

THE EXPERT'S VOICE®

SECOND EDITION

Pro Git

*EVERYTHING YOU NEED TO
KNOW ABOUT GIT*

Scott Chacon and Ben Straub

Apress®

Pro Git

Version 2.1.1-4-g7300157, 2021-08-17

Table of Contents

Licence	
Preface by Scott Chacon	1
Preface by Ben Straub	2
Dedications	4
Introduction	5
Pričetek	6
O nadzoru različic	8
Kratka zgodovina Git-a	8
Osnove Git	12
The Command Line	12
Git namesitev	16
Prva namestitev Git-a	16
Pridobitev pomoči	20
Povzetek	21
Osnove Git	22
Pridobitev repozitorija Git	23
Snemanje sprememb repozitorija	23
Pregled zgodovine pošiljanja	24
Razveljavljanje stvari	37
Delo z daljavami	44
Označevanje	47
Git aliasi	52
Povzetek	56
Veje Git	57
Veje na kratko	58
Osnove vej in združevanja	58
Upravljanje vej	65
Potek dela z vejami	73
Oddaljene veje	75
Ponovno baziranje (rebasing)	78
Povzetek	88
Git na strežniku	97
Protokoli	98
Pridobiti Git na strežnik	98
Generiranje vaših javnih ključev SSH	103
Nastavitev strežnika	105
Prikriti proces Git	107
Pametni HTTP	109

GitWeb	110
GitLab	112
Tretje osebne opcije gostovanja	114
Povzetek	118
Distribuirani Git	118
Distribuirani poteki dela	120
Prispevanje projektu	120
Vzdrževanje projekta	123
Povzetek	146
GitHub	161
Namestitev in konfiguracija računa	162
Prispevanje k projektu	162
Vzdrževanje projekta	167
Upravljanje organizacije	186
Skriptni GitHub	201
Povzetek	204
Orodja Git	215
Revision Selection	216
Interactive Staging	216
Stashing and Cleaning	224
Signing Your Work	228
Searching	234
Rewriting History	238
Reset Demystified	242
Advanced Merging	248
Rerere	269
Debugging with Git	288
Submodules	295
Bundling	298
Replace	317
Credential Storage	321
Povzetek	329
Prilagoditev Git-a	334
Git Configuration	335
Git Attributes	335
Git kljuka	345
An Example Git-Enforced Policy	354
Povzetek	357
Git in drugi sistemi	366
Git kot klient	367
Migracija na Git	367

Povzetek	413
Notranjost Git-a	429
Napeljava in porcelan	430
Git Objects	430
Git References	431
Packfiles	441
The Refspec	445
Transfer Protocols	448
Maintenance and Data Recovery	451
Environment Variables	456
Povzetek	464
Appendix A: Git v drugih okoljih	469
Grafični vmesniki	471
Git v Visual Studiu	471
Git v Eclipse	476
Git V Bash-u	478
Git v Zsh	478
Git v Powershell-u	479
Povzetek	481
Appendix B: Vključevanje Git-a v vašo aplikacijo	482
Git v ukazni vrstici	
Libgit2	483
JGit	483
Appendix C: Git Commands	488
Setup and Config	493
Getting and Creating Projects	493
Basic Snapshotting	494
Branching and Merging	495
Sharing and Updating Projects	497
Inspection and Comparison	500
Debugging	502
Patching	502
Email	503
External Systems	504
Administration	505
Plumbing Commands	506
Index	506
	508

Licence

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Preface by Scott Chacon

Welcome to the second edition of Pro Git. The first edition was published over four years ago now. Since then a lot has changed and yet many important things have not. While most of the core commands and concepts are still valid today as the Git core team is pretty fantastic at keeping things backward compatible, there have been some significant additions and changes in the community surrounding Git. The second edition of this book is meant to address those changes and update the book so it can be more helpful to the new user.

When I wrote the first edition, Git was still a relatively difficult to use and barely adopted tool for the harder core hacker. It was starting to gain steam in certain communities, but had not reached anywhere near the ubiquity it has today. Since then, nearly every open source community has adopted it. Git has made incredible progress on Windows, in the explosion of graphical user interfaces to it for all platforms, in IDE support and in business use. The Pro Git of four years ago knows about none of that. One of the main aims of this new edition is to touch on all of those new frontiers in the Git community.

The Open Source community using Git has also exploded. When I originally sat down to write the book nearly five years ago (it took me a while to get the first version out), I had just started working at a very little known company developing a Git hosting website called GitHub. At the time of publishing there were maybe a few thousand people using the site and just four of us working on it. As I write this introduction, GitHub is announcing our 10 millionth hosted project, with nearly 5 million registered developer accounts and over 230 employees. Love it or hate it, GitHub has heavily changed large swaths of the Open Source community in a way that was barely conceivable when I sat down to write the first edition.

I wrote a small section in the original version of Pro Git about GitHub as an example of hosted Git which I was never very comfortable with. I didn't much like that I was writing what I felt was essentially a community resource and also talking about my company in it. While I still don't love that conflict of interests, the importance of GitHub in the Git community is unavoidable. Instead of an example of Git hosting, I have decided to turn that part of the book into more deeply describing what GitHub is and how to effectively use it. If you are going to learn how to use Git then knowing how to use GitHub will help you take part in a huge community, which is valuable no matter which Git host you decide to use for your own code.

The other large change in the time since the last publishing has been the development and rise of the HTTP protocol for Git network transactions. Most of the examples in the book have been changed to HTTP from SSH because it's so much simpler.

It's been amazing to watch Git grow over the past few years from a relatively obscure version control system to basically dominating commercial and open source version control. I'm happy that Pro Git has done so well and has also been able to be one of the few technical books on the market that is both quite successful

and fully open source.

I hope you enjoy this updated edition of Pro Git.

Preface by Ben Straub

The first edition of this book is what got me hooked on Git. This was my introduction to a style of making software that felt more natural than anything I had seen before. I had been a developer for several years by then, but this was the right turn that sent me down a much more interesting path than the one I was on.

Now, years later, I'm a contributor to a major Git implementation, I've worked for the largest Git hosting company, and I've traveled the world teaching people about Git. When Scott asked if I'd be interested in working on the second edition, I didn't even have to think.

It's been a great pleasure and privilege to work on this book. I hope it helps you as much as it did me.

Dedications

To my wife, Becky, without whom this adventure never would have begun. — Ben

This edition is dedicated to my girls. To my wife Jessica who has supported me for all of these years and to my daughter Josephine, who will support me when I'm too old to know what's going on. — Scott

Introduction

You're about to spend several hours of your life reading about Git. Let's take a minute to explain what we have in store for you. Here is a quick summary of the ten chapters and three appendices of this book.

In **Chapter 1**, we're going to cover Version Control Systems (VCSs) and Git basics—no technical stuff, just what Git is, why it came about in a land full of VCSs, what sets it apart, and why so many people are using it. Then, we'll explain how to download Git and set it up for the first time if you don't already have it on your system.

In **Chapter 2**, we will go over basic Git usage—how to use Git in the 80% of cases you'll encounter most often. After reading this chapter, you should be able to clone a repository, see what has happened in the history of the project, modify files, and contribute changes. If the book spontaneously combusts at this point, you should already be pretty useful wielding Git in the time it takes you to go pick up another copy.

Chapter 3 is about the branching model in Git, often described as Git's killer feature. Here you'll learn what truly sets Git apart from the pack. When you're done, you may feel the need to spend a quiet moment pondering how you lived before Git branching was part of your life.

Chapter 4 will cover Git on the server. This chapter is for those of you who want to set up Git inside your organization or on your own personal server for collaboration. We will also explore various hosted options if you prefer to let someone else handle that for you.

Chapter 5 will go over in full detail various distributed workflows and how to accomplish them with Git. When you are done with this chapter, you should be able to work expertly with multiple remote repositories, use Git over e-mail and deftly juggle numerous remote branches and contributed patches.

Chapter 6 covers the GitHub hosting service and tooling in depth. We cover signing up for and managing an account, creating and using Git repositories, common workflows to contribute to projects and to accept contributions to yours, GitHub's programmatic interface and lots of little tips to make your life easier in general.

Chapter 7 is about advanced Git commands. Here you will learn about topics like mastering the scary *reset* command, using binary search to identify bugs, editing history, revision selection in detail, and a lot more. This chapter will round out your knowledge of Git so that you are truly a master.

Chapter 8 is about configuring your custom Git environment. This includes setting up hook scripts to enforce or encourage customized policies and using environment configuration settings so you can work the way you want to. We will also cover building your own set of scripts to enforce a custom committing policy.

Chapter 9 deals with Git and other VCSs. This includes using Git in a Subversion (SVN) world and converting projects from other VCSs to Git. A lot of organizations still use SVN and are not about to change, but by this point you'll have learned the incredible power of Git—and this chapter shows you how to cope if you still have to use a SVN server. We also cover how to import projects from several different systems in case you do convince everyone to make the plunge.

Chapter 10 delves into the murky yet beautiful depths of Git internals. Now that you know all about Git and can wield it with power and grace, you can move on to discuss how Git stores its objects, what the object model is, details of packfiles, server protocols, and more. Throughout the book, we will refer to sections of this chapter in case you feel like diving deep at that point; but if you are like us and want to dive into the technical details, you may want to read Chapter 10 first. We leave that up to you.

In **Appendix A** we look at a number of examples of using Git in various specific environments. We cover a number of different GUIs and IDE programming environments that you may want to use Git in and what is available for you. If you're interested in an overview of using Git in your shell, in Visual Studio or Eclipse, take a look here.

In **Appendix B** we explore scripting and extending Git through tools like libgit2 and JGit. If you're interested in writing complex and fast custom tools and need low level Git access, this is where you can see what that landscape looks like.

Finally in **Appendix C** we go through all the major Git commands one at a time and review where in the book we covered them and what we did with them. If you want to know where in the book we used any specific Git command you can look that up here.

Let's get started.

Pričetek

To poglavje bo o pričetku z Git-om. Pričeli bomo z razlago ozadja o orodjih za kontrolo verzij, nato se premaknili na to, kako pognati Git na vašem sistemu in končno kako ga nastaviti, da začnete z delom. Na koncu tega poglavja bi morali razumeti, zakaj je Git na voljo, zakaj bi ga morali uporabljati in pripravljeni bi morali biti za delo z njim.

O nadzoru različic

Kaj je "nadzor različic" in zakaj bi morali za to skrbeti? Nadzor različic je sistem, ki zapisuje spremembe v datoteko ali skupek datotek tekom časa, da lahko kasneje prikličete določeno različico. Za primere v tej knjigi boste uporabljali izvorno kodo programske opreme kot datoteke, ki bodo nadzirane v različicah, vendar v realnosti lahko to naredite s skoraj katerikoli tipom datotek na računalniku.

Če ste grafični ali spletni oblikovalec in želite slediti vsaki verziji slike ali postavitve (kar nadvse verjetno želite), je sistem nadzora različic (VCS) zelo modra odločitev za uporabo. Omogoča vam povrniti datoteke nazaj v prejšnje stanje, povrniti celoten projekt nazaj v prejšnje stanje, primerjati spremembe tekom časa, pogledati, ko je zadnji kaj spremenil, kar bi lahko povzročalo težavo, kdo je predstavil težavo in kdaj ter še več. Uporaba VCS tudi v splošnem pomeni, da če kaj zamočite ali izgubite datoteke, lahko enostavno stvari povrnete. Za dodatek dobite vse to za zelo majhno ceno.

Lokalni sistemi nadzora različic

Metoda izbire nadzora različic veliko ljudi je kopiranje datotek v drug direktorij (mogoče časovno označen direktorij, če so pametni). Ta pristop je zelo pogost, ker je tako enostaven, vendar je tudi zelo odprt za napake. Enostavno je pozabiti v katerem direktoriju se nahajate in po nesreči pišete v napačno datoteko ali prepisete datoteke, ki jih niste želeli.

Za spoprijemanje s to težavo so programerji že davno nazaj razvili lokalne VCS-je, ki so imeli enostavno podatkovno bazo, ki je shranjevala vse spremembe na datotekah pod nadzorom različic.

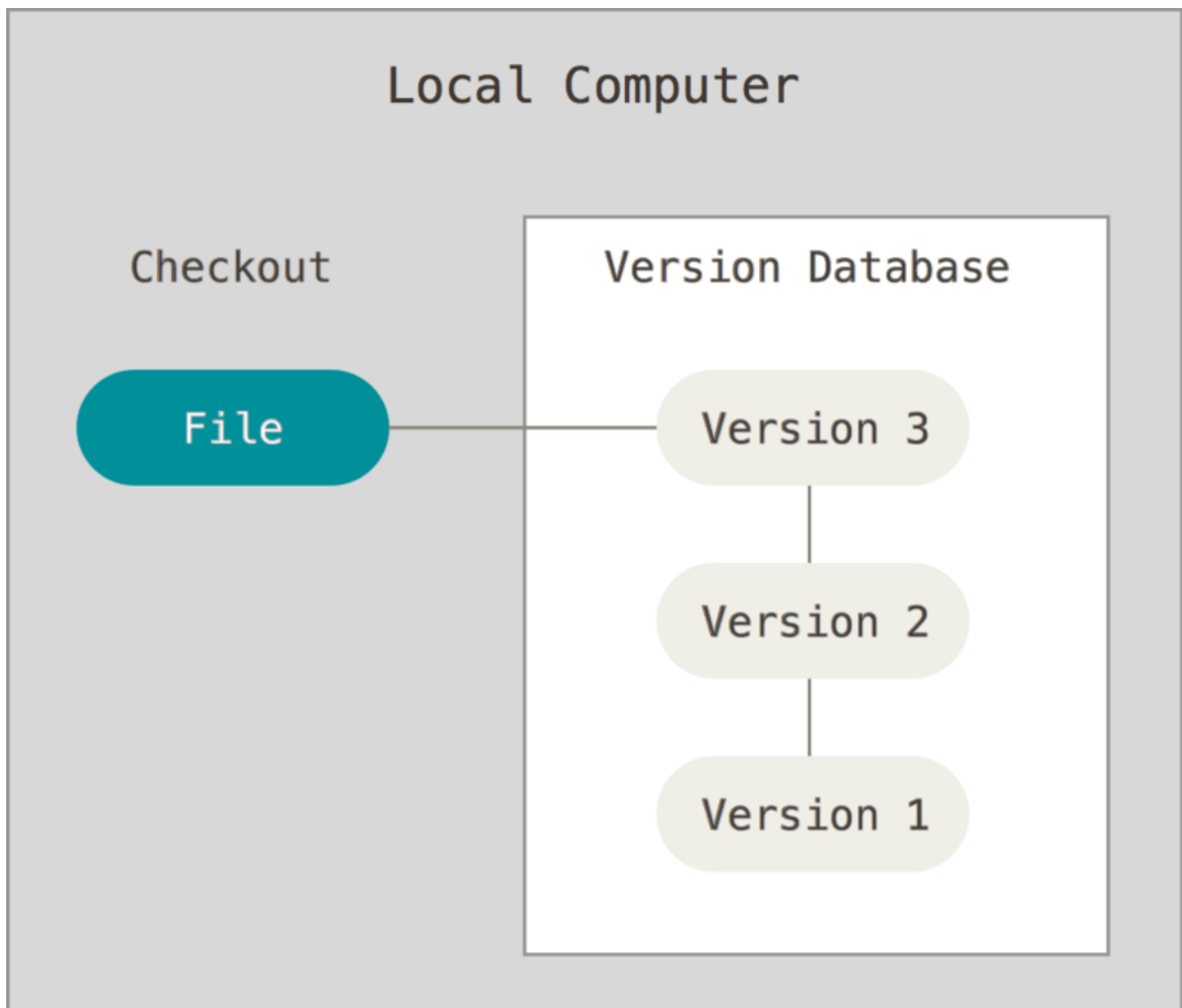


Figure 1. Local version control.

Eden priljubljenejših VCS orodij je bil sistem imenovan RCS, ki je še danes distribuiran na mnogih računalnikih. Celotno popularni Mac OS X operacijski sistem vključuje ukaz `rcs`, ko namestite razvojna orodja. RCS deluje tako, da ohranja skupke popravkov (t.j. razlike med datotekami) v posebni obliki na disku; nato lahko ponovno ustvari, kako je katerakoli datoteka izgledala v katerikoli točki časa z dodajanjem vseh popravkov.

Centralizirani sistemi nadzora različic

Naslednja glavna težava, na katero ljudje naletijo je, da potrebujejo sodelovati z razvijalci na drugih sistemih. Za spoprijemanje s tem problemom so bili razviti centralizirani sistemi nadzora različic (CVC). Ti sistemi, ki so CVS, Subversion in Perforce imajo en strežnik, ki vsebuje vse različice datotek in število klientov, ki prenesejo datoteke iz tega centralnega mesta. Za mnogo let je bil standard za nadzor različic.

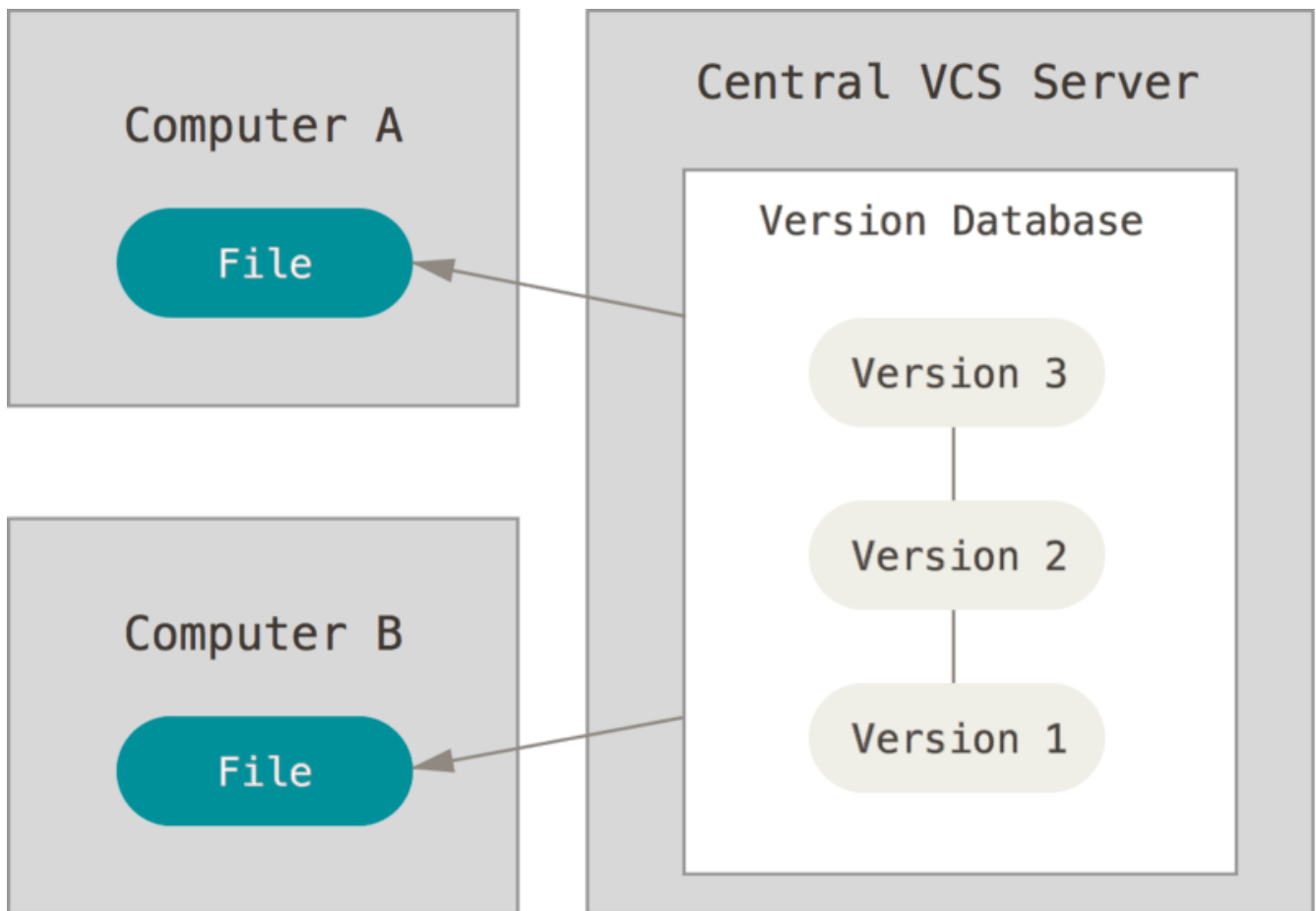


Figure 2. Centralized version control.

Ta namestitev ponuja mnogo prednosti, posebej preko lokalnih VCS-jev. Na primer, vsakdo ve do določene mere kaj kdorkoli drug na določenem projektu dela. Administratorji imajo drobno-zrnat nadzor nad te, kdo lahko kaj naredi in to je precej enostavnejše za administrirati CVCS, kot pa se spoprijemati z lokalnimi podatkovnimi bazami na vsakem klientu.

Vendar ta namestitev ima tudi nekatere resne slabosti. Najbolj očitna je odpoved ene same točke, ki jo centralizirani strežnik predstavlja. Če ta strežnik odpove za eno uro, potem med to uro nihče ne more sodelovati ali shraniti sprememb različic na karkoli, na čemer delajo. Če se trdi disk, na katerem je centralna podatkov baza, poškoduje in ustrezne varnostne kopije niso bile ohranjene, boste izgubili absolutno vse - celotno zgodovino projekta razen kateregakoli samega posnetka imajo uporabniki na svojih lokalnih napravah. Lokalni sistemi VCS trpijo za tem problemom - kadarkoli imate celotno zgodovino projekta na enem mestu, tvegate, da boste izgubili vse.

Distribuirani sistemi nadzora različic

To je mesto, kjer distribuirani sistemi nadzora različic (DVCS) pristopijo. V DVCS (kot je Git, Mercurial, Bazaar ali Darcs) klienti ne samo prenesejo zadnjega posnetka datotek: v celoti kopirajo repozitorij. V primeru, da katerikoli strežnik umre in na teh sistemih se je sodelocalo, se lahko kopira repozitorij katerega koli klienta na strežnik in se ga povrne. Vsak klon je resnično celotna varnostna kopija vseh podatkov.

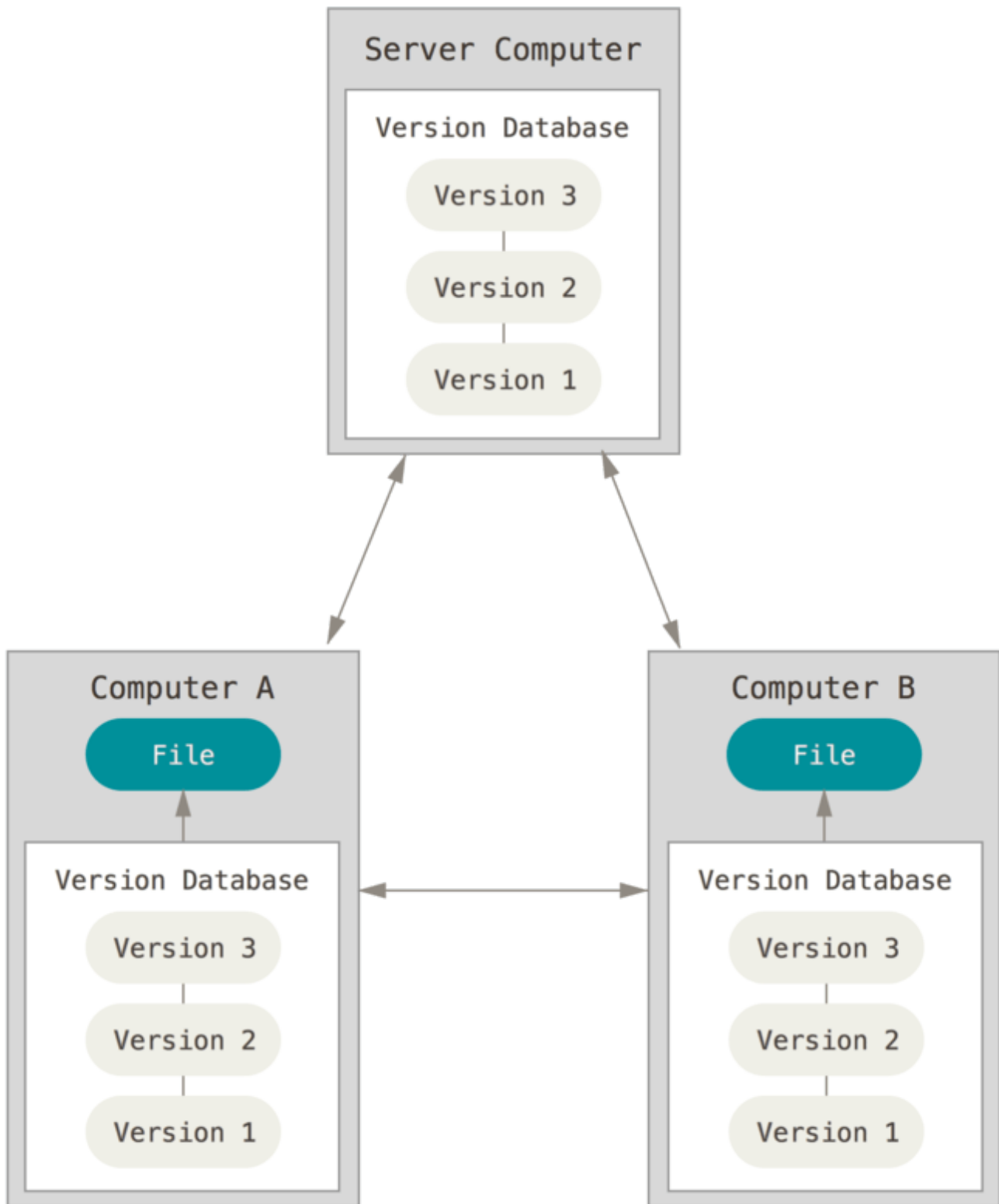


Figure 3. Distributed version control.

