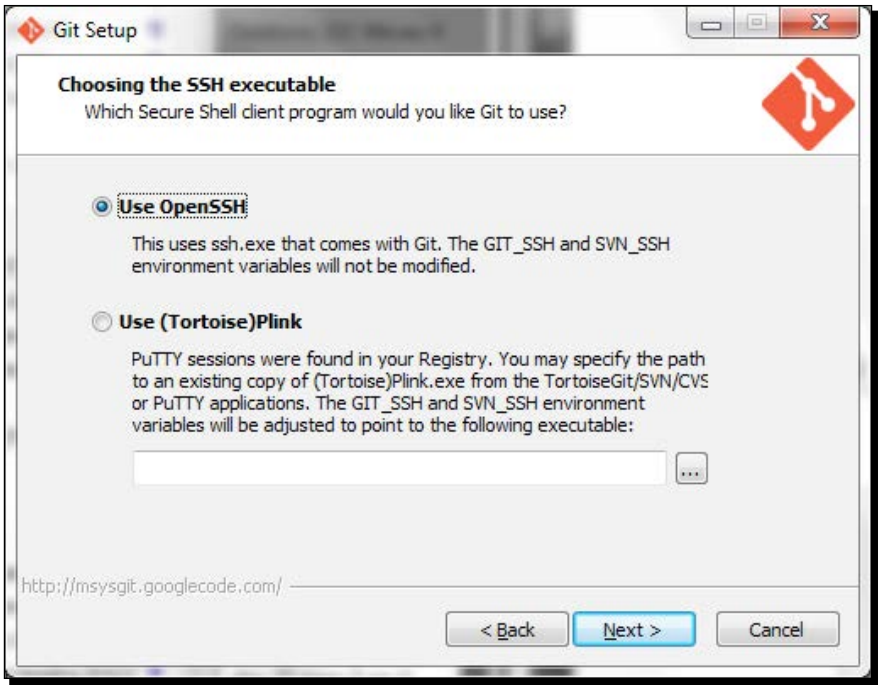


This setting is basically for people who will be using command lines to control Git and its activities. This option is going to allow us to command Git additionally from the native Windows command-line interface DOS.

After making the selection, click on the **Next** button.

7. The remaining two main steps are configurations that we will need while working remotely and/or collaboratively across operating systems.

If and only if you have any SSH sessions in your registry will the installation file detect that and prompt you with the following screen:



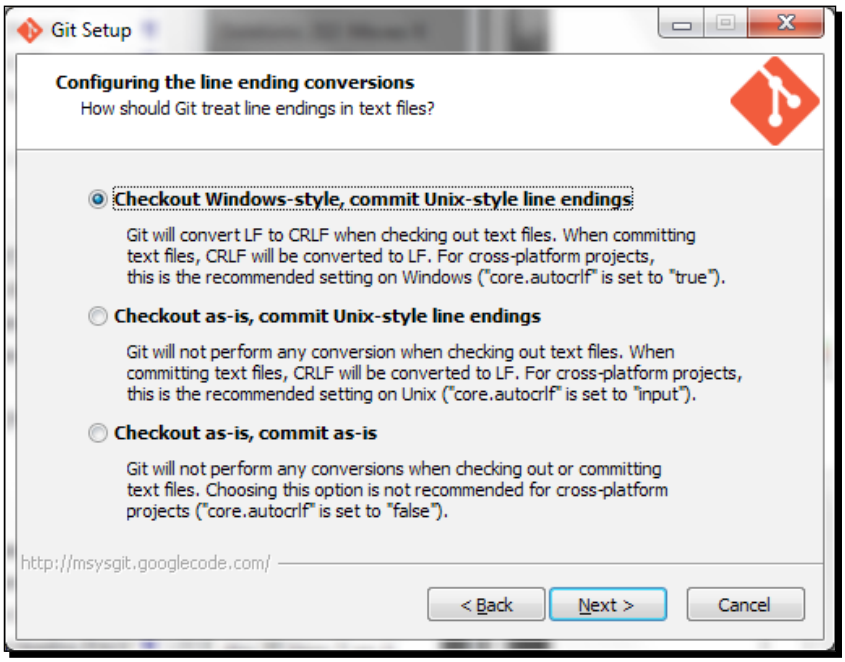
If you are a user to whom the very term SSH is new but happen to have SSH sessions in your system through other means, or an experienced user who wants to switch to OpenSSH, go ahead and select the **Use OpenSSH** option.

If you are comfortable using other SSH applications to connect to remote systems, select the **Use (Tortoise)Plink** option. Make a selection on the **Choosing the SSH executable** screen and click on **Next**.

OpenSSH keys are created with ssh-keygen and cached with ssh-agent. With the putty suite, keys are created with the graphical program puttygen and loaded/cached with pageant, and SSH is done using putty.

8. Anybody who has worked with files across different types of operating systems will definitely know about the problem with different styles of line endings.

9. Now you need to tell Git how it has to treat those line endings. In the following screen select **Checkout Windows-style, commit Unix-style line endings** so as to ensure that there is no clash between **carriage return line feed** (**CRLF**) and **line feed** (**LF**) when working across platforms:



That's about it. Your installer should finish the installation now.

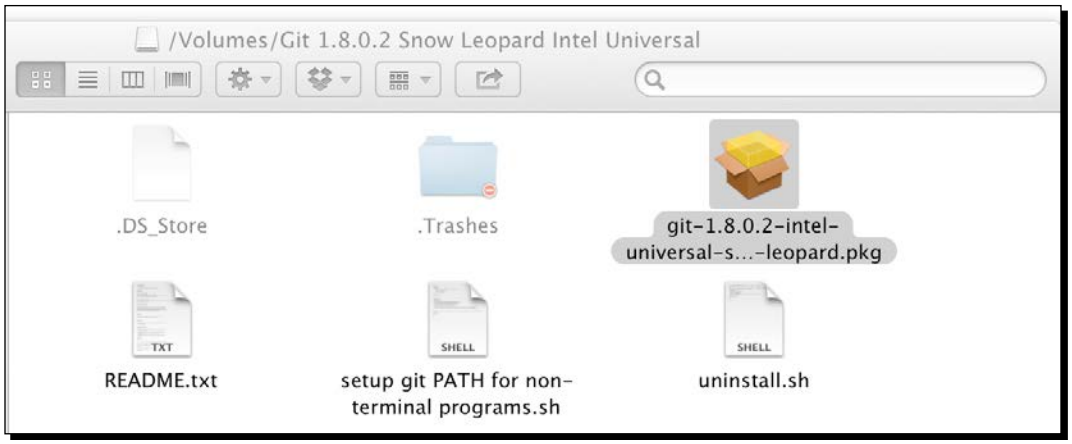
What just happened?

Congratulations! Your Windows machine is now ready to control the versions of any content with Git.

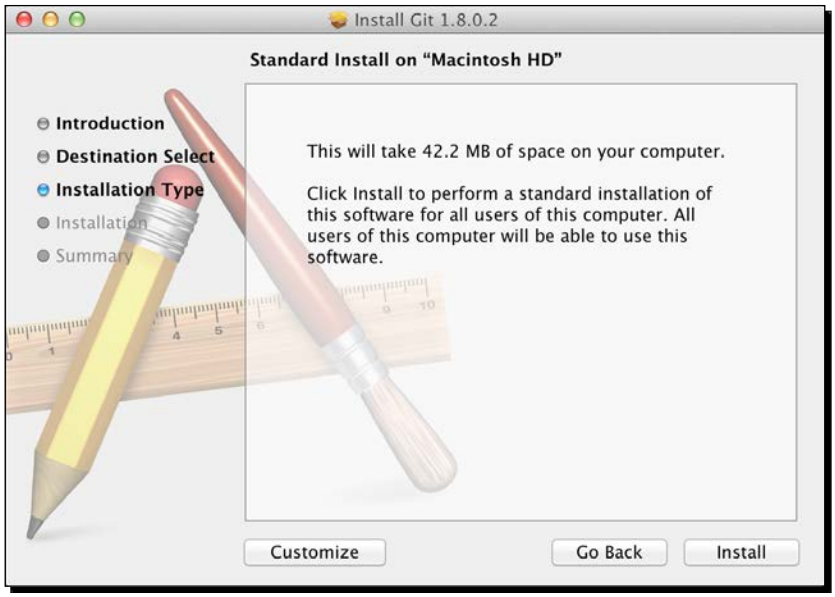
Time for action – installing Git on Mac

Perform the following steps:

1. Double-click on the `.dmg` file to get started with the installation. The following screen should appear:

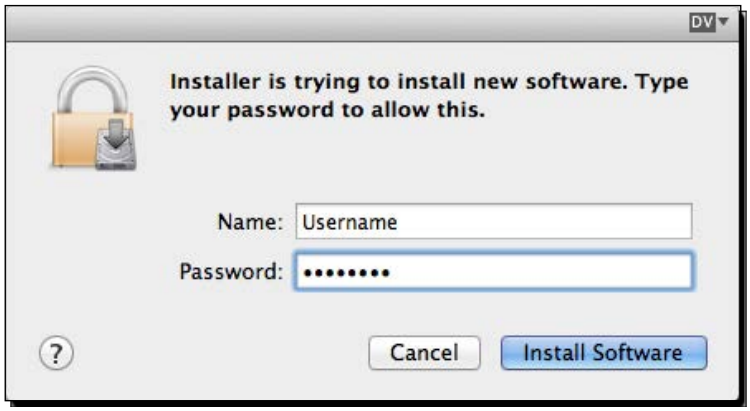


2. Double-click on the `.pkg` file to start the installation process. The window appearing in front of you welcomes you by providing information on what's going to happen henceforth.
3. On clicking the **Continue** button, you will be provided with information about how much space the software is going to occupy on your disk, along with access level of this software for other users of your computer, as shown in the following screenshot:

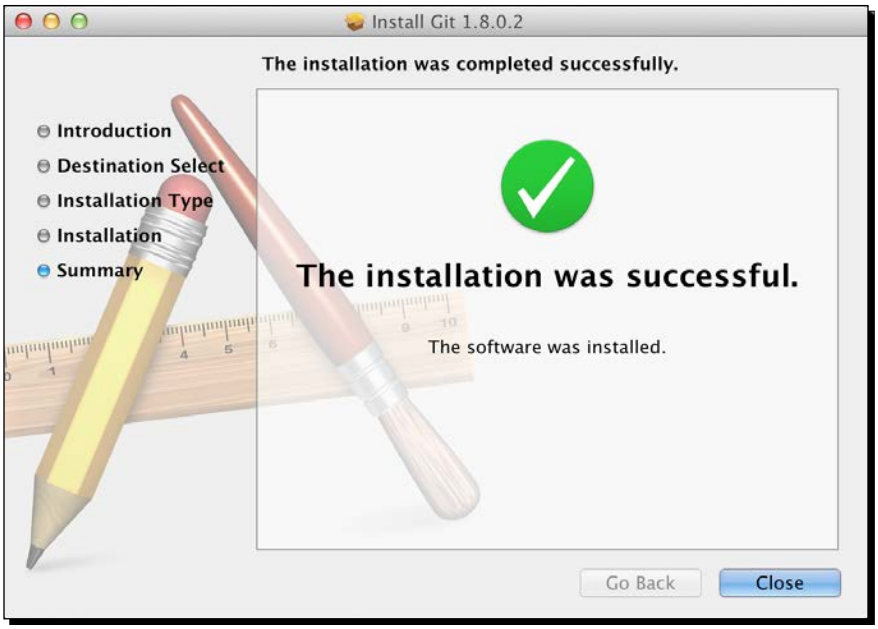


4. You can customize this as per your needs, but as of now let's go ahead with the defaults and install it for all users by clicking on the **Install** button.

5. You will be asked to provide your administrative password for continuing the installation.



6. If the authentication is successful your installation will be finished, which is indicated by a success message, as shown in the following screenshot:



What just happened?

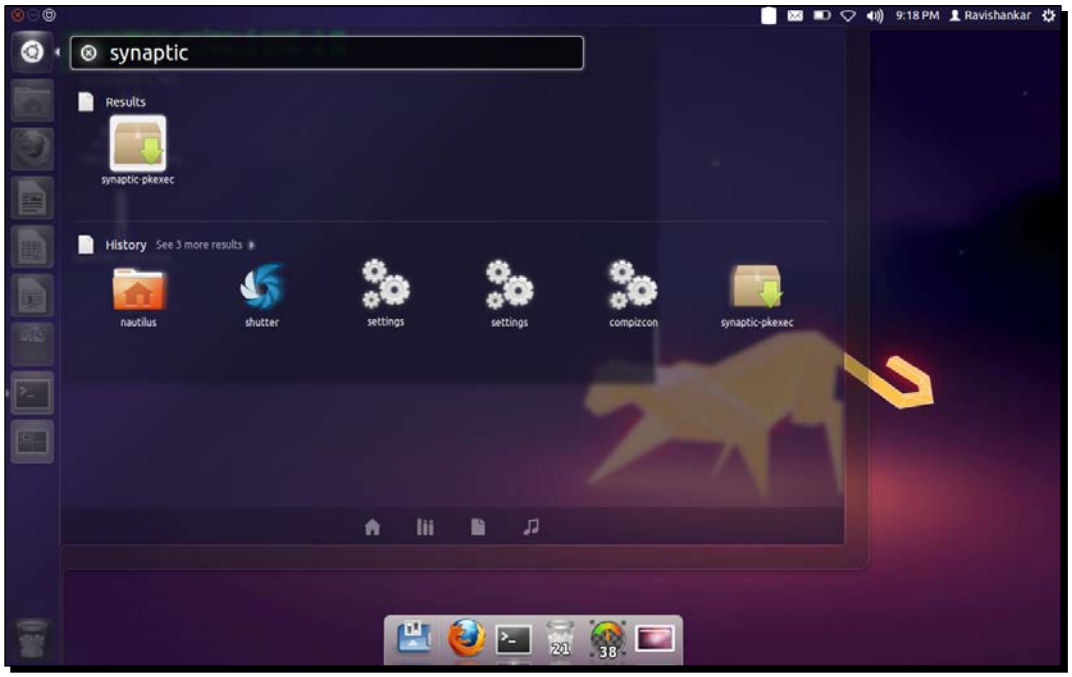
Congratulations! Your Mac machine is now ready to control the versions of any content with the help of Git.

Time for action – installing Git on Linux

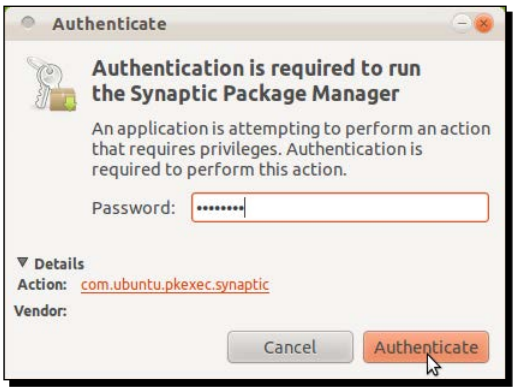
As we discussed earlier, we are going to perform a Git installation on a Linux operating system with the help of your distribution's inbuilt **graphical software management system**. In this tutorial I have used a distribution called Ubuntu, which is based (downstream) on the famous Debian operating system.

In here the software management system is called synaptic. Perform the following steps:

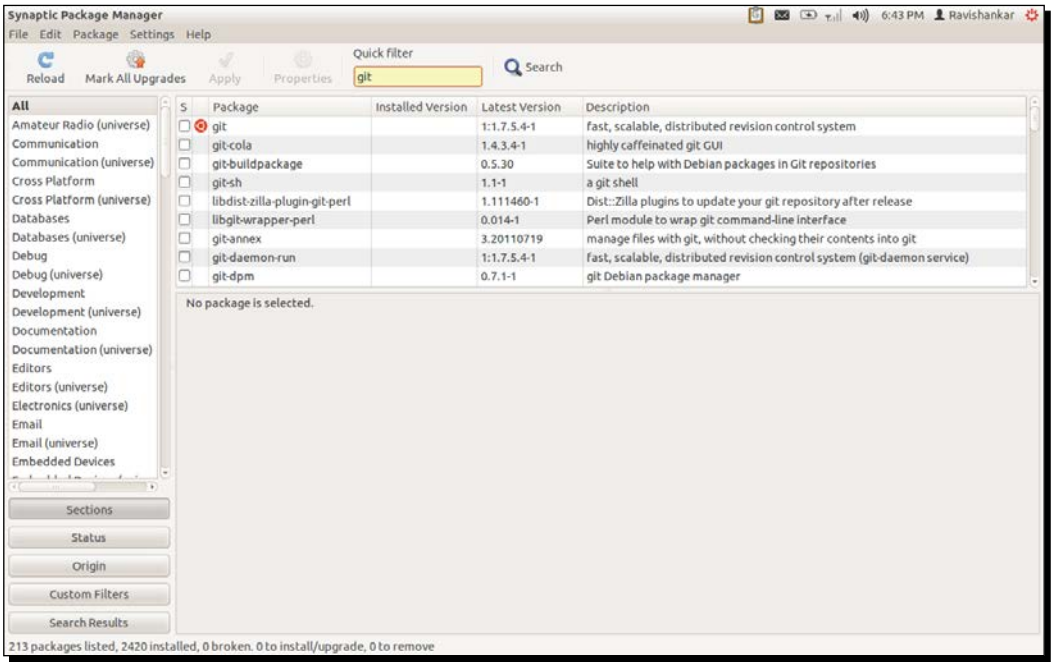
1. Open up the **run utility** prompt by pressing *Alt* + *F2* and type `synaptic`.



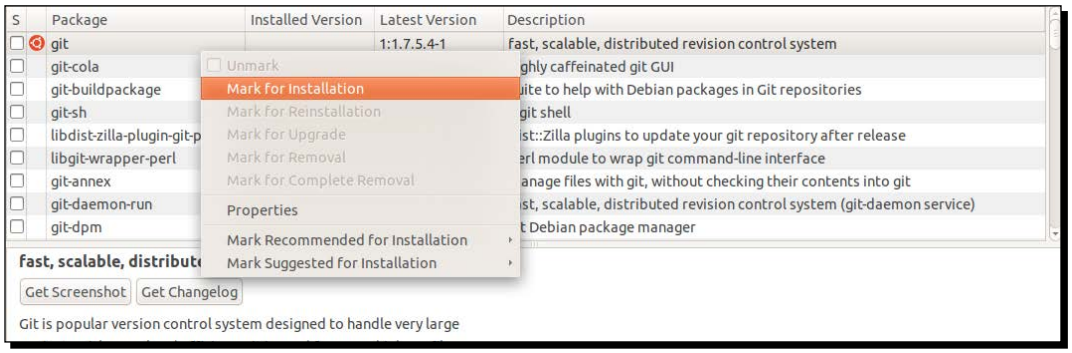
2. Matching tools automatically get displayed below; let's click on the first one, which says **synaptic-pkexec**. This pops up an authentication dialog box, since the installation requires elevated privileges, as stated earlier. So give it your password that has administrative access and click on **Authenticate**.



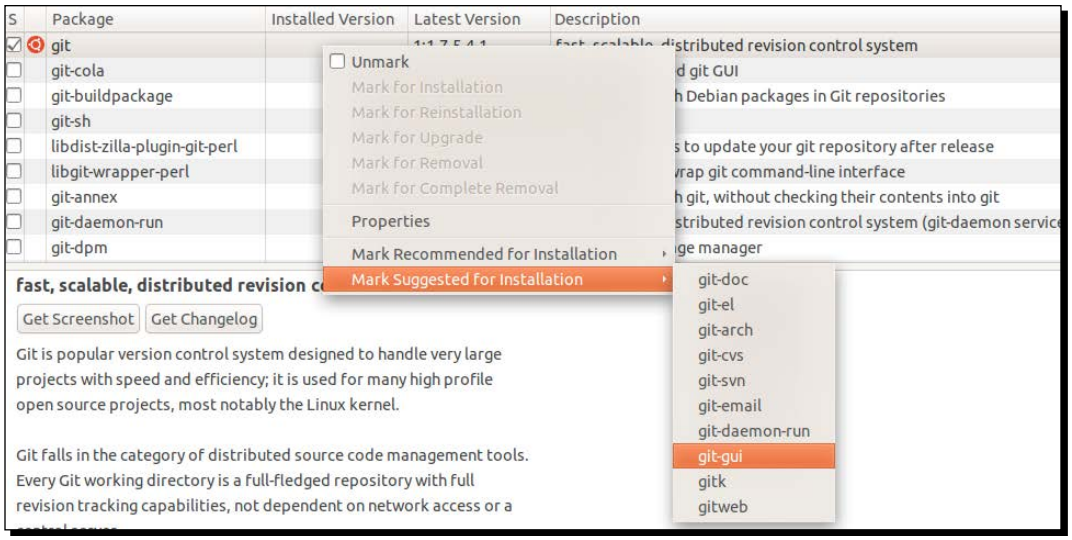
3. This should open up the **Synaptic Package Manager** window, which shows you the available packages on the centre pane and available repository sources on the left. Let's type the name of the package that we want to install now which is `git` in the **Quick filter** text box. It automatically populates the matching packages on the content pane below it.



4. Now we can see a package named **git** whose version, which is shipped by default, is **1.7.5.4** (don't worry about the version mismatch; the concepts we are going to learn henceforth are the same for all) and its description says **fast, scalable, distributed revision control system**. This is the package that we want to install, so let's right-click on it and select **Mark for Installation**.

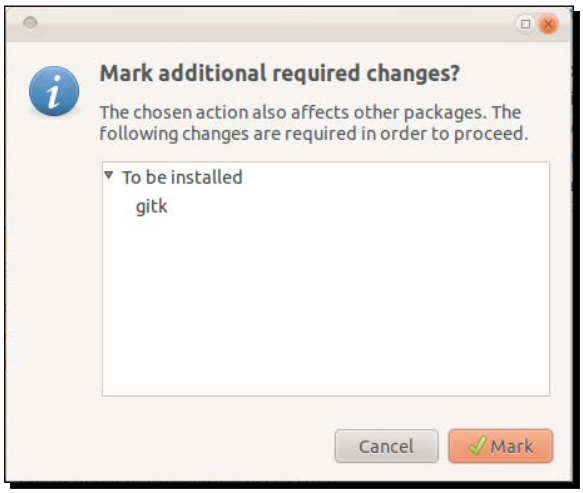


5. Now the checkbox next to the package name will have a tick option indicating that you have selected that particular package for installation. To handle things much more easily we are going to need two more packages associated with Git, namely **Git GUI** and **Gitk.** So let's right-click on the same package again and select the option **Mark Suggested for Installation** and select **git-gui** from it.

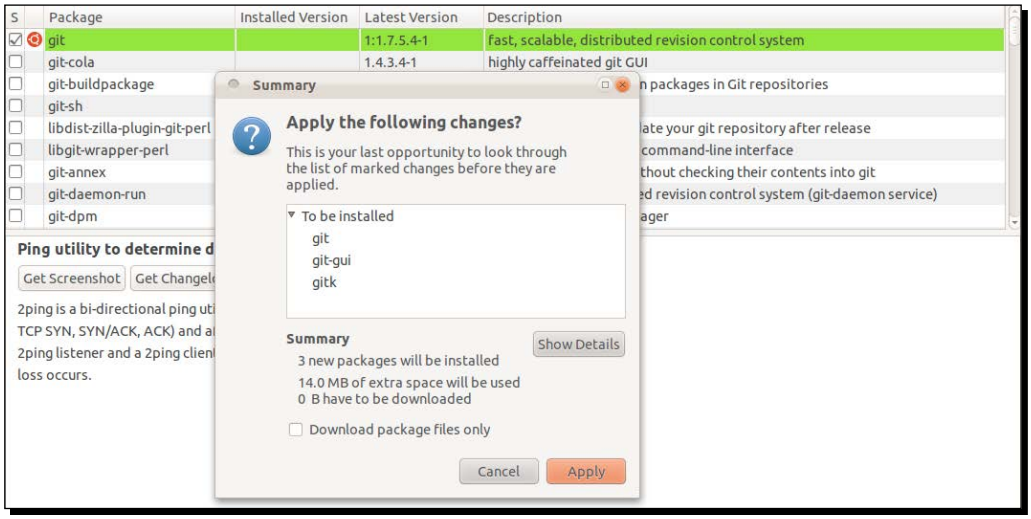


6. The package manager will prompt you, asking about the required package changes, that is, adding **gitk** to the installable list; so click on **Mark**. Now that you have marked your packages to be installed let's go ahead! Wait, we did not select the gitk package which we said we would need! Yes that's right, we didn't. But that will be automatically installed, as it is a dependent package. You will see that in the oncoming steps.

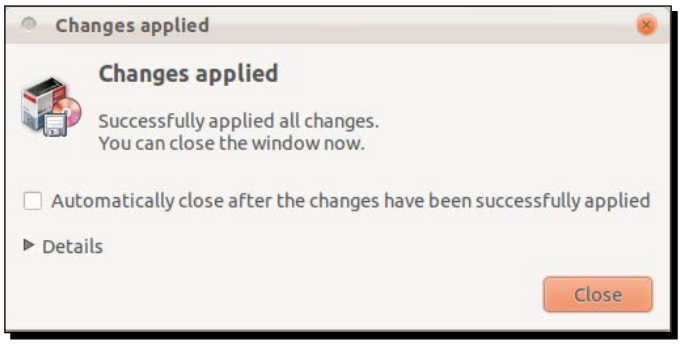
7. Now go ahead and click on the green **Apply** button on the shortcut bar below the menu bar. You will be prompted with a confirmation dialog box as follows:



8. On clicking on **Mark** you will be given a summary of the packages that are going to be installed and asked for confirmation.



9. Upon confirmation the installation gets started and when done you get to see a success page as follows:



What just happened?

Congratulations! Your Linux machine is now ready to control the versions of any content with the help of Git.

Summary

Having learned about the power of Git in the previous chapter, in this chapter we got a copy of Git and made it your own.

Then we successfully installed Git on your operating system, making it usage ready.

Now that we've got Git installed, our next step will be to personalize it, after which we will see various magic words that will make the tool help us to increase our productivity. This is the topic of the next chapter.

3

Polishing Your Routine – How to Increase Everyday Productivity

We scratched the surface of version control system concepts to understand how Git can make us more efficient in doing our work. Without any delay we also quickly grabbed a copy of it and got it up and running in our systems.

I hear you shouting with excitement, "Let's dive in!".

Aye aye, Captain, here we go. In this chapter you will look at five important concepts, which is all you really need most of the time in your workplace:

- Initiating the process
- Adding your files to the cabin (repository)
- Committing the added files
- Checking out
- Resetting

That's right, just five concepts are all you need to create a difference. And of course we shall learn how to get back on track with Git's built-in helper functions if you get lost along the way.

Suit up – getting ready for your Git

Let's say you have a magic wand, and it will do exactly what you order it to do! Yeah, that's right, you have Git now. You need to command Git to do what it has to do for you.

Sounds fun, right?

We already read that to maintain multiple versions of files they have to be kept inside a directory (folder), so we shall create a directory called `Workbench` on your desktop to learn the concepts explained in chapters hands on.

When it comes to handling computers there are people who would like to get the job done with either of the following:

- GUI mode (graphical user interface)
- CLI mode (command-line interface)

A combination of both can also be used. In the interest of serving a diverse audience, we shall try to cover both modes of implementation.

Initiation

Initiation is nothing but a process of pointing your finger towards that directory so that Git will know it has to monitor its contents for changes from then on.

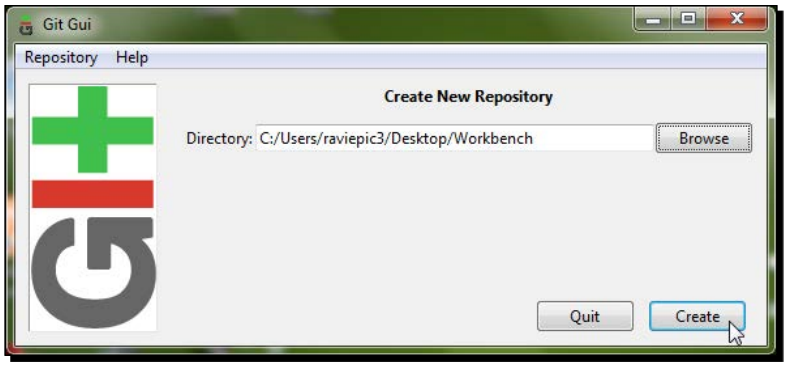
As we discussed earlier, we shall cover both ways (GUI and CLI) of performing these operations.

Time for action – initiation in GUI mode

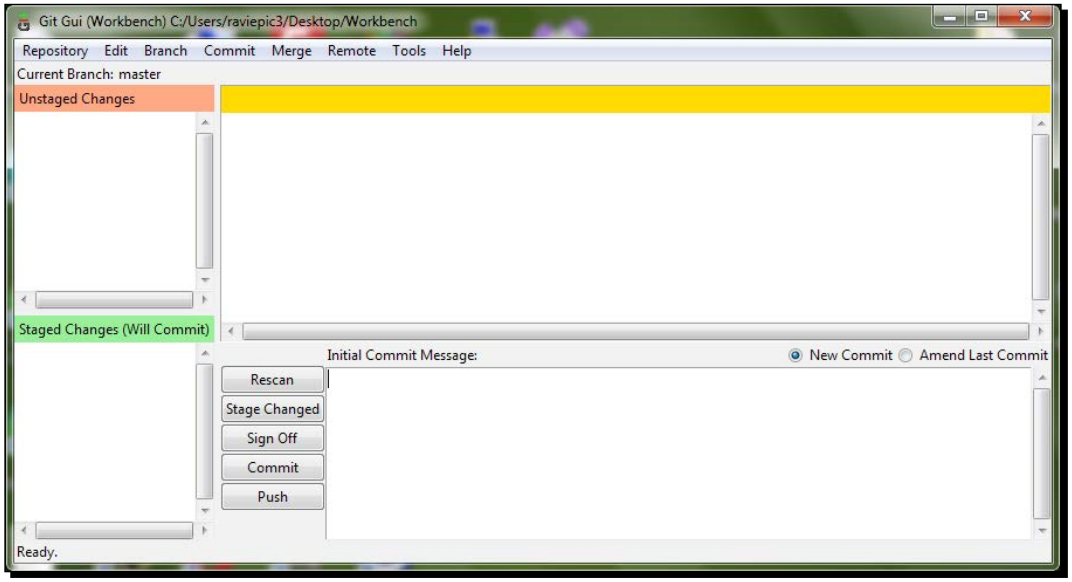
To create/initiate a repository, perform the following steps:

1. Open **Git Gui** from your desktop or from your applications menu and select the option **Create New Repository**, as shown in the following screenshot:

2. Git will present you with a new screen, expecting from you the location of the directory that you want to make a repository. So click the on the **Browse** button, select our **Workbench** directory from the desktop, and click on the **Create** button.



3. You should get a screen as follows:



Now don't close this window; we shall continue from this screen for our remaining concepts.

What just happened?

You have successfully commanded Git to monitor your `Workbench` directory and its contents.

The previous image showed the master page, which we will be interacting with very often. It consists of four panes; let's call them the following:

- **Unstaged Changes** pane (top left)
- **Staged Changes** pane (bottom left)
- **Differential Content** pane (top right)
- **Action** pane (bottom right)

In our example we created a new directory called `Workbench` and initiated it as a repository. You can also follow the same procedures to convert an existing directory that already holds your files into a repository for Git to monitor. When you do that, your files inside the repository will initially appear in the **Unstaged Changes** pane.

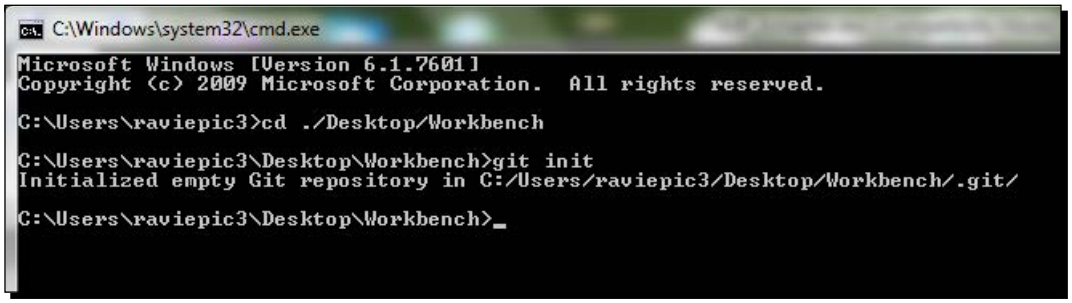
Time for action – initiation in CLI mode

For those who like to hear the sounds of more keystrokes instead of clicks, there's always **command-line interface** (**CLI**) mode.

There has been a constant increase in the percentage of people who are quick with keyboard's key strokes, giving more priority to performing operations using keystrokes instead of mouse clicks wherever possible. This is also a main reason why **Gmail** introduced shortcut keys for almost all of its functions.

To create or initiate the repository using the command-line interface mode you need to do the following:

1. Open your **shell** (command prompt in Windows or Terminal/Console in Mac/Linux).
2. Go to your `Workbench` directory on your desktop using the `cd` (change directory) command.
3. Once you are inside the `Workbench` directory type `git init` and hit *Enter* to complete the initiation process.
4. You should get a status message from Git saying **Initialized empty Git repository in your/path/to/Workbench/directory/goes/here**.



Ah! The sound of keystrokes, so good to hear.

What just happened?

You have successfully commanded GIT to monitor our `Workbench` directory and its contents. `Init` is the operational keyword that initializes the repository.