Dalje, mnogo teh sistemov se precej dobsto spoprijema z mnogimi oddaljenimi repozitoriji, na katerih se lahko dela, tako da lahko sodelujete z različnimi skupinami ljudi na različne načine simultano znotraj istega projekta. To vam omogoča nastaviti nekaj tipov poteka dela, ki niso možni na centraliziranih sistemih, kot so hierarhični modeli.

Kratka zgodovina Git-a

Kot z veliko drugimi stvarmi v življenju, je Git pričel z nekoliko kreativne destrukcije in ognjene kontroverznosti.

Jedro Linux je projekt odprto kodne programske opreme dokaj velikega obsega. Za večino življenske dobe vzdrževanja jedra Linux (1991-2002), so bile spremembe programske opreme poslane okrog popravkov in arhiviranih datotek. V 2002 je projekt jedra Linux pričel uporabljati lastniškega DVCS-ja imenovanega BitKeeper.

V 2005 se je odnos med skupnostjo, ki je razvijala jedro Linux in komercialnim podjetjem, ki je razvilo BitKeeper pokvaril in status brezplačnega orodja je bil preklican. To je pozvalo razvijalsko skupnost Linux (in posebej Linus Torvalds-a, ustvarjalca Linux-a), da razvije svoje lastno orodje na osnovi lekcij, ki so se jih naučili med uporabo BitKeeper-ja. Nekaj ciljev novega sistema, kot sledi:

- Hitrost
- Enostaven načrt
- Močna podpora za ne-linearno razvijanje (tisoče vzporednih vej)
- V celoti distribuirano
- Zmožnost upravljanja velikih projektov kot je jedro Linux učinkovito (hitrost in velikost podatkov)

Od njegvega rojstva v 2005 se je Git razvil in postal zrel ter enostaven za uporabo ob še vednem ohranitvi teh začetnih kvalitiet. Je izredno hiter, je zelo učinkovit na velikih projektih in ima neverjeten sistem vej za nelinearen razvoj (glejte [Veje Git](#)).

Osnove Git

Torej, kaj je Git v svoji lupini? To je pomembna sekcija za absorbiranje, ker če razumete, kaj je Git in osnove kako deluje, potem bo za vas uporaba Git-a efektivno verjetno precej veliko enostavnejša. Kot se boste učili uporabljati Git, poskusite počistiti svoj um pred stvarmi, ki jih morda veste o drugih VCS-jih, kot sta Subversion in Perforce; to vam bo pomagalo se izogniti subtilni zmešnjavi, ko uporabljate orodje. Git shranjuje in razmišlja o informacijah precej različno kot te ostali sistemi, vendar uporabniški vmesnik je precej podoben in razumevanje teh razlik vam bo pomagalo, da ne postanete še bolj zmedeni, medtem ko ga uporabljate.

Posnetki, ne razlike

Glavna razlika med Git in katerimkoli ostalim VCS-jem (Subversion in prijatelji vključeni) je način kako Git razmišlja o svojih podatkih. Konceptualno, večina ostalih sistemov shranjujejo informacije kot seznam sprememb na osnovi datotek. Ti sistemi (CVS, Subversion, Perforce, Bazaar itd) razmišljajo o informacijah, ki jih hranijo kot skupek datotek in sprememb narejenih na vsaki datoteki tekom časa.

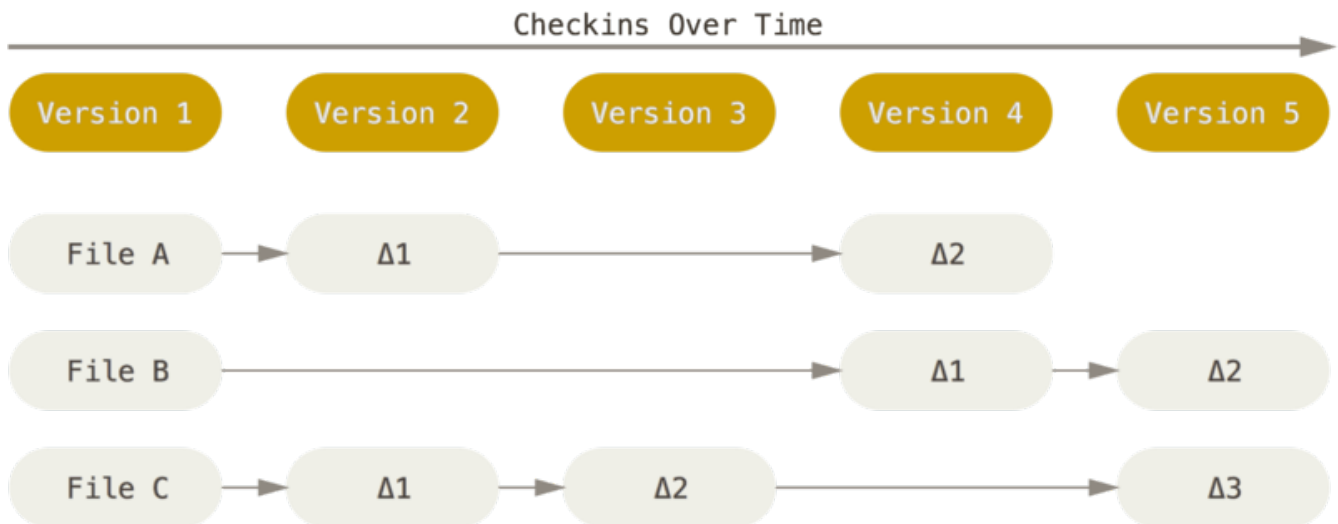


Figure 4. Storing data as changes to a base version of each file.

Git ne razmišlja o shranjevanju svojih podatkov na ta način. Namesto tega Git razmišlja o svojih podatkih bolj kot skupek posnetkov miniaturnega datotečnega sistema. Vsakič ko pošljete ali shranite stanje vašega projekta v Git v osnovi naredi sliko, kako vse vaše datoteke izgledajo v tem trenutku in shrani referenco na tisti posnetek. Za učinkovitost, če se datoteke niso spremenile, jih Git ne shrani ponovno, samo povezavo na prejšnjo identično datoteko, ki jo že ima shranjeno. Git razmišlja o svojih podatkih bolj kot **tok posnetkov**.

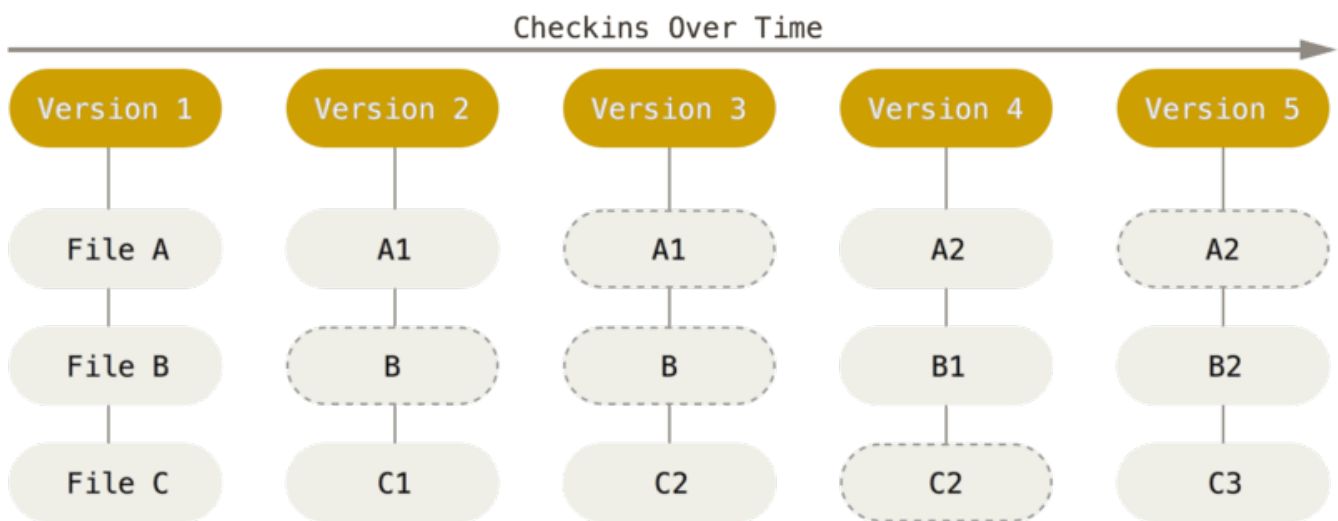


Figure 5. Storing data as snapshots of the project over time.

To je pomembna razlika med Git in skoraj vsemi ostalimi VCS-ji. Git ponovno preuči skoraj vsak aspekt nadzora različic, ki jih večina ostalih sistemov kopira iz prejšnjega generiranja. To naredi Git bolj kot mini datotečni sistem z nekaj zelo močnimi orodji zgrajenimi na njem, namesto enostavnosti pri VCS. Raziskali bomo prednosti, ki jih pridobite z razmišljanjem o vaših podatkih na ta način pri pokrivanju veje v [Veje Git](#).

Skoraj vsaka operacija je lokalna

Večina operacij v Git-u potrebuje za delovanje samo lokalne datoteke in vire - v splošnem ni potrebnih nobenih informacij iz drugega računalnika na vašem omrežju. Če ste vajeni na CVCS, kjer ima večina operacij tisto ceno latence omrežja, boste mislili, da

so bogovi hitrosti blagoslovili ta aspekt Git-a z nezemeljskimi močmi. Ker imate vso zgodovino projekta ravno tam na vašem lokalnem disku, se zdi večina operacij takojšnjih.

Na primer, za brskanje po zgodovini projekta, Git ne potrebuje iti ven na strežnik, da dobi zgodovino in jo prikaže za vas - enostavno jo prebere direktno iz vaše lokalne podatkovne baze. To pomeni, da vidite zgodovino projekta skoraj takoj. Če želite videti spremembe predstavljene med trenutno verzijo datoteke in datoteko pred mesecem, lahko Git poišče datoteko za mesec naza in naredi kalkulacijo lokalnih razlik, namesto da bi spraševal oddaljeni strežnik, da to naredi ali da potegne starejšo verzijo datoteke iz oddaljenega strežnika, da to naredi lokalno.

To tudi pomeni, da je zelo malo česar ne morete narediti, ko ste brez povezave ali brez VPN povezave. Če greste na letalo ali vlak in želite narediti nekaj dela, lahko veselo pošiljate dokler ne pridete na omrežno povezavo, da naložite. Če greste domov in ne morete pravilno nastaviti vašega VPN klienta, lahko še vedno delate. V veliko drugih sistemih je to ali nemogoče ali boleče. V Perforce-u na primer ne morete narediti veliko, ko niste povezani na strežnik; in v Subversion ter CVS lahko urejate datoteke, vendar ne morete pošiljati sprememb v vašo podatkovno bazo (ker je vaša podatkovna baza brez povezave). To mogoče ni velik problem, vendar lahko boste presenečeni, kakšno veliko razliko lahko naredi.

Git ima integriteto

Vse v Git-u je kontrolirano preko vsot preden je shranjeno in je nato sklicano glede na to kontrolno vsoto. To pomeni, da je nemogoče spremeniti vsebino katerekoli datoteke ali direktorija brez, da bi Git o tem vedel. Ta funkcionalnost je vgrajena v Git na najmanjšem nivoju in je integral njene filozofije. Ne morete izgubiti podatkov v tranzitu ali dobiti pokvarjene datoteke brez, da bi bil Git sposoben to zaznati.

Mehanika, ki jo Git uporablja za to kontroliranje vsot se imenuje SHA-1 zgoščena vrednost. To je 40-znakovni niz sestavljen iz znakov šestnajstiškega zapisa (0-9 in a-f) in preračunan na osnovi vsebine datoteke ali strukture direktorijev v Git-u. Zgoščena SHA-1 vrednost izgleda nekako takole:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Te zgoščene vrednosti boste videli v Git-u vse povsod, ker jih veliko uporablja. V bistvu Git shranjuje vse v svojo podatkovno bazo ne preko imena datoteke vendar preko zgoščene vrednosti svojih vsebin.

Git v splošnem samo doda podatke

Ko delate akcije v Git-u, skoraj vse od njih samo dodajo podatke v podatkovno bazo Git. Težko je narediti, da bo sistem naredil karkoli, česar se ne da povrniti ali izbrisati podatke na kakršen koli način. Kot v kateremkoli VCS-ju, lahko izgubite ali pokvarite spremembe, ki jih še niste poslali; vendar ko pošljete posnetek v Git, je zelo težko kaj izgubiti, posebej, če pogostokrat pošljete vašo podatkovno bazo v drug repozitorij.

To naredi uporabo Git-a užitek, ker vemo, da lahko experimentiramo brez nevarnosti po resnih uničenjih stvari. Za bolj poglobljen pogled na to, kako Git shranjuje svoje podatke in kako lahko povrnete podatke, ki se zdijo izgubljeni, gletjte [Razveljavljanje stvari](#).

Tri stanja

Sedaj, bodite pozorni. To je glavna stvar, ki si jo morate zapomniti o Git-u, če želite, da gre preostanek procesa učenja gladko. Gi ima tri glavna stanja, v katerih vaše datoteke lahko obstajajo: poslano (committed), spremenjene (modified) in dane v vmesno fazo (staged). Poslano pomeni, da so podatki varno shranjeni v vaši lokalni podatkovni bazi. Spremenjeno pomeni, da ste spremenili datoteko, vendar je še niste poslali v vašo podatkovno bazo. Uprizorjeno pomeni, da ste označili spremenjeno datoteko v njeni trenutni verziji, da gre v naslednji posnetek pošiljanja.

To nas vodi k trem glavnim sekcijam Git projekta: Git direktorij, delovni direktorij in vmesno področje.

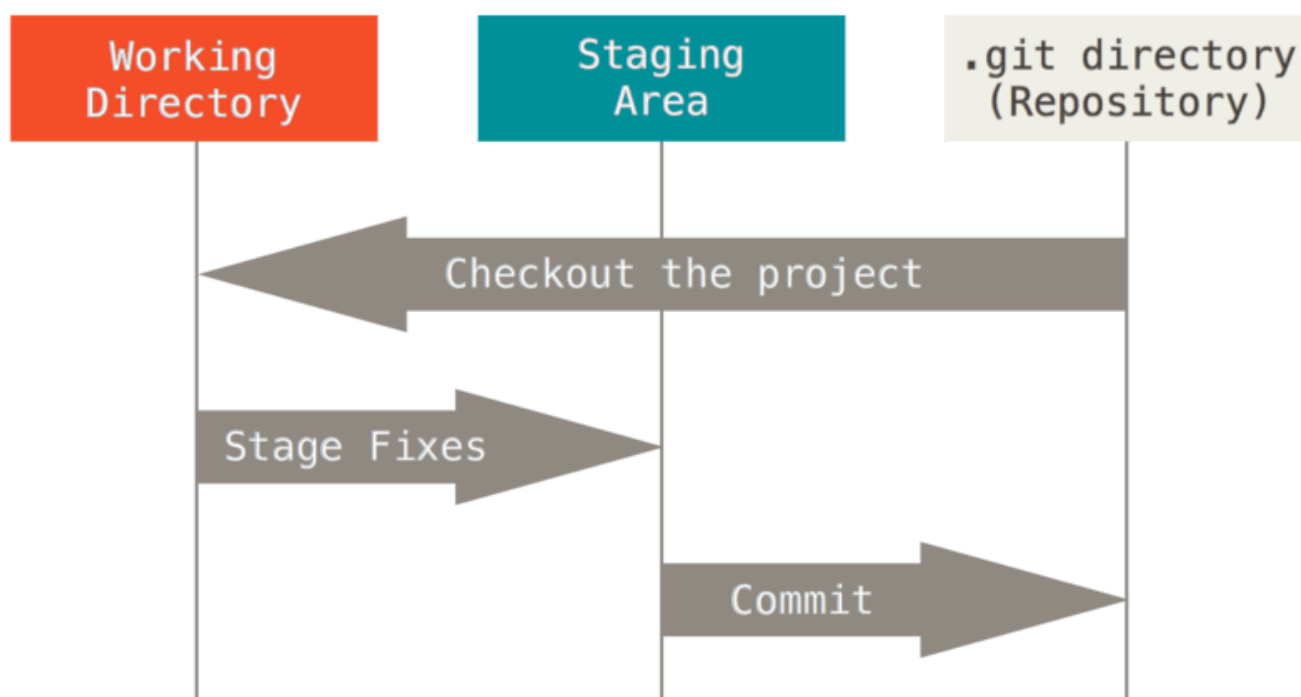


Figure 6. Working directory, staging area, and Git directory.

Git direktorij je, kjer Git shranjuje meta podatke in objektno podatkovno bazo za vaš projekt. To je najpomembnejši del Git-a in je, kar je kopirano, ko klonirate repozitorij iz drugega računalnika.

Delovni direktorij je en sam prenos ene verzije projekta. Te datoteke so potegnjene iz kompresirane podatkovne baze v Git direktoriju in podane na disk za vas, da jih uporabite ali spremenite.

Uprizoritveno področje je datoteka, v splošnem vsebovana v vašem Git direktoriju, ki shranjuje informacije o tem, kaj bo šlo v vaše naslednje pošiljanje. Včasih je sklicano kot ``index``, vendar je tudi pogosto sklicano kot vmesno področje.

Osnovni Git potek dela gre nekako takole:

1. Spremenite datoteke v vašem delovnem direktoriju.
2. Datoteke date v vmesno fazo, dodate njihove posnetke v vaše vmesno področje.
3. Jih pošljete, kar vzame datoteke kakršne so v vmesnem področju in shrani ta posnetek dokončno v vaš Git repozitorij.

Če določena verzija datoteke je v Git direktoriju, je smatrana za poslano. Če je spremenjena, vendar je bila dodana v vmesno področje, je dana v vmesno fazo. In če je bila spremenjena odkar je bila prenesena, vendar ni bila dana v vmesno fazo, je spremenjena. V poglavju [Osnove Git](#) se boste naučili več o teh stanjih in kako jih lahko ali koristite ali preskočite vmesno fazo v celoti.

The Command Line

Obstoja veliko različnih načinov za uporabo Git-a. Obstoja izvorno orodje ukazne vrstice in na voljo je mnogo grafičnih vmesnikov različnih zmogljivosti. Za to knjigo bomo uporabljali Git na ukazni vrstici. Kot prvič, ukazna vrstica je edino mesto, kjer lahko poženete **vse** ukaze Git - večina GUI-jev samo implementira nekaj podmnožic funkcionalnosti Git zaradi enostavnosti. Če veste, kako pognati verzijo ukazne vrstice, lahko verjetno tudi ugotovite, kako pognati verzijo GUI, medtem ko nasprotno ni nujno res. Tudi medtem ko je izbira grafičnega klienta stvar osebnega okusa, imajo vsi uporabniki nameščena in na voljo orodja ukazne vrstice.

Torej pričakovali bomo, da veste, kako odpreti Terminal v Mac-u ali ukazno vrstico ali Powershell v Windows. Če ne veste o čem govorimo, bi se bilo dobro ustaviti in to hitro raziskati, da lahko sledite preostalim primerom in opisom v tej knjigi.

Git namestitev

Preden začnete uporabljati Git, ga morate narediti na voljo na vašem računalniku. Četudi je že nameščen, ga je verjetno dobra ideja posodobiti na zadnjo verzijo. Before you start using Git, you have to make it available on your computer. Even if it's already installed, it's probably a good idea to update to the latest version. You can either install it as a package or via another installer, or download the source code and compile it yourself.

NOTE

Ta knjiga je bila napisana z uporabo verzije Git **2.0.0**. Čeprav bi morala večina ukazov, ki jih uporabljamo, delovati celo v starejših verzijah Git-a, nekateri od njih pa lahko ne delujejo ali delujejo nekoliko drugače, če uporabljate starejšo verzijo. Ker je Git precej odličen pri ohranjanju združljivosti za nazaj, katerakoli verzija po 2.0 bi morala delovati popolnoma v redu.

Namestitev na Linux

Če želite namestiti Git na Linux preko binarnega namestitvenega programa, lahko v

splošnem to naredite preko osnovnega orodja upravljalnika paketov, ki prihaja z vašo distribucijo. Če ste na Fedori na primer, lahko uporabite yum:

```
$ sudo yum install git
```

Če ste na distribuciji osnovani na Debian-u kot je Ubuntu, poskusite apt-get:

```
$ sudo apt-get install git
```

Za več opcij so na voljo navodila za namestitev na nekaj različnih okusih Unix-a na spletni strani Git, na <http://git-scm.com/download/linux>.

Namestitev na Mac

Na voljo je nekaj načinov za namestitev Git-a na Mac. Najenostavnejše je verjetno namestiti Xcode orodja ukazne vrstice. Na Mavericks (10.9) in višjih verzijah lahko to naredite enostavno s poskusom pogona *git* iz terminala takoj na začetku. Če ga še nimate nameščenega, vas bo pozval za namestitev.

Če želite bolj posodobljeno verzijo, lahko tudi namestite preko binarnega namestitvenega programa. OSX Git namestitveni program je vzdrževan in na voljo za preno na spletni strani Git na <http://git-scm.com/download/mac>.

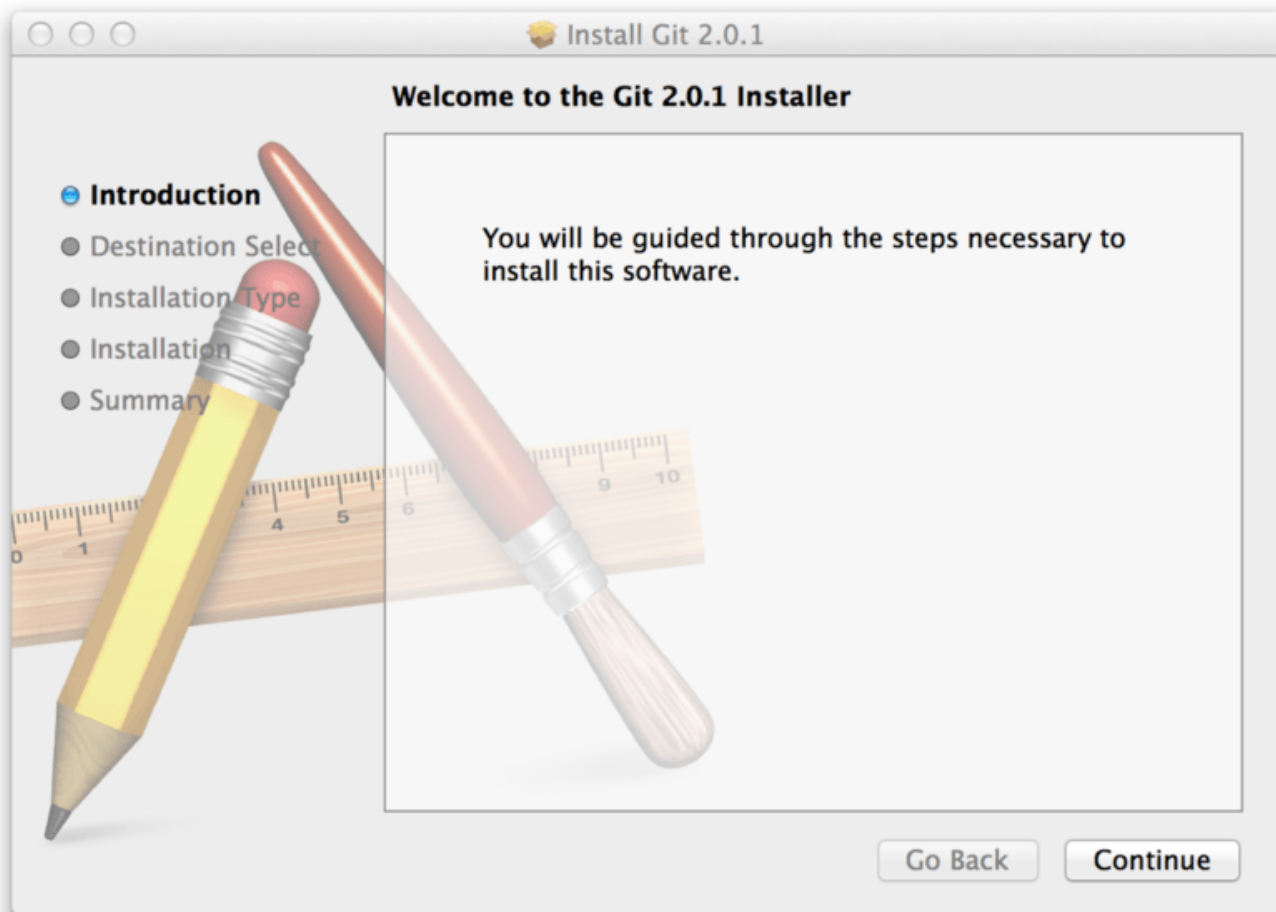


Figure 7. Git OS X Installer.

Lahko ga tudi namestite kot del GitHub-a za Mac namestitev. Njihovo GUI orodje Git ima tudi opcijo za namestitev orodij ukazne vrstice. Lahko prenesete to orodje iz GitHuba za spletno stran Mac na <http://mac.github.com>.

Namestitev na Windows

Na voljo je tudi nekaj načinov namestitve Git-a na Windows. Najbolj uradna gradnje je na voljo za prenos iz Git spletne strani. Samo obiščite <http://git-scm.com/download/win> in prenos se bo avtomatsko začel. Bodite pozorni, da ta projekt se imenuje Git za Windows (imenovan tudi msysGit), ki je ločen od samega Git-a; za več informacij o njem, pojdite na <http://msysgit.github.io/>.

Drug enostaven način, da dobite nameščen Git je namestitev GitHub-a za Windows. Namestitveni program vključuje verzijo ukazne vrstice Gita kot tudi GUI. Deluje tudi s Powershell-om in nastavi trdno predpomenje poverilnic in razumne CRLF nastavitve. Naučili se bomo več o teh stvareh nekoliko kasneje, vendar zadosti je reči, da so to stvari, ki jih želite. Lahko tudi prenesete to iz spletne strani GitHub za Windows na <http://windows.github.com>.

Namestitev iz izvirne kode

Nekateri uporabniki morda najdejo uporabnejše namestiti Git iz izvirne kode, ker dobijo najnovjšo verzijo. Binarni namestitveni program je lahko nekoliko zadaj, čeprav

kot je Git postal zrel v zadnjih nekaj letih, to naredi manj razlik.

Če želite namestiti Git iz izvorne kode, morate imeti sledeče knjižnice, od katerih je Git odvisen: curl, zlib, openssl, expat in libiconv. Na primer, če ste na sistemu, ki ima yum (kot je Fedora) ali apt-get (kot je sistem osnovan na Debian-u), lahko uporabite enega izmed teh ukazov za namestitev minimalnih odvisnosti za prevajanje in namestitev zagonskih Git datotek:

```
$ sudo yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

```
$ sudo apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

Da lahko dodate dokumentacijo različnih formatov (doc, html, info), so zahtevane sledeče odvisnosti:

```
$ sudo yum install asciidoc xmlto docbook2x
```

```
$ sudo apt-get install asciidoc xmlto docbook2x
```

Ko imate vse potrebne odvisnosti, lahko greste naprej in vzamete zadnjo označeno izdajo paketa (tarball) iz nekaj mest. Lahko ga dobite preko Kernel.org strani na <https://www.kernel.org/pub/software/scm/git>, ali zrcalne slike na spletni strani GitHub na <https://github.com/git/git/releases>. V splošnem je nekoliko jasnejše, kaj je zadnja verzija na strani GitHub, vendar stran kernel.org ima tudi podpis izdaj, če želite preveriti vaš prenos.

Nato prevedite in namestite:

```
$ tar -zxf git-2.0.0.tar.gz
$ cd git-2.0.0
$ make configure
$ ./configure --prefix=/usr
$ make all doc info
$ sudo make install install-doc install-html install-info
```

Ko je to urejeno, lahko dobite Git preko samega git-a za posodobitve:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```


Prva namestitvev Git-a

Sedaj ko imate Git na vašem sistemu, boste želeli opraviti nekaj stvari, da prilagoditev vaše okolje Git. Te stvari bi morali narediti samo enkrat na katerem koli danem računalniku; ohranile se bodo tekom nadgradenj. Lahko jih tudi kadarkoli spremenite s ponovnim pogonom ukazov.

Git prihaja z orodjem imenovanim `git config`, ki vam omogoča dobiti in nastaviti konfiguracionjske spremenljivke, ki krmilijo vse aspekte, kako Git izgleda in deluje. Te spremenljivke so lahko shranjene na treh različnih mestih:

1. `/etc/gitconfig` datoteka: Vsebuje vrednosti za vsakega uproabnika na sistemu in vse njegove repozitorije. Če podate opcijo `--system` k `git config`, bere in piše iz te datoteke posebej.
2. `~/.gitconfig` ali `~/.config/git/config` datoteka: Določa vašega uporabnika. Git lahko naredite, da bere in piše v to datoteko posebej z dodajanjem opcije `--global`.
3. Datoteka `config` v direktoriju Git (to je, `.git/config`) kateregakoli repozitorija, ki ga trenutno uporabljate: Specifičnega temu enemu repozitoriju.

Vsak nivo prepíše vrednosti iz prejšnjega nivoja, tako, da so vrednosti v `.git/config` adut tistim v `/etc/gitconfig`.

Na sistemih Windows, Git poišče datoteko `.gitconfig` v direktoriju `$HOME` (`C:\Users\%USER` za večino ljudi). Tudi še vedno pogleda v `/etc/gitconfig`, čeprav je relativno glede na MSys vrhovni direktorij, ki je kjerkoli se odločite namestiti Git na vašem sistemu Windows, ko poženete namestitveni program.

Vaša indentiteta

Prva stvar, ki jo bi morali narediti, ko nameščate Git je nastaviti vaše uporabniško ime in naslov e-pošte. To je pomembno, ker vsako Git pošiljanje uporablja te informacije in je nespremenljivo zapečeno v pošiljanje, ki ste ga začeli ustvarjati:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Ponovno morate to narediti samo enkrat, ko podate opcijo `--global`, ker Git bo vedno uporabil te informacije za karkoli boste naredili na tem sistemu. Če želite prepisati to z različnim imenom ali naslovom e-pošte za določene projekte, lahko poženete ukaz brez opcije `--global`, ko ste v tem projektu.

Mnogo GUI orodij vam bo pomagalo to narediti, ko jih prvič uporabite.

Vaš urejevalnik

Sedaj ko je vaša identiteta nastavljena, lahko nastavite privzeti tekstovni urejevalnik, ki bo uporabljen, ko Git potrebuje, da vtipkate sporočilo. Če ni nastavljen, Git

uporablja vaš privzeti urejevalnik sistema, ki je običajno Vim. Če želite uporabiti drug urejevalnik, kot je Emacs, lahko to naredite sledeče:

```
$ git config --global core.editor emacs
```

WARNING

Vim in Emacs sta popularna urejevalnika besedil pogosto uporabljena s strani razvijalcev na sistemih osnovanih na Unix-u kot sta Linux in Mac. Če niste seznanjeni z nobenim od teh urejevalnikov ali ste na sistemu Windows, boste morda potrebovali poiskati navodila, kako nastaviti vaš priljubljeni urejevalnik z Git-om. Če ne nastavite urejevalnika na ta način in ne veste kaj sta Vim ali Emac, boste verjetno v precej nerodni situaciji, ko bosta zagnana.

Preverjanje vaših nastavitev

Če želite preveriti vaše nastavitve, lahko uporabite ukaz `git config --list` za izpis vseh nastavitev, ki jih lahko Git najde v tistem trenutku:

```
$ git config --list
user.name=John Doe
user.email=johndoe@example.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Lahko boste videli ključe več kot enkrat, ker Git prebere isti ključ iz različnih datotek (`/etc/gitconfig` in `~/.gitconfig` na primer). V tem primeru Git uporablja zadnjo vrednost za vsak unikatni ključ, ki ga vidi.

Lahko tudi preverite, kaj Git razmišlja o določeni vrednosti ključa z vtipkanjem `git config <key>`:

```
$ git config user.name
John Doe
```

Pridobitev pomoči

Če kadarkoli potrebujete pomoč med uporabo Git-a, so na voljo trije načini, da dobite pomoč strani priročnika (manpage) za katerikoli ukaz Git:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

Na primer, dobite lahko pomoč manpage za ukaza config, če poženetete

```
$ git help config
```

Ti ukazi so lepi, ker lahko do njih dostopate kjerkoli, celo brez povezave. Če stani priročnika in ta knjiga niso dovolj in potrebujete pomoč preko osebe, lahko poskusite `#git` ali `#github` kanal na Freenode IRC strežniku (irc.freenode.net). Ti kanali so pogosto napolnjeni s stotinami ljudi, ki velik ovedo o Git-u in so pogosto pripravljeni pomagati.

Povzetek

Morali bi imeti osnovno znanje o tem, kaj je Git in kako se razlikuje od centraliziranih sistemov kontrole verzij, ki ste jih morda uporabljali prej. Sedaj bi tudi morali imeti delujočo verzijo Git-a na vašem sistemu, ki je nastavljen z vašo osebno identiteto. Sedaj je čas, da se naučite nekaj Git osnov.

Osnove Git

Če lahko preberete samo eno poglavje, da pričnete z Git-om, je to to. To poglavje pokriva vsak osnovni ukaz, ki ga potrebujete izvesti za glavnino stvari, za katere boste eventuelno porabili čas pri opravljanju z Git-om. Do konca tega poglavja bi morali biti zmožni nastaviti in inicializirati repozitorij, začeti in ustaviti sledenje datotek, jih dati v vmesno fazo in poslati spremembe. Pokazali vam bomo tudi, kako nastaviti Git, da ignorira določene datoteke in vzorce datotek, kako razveljaviti napake hitro in enostavno, kako brskati po zgodovini vašega projekta in pogledati spremembe med pošiljanji ter kako potisniti in povleci iz oddaljenega repozitorija.

Pridobitev repozitorija Git

Projekt Git lahko dobite z dvema glavnima pristopoma. Prvi vzame obstoječi projekt ali direktorij in ga uvozi v Git. Drugi klonira obstoječi Git repozitorij iz drugega strežnika.

Inicializacija repozitorija v obstoječi direktorij

Če pričenjate slediti obstoječi projekt v Git-u, boste morali iti v direktorij projekta in vpisati

```
$ git init
```

To ustvari nov poddirektorij imenovan `.git`, ki vsebuje vse vaše potrebne datoteke repozitorija - skelet Git repozitorija. Na tej točki ni še nič sledeno v vašem projektu. (Glejte [Notranjost Git-a](#) za več informacij o točno, katere datoteke so vsebovane v direktoriju `.git`, ki ste ga ravno ustvarili.)

Če želite začeti kontrolo verzij obstoječih datotek (v nasprotnem praznem direktoriju), bi morali verjetno začeti slediti tem datotekam in narediti začetno pošiljanje. To lahko naredite z nekaj `git add` ukazi, ki določajo datoteke, katerim želite slediti, ter nato `git commit`:

```
$ git add *.c
$ git add LICENSE
$ git commit -m 'initial project version'
```

Šli bomo skozi, kaj te ukazi naredijo v samo minuti. Na tej točki, imate Git repozitorij s sledenimi datotekami in začetnim pošiljanjem.

Kloniranje obstoječega repozitorija

Če želite dobiti kopijo obstoječega repozitorija Git - na primer projekt, kateremu želite prispevati - je ukaz, ki ga potrebujete `git clone`. Če ste že seznanjeni z ostalimi VCS sistemi, kot je Subversion, boste opazili, da je ukaz "clone" in ne "checkout". To je

pomembna razlika - namesto, da dobite samo delovno kopijo, Git dobi polno kopijo od skoraj vseh podatkov, ki jih strežnik ima. Vsaka verzija vsake datoteke za zgodovino projekta je potegnjena privzeto, ko poženete `git clone`. V bistvu, če se vaš disk strežnika pokvari, lahko pogosto uporabite skoraj katerikoli klon katerega klienta, da nastavite strežnik nazaj v stanje, v katerem je bil, ko je bil kloniran (mora boste izgubili nekatere kovelje strežniške strani in podobno, vendar vsi podatki v verzijah bi bili tam - glejte [Pridobiti Git na strežnik](#) za več podrobnosti).

Repozitorij klonirate z `git clone [url]`. Na primer, če želite klonirati Git povezano knjižnico imenovano `libgit2`, lahko to naredite sledeče:

```
$ git clone https://github.com/libgit2/libgit2
```

To ustvari direktorij imenovan "libgit2", inicializira `.git` direktorij znotraj njega, potegne vse podatke za ta repozitorij in preveri delovno kopijo zadnje verzije. Če greste v novi `libgit2` direktorij, boste tam videli datoteke projekta, pripravljene za delo ali uporabo. Če želite klonirati repozitorij v direktorij imenovan nekaj drugega kot "libgit2", lahko to določite kot naslednjo opcijo ukazne vrstice:

```
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

Ukaz naredi enako stvar kot prejšnji, vendar ciljni direktorij je imenovan `mylibgit`.

Git ima število različnih prenosnih protokolov, ki jih lahko uporabite. Prejšnji primer uporablja `https://` protokol, vendar lahko pogledate tudi `git://` ali `user@server:path/to/repo.git`, ki uporablja SSH prenosni protokol. [Pridobiti Git na strežnik](#) vam bo predstavil vse opcije, ki so na voljo in jih lahko strežnik nastavi za dostopanje vašega Git repozitorija ter prednosti in slabosti vsake.

Snemanje sprememb repozitorija

Imate izdelan repozitorij Git in izpis ali delovno kopijo datotek za ta projekt. Narediti moreate nekaj sprememb in poslati posnetke teh sprememb v vaš repozitorij vsakič, ko projekt doseže stanje, ki ga želite posneti.

Pomnite, da je lahko vsaka datoteka v vašem delovnem direktoriju v dveh stanjih: sledena ali nesledena. Sledene datoteke so datoteke, ki so bile v zadnjem posnetku; so lahko nespremenjene, spremenjene ali dane v vmesno fazo. Nesledene datoteke so vse ostale - katerakoli datoteka v vašem delovnem direktoriju, ki ni bila v vašem zadnjem posnetku in ni v vašem področju vmesne faze. Ko prvič klonirate repozitorij, bodo vse vaše datoteke sledene in nespremenjene, ker ste jih ravnokar izpisali in jih niste kakorkoli urejali.

Kot boste urejali datoteke, jih Git vidi kot spremenjene, ker ste jih spremenili od zadnjega pošiljanja. Te spremenjene datoteke date v vmesno fazo in nato pošljete vse vaše spremembe v vmesni fazi in cikel se ponovi.

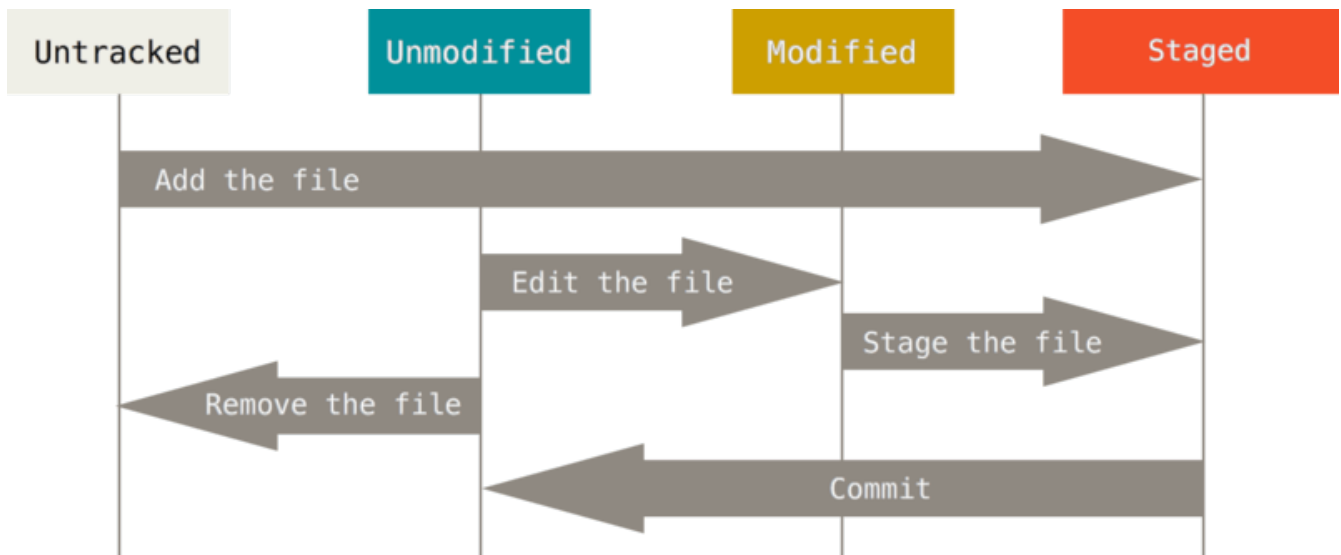


Figure 8. The lifecycle of the status of your files.

Preverjanje status vaših datotek

Glavno orodje, ki ga uporabljate, da določite katere datoteke so v kakšnem stanju je ukaz `git status`. Če ta ukaz poženete direktno po kloniranju, bi morali videti nekaj takega:

```
$ git status
On branch master
nothing to commit, working directory clean
```

To pomeni, da imate čisti delovni direktorij - z drugimi besedami, ni sledenih ali spremenjenih datotek. Git tudi ne vidi kakršnihkoli nesledenih datotek, drugače bi bile tu izpisane. Končno ukaz vam pove na kateri veji ste in vas obvesti, da ne izhaja iz iste veje na strežniku. Za sedaj je ta veja vedno “master”, kar je privzeto; o tem ne boste tu skrbeli. [Veje Git](#) bo šlo čez veje in reference v podrobnosti.

Recimo, da dodate novo datoteko v vaš projekt, enostavna datoteka README. Če datoteka prej še ni obstajala, in poženete `git status`, boste videli vašo nesledeno datoteko kot:

```
$ echo 'My Project' > README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

  README

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

Vidite lahko, da vaša nova datoteka README ni sledena, ker je pod “Untracked files”, ki kaže v vaš izpis statusa. Nesledeno v osnovi pomeni, da Git vidi datoteko, ki je niste imeli v prejšnjem posnetku (commit); Git je ne bo začel vključevati v vaše

poslane posnetke dokler mu tega eksplicitno ne poveste.

To dela zato, da po ne sreči ne začnete vključevati generiranih binarnih datotek ali ostalih datotek, ki jih niste mislili vključiti. Hoteli boste začeti vključiti README, tako da začnimo s sledenjem datoteke.

Tracking New Files

Da začnete slediti novi datoteki, uporabite ukaz `git add`. Da začnete slediti datoteki README, lahko pošete sledeče:

```
$ git add README
```

Če ponovno pošete vaš ukaz statusa, lahko vidite, da je vaša datoteka README sedaj sledena in dana v vmesno fazo za pošiljanje:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   README
```

Lahko poveste, da je dana v vmesno fazo, ker je pod glavo "Changes to be committed". Če pošljete na tej točki, bo verzija datoteke v času, ko ste pognali `git add` v zgodovini posnetka. Morda se spomnite, da ko ste prej pognali `git init`, ste nato pognali `git add (files)` - to je bil začetek sledenja datotek v vašem direktoriju.

Ukaz `git add` vzame ime poti za ali datoteko ali direktorij; če je direktorij, ukaz doda vse datoteke v tem direktoriju rekurzivno.

Staging Modified Files

Spremenimo datoteko, ki je bila že sledena. Če spremenite prej sledeno datoteko imenovano "CONTRIBUTING.md" in nato pošete vaš `git status` ukaz ponovno, dobite nekaj, kar izgleda takole:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

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

    modified:   CONTRIBUTING.md
```

Datoteka "CONTRIBUTING.md" se pojavi pod sekcijo imenovano "Changed but not staged for commit" - kar pomeni, da je sledena datoteka bila spremenjena v delujočem direktoriju, vendar še ni bila dana v vmesno fazo. Za dodajanje v vmesno fazo, pošete ukaz `git add`. `git add` je ukaz z večimi pomeni - uporabite ga za začetek sledenja novih datotek, da date datoteke v vmesno fazo in naredite druge stvari kot je označevanje konfliktov združevanja za rešene. Lahko je v pomoč razmišljanje o tem bolj v smislu "dodajte to vsebino naslednjemu pošiljanju" kot pa "dodajte to datoteku projektu". Poženimo `git add` sedaj za dajanje v vmesno fazo datoteki "CONTRIBUTING.md" in nato ponovno pošete `git status`:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Obe datoteki sta dani v vmesno fazo in bosta šli v vaše naslednje pošiljanje. Na tej točki predpostavimo, da se spomnite neke majhne spremembe, ki jo želite narediti v `CONTRIBUTING.md` preden jo pošljete. Ponovno jo odprete in naredite to spremembo in že ste pripravljeni na pošiljanje. Vendar pošete `git status` še enkrat:


```
$ vim CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

Kaj za vruga? Sedaj je `CONTRIBUTING.md` izpisan tako kot vmesna faza *kot tudi* brez vmesne faze. Kako je to mogoče? Izkaže se, da Git da datoteko v vmesno fazo točno tako kot je, ko poženete ukaz `git add`. Če pošljete sedaj, bo verzija `CONTRIBUTING.md` kakršna je bila, ko ste nazadnje pognali ukaz `git add` in bo šla v pošiljanje, ne pa verzija datoteke kot izgleda v vašem delovnem direktoriju, ko poženete ukaz `git commit`. Če spremenite datoteko po tem, ko poženete `git add`, morate pognati `git add` ponovno, da date v vmesno fazo zadnjo verzijo datoteke:

```
$ git add CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README
    modified:   CONTRIBUTING.md
```

Kratek status

Medtem ko je izpis `git status` precej celovit, je tudi precej gostobeseden. Git ima tudi kratko zastavico statusa, da lahko vidite vaše spremembe na bolj kompakten način. Če poženete `git status -s` ali `git status --short` dobite veliko bolj poenostavljen izpis iz ukaza.

```
$ git status -s
M README
MM Rakefile
A lib/git.rb
M lib/simplegit.rb
?? LICENSE.txt
```

Nove datoteke, ki niso sledene imajo zraven njih `??`, nove datoteke, ki so bile dodane v vmesno fazo imajo `A`, spremenjene datoteke imajo `M` in tako dalje. Obstajata dva stolpca za izpis - levi stolpec označuje, da je bila datoteka dana v vmesno fazo in desni stolpec označuje, da je spremenjena. Torej na primer v tem izpisu je datoteka `README` spremenjena v delovnem direktoriju, vendar še ni dana v vmesno fazo, medtem kot je datoteka `lib/simplegit.rb` spremenjena in dana v vmesno fazo. `Rakefile` je bila spremenjena, dana v vmesno fazo in nato ponovno spremenjena, torej so spremembe na njej, ki so tako dane v vmesno fazo in ne.

Ignoriranje datotek

Pogostokrat boste imeli razred datotek, ki jih ne želite, da jih Git avtomatično doda ali celo prikazuje kot sledene. Te so v splošnem avtomatsko generirane datoteke, kot so datoteke dnevnika ali datoteke producirane z vašim sistemom gradnje. V teh primerih lahko ustvarite vzorec seznama datotek, ki se ujema z imeni `.gitignore`. Tu je primer `.gitignore` datoteke:

```
$ cat .gitignore
*.oa
*~
```

Prva vrstica pove Git-u, naj ignorira katerekoli datoteke, ki se končajo z `“.o”` ali `“.a”` - objekti in arhivske datoteke, ki so lahko produkt gradnje vaše kode. Druga vrstica pove Git-u naj ignorira vse datoteke, ki se končajo s tilde (`~`), ki je uporabljena s strani mnogih tekstovni urejevalnikov kot je Emacs, da označuje začasne datoteke. Lahko tudi vključite dnevnik, tmp ali pid direktorij; avtomatsko generirano dokumentacijo; itd. Nastavitev `.gitignore` datoteke preden pričnete je v splošnem dobra ideja, da po ne sreči ne pošljete datotek, ki jih v resnici ne želite imeti v vašem Git repozitoriju.

Pravila vzorcev, ki jih lahko vključite v `.gitignore` datoteki so sledeča:

- Prazne vrstice ali vrstice, ki se začnejo z `#` so ignorirane.
- Standardni glob vzorci delujejo.
- Vzorce lahko začnete poševnico (`/`), da se izognete rekurziji.
- Lahko zaključite vzorce s poševnico (`/`), da določite direktorij.
- Lahko negirate vzorec tako, da ga začnete s klicajem (`!`).

Glob vzorci so verjetno poenostavljeni splošni izrazi, ki jih lupina uporablja. Zvezdica (`*`) se ujema z nič ali več znaki; `[abc]` se ujema s katerimkoli znakom znotraj oglatih oklepajev (v tem primeru a, b, ali c); vprašaj (`?`) se ujema z enim znakom; in zaviti oklepaji, ki zapirajo znake ločene s pomišljaji (`[0-9]`) se ujema s katerim koli znakom med njimi (v tem primeru 0 do 9). Lahko uporabite dve zvezdici, da se ujema direktorije; `a/**/z` se ujema z `a/z`, `a/b/z` `a/b/c/z` itd.

Tu je drug primer datoteke `.gitignore`:

```
# no .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in the build/ directory
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .txt files in the doc/ directory
doc/**/*.*txt
```

TIP

GitHub upravlja precej zgoščen seznam dobrih primerov `.gitignore` datotek za ducate projektov in jezikov na <https://github.com/github/gitignore>, če želite začetno točko za vaš projekt.

Ogled vaših sprememb v vmesni fazi in izven vmesne faze

Če je ukaz `git status` za vas preveč nejasen - želite vedeti točno, kaj ste spremenili, ne samo katere datoteke so bile spremenjene - lahko uporabite ukaz `git diff`. `git diff` bomo pokrili v več podrobnostih kasneje, vendar ga boste uporabljali najpogosteje za odgovor na ti dve vprašanji: Kaj ste spremenili vendar še ni dano v vmesno fazo? In kaj ste dali v vmesno fazo, da boste poslali? Čeprav `git status` odgovori ta vprašanja zelo splošno z izpisom seznama imen datotek, vam `git diff` prikaže točne vrstice, ki so bile dodane in odstranjene - popravek kot je bil.