Behind the screen

This initiation process will create a directory called `.git` inside our `Workbench` directory. This directory is usually made **read-only** and **hidden** by Git to safeguard itself from accidental deletion or tampering by users. It's the place where Git will hold all the history about your files and changes made to them.

So be careful with that directory; deleting it will *wipe out* the entire history of your files present in that directory.

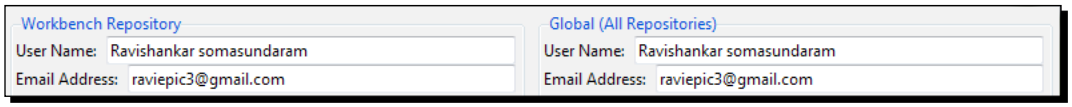
Configure Git

Gear up your Git installation for usage by configuring it properly. There are several reasons why you should configure Git before starting to use it, but discussing all of them now would be premature, so we shall learn about them as and when the necessity occurs. For now, as a bare minimum configuration to get started, we will tell our name and e-mail address to Git so that it can log the changes under our identity.

Time for action – configure Git in GUI mode

To convey to Git our name and e-mail address using GUI mode follow these procedures:

1. Select **Options** from the **Edit** menu of the screen that you left open after the initiation process.



   The configuration screen is divided into two halves.

   - Local configuration (left side – particularly our Workbench Repository)
   - Global configuration (right side – applies to all the repositories created using this installation)

2. Don't let the big screen with numerous options overwhelm you. Let's focus on the top portion alone for now and type our name and e-mail address in both local and global configurations as shown in the previous image, and hit the **Save** button.

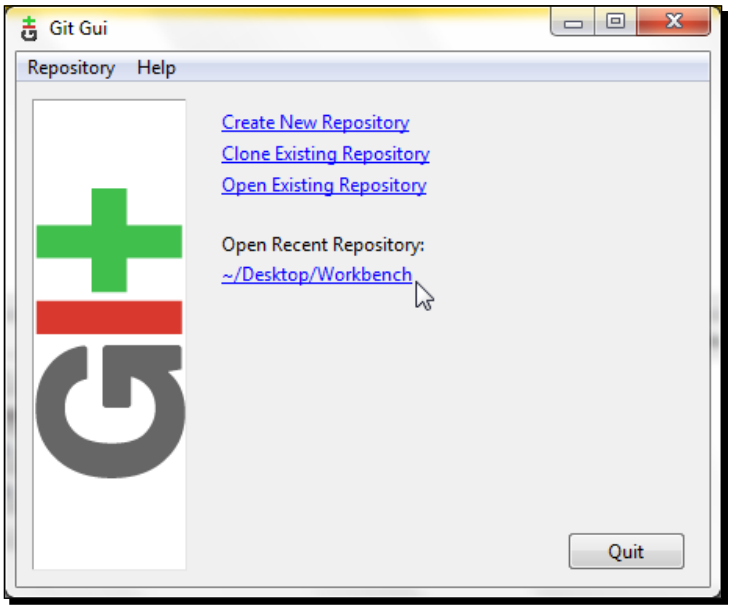
What just happened?

By giving out our username and e-mail address both locally and globally we have provided ways for Git to identify and group the changes made to files present in any repository.

Out of flow

Just in case you closed the screen after the initiation process and were wondering how to go about getting to the same screen again, don't worry. There are two ways of getting back.

1. Open up **Git Gui** where you will see a newly added option called **Open Recent Repository**, under which you can find our **Workbench** repository.

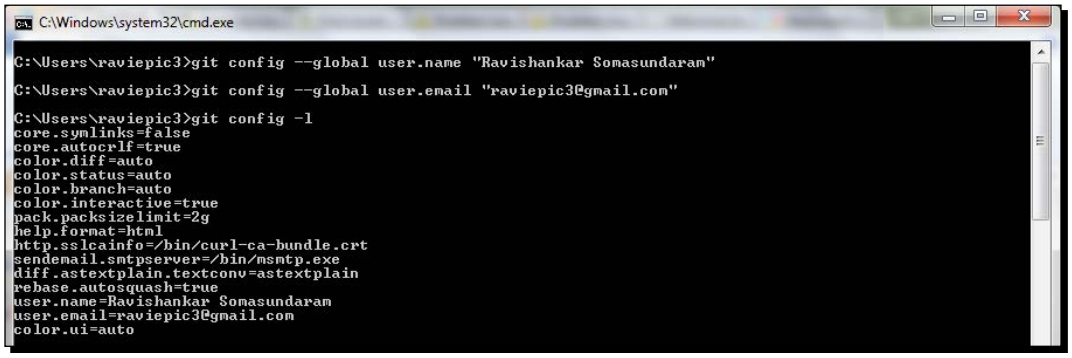


2. Locate the `Workbench` directory on the desktop and right-click with your mouse on the folder. In the menu select **Git GUI here**. People who want to switch from CLI mode to GUI mode can use this option as well.

Time for action – configure Git in CLI mode

To configure Git using CLI you can use the following commands:

```
git config --global user.name "your full name"
git config --local user.name "your full name"
git config --global user.email "your email id"
git config --local user.email "your email id"
git config -l
```



What just happened?

By giving out our username and e-mail address both locally and globally we have provided ways for Git to identify and group the changes made to files present in any repository.

`config` is the operational keyword that needs to be used with `git` to set up the configuration of Git. To set a global value we add the `--global` parameter with the command, and to set a local value we add the `--local` parameter with the command.

As the name indicates, global configuration is nothing but a global value for all repositories created in the system by that system user, whereas local configuration is the exact opposite. As you would have guessed by now, the parameters `user.name` and `user.email` are used to record the user's name and e-mail address, respectively.

To get a list of configurations set till date you can use the last command, which had the `-l` parameter. It lists all the configuration variables for you.

Adding your files to your directory

Now that you have set a perfect base to operate on, let's move one step ahead by adding your files to the repository that you have created.

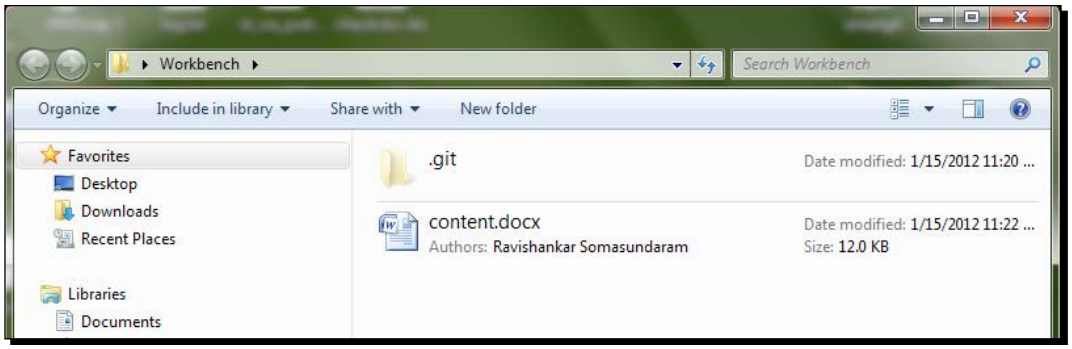
Whoa, wait! What's that term that we often came across earlier – **repository**?

Moving forward, we will address a directory/folder that has been pointed to Git to monitor as a repository.

Yeah, baby, learn Git lingo and impress your date! The process of adding files is as simple as copying and pasting or creating your files inside our repository and asking Git to watch them.

Time for action – adding files to your directory (GUI and CLI mode)

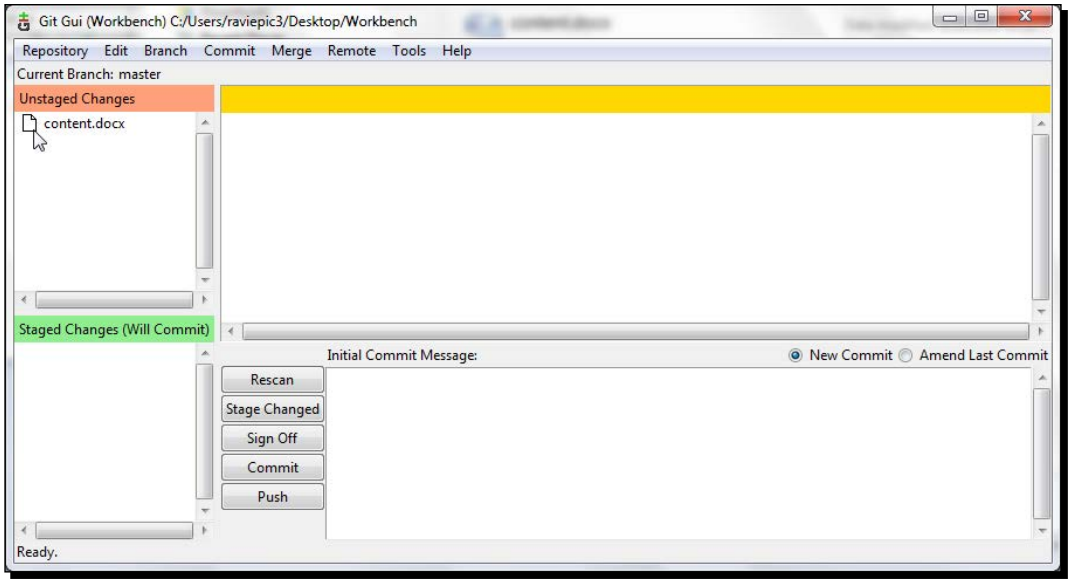
Let's create a Word document called **content.docx** that contains the text "I love working with Git. It's a simple, fast, and superb version control system" to learn and put in practice the functionalities mentioned at the beginning of our chapter (people who are not able to create a `.docx` file can proceed with any other document format such as `.odt`, `.txt`, and so on).



Git will report to you about the files that have been added to our repository and will stand by for your instructions to proceed. Now we can go ahead and tell Git to monitor these files for changes by performing the steps that we will discuss next.

If you are using GUI mode, perform the following steps:

1. Click on the **Rescan** button (or press *F5* on your keyboard) present in the **Action** pane.



2. Click on the page-like icon next to the filename to push the file to the **Staged Changes** pane.

If you are using the CLI mode, use the following commands:

```
git status
git add content.docx
```

What just happened?

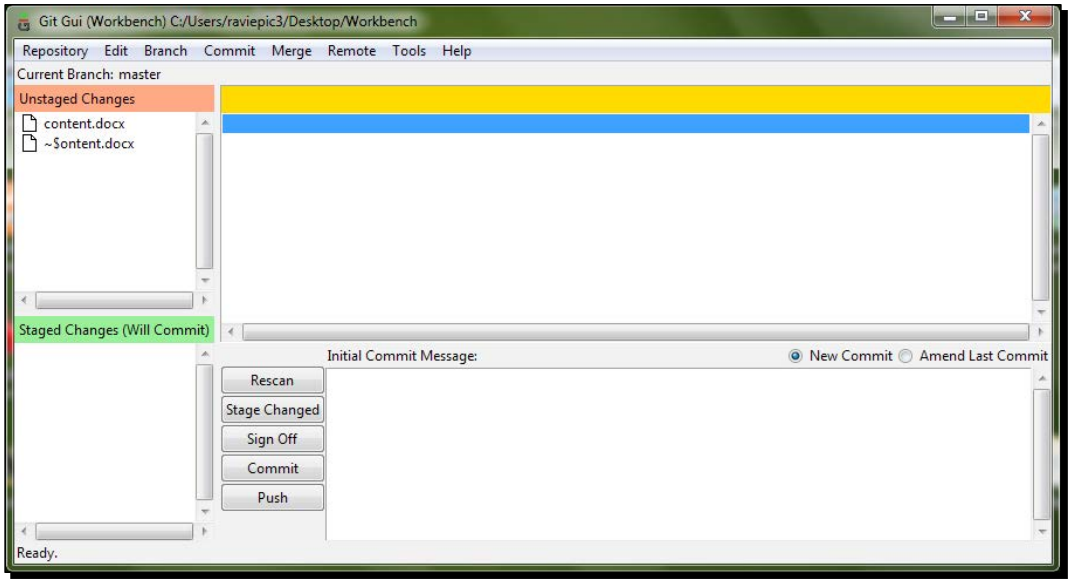
We have successfully added our files to the repository.

By clicking on the **Rescan** button or typing the `git status` command we ordered our slave to list the changes that were made to the repository since its previous state. These changes are called unstaged changes, meaning changes that have happened since our last confirmed state of the repository.

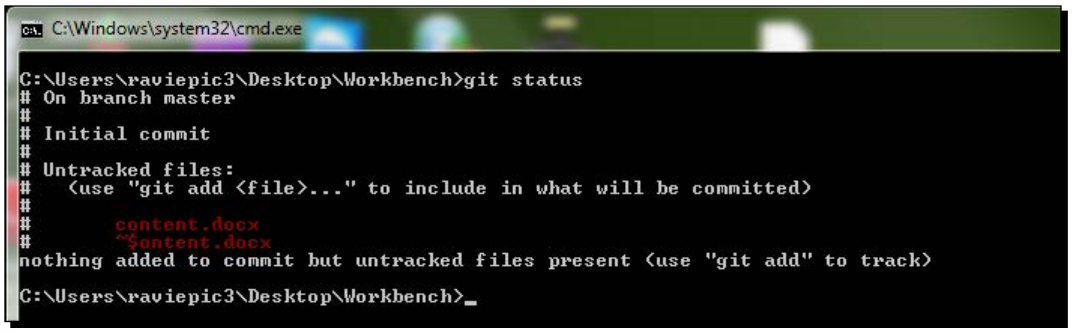
These changes have to be confirmed by the user by moving them to the staged changes state, which is done by clicking on the file icon next to the filename or using the `git add` command.

Ignore 'em

We just saw ways to put your files under Git's radar but there are numerous situations where one might want to avoid adding certain files into one's working repository. As a live case, after adding some content in the `content.docx` file and trying to add your files into the repository as seen in the previous step, some might have encountered the situation where Git reports (of course, after refreshing the Git GUI or using the `git status` command in CLI) changes made in two files, **content.docx** and **~$ontent.docx**, as shown in the following screenshot:

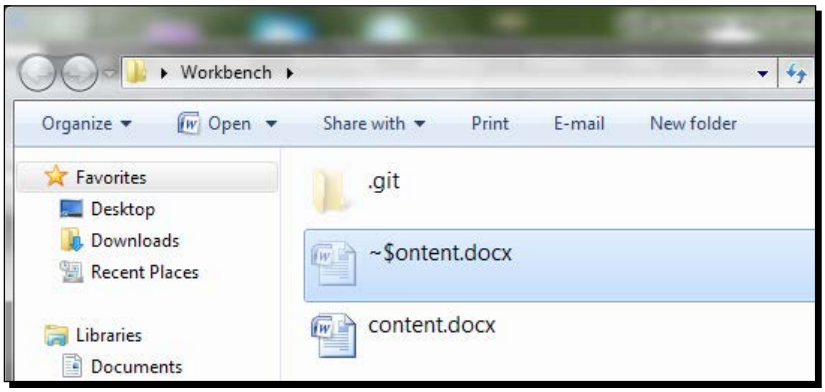


This happens only if the opened `content.docx` Word document is not closed before refreshing or hitting the `git status` command.



This is because Microsoft's Word application has a habit of saving your current workspace at regular intervals (which can be configured) in a temporary file for disaster recovery.

It is only because of this mechanism that Word prompts you with a file recovery dialog from where you can retrieve your latest changes when a proper save is not done before abrupt closure of the document.



Not only Microsoft Word but all smart applications and editors follow such a procedure to comfort the end users. These files are automatically deleted once the corresponding source file is saved and closed properly. There would be no value added in controlling the versions of these temporary files.

So while adding files to your repository it is important that you exclude these temporary files before proceeding to the committing concept as the reversal would be a painful process.

This way of adding files to Git holds good for a few files, but when it comes to handling several files in the repository, clicking on the icon next to each of them or performing a `git add` for each file is going to be a time consuming and tiresome activity.

Bulk operations

When you want to move several files from the **Unstaged Changes** area to the **Staged Changes** area you can use the following:

- **GUI**: Press *Ctrl* + *I* and select **yes** if there is a prompt about adding unknown files instead of clicking at each and every icon next to that file.
- **CLI**: The command `git add .` is the equivalent of pressing *Ctrl* + *I* when using GUI mode. It will stage all your changes at a single shot. The use of **wildcard characters** like `*.docx` is also permitted.

  ```
  git add .
  git add *.docx
  ```

Using these options we can eliminate our tiresome process of adding a single file at a time, but it defeats the objective of excluding the temporary files from being added to the repository. So how do we combine the power of bulk operations along with the control to exclude certain files or file types?

.gitignore to the rescue

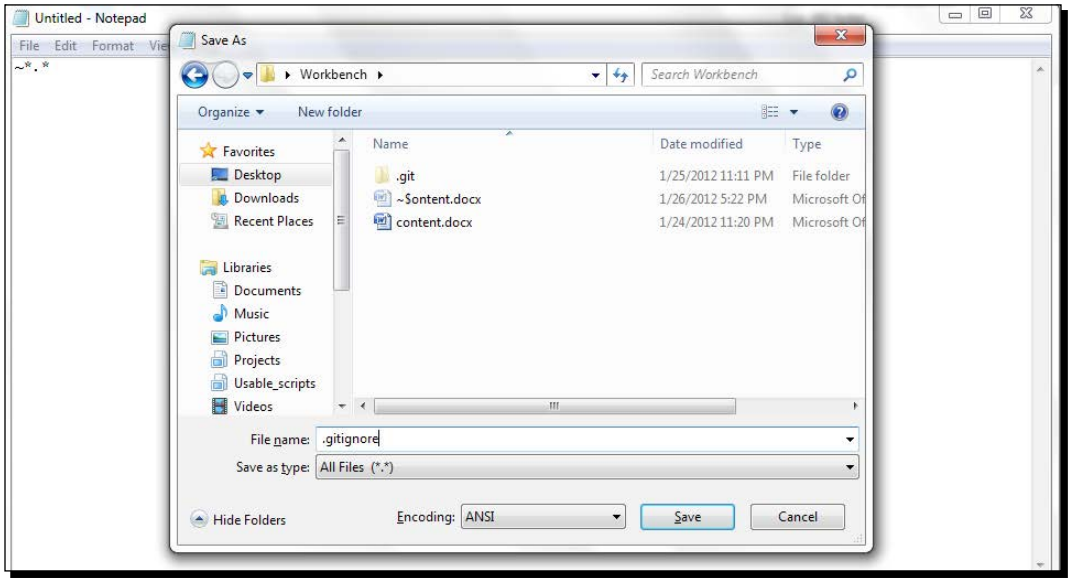
To handle this smartly, Git has a provision. By creating a file called `.gitignore` inside the repository and entering the names of files or pattern of the filenames we can make Git exclude them.

Time for action – usage of .gitignore

1. Open your text editor and type the following:

   ```
   ~*.*
   ```

2. Save the file as `.gitignore` inside our **Workbench** repository, as shown in the following screenshot:



Make sure to select the **All Files** option from the **Save as type** listbox when you save the file.

What just happened?

We have successfully commanded Git to ignore the temporary file created by the Word application. Go ahead and refresh your GUI or get the status from your CLI now. The only addition to your `content.docx` file in the **Unstaged Changes** area would be the `.gitignore` file and not the temporary file.

Every time Git wants to check for new files (untracked changes) present in the repository it checks with the `.gitignore` file for exclusions. By observing the temporary file's name (`~$ontent.docx`) we can guess that any temporary file created by Word is going to start with the special character **~** so we put an entry in `.gitignore` to match all files starting with that character. The very entry `~*.*` under the `.gitignore` file says to exclude any filename starting with the character ~ with any extension.

Though addition of the `.gitignore` file itself is a one-time process, the exclusion rules inside the file have to be updated as per the nature and content type of the files added in the repository as required.

Undo addition

At any given point of time before committing, if you want to move a file from the **Staged Changes** to the **Unstaged Changes** area you can do the following:

- **GUI**: Click on the tick icon next to that particular filename present in the **Staged Changes** pane
- **CLI**: Use the following command:

  ```
  git reset filename.extension
  ```

Committing the added files

Until now we have initiated the repository, added our files into the repository, and confirmed those changes by staging them (pushing them to the staged changes stage) but until they are committed the files are not said to be under version control. (This is because only when you commit does Git record the content of the files and save it as a new phase of that file/files, so that next time it can identify whether the files have any change of content by comparing the existing version to the last saved version).

This is a new addition to your Git lingo: This process is called **committing**.

So let's make an initial commit of your files. The first time you add a file to the repository and make a commit, Git registers the new file. Any further commits made to these files inside the same repository will be a commit for the changes based on the previous version of the same file available in the repository.

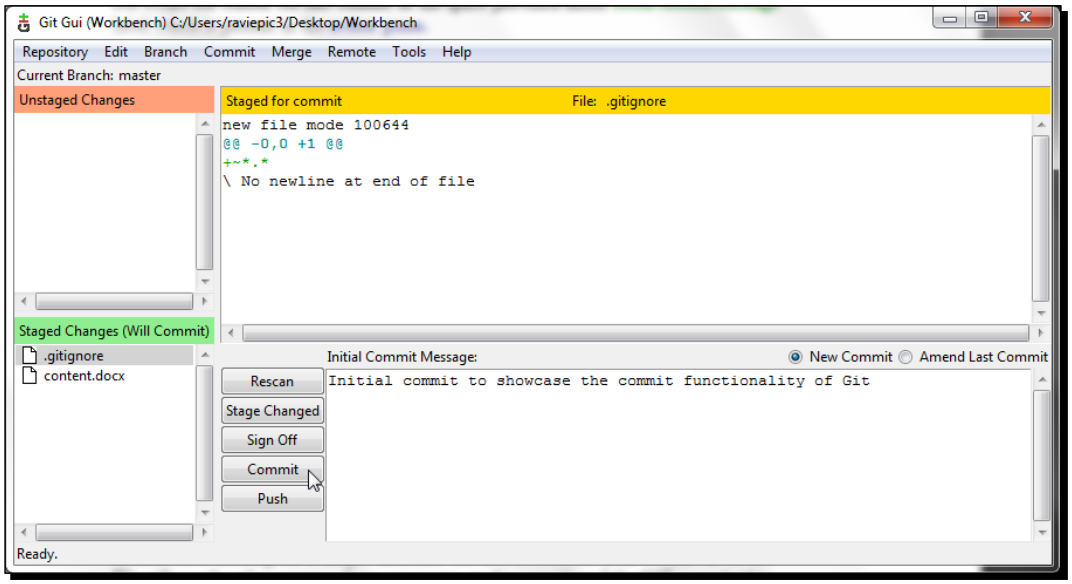
Though Git follows your orders it has a healthy habit of associating a comment at the time of every single commit so that it can learn about your behavior and moods with respect to various file types and build an artificially intelligent system based on observed patterns to automate your routines.

Basically comments that you provide at each commit is just to help yourself or any other person reading the history of your repository understand the purpose of, and/or changes to, the files.

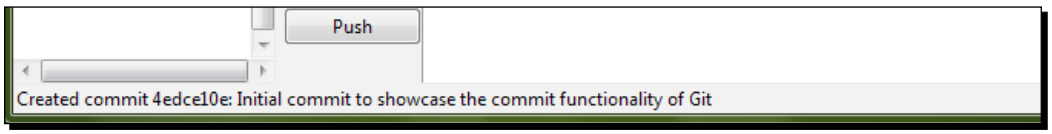
It's good to make a comment, which can be anything informative. Having learned the theory behind it, let's see it in action.

Time for action – committing files in GUI mode

1. Let's type our reason for this commit in the space provided under the **Initial Commit Message** label present in the **Action** pane.



2. Click on the **Commit** button. Once the commit is done Git gives you a status message at the bottom of the pattern **status commit ID: your comment for the commit**.

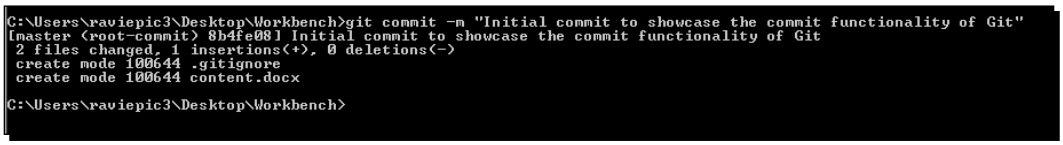


Commit ID is nothing but a unique identifier for Git to recollect your commit in future. We will see the other usages of our comments on the commit and the Git commit ID in the oncoming functionalities.

Time for action – committing files in CLI mode

Assuming you already have the command prompt opened by doing the steps mentioned under initiation process, give Git the following command:

```
git commit –m "your comments for the commit"
```



If you see a status message similar to the one mentioned previously it's a sign of an affirmation.

What just happened?

You have successfully committed your files to the repository. Henceforth any changes made to these files will be relative.

Let's see what happens when you change the contents of the file inside the repository.

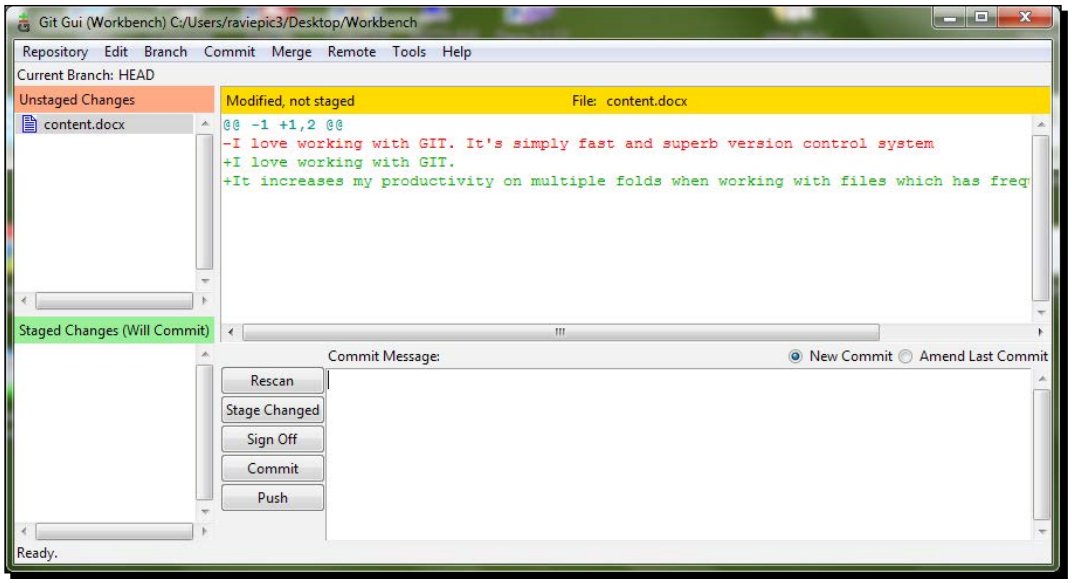
I suddenly feel that I need to convey how Git impacts my work instead of just saying "It's a simple, fast, and superb version control system" in our `content.docx` file. So I am replacing this with the text "It increases my productivity manyfold when working with files that have frequent content changes."

Git tracks the change and indicates it to us when asked about a status update.

Time for action – rescan in GUI mode

If you already have **Git Gui** open then just hit the **Rescan** button to get the latest status update from Git. If you don't have the tool opened already, I'm assuming you know how to open it.

And you see, it shows the files that have changed from their earlier state in the **Unstaged Changes** area.



You recently learned how to stage a file's change and how to commit it, so I'll leave the rest to you. Just so that you know, my commit message for this commit was "Added more text that explains why I use Git."

The **Content** pane shows the change that you have made in the file. Green text indicates addition and red text indicates deletion when compared to the previous version of the file. We shall explore more about this in later chapters.

For CLI lovers, we have been using the `status` command from the time we added files to check the status of the repository, and it is no different here. Employ the `git status` command to get to know about the changes in your repository.

Checking out

Well, until now we have been moving forward in versioning our files by giving orders to Git with the concepts we learned. Whatever you have learned up to now is just a one way process!

To make it clearer – *how do you feel about not knowing how to use the undo and redo features of your Word application?*

So let's learn how to travel back in time with respect to content using Git.

Checking out is one of the processes that helps you jump to and fro between the changes that you have made in any single file or the entire subset of files that you have in your repository at the time you committed.

You can go back to a commit that you have made earlier to view the contents of a single file or group of files and return to the latest version of the same file with the latest changes – all in an instant.

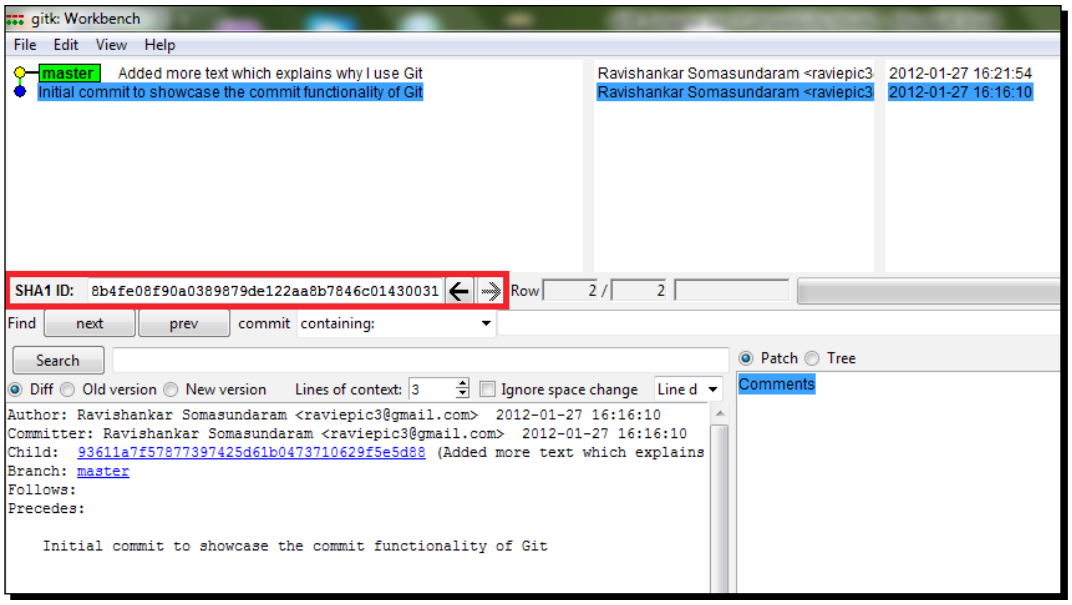
How good is that?

There are several things you can do other than just viewing the file in an earlier commit, which we will discuss in later chapters under the topic called branching.

Having learned the theory behind it, let's put it in action.

Time for action – checking out using GUI mode

1. Select the **Repository** menu and then the **Visualize All Branch History** option in the opened **Git Gui** screen to open **gitk**; you will get a screen like the following:



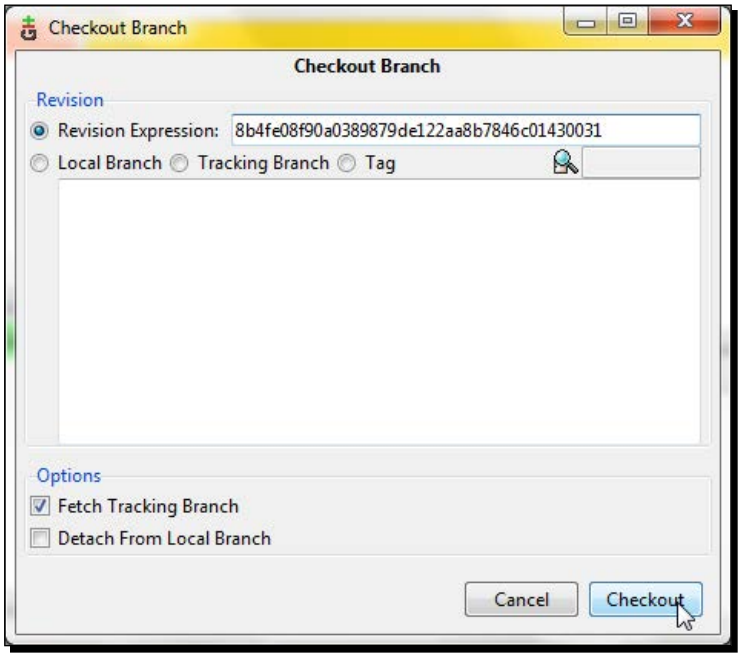
Gitk is a powerful graphical repository browser allowing us to perform various kinds of operations such as visualizing the repository, tagging, resetting, and so on.

Again, don't worry about the overwhelming information on the screen; we shall get there step-by-step.

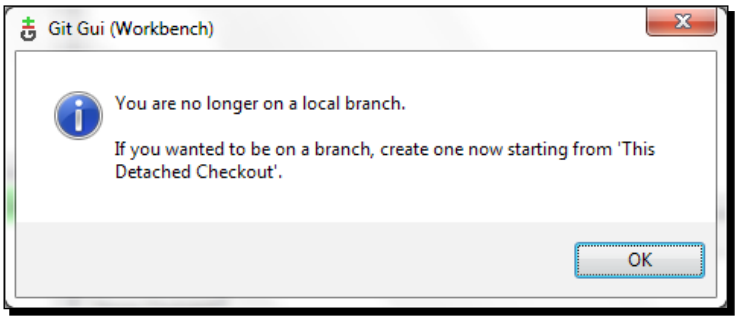
For now let's focus on the top-left pane, which shows a pathway in which the colored circles indicate the commits you have made; alongside the circles are your comments.

And directly beneath it is a field called **SHA1 ID**, which shows you the commit ID for the commit that you have selected above. As we discussed earlier we will use this commit ID to identify a particular commit to travel back in time.

2. Select our first commit, which says **Initial commit to showcase the commit functionality of Git**, to get its commit ID displayed in the **SHA1 ID** field and copy the ID (by double-clicking to select the entire line's content and pressing *Ctrl* + *C* to copy it).
3. Switch to **Git Gui** and select **Branch | Checkout** to open the checkout operation window (alternatively you can press *Ctrl* + *O*). Paste the SHA1 ID that you have copied into the **Revision Expression** textbox and click on the **Checkout** button, as shown in the following screenshot:



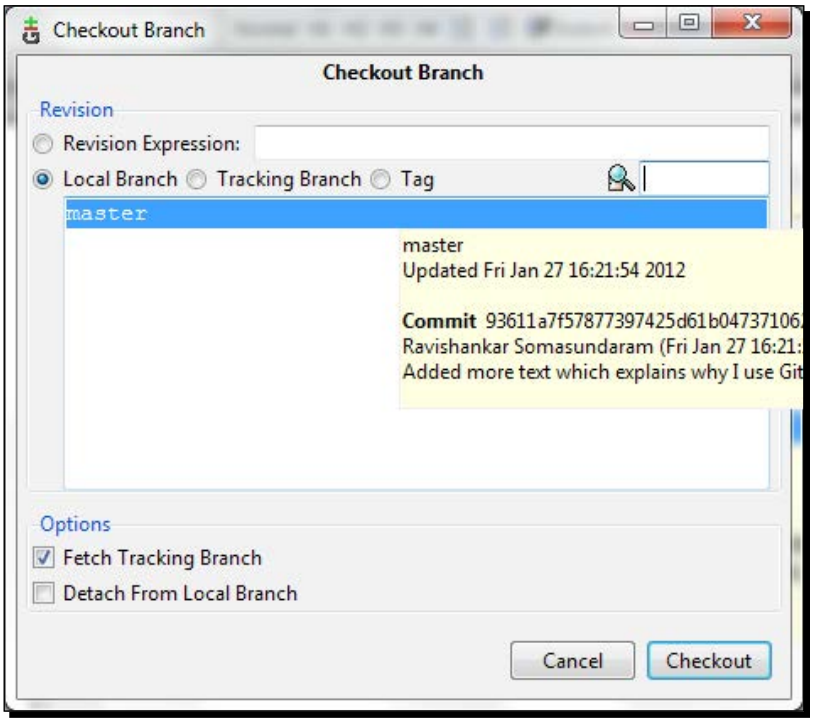
4. Click on the **OK** button on the dialog window that appears (we will discuss more about the term detached checkout in later chapters under the branching topic).



What just happened?

You have successfully travelled back in time. If we open our document now, we can see the content we had initially created in the document.

At any given point of time you can revert to your latest changes by selecting **Branch** | **Checkout** | **Localbranch**; ensure **master** is selected, and click on the **Checkout** button.



As you can see, you have jumped back to your contents with the latest changes.

Yeah, awesome, isn't it?

Time for action – checking out using CLI mode

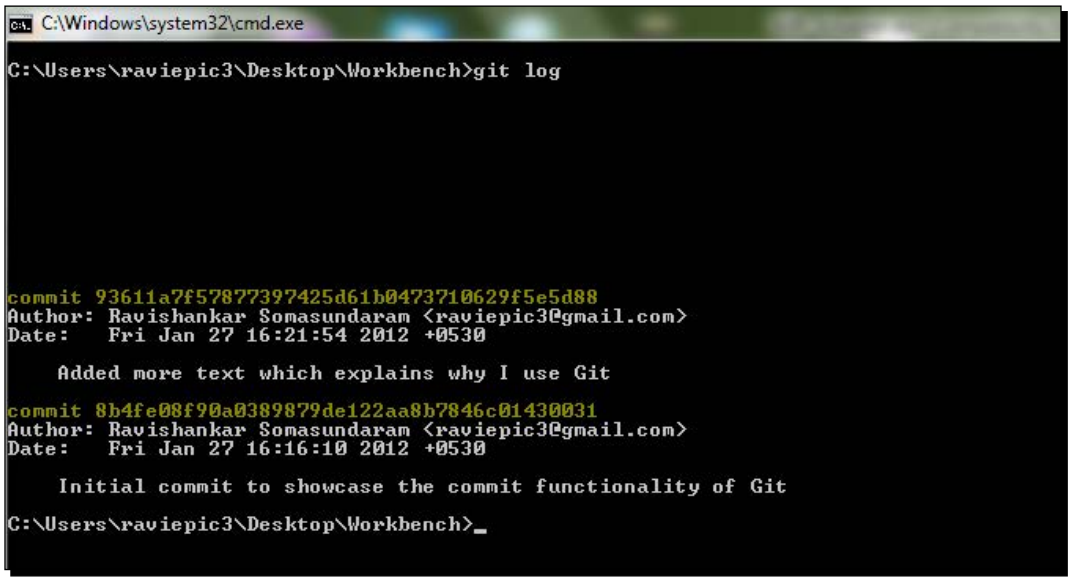
1. Let's learn two more commands to add to your Git lingo.

   `Git log`

   `Git checkout ___commit_id___`

   `Git log` is for showing the history of a repository; it gives us information such as commit ID, author, date, and the commit comment given by us.

   We need the commit ID for use later.



   Don't worry about memorizing a sequence of 40 characters. Our magic wand, Git, does the hard work of filling in the remaining characters for you to identify a commit if you supply it with the first five characters.

2. Let's see it in action.

```
commit 93611a7f57877397425d61b0473710629f5e5d88
Author: Ravishankar Somasundaram <raviepic3@gmail.com>
Date:   Fri Jan 27 16:21:54 2012 +0530

    Added more text which explains why I use Git

commit 8b4fe08f90a0389879de122aa8b7846c01430031
Author: Ravishankar Somasundaram <raviepic3@gmail.com>
Date:   Fri Jan 27 16:16:10 2012 +0530

    Initial commit to showcase the commit functionality of Git

C:\Users\raviepic3\Desktop\Workbench>
C:\Users\raviepic3\Desktop\Workbench>
C:\Users\raviepic3\Desktop\Workbench>
C:\Users\raviepic3\Desktop\Workbench>git checkout 8b4fe
Note: checking out '8b4fe'.

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 8b4fe08... Initial commit to showcase the commit functionality of Git

C:\Users\raviepic3\Desktop\Workbench>
```

Now you have travelled back to a previous commit, and your files will contain the contents of the previous commit. You can view the contents of the file now.

When you have checked back to a previous commit you are hanging in the air; any changes to your files now will be lost once you go back to the master. We'll see how to handle this in later chapters with a concept called branching.

3. To return to the latest changes run `git checkout master`; this will bring you to the latest changes.

```
C:\Users\raviepic3\Desktop\Workbench>git checkout master
Previous HEAD position was 8b4fe08... Initial commit to showcase the commit functionality of Git
Switched to branch 'master'

C:\Users\raviepic3\Desktop\Workbench>_
```

If you see a message similar to the one in the previous screenshot, you have returned to your latest changes. Again, you can view the contents of the file.

Resetting

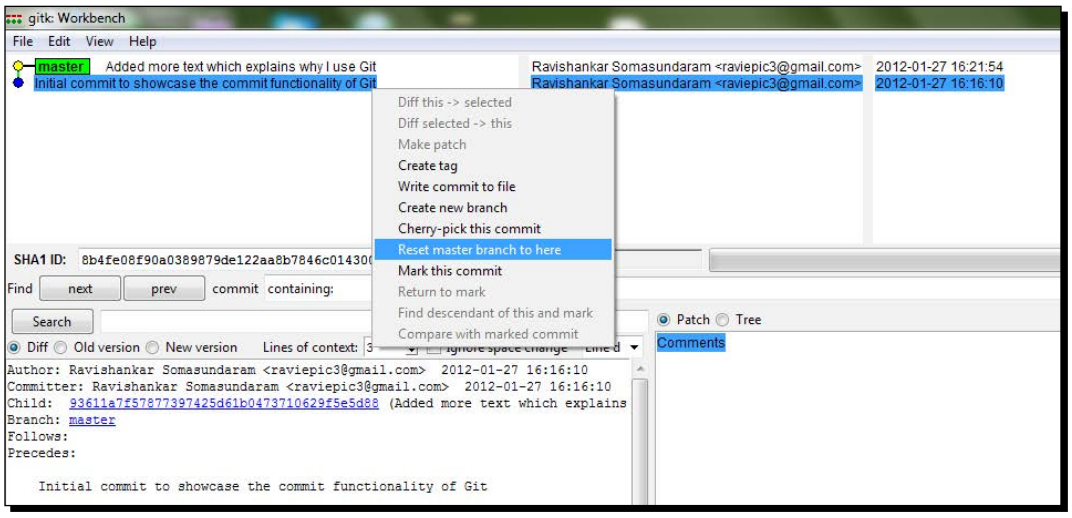
Unlike the checkout function that we learned previously, resetting is a permanent travel back in time with respect to the content. There are three types of resetting.

- Soft
- Hard
- Mixed

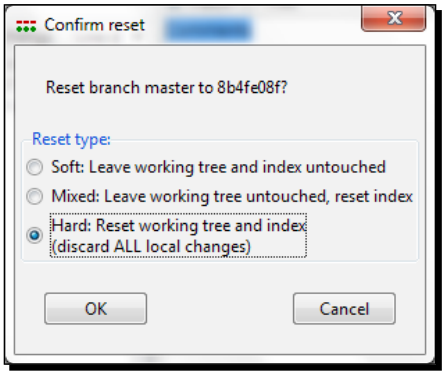
Our aim of ignoring all the changes made after a particular commit can be achieved only by performing a **hard reset**, so we will learn about the hard type alone in this chapter.

Time for action – reset using GUI mode

1. Select the **Repository** menu and then the **Visualize All Branch History** option on the opened **Git Gui** screen to open Gitk.
2. On the top-left panel you can see how your repository history is shaping up. Right now it's linear with two commits. Now right-click on the first commit, which has the commit message **Initial commit to showcase the commit functionality of Git**, and choose the **Reset master branch to here** option, as shown in the following screenshot:



3. You will get a confirmation dialog box with three types of reset options as discussed earlier; let's select **Hard** and click on the **OK** button, as shown in the following screenshot:

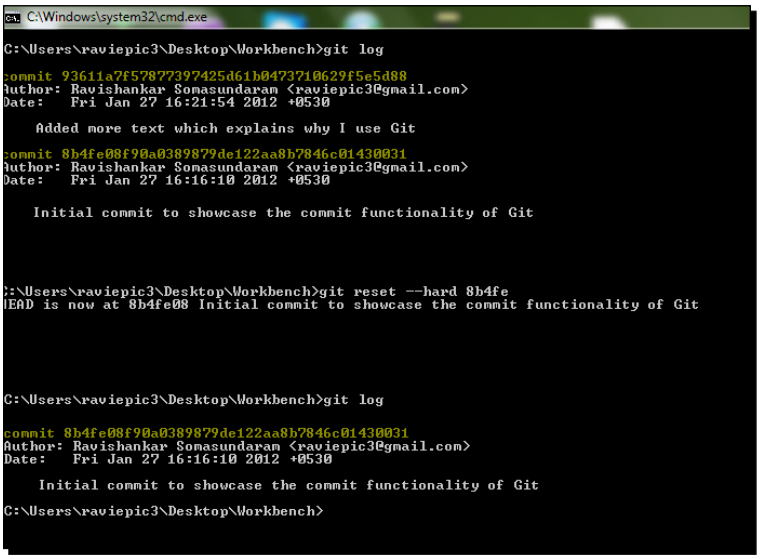


4. Gitk should automatically reload to show you the altered history of our repository. If it does not reload by itself we can manually do it by selecting the **File** | **Reload** option or pressing *Ctrl + F5*.

Time for action – reset using CLI mode

Resetting can be done by using the following commands in the CLI mode:

```
git log
git reset --hard 8b4fe
```



`Git log` is used to get to know the commit ID of the particular commit that you want to reset and the command `git reset --hard your_commitid` is to convey to Git that you want to reset all changes that have happened after the commit mentioned by its ID.

What just happened?

Congratulations! We have successfully reset our repository to an earlier state permanently. You can verify this by checking the content of your files and logs of your repository.

Git help

Git is a continuous learning platform. No matter how good you are with it already, the chances are you will learn something new every time you use it because there are multiple ways of doing things. Any command you will need to get started with Git CLI to perform basic operations always has the following pattern: `git operation_keyword parameters and/or values`.

When we say that almost all operations are local/offline in Git, we mean it!

Git has a built-in help module that can help you whenever you are unsure about the usage of a specific command or even the command itself. You can immediately refer to the built-in documentation by using the following commands:

- `git help` to get a list of command-line parameters and most commonly used operation keywords with description
- `git help operation_keyword` to get a complete reference sheet of that particular operation keyword opened in your default browser

Have a go hero – try out the help module

Try listing out the commonly used Git commands, pick one command, and try opening up the helper page for it.

Summary

We have learned how to do the following in both the GUI and CLI modes:

- Initiate a repository
- Configure Git
- Add files to our repository
- Ignore unwanted files being added to our repository
- Commit the new files/changes in existing files
- Check out to previous commits in case we need to refer old data
- Reset the repository to permanently travel back to an earlier recorded state
- Use the built-in help modules

Very soon you will learn how to do the following and much more:

- Maintain multiple environments and switch between them as though they are logged into multiple user accounts
- Continue making changes from a previous commit, thereby maintaining multiple routes (technically called branches) from one source

4

Split the Load – Distributed Working with Git

So far we have worked entirely with a single local repository. It's high time to go one step further and explore the most highlighted feature of Git in earlier chapters: distributed working with Git.

In this chapter you will learn the essentials for collaborative development:

- How to share your files/projects over the Internet and intranet
- Various concepts such as:
  - Git clone
  - Git fetch
  - Git merge
  - Git pull
  - Git push
  - Git remote

These concepts are involved in sharing your files over the Internet and intranet *progressively* and *continuously*.

Why share your files

Let's take the same computer gaming analogy which has helped us to understand the basics of Git.

Scenario 1: single player

Think of your favorite game that allows you to save the state of the game in your system at any given point of time and resume it later on. Now let's consider a situation where you are in some remote location with access to a computer and want to resume the game, but are not able to because the saved game file is not accessible from that system.

Apply the same situation to your data files. On average we spend most of our time of the day at two to three different locations; think about how productive it might be to *continue our work across systems* without having to start from scratch on each new system we lay our hands on.

Scenario 2: multiple players – one at a time

Think of your favorite adventure game that has multiple levels. Consider a scenario where you are stuck in a level without knowing how to proceed forward. After desperate attempts, which ended in vain, you suddenly realize that your friend is an expert on that level, and you want to use your friend's help. So you quickly share the last saved state of the game file with him so he can finish that level for you, save the state, and push the file back to you, which will enable you to continue the game.

The same situation can apply to you when you are working with datafiles, especially when you are working as a team where different people take care of different parts of a bigger task to produce a single result. Another possibility might be that you want the domain experts to handle specific portions of the work, and so on.

When it comes to sharing files over the network there can be only two modes.

- Internet
- Intranet

The appropriate method is employed based on proximity.

Kid's play – push and pull for a remote source

Before getting into the concept of a **distributed file system** as in *Scenario 1* or **collaborative development** as in *Scenario 2*, it's time we add five more entries to our Git lingo, namely:

- Git clone
- Git fetch
- Git merge
- Git push
- Git remote

Let's quickly understand what these terms mean and where can they be put to use.

Cloning ain't banned here

Yes, we are talking about Git's **clone** functionality. Git clone is used when we need an exact replica or a copy of an existing repository along with its history.

So a question may arise as to how all the cloned repositories maintain sync with each other.

Well, the answer to that lies in the remaining four Git commands, which are listed previously after `git clone`, namely `git fetch`, `git merge`, `git push`, and `git remote`.

- **Git fetch**: This command is used to fetch the changes from source to destination.
- **Git merge**: Merge is used to combine two workspaces (technically called branches) into one. It is frequently used to combine the current user's workspace with the one from the remote user, after fetching the changes from the remote source.

Git pull: Executing `git pull` will internally execute `git fetch` followed by `git merge`. Hence, it is used as an alternate to fetch plus merge.

- **Git push**: This command is used to push our contents from source to destination.
- **Git remote**: This command is used to manage one's source and destination. It says where and how you can share your work with others and vice versa.

Any operation which enables data sharing makes use of remote connections, which are established by `git remote`. Here, `git fetch`, `git push`, and `git pull` make use of the remote connections established by `git remote`.

Now that we have a heads up on a few concepts, let's see how they are put to use.

Scenario 1: solution

We shall learn how to utilize Git to serve you in the case of *Scenario 1* as mentioned previously.

Going public – sharing over the Internet

There are several online Git hosting providers available for use with different pricing models. Broadly speaking a few of them offer a free service for limited functional use and ask you to pay for additional usage; a few others offer full functional access for a limited time and ask you to choose a payment plan to continue, and there are a few others who combine a bit of both.

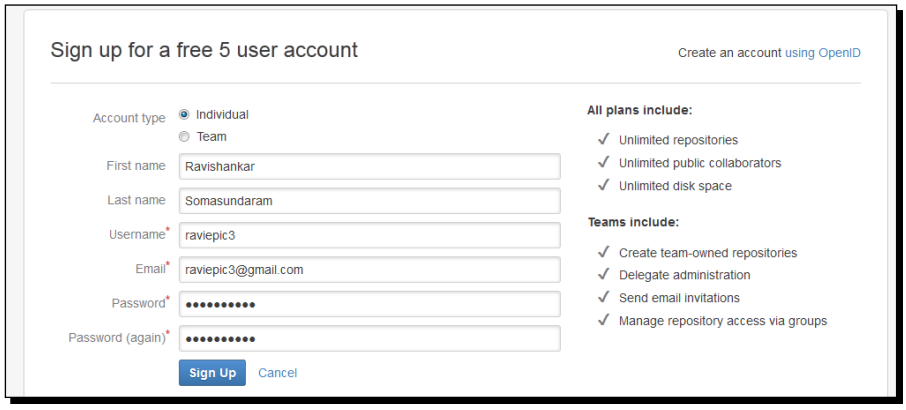
I am going to choose Bitbucket, a reliable service provider belonging to the third category, from now on to take you through the concepts related to sharing over the Internet.

Bitbucket is a product of Atlassian, which currently offers free, unlimited public and private repositories with the only restriction being the number of users with whom your private repositories are shared. This means we can share our private repository over the Internet with five people who have read and write access to it, for free.

There are a few other competitive products such as GitHub, Codaset, and others. We chose Bitbucket as it provides private repositories for free.

A bit of Bitbucket

Let's do a quick signup for their services; open up your browser, go to `http://bitbucket.org`, click on the **Pricing and Signup** button, and then click on the first **Sign up** button under the **free** quota. It then leads you to the registration page where you choose an individual account type for now (it is possible to have a entire team use a single account) and choose your username and password for the **username** and **password** fields respectively, and enter your active e-mail address in the **Email address** field, whereas your **First name** and **Last name** are optional fields as shown in the following screenshot:



After completing the procedures you can expect a confirmation e-mail from Bitbucket to validate your e-mail address.

As an alternative to going through this entire sign up process, you can also sign in with your OpenID if you have one.

The beauty of Bitbucket is that it has keyboard shortcuts for almost all actions like in Gmail. Similar to Gmail you can press *Shift* + *?* to see the list of shortcuts available. The following is a tabulation of frequently used shortcut keys for your reference:

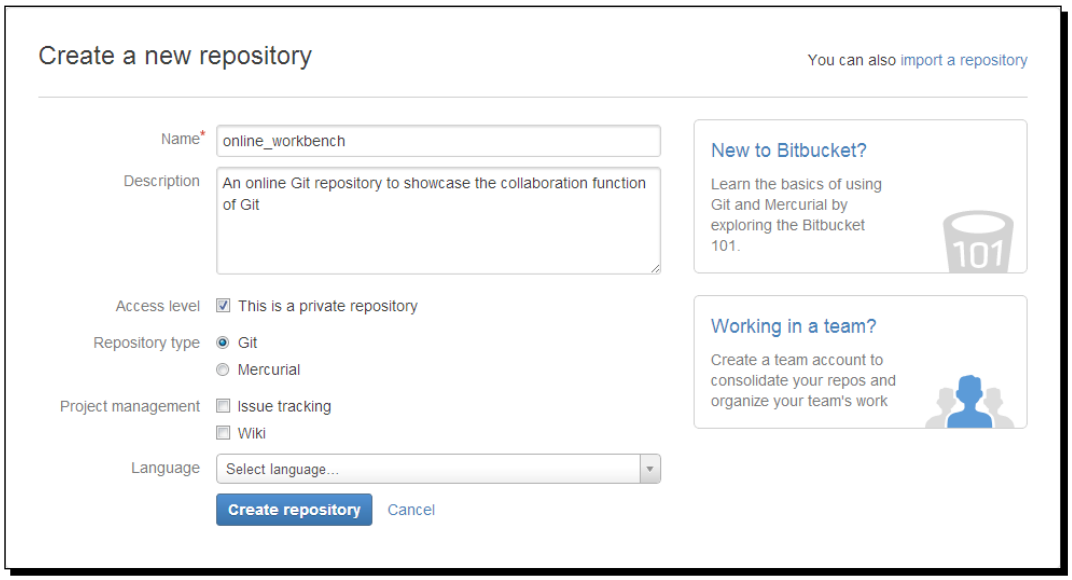
Key combination	Action
?	Display keyboard shortcuts help.
c + *r*	Create repository.
i + *r*	Import repository.
g + *d*	Go to dashboard.
g + *a*	Go to account settings.
g + *i*	Go to inbox.
/	Focus on the site search. Puts your cursor on the site field.
Esc	Dismiss the help dialog or remove the focus from a form field.
u	Go back up the stack you just went down with the shortcuts. Like the back button in a browser, this takes you back through the Bitbucket pages you just paged through.

Let's start our journey by creating a new repository in your account. You can either press *c* + *r*, or click on the **Create repository** option from the **Repositories** menu at the top, or simply click on the **Create a repository** link from the repositories block at your right side and this will take you to the page that will guide you in creating a new repository/repo (repo is a widely used shortform for repository).

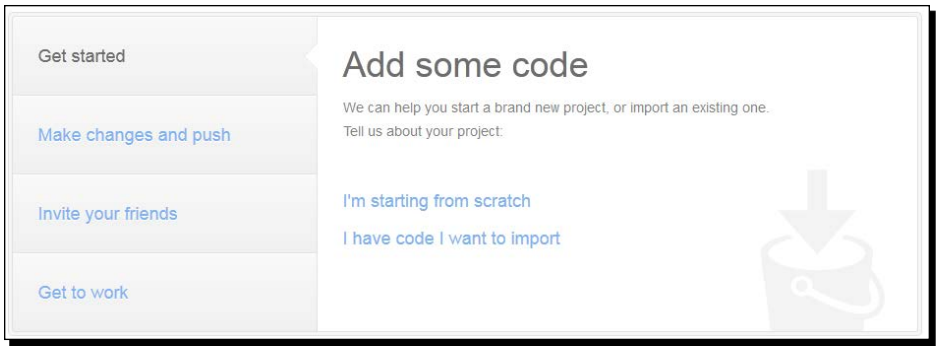
Field name	Value	Reason
Name	`online_workbench`	We are going to import the same `Workbench` repository on our desktop to this online portal.
Description	`An online Git repository to showcase the collaboration function of Git`	This is a brief description of your repository. You can have your own description here that best describes the purpose of the repository.

Field name	Value	Reason
Access level	`Checked`	A private repo is only visible to you and those you give access to (more about this later). If this box is unchecked, everyone can see your repo.
Repository type	`Git`	Bitbucket supports both Git and Mercurial version control systems. As we are going to import a Git repository, let's select that.

Enter the values in the fields as shown in the following screenshot:



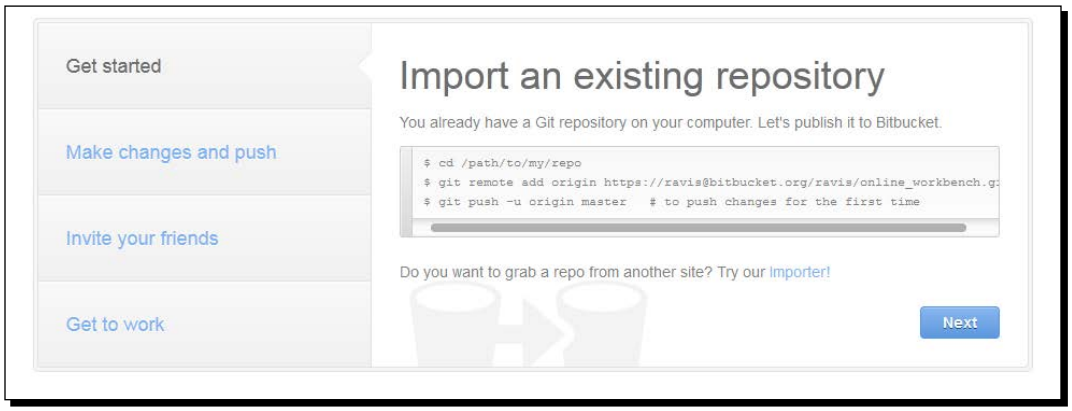
Click on the **Create repository** button to complete the repository creation process. Now that you have an empty repository, Bitbucket prompts us for immediate action as shown in the following screenshot:



Here we have two different startup options.

- Create a new directory in our machine, initialize it as a repository, and link that to the remote Bitbucket repository that we just created, which is represented by the **I'm starting from scratch** link
- Skip to the later part, which is linking our existing repository to the remote Bitbucket repository and pushing our contents to it, which is represented by the option **I have code I want to import** link

As we already have our repository created, let's select the second option which leads us to a screen as follows:



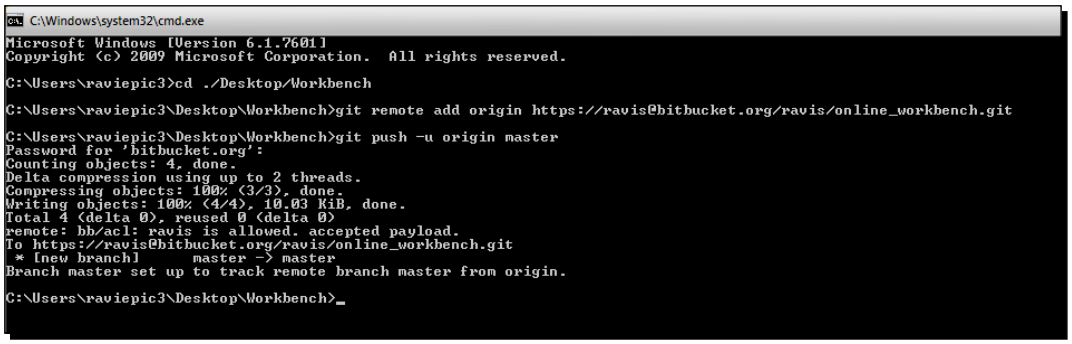
We are getting to the core part now. Shown in the screen are instructions for CLI users to link the `Workbench` repository from the desktop to the `online_workbench` repository in Bitbucket.

Time for action – adding a remote origin using CLI mode

Linking or adding a remote origin to your repository (yet another addition to your Git lingo) is a simple process. Fire up your command-line interface and enter the following commands:

```
cd /path/to/your/Workbench/repo
git remote add origin https://your_bitbucket_repo_identity_here/online_workbench.git
git push -u origin master
```

After the execution of the `git push` command you will be prompted for your Bitbucket account password to complete the process as follows:



If you see a similar message in your window, the linkage cum transfer was successful.

What just happened?

We just created a remote link for our `Workbench` repository with the `online_workbench` repository and pushed our files to it making them available online, thus opening the door for a distributed file system, using the CLI mode.

`git remote add` is the command to add a Git repository identified by its path to your present repository's configuration file so that your changes in one repository get tracked in another. Let's just say **origin** is nothing but an alias for the path representing the remote repository.

The parameter `–u origin master`, which is used with `git pull` is to default the repositories' push and pull operations to that specified remote branch.

If `–u` is not used initially, then for each and every pull and push request we need to specify `origin master` along with the request. Now it's enough for us to say `git push` for pushing and `git pull` for pulling.

This means you can continue your work from anywhere if you have access to a computer with Git and your application software installed on it (it's Microsoft Word in this case as we are dealing with a Word document).

Time for action – resume your work from anywhere using CLI mode

Now let's enter the second phase, where we would like to resume our work from a remote machine.

There are only three stages involved here.

1. Clone the repository from the server.

   ```
   git clone https://raviepic3@bitbucket.org/raviepic3/online_
   workbench.git /path/where/you/would/like/the/clone_to_be
   ```

2. Make your changes to the files needed.
3. Add/stage the modifications made in files, commit, and push.

   ```
   git add *
   git commit –m 'Your commit message'
   git pull
   git push
   ```

As an alternative to `git pull`, we can also use `git fetch` followed by `git merge @{u}`.

What just happened?

We just practiced a working solution for maximizing productivity by effectively handling situations as described in *Scenario 1*.

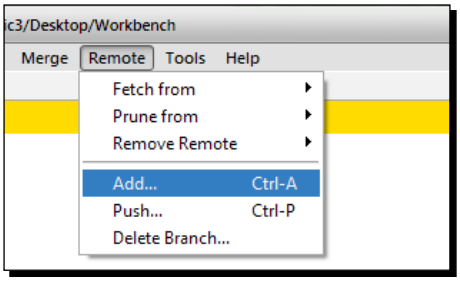
`Git add *` stages/adds all your changes, which is confirmed and recorded by the `git commit` command. `Git pull` is used to check whether there are unsynced updates in the server; if present, they are synced appropriately followed by `git push`, which updates the server's files with the changes that you have made and committed in your local repository.

You might happen to think why we are doing `git pull` before `git push` when our sole intention was to push the updated files to the server. Wonderful question – hold that thought right there; you will get to know more about it when we discuss the concepts of branching.

Time for action – adding a remote origin using GUI mode

Linking or adding a remote origin to our `Workbench` repository present in our desktop and syncing the contents using the Git GUI is performed as follows:

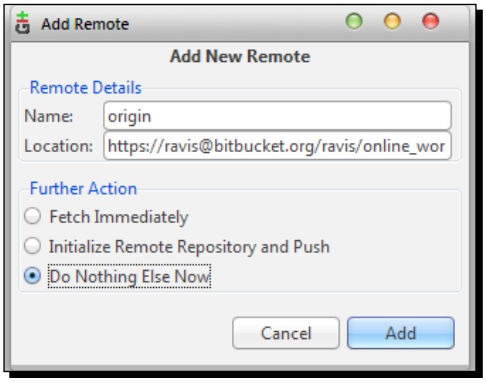
1. Open up your Git GUI window for our `Workbench` repository from the desktop.
2. Click on the **Add** option from the **Remote** menu in your GUI window.