Recimo, da urejate in dajete v vmesno fazo datoteko `README` in nato uredite datoteko `CONTRIBUTING.md` brez dajanja v vmesno fazo. Če pošete vaš ukaz `git status`, ponovno vidite nekaj takega:

```

$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   README

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

    modified:   CONTRIBUTING.md

```

Da vidite, kaj ste spremenili, vendar ni še dano v vmesno fazo, vtipkajte `git diff` brez nobenih argumentov:

```

$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
 Please include a nice description of your changes when you submit your PR;
 if we have to read the whole diff to figure out why you're contributing
 in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.

If you are starting to work on a particular area, feel free to submit a PR
that highlights your work in progress (and note in the PR title that it's

```

Ukaz primerja, kaj je v vašem delovnem direktoriju s tem, kar je v vaši vmesni fazi. Rezultat vam pove spremembe, ki ste jih naredili in ki niso še dane v vmesno fazo.

Če želite videti, kaj ste dali v vmesno fazo in bo šlo v vaše naslednje pošiljanje, lahko uporabite `git diff --staged`. Ta ukaz primerja vaše spremembe dane v vmesno fazo z vašim zadnjim pošiljanjem:

```

$ git diff --staged
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+My Project

```

Pomembno je omeniti, da `git diff` sam po sebi ne prikazuje vseh sprememb, narejenih od vašega zadnjega pošiljanja - samo spremembe, ki še vedno niso dane v vmesno fazo. To je lahko zmedeno, ker če ste dali v vmesno fazo vse vaše spremembe, `git diff` ne bo dal nobenega izpisa.

Za drug primer, če date datoteko `CONTRIBUTING.md` v vmesno fazo in jo nato uredite, lahko uporabite `git diff`, da vidite spremembe v datoteki, ki je dana v vmesno fazo in spremembe, ki še niso dane v vmesno fazo. Če naše okolje izgleda takole:

```
$ git add CONTRIBUTING.md
$ echo 'test line' >> CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

Sedaj lahko uporabite `git diff`, da vidite, kaj še vedno ni dano v vmesno fazo

```
$ git diff
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 643e24f..87f08c8 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -119,3 +119,4 @@ at the
  ## Starter Projects

  See our [projects
list](https://github.com/libgit2/libgit2/blob/development/PROJECTS.md).
+# test line
```

in `git diff --cached`, da vidite, kaj ste dali v vmesno fazo do sedaj (`--staged` in `--cached` sta sinonima):

```
$ git diff --cached
diff --git a/CONTRIBUTING.md b/CONTRIBUTING.md
index 8ebb991..643e24f 100644
--- a/CONTRIBUTING.md
+++ b/CONTRIBUTING.md
@@ -65,7 +65,8 @@ branch directly, things can get messy.
Please include a nice description of your changes when you submit your PR;
if we have to read the whole diff to figure out why you're contributing
in the first place, you're less likely to get feedback and have your change
-merged in.
+merged in. Also, split your changes into comprehensive chunks if your patch is
+longer than a dozen lines.
```

If you are starting to work on a particular area, feel free to submit a PR that highlights your work in progress (and note in the PR title that it's

Git Diff in an External Tool

NOTE

Nadaljevali bomo z uporabo ukaza `git diff` na različne načine skozi preostanek knjige. Je še drug način pogledati te spremembe, če imate raje grafično ali zunanji diff pregledovalnik namesto tega. Če poženete `git difftool` namesto `git diff`, lahko pogledate katerekoli od teh sprememb v programu kot je Araxis, emerge, vimdiff in več. Poženite `git difftool --tool-help`, da vidite, kaj je na voljo na vašem sistemu.

Pošiljanje vaših sprememb

Sedaj, ko je vaša vmesna faza nastavljena na način, kot ga želite, lahko pošljete vaše spremembe. Pomnite, da karkoli, kar še ni dano v vmesno fazo - katerekoli datoteke, ki ste jih ustvarili ali spremenili in na njih še niste pognali `git add` odkar ste jih uredili - ne bodo šle v to pošiljanje. Ostale bodo nespremenjene datoteke na vašem disku. V tem primeru, recimo, da zadnjič, ko ste pognali `git status`, ste videli, da je vse dano v vmesno fazo, torej ste pripravljeni, da pošljete vaše spremembe. Najenostavnejši način za pošiljanje je vpis `git commit`:

```
$ git commit
```

To zažene vaš urejevalnik po izbiri. (To je nastavljeno v vaši spremenljivki okolja `$EDITOR` lupine - običajno vim ali emacs, vendar lahko nastavite s čimerkoli želite z uporabo `git config --global core.editor` ukazom kot ste videli v [Pričetek](#)).

Urejevalnik prikaže sledeči tekst (ta primer je Vim zaslon):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   new file:   README
#   modified:   CONTRIBUTING.md
#
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Vidite lahko, da privzeto sporočilo pošiljanja vsebuje zadnji izpis ukaza `git status`, ki je zakomentiran in ima eno prazno vrstico na vrhu. Te komentarje lahko odstranite in vpišete vaše sporočilo pošiljanja, ali jih pustite tam, da vam pomagajo se spomniti, kaj pošiljate. (Za še bolj eksplicitni opomnik, kaj ste spremenili, lahko podate opcijo `-v` k `git commit`. To tudi doda razliko vaše spremembe v urejevalnik, da lahko točno vidite, katere spremembe pošiljate.) Ko zapustite urejevalnik, Git ustvari vaše pošiljanje s sporočilom pošiljanja (s komentarji in odstranjeno razliko).

Alternativno lahko vpišete vaše sporočilo pošiljanja v vrsticah z ukazom `commit`, ki ga določite po zastavicah `-m`, takole:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 README
```

Sedaj ste ustvarili vaše prvo pošiljanje! Lahko vidite, da vam je pošiljanje dalo izpis o samem sebi: v katero vejo ste poslali (`master`), katera je SHA-1 preverjena vsota, ki jo pošiljanje ima (`463dc4f`), koliko datotek je bilo spremenjenih in statistiko o dodanih in odstranjenih vrsticah v pošiljanju:

Zapomnite si, da pošiljanje snema posnetke, ki ste jih nastavili v vaši vmesni fazi. Karkoli, kar niste dali v vmesno fazo, še vedno čaka spremenjeno; lahko naredite drugo pošiljanje, da to dodate v vašo zgodovino. Vsakič, ko izvedete pošiljanje, snemate posnetek vašega projekta, ki ga lahko povrnete ali primerjate kasneje.

Preskočitev vmesne faze

Čeprav je posebej uporabna za izdelovanje pošiljanj točno tako, kakor jih želite, je vmesna faza včasih bolj kompleksna, kot jo potrebujete v vašem poteku dela. Če želite preskočiti vmesno fazo, Git ponuja enostavno bližnjico. Dodajanje opcije `-a` ukazu `git commit` naredi, da Git avtomatično da vsako datoteko, ki je že sledena preden naredi pošiljanje in vam omogoči preskočiti del `git add`:

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

       modified:   CONTRIBUTING.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 file changed, 5 insertions(+), 0 deletions(-)
```

Bodite pozorni, kako vam ni treba pognati `git add` na datoteki “CONTRIBUTING.md” v tem primeru pred vašim pošiljanjem.

Odstranjevanje datotek

Da odstranite datoteko iz Git-a, jo morate odstraniti iz vaših sledenih datotek (bolj točno, odstraniti iz vaše vmesne faze) in nato poslati. Ukaz `git rm` naredi to in tudi odstrani datoteko iz vašega delovnega direktorija, da je ne vidite kot nesledeno datoteko v naslednjem času.

Če enostavno odstranite datoteko iz vašega delovnega direktorija se prikaže pod “Changed but not updated” (to je *unstaged*) področju vašega izpisa `git status`:

```
$ rm PROJECTS.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

       deleted:    PROJECTS.md

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

Nato, če poženete `git rm` da odstranjevanje datoteke v vmesno fazo:


```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

       deleted:    PROJECTS.md
```

Naslednjič, ko pošiljate, bo datoteka odstranjena in ne bo več sledena. Če ste spremenili datoteko in jo že dodali v indeks, morate prisiliti odstranjevanje z opcijo `-f`. To je varnostna lastnost, da prepreči po nesreči odstranjevanje podatkov, ki še niso bili posneti v posnetku in ne morejo biti povrnjeni iz Git-a.

Druga uporabna stvar, ki jo morda želite narediti je slediti datoteki v vašem delovnem drevesu, vendar odstraniti iz vaše vmesne faze. Z drugimi besedami, morda želite slediti datoteki na vašem trdem disku vendar vam je ni treba slediti. To je posebej uporabno, če pozabite dodati nekaj v vašo datoteko `.gitignore` in jo po nesreči date v vmesno fazo, kot je velika datoteka dnevnika ali skupek prevedenih datotek `.a`. Da to naredite, uproabite opcijo `--cached`:

```
$ git rm --cached README
```

Lahko podate datoteke, direktorije in vzorce datotek-glob k ukazu `git rm`. To pomeni, da lahko naredite stvari kot je

```
$ git rm log/\*.log
```

Bodite pozorni na obratno poševnico (`\`) na začetku `*`. To je potrebno, ker Git dela svoje lastno razširjanje imen datotek k dodatku vašega razširjanja imen datotek lupine. Ta ukaz odstrani vse datoteke, ki imajo razširitev `.log` v direktoriju `log/`. Ali pa lahko naredite nekaj takega:

```
$ git rm \*~
```

Ta ukaz odstrani vse datoteke, ki se končajo z `~`.

Premikanje datotek

Z razliko od ostalih sistemov VCS, Git eksplicitno ne sledi premikanju datotek. Če v Git-u preimenujete datoteko, niso nobenih metapodatki shranjeni v Git, da vam pove, da ste preimenovali datoteko. Vendar je Git precej pameten glede ugotavljanja po dejstvu - z detekcijo premikanja datotek se bomo ukvarjali nekoliko kasneje.

Torej je nekoliko nejasno, da Git ima ukaz `mv`. Če želite preimenovati datoteko v Git-

u, lahko poženete nekaj takega

```
$ git mv file_from file_to
```

in deluje odlično. V bistvu, če poženete nekaj takega in pogledate status, boste videli, da Git smatra kot preimenovano datoteko:

```
$ git mv README.md README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README
```

Vendar to je ekvivalentno pogonu nečesa takega:

```
$ mv README.md README
$ git rm README.md
$ git add README
```

Git ugotovi, da gre za preimenovanje implicitno, torej ni pomembno, če preimenujete datoteko na ta način ali z ukazov `mv`. Edina realna razlika je, da `mv` je en ukaz namesto treh - gre za funkcijo udobja. Bolj pomembno, lahko uporabite katerokoli orodje želite za preimenovanje datoteke in naslovite `add/rm` kasneje, preden pošljete.

Pregled zgodovine pošiljanja

Ko ste ustvarili nekaj pošiljanj ali če ste klonirali repozitorij z obstoječo zgodovino pošiljanj, boste verjetno želeli pogledati nazaj, da vidite, kaj se je zgodilo. Najosnovnejše in močno orodje za to je ukaz `git log`.

Te primeri uporabljajo zelo enostaven projekt imenovan "simplegit". Da dobite projekt, poženite

```
git clone https://github.com/schacon/simplegit-progit
```

Ko poženete `git log` v tem projektu, bi morali dobiti izpis, ki izgleda nekako takole:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 10:31:28 2008 -0700

    first commit
```

Privzeto brez argumentov `git log` izpiše pošiljanja, ki so bila narejena v tem repozitoriju v obratnem kronološkem vrstnem redu - to je, najnovejša pošiljanja se prikažejo prva. Kot vidite, ta ukaz izpiše vsako pošiljanje z njegovo SHA-1 preverjeno vsoto, avtorjevim imenom in e-pošto, napisanim datumom in sporočilom pošiljanja.

Veliko število in različne opcije ukaza `git log` so na voljo, da prikažejo točno to, kar iščete. Tukaj bomo prikazali nekaj najbolj popularnih.

Ena najbolj uporabnih opcij je `-p`, ki prikaže razlike predstavljene v vsakem pošiljanju. Lahko uporabite tudi `-2`, ki omeji izpis na samo zadnja dva vnosa:

```

$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,7 @@ require 'rake/gempackagetask'
  spec = Gem::Specification.new do |s|
    s.platform = Gem::Platform::RUBY
    s.name      = "simplegit"
-   s.version  = "0.1.0"
+   s.version  = "0.1.1"
    s.author   = "Scott Chacon"
    s.email    = "schacon@gee-mail.com"
    s.summary  = "A simple gem for using Git in Ruby code."

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
  end

  end

-
- if $0 == __FILE__
-   git = SimpleGit.new
-   puts git.show
- end
\ No newline at end of file

```

Ta opcija prikaže enake informacije vendar z razliko, ki direktno sledi vsakemu vnosu. To je zelo uporabno za pregled kode ali za hitro brskanje, kaj se zgodi med serijo pošiljanj, ki jih je sodelavec dodal. Lahko tudi uporabite serijo opcij povzetkov z `git log`. Na primer, če želite videti nekaj skrajšanih statistik za vsako pošiljanje, lahko uporabite opcijo `--stat`:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
```

```
changed the version number
```

```
Rakefile | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700
```

```
removed unnecessary test
```

```
lib/simplegit.rb | 5 -----
1 file changed, 5 deletions(-)
```

```
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 10:31:28 2008 -0700
```

```
first commit
```

```
README | 6 ++++++
Rakefile | 23 ++++++
lib/simplegit.rb | 25 ++++++
3 files changed, 54 insertions(+)
```

Kot lahko vidite, opcija `--stat` izpiše pod vsakim vnosom pošiljanja seznam spremenjenih datotek, koliko datotek je bilo spremenjenih in koliko vrstic v teh datotekah je bilo dodanih ali odstranjenih. Doda tudi povzetek informacij na konec.

Druga resnično uporabna opcija je `--pretty`. Ta opcija spremeni izpis dnevnika v oblike druge kot privzete. Nekaj vnaprej vgrajenih opcij vam je na voljo za uporabo. Opcija `oneline` izpiše vsako pošiljanje na eno vrstico, ki je uporabna, če iščete veliko pošiljanj. Kot dodatek, opcije `short`, `full` in `fuller` prikažejo izpis v skoraj enaki obliki, vendar z manj ali več informacijami v zaporedju:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Najbolj zanimiva opcija je `format`, ki vam omogoča določiti vašo lastno obliko izpisa dnevnika. To je posebej uporabno, ko generirate izpis za strojno prevajanje - ker določate obliko eksplicitno, veste, da ne bo spremenilo s posoditvami Git-u:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 6 years ago : changed the version number
085bb3b - Scott Chacon, 6 years ago : removed unnecessary test
a11bef0 - Scott Chacon, 6 years ago : first commit
```

Useful options for `git log --pretty=format` izpiše nekaj bolj uporabnih opcij, ki jih oblika vzame.

Table 1. Useful options for `git log --pretty=format`

Option	Description of Output
<code>%H</code>	Commit hash
<code>%h</code>	Abbreviated commit hash
<code>%T</code>	Tree hash
<code>%t</code>	Abbreviated tree hash
<code>%P</code>	Parent hashes
<code>%p</code>	Abbreviated parent hashes
<code>%an</code>	Author name
<code>%ae</code>	Author e-mail
<code>%ad</code>	Author date (format respects the <code>--date=option</code>)
<code>%ar</code>	Author date, relative
<code>%cn</code>	Committer name
<code>%ce</code>	Committer email
<code>%cd</code>	Committer date
<code>%cr</code>	Committer date, relative
<code>%s</code>	Subject

Lahko se sprašujete, kaj je razlika med *author* in *committer*. Avtor (author) je oseb, ki je prvotno napisala delo, napram prispevalec (committer) je oseba, ki je zadnja dodala delo. Torej, če ste poslali popravek projektu in eden izmed članov jedra doda popravek, oba od vas dobita zasluge - vi kot avtor, in član jedra kot pošiljatelj. To distribucijo bomo pokrili nekoliko bolj v [Distribuirani Git](#).

Ena vrstica in oblika opcij so posebej uporabne z drugimi opcijami `log`, klicanimi `--graph`. Ta opcija doda lep manjši ASCII graf, ki prikazuje vašo vejo in zgodovino združevanj:

```

$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local

```

Ta tip izpisa bo postal bolj zanimiv, ko bomo šli skozi razvejanje in združevanje v naslednjem poglavju.

To so samo nekatere enostavne opije oblike izpisa za `git log` - jih je pa še veliko več. [Common options to git log](#) izpisuje opcije, ki smo jih pokrili do sedaj, kot tudi nekatere ostale pogoste opcije oblikovanja, ki so lahko uporabne, skupaj s tem kako spremenijo izpis ukaza dnevnika.

Table 2. Common options to `git log`

Option	Description
<code>-p</code>	Show the patch introduced with each commit.
<code>--stat</code>	Show statistics for files modified in each commit.
<code>--shortstat</code>	Display only the changed/insertions/deletions line from the <code>--stat</code> command.
<code>--name-only</code>	Show the list of files modified after the commit information.
<code>--name-status</code>	Show the list of files affected with added/modified/deleted information as well.
<code>--abbrev-commit</code>	Show only the first few characters of the SHA-1 checksum instead of all 40.
<code>--relative-date</code>	Display the date in a relative format (for example, "2 weeks ago") instead of using the full date format.
<code>--graph</code>	Display an ASCII graph of the branch and merge history beside the log output.
<code>--pretty</code>	Show commits in an alternate format. Options include oneline, short, full, fuller, and format (where you specify your own format).

Omejevanje izpisa dnevnika

Kot dodatek k opcijam oblike izpisa, `git log` vzame število uporabnih opcij omejevanja - to so opcije, ki vam omogočajo prikazati samo podskupek pošiljanj. Videli ste že eno tako opcijo - opcija `-2`, ki prikaže samo zadnji dve pošiljanji. V bistvu lahko naredite `<n>`, kjer je `n` katerokoli celo število za prikaz zadnjih `n` pošiljanj. V realnosti zelo verjetno ne boste uporabljali tega pogostokrat, ker Git privzeto preusmeri vse izpise

skozi paginacijo, tako da lahko vidite samo eno stran izpisa dnevnika istočasno.

Vendar opcije časovnega omejevanja kot sta `--since` in `--until` sta zelo uporabni. Na primer ta ukaz dobi seznam pošiljanj, ki so bila narejena v zadnjih dveh tednih:

```
$ git log --since=2.weeks
```

Ta ukaz deluje z veliko oblikami - lahko določite določen datum kot je "2008-01-15" ali relativni datum kot je "2 year 1 day 3 minutes ago".

Lahko tudi filtrirate seznam pošiljanj, da se ujema z nekaterimi kriteriji. Opcija `--author` vam omogoča filtrirati na določenega avtorja in opcija `--grep` vam omogoča iskati za ključnimi besedami v sporočilih pošiljanj. (Bodite pozorni, da če želite določiti tako opciji `author` in `grep`, morate dodati `--all-match` drugače bo ukaz ujemal pošiljanja s katero koli opcijo.)

Drug resnično uporaben filter je opcija `-S`, ki vzame niz in samo prikaže ukaze, ki predstavljajo spremembe kodi, ki je bila dodana ali odstranjena nizu. Na primer, če ste želeli najti zadnje pošiljanje, ki je dodalo ali odstranilo referenco določeni funkciji, lahko kličete:

```
$ git log -Sfunction_name
```

Zadnja resnično uporabna opcija za podati `git log` kot filter je pot `- path`. Če določite direktorij ali ime datoteke, lahko omejite izpis dnevnika na pošiljanja, ki so predstavila spremembe tem datotekam. To je vedno zadnja opcija in je v splošnem dodana predhodno z dvojnimi pomišljajem (`--`), da loči poti od opcij.

V [Options to limit the output of git log](#) bomo izpisali te in nekaj ostalih pogostih opcij za vašo referenco.

Table 3. Options to limit the output of `git log`

Option	Description
<code>-(n)</code>	Show only the last n commits
<code>--since, --after</code>	Limit the commits to those made after the specified date.
<code>--until, --before</code>	Limit the commits to those made before the specified date.
<code>--author</code>	Only show commits in which the author entry matches the specified string.
<code>--committer</code>	Only show commits in which the committer entry matches the specified string.
<code>--grep</code>	Only show commits with a commit message containing the string
<code>-S</code>	Only show commits adding or removing code matching the string

Na primer, če želite videti, katera pošiljanja, ki so spremenila datoteke testiranja v zgodovini Git izvirne kode so bila poslana od Junio Hamano in niso bila združena v mesecu oktobru 2008, lahko pošete nekaj takega:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
  --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attributes are in use
acd3b9e - Enhance hold_lock_file_for_{update,append}() API
f563754 - demonstrate breakage of detached checkout with symbolic link HEAD
d1a43f2 - reset --hard/read-tree --reset -u: remove unmerged new paths
51a94af - Fix "checkout --track -b newbranch" on detached HEAD
b0ad11e - pull: allow "git pull origin $something:$current_branch" into an unborn
branch
```

Eden skoraj 40,000 pošiljanj v zgodovini Git izvirne kode, ta ukaz prikaže 6, ki se ujemajo tem kriterijem.

Razveljavljanje stvari

V katerikoli fazi boste morda želeli nekaj razveljaviti. Tu bomo pregledali nekaj osnovnih orodij za razveljavljanje sprememb, ki ste jih naredili. Bodite previdni, ker ne morete vedno razveljaviti nekaterih od teh razveljavitev. To je eno izmed področij v Git-u, kjer lahko izgubite nekaj dela, če to naredite nepravilno.

Ena izmed pogostih razveljavitev se zgodi, ko prezgodaj pošljete in možno pozabite dodati nekaj datotek ali naredite zmedo z vašimi sporočili pošiljanja. Če želite ponovno preizkusiti to pošiljanje, lahko pošete ukaz z opcijo **--amend**:

```
$ git commit --amend
```

Ta ukaz vzame vašo področje vmesne faze in ga uporabi za pošiljanje. Če niste naredili sprememb od vašega zadnjega pošiljanja (na primer ste pognali ta ukaz takoj za prejšnjim pošiljanjem), potem bo vaš posnetek izgledal točno enako in vse, kar boste spremenili je vaše sporočilo pošiljanja.

Zažene se isti urejevalnik pošiljanja sporočila, vendar že vsebuje sporočilo vašega prejšnjega pošiljanja. Sporočilo lahko uredite enako kot vedno, vendar prepíše vaše prejšnje pošiljanje.

Kot primer, če pošljete in nato ugotovite, da ste pozabili dati spremembe v vmesno fazo v datoteki, katero želite dodati temu pošiljanju, lahko naredite nekaj takega:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

Končate z enim ukazom - drugo pošiljanje zamenja rezultate prvega.

Povrnitev datoteke iz vmesne faze

Naslednji sekciji demonstrirata, kako prerekati vaše področje vmesne faze in spremembe delovnega direktorija. Lep del je, da ukazi, ki ste jih uporabili za določanje stanja teh dveh področij vas tudi spominjajo, kako razveljaviti spremembe na njih. Na primer, recimo, da ste spremenili dve datoteki in jih želite poslati kot dve ločeni spremembi, vendar po nesreči vpišete `git add *` in date obe v vmesno fazo. Kako lahko povrnete eno izmed dveh iz vmesne faze? Ukaz `git status` vas opomni:

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

    renamed:    README.md -> README
    modified:   CONTRIBUTING.md
```

Točno pod tekstom "Changes to be committed", pove, da uporabite `git reset HEAD <file>...` za povrnitev iz vmesne faze. Torej uporabimo ta nasvet za povrnitev datoteke `CONTRIBUTING.md` iz vmesne faze:

```
$ git reset HEAD CONTRIBUTING.md
Unstaged changes after reset:
M   CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    renamed:    README.md -> README

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

    modified:   CONTRIBUTING.md
```

Ukaz je nekoliko čuden, vendar deluje. Datoteka `CONTRIBUTING.md` je spremenjena vendar ponovno ni v vmesni fazi.

NOTE

Medtem ko `git reset` je lahko nevaren ukaz, če ga pokličete z `--hard` v tem primeru datoteka v vašem delovnem direktoriju ni dotaknjena. Klicanje `git reset` brez opcije ni nevarno - se samo dotakne vašega področja vmesne faze.

Za sedaj ta čarobna molitev je vse, kar potrebujete vedeti o ukazu `git reset`. Šli bomo v veliko večje podrobnosti o tem kaj `reset` naredi in kako ga osvojiti, da dela res zanimive stvari v [Reset Demystified](#).

Povrnitev sprememb spremenjene datoteke

Kaj če ugotovite, da ne želite obdržati sprememb v datoteki `CONTRIBUTING.md`? Kako jo lahko enostavno razveljavite - povrnete nazaj v stanje, kako je izgledala, ko ste zadnjič poslali (ali začetno klonirali ali kakorkoli ste jo dobili v vaš delovni direktorij)? Na srečo vam prav tako `git status` pove, kako to narediti. V izpisu zadnjega primera področje pred vmesno fazo izgleda takole:

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

   modified:   CONTRIBUTING.md
```

Pove vam precej jasno, kako zavreči spremembe, ki ste jih naredili. Naredimo, kar pravi:

```
$ git checkout -- CONTRIBUTING.md
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   renamed:   README.md -> README
```

Vidite lahko, da so bile spremembe povrnjene.