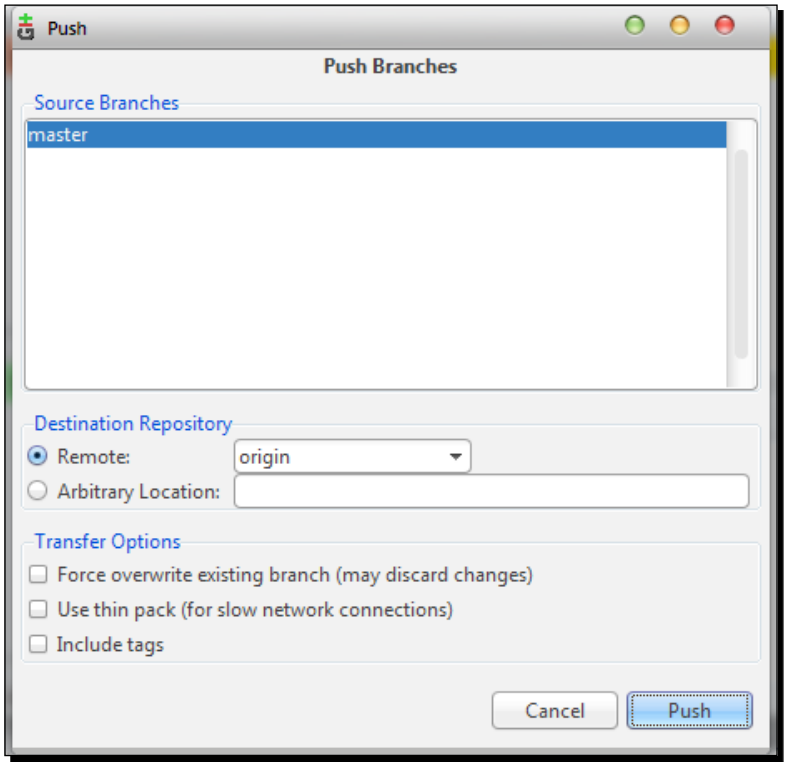
3. This opens up the **Add New Remote** window where we enter the following details:

Field name	Value
Name	`origin`
Location	`https://your_bitbucket_repo_identity_here/online_workbench.git`
Further Action	**Do Nothing Else Now**

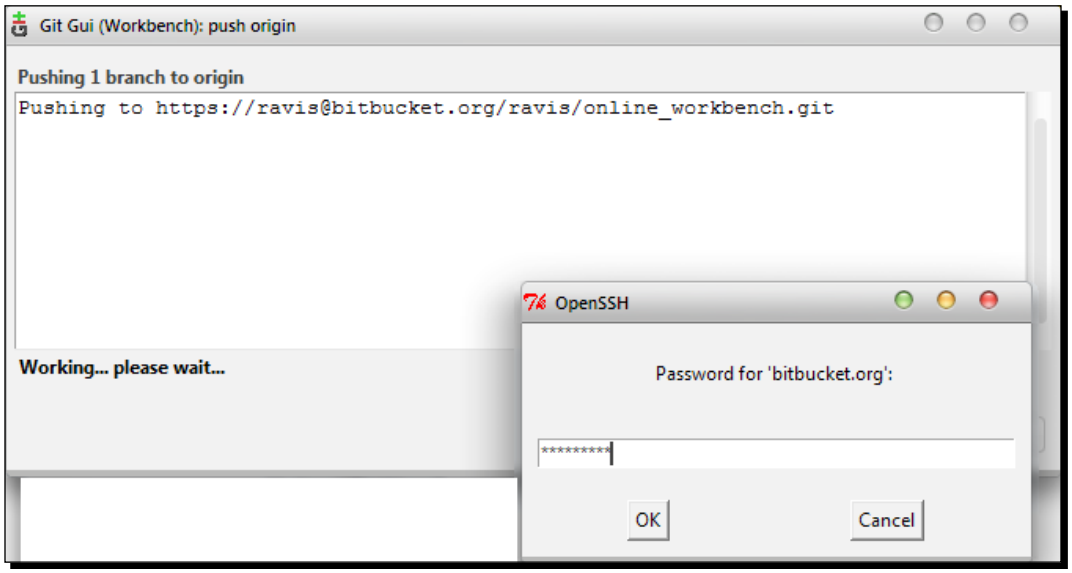
4. This is shown in the following screenshot; click on the **Add** button:



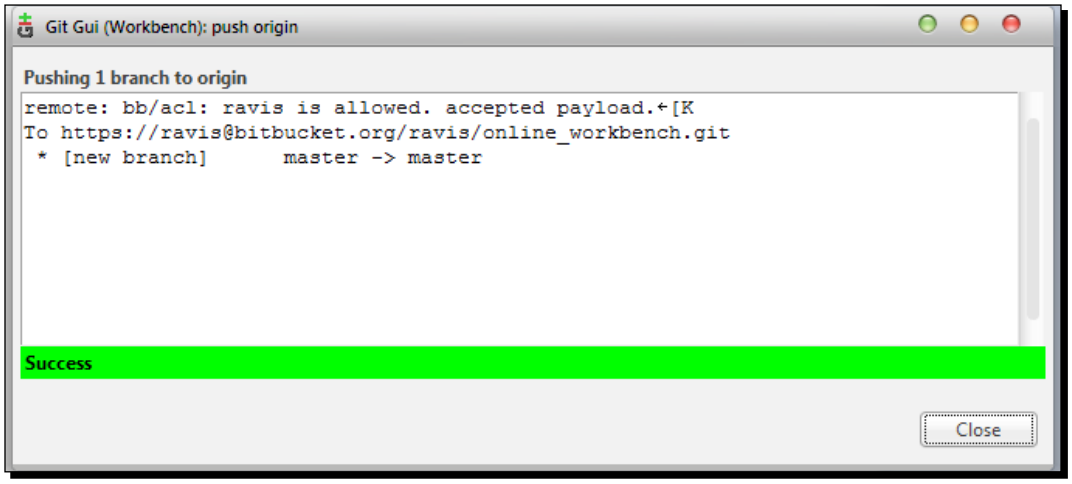
5. We have now successfully added a remote to our `Workbench` repository.
6. To push our code to the `online_workbench` repository, go to the same **Remote** menu and select the option **Push**, which will lead you to the **Push Branches** window as shown in the following screenshot:



7. By default, **master** will be selected under **Source Branches**, and **origin** will be selected in the listbox for the **Remote** option under **Destination Repository**. Leave it as it is, click on the **Push** button, and wait for some time; it should lead you to a screen where you will be prompted for your Bitbucket account password to proceed as shown in the following screenshot:



8. Upon successful authentication, your content will be synced with the `online_workbench` repository, which can be understood from the following screenshot:

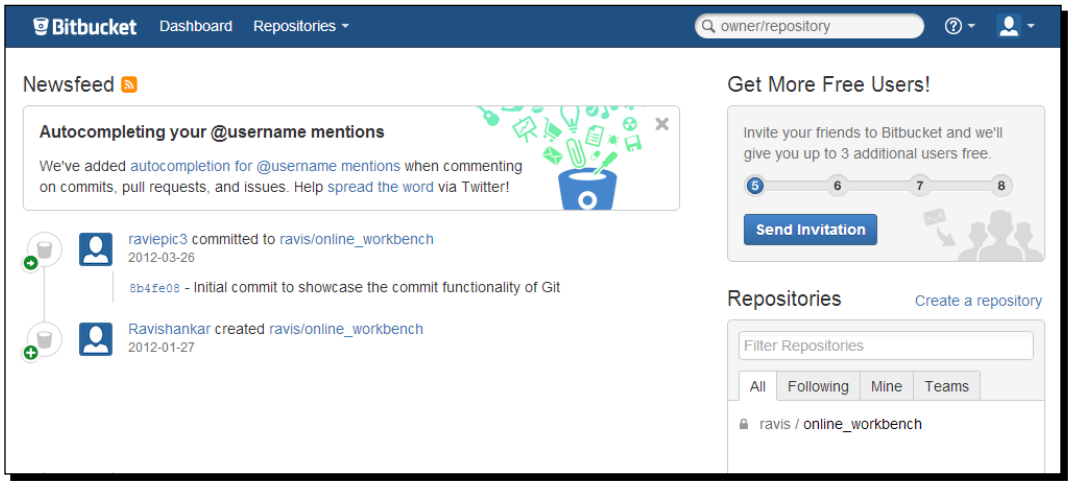


It says that the master branch of your local `Workbench` repository is synced with the master branch of the `online_workbench` repository (more on branches in later chapters).

What just happened?

We just created a remote link for our `Workbench` repository with the `online_workbench` repository and pushed our files to it making them available online, thus opening the door for a distributed file system, using GUI mode.

Now if you open your Bitbucket account in your browser you will see history updated on your dashboard as shown in the following screenshot:



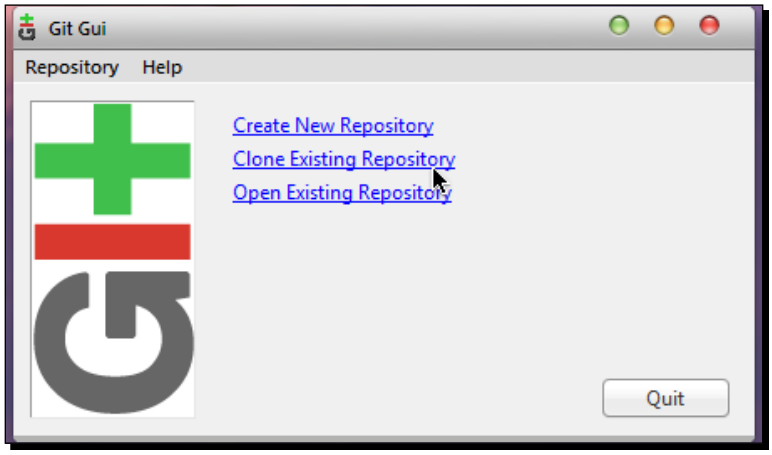
This means you can continue your work from anywhere if you have access to a computer with Git and your application software installed on it (it's Microsoft Word in this case as we are dealing with a Word document).

Go ahead and browse through the different tabs available to make yourself comfortable with it. Once you are done with it, let's move to the next half to see how we can resume our work form distributed locations.

Time for action – resume your work from anywhere using GUI mode

Here, we are reaping the benefits for what we did earlier by creating an online repository, remoting to it, and syncing our local files to the online one. Resuming your work on any machine you lay your hands on is an easy three-phase process.

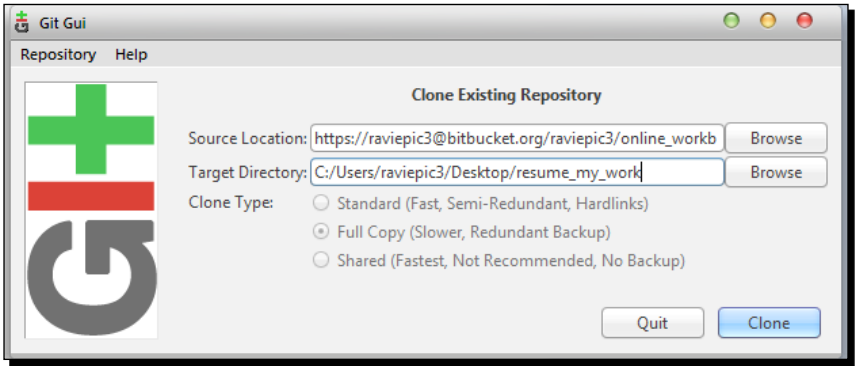
1. Clone the repository from the server.
    i. Open Git GUI and select the **Clone Existing Repository** option as shown in the following screenshot:



    ii. This leads you to the respective window where you are prompted for **Source Location** and **Target Directory** where you enter the values as follows:

Field name	Value
Source location	`https://your_name@bitbucket.org/username/online_workbench.git`
Target Directory	`/Path/where/you/want/to/have/the_cloned_repository_for_ease_of_work`

This is shown in the following screenshot; click on the **Clone** button:

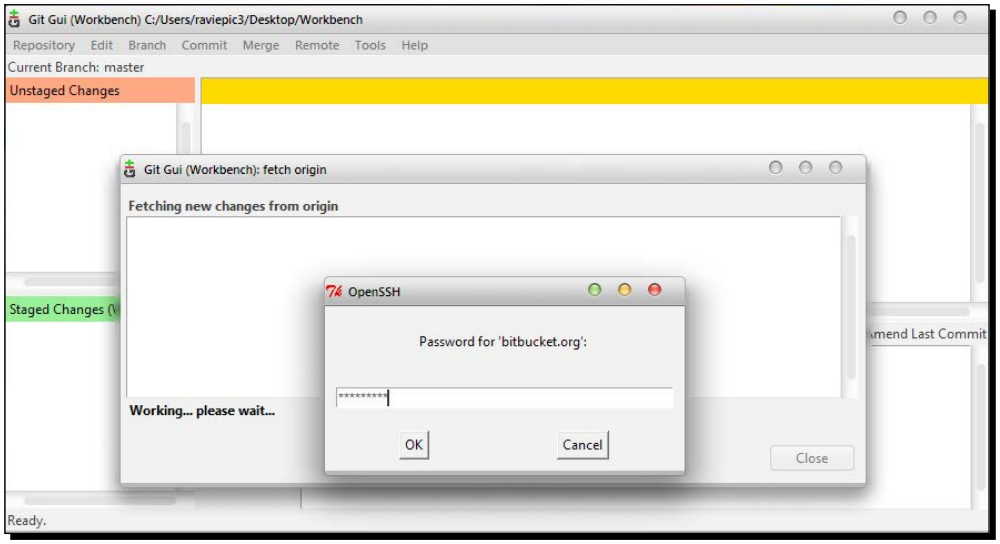


iii. Once the clone process starts you will be prompted for your Bitbucket account password. Upon successful authentication, you will have a cloned repository with the files with which you can resume your work.

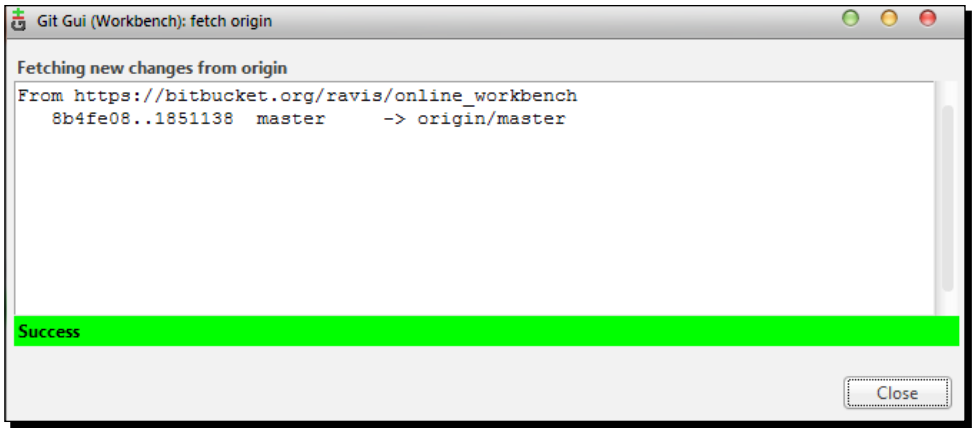
2. Make your changes to the files as needed.

3. Add/stage the modifications made in files, commit, fetch, merge, and push.

i. We already know how to add/stage the modifications made to files and commit them to the repository. So let's start from fetch now. To perform a fetch operation, go to the **Remote** | **Fetch from** | **Origin** menu option. This should bring you the remote fetch window, which will prompt you for your Bitbucket account password as shown in the following screenshot:

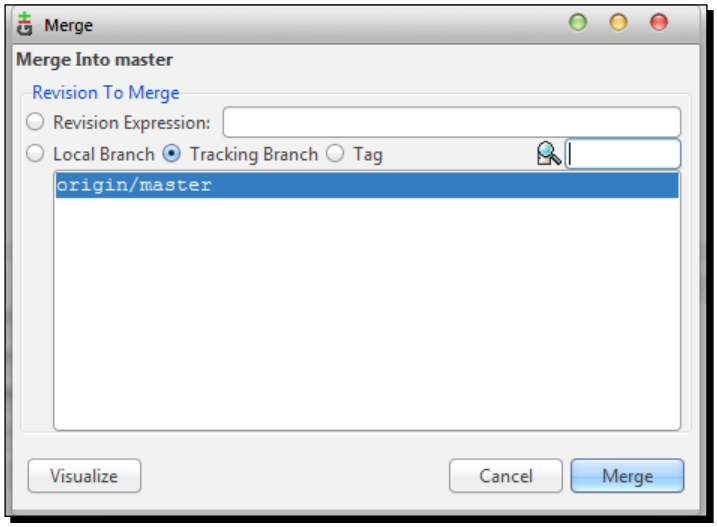


Upon entering the correct password and a successful authentication, if there are any new changes to the files in the server that are not updated in your local repository, those changes are synced.



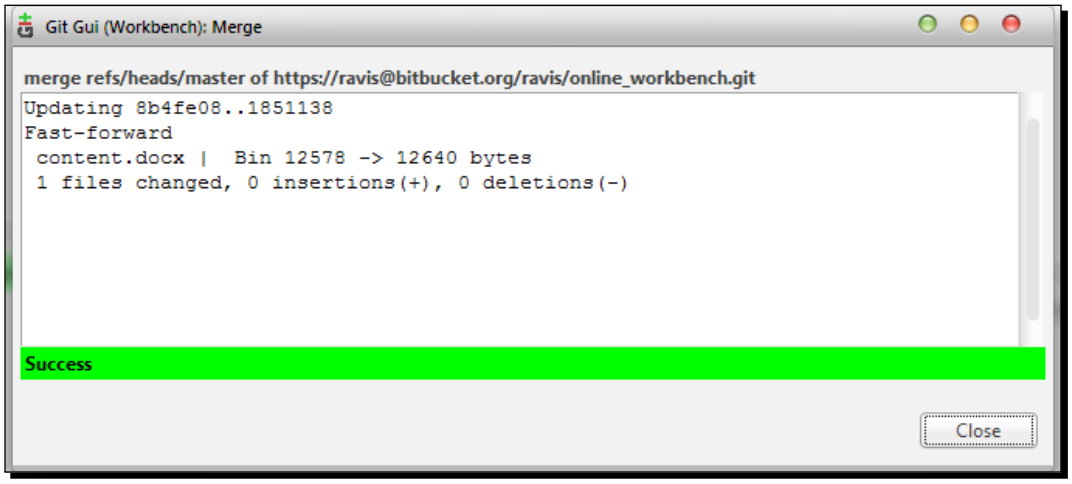
The previous screenshot shows you the sync process and the status of the sync. Upon success we can close the window and proceed to merge these two workspaces.

ii. Merging two workspaces, namely local master (your local workspace, which you have been using to make changes) and remote master (the workspace which is present in the server), is performed by selecting the **Merge** | **Local merge** menu option. This opens up a local merge window as shown in the following screenshot:



The default selected option would be **origin/master**; leave it as it is, and click on the **Merge** button.

iii. If there are no conflicts in merging you should see a success message like the one shown in the following screenshot:



This marks that you have successfully down-synced the contents present in the server with yours. Now let's up-sync your content with the one present in the servers by using the `git push` functionality, which can be accessed from the **Remote** | **Push** menu option.

What just happened?

We just practiced a working solution for maximizing productivity by effectively handling situations as described in *Scenario 1*, using Git GUI.

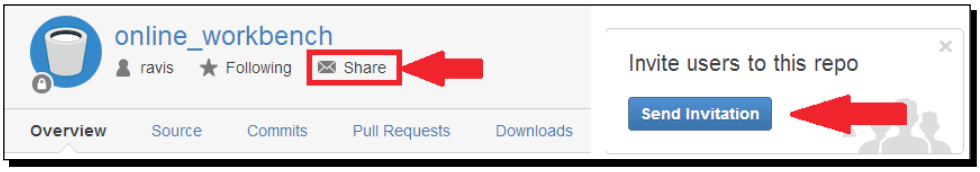
Scenario 2: solution

Handling *Scenario 2* is very easy now that we know how we handled *Scenario 1*. The only addition to *Scenario 2* when compared to the former is the involvement of multiple people to the same repository.

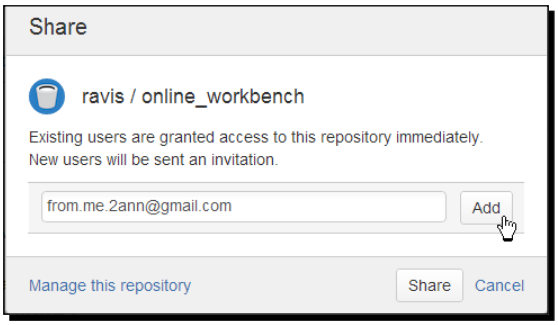
Inviting users to your Bitbucket repository

Inviting your friend to access your game file so that he can finish that level for you is an easy two-step process from your side as follows:

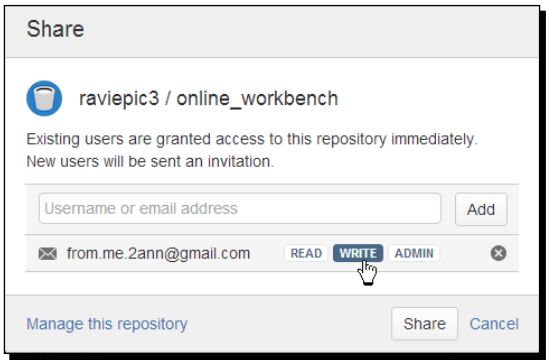
1. From your repository homepage, click on the **Share** icon or the **invite** button as shown in the following screenshot:



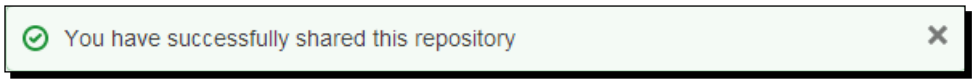
2. This will prompt you to enter the details about the user whom you wanted to invite or share your repository with. If it's an existing user you can enter his/her username and if it's a new user you can enter his/her e-mail ID and click on the **Add** button as shown in the following screenshot:



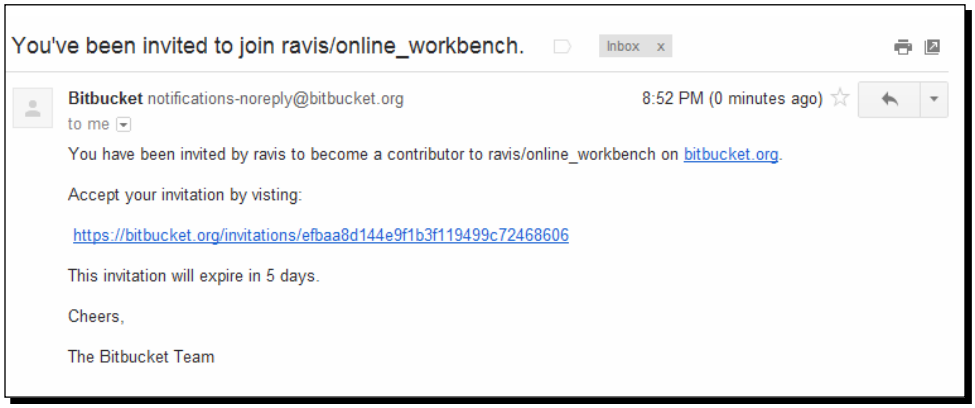
3. Now the username/e-mail ID gets added to the list and you will be prompted to specify the access level for the user that you have added. Click on the **Write** button and then click on the **Share** button as shown in the following screenshot:



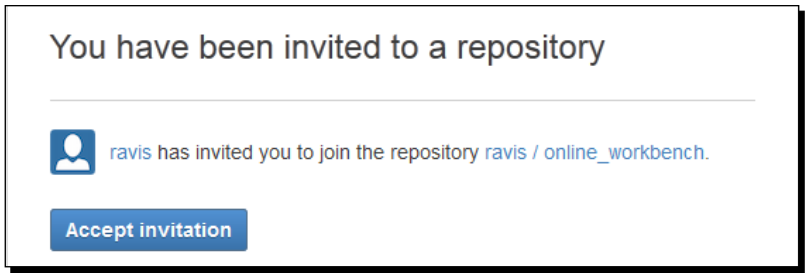
4. That's it! You will see a success message at the top of your page, as shown in the following screenshot, as an acknowledgement for sharing:



And the user you have added will receive an e-mail mentioning that you wanted to share your repository with him/her as shown in the following screenshot:

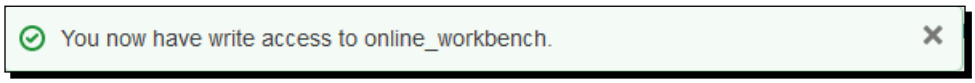


5. Upon clicking the link you will be given two options.
   - **Sign up**: If your friend is a new user of Bitbucket, he/she needs to go through the registration process as discussed earlier in this chapter. Post registration you will be routed to your dashboard.
   - **Log in with your existing username**: If your friend already has a Bitbucket account and once he/she logs in with his/her credentials, he/she will be prompted for acceptance to the shared repository as shown in the following screenshot:



When the **Accept invitation** button is clicked, the user will be taken to his/her dashboard.

And the dashboard will contain affirmation in the form of an onscreen notification as shown in the following screenshot along with an e-mail which is sent to your registered e-mail ID which contains the details of the repository that you have been given access to:



And an e-mail is sent to that user with the details about the repository which he/she has access to.

What just happened?

Voila! We have successfully practiced a working solution for handling *Scenario 2* cases effectively.

This means you can split a bigger task into smaller ones, and share those tasks and related files with others so that they can fill in their sectors to produce a common output.

Staying local – share over the intranet

There are situations where you work within a local network, like in different floors of a building, and don't want to upload your files to the web due to various reasons such as cost involved, bandwidth consumption for every put and get, security, and others.

In such cases there are several ways to handle this – the most commonly used are:

- Gitolite server
- Common shared directory with bare repositories

We shall look at procedures to create a bare repository inside a shared directory so that it can be shared within your network.

Concept of a bare repository

As soon as it is said that you need a bare repository to share your files with others, a few basic questions you might have in your mind would be:

- What is a bare repository?
- Why do we need such a thing to share the files of our repository with others?

Let's see them one by one.

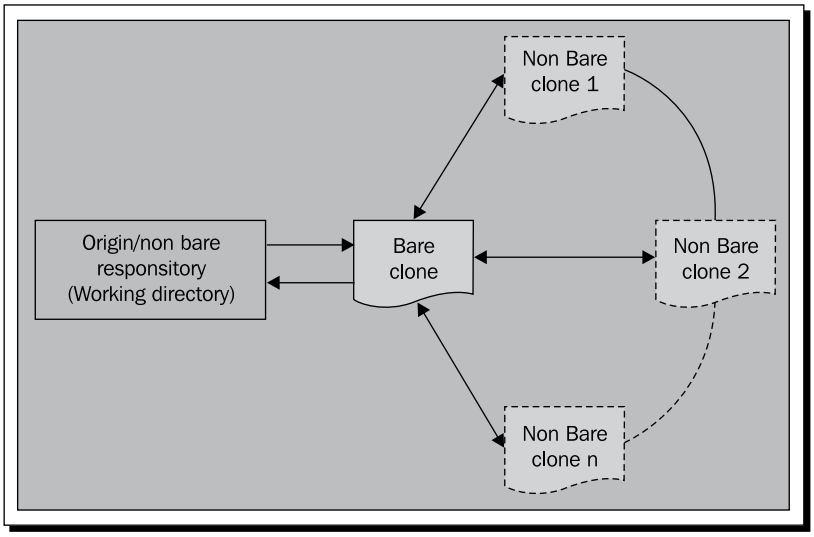
- **Bare repository**: A bare repository is the one where no working directory is present.
- **Working directory**: This is nothing but a directory with your source files, for example, `content.docx` inside the `Workbench` directory.

  The contents of the `.git` directory alone would be the contents of your entire directory if it is a bare repository.

- **Why bare repository**: Think about a situation where there are multiple people working on the same file at the same time. Now what happens when you are in the process of changing some content in the file from the repository and another person working on the same file makes his own changes and pushes it to your repository!

  The contents of your file will be altered, or the file itself may cease to exist based on the actions performed from the other end, whereas you would have the file opened for manipulation.

  It causes a great deal of confusion in handling such scenarios, so the people who have created Git have done the smart thing of avoiding such situations by implementing the bare repository concept. This bare repository acts as a middle man between all such clones and your source repository, which contains the working directory. So you cannot simply push from a clone to the source of the clone, if the source contains the working directory.



Let's create a bare repository and take a quick peek into it to understand it better.

Time for action – creating a bare repository in CLI mode

The command for creating a bare repository would be the same as the one that you used to clone a repository except for the `--bare` parameter, which makes all the difference.

```
git clone --bare C:\Users\raviepic3\Desktop\Workbench C:\generic_share\
Bare_Workbench
```

Executing the preceding code in your console should create a bare clone of our `Workbench` repository in your common shared folder called `generic_share`.

Time for action – creating a bare repository in GUI mode

Creating a bare clone from an already existing repository using GUI is an easy process. All you need to do is:

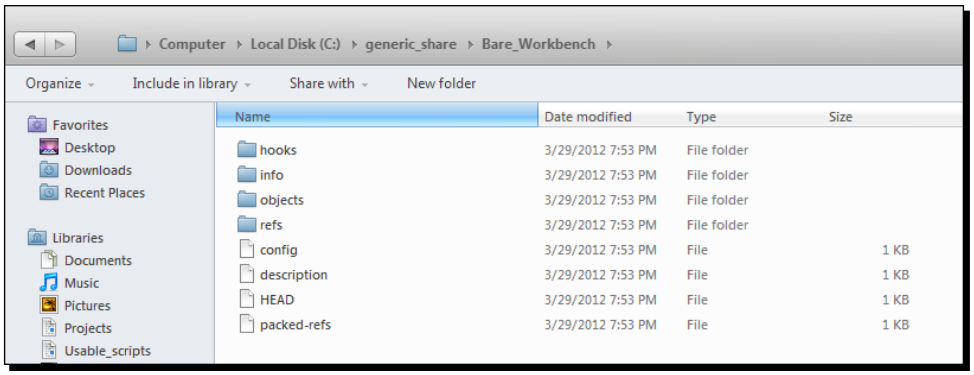
1. Copy the `.git` directory from the existing repository and paste it with a `different_name.git` (whatever name you want to give to your new bare repository) outside the repository.

   In our case we have a non bare repo called `Workbench` at `C:\Users\raviepic3\Desktop\` inside which we have `content.docx`. And now I want to create a new bare repository from this using GUI. I'll copy `C:\Users\raviepic3\Desktop\Workbench\.git` and paste it as `C:\generic_share\Bare_Workbench.git`.

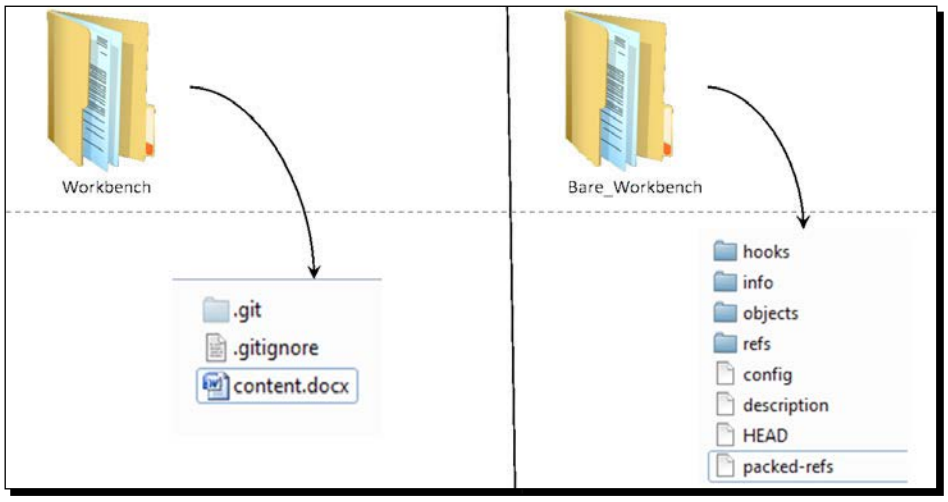
2. Open the `config` file inside `Bare_Workbench.git` with a text editor and find the line which says `bare = false` and replace the string `false` with `true`.
3. Save and exit.

What just happened?

By performing the previous actions through CLI or GUI we have just created a bare repository out of the `Workbench` repository inside a directory called `generic_share` under the name of `Bare_Workbench` whose contents are as shown in the following screenshot:



For a better understanding, a content comparison between the two repositories is shown in the following figure:



If you are in a local network, you can control who has access to the repository by controlling the visibility of the shared folder `generic_share`, the same way you control visibility of other shared folders within your network.

Summary

We have learned what is and how to:

- Clone a repository
  - Differentiate between bare and non bare repositories, their usage, and implementation
- Add a remote to a repository
- Fetch, merge, and push content to the added remote repository or a cloned repository
  - The pull operation and its alternates

Additionally, you have also learned how to:

- Share your repositories over the Internet and intranet using:
  - Git CLI
  - Git GUI
- You are also ready to get productive with the concepts that you have learned starting from day one, as you already have a Bitbucket account with which you can create and manage unlimited public and private repositories and share them with at the most five users without spending a penny on it

5

Be a Puppet Master – Learn Fancy Features to Control Git's Functions

Having laid a perfect base by learning, practicing, and validating the concepts covered so far, we are ready to learn some fancy features that will build on our base knowledge to extend the functional usage.

In this chapter you will learn concepts that will help you to perform the following with respect to the content in your repositories:

- Shortlog
- Log search
- Clean
- Tag

Why learn such fancy features

Why would you need the "S" power in *Contra* or an M4 rifle in *Counter-Strike* when you have the default weapons?

Though you will be able to achieve the ultimate goal of killing the opponent even with the basic tools given to you, usage of those specialized tools facilitate the ease of achieving the said goal, that's why.

Similarly these functions that we are about to learn are going to provide us with easy ways of controlling Git to provide data according to the various situations which you can decide where it fits in, based on your job role. And moreover, it's fun to have some ready-made tricks in your pocket to pull out when the need arises, making you the apt person for the job.

Prerequisites

To learn these concepts better, we need to have a repository where there are quite a few commits and multiple people involved in the development of the repository's content. So we shall download any repository from famous Git hosting sites like GitHub or Bitbucket itself.

Here, I have downloaded a project called cappuccino and reduced the repository to fit our purpose.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

You can also download your own from the Git hosting sites specified previously to carry out your learning.

Shortlog

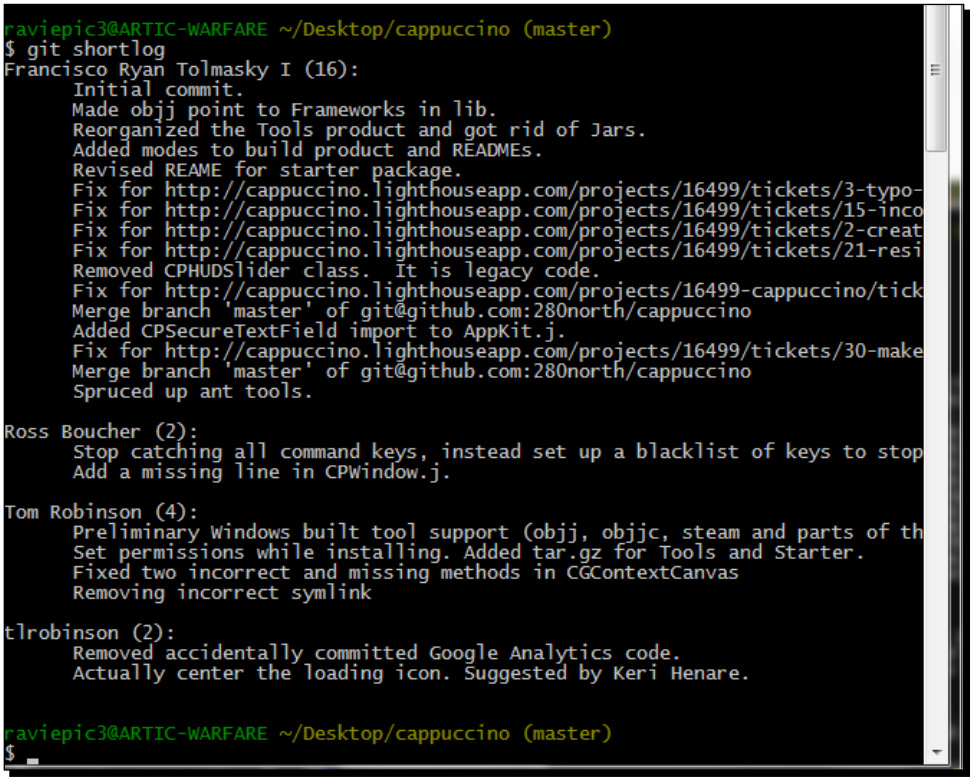
Though information is wealth, sometimes Too Much Information kills the purpose it serves. Think about the value that is associated with the **filter option** in a spreadsheet application. Shortlog is one such function to command Git to limit the information it shows when viewing logs. It arranges all the users who were involved in building the data of the repository in alphabetical order mentioning the number of commits they made along with their commit descriptions.

Time for action – getting acquainted with shortlog

Let's quickly fire up your CLI window from our `cappuccino` repository and try out the following command:

```
git shortlog
```

This should give an output as shown in the following screenshot:



What just happened?

We just listed out comments of 24 commits segregated by the authors who were responsible for those commits; the authors were arranged in alphabetical order in a single shot. Sounds compact and presentable, right?

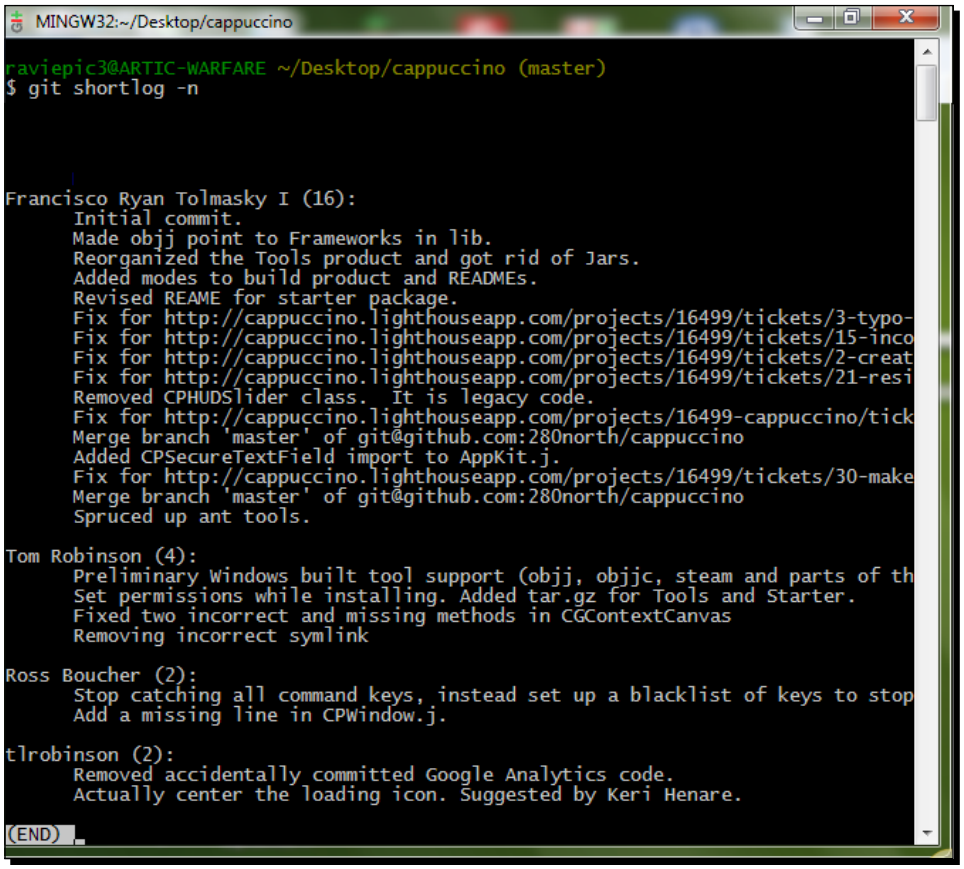
It's not over yet! Shortlog has a few defined parameters which can be used to reorder or narrow down your search to extract a particular set of information from your logs.

Time for action – parameterizing shortlog

Just add the parameter –n to your earlier command.

```
git shortlog -n
```

You should see an output as shown in the following screenshot:



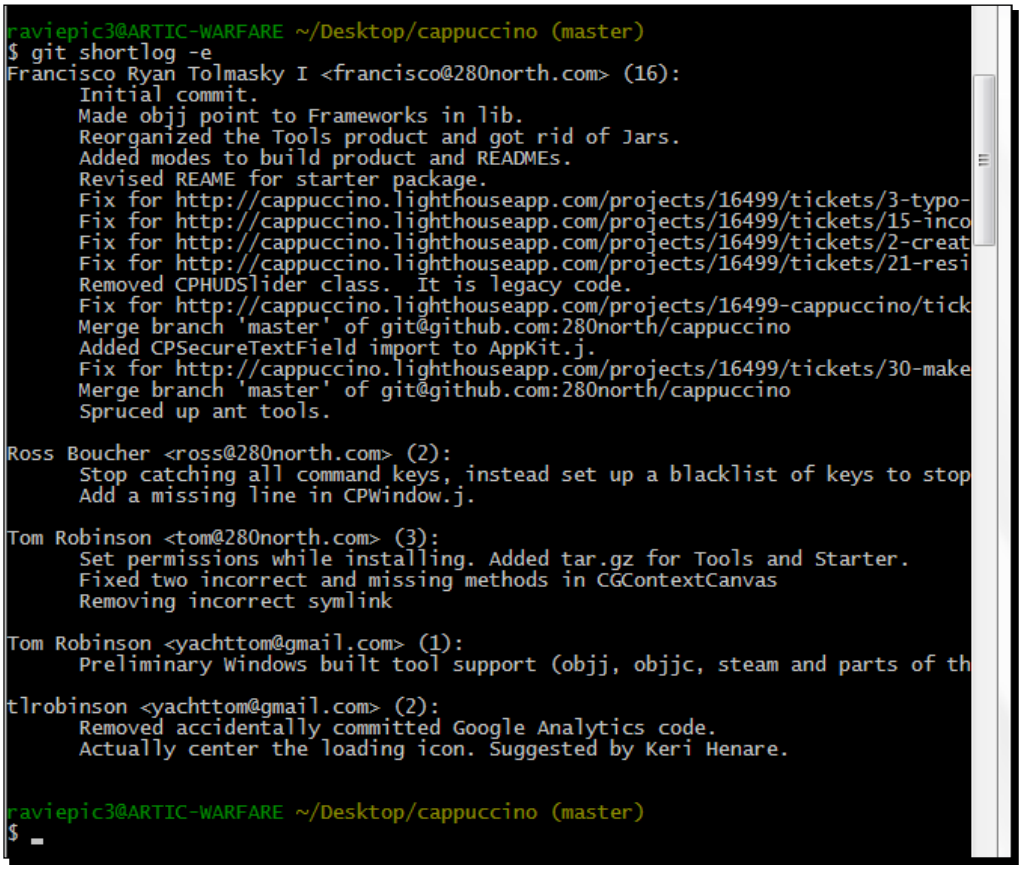
What just happened?

By adding the –n (numbered) parameter, what you have is an output that is weighted based on the number of commits instead of alphabetical ordering.

Now that we have got the idea, let's quickly run through the remaining parameters which we can put to use. To get metadata such as the e-mail of the author appended to the existing output, we shall use the –e parameter.

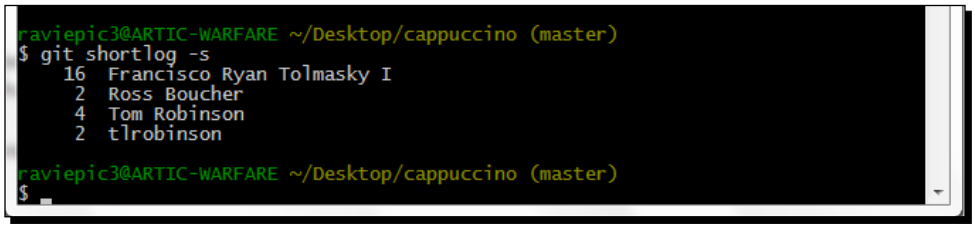
```
git shortlog -e
```

You can expect an output as shown in the following screenshot:



Wondering how you can quickly get the number of stages/commits that the repository has gone through from different users? Allow me to introduce the `-s` parameter, which should give us the count history for each user.

```
git shortlog –s
```



To summarize the parameters with their functions, refer to the following tabulation:

Parameter		**Action description**
Short form	**Full form**	
`-n`	`--numbered`	Sorts output according to the number of commits per author instead of alphabetical order
`-s`	`--summary`	Provides commit count history for each user
`-e`	`--email`	Gets the e-mail address of each author involved in committing to our repository
`-h`	`--help`	Prints a short usage message

Log search – git log

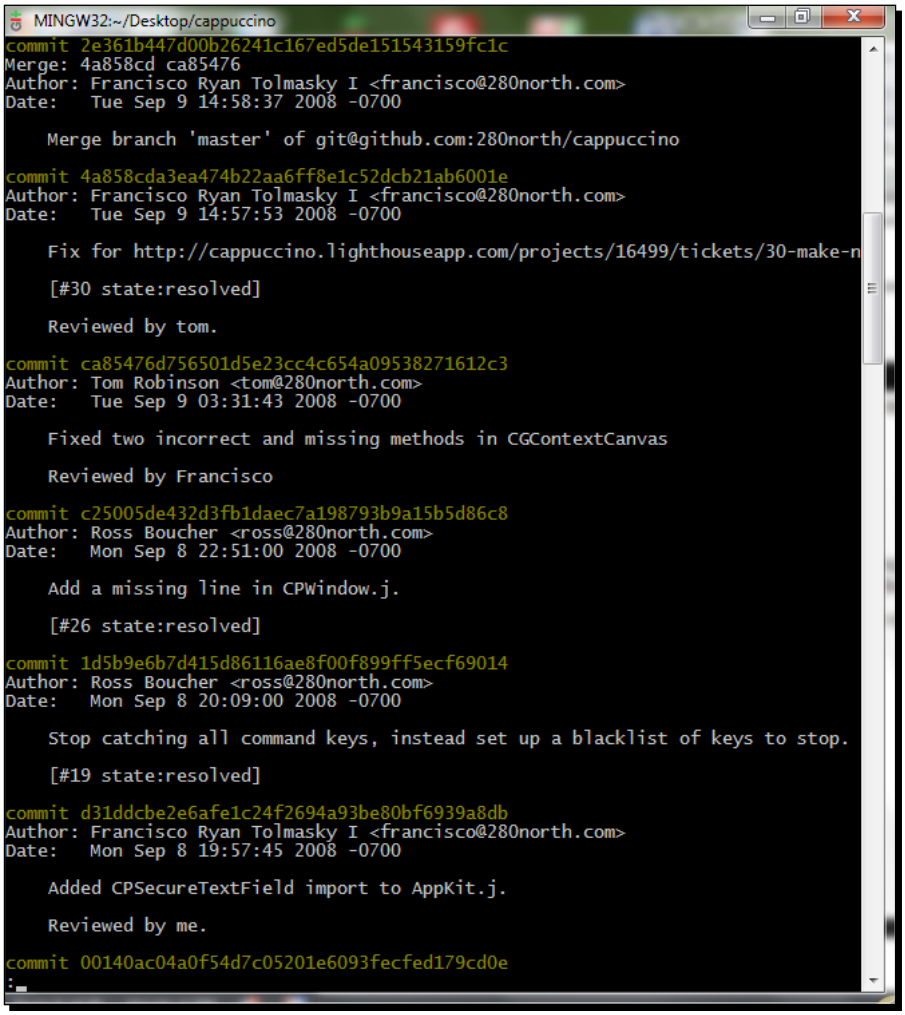
In continuation with the shortlog, add a few more weapons to your arsenal that will aid in data extraction as per your needs. What you know about Git log from earlier chapters might be as a command to view commit ID and related meta-information alone. But what you will learn here is how flexible the logging command is by itself and a glimpse of what it packs.

Time for action – skip commit logs

Let's quickly try out the following in the CLI window that we opened earlier from the `cappuccino` repository:

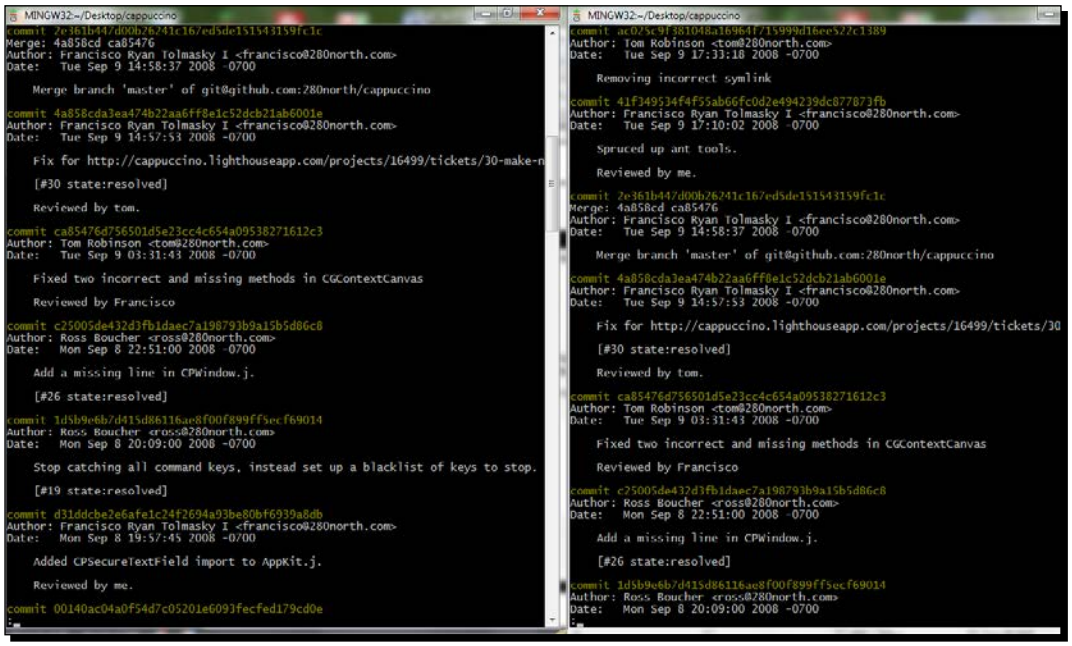
```
git log --skip=2
```

This should give an output as follows:



What just happened?

Though the output might look similar to the usual `git log` ones at first, when they are compared you will see that you have skipped off the last two commits from listing.



The `--skip=number` makes this possible. The `number` parameter can take any integer value to skip that number of commits for you.

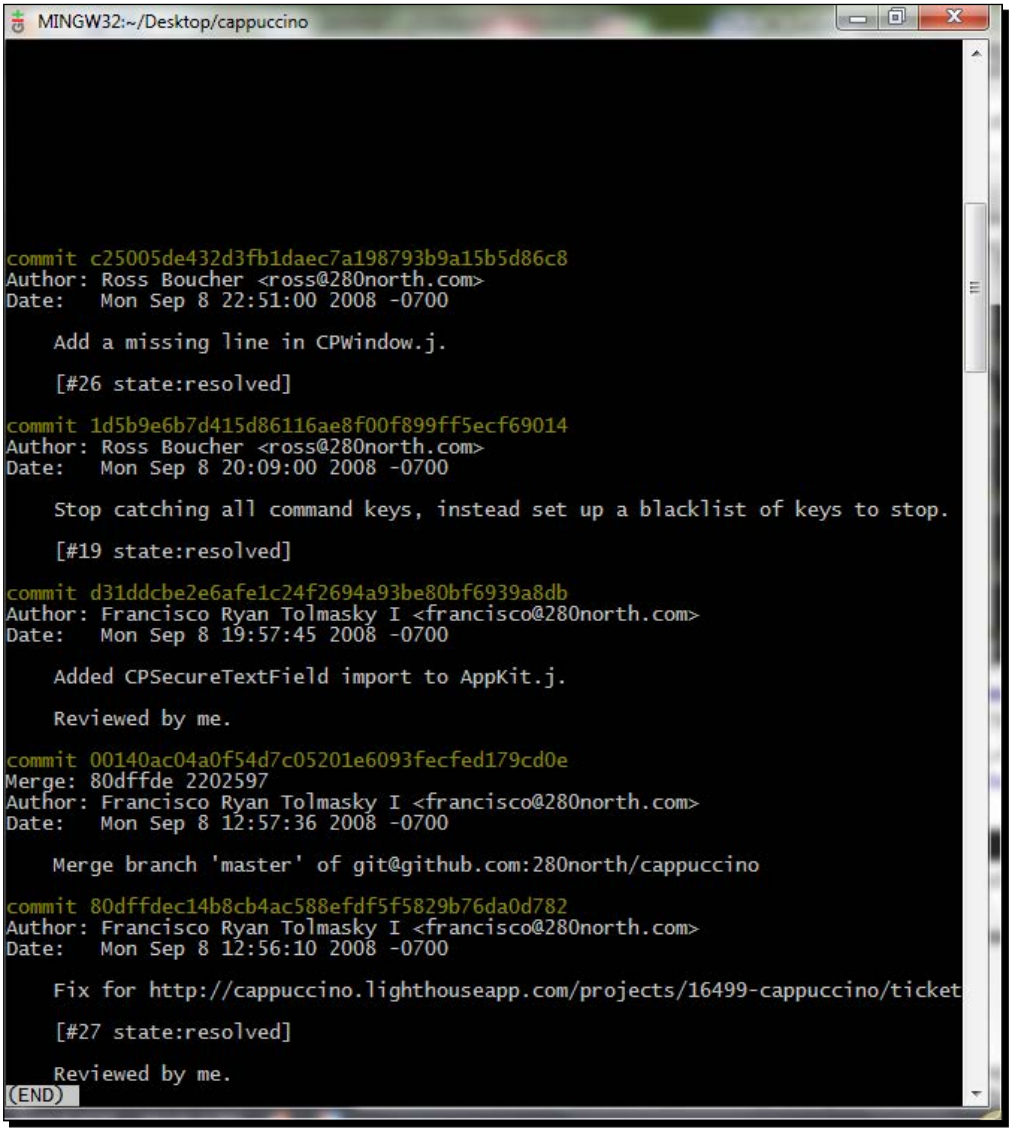
There would be numerous situations where you would want to filter content based on date. This can be done with the `since/after` and `until/before` operators with `git log`.

Time for action – filter logs with date range

After performing a `git log`, select two dates, which we shall use to filter in our following command:

```
git log --since=2008-09-08 --until=2008-09-09
```

This should give an output as follows:



What just happened?

Using the `--since=date` operator filters the logs starting from the specified date and limits it before the date specified using the `–until` parameter.

Note that you can also specify relative dates like `–since=2.days` or `--since=3.months` to filter the output.

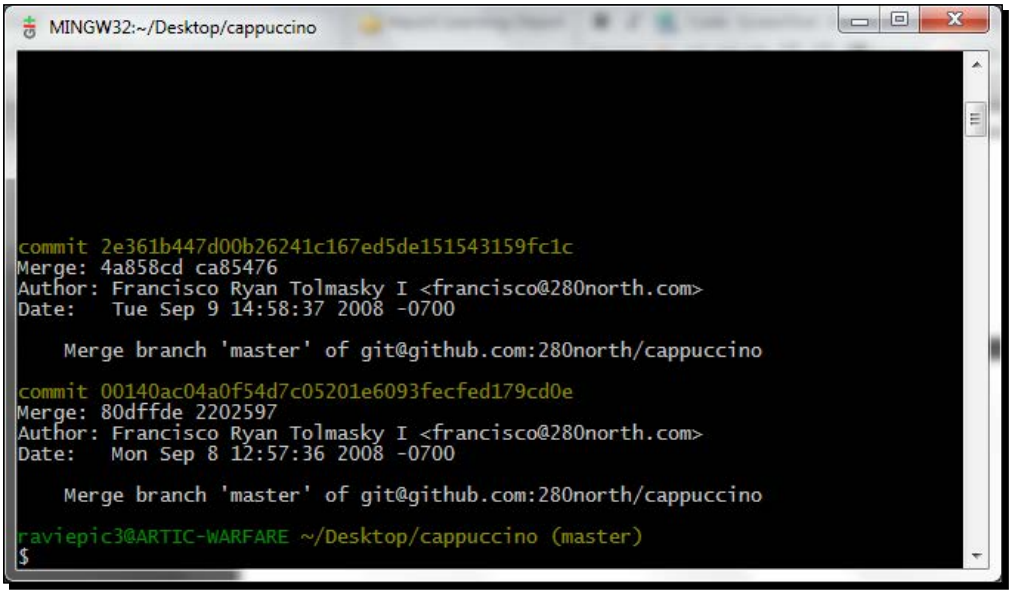
If you are wondering if there is a way to perform a search based on a keyword across commits – yes you can, with the `–grep` parameter of `git log`.

Time for action – searching for a word/character match

Type the following command in your CLI window:

```
git log --grep="Merge"
```

This should give an output as follows:



What just happened?

We have used the `grep` utility associated with `git log` to search for a given keyword, "Merge" in our case, across the commit messages provided for different commits.

To perform a search without being case sensitive, the `-i` option can be appended at the end of the previous command.

To summarize the parameters with their functions, refer to the following tabulation:

Parameter	Action description
`--skip=number`	Skips the number of commits in showing the log output
`--since,after=<date>`	Shows commits made since the given date
`--until,before=<date>`	Shows commits made until the given date
`--grep=<pattern>`	Limits the log output matching the commit message with the given pattern

Clean

There are several situations where we would need to *Ctrl + Z* our bloopers away when handling files. One classic example is in the middle of working when you unzip a ZIP package inside a directory that already contains some files, only to discover that the ZIP package straight away put those files inside the directory without creating a separate directory for the files which were unpacked. I see you nodding, and I know you're smiling.

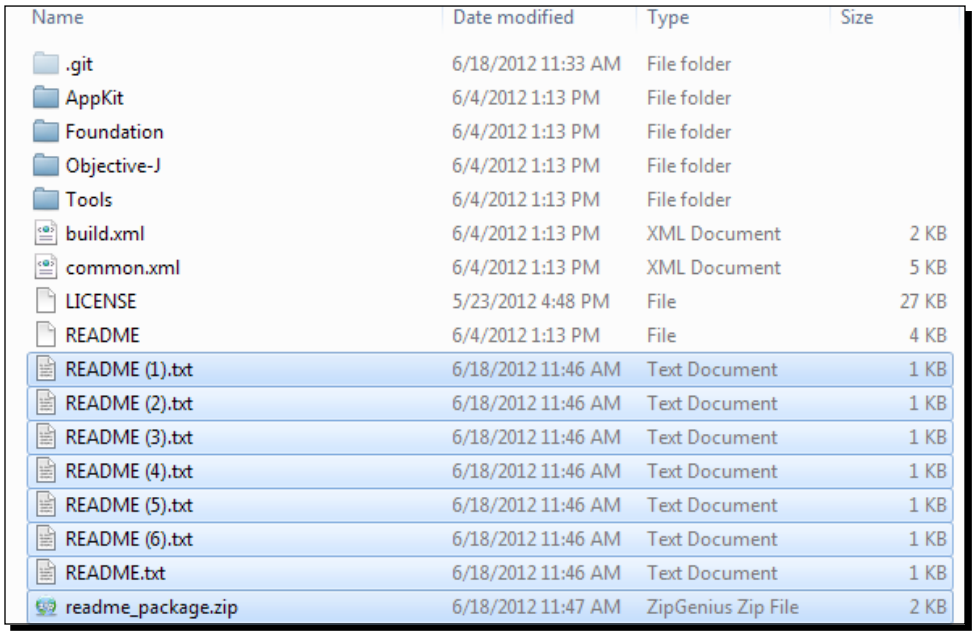
Well, situations like these can be easily handled if the directory where you unzipped the zip package was watched by Git (a Git repository). Let's reproduce this scenario and see how it can be handled within seconds.

Time for action – emulate the mess

Perform the following steps:

1. Download the ZIP package named `readme_package.zip` from `http://www.packtpub.com/support` and place it in the `cappuccino` repository on which we have been working to learn these commands.

2. Unzip the contents of the ZIP right inside the `cappuccino` directory in a way so you see seven README files as shown in the following screenshot:



Name	Date modified	Type	Size
.git	6/18/2012 11:33 AM	File folder	
AppKit	6/4/2012 1:13 PM	File folder	
Foundation	6/4/2012 1:13 PM	File folder	
Objective-J	6/4/2012 1:13 PM	File folder	
Tools	6/4/2012 1:13 PM	File folder	
build.xml	6/4/2012 1:13 PM	XML Document	2 KB
common.xml	6/4/2012 1:13 PM	XML Document	5 KB
LICENSE	5/23/2012 4:48 PM	File	27 KB
README	6/4/2012 1:13 PM	File	4 KB
README (1).txt	6/18/2012 11:46 AM	Text Document	1 KB
README (2).txt	6/18/2012 11:46 AM	Text Document	1 KB
README (3).txt	6/18/2012 11:46 AM	Text Document	1 KB
README (4).txt	6/18/2012 11:46 AM	Text Document	1 KB
README (5).txt	6/18/2012 11:46 AM	Text Document	1 KB
README (6).txt	6/18/2012 11:46 AM	Text Document	1 KB
README.txt	6/18/2012 11:46 AM	Text Document	1 KB
readme_package.zip	6/18/2012 11:47 AM	ZipGenius Zip File	2 KB

3. Now open your CLI window and type the following command:

```
git status
```

This will give you the current status as follows:

```
raviepic3@ARTIC-WARFARE ~/Desktop/cappuccino (master)
$ git status
# On branch master
# Your branch is behind 'origin/master' by 5420 commits, and can be fast-forward
ed.
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       README (1).txt
#       README (2).txt
#       README (3).txt
#       README (4).txt
#       README (5).txt
#       README (6).txt
#       README.txt
#       readme_package.zip
nothing added to commit but untracked files present (use "git add" to track)
```

What just happened?

We have successfully emulated the scenario of accidental unpacking as discussed earlier.

But under Git, the files that you have unpacked are listed down as untracked files. This means that irrespective of the number of files that have got mixed because of the unzip action with your original files, Git can easily identify them and let you know about it.

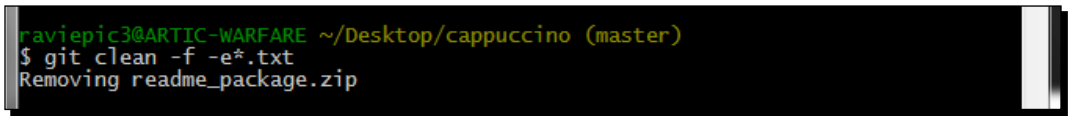
We can remove either all or a selected few files by specifying a pattern, which when matched, the clean command will skip them from removal. We shall get to know more about this in the following *Time for action* items.

Time for action – clean up your mess with pattern match

Let's skip those README files from deletion and first remove the ZIP package which we copied and pasted inside our repository. Type the following command in your terminal:

```
git clean –f –e*.txt
```

This should give you an output as shown in the following screenshot:



What just happened?

`Git clean` needs a `force` (`–f`) operator to be specified to remove files that are not monitored by it, whereas the `–e` operator takes a pattern following it and excludes the files matching the specified criteria.

In our case `*.txt` was the pattern, which matched all the `.txt` files that were created in the repository as a result of unpacking, therefore the only file left out was the `readme_package.zip` file, which got removed.

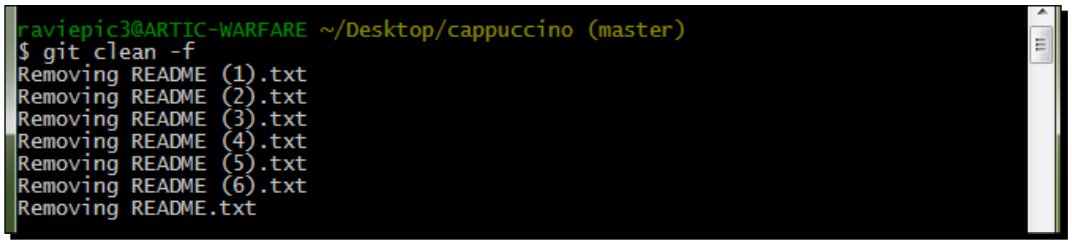
We can avoid specifying the `–f` parameter every time we run a `git clean` command if we set the `clean.requireForce` configuration variable to `false`.

Time for action – wipe out your mess completely, no exceptions

To remove these files, which have been populated in our repository unwantedly, you just call for a napalm strike on them! That's right, from the earlier command you just exclude the exclusion part.

```
git clean –f
```

This should give an output as shown in the following screenshot:



What just happened?

The `clean` command removes all untracked files from your current repository. The `–f` parameter forces `git clean` to remove those untracked files from your repository.

The following is the list of parameters you can put to use with `git clean`:

Parameter		Action description
Short form	**Full form**	
`-f`	`--force`	Removes untracked files
`-d`		Removes untracked directories along with files
`-n`	`--dry-run`	Doesn't remove anything, but show what will be done
`-q`	`--quiet`	Stays quiet and only reports errors but not the files that were successfully removed
`-e<pattern>`	`--exclude=<pattern>`	Excludes the files matching the specified pattern along with the ones specified in `.gitignore` (per directory)

Tagging

Tagging comes in handy when you want to mark a specific point in your history with some metadata and refer to it henceforth with the same tag. We have two types of tags in Git.

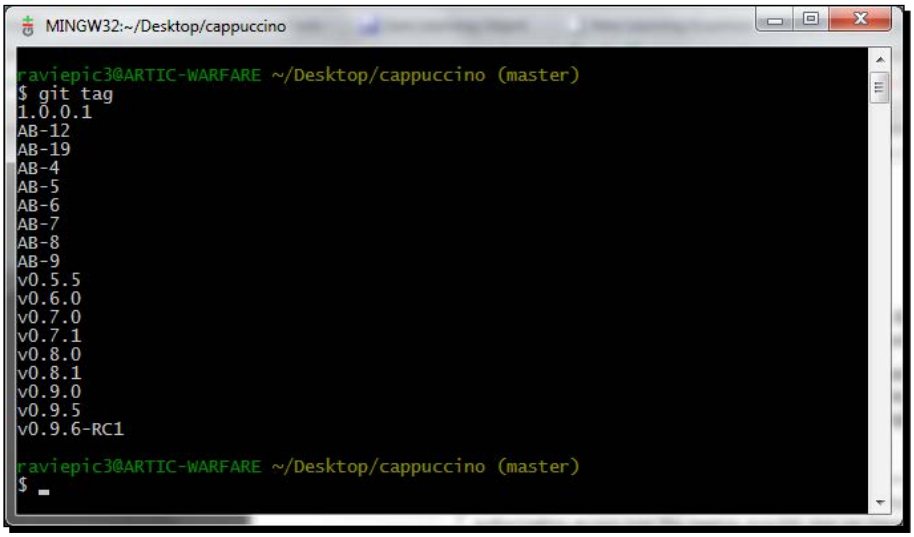
- **Lightweight**: This method of tagging tracks the tag name alone without worrying about by whom or when the tag was created. This might come in handy when you have only one person working on the files in your repository or if the tags that you create are just for simple reference of different phases that your project files in the repository have gone through.
- **Annotated**: This method of tagging tracks the author's name, time of tag creation, and the tag name with a description, if given. This might come in handy when you want a retracement to be done on the attained milestones or when you have multiple people working on the same repository, and so on.

When you use Annotated tags, it is possible for the project owner to maintain authorization access over the tagging process; in advanced setups it is even possible to have control over who tags, when they do, and mark their authorization identity for future reference.

Having said enough, let's understand it better by trying it out.

Time for action – lightweight/unannotated tagging

1. First, let's list out the existing tags with the `cappuccino` repository by executing `git tag` in your CLI window. This should produce an output like the following:



The supplied `git tag` command has retrieved all the tags available in the repository and lists them in alphabetical order.

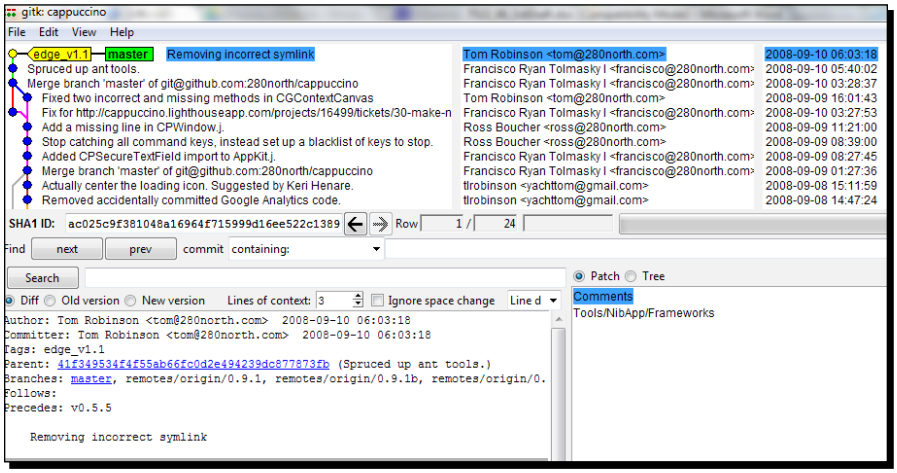
2. Now let's create a lightweight tag in our `cappuccino` repository by executing the following:

```
git tag edge_v1.1
```

If there are no errors returned, your tag should have been created. You can verify it by doing a tag listing, which we learned in the previous step.