IMPORTANT

Pomembno je razumeti, da je `git checkout -- [file]` nevaren ukaz. Katerekoli spremembe, ki jih naredite tej datoteki izginejo – samo kopirali ste drugo datoteko preko nje. Nikoli ne uporabite tega ukaza, razen če absolutno veste, da ne želite datoteke.

Če radi sledite spremembam, ki ste jih naredili na tej datoteki, vendar jo morate še vedno spraviti iz poti za sedaj, bomo šli skozi skrito shranjevanje in razvejanje v [Veje Git](#) ; to so splošno res boljši načini za to.

Pomnite, da karkoli je *poslano* v Git je lahko skoraj vedno povrnjeno. Celo pošiljanja, ki so bila na vejah, ki so bile izbrisane ali pošiljanja, ki so bila prepisana z opcijo pošiljanja `--amend`, so lahko povrnjena (glejte [Data Recovery](#) za povrnitev podatkov). Vendar karkoli, kar izgubite in ni bilo nikoli poslano, verjetno nikoli ne boste več videli.

Delo z daljavami

Da imate možnost sodelovanja na kateremkoli Git projektu, morate vedeti kako upravljati vaše oddaljene repozitorije. Oddaljeni repozitoriji so verzije vašega projekta, ki je gostovan na internetu ali nekje na omrežju. Imate lahko nekaj njih, vsak od njih v splošnem je ali za vas samo za branje ali za branje/pisanje. Sodelovanje z drugimi vključuje upravljanje teh oddaljenih repozitorijev in potiskanje in poteg podatkov vanje ali iz njih, ko potrebujete deliti delo. Upravljanje oddaljenih repozitorijev vključuje vedeti kako dodati oddaljene repozitorije, odstraniti daljave, ki niso več veljavne, upravljati različne oddaljene veje in jih definirati kot sledene ali ne in več. V tej sekciji bomo pokrili nekaj od teh oddaljenih upravljalnih veščin.

Prikaz vaših daljav

Da vidite, katere oddaljene strežnike ste nastavili, lahko poženete ukaz `git remote`. Izpiše kratka imena vsake daljave, ki ste jo določili. Če ste klonirali vaš repozitorij, bi morali videti vsak t.i. origin (izvor) - to je privzeto ime, ki ga Git da strežniku iz katerega ste klonirali:

```
$ git clone https://github.com/schacon/ticgit
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 268.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

Lahko tudi določite `-v`, ki vam pokaže URL-je, ki jih je Git shranil za kratko ime, ki bo uporabljeno, ko se bo bralo in pisalo na to daljavo:

```
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
```

Če imate več kot eno daljavo, ukaz izpiše vse. Na primer, repozitorij z večimi daljavami za delo z večimi sodelavci, lahko izgleda nekako takole.

```
$ cd grit
$ git remote -v
bakkdoor https://github.com/bakkdoor/grit (fetch)
bakkdoor https://github.com/bakkdoor/grit (push)
cho45 https://github.com/cho45/grit (fetch)
cho45 https://github.com/cho45/grit (push)
defunkt https://github.com/defunkt/grit (fetch)
defunkt https://github.com/defunkt/grit (push)
koke git://github.com/koke/grit.git (fetch)
koke git://github.com/koke/grit.git (push)
origin git@github.com:mojombo/grit.git (fetch)
origin git@github.com:mojombo/grit.git (push)
```

To pomeni, da lahko potegnemo sodelavce iz kateregakoli izmed teh uporabnikov precej enostavno. Morda moramo dodatno imeti pravice za potiskanje v enega ali več od le-teh, vendar tega ne moremo tu povedati.

Bodite pozorni, da te daljave uporabljajo številne protokole; več o njih bomo pokrili v [Pridobiti Git na strežnik](#).

Dodajanje oddaljenih repozitorijev

Omenili in podali smo nekaj demonstracij dodajanja oddaljenih repozitorijev v prejšnjih sekcijah, vendar tu je tudi, kako to narediti eksplicitno. Da dodamo nov oddaljeni Git repozitorij kot kratko ime, se lahko nanj sklicujete enostavno, poženete `git remote add [shortname] [url]`:

```
$ git remote
origin
$ git remote add pb https://github.com/paulboone/ticgit
$ git remote -v
origin https://github.com/schacon/ticgit (fetch)
origin https://github.com/schacon/ticgit (push)
pb https://github.com/paulboone/ticgit (fetch)
pb https://github.com/paulboone/ticgit (push)
```

Sedaj lahko uporabite niz `pb` v ukazni vrstici namesto celotnega URL-ja. Na primer, če želite ujeti vse informacije, ki jih ima Paul, vendar jih vi še nimate v vašem repozitoriju, lahko poženete `git fetch pb`:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
* [new branch]      master      -> pb/master
* [new branch]      ticgit      -> pb/ticgit
```

Paulova master veja je sedaj dostopna lokalno kot **pb/master** - lahko jo združite v eno izmed svojih vej, ali pa lahko izpišete lokalno vejo na tisti točki, če jo želite preiskati. (Šli bomo skozi, kaj so veje in kako jih uporabljati v več podrobnostih v [Veje Git](#).)

Pridobivanje in poteg iz vaših daljav

Kot ste ravnokar videli, da dobite podatke iz vaših oddaljenih projektov, lahko poženete:

```
$ git fetch [remote-name]
```

Ukaz gre v oddaljeni projekt in potegne vse podatke iz tega oddaljenega projekta, ki jih še nimate. Ko to naredite bi morali imeti reference na vse veje iz te daljave, ki jih lahko združite ali raziščete kadarkoli.

Če klonirate repozitorij, ukaz avtomatsko doda ta oddaljeni repozitorij pod ime "origin". Torej **git fetch origin** ujame katerokoli delo, ki je bilo poslano na ta strežnik odkar ste klonirali (ali vsaj ujeli iz njega). Pomembno je opaziti, da ukaz **git fetch** potegne podatke v vaš lokalni repozitorij - avtomatsko ga ne združi s katerimkoli delom ali spremeni, na čemer trenutno delate. Združiti jih morate ročno v vaše delo, ko ste pripravljeni.

Če imate nastavljeno vejo, da sledi oddaljeni veji (glejte naslednjo sekcijo in [Veje Git](#) za več informacij), lahko uporabite ukaz **git pull**, da avtomatsko ujame in nato združi oddaljeno vejo v vašo trenutno vejo. To je lahko enostavnejši ali bolj udoben potek dela za vas; in privzeto ukaz **git clone** avtomatsko nastavi vašo lokalno master vejo, da sledi oddaljeni master veji (ali kakorkoli se privzeta veja imenuje) na strežniku iz katerega ste klonirali. Pogon **git pull** v splošnem ujame podatke iz strežnika, iz katerega ste prvotno klonirali in jih skuša avtomatsko združiti v kodo na kateri trenutno delate.

Potiskanje v vaše daljave

Ko imate vaš projekt na točki, ki jo želite deliti, morate potistniti smeri toka Ukaz za to je enostaven: **git push [remote-name] [branch-name]**. Če želite potisniti vašo master vejo v vaš **origin** strežnik (ponovno, kloniranje v splošnem nastavi oba od teh imen za vas avtomatsko), nato lahko poženete to s potiskanjem kateregakoli pošiljanja, ki ste ga naredili nazaj na strežnik:

```
$ git push origin master
```

Ta ukaz deluje samo če ste klonirali iz strežnika za katerega imate pravice pisati in če nihče vmes ne potisne. Če vi in še kdo drug klonirate istočasno in on potisne proti toku in nato vi potisnete navzgor, bo vaše potiskanje pravično zavrnjeno. Najprej boste morali potegniti njegovo delo in ga vdelati v vaše preden lahko potiskate. Glejte [Veje Git](#) za več podrobnih informacij, kako lahko potiskate na oddaljene strežnike.

Preverjanje daljave

Če želite videti več informacij o določeni daljavi, lahko uporabite ukaz `git remote show [remote-name]`. Če poženete ta ukaz z določenim kratkim imenom, kot je na primer `origin`, dobite nekaj takega:

```
$ git remote show origin
* remote origin
Fetch URL: https://github.com/schacon/ticgit
Push URL: https://github.com/schacon/ticgit
HEAD branch: master
Remote branches:
  master                tracked
  dev-branch            tracked
Local branch configured for 'git pull':
  master merges with remote master
Local ref configured for 'git push':
  master pushes to master (up to date)
```

Izpiše URL za oddaljeni repozitorij kot tudi informacije sledene veje. Ukaz vam pove v pomoč, da če ste na master veji in poženete `git pull`, bo avtomatsko združil master vejo na daljavi ko ujame vse oddaljene reference. Izpiše tudi oddaljene reference, ki jih je potegnil.

To je enostaven primer, na katerega ste verjetno naleteli. Ko uporabljate Git bolj intenzivno, vendar lahko vidite veliko več informacij iz `git remote show`:

```

$ git remote show origin
* remote origin
URL: https://github.com/my-org/complex-project
Fetch URL: https://github.com/my-org/complex-project
Push URL: https://github.com/my-org/complex-project
HEAD branch: master
Remote branches:
  master                tracked
  dev-branch            tracked
  markdown-strip        tracked
  issue-43              new (next fetch will store in remotes/origin)
  issue-45              new (next fetch will store in remotes/origin)
  refs/remotes/origin/issue-11  stale (use 'git remote prune' to remove)
Local branches configured for 'git pull':
  dev-branch merges with remote dev-branch
  master    merges with remote master
Local refs configured for 'git push':
  dev-branch          pushes to dev-branch          (up to
date)
  markdown-strip     pushes to markdown-strip      (up to
date)
  master              pushes to master          (up to
date)

```

Ta ukaz pokaže, na katero vejo se avtomatsko potiska, ko poženete `git push`, ko ste na določenih vejah. Pokaže vam tudi, katerih oddaljenih vej na strežniku še nimate, katere oddaljene veje imate in so bile odstranjene iz strežnika in več vej, ki so avtomatsko združene, ko poženete `git pull`.

Odstranjevanje in preimenovanje oddaljenih vej

Če želite preimenovati referenco, lahko poženete `git remote rename`, da spremenite kratko ime daljave. Na primer, če želite preimenovati `pb` v `paul`, lahko to naredite z `git remote rename`:

```

$ git remote rename pb paul
$ git remote
origin
paul

```

Vredno je omeniti, da to tudi spremeni imena vaših oddaljenih vej. Kar je bilo včasih sklicevano na `pb/master` je sedaj na `paul/master`.

Če želite odstraniti daljavo zaradi nekega razloga - ste prenesli strežnik ali ne uporabljate več določene preslikave, ali mogoče nekdo, ki je prispeval, sedaj ne prispeva več lahko uporabite `git remote rm`:


```
$ git remote rm paul
$ git remote
origin
```

Označevanje

Kot večina VCS-jev ima Git zmožnost označevanja določenih točk v zgodovini kot pomembne. Običajno ljudje uporabljajo to funkcionalnost za določanje točk izdaj (v1.0 in tako naprej). V tej sekciji se boste naučili kako izpisati oznake (tag-e) na voljo, kako ustvariti nove oznake in kateri različni tipi oznak so na voljo.

Izpisovanje vaših oznak

Izpisovanje oznak, ki so na voljo v Git-u je precej enostavno. Samo vtipkate `git tag`:

```
$ git tag
v0.1
v1.3
```

Ta ukaz izpiše oznake v abecednem vrstnem redu; vrstni red, v katerem se pojavijo, nima neke prave pomembnosti.

Lahko tudi iščete oznake z določenim vzorcem. Git izvorni repozitorij na primer vsebuje več kot 500 oznak. Če vas zanima pogledati samo serijo 1.8.5, lahko poženetete to:

```
$ git tag -l 'v1.8.5*'
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Ustvarjanje oznak

Git uporablja dva glavna tipa oznak: enostavnega in anotacije.

Enostavna oznaka je zelo podobna veji, ki se ne spremeni - je samo kazalec na določeno pošiljanje.

Anotirane oznake so po drugi strani shranjene kot polni objekti v podatkovni bazi Git. Imajo preverjene vsote; vsebujejo ime označevalca, e-pošto in datum; imajo sporočilo

oznake; in so lahko podpisane in preverjene z GNU Privacy Guard (GPG). V splošnem je priporočljivo, da ustvarite anotirane oznake, da imate lahko vse te informacije; vendar če želite začasno oznako ali zaradi kakšnega razloga ne želite imeti ostalih informacij, so na voljo tudi lahke oznake.

Anotirane oznake

Ustvarjanje anotirane oznake v Git-u je enostavno. Najenostavnejši način je določiti `-a`, ko pošete ukaz `tag`:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

`-m` določa sporočilo označevanja, ki je bilo shranjeno z oznako. Če ne določite sporočila za anotirano oznako, Git zažene vaš urejevalnik, da ga lahko vpišete vanj.

Vidite lahko podatke oznake skupaj s pošiljanjem, ki je bilo označeno z uporabo ukaza `git show`:

```
$ git show v1.4
tag v1.4
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:19:12 2014 -0700

my version 1.4

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

To pokaže informacije oznake, datum pošiljanja, ko je bilo označeno in sporočilo anotacije pred prikazom informacij pošiljanja.

Lahke oznake

Drug način za oznako pošiljanja je z lahko oznako. To je v osnovi preverjena vsota pošiljanja v datoteki - nobene druge informacije se ne ohrani. Da ustvarite lahko oznako, ne dodajte opcij `-a`, `-s` ali `-m`:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

Tokrat, če poženete `git show` na oznaki, ne vidite dodatnih informacij oznake. Ukaz samo prikazuje pošiljanje:

```
$ git show v1.4-lw
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Označevanje kasneje

Pošiljanja lahko označite tudi po tem, ko se prestavili preko njih. Predpostavimo, da vaša zgodovina pošiljanja izgleda takole:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddffed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Sedaj predpostavimo, da ste pozabili označiti projekt pri v1.2, ki je bil pri “updated rakefile” pošiljanju. Lahko ga dodate za tem dejstvom. Da označite to pošiljanje, določite preverjeno vsoto pošiljanja (ali del nje) na koncu ukaza:

```
$ git tag -a v1.2 9fceb02
```

Vidite lahko, da ste označili pošiljanje:

```

$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...

```

Deljenej oznak

Privzeto ukaz `git push` ne prenese oznak na oddaljene strežnike. Morali boste eksplicitno poriniti oznake na deljeni strežnik za tem, ko ste jih naredili. Ta proces je točno kot deljenje oddaljenih vej - lahko pošete `git push origin [tagname]`.

```

$ git push origin v1.5
Counting objects: 14, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 2.05 KiB | 0 bytes/s, done.
Total 14 (delta 3), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]          v1.5 -> v1.5

```

Če imate veliko oznak, ki jih želite poriniti naenkrat, lahko uporabite tudi opcijo `--tags` ukazu `git push`. To bo preneslo na oddaljeni strežnik vse vaše oznake, ki še niso tam.

```

$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]          v1.4 -> v1.4
 * [new tag]          v1.4-lw -> v1.4-lw

```

Sedaj, ko nekdo drug klonira ali potegne iz vašega repozitorija, bo dobil tudi vse vaše oznake.

Izpisovanje oznak

V resnici ne morete izpisati (check out) oznake v Git-u, saj se ne morejo prenašati naokoli. Če želite dati verzijo vašega repozitorija v vaš delovni direktorij, ki izgleda kot določena oznaka, lahko ustvarite novo vejo na določeni oznaki z `git checkout -b [branchname] [tagname]`:

```
$ git checkout -b version2 v2.0.0
Switched to a new branch 'version2'
```

Seveda, če to naredite in naredite pošiljanje, bo vaša veja `version2` malenkost drugačna kot vaša `v2.0.0` oznaka, saj bo prenešana naprej z novimi spremembami, torej le bodite pazljivi.

Git aliasi

Preden končamo to poglavje o osnovah Git-a, je samo še en manjši napotek, ki lahko naredi vašo Git izkušnjo enostavnejši, lažjo in bolj znano: aliasi. Ne bomo se sklicevali nanje ali sklepali, da ste jih uporabljali kasneje v knjigi, vendar bi verjetno morali vedeti, kako jih uporabljati.

Git avtomatsko ne sklepa na podlagi vašega ukaza, če ga vpišete parcialno. Če ne želite vpisovati celotnega teksta vsakega ukaza Git, lahko enostavno nastavite alias za vsak ukaz z uporabo `git config`. Tu je nekaj primerov, ki jih morda želite nastaviti:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

To pomeni, da na primer namesto, da vpisujete `git commit`, morate samo vpisati `git ci`. Ko nadaljujete z uporabo Git-a, boste verjetno pogosto uporabljali tudi ostale ukaze; ne čakajte z izdelavo novih aliasov.

Ta tehnika je lahko zelo uporabna pri izdelavi ukazov, za katere smatrate, da bi morali obstajati. Na primer, da popravite uporabnost problema, na katerega ste naleteli z dajanjem datoteke izven vmesne faze, lahko dodate vaš lasten alias za dajanje izven vmesne faze Git-u:

```
$ git config --global alias.unstage 'reset HEAD --'
```

To naredi sledeča ukaza ekvivalentna temu:

```
$ git unstage fileA
$ git reset HEAD fileA
```

To izgleda bolj jasno. Je tudi skupno dodati ukaz `last` sledeče:

```
$ git config --global alias.last 'log -1 HEAD'
```

Na ta način, lahko vidite zadnje pošiljanje enostavno:

```
$ git last
commit 66938dae3329c7aebc598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

Signed-off-by: Scott Chacon <schacon@example.com>
```

Kot lahko poveste, Git enostavno zamenja nov ukaz s čimerkoli kar nastavite alias zanj. Vendar morda boste želeli pognati zunanji ukaz namesto podukaza Git. V tem primeru začnete ukaz z znakom `!`. To je uporabno, če pišete vaša lastna orodja, ki delajo z repozitorijem Git. Demonstriramo lahko z dodajanjem aliasa `git visual`, da požene `gitk`:

```
$ git config --global alias.visual '!gitk'
```

Povzetek

Na tej točki, lahko naredite osnovne lokalne operacije Git - ustvarite ali klonirate repozitorij, naredite spremembe, jih date v vmesno fazo ali jih pošljete ter pogledate zgodovino vseh sprememb preko katerih je šel repozitorij skozi. Naslednje bomo pokrili ubijalsko lastnost Git-a: njegov model vej.

Veje Git

Skoraj vsak VCS ima neko obliko podpore vej. Veje pomenijo, da naredite raznolikost iz glavne linije razvoja in nadaljujete delo brez vpletanja s to glavno linijo. V mnogih orodjih VCS, je to nekako drag proces, pogosto zahteva od vas, da izdelate novo kopijo vašega direktorija izvorne kode, ki lahko vzame dalj časa za večje projekte.

Nekateri se sklicujejo na Gitov model razvejanja kot njegovo “ubijalsko lastnost” in zagotovo postavi Git stran od VCS skupnosti. Zakaj je tako posebno? Način Git vej je izredno lahek, operacije vej naredi skoraj takojšnje in preklapanje nazaj in naprej med vejami je v splošnem tudi tako hitro. Z razliko od ostalih VCS-jev, Git spodbuja potek dela, ki pogosto naredi veje in jih združi, celo večkrat na dan. Razumevanje in osvojitve te lastnosti vam da močno in unikatno orodje in lahko v celoti spremeni način, kako razvijate.

Veje na kratko

Za resnično razumevanje, na kakšen način Git dela razvejanje, se moramo vrniti korak nazaj in raziskati, kako Git shranjuje svoje podatke.

Kakor se morda spomnite iz [Pričetek](#), Git ne shranjuje podatkov kot serije skupkov sprememb ali razlik, vendar namesto tega kot serije posnetkov.

Ko naredite pošiljanje, Git shrani objekt pošiljanja, ki vsebuje kazalec k posnetku vsebine, ki ste jo dali v vmesno fazo. Ta objekt tudi vsebuje ime avtorja in e-pošto, sporočilo, ki ste jo vpisali in kazalce k pošiljanju ali pa pošlje to, kar je direktno prišlo pred tem pošiljanjem (svoj starš od staršev): nobenih staršev za začetne pošiljanje, en starš za običajno pošiljanje in več staršev za pošiljanje, ki je rezultat združevanja dveh ali več vej.

Da to vizualiziramo, predpostavimo, da imate direktorij, ki vsebuje tri datoteke in vse date v vmesno fazo in pošljete. Dajanje datotek v vmesno fazo preveri vsote vsake (SHA-1 zgoščevanje, ki smo ga omenili v [Pričetek](#)) shrani to verzijo datoteke v Git repozitorij (Git se sklicuje nanj kot blob) in doda to preverjeno vsoto v vmesno fazo:

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Ko ustvarite pošiljanje z pogonom `git commit`, Git preveri vsote za vsak poddirektorij (v tem primeru samo vrhnji direktorij projekta) in shrani te objekte drevesa v Git repozitorij. Git nato ustvari objekt pošiljanja, ki ima meta podatke in kazalec na vrhnje drevo projekta, da lahko ponovno ustvari posnetek, ko je potrebno.

Vaš repozitorij Git sedaj vsebuje pet objektov: en blob za vsebine vsake od vaših treh datotek, eno drevo, ki izpisuje seznam vsebine direktorija in določa katera imena datotek so shranjena kot blob-i in eno pošiljanje s kazalcem na vrhnje drevo in vse meta podatke pošiljanja.

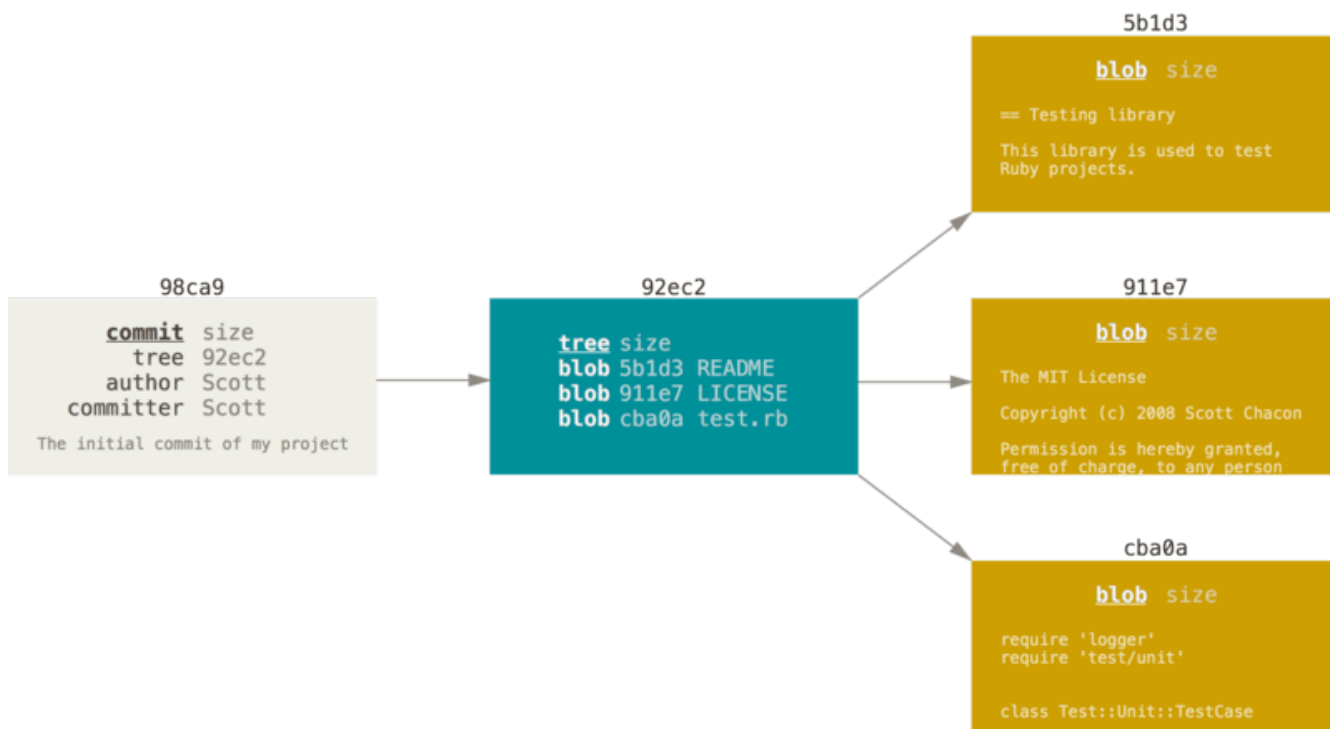


Figure 9. A commit and its tree

Če naredite nekaj sprememb in nato ponovno pošljete, naslednje pošiljanje shrani kazalec k pošiljanju, ki je prišlo takoj pred tem.

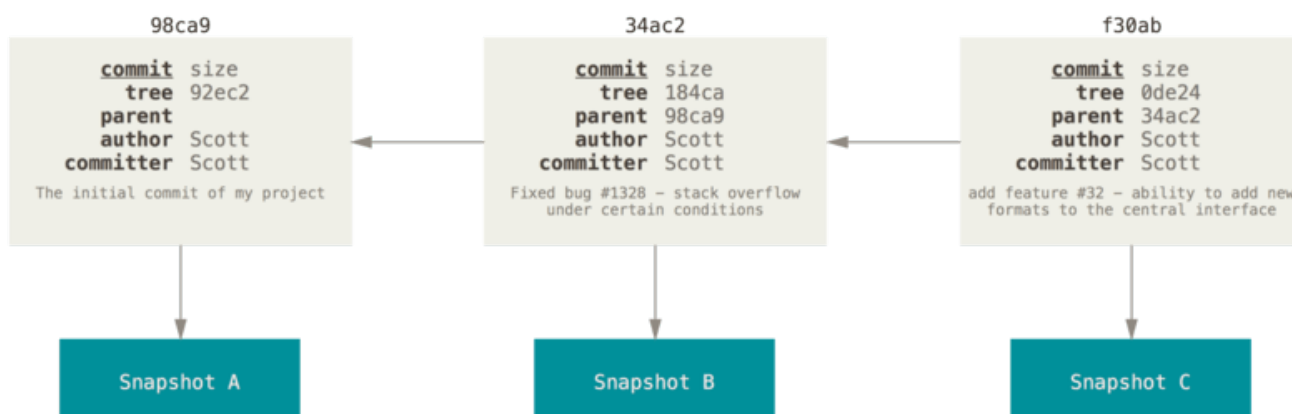


Figure 10. Commits and their parents

Veja v Git-u je enostavno lahek prenosni kazalec k enemu od teh pošiljanj. Privzeto ime veje v Git-u je **master**. Kot ste začeli delati pošiljanja, imate dano master vejo, ki kaže na zadnje pošiljanje, ki ste ga naredili. Vsakič, ko pošljete, ga premakne naprej avtomatsko.

NOTE

Veja “master” v Git-u ni posebna veja. Je točno kot katerakoli druga veja. Edini razlog, da ima skoraj vsake repozitorij eno, je, da jo ukaz **git init** ustvari privzeto in večina ljudi se ne trudi tega spremeniti.

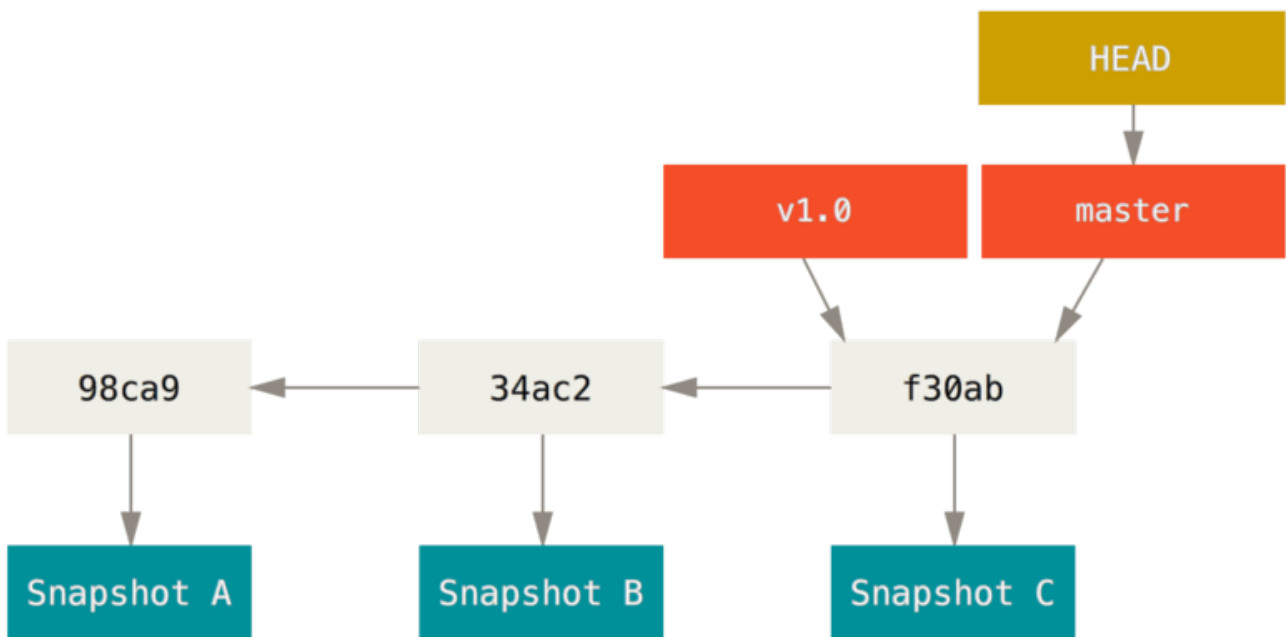


Figure 11. A branch and its commit history

Ustvarjanje nove veje

Kaj se zgodi, če ustvarite novo vejo? No, to naredi nov kazalec za vas, da se premika okoli. Recimo, da ustvarite novo vejo imenovano testing. To naredite z ukazom `git branch`:

```
$ git branch testing
```

To ustvari nov kazalec na isto pošiljanje, na katerem ste trenutno.

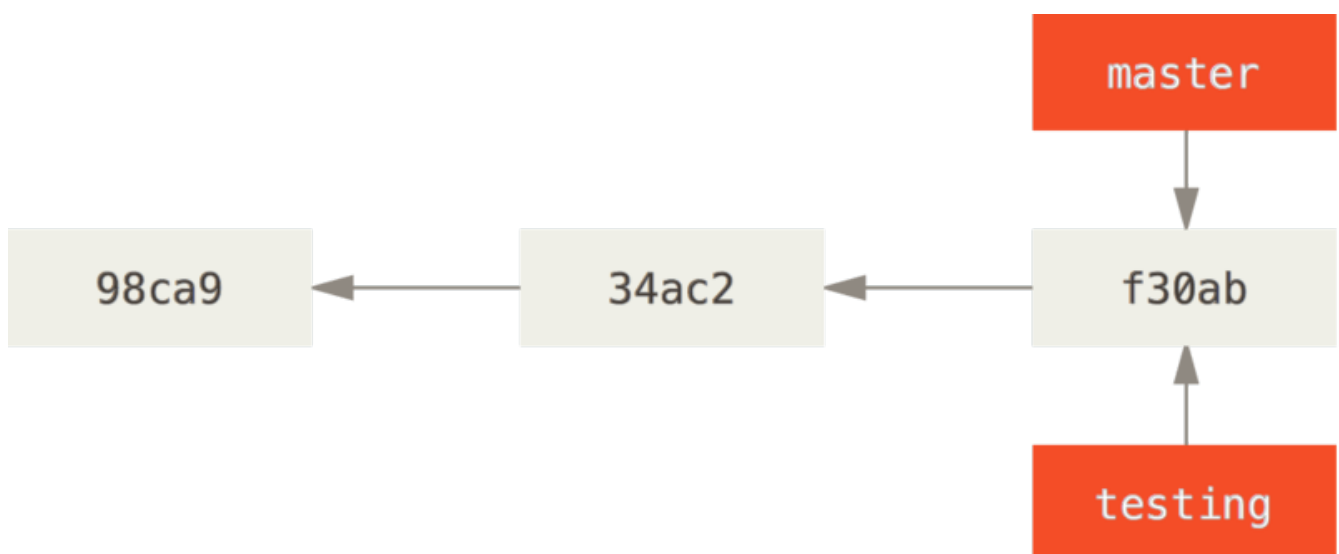


Figure 12. Two branches pointing into the same series of commits

Kako Git ve, na kateri veji ste trenutno? Ima poseben kazalec imenovan `HEAD`. Bodite pozorni, da je to precej različno od koncepta `HEAD` v drugih VCS-jih, ki ste ga morda vajeni, kot sta Subversion ali CVS. V Git-u je to kazalec na lokalno vejo, kjer ste

trenutno. V tem primeru ste še vedno na master. Ukaz `git branch` je samo *ustvaril* novo vejo - ni preklopil na to vejo.

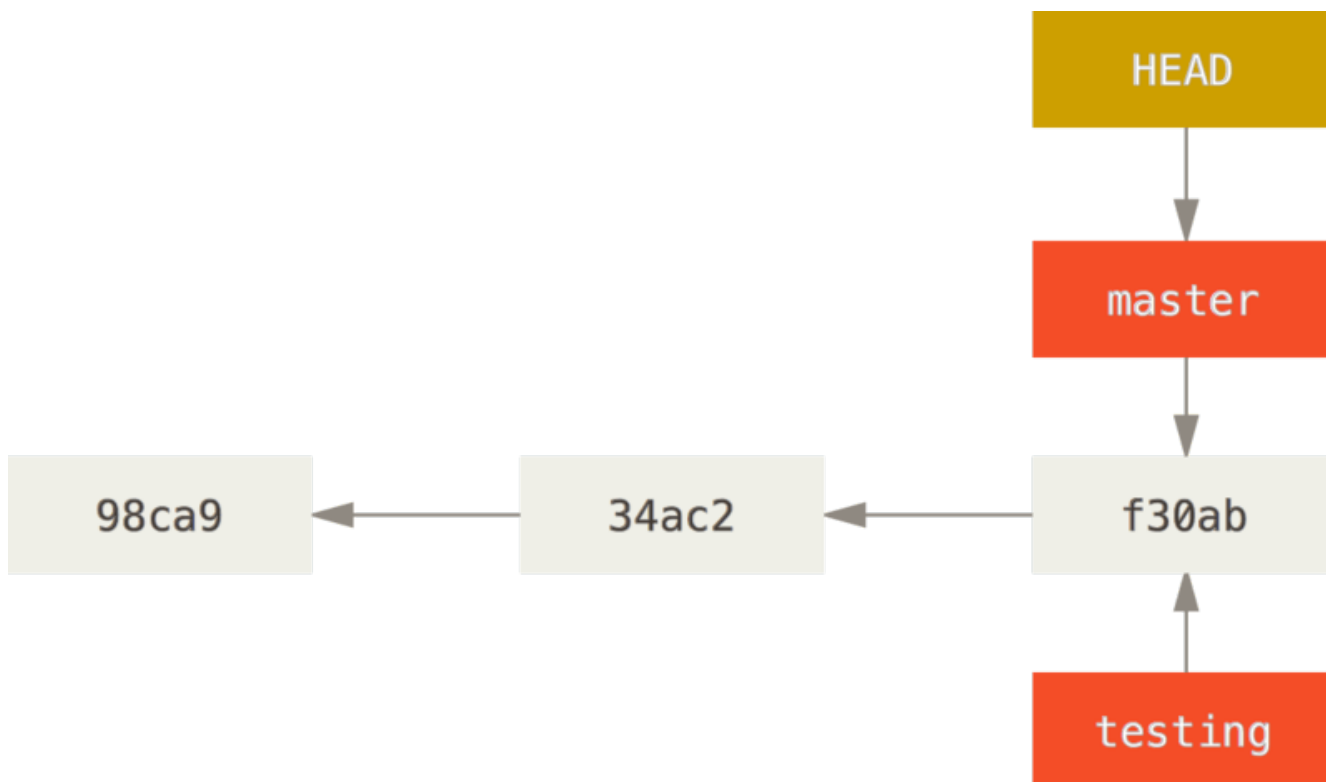


Figure 13. HEAD pointing to a branch

To lahko enostavno pogledate, da poženete ukaz `git log`, ki vam prikaže, kam kazalec veje kaže. Ta opcija se imenuje `--decorate`.

```
$ git log --oneline --decorate
f30ab (HEAD, master, testing) add feature #32 - ability to add new
34ac2 fixed bug #1328 - stack overflow under certain conditions
98ca9 initial commit of my project
```

Vidite lahko “master” in “testing” veji, ki sta ravno tam zraven pošiljanja `f30ab`.

Switching Branches

Da preklopite obstoječo vejo, poženete ukaz `git checkout`. Preklopimo na novo testing vejo:

```
$ git checkout testing
```

To prestavi `HEAD`, da kaže na vejo `testing`.



Figure 14. HEAD points to the current branch

Kaj je pomembnost tega? Torej naredimo drugo pošiljanje:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

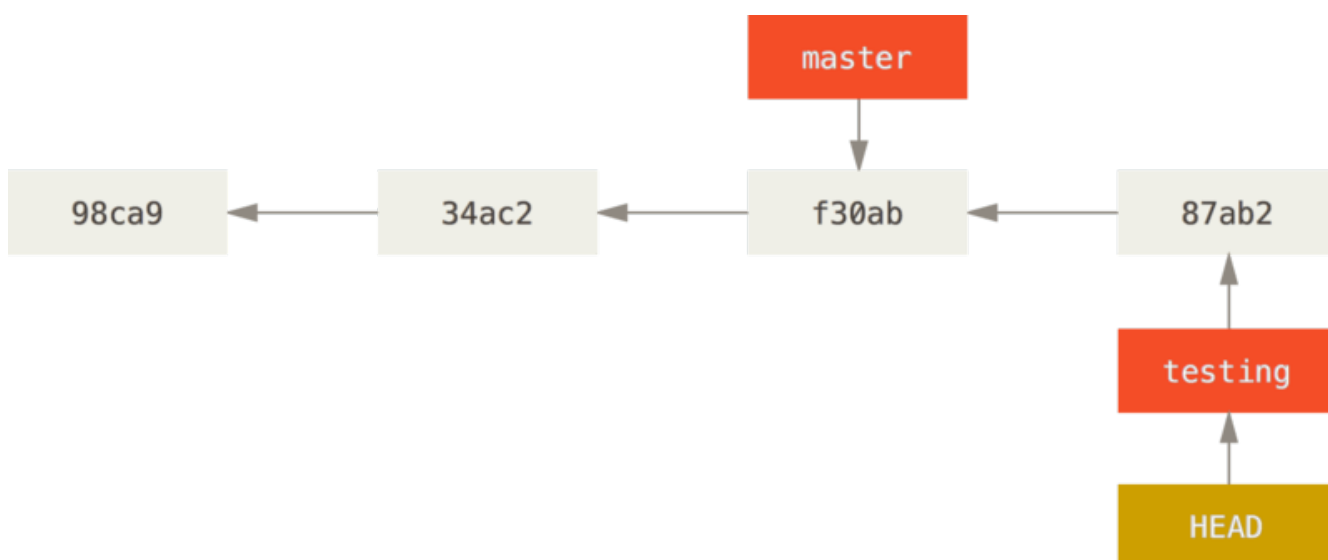


Figure 15. The HEAD branch moves forward when a commit is made

To je zanimivo, ker je sedaj vaša testing veja premaknjena naprej, vendar vaša master veja še vedno kaže na pošiljanje, kjer ste bili, ko ste pognali `git checkout`, da ste preklopili veje. Preklopimo nazaj na master vejo:

```
$ git checkout master
```

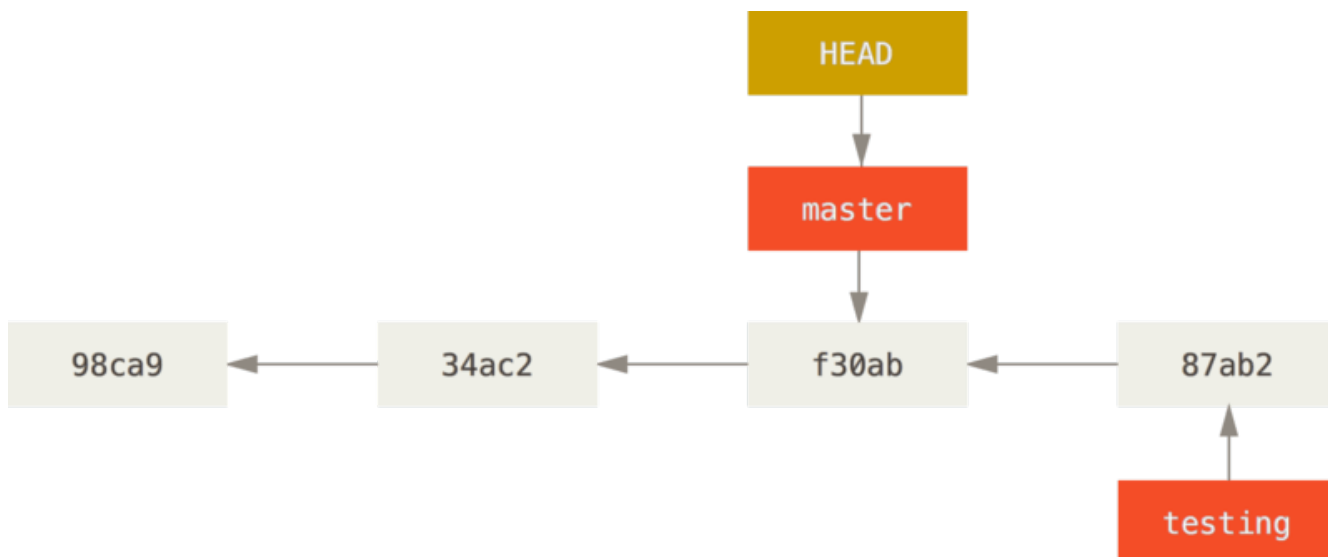


Figure 16. HEAD moves when you checkout

Ta ukaz naredi dve stvari. Premaknil je HEAD kazalec nazaj na točko master veje in povrnil datoteke v vašem delovnem direktoriju nazaj na posnetek, kamor master kaže. To tudi pomeni, da se bodo spremembe, ki jih delate od te točke naprej, razlikovale od starejše verzije projekta. Pomembno je presneti nazaj delo, ki ste ga naredili na vaši testing veji, ta lahko greste v različno smer.

Switching branches changes files in your working directory

NOTE

Pomembno je opozoriti, da ko preklopite veje v Git-u, se datoteke v vašem delovnem direktoriju spremenijo. Če ste preklopili na starejšo vejo, bo vaš delovni direktorij prestavljen nazaj, da bo izgledal tako kot je prejšnjič, ko ste poslali na tisti veji. Če Git to ne more narediti čisto, vam ne bo dovolil preklopiti.

Naredimo nekaj sprememb in ponovno pošljimo:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

Sedaj se je vaša zgodovina projekta spremenila (glejte [Divergent history](#)). Ustvarili in preklopili ste vejo, naredili nekaj dela na njej in nato preklopili nazaj na vašo glavno vejo in naredili drugo delo. Obe od teh sprememb so izolirane v ločenih vejah: lahko preklopite nazaj in naprej med vejama in ju združite skupaj, ko ste pripravljeni. In naredili ste vse to z enostavnimi ukazi `branch`, `checkout` in `commit`.

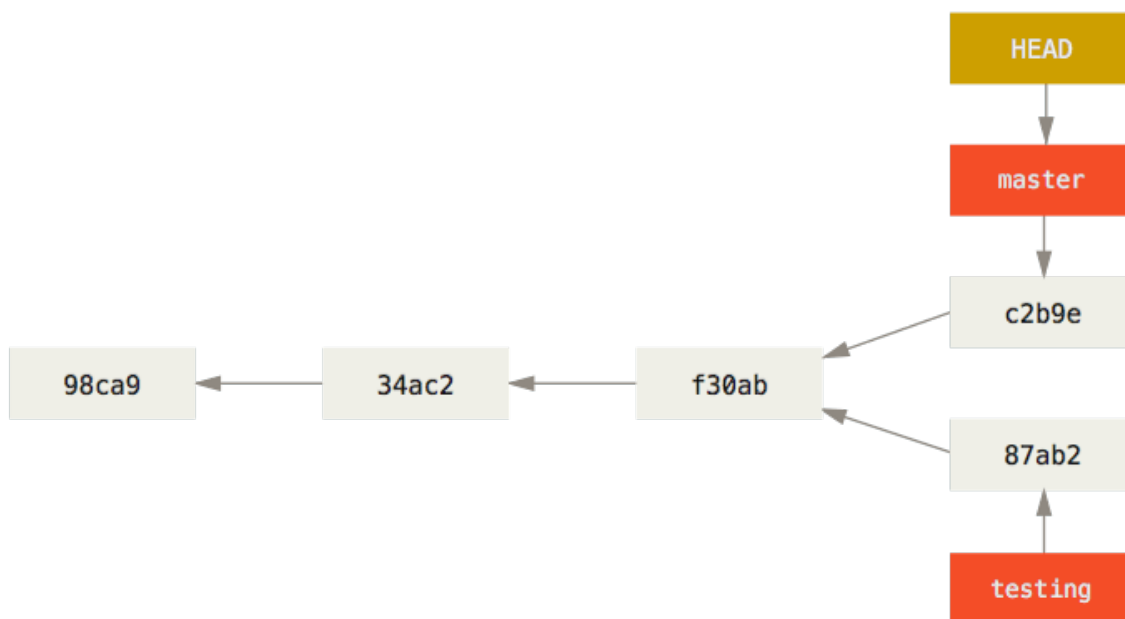


Figure 17. Divergent history

Lahko tudi vidite to enostavno z ukazom `git log`. Če poženete `git log --oneline --decorate --graph --all` bo izpisal zgodovino vaših pošiljanj, prikazal, kje so kazalci vej in kako se je vaša zgodovina spremenila.

```
$ git log --oneline --decorate --graph --all
* c2b9e (HEAD, master) made other changes
| * 87ab2 (testing) made a change
|/
* f30ab add feature #32 - ability to add new formats to the
* 34ac2 fixed bug #1328 - stack overflow under certain conditions
* 98ca9 initial commit of my project
```

Ker je veja v Git-u dejansko enostavna datoteka, ki vsebuje 40 znakovno SHA-1 preverjeno vsoto pošiljanja, kamor kaže, so veje poceni za izdelati in uničiti. Ustvarjanje nove veje je hitro in enostavno kakor napisati 41 bajtov v datoteko (40 znakov in nova vrstica).

To je v ostrem kontrastu z načinom večine vej starejših VCS orodij, ki vključujejo kopiranje vseh datotek projekta v drug direktorij. To lahko vzame nekaj sekund ali celo minut, odvisno od velikosti projekta, kjer je v Git-u proces vedno takojšnji. Tudi ker snemate starše, ko pošljete, iščete ustrezno združevalno osnovo, saj je združevanje narejeno avtomatično za nas in je v splošnem zelo enostavno narediti. Te lastnosti pomagajo spodbujati razvijalcem ustvariti in uporabiti veje pogostokrat.

Poglejmo zakaj bi to morali tako narediti.

Osnove vej in združevanja

Pojdimo skozi enostaven primer vej in združevanja s potekom dela, ki ga morda uporabljate v realnem svetu. Sledili boste tem korakom:

1. Naredite delo na spletni strani.
2. Ustvarite vejo za novo zgodbo na kateri delate.
3. Naredite nekaj dela na tej veji.

V tej fazi boste prejeli klic, da je druga težava kritična in potrebujete sprotni popravek. Naredili boste sledeče:

1. Preklopili na vašo produkcijsko vejo.
2. Ustvarili vejo, da dodate sprotni popravek.
3. Ko je testiran, združite vejo sprotnega popravka in potisnete v produkcijo.
4. Preklopite nazaj na vašo prvotno zgodbo in nadaljujete delo.

Osnove vej

Najprej recimo, da delate na vašem projektu in imate že nekaj pošiljanj.

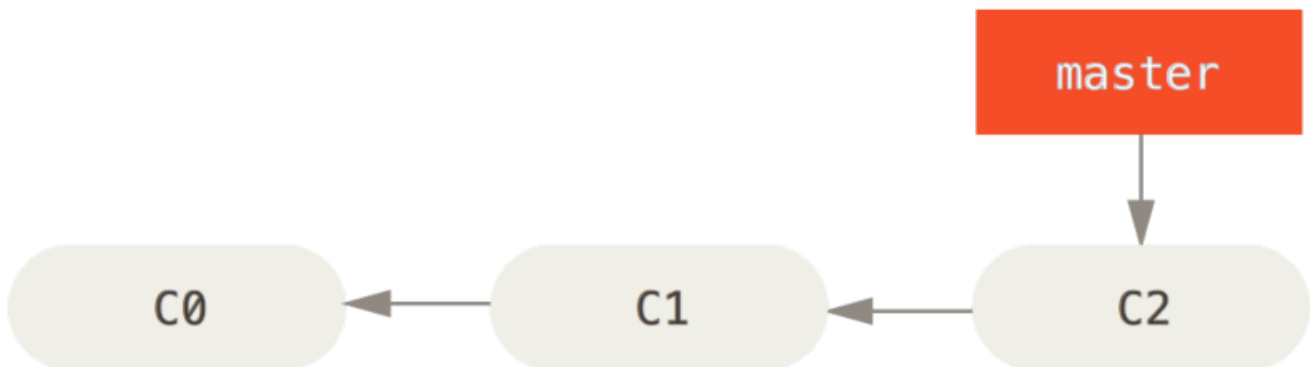


Figure 18. A simple commit history

Odločili ste se, da boste delali na težavi #53 v kateremkoli težavam-sledilnem sistemu, ki ga vaše podjetje uporablja. Da ustvarite vejo in nanjo preklopite istočasno, lahko poženete ukaz `git checkout` s stikalom `-b`:

```
$ git checkout -b iss53  
Switched to a new branch "iss53"
```

To je bližnjica za:

```
$ git branch iss53  
$ git checkout iss53
```

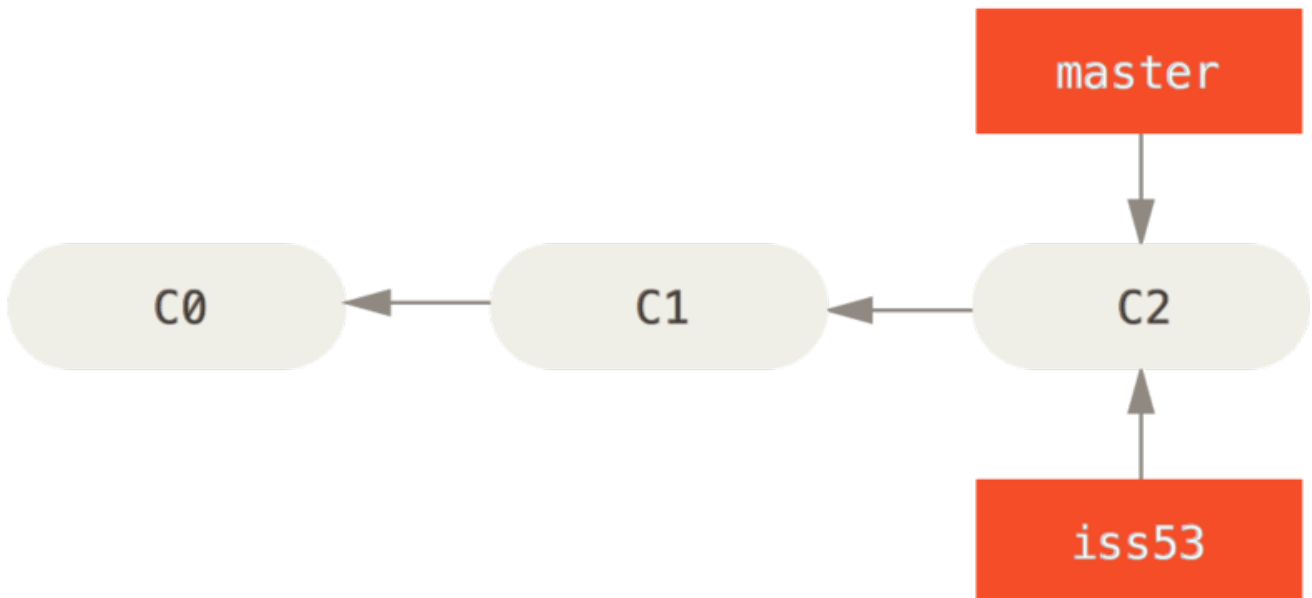


Figure 19. Creating a new branch pointer

Delate na vaši spletni strani in naredite nekaj pošiljanj. Da to naredite premakne vejo `iss53` naprej, ker ste jo izpisali (to pomeni, vaš `HEAD` kaže nanjo):

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

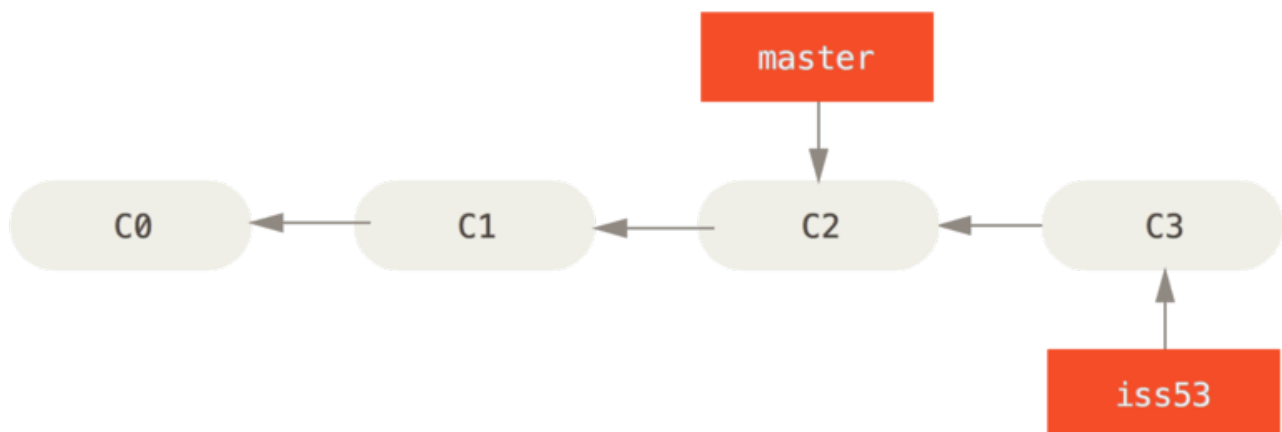


Figure 20. The `iss53` branch has moved forward with your work

Sedaj dobite klic, da je težava s spletno stranjo in jo potrebujete takoj popraviti. Z Gitom vam ni treba nalagati vašega popravka skupaj s spremembami `iss53`, ki ste jih naredili in ni vam treba vložiti veliko napora v povračanje teh sprememb preden lahko delate na uporabi vašega popravka na to, kar je v produkciji. Vse kar morate narediti je preklopiti nazaj na vašo vejo `master`.

Vendar preden to naredite, pomnite, da če ima vaš delovni direktorij ali vmesna faza neposlane spremembe, ki so v konfliktu z vejo, ki jo izpisujete, vam Git ne bo dovolil preklopiti vej. Najbolje je imeti čisto delovno stanje, ko preklapljate veje. Obstajajo načini, da se temu izognete (v glavnem, skrivanje in pošiljanje sprememb), kar

bomo pokrili kasneje v [Stashing and Cleaning](#). Za sedaj predpostavimo, da ste poslali vse vaše spremembe, tako da lahko preklopite nazaj na vašo vejo master:

```
$ git checkout master
Switched to branch 'master'
```

Na tej točki vaš delovni direktorij projekta je točno tak, kakor je bil preden, ste pričeli delati na težavi #53 in sedaj se lahko skoncentrirate na vaš sproti popravek. To je pomembna točka za zapomniti: ko preklapljate veje, Git ponastavi vaš delovni direktorij, da izgleda kot je, ko ste zadnjič poslali na to vejo. Doda, odstrani in spremeni datoteke avtomatično, da zagotovi, da je vaša delovna kopija taka, kot je izgledala veja na vašem zadnjem pošiljanju vanjo.

Naslednje imate za narediti sproti popravek. Ustvarimo vejo sprotnega popravka na kateri delate dokler ni končan:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 1fb7853] fixed the broken email address
1 file changed, 2 insertions(+)
```

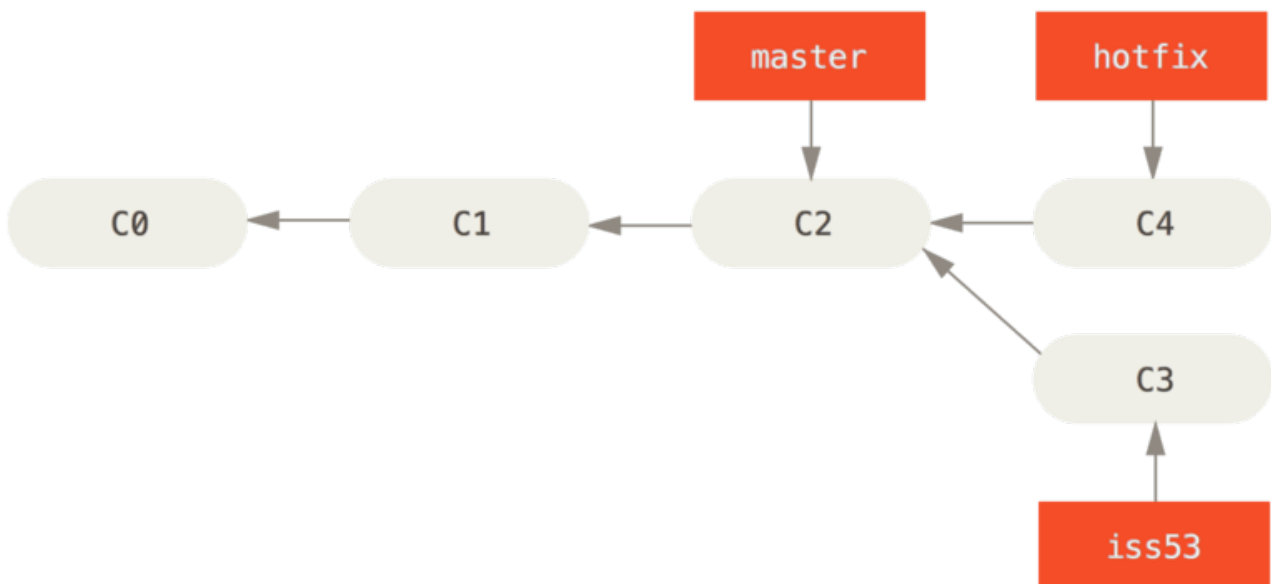


Figure 21. Hotfix branch based on `master`

Lahko poženete vaše teste, zagotovite, da je sproti popravek, kar želite in ga združite nazaj v vašo vejo master, da naložite v produkcijo. To naredite z ukazom `git merge`:


```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Opazili bste frazo “fast-forward” v tem združevanju. Ker je bilo pošiljanje, kamor kaže veja, ki ste jo združili, direktno gorvodno pošiljanja na katerem ste, Git enostavno premakne kazalec naprej. Da dodamo frazo na drug način, kot poskušate združiti eno pošiljanje z drugim, se to lahko doseže s sledenjem zgodovine prvega pošiljanja, Git poenostavi stvari, tako da prestavi kazalec naprej, ker ni divergentnega dela za združiti skupaj - to se imenuje “fast-forward”.

Vaša sprememba je sedaj posnetek pošiljanja, ki kaže na vejo `master` in lahko naložite popravek.

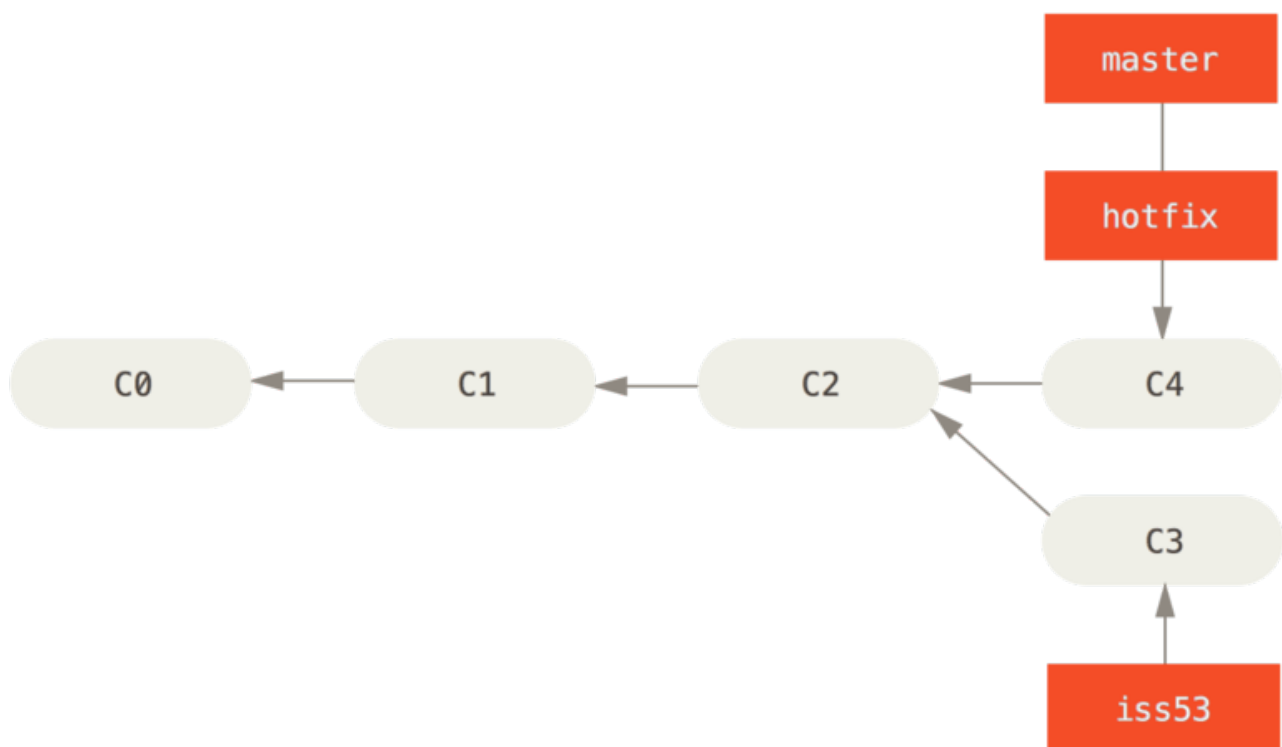


Figure 22. `master` is fast-forwarded to `hotfix`

Ko je vaš super pomemben pomemben popravek naložen, ste pripravljeni preklopiti nazaj k delu, ki ste ga delali preden ste bili zmoteni. Vendar najprej boste izbrisali vejo `hotfix`, ker je ne potrebujete več - veja `master` kaže na enako lokacijo. Lahko jo izbrišete z opcijo `-d` ukazu `git branch`:

```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

Sedaj lahko vaše delo preklopite nazaj na vašo vejo dela v teku na težavi #53 in

nadaljujete delo na njej

```
$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

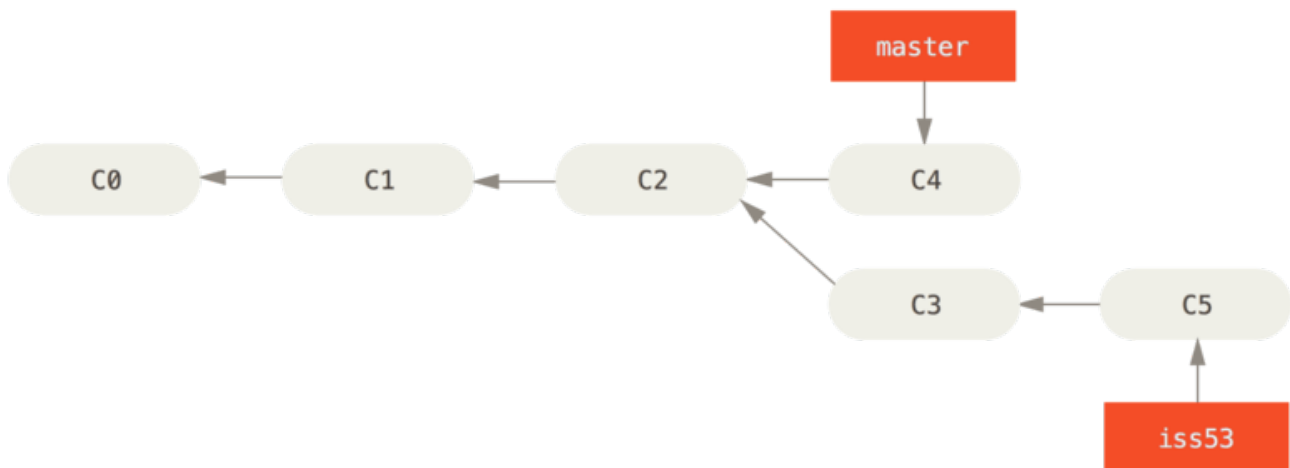


Figure 23. Work continues on `iss53`

Tu je vredno omeniti, da delo, ki ste ga naredili na vaši veji `hotfix` ni vsebovano na datotekah v vaši veji `iss53`. Če jo potrebujete potegniti notri, lahko združite vašo vejo `master` v vašo vejo `iss53` s pogonom `git merge master` ali pa lahko počakate integracijo teh sprememb, dokler se ne odločite potegniti veje `iss53` nazaj v `master` kasneje.

Basic Merging

Predpostavimo, da ste se odločili, da je vaša težava #53 končana in pripravljena, da je združena v vašo vejo `master`. Da to naredite boste združili vašo vejo `iss53` v `master`, tako kot ste prej združili vašo vejo `hotfix`. Vse kar morate narediti je izpisati vejo, ki jo želite združiti in nato pognati ukaz `git merge`:

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html | 1 +
1 file changed, 1 insertion(+)
```

To izgleda nekoliko različno kot združitev `hotfix`, kar ste prej naredili. V tem primeru se je vaša zgodovina razvoja oddaljila od neke starejše točke. Ker pošiljanje na veji, na

kateri se nahajate, ni direktni prednik veje, ki jo združujete, mora Git narediti nekaj dela. V tem primeru Git naredi enostavno tri-načinsko združitev z uporabo dveh posnetkov, ki kažeta na vejo tips in pogostega prednika od dveh.

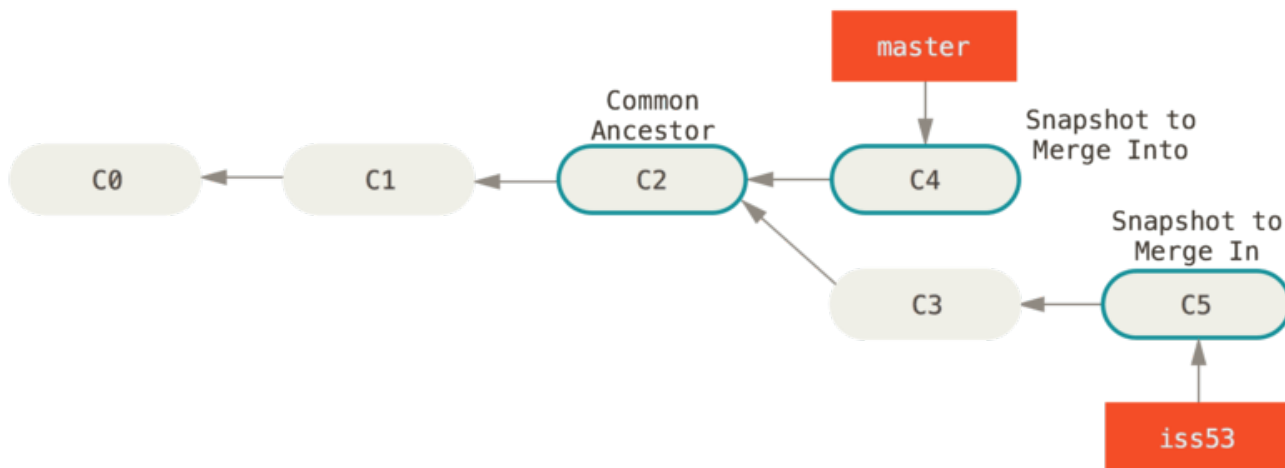


Figure 24. Three snapshots used in a typical merge

Namesto samo premikanja kazalca veje naprej, Git ustvari nov posnetek, ki rezultira iz te tri-načinske združitve in avtomatično ustvari novo pošiljanje, ki kaže nanjo. Na to se sklicuje kot pošiljanje združitve in je posebno v tem, da ima več kot samo enega starša.

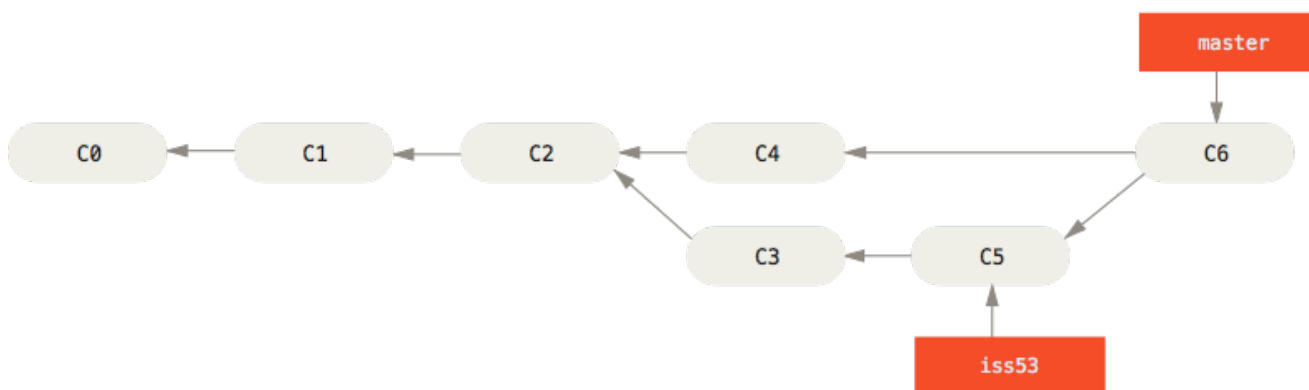


Figure 25. A merge commit

Vredno je pokazati, da Git določa najboljšega pogostega prednika za uporabo za svojo bazo združitev; to je različno od starejših orodij kot je CVS ali Subversion (pred verzijo 1.5), kjer je razvijalec, ki je delal združitev, moral ugotoviti najboljšo osnovo združitve sam. To naredi združevanje veliko bolj enostavno v Git-u kot v teh starejših sistemih.

Sedaj, ko je vaše delo združeno, nimate več potrebe po veji `iss53`. Problem lahko zaprete v vašem sistemu sledenja problemov in izbrišete vejo:

```
$ git branch -d iss53
```

Basic Merge Conflicts

Včasih ta proces ne gre gladko. Če ste spremenili isti del neke datoteke različno v dveh vejah, ki jih združujete skupaj, jih Git ne bo mogel združiti čisto. Če je vaš popravek za težavo #53 spremenil isti del datoteke kot **hotfix**, boste dobili konflikt združevanja, ki izgleda nekako takole:

```
$ git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Ti ni avtomatično ustvaril pošiljanja združevanja. Ustavil je proces, dokler ne rešite konflikta. Če želite videti, katere datoteke niso združene na katerikoli točki po konfliktu združevanja, lahko poženete **git status**:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

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

   both modified:   index.html

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

Karkoli, kar ima konflikte združevanja in ni bilo rešeno je izpisano kot nezdruženo. Git doda standardne označevalce konflikta ločljivosti k datotekam, ki imajo konflikte, tako da jih lahko odprete ročno in rešite te konflikte. Vaša datoteka vsebuje sekcijo, ki izgleda nekako takole:

```
<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>> iss53:index.html
```

To pomeni, da verzija v **HEAD** (vaše veje **master**, ker to je bilo, kar ste imeli izpisano, ko ste pognali vaš ukaz združevanja) je vrhnji del tega bloka (vse nad **=====**), medtem ko

verzija v vaši veji `iss53` izgleda kot vse v spodnjem delu. Da rešite konflikt, morate ali izbrati eno stran ali drugo ali združiti vsebino sami. Na primer ta konflikt lahko rešite z zamenjavo celotnega bloka s tem:

```
<div id="footer">
please contact us at email.support@github.com
</div>
```

Ta ločljivost ima malo vsake sekcije in `<<<<<<`, `=====`, and `>>>>>>` vrstice so bile v celoti odstranjene. Ko ste rešili vsakega od teh sekcij v vsaki konfliktni datoteki, pošnite `git add` na vsaki datoteki, da jo označite kot rešeno. Dajanje datoteke v vmesno fazo jo označi kot rešeno v Git-u.

Če želite uporabiti grafično orodje, da rešite te težave, lahko pošnete `git mergetool`, ki zažene ustrezno vizualno združevalno orodje in vas sprehodi skozi konflikte:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Če želite uporabiti drugo orodje združevanja namesto privzetega (Git izbere `opendiff` v tem primeru, ker je bil ukaz pognan na Mac-u), vidite lahko vsa podprta orodja izpisana na vrhu za “one of the following tools.” Samo vpišite ime orodja, ki bi ga raje uporabljali.

NOTE Če potrebujete bolj napredna orodja za reševanje prepredenih konfliktov združevanja, bomo pokrili več o združevanju v [Advanced Merging](#).

Ko zapustite orodje združevanja, vas Git vpraša, če je bila združitev uspešna. Če poveste skripti, da je bila, da datoteko v vmesno fazo, da jo označi kot rešeno za vas. Ponovno lahko pošnete `git status`, da potrdite, da so bili vsi konflikti rešeni:

```
$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

  modified:   index.html
```

Če ste s tem zadovoljni in potrdite, da je vse, kar je imelo s konflikti bilo dano v vmesno fazo, lahko vpišete `git commit`, da končate pošiljanje združevanja. Sporočilo pošiljanja privzeto izgleda nekako takole:

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#   .git/MERGE_HEAD
# and try again.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# All conflicts fixed but you are still merging.
#
# Changes to be committed:
#   modified:   index.html
#
```

To sporočilo lahko spremenite s podrobnostmi o tem, kako ste rešili združevanje, če razmišljate, da bi bilo v pomoč ostalim, da pogledajo to združevanje v prihodnosti - zakaj ste to naredili, če ni očitno.

Upravljanje vej

Sedaj, ko ste izdelali, združili in izbrisali nekaj vej, pogledajmo nekaj orodij upravljanja vej, ki bodo v pomoč, ko boste začeli uporabljati veje ves čas.