3. Or you can visualize it by opening gitk, which will give you the following appearance:

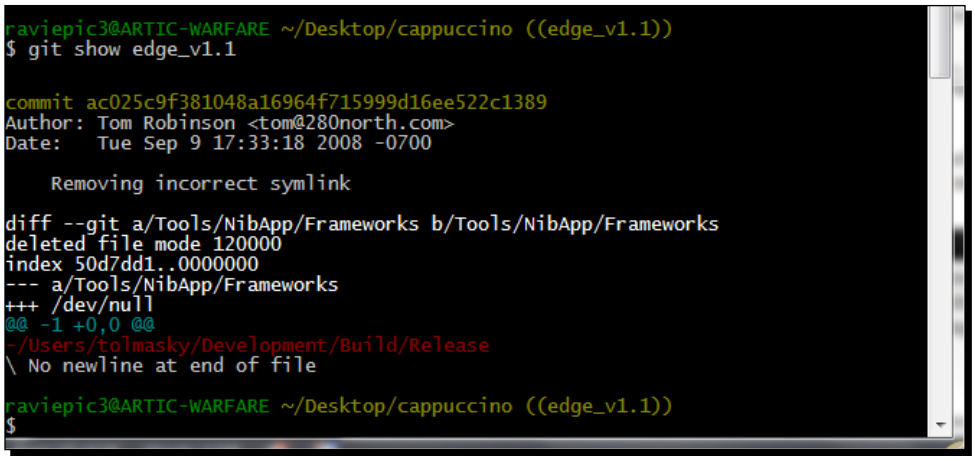
You can open gitk either from Git Gui's repository menu or by just typing `gitk` in your CLI window.



4. If you want to browse the changes to the files for a given commit that is tagged, you can either look at the bottom left of the gitk window or use `git show <tagname>` as shown in the following line:

   `git show edge_v1.1`

   This will give an output as shown in the following screenshot:



What just happened?

We have successfully created and attached a lightweight/unannotated tag to a specific commit. We also learned to list out all the tags available in the repository and if needed, view granular level changes associated with any given tag.

We read about referencing a commit with the tag names instead of the commit SHA1 ID. Let's understand what this means, practically.

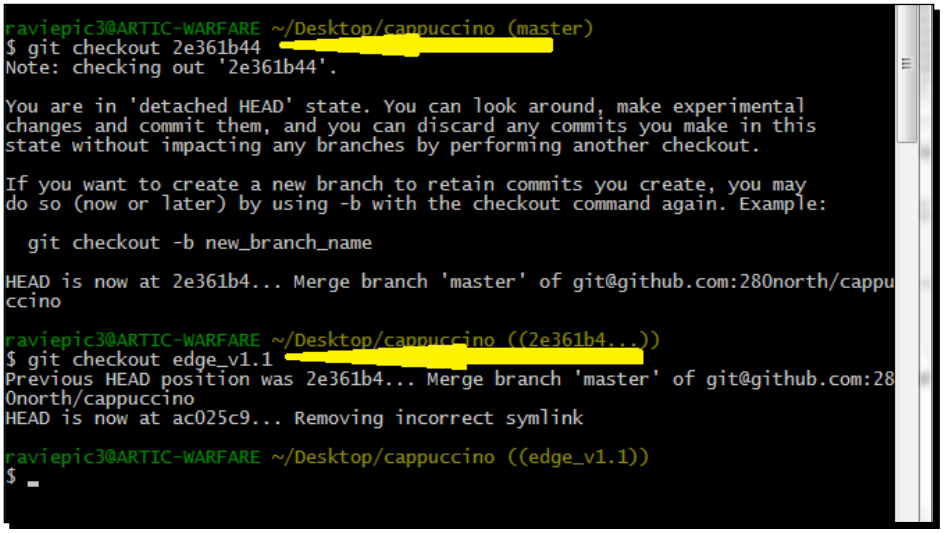
Time for action – referencing tags

We have learned `git checkout` as a function to travel back in history. As you know, this process needs the SHA1 ID of the commit, which you would like to visit. Now let's see how it can be done when it comes to handling tags. Type the following commands in your CLI window:

`git checkout 2e361b44`

`git checkout edge_v1.1`

This should give you an output as shown in the following screenshot:



What just happened?

We checked out to a commit made at an earlier date using the usual checkout (`git checkout` SHA1 `ID`) method but came back to the latest commit with the usage of the tag name associated with the commit (`edge_v1.1`).

Time for action – annotated tagging

Having learned the method for creating an unannotated/lightweight tag, the process of creating an annotated tag is as simple as adding the `–a` parameter to it. Let's type the following in the CLI window:

```
git tag –a ann_v1.1 –m 'Annotated tag v1.1'
```

What just happened?

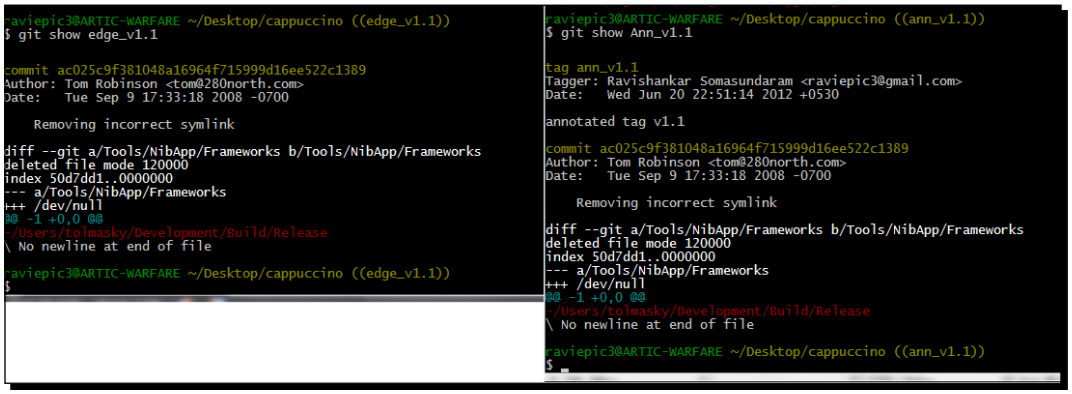
If the command does not return any error upon execution, it indicates that your annotated command has been created successfully. We have created an annotated tag called Ann_v1.1. Here –a denotes the annotated tag name and –m takes a string value as the description for the created tag.

Note that we have created both the annotated and unannotated tag for the same commit so that they can be compared later.

Simple exercise

1. To verify the creation process, do a tag listing and verify that the tag that you created exists there.
2. Check out the content changes associated with the commit for which you have created an annotated tag.

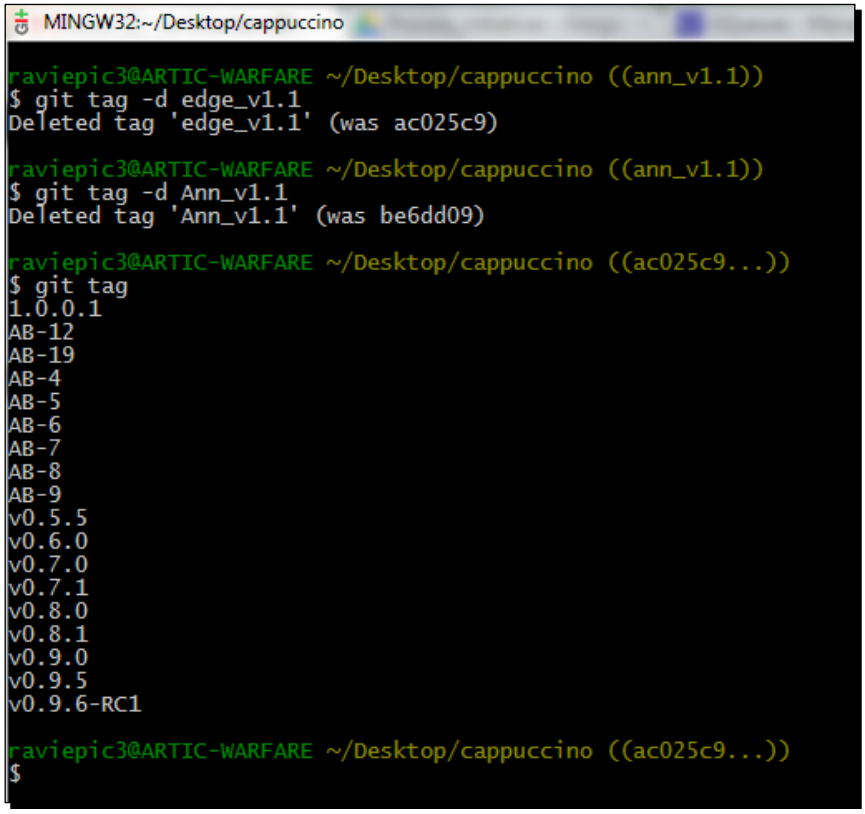
To understand the difference between the lightweight and annotated tag, their outputs are shown side by side.



While creation differs a little bit, the procedure for the deletion of both the annotated and unannotated tags is identical. You just associate the –d parameter followed by the tag name as follows:

```
git tag –d edge_v1.1
git tag –d Ann_v1.1
```

This should produce an affirmation as shown in the following screenshot:



What just happened?

We just deleted the tags `edge_v1.1` and `Ann_v1.1`, which we created in the process of learning about tags inside the `cappuccino` repository.

The `–d` parameter will delete the tag whose name is passed as an argument to it, irrespective of it being an annotated or unannotated tag. You can perform a tag list to verify this as shown in the previous image after the deletion.

Summary

We have learned what is and how to:

- Use shortlog:
  - To sort and quantify commits made per author
  - To summarize commit logs
  - To get metadata like e-mail addresses for each author performing commits to our repository
- Use different parameters with `git log`, which enables us to:
  - Skip the specified number of commits while displaying log entries
  - Extract data from commit logs within a date range
  - Search for a string value across commit messages to identify commits

Additionally, we have also learned how to clean our repository in case we or anybody else has messed it up by injecting files into it that do not belong there.

Not only that, we also saw, in practice, how to mark the milestones of our repositories' content using annotated and lightweight tags.

6

Unleash the Beast – Git on Text-based Files

We have seen two different modes of working with our content managed by Git in earlier chapters, with a gaming analogy calling it the single/solo player mode and multiplayer mode.

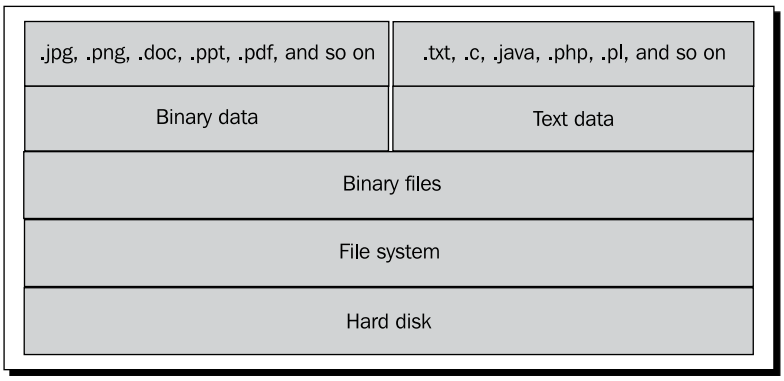
Hold on to your seats. This chapter is an answer to your long awaited question: What about the real multiplayer mode with several people playing in parallel? In other words, having multiple people work on the same stuff at the same time.

The concepts that we are going to learn about in this chapter are as follows:

- Merging and
- Resolving conflicts, making your way through the content in the way you or your team want to

Git for text-based files – an introduction

Git arms itself with several functionalities when it comes to handling text files. On a higher level let's understand the different file types and what they really mean using the following stack:

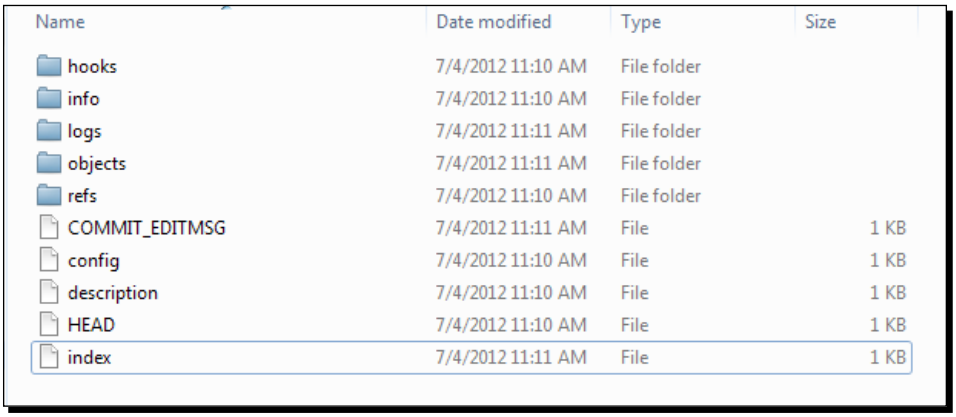


Going from the top to the bottom, it is a layered approach starting with how a user sees a file, how a computer sees it, until its storage at the bottommost layer.

Binary data: Any file whose contents can be read only through specific programs, such as Microsoft Word for documents and picture viewer for images, is called binary data/values.

Text data: Any file, irrespective of its extension or nature, whose content is pure text and can be opened with a normal text editor like notepad or WordPad is considered to contain text data.

To quote an example for giving more clarity, open up your `.git` directory inside your project, and you will see a file structure as shown in the following screenshot:



Here the file called **index** is considered to contain binary data whereas the files **COMMIT_EDITMSG**, **config**, **description**, and **HEAD** are considered to have textual data. Open them with your text editor and you'll understand why.

Make sure you don't alter anything in those files or else you might end up with a screwed up version of your repository, which might need a few maneuvers to fix, which we are not interested in at the moment.

As another example, you can also try opening an image file with a text editor to view its raw content.

This is the basic difference that we were talking about. Let's talk more about it once we have understood the concept of **branching** and **merging**.

Multiplayer mode – multiple players at a time

Let's continue our gaming analogy from earlier chapters to relate to the multiplayer mode concept we have learned so far.

Multiple players – one at a time

Think of your favorite adventure game that has multiple levels. Consider a scenario where you are stuck in a level without knowing how to proceed forward. After desperate attempts, which ended in vain, you suddenly realize that your friend is an expert on that level, and you want to use your friend's help. So you quickly share the last saved state of the game file with him with which he can finish that level for you, save the state, and push the file back to you, which will enable you to continue the game.

The same situation can apply to you when you are working with data files, especially when you are working as a team where different people take care of different parts of a bigger task to produce a single result. Another possibility might be that you want the domain experts to handle specific portions of the work, and so on.

This also means that having multiple people working on the same document one topic at a time, where one passes the file to another to get the work done in a sequence, might go smoothly, but having multiple people working on the same file on the same topic might end up in chaos when it comes to files containing binary data.

Multiple players – all hands on deck (many at a time)

I'm a big fan of **first person shooter** (**FPS**) games. *Counter-Strike* stays at the top of my list even today. Let's take *Counter-Strike* or any other team game for comparison here. Each team member will be a specialist in not only one but two or three weapons to adapt to situations. And when required, they pitch in to take out the opponent together and complete the objective.

Similarly, when you deal with textual data in files, you can have multiple people collaboratively working on the same file, topic, and line, and manage to produce a unified output with Git. Let's learn how to put this feature to better use.

Sharing your repository

There are two commonly used modes to share your repository with others.

- Intranet
- Internet

Having got used to the way of sharing over the Internet using Bitbucket, this time let's emulate sharing over the intranet using the bare repository concepts we learned in *Chapter 4*, *Split the Load – Distributed Working with Git*.

If you are not able to recall, I suggest you go through the *Staying local – share over intranet* topic in *Chapter 4*, *Split the Load – Distributed Working with Git* to understand what the bare repository is, why we need one, and how it operates.

Time for action – getting ready to share

To keep things clear and simple we shall start with a fresh instance with minimal data, so that the changes are evident.

1. Let's create a new directory and call it `collab_source`.
2. Within the directory create a new text file called `mycontent.txt`.
3. Open up the text file that you just created, and enter the following:

```
Unchanged first line from source
Second line
Third line
```

4. Save and close the file.

5. Now make the `collab_source` directory a Git repository; then add the file `mycontent.txt` and make a commit with a commit message saying "Base commit from source".
6. Now this copy will reside in your machine for your own manipulation. Let's create a bare repository from your copy to put it in a common place from where your team members can clone it to have their own copy of the files.
7. For creating a bare clone of your existing repository, use the following command:

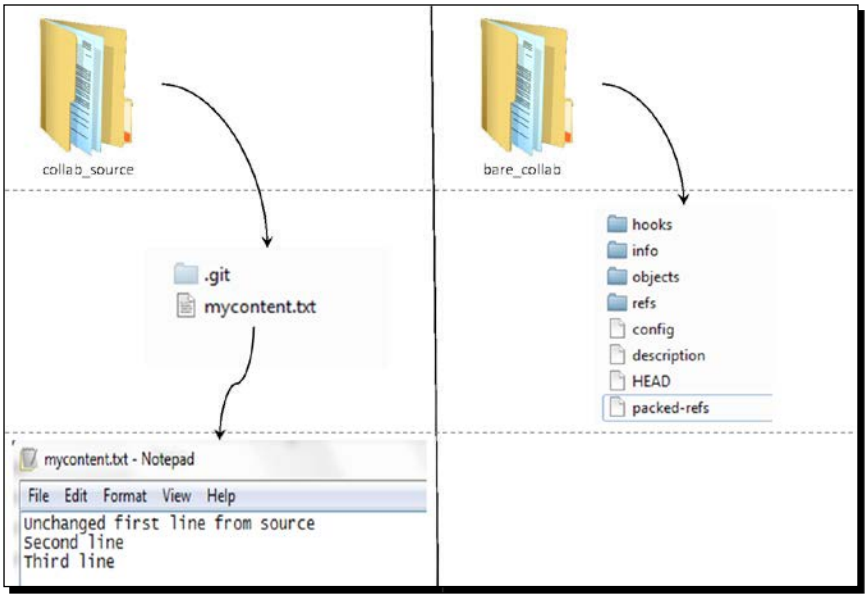
```
git clone --bare /your/path/to/collab_source /your/path/to/bare_collab
```

For this example, to convey the concept I have cloned the `bare_collab` repository in my local system itself instead of a common network share directory. But the procedures are one and the same.

The main aim of this topic is to convey the multiuser concept so I have chosen only one mode (CLI) of execution. Mostly the GUI equivalent of these commands is already known to you. In case of an exception a quick note on the GUI options is provided.

What just happened?

We have created a source repository with our content and then cloned a bare repository out of our source repository, which in turn has opened up the clone option for our team members. If you have followed the preceding steps, you should see a structure like the one shown in the following figure:



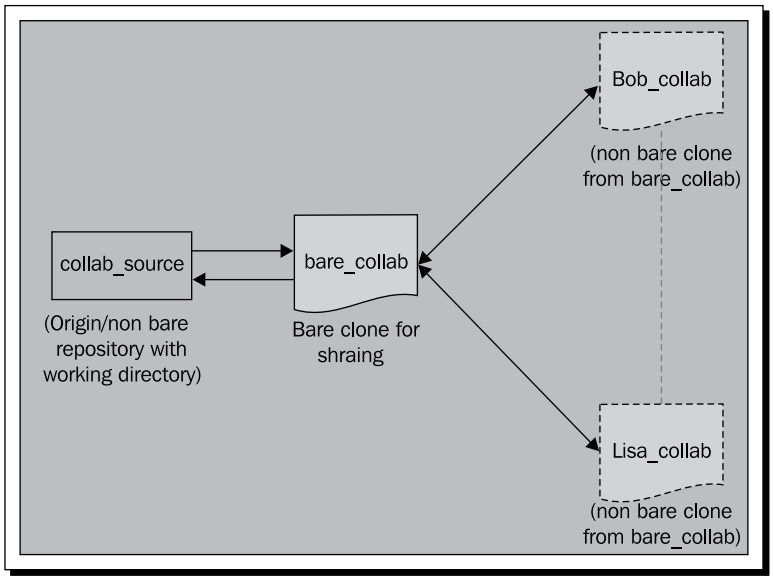
Time for action – distributed work force

Now to get their own copy of the files, Bob and Lisa, our fictitious team members, can execute the usual `git clone` command with the source as the `bare_collab` repository and the destination as their preferred location.

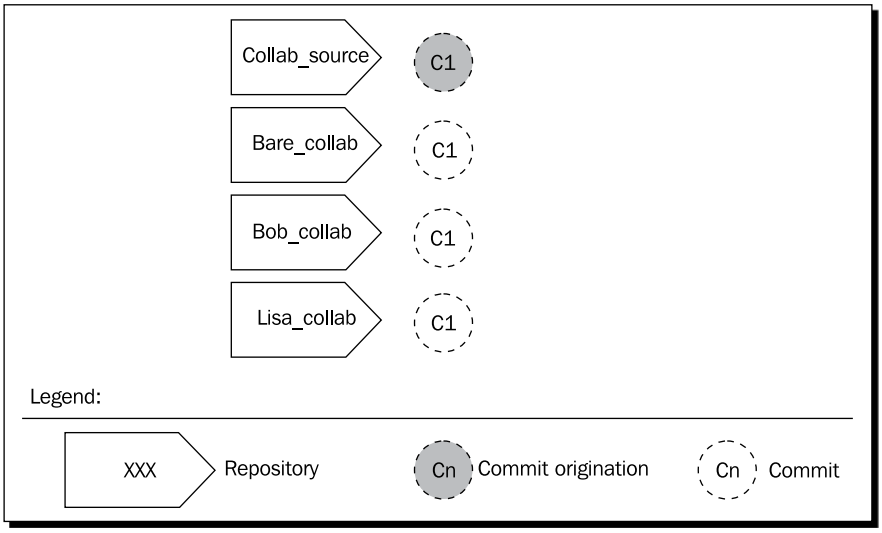
```
git clone /path/to/repository/bare_collab /path/of/local/copy/Bob_collab
git clone /path/to/repository/bare_collab /path/of/local/copy/Lisa_collab
```

What just happened?

Unless Git reported an error, Bob and Lisa have cloned the files from our bare repository named `bare_collab`. Now, the structure of how these repositories evolved is shown in the following figure:



And how the commit tree looks like is shown in the following figure:

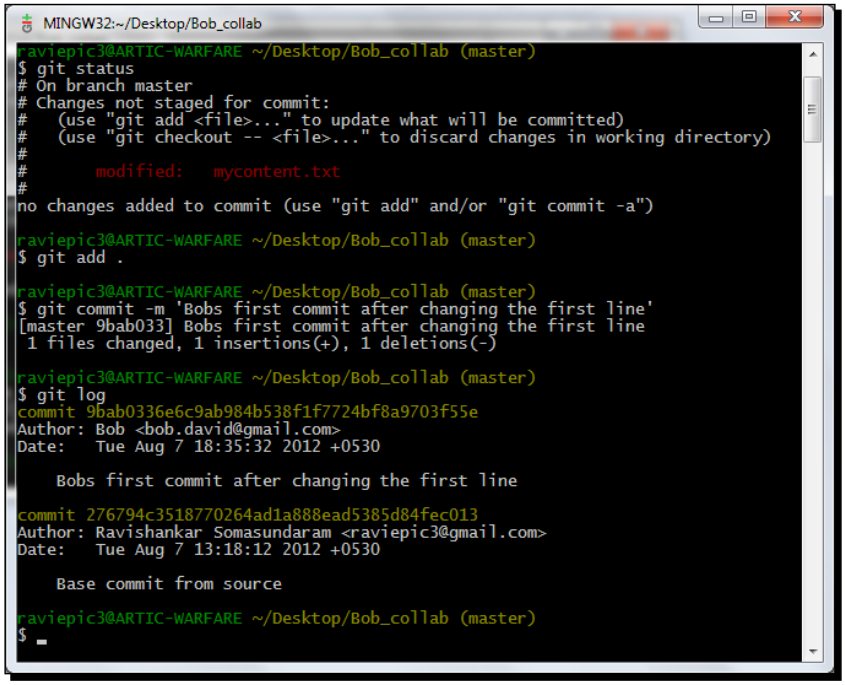


Time for action – Bob's changes

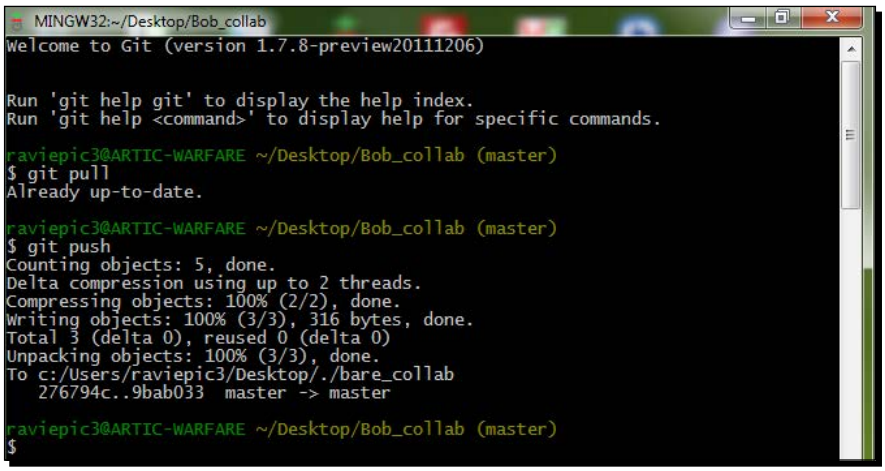
1. Now Bob feels that he needs to change the content of the file. So he opens and changes the first line's text to "First line from source - Changed by Bob" so that the content of the file looks like the following lines:

```
First line from source - Changed by Bob
Second line
Third line
```

2. Then he adds the change and commits the same as shown in the following screenshot:

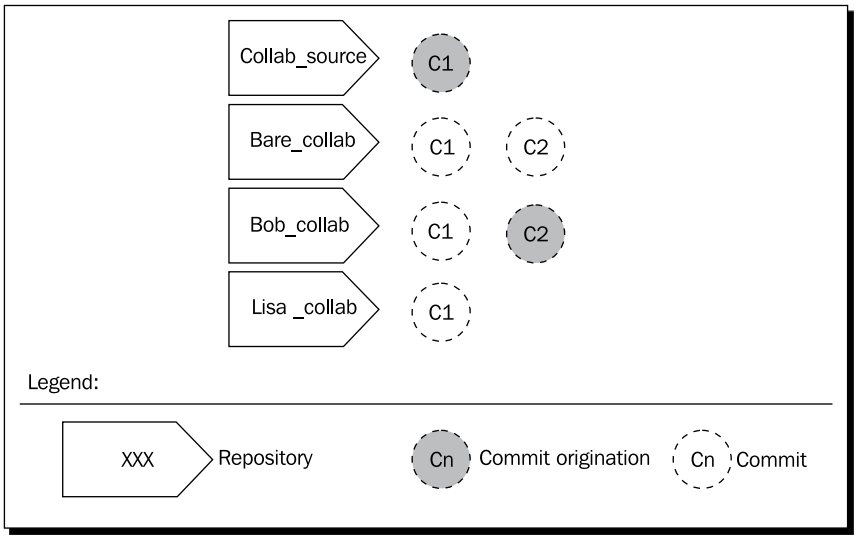


3. In the interest of sharing the change with team members, he wants to push his changes to the common bare repository but as a rule of thumb, when working with multiple people on Git, pull before pushing so as to incorporate the changes first in case somebody has already pushed before you. Bob does a `git pull` *first* and *then* a `git push` as shown in the following screenshot:



What just happened?

Because of this push operation, the bare repository has progressed its level along with Bob's changes whereas the repository at our machine (`collab_source`) and Lisa's are still behind. Now the commit tree looks like the following:

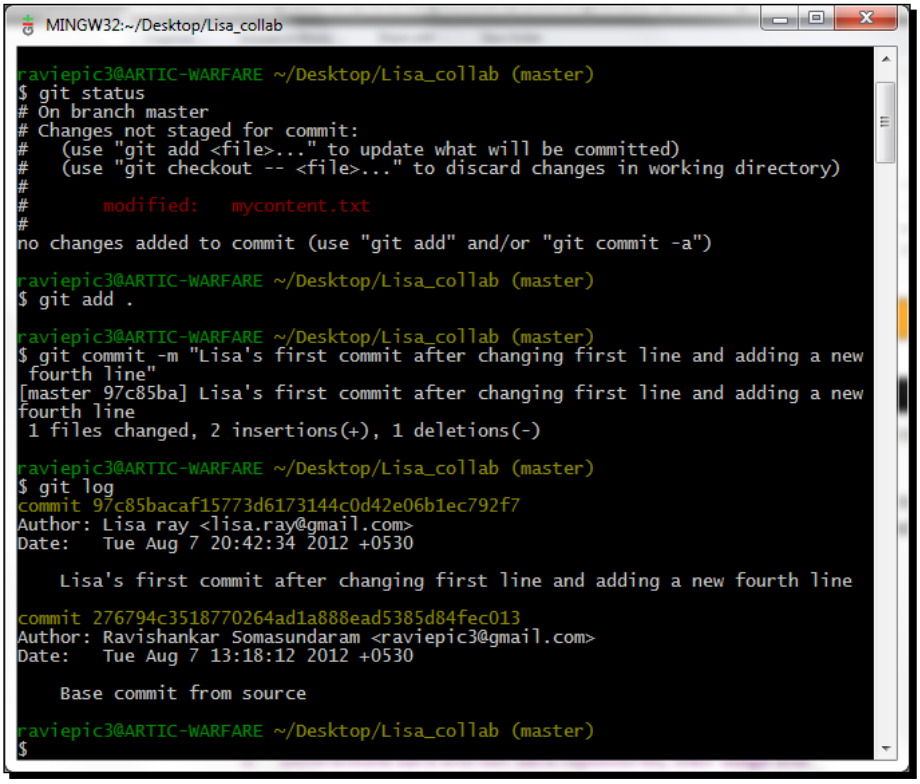


Time for action – Lisa's changes

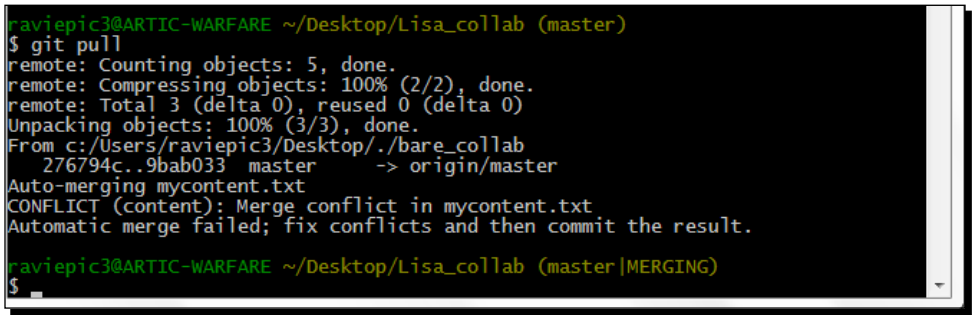
1. While Bob was doing all these operations, Lisa made her own changes. She also happened to change the first line of the file and appended one more line to it, which made the content of the file look like the following:

```
Unchanged first line from source = Not any more ;)  - Lisa
Second line
Third line
Fourth line by Lisa
```

2. Then she adds the change and commits the same as shown in the following screenshot:



3. In the interest of sharing the change with team members, Lisa wants to push her changes to the common bare repository, but as the rule of the thumb, when working with multiple people, do a pull before pushing so as to incorporate the changes first in case somebody has already pushed before you. She does a `git pull` *first*, which gives her the following error message:



What just happened?

Lisa made changes, added, committed, and when she tried to pull from the central `bare_collab` repository, got bumped into a merge conflict.

If you focus on the last three lines, it would be evident why the pull got stopped. Git automatically tried to merge the changes Lisa made with the changes already pushed by Bob in the file `mycontent.txt`. Because both have changed the first line, Git smartly stops the merging and asks us to fix the conflicts.

Time for action – Lisa examines the merge conflict

Lisa opens up the file to see the conflict that is stopping her and finds a pattern as follows:

```
<<<<<<< HEAD
Unchanged first line from source = Not any more ;)  - Lisa
=======
First line from source - Changed by Bob
>>>>>>> 9bab0336e6c9ab984b538f1f7724bf8a9703f55e
Second line
Third line
Fourth line by Lisa
```

What just happened?

The first line, which has continuous left arrows with the word HEAD is nothing but Lisa's current position in the repository. The next line shows the changes made by her to the file.

This is followed by continuous = signs, which marks the end of Lisa's content and beginning of Bob's content. It's a separator, which is immediately followed by Bob's content in the next line, which is then followed by the commit ID generated when Bob made that change.

Time for action – Lisa resolves the merge conflict

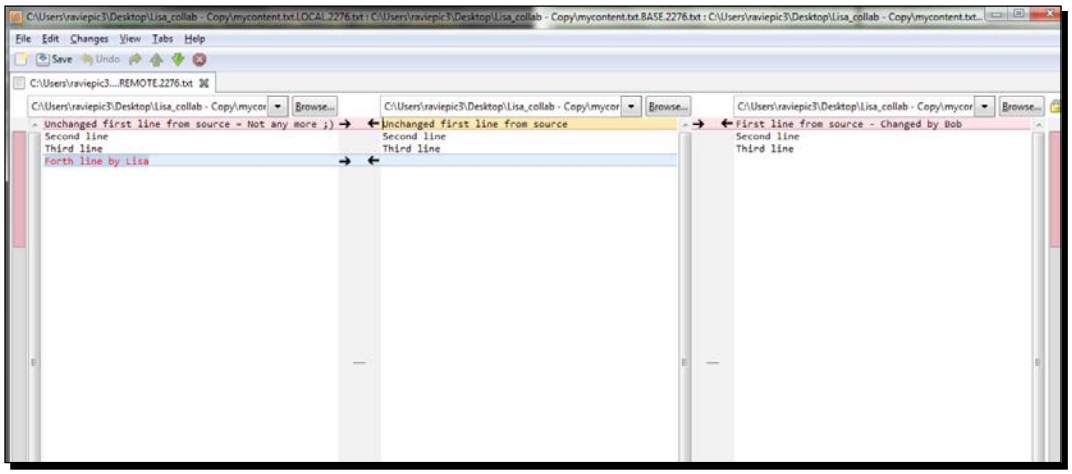
Perform the following steps:

1. Resolving the conflict is a very simple procedure. You are given four choices.
    - Specify an order and have both the changes (which in our case are Lisa's and Bob's changes)
    - Delete the existing change and impose yours
    - Delete your change and apply the change fetched
    - Delete both

However, the fourth option is very unlikely to happen.

To perform any of these operations on the content, one can use a common text editor or an interactive merge tool, which will give you three views (local, base, and remote) using which you need to solve your commits.

Local view is the current modified version, **base** is our earlier version before modification, which gets decided by Git automatically, and **remote** is the modified remote version, which we are trying to fetch and merge. You need to move and order your changes along with the remote version using the arrows and indicators available. A screenshot of how an interactive merge tool (I have showcased **meld**, which is a python based utility) looks as follows:



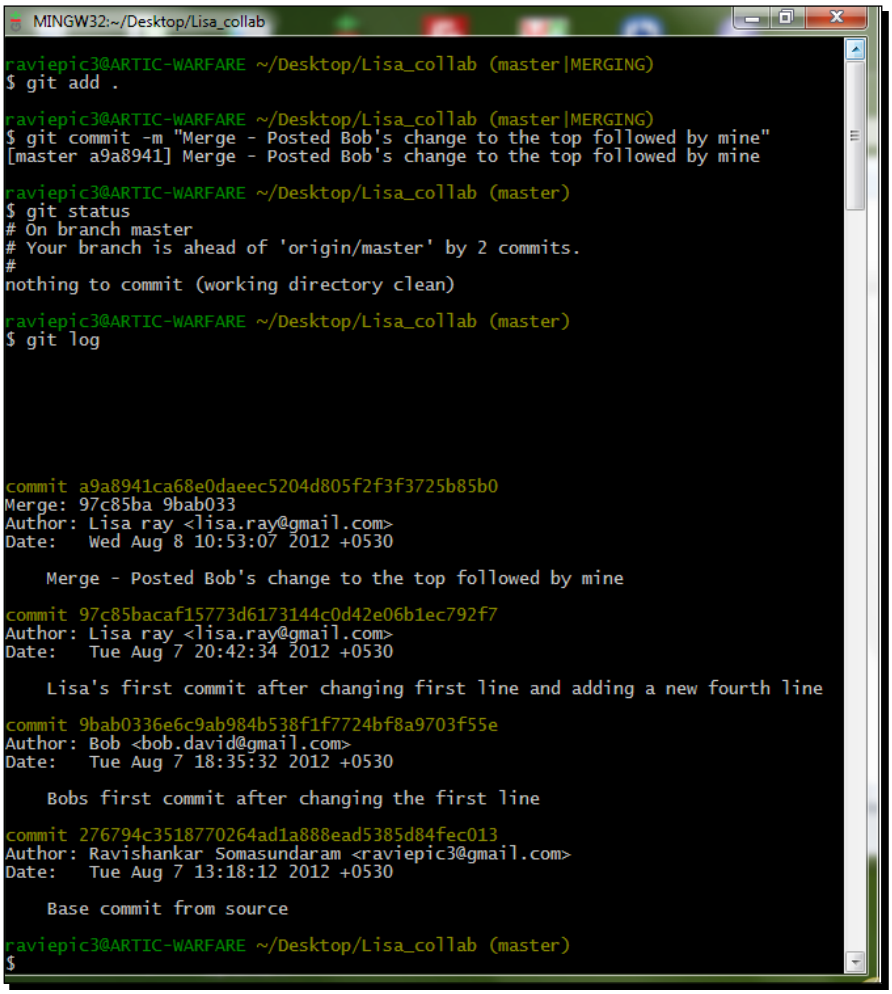
We will use a normal text editor to solve this situation now, so as to understand the underlying concept.

Lisa is going to go with the first option of ordering and incorporating both the changes; let's see how she does it.

2. Lisa feels that having Bob's changes up the order is better followed by hers. After deciding the order, she opens the file using an ordinary text editor and changes the content as follows:

```
First line from source - Changed by Bob
Unchanged first line from source = Not any more ;)  - Lisa
Second line
Third line
Fourth line by Lisa
```

After making the previously mentioned changes, she *adds* the change and *commits* it with the message "Merge - Posted Bob's change to the top followed by mine" as shown in the following screenshot:



3. Following this, Lisa pushes her commits to our central bare repository (`bare_collab`) with the usual `push` command, as shown in the following screenshot:

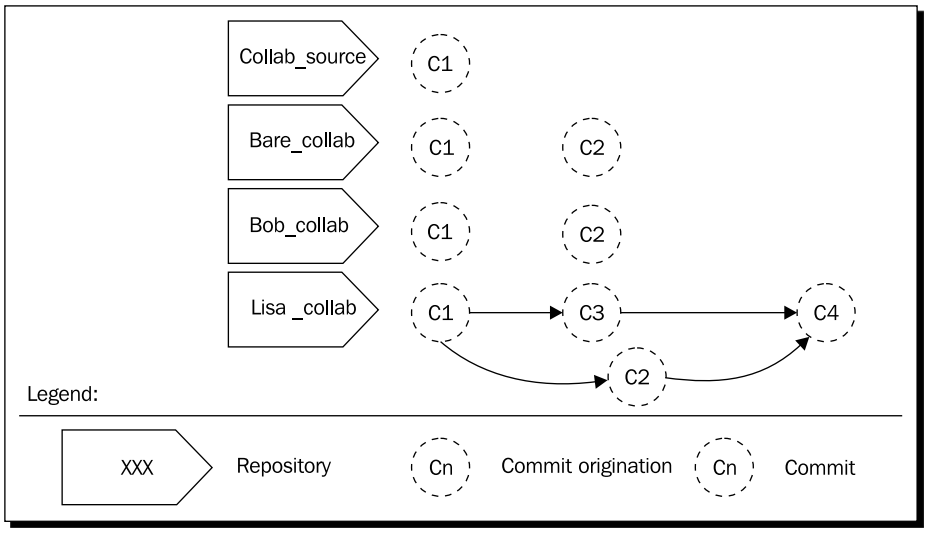
```
raviepic3@ARTIC-WARFARE ~/Desktop/Lisa_collab (master)
$ git push
Counting objects: 10, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 665 bytes, done.
Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
To c:/Users/raviepic3/Desktop/./bare_collab
   9bab033..a9a8941  master -> master

raviepic3@ARTIC-WARFARE ~/Desktop/Lisa_collab (master)
$
```

What just happened?

Lisa has successfully resolved the conflicts and made her changes available to other members of the team by pushing the changes to the central bare repository.

This should give a commit tree structure as shown in the following flow diagram:

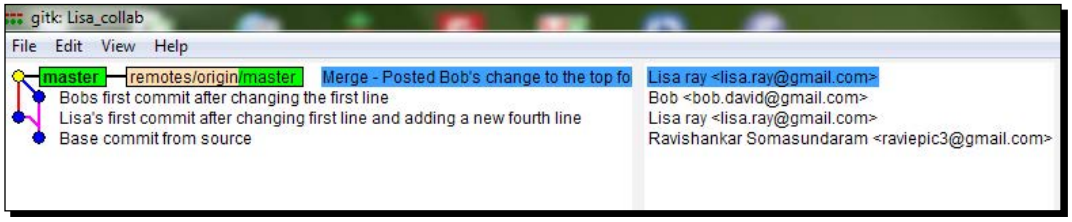


As you can see from the diagram, C3 (the local commit made by Lisa) and C2 (the commit made and shared by Bob) are being merged to form a new commit called merge commit C4.

At any given point of time you can get such graphical representations both from the GUI as well as CLI mode.

GUI mode – get the repository's history graph

Select **Visualize all branch history** form the **Repository** menu of Git Gui. Once Gitk opens, at the top left you have your repository's history visualization. Lisa's visualization is as shown in the following screenshot:

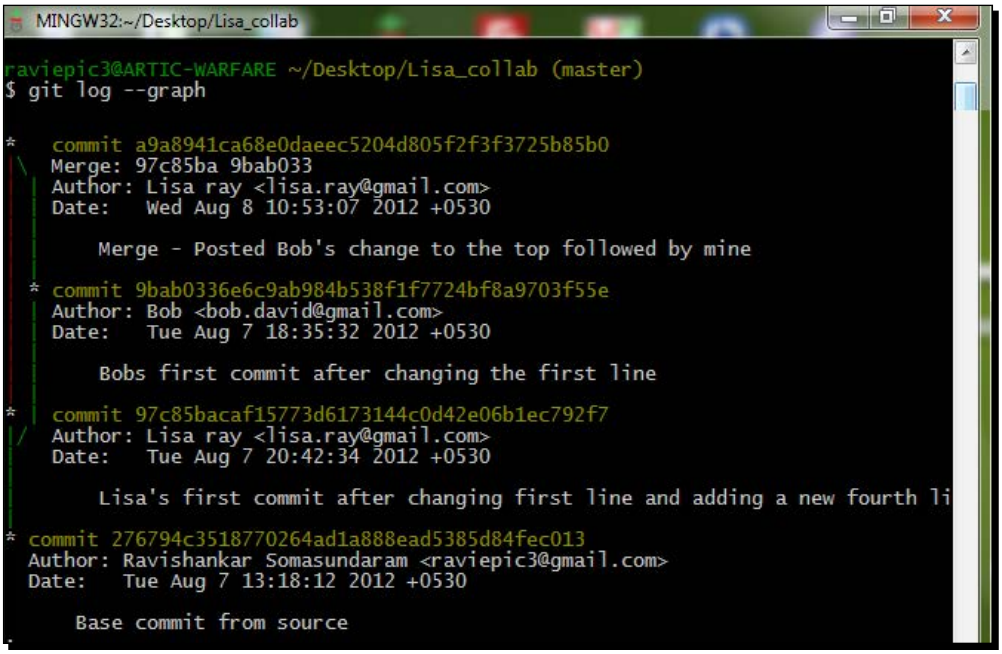


CLI mode – get the repository's history graph

In your terminal/console, switch to the Git repository's location and then use the following command to get a tree representation of its history:

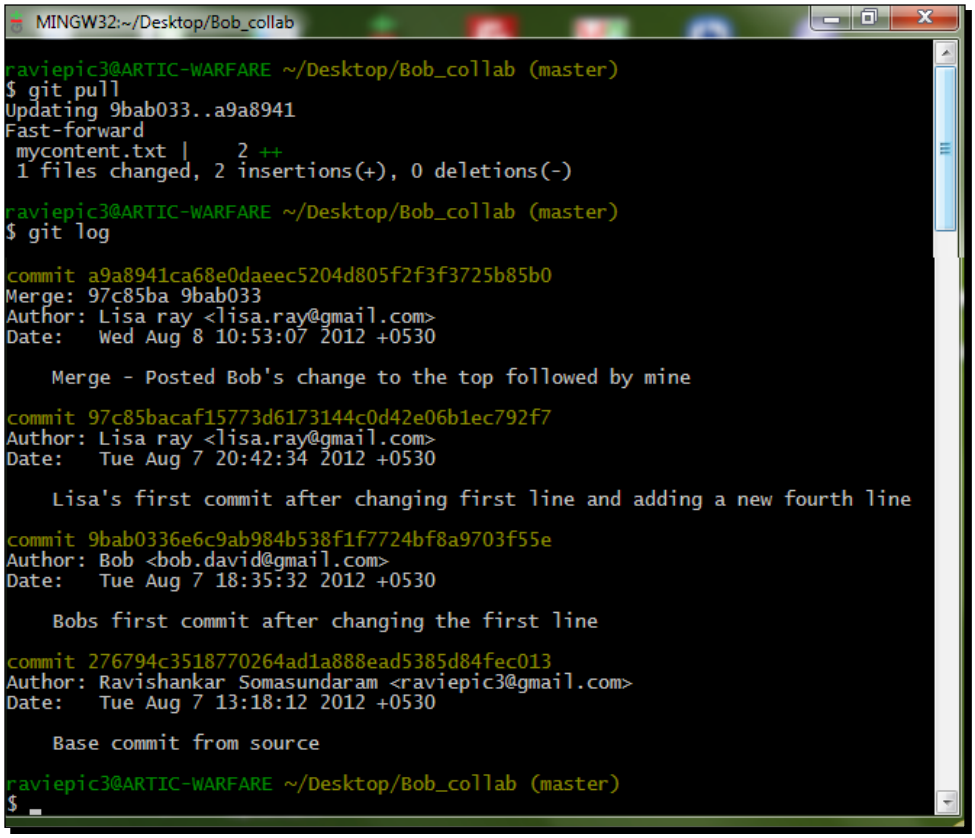
```
git log --graph
```

You will see a representation as shown in the following screenshot:



Time for action – team members get sync with the central bare repo

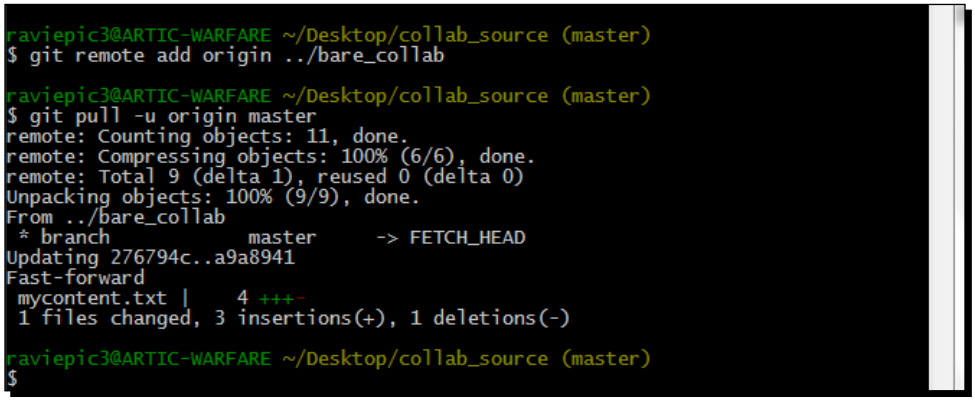
1. Bob feels it's been a while since he received updates from the bare repos so he pulls to get the latest changes, with the results being as follows:



2. Well, finally let's not forget the source repository, mother of all these repositories, for the update. Before doing a `git pull` from there we should point to the origin as the bare repository, and then perform a pull operation. The commands are as follows:

```
git remote add origin /path/to/bare_collab
git pull –u origin master
```

This gives us an output as shown in the following screenshot:



As we have learned in *Time for action – adding a remote origin* in *Chapter 4*, *Split the Load – Distributed Working with GIT*, adding a remote would be a one time operation.

We can manually open the file or do a `git log` to see the changes taking effect across repositories.

What just happened?

We have successfully synced the changes across different repositories made by different people on the same file, and the same line at the same time thereby achieving the promised collaborative work environment.

Summary

We have learned:

- The difference between files' content
- How powerful Git can be on textual files

Additionally, we have also learned how to merge and manage conflicts occurring while merging.

Not only that, we did a role play and practically learned how to set up a collaborative work environment with multiple people working on the same file, topic, and even the same line. We also learned how to integrate work from different people together to form one output.

7
Parallel Dimensions – Branching with Git

Cheap branching and merging are the two most well known and applauded features of Git. In this chapter we shall see what branching is, why you need a branch, and when you need a branch. And since we have already tried our hand at merging, having used it to merge two files for solving a merge conflict in Chapter 6, Unleash the Beast – Git on Text-based Files, we shall go one step ahead and see how to merge branches when needed.

All these are explored from an organization's point of view. We will also learn and practice methods to simplify our work by:

- Creating simple alias for frequently used lengthy commands
- Chaining of multiple commands for frequently used workflows

What is branching

Branching in Git is a function that is used to launch a separate, similar copy of the present workspace for different usage requirements. In other words branching means diverging from whatever you have been doing to a new lane where you can continue working on something else without disturbing your main line of work.

Let's understand it better with the help of the following example.

Suppose you are maintaining a checklist of some process for a department in your company, and having been impressed with how well it's structured, your superior requests you to share the checklist with another department after making some small changes specific to the department. How will you handle this situation?

An obvious way without a version control system is to save another copy of your file and make changes to the new one to fit the other department's needs. With a version control system and your current level of knowledge, perhaps you'd clone the repository and make changes to the cloned one, right?

Looking forward, there might be requirements/situations where you want to incorporate the changes that you have made to one of the copies with another one. For example, if you have discovered a typo in one copy, it's likely to be there in the other copy because both share the same source. Another thought – as your department evolves, you might realize that the customized version of the checklist that you created for the other department fits your department better than what you used to have earlier, so you want to integrate all changes made for the other department into your checklist and have a unified one.

This is the basic concept of a branch – a line of development which exists independent of another line both sharing a common history/source, which when needed can be integrated. Yes, a branch always begins life as a copy of something and from there begins a life of its own.

Almost all VCS have some form of support for such diverged workflows. But it's Git's speed and ease of execution that beats them all. This is the main reason why people refer to branching in Git as its killer feature (we'll cover the intricacies of Git branching in the next chapter).

Why do you need a branch

To understand the why part, let's think about another situation where you are working in a team where different people contribute to different pieces existing in your project.

Your entire team recently launched phase one of your project and is working towards phase two. Unfortunately, a bug that was not identified by the quality control department in the earlier phases of testing the product pops up after the release of phase one (yeah, been there, faced that!).

All of a sudden your priority shifts to fixing the bug first, thereby dropping whatever you've been doing for phase two and quickly doing a hot fix for the identified bug in phase one. But switching context derails your line of work; a thought like that might prove very costly sometimes. To handle these kind of situations you have the branching concept (refer to the next section for visuals), which allows you to work on multiple things without stepping on each other's toes.

There might be multiple branches inside a repository but there's only one active branch, which is also called current branch.

By default, since the inception of the repository, the branch named **master** is the active one and is the only branch unless and until changed explicitly.

Naming conventions

There are a bunch of naming conventions that Git enforces on its branch names; here's a list of frequently made mistakes:

- A branch name cannot contain the following:
  - A space or a white space character
  - Special characters such as colon (:), question mark (?), tilde (~), caret (^), asterisk (*), and open bracket ([)
- Forward slash (/) can be used to denote a hierarchical name, but the branch name cannot end with a slash

  For example, my/name is allowed but myname/ is not allowed, and myname\ will wait for inputs to be concatenated
  - Strings followed by a forward slash cannot begin with a dot (.)

    For example, my/.name is not valid
  - Names cannot contain two continuous dots (..) anywhere

When do you need a branch

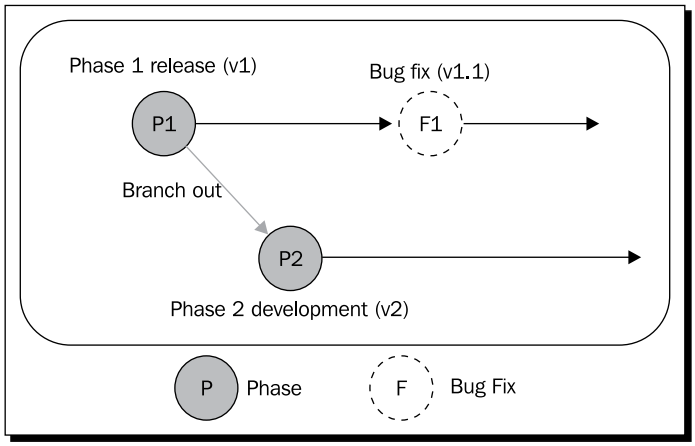
With Git, There are no hard and fast rules on when you can/need to create a branch. You can have your own technical, managerial, or even organizational reasons to do so. Following are a few to give you an idea:

- A branch in development of software applications is often used for self learning/ experimental purposes where the developer needs to try a piece of logic on the code without disturbing the actual released version of the application
- Situations like having a separate branch of source code for each customer who requires a separate set of improvements to your present package
- And the classic one – few people in the team might be working on the bug fixes of the released version, whereas the others might be working on the next phase/release

- For few workflows, you can even have separate branches for people providing their inputs, which are finally integrated to produce a release candidate

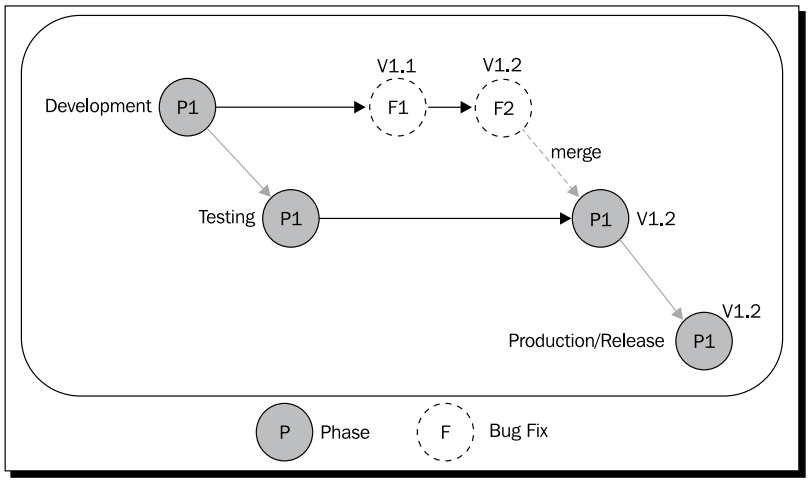
Following are flow diagrams for few workflows to help us understand the utilization of branching:

- Branching for a bug fix can have a structure as shown the following diagram:



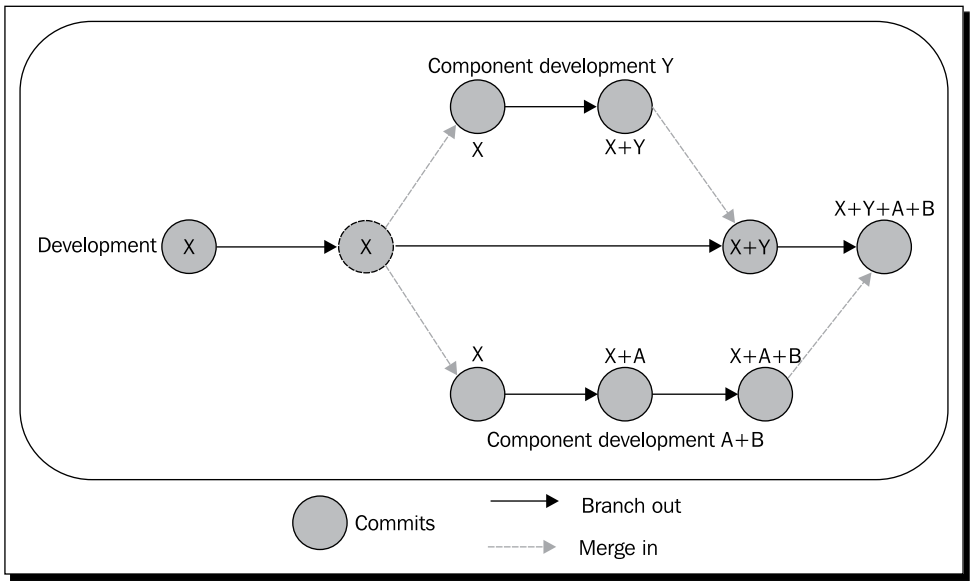
  This explains that when you are working on P2 and find a bug in P1, you need not drop your work, but switch to P1, fix it, and return back to P2.

- Branching for each promotion is as shown in the following diagram:



This explains how the same set of files can be managed across different phases/promotions. Here, P1 from development has been sent to the testing team (a branch called testing will be given to the testing team) and the bugs found are reported and fixed in the development branch (v1.1 and v1.2) and merged with the testing branch. This is then branched as production or release, which end users can access.

- Branching for each component development is as shown in the following diagram:



Here every development task/component build is a new independent branch, which when completed is merged into the main development branch.

Practice makes perfect: branching with Git

I'm sure you have got a good idea about what, why, and when you can use branches when dealing with a Git repository. Let's fortify the understanding by creating a few use cases.

Scenario

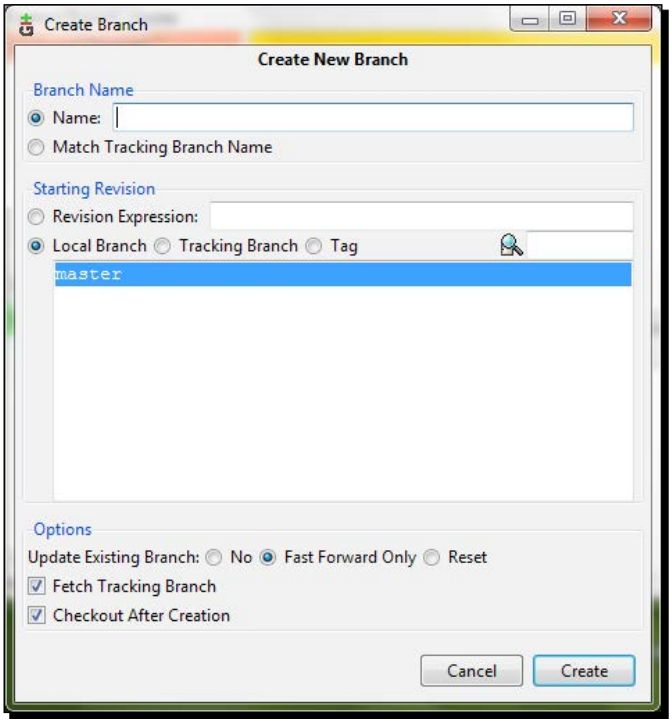
Suppose you are the training organizer in your organization and are responsible for conducting trainings as and when needed. You are preparing a list of people who you think might need business communication skills training based on their previous records.

As a first step, you need to send an e-mail to the nominations and check their availability on the specified date, and then get approval from their respective managers to allot the resource. Having experience in doing this, you are aware that the names picked by you from the records for training can have changes even at the last minute based on situations within the team. So you want to send out the initial list for each team and then proceed with your work while the list gets finalized.

Time for action – creating branches in GUI mode

Whenever you want to create a new branch using Git Gui, execute the following steps:

1. Open Git Gui for the specified repository.
2. Select the **Create** option from the **Branch** menu (or use the shortcut keys *Ctrl* + *N*), which will give you a dialog box as follows:



3. In the **Name** field, enter a branch name, leave the remaining fields as default for now, and then click on the **Create** button.

What just happened?

We have learned to create a branch using Git Gui. Now let's go through the process mentioned for the CLI mode and perform relevant actions in Git Gui.

Time for action – creating branches in CLI mode

1. Create a directory called BCT in your desktop. BCT is the acronym for Business Communication Training.
2. Let's create a text file inside the BCT directory and name it `participants`.
3. Now open the `participants.txt` file and paste the following lines in it:

   ```
   Finance team
   ```

   - `Charles`
   - `Lisa`
   - `John`
   - `Stacy`
   - `Alexander`

4. Save and close the file.
5. Initiate it as a Git repository, add all the files, and make a commit as follows:

   ```
   git init
   git add .
   git commit –m 'Initial list for finance team'
   ```

6. Now, e-mail those people followed by an e-mail to their managers and wait for the finalized list.
7. While they take their time to respond, you should go ahead and work on the next list, say for the marketing department. Create a *new branch* called `marketing` using the following syntax:

   ```
   git checkout –b marketing
   ```

8. Now open the `participants.txt` file and start entering the names for the marketing department below the finance team list, as follows:

   ```
   Marketing team
   ```

   - `Collins`
   - `Linda`
   - `Patricia`
   - `Morgan`

Before you finish finding the fifth member of the marketing team, you receive a finalized list from the finance department manager stating he can afford only three people for the training as the remaining (Alexander and Stacy) need to take care of other critical tasks. Now you need to alter the finance list and fill in the last member of the marketing department.

9. Before going back to the finance list and altering it, let's add the changes made for the marketing department and commit it.

```
git add .
git commit –m 'Unfinished list of marketing team'
git checkout master
```

10. Open the file and delete the names Alexander and Stacy, save, close, add the changes, and commit with the commit message `Final list from Finance team`.

```
git add .
git commit –m "Final list from Finance team"
git checkout marketing
```

11. Open the file and add the fifth name, Amanda, for the marketing team, save, add, and commit.

```
git add .
git commit –m "Initial list of marketing team"
```

12. Say the same names entered for marketing have been confirmed; now we need to merge these two lists, which can be done by the following command.

```
git merge master
```

13. You will get a merge conflict as shown in the following screenshot:

```
raviepic3@ARTIC-WARFARE ~/Desktop/BCT (marketing)
$ git merge master
Auto-merging participants.txt
CONFLICT (content): Merge conflict in participants.txt
Automatic merge failed; fix conflicts and then commit the result.

raviepic3@ARTIC-WARFARE ~/Desktop/BCT (marketing|MERGING)
$
```

14. Open the `participants.txt` file and resolve the merge conflict as as learned in *Chapter 6*, *Unleash the Beast – Git on Text-based Files*, then add the changes, and finally commit them.

What just happened?

Without any loss of thought or data, we have successfully adopted the changes on the first list, which came in while working on the second list, with the concept of branching – without one interfering with another.

As discussed, a branch begins its life as a copy of something else and then has a life of its own.

Here, by performing `git checkout -b branch_name` we have created a new branch from the existing position.

Technically, the so-called existing position is termed as the position of HEAD and this type of lightweight branches, which we create locally, are called topic branches. Another type of branch would be the remote branch or remote-tracking branch, which tracks somebody else's work from some other repository. We already got exposed to this while learning the concept of cloning.

The command `git checkout -b branch_name` is equivalent to executing the following two commands:

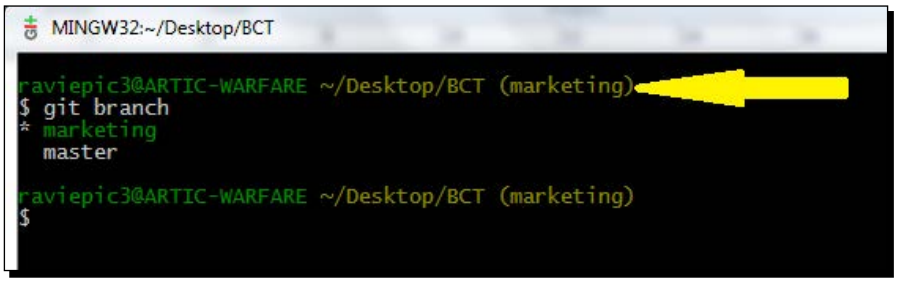
- `git branch branch_name`: Creates a new branch of the given name at the given position, but stays in the current branch
- `git checkout branch_name`: Switches you to the specified branch from the current/active branch

When a branch is created using Git Gui, the checkout process is automatically taken care of, which results in it being in the created branch.

The command `git merge branch_name` merges the current/active branch with the specified branch to incorporate the content. Note that even after the merge the branch will exist until it's deleted with the command `git branch -d branch_name`.

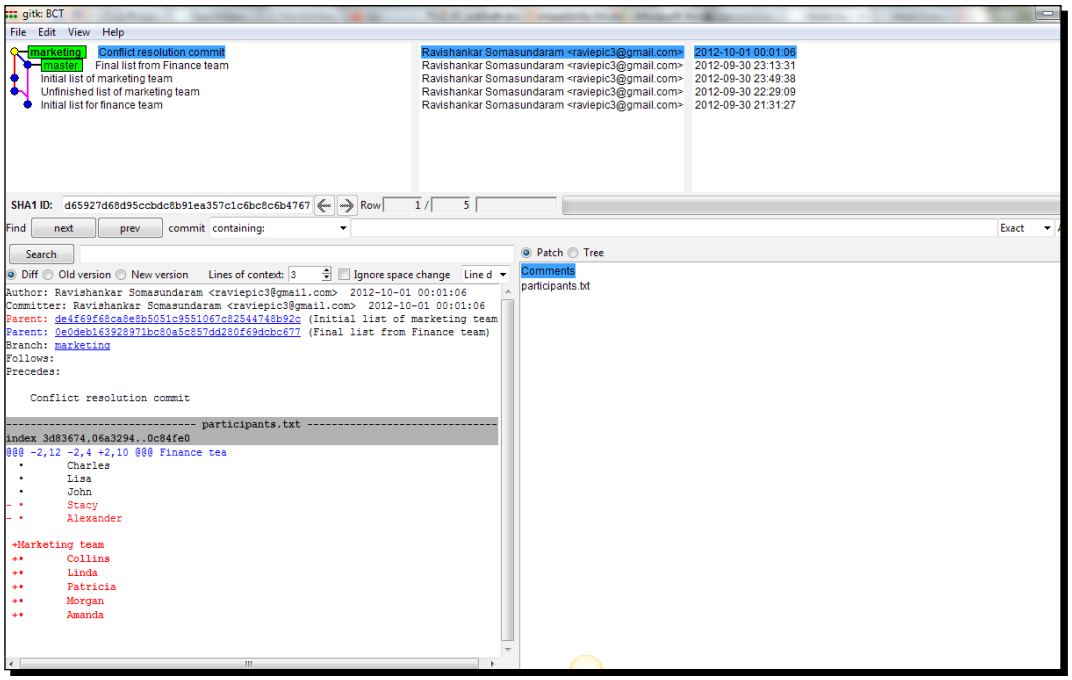
In cases where you have created and played with a branch whose content you don't want to merge with any other branch and want to simply delete the entire branch, use `-D` instead of `-d` in the command mentioned earlier.

To view a list of branches available in the system, use the command git branch as shown in the following screenshot:



As shown in the screenshot, the branches available in our BCT repository right now are **marketing** and **master**, with **master** being the *default branch* when you create a repository. The branch with a star in front of it is the active branch. To ease the process of identifying the active branch, Git displays the active branch in brackets (`branch_name`) as indicated with an arrow.

By performing this exercise we have learned to create, add content, and merge branches when needed. Now, to visually see how the history has shaped up, open gitk (by typing `gitk` in the command-line interface or by selecting **Visualize All Branch History** from the **Repository** menu of Git Gui) and view the top left corner. It will show a history like in the following screenshot:



Homework

Try to build a repository alongside the idea explained with the last flow diagram given in the *When do you need a branch* section. Have one main line branch called development and five component development branches, which should be merged in after the customizations are made to its source.

.config file – play with shortcuts

As the name conveys, this text file, which is present inside your `.git` directory, is our project/repository-specific configuration file. It can also contain aliases to commands which you frequently use. An example of adding an alias is illustrated in the following section.

Time for action – adding simple aliases using CLI

In your command-line window, type the following:

```
git config --local alias.ad add
git config --local alias.st status
```

Now open your `.config` file, which is present inside the repository with your favorite text editor and you will see the following lines at the bottom:

```
[alias]
   ad = add
   st = status
```

What just happened?

We have successfully created aliases for the Git commands `add` and `status`. To verify this, switch back to your command-line window and type the command `git st` and observe the output, which will be a spot on match to your `git status` command. Similarly we can use `git ad` as a substitute for the `git add` command.

We can also chain two or more commands with one single alias. Let's learn how to do this.

Time for action – chain commands with a single alias using CLI

As learned that the `.config` file is a plain text file, let's familiarize ourselves by opening and editing it directly this time instead of going via the command line.

1. Open your favorite text editor and the `.config` file with it if you have not done so already.

2. Due to the actions performed by the commands in the earlier section, there will be a section created at the bottom of the file called `[alias]` under which we would have entries for `ad` and `st`. Add another line after that and paste the following characters:

```
ast = !git add . && git st
bco = "!f(){ git branch ${1} && git checkout ${1}; };f"
ct = "!f(){ git commit -m \"${1}\";};f"
```

   It should appear as follows:

```
[alias]
ad = add
st = status
ast = !git add . && git st
bco = "!f(){ git branch ${1} && git checkout ${1}; };f"
ct = "!f(){ git commit -m \"${1}\";};f"
```

3. Switch back to your command line and execute the following command:

```
Git bco check_branch
```

4. Now add a new file called `testfile.txt` with some content to your repository and execute the following commands:

```
Git ast
Git ct "Created test branch, file to practice alias functionality"
```

What just happened?

We have successfully chained multiple commands under a single alias.

From now on, whenever we need to create a branch inside this repository, we can do it by using the `Git bco` command.

Similarly whenever you need to add all the changes and view the status of the repository in series, we can use the following command:

```
Git ast
```

Whenever you need to do a commit on your repository, instead of `git commit –m "your_commit_messsage_here"`, we can use the following command:

```
Git ct "your_commit_message_here"
```

Notice the difference between adding an alias through the command line and our recent modification directly on the file. The commands we have added are direct shell commands which when inserted inside the `.config` file must have a prefix of an exclamation symbol (!).

`git add .` adds all the changes made to the files present in your repository whereas the `&&` symbol is used to join another command, namely `git st` with the previous one. `Git st` displays the status of the repository. Because we have already created an alias for `status`, which is `st`, we have the convenience of using that here.

Don't get afraid on seeing the next two lines which have all those curly braces pointing at you; all you need to know is that we have written a shell script which has a function `f()` within which we have chained the commands for execution. And like discussed, any shell commands have to be prefixed with an exclamation (!) symbol.

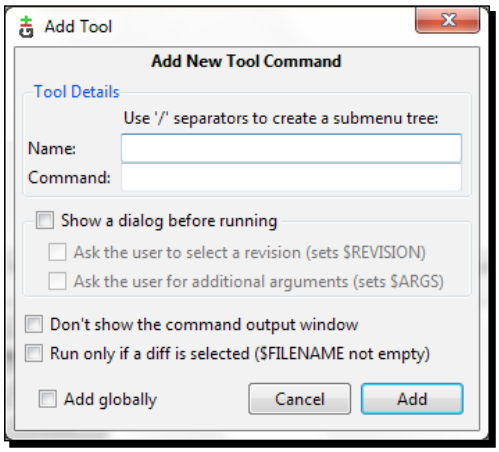
`${1}` is a magic object technically called a variable which does the job of fetching the user values (`check_branch`) and inserting them next to the command dynamically, such that wherever you use `${1}`, it's filled with the value that was provided by the user.

Note that all the configuration changes that we have made are with respect to the `.config` file of one particular repository only and hence all these customizations will stay local. To make it global these changes need to be made inside your global `.gitconfig` file. This usually resides inside the `C:\Users\your_username` directory if you are on Windows and inside the `~/` directory if you are on Mac or Linux.

Time for action – adding complex aliases using GUI

Git Gui already has shortcuts for pretty much everything you will usually need, which we have been learning as we come across different topics, so let's understand how to chain commands using Git Gui.

1. Open Git Gui and select the **Add** option from the **Tools** menu, which will give you an **Add Tool** window as follows:

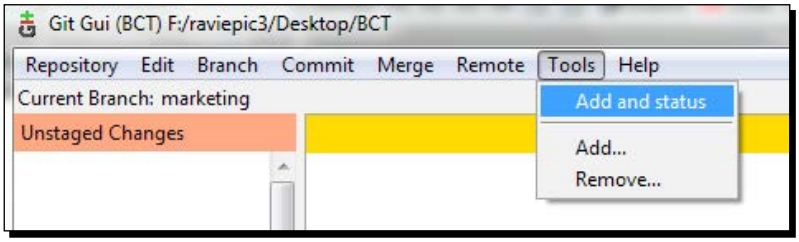


2. Enter the following values in the respective fields:

Field name	Field value
Name	`Add and status`
Command	`git add . && git status`

3. Click on the **Add** button.

   Now you will see the newly created alias as a menu item inside the **Tools** menu, as shown in the following screenshot:



What just happened?

We have practically learned that we can create comfortable aliases for lengthy commands that we frequently use. We also learned and practiced methods to combine multiple commands and execute them in order, using both the CLI and GUI modes.

Homework

Create a simple alias for `git log`.

Then, create a chain with two commands and call it `clog` (`git commit` and `git log`) in such a way that when you type `git clog "my_commit_message_here"`, your changes will shift from the **changes to be committed** state to **nothing to commit** state (which means the changes that were added but not committed are now committed with the commit message provided) continued by a listing of all the commits and their relevant details (which are usually displayed when using the command `git log`).

Summary

We have learned:

- What branching is
- How and when it can be used with different workflows

We also practiced elements on how to work with different parts of the same repository without one interfering with the other and merging these different parts to incorporate content when needed.

Additionally, we also took a dip into the usage of aliases and practically performed the creation of a simple alias for frequently used lengthy commands and the chaining of multiple commands for frequently used workflows.

8

Behind the Scenes – Basis of Git Basics

Those who had an awestruck expression on their face after experiencing the awesomeness of Git would probably be wondering about the mechanisms behind the magic.

This chapter is dedicated to users who are serious about getting to know the intricacies of the following operations:

- init
- add
- commit
- status
- clone
- fetch
- merge
- remote
- pull
- push
- tag
- branch
- checkout

We begin by understanding the composition of a Git repository, followed by an analysis of the ways in which Git intelligently manages content, and finally take an overview of ways through which Git sees the relation between relations in order to store and transfer content.

Two sides of Git: plumbing and porcelain

Irrespective of the number of features highlighted in the sales brochure for your swanky new car, it has to have a user friendly interface through which you can really appreciate and enjoy the finer things it has to offer. Though the core work is done inside, the interface outside serves as an enabler.

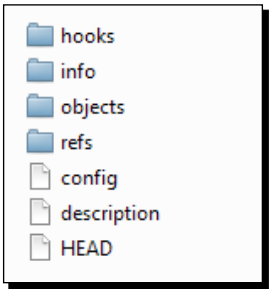
Similarly Git works on both the inner and outer levels with the following commands:

- **Plumbing commands**: These commands take care of the low level operations, which form the fundamental base on which Git is built
- **Porcelain commands**: These are the ones that cover the underlying plumbing operations at a high level with easy and appealing names for end users

The commands that we have learned in earlier chapters are of the porcelain type. Let's look behind the scenes for each one of them.

Git init

What you know is that this command creates a new subdirectory named `.git`, which is the source of versioning. Let's move one step further and explore the contents of the `.git` directory, which should have a directory structure as shown in the following screenshot:



Hooks

Hooks are customization scripts that can be injected into various Git commands and its operations. It is possible to write our own hook and such a hook has to go into this directory.

There are a bunch of sample hooks automatically created inside this directory as part of `git init` but not activated until we manually rename `hook_name.sample` to `hook_name`. To learn more about the various hooks present in the directory open up the help document by typing `git help hooks` in your command line.

Info

Additional information about the repository is recorded in this directory. Presently the only file inside would be the one called `exclude`. This file serves as a master list of the files to be excluded from being tracked by Git.

Sounds familiar, doesn't it? Indeed, the `.gitignore` file performs the same operation except for the fact that any exclusion pattern written in the exclude file is reflected only in the local repository and not in any subsequent clones; whereas when written on `.gitignore`, it becomes a part of your history, which can be subjected to other Git functions such as add, commit, merge, clone, pull, push, and others.

Config

The name conveys it all; this text file is our project/repository-specific configuration file. We would have covered the finer workings of this file in earlier chapters, but the content we'd need to cover would go beyond the scope of this book.

This is where Git maintains the entries for a remote section to or from wherever the repository is cloned or data is exchanged. It also contains some core settings such as whether the repository is a bare repository or not.

Description

There is a package called gitweb, which comes with your Git installation and will allow us to set up a web interface for our Git repositories. This means that the repository can be browsed using any web browser.

This description file contains a user-defined description of the repository, which is used by the gitweb program to display it to the clients who are requesting a listing of repositories.

Objects

As you have understood correctly, like any other VCS repository, a Git repository is nothing but a database containing all the data that is needed to retain, reproduce, and manage the revisions and history of your files, but the way Git handles these operations is what makes it stand apart from others.

And this is possible because of the way Git considers everything that goes into it as objects. There are four types of objects namely blobs, trees, commits, and tags with which it pulls such a trick.

Blob

I'm sure that you're familiar with the building blocks game; we've all played it at some point in our lives. When you think about it, you will recollect that irrespective of the type of structure you build, it's basically made up of several independent blocks put together. And when you are done with playing or want to preserve the incomplete structure to continue later on, we put it in a cover or a box and store it safely.

Similarly when it comes to handling data on a computer, irrespective of whether it's an image, or an audio or video clip, or a PDF document, it's basically constructed from several bits of binary data. A **binary large object** (**blob**) is nothing but a collection of binary data stored inside a box/cover as a single entity for later use.

Here, blobs store any type of data irrespective of their structure. They concentrate on the content alone and not on the metadata of that content – not even the location of the file or its name.

Trees

Tree objects are Git's internal representation of directories and the structure of your content. They're similar to a directory in your file system, which refers to files and/or other directories. Here, Git tree objects can refer to Git blobs and/or other Git tree objects.

Commits

The commit object holds all the metadata for changes introduced to the repository's content. Metadata includes the author for the change, the committer of the change (yes, it's possible to have two different people) along with their e-mail addresses, the date, and the time.

Tags

The tag object carries a human readable name, which can be attached to other objects, usually a commit object for easy retrieval and other reasons that we saw under the tagging topic in previous chapters.

HEAD

HEAD is like a pointer which points the Git engine to the active branch (the branch we are currently working on) for further operations. When opened using a text editor, you will see the following if you are in the master branch:

```
ref: refs/heads/master
```

And you will see the following if you are presently working on the `test_release` branch, and so on and so forth:

```
ref: refs/heads/test_release
```

Refs

If you have ever wondered how reaching `google.com` and `173.194.35.39` from your browser both give you the same Google search page, you will realize that there should be a reference somewhere that maps these two. Another simple example: bring your attendance register where everybody's name is mapped to a unique employee/student ID, which can be used to identify one person amongst several others with the same name and vice versa.

Similarly the `refs` directory serves the purpose of referencing for Git on a few operations. It stores the SHA-1 IDs of important points in the repository, such as tags and branches. Metadata for the tags is stored inside another directory situated at `refs/tags` and metadata for branches is stored inside a different directory situated at `refs/heads`.

Each branch name is a file inside the `heads` directory, and the content of such files contains the SHA-1 ID of the commit from where that particular branch was created (the parent in Git terms). The same is the case for tags as well – each tag name is a file inside the `tags` directory, which has a single SHA-1 ID for its reference.

Bumper alert – directories inside heads and tags

We have spoken about files inside both the `tags` and `heads` directory, which represent the tag and branch names that you have created in the repository. Don't get puzzled if you happen to see one or more directory structures inside the `heads` and `tags` directory.

This is simply a representation of the hierarchically structured name that one would have given for the branch or tag. Things will get much clearer after looking at the following example, which focuses on the branching concept, which is also applicable for tags.

Create a branch with the name `mybranch` (`git branch mybranch`). This will create a file called `mybranch` located at `heads/mybranch`, whereas creating a branch with an hierarchical name like `kamia/kashin` (`git branch kamia/kashin`) will create a file called `kashin` located at `heads/kamia/kashin`.

So far, we have explored the important segments of a freshly initiated (new) Git repository that has no commits as yet. However there is one more key player called index, which gets created as soon as you add content to your repository.

Index

The index file is where Git stores your staging area information to be committed. To put it simply, the content of the index file becomes your next commit. In other words this is the place where you keep your files that you want to be committed to your repository.

Git – a content tracking system

It's important for us to understand how Git perceives data; it is not through the filename or the file's location in the directory structure; rather, it emphasizes the file's content. This means that when two or more files, irrespective of where they are located inside the repository, have the same content, Git sees the relation between them through their hashes.

Computing the hash is the first task for Git before storing any data permanently. The hash value for a given content is unique across the globe. This means that the hash value for a file containing "Hello world" in your computer is the same as mine or anyone else's.

Finding out the similarities, Git puts the content under one single blob object and stores it. Note that only one copy of the content is stored in the background thus minimizing hardware usage and when asked to reproduce, it can bring out the exact storage pattern with the usage of its metadata stored with tree objects.

This hash computing happens whenever required, at various stages, thereby even a small change in one of those files will deliver a new hash, which makes Git store it separately. Since these processes revolve (with major emphasis) around the content irrespective of the file's name or location, Git is often called a content tracking system.

Git add

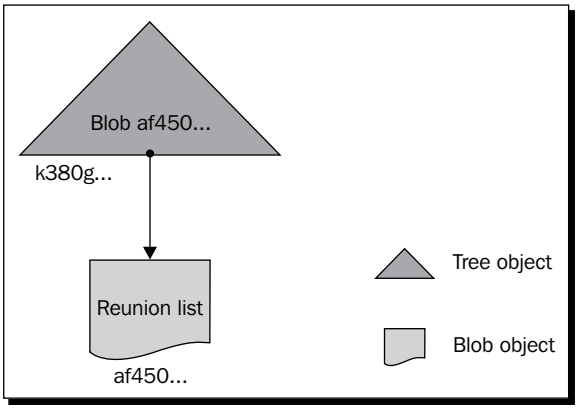
When add is executed, Git updates the index using the current content found in the working tree (staging your changes), and prepares the content staged for the next commit, which involves the following steps:

1. Computing the hash for the content.
2. Deciding whether to create new content, or link to an existing blob object.
3. Actual creation or linking of blob takes place.
4. Creation of a tree object to track location of the content.

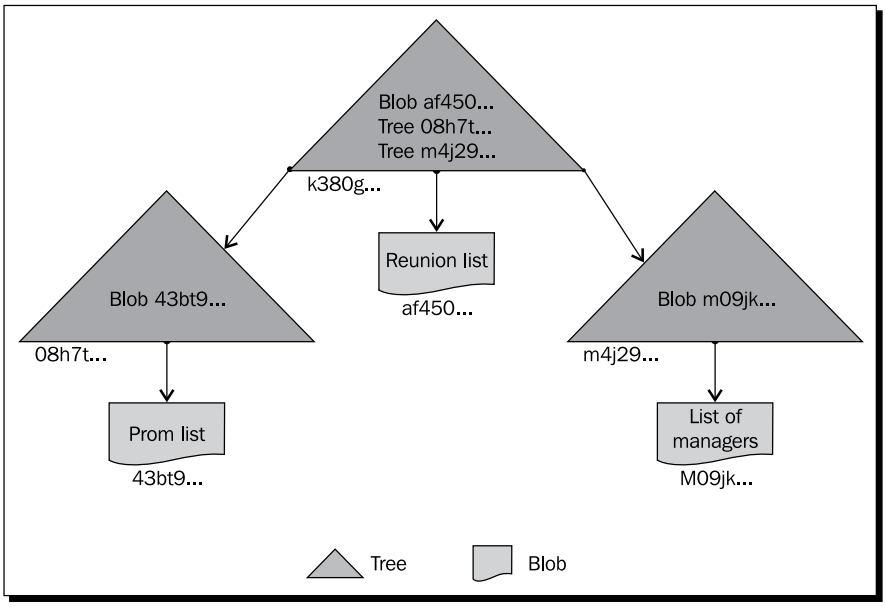
At this point the index is said to hold a snapshot of the content in the working tree for the next commit.

As you already know, this command can be performed multiple times before a commit. It only adds the content of the specified file(s) at the time the add command is run; if you want subsequent changes included in the next commit, you must run `git add` again to add the new content to the index.

More importance is to be given to the process where both the blob and tree objects get created and linked with their respective hash IDs, as shown in the following figure.



As discussed earlier, a tree can not only point to a blob but also to another tree forming a hierarchical network, as shown in the following figure:

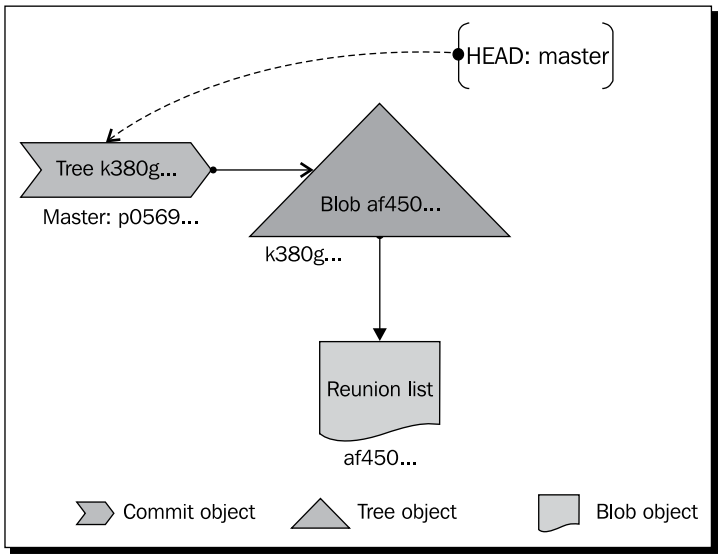


Git commit

When the `commit` command is executed, a commit object gets created with the metadata of the content/changes that were added earlier using the `git add` command. The metadata includes the following:

- Name of the person who authored the change and the relevant date and time along with the time zone settings
- Name of the person who committed the change and the relevant date and time along with the time zone settings

Then the created commit object gets linked to the tree object, which has already linked with the blob thus completing the versioning process as shown in the following figure:



Note that the head contains the branch name and not the SHA-1 ID of the commit that it is pointing to. This is because it becomes tough to identify a branch with its commit IDs when the volume and position of commits inside a branch keep changing, hence the statement "branch moves".

Do not worry about the blob and tree objects, which are created as a part of the `add` operation when not committed; these are destroyed as part of the garbage collection process after a few months.

Now if you do a `git status` you will see that the changes you staged are not in the staged changes state any longer.

Git status

When the `status` command is executed, Git checks for the file's path and size. If there are no differences, it leaves it as it is, but if any differences are found, it goes ahead and computes the hash with which it checks for a relation to other hashes, as we saw earlier.

The file path comparison as such happens in the following stages:

Stage number	Comparison	Related status message
1	File path present in index versus recent commit (HEAD commit)	Changes to be committed
2	File path present in index versus working tree	Changes not staged for commit
3	Paths in the working tree that are not tracked by Git (and are not ignored by `gitignore` or the exclude file)	Changes not staged for commit

The first status denotes changes that have already been added (staged) but not committed. So executing `git commit` would complete the versioning process.

The second and third statuses denote that the changes are not yet added (staged) for a commit. So to complete the versioning process, we need to add them first using `git add` and then `git commit`.

Git clone

When the `clone` command is executed, the internal process order would be as follows:

1. Create the destination directory if it does not exist and execute `git init` on it.
2. Set up remote tracking branches in the destination repository for each branch present in the source repository (`git remote`).
3. Fetch the objects, refs (inside the `.git` directory).
4. Finally do a checkout.

Git remote

When the `remote` command is executed, Git lists down all the remotes added to the repository by reading it from the remote section of the local config file located at `.git/config`. An example of the content inside the config file is as follows:

```
[remote "capsource"]
url = https://github.com/cappuccino/cappuccino
fetch = +refs/heads/*:refs/remotes/capsource/*
```

The name `capsource` was the alias we gave preceding the URL while adding a new remote to the repository. Under this section two reference parameters are captured:

Reference parameter	Description
`url`	This is the URL of the remote repository that you want to track, share, and get content from, within your repository.
`Fetch`	This is to convey to Git the refs (branches and tags) from the remote that are to be tracked. By default, it tracks all refs from the remote repository specified by `refs/heads/*`. These are placed under your local repository's directory `capsource` located at `refs/remotes/capsource/*`.

Git branch

When the `branch` command is executed, it performs the following steps:

1. Collects all branch names from `.git/refs/heads/`.
2. Finds the active/current working branch with the help of the entry in the HEAD located at `.git/HEAD`.
3. Displays all the branches in ascending order with an asterisk (*) mark next to the active branch.

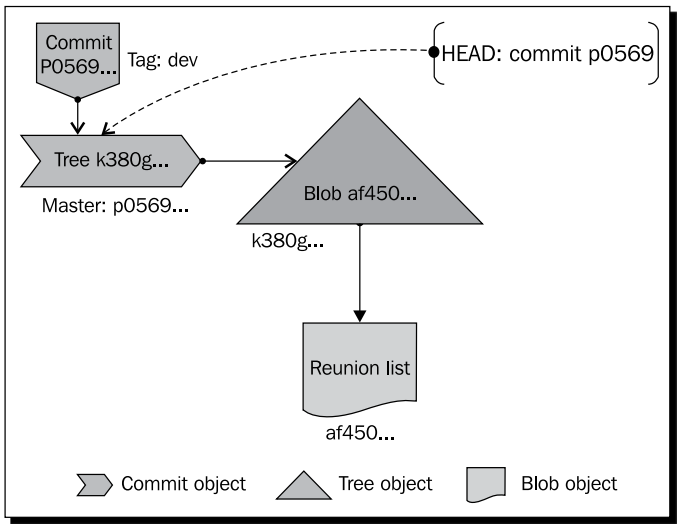
Note that the branches listed this way are only local branches of your repository. When you want all branches listed inclusive of remote tracking branches, which are stored inside `.git/refs/remotes/`, you will use `git branch –a`.

Git tag

When the `tag` command is executed, Git performs the following steps:

1. Gets the SHA-1 ID of the referred commit.
2. Validates the given tag name with the existing tag names.
3. If it's a new name, it validates the name with the naming conventions.
4. If the name abides by the rules, a tag object gets created with the given name mapped to the acquired SHA-1 ID, which is found inside `.git/refs/tags/`.

The following figure shows the association of the tag object along with other objects:



Git fetch

When `fetch` is executed, Git performs the following steps:

1. Checks for the URL or remote name, which points to a valid Git repository specified in the command `git fetch remote_name` (or) `url`.
2. If none is specified, it reads the config file to see if there is any default remote.
3. If found, it fetches the named refs (heads and tags) from the remote repository along with their associated objects.
4. The retrieved ref names are stored in `.git/FETCH_HEAD` to aid a possible merge operation in the future.

Git merge

While executing the `merge` command, Git will perform the following steps:

1. Identify both the merge candidates from the `heads` directory based on specified parameters.

2. Find the common ancestor of both heads and load all their objects in memory.
3. Perform a diff (difference) between the common ancestor and head one.
4. Apply the diff with head two.
5. If there are changes in common areas across heads, indicate the conflict with markers and inform the user about it (expecting the user to solve the conflict, add the changes, and make a commit).
6. If there are no conflicts, merge those contents, and make a merge commit mentioning metadata stating this.

Git pull

On executing the `pull` command, Git internally performs the following operations:

1. `Git fetch` with the given parameters.
2. Calls `git merge` to merge the retrieved branch head into the current branch.

Git push

On executing the `push` command, Git will perform the following:

1. Identify current branch.
2. Look up the existence of a default remote in the config file (if none is found, it prompts you to provide the remote name or URL as a parameter while executing `git push`).
3. Get to know the remote's URL and the heads (branches) tracked.
4. Check whether the remote has changed since the last time you fetched changes from it.
    - Get the list of references from the remote repositories (using `git ls-remote`).
    - Check the existence of the entries from the list with the local history. If the reference from the remote is a part of the local repository's history, it's evident that there are no other changes since the last time you fetched/pulled from the remote. So Git will allow you to directly push your changes to the remote. If it's not a part of your local repository's history, Git understands that the remote repository has undergone some changes since the last time you fetched/pulled from it. So it will ask you to first do a `git fetch` or `git pull` before pushing.

Git checkout

When `checkout` is executed without any parameters, Git performs the following steps:

1. Fetches the named paths in the working tree.
2. Fetches the related objects from the index.
3. Updates the contents of the working tree with the ones from the index.

However the behavior changes according to the parameters used.

Parameter	Description
`-b`	This is used to spawn a new branch from the checked out position mentioned with the commit ID. `git checkout –b <your_branch_name>` is a short form of `git checkout branch` followed by `git checkout <branch_name>`. This command creates a new reference inside `.git/refs/heads/` with that particular commit ID.

`--track`	This parameter is used to set up the upstream configuration usually while creating a new branch with the `–b` parameter.
	When executed, a separate section is added to the `.config` file inside the `.git` directory as follows:
	`[branch "master"]`
	`remote = origin`
	`merge = refs/heads/master`
	This happens when a command like `git checkout --track -b master origin/master` is executed.

Relation across relations – Git packfiles

We saw how Git sees the relation across files through its content and intelligently chooses between whether to create a new blob for the content or have an existing blob referenced to it. We also understood that even a small change in content will cause Git to store a separate blob because the SHA-1 ID will change.

Think about a situation where you have two text files, 5 MB each, with the same content but in different locations. Git will accordingly create a single blob as the same content will result in the same SHA-1 ID, thus saving space.

Now, append a line to the content of one of the files. Git will now create a new blob (5+ MB in size) for the second file, which has changed. Observing this behavior of having two nearly identical blobs of 5 MB, a few questions might arise.

- Why does Git create a new blob for the entire content?
- Why not still have the same old blob shared between both files, and additionally create a new blob for the difference brought into the second file alone, thus reducing storage and being more efficient?

Well, these are good questions; Git has an answer that addresses those with something called **packfiles**. The objects created as mentioned in the scenario we just discussed are called loose objects, and automatically but occasionally Git packs up several of these loose objects into a single binary called a packfile.

Transferring packfiles

Git not only supports the transferring of refs and their associated plain blob, tree, commit, and tag objects but also packfiles on operations such as clone, fetch, push, and pull. Talking on a higher level, Git has two sets of protocols for transferring data between remotes.

- One for pushing data from the client to the server
- Another for fetching data from the server to the client

Implemented side	Process invoked	Description
Server side	Upload-pack	Invoked by `git fetch-pack`, it learns what objects the other side is missing, and sends them after packing.
Client side	Fetch-pack	This is responsible for receiving missing packages from another repository. This command is usually not called directly by the end user, instead `git fetch`, which is a higher level wrapper of this command, is executed.
Server side	Receive-pack	Invoked by `git send-pack`, this receives what is pushed into the repository.
Client side	Send-pack	This is responsible for pushing objects over Git protocol to another repository. This command is usually not called directly by the end user, instead `git push`, which is a higher level wrapper of this command, is executed.

Summary

We have learned about the following:

- The structure of a Git repository and the role each one of them plays in the versioning process
- The different objects and how Git smartly manages the content using those objects

Additionally, we have also learned in detail about the internals of commands such as `init`, `add`, `commit`, `status`, `clone`, `fetch`, `merge`, `remote`, `pull`, `push`, `tag`, `branch`, and `checkout`, which we have used in earlier chapters to master the versioning concept.

Not only that, we also viewed at a high level about how Git not only understands relations between files based on their complete content but also partial content in the form of packfiles.

Index

Symbols

A

B

C

D

F

G

H

I

S

T

U

V

W

X

Thank you for buying

Git: Version Control for Everyone Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

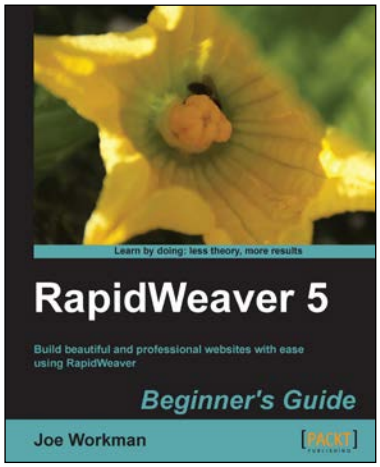
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING



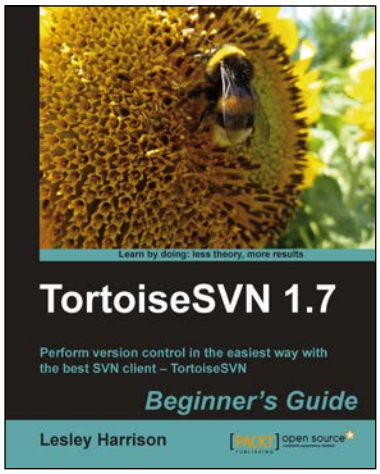
RapidWeaver 5 Beginner's Guide

ISBN: 978-1-84969-205-2 Paperback: 362 pages

Build beautiful and professional websites with ease using RapidWeaver

1. Jump into developing websites on your Mac with RapidWeaver
2. Step-by-step tutorials for novice users to get your websites built and published online
3. Advanced tips and exercises for existing RapidWeaver users
4. A great A-Z guide for building websites irrespective of your level of expertise

---

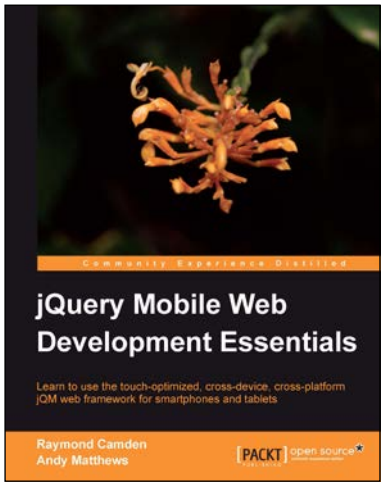


TortoiseSVN 1.7 Beginner's Guide

ISBN: 978-1-84951-344-9 Paperback: 260 pages

Perform version control in the easiest way with the best SVN client – TortoiseSVN

1. Master version control techniques with TortoiseSVN without the need for boring theory
2. Revolves around a real-world example based on a software company
3. The first and the only book that focuses on version control with TortoiseSVN
4. Reviewed by Stefan Kung, lead developer for the TortoiseSVN project



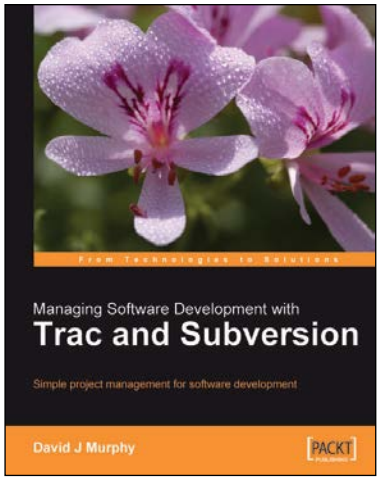
jQuery Mobile Web Development Essentials

ISBN: 978-1-84951-726-3 Paperback: 246 pages

Learn to use the touch-optimized, cross-device, cross-platform jQM web framework for smartphones and tables

1. Create websites that work beautifully on a wide range of mobile devices with jQuery mobile
2. Learn to prepare your jQuery mobile project by learning through three sample applications
3. Packed with easy to follow examples and clear explanations of how to easily build mobile-optimized websites

---



Managing Software Development with Trac and Subversion

ISBN: 978-1-84719-166-3 Paperback: 120 pages

Simple project management for software development

1. Managing software development projects simply
2. Configuring a project management server
3. Installing, configuring, and using Trac
4. Installing and using Subversion