Ukaz `git branch` naredi več kot samo ustvari in izbriše veje. Če ga poženete brez argumentov, dobite enostaven seznam vaših trenutnih vej:

```
$ git branch
  iss53
* master
  testing
```

Bodite pozorni na znak `*`, ki je predpona veje `master`: označuje, da je veja, ki ste jo trenutno odprli (t.j. veja, kamor kaže `HEAD`). To pomeni, da če pošljete na tej točki, bo veja `master` premaknjena naprej z vašim novim delom. Da vidite zadnje pošiljanje na vsaki veji, lahko poženete `git branch -v`:

```
$ git branch -v
  iss53 93b412c fix javascript issue
* master 7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

Uporabni opciji `--merged` in `--no-merged`, lahko filtrirajo ta seznam na veje, ki ste jih že ali še niste združili v vejo, na kateri trenutno delate. Da vidite katere veje so že združene v vejo, na kateri ste, lahko poženete `git branch --merged`:

```
$ git branch --merged
  iss53
* master
```

Ker ste že združili `iss53` prej, jo vidite na vašem seznamu. Veje na tem seznamu brez `*` spredaj so v splošnem v redu za brisanje z `git branch -d`; ste že vkomponirali njihovo delo v drugo vejo, torej ne boste ničesar izgubili.

Da vidite vse veje, ki vsebujejo delo, ki ga še niste združili, lahko poženete `git branch --no-merged`:

```
$ git branch --no-merged
  testing
```

To vam pokaže vašo drugo vejo. Ker vsebuje delo, ki še ni bilo združeno, poskus brisanja z `git branch -d` ne bo uspešen:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Če res želite zbrisati vejo in izgubiti to delo, ga lahko prisilite s `-D`, kot pomagalno sporočilo nakazuje.

Potek dela z vejami

Sedaj, ko imate osnove vej in združevanja, kaj lahko ali bi morali narediti z njimi? V tej sekciji bomo pokrili nekaj skupnih potekov dela, kar ta lahkotna razvejanja omogočajo, da se lahko odločite, če bi ga želeli vkomponirati v vaš lastni razvojni cikel.

Dolgo trajajoče veje

Ker Git uporablja enostavno tri-načinsko združevanje, je združevanje iz ene veje v drugo večkrat skozi daljšo časovno obdobje v splošnem enostavno. To pomeni, da imate nekaj bej, ki so vedno odprte in da jih uporabljate za različne faze vašega razvojnega cikla; lahko združite pogostokrat iz nekaj njih v druge.

Mnogi Git razvijalci imajo potek dela, ki objema ta pristop, kot je imetje samo kode, ki je v celoti stabilna v njihovi `master` veji - verjetno samo koda, ki je bila ali bo izdana. Imajo drugo vzporedno vejo imenovano `develop` ali `next`, iz katere delajo ali uporabljajo za testiranje stabilnosti - ni potrebno vedno stabilna, vendar kadarkoli doseže stabilno stanje, je lahko združena v `master`. Uporabljena je za poteg tematske beje (kratko trajajoče veje, kot vaša prejšnja veja `iss53`), ko so pripravljeni, da zagotovijo, da gre skozi vse teste in ne predstavlja novih hroščev.

V realnosti, govorimo o kazalcih, ki se premikajo gor po črti pošiljanja, ki ga delate. Stabilne veje so nižje na črti v vaši zgodovini pošiljanja in najnovejše veje so na vrhu zgodovine.

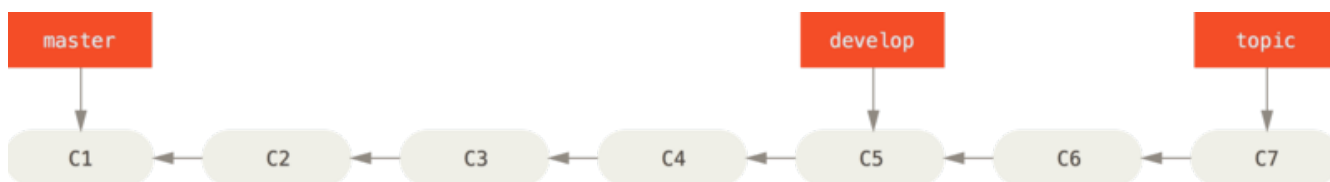


Figure 26. A linear view of progressive-stability branching

V splošnem je enostavnejše razmišljati o njih kot delovnih silosih, kjer so skupki pošiljanja absolvirajo k bolj stabilnim silosom, ko so v celoti testirani.

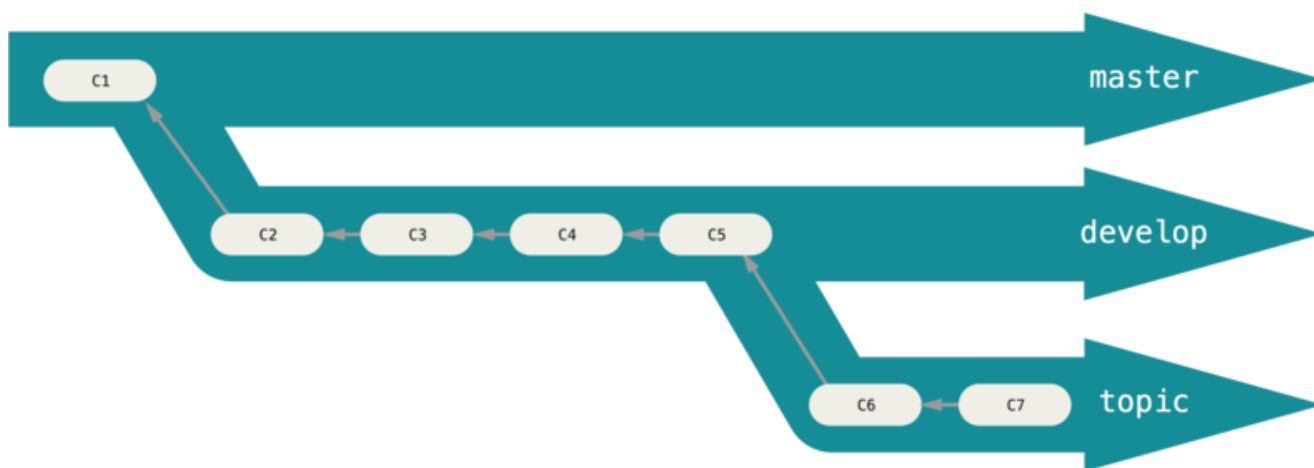


Figure 27. A "silo" view of progressive-stability branching

Lahko nadaljujete s takim delom na mnogih nivojih stabilnosti. Nekateri večji projekti

imajo tudi vejo `proposed` ali `pu` (proposed updates), ki ima integrirano vejo, ki še ni pripravljena, da gre v vejo `next` ali `master`. Ideja je, da vaše veje so na različnih nivojih stabilnosti; ko dosežejo bolj stabilen nivo, so združene v vejo nad njimi. Ponovno, imeti več dolgo trajajočih vej ni potrebno, vendar je pogostokrat v pomoč, posebej kot imate opravka z velikimi in kompleksnimi projekti.

Tematske veje

Tematske veje na drugi strani so uporabne v projektih karkšnihkoli velikosti. Tematska veja je kratko trajajajoča veja, ki ste jo izdelali in uporabljate za eno določeno lastnost ali povezano delo. To je nekaj, kar verjetno še nikoli niste počeli z VCS prej, ker je v splošnem predrago za izdelati in združevati veje. Vendar v Git-u je pogosto izdelati, delati na, združiti in izbrisati veje nekajkrat na dan.

To ste videli v zadnji sekciji pri ustvarjanju vej `iss53` in `hotfix`. Naredili ste nekaj pošiljanj na njih in jih takoj izbrisali po združitvi v glavno vejo. Ta tehnika vam omogoča kontekstni preklon hitro in v celoti - ker je vaše delo razdeljeno v nekaj silosov, kjer vse spremembe v tej veji morajo biti povezane z določeno temo, je enostavnejše videti, kaj se je zgodilo med pregledom kode in podobnim. Lahko sledite spremembam tam nekaj minut, dni ali mesecev in jih združite, ko ste pripravljeni ne glede na vrstni red v katerem ste ustvarili ali delali na njem.

Premislite o primeru, ko delate na nekem delu (na `master`), razvejate za težavo (`iss91`), delate na njej za nekaj časa, razvejate drugo vejo, da poskusite drug način upravljanja z isto stvarjo (`iss91v2`), se vrnete na vašo vejo `master` in delate tam nekaj časa in nato razvejate, da naredite nekaj dela za katerega niste prepričani, da je dobra ideja (veja `dumbidea`). Vaša zgodovina pošiljanja bo izgledala nekako takole:

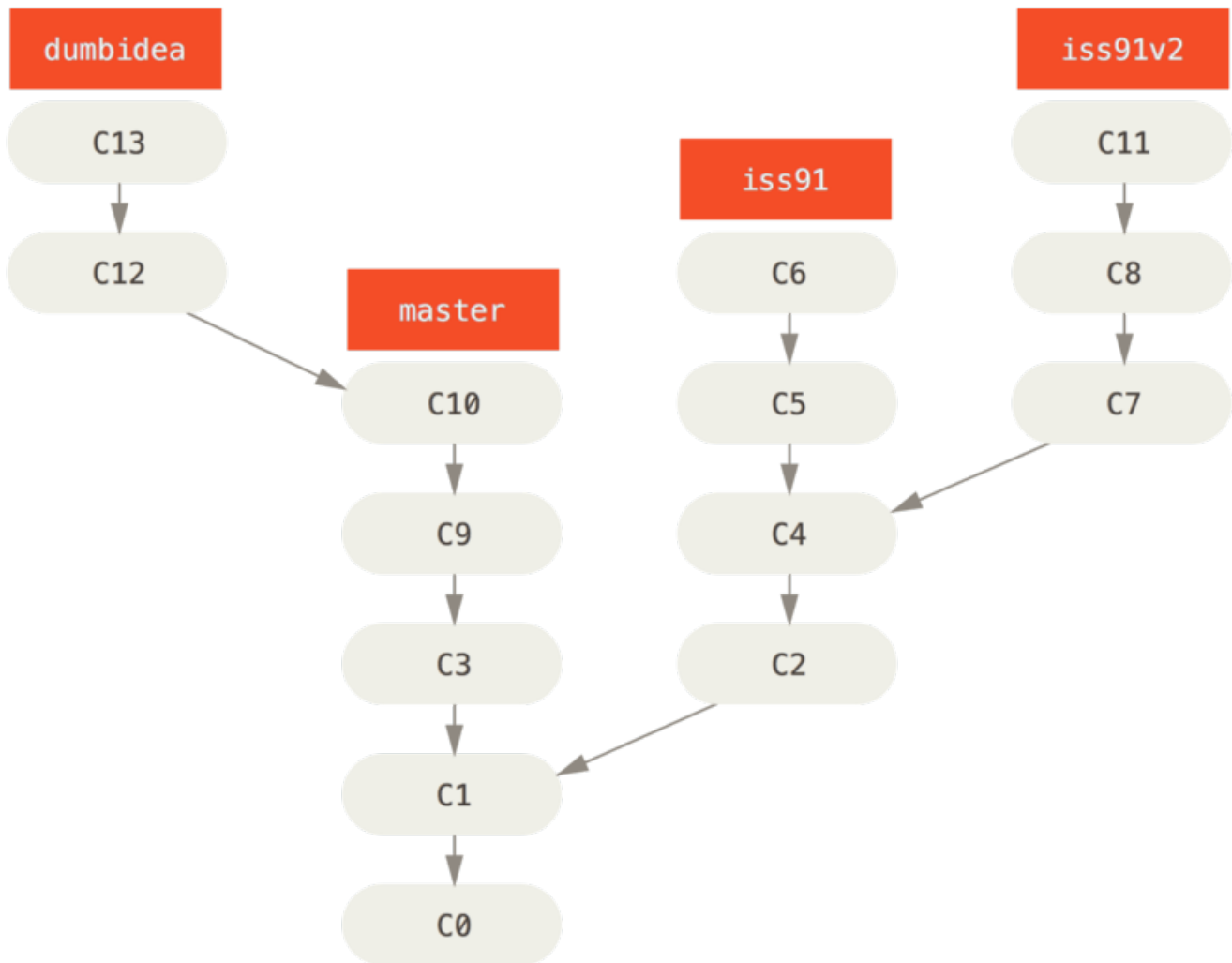


Figure 28. Multiple topic branches

Sedaj recimo, da se odločite, da imate raje drugo rešitev za vašo težavo (*iss91v2*); in ste pokazali vejo *dumbidea* vašim sodelavcem in se izkaže, da je genialna. Lahko vržete stran originalno vejo *iss91* (izgubite nekaj pošiljanja *C5* in *C6*) ter združite v drugi dve. Vaša zgodovina potem izgleda takole:

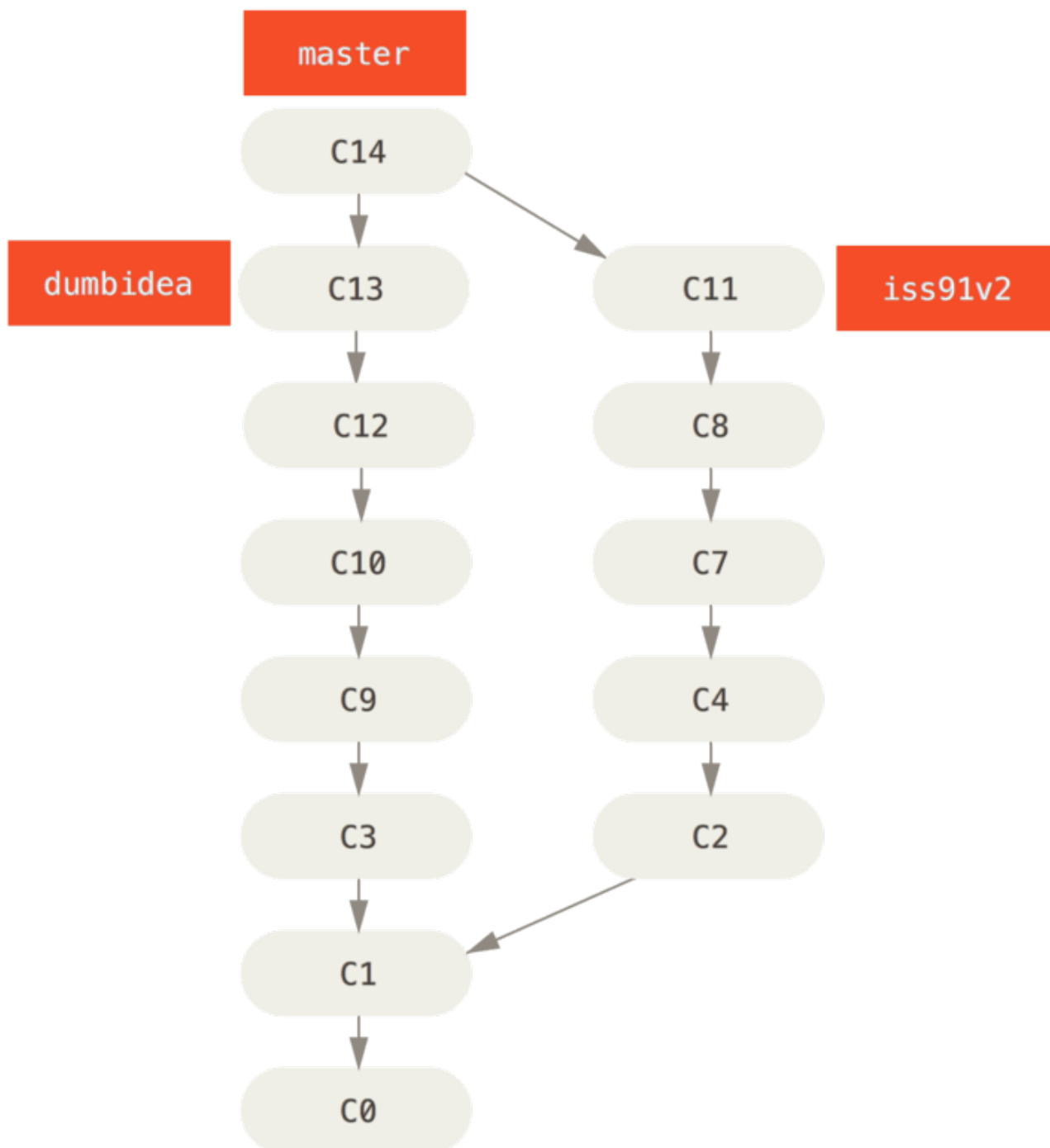


Figure 29. History after merging `dumbidea` and `iss91v2`

Šli bomo v več podrobnosti o različnih možnih potekih dela za vaš Git projekt v [Distribuirani Git](#), tako da preden se odločite, katero shemo razvejanja bo vaš naslednji projekt uporabljal, bodite pozorni, da preberete to poglavje.

Pomembno si je zapomniti, da ko delate vse to, da te veje so v celoti lokalne. Ko razvejujete in združujete, je vse narejeno samo v vašem Git repozitoriju - ne dogaja se nobena komunikacija s strežnikom.

Oddaljene veje

Oddaljene reference so reference (kazalci) v vaših oddaljenih repozitorijih, vključno z

vejami, oznakami itd. Polni seznam oddaljenih referenc lahko dobite eksplicitno z `git ls-remote (remote)`, ali `git remote show (remote)` za oddaljene veje kot tudi več informacij. Vseeno, bolj pogosti način je izkoristiti veje, ki sledijo daljavam.

Veje, ki sledijo daljavam so reference stanja oddaljenih vej. So lokalne reference, ki jih ne morete premakniti; premikajo se avtomatsko za vas kadarkoli naredite kakršnokoli omrežno komunikacijo. Veje, ki sledijo daljavam se obnašajo kot zaznamki, da vas spominjajo, kje so bile veje v vaših oddaljenih repozitorijih, ko ste se zadnjič nanje povezali.

Imajo obliko `(remote)/(branch)`. Na primer, če želite videti, kako je veja `master` na vaši daljavi `origin` izgledala od zadnjič, ko ste komunicirali z njim, bi preverili vejo `origin/master`. Če ste delali na težavi s partnerjem in je potisnil na vejo `iss53`, imate morda vašo lastno lokalno vejo `iss53`; vendar veja na strežniku bi kazala na pošiljanje pri `origin/iss53`.

To je lahko nekoliko zmedeno, torej pogledjmo primer. Recimo, da imate strežnik Git na vašem omrežju pri `git.ourcompany.com`. Če klonirate iz tega, jo ukaz Git `clone` avtomatično poimenuje `origin` za vas, potegne vse njene podatke, ustvari kazalec, kjer je njena `master` veja in jo lokalno poimenuje `origin/master`. Git vam da tudi vašo lokalno vejo `master`, ki se začne na istem mestu kot veja izvora (origin) `master`, torej imate nekaj za delati iz tega.

“origin” is not special

NOTE

Tako kot ime veje “master” nima nobenega posebnega pomena v Git-u, niti nima “origin”. Medtem ko je “master” privzeto ime za začetno vejo, ko poženete `git init`, ki je edini razlog, da je široko uporabljen, “origin” je privzeto ime za daljavo, ko poženete `git clone`. Če namesto tega poženete `git clone -o booyah`, potem boste imeli `booyah/master` kot vašo privzeto oddaljeno vejo.

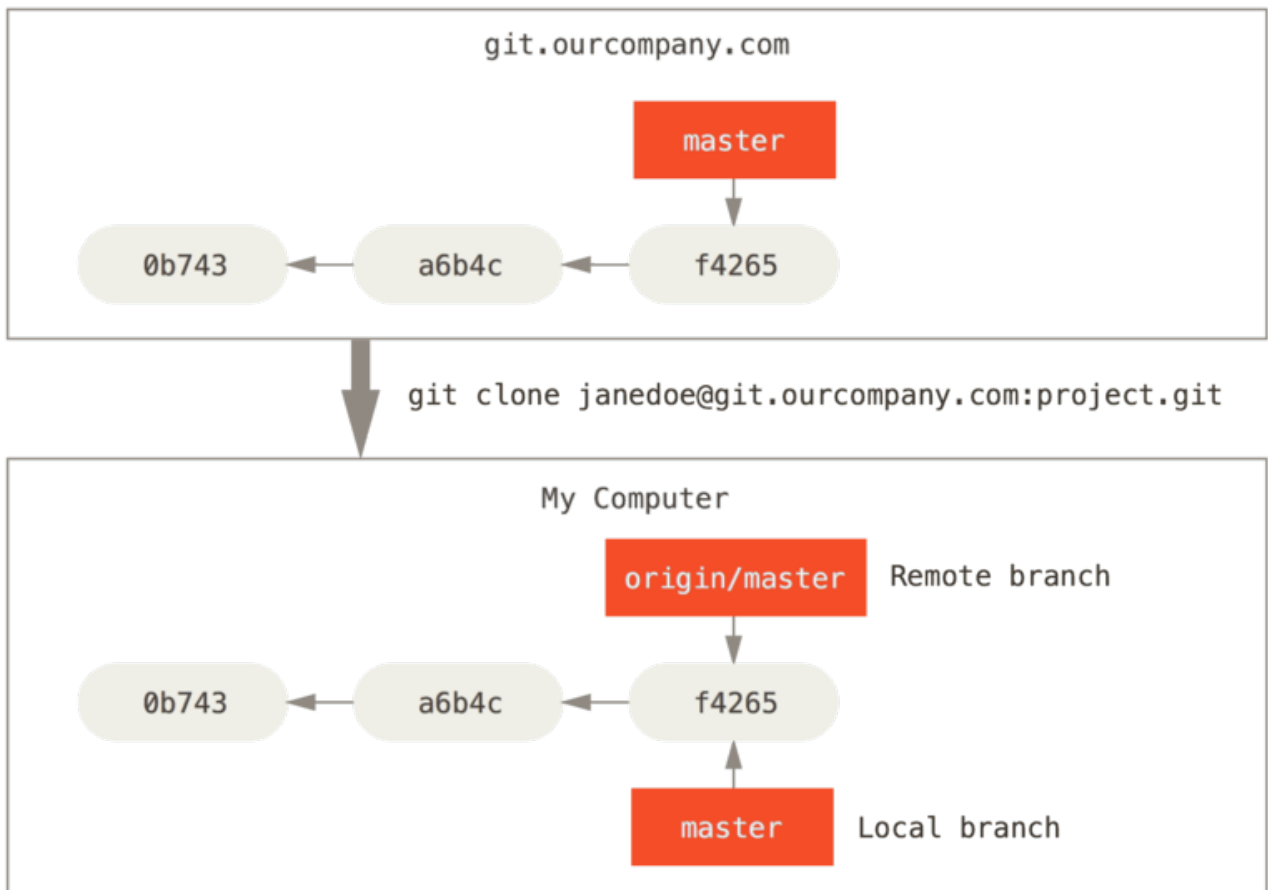


Figure 30. Server and local repositories after cloning

Če naredite nekaj dela na vaši lokalni master veji in vmes nekdo drug potisne na `git.ourcompany.com` in posodobi njegovo `master` vejo, potem se bodo vaše zgodovine premaknile naprej različno. Tudi, dokler ostanete izven kontakta z vašim izvornim strežnikom, se vaš `origin/master` kazalec ne premakne.

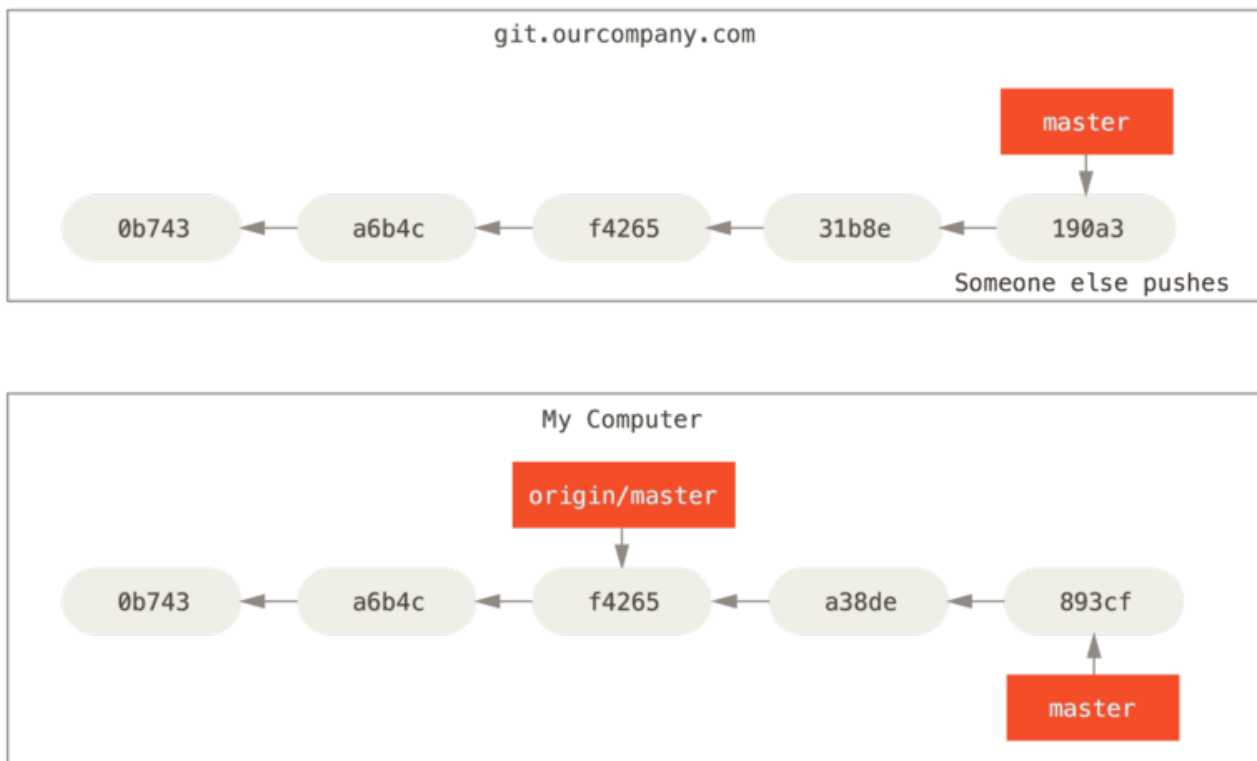


Figure 31. Local and remote work can diverge

Da sinhronizirate vaše delo, poženete ukaz `git fetch origin`. Ta ukaz poišče strežnik, kateri je “origin” (v tem primeru je `git.ourcompany.com`), ujame kakršnekoli podatke iz njega, ki jih še nimate in posodobi vašo lokalno podatkovno bazo, premakne vaš kazalec `origin/master` na njegovo novo, bolj posodobljeno pozicijo.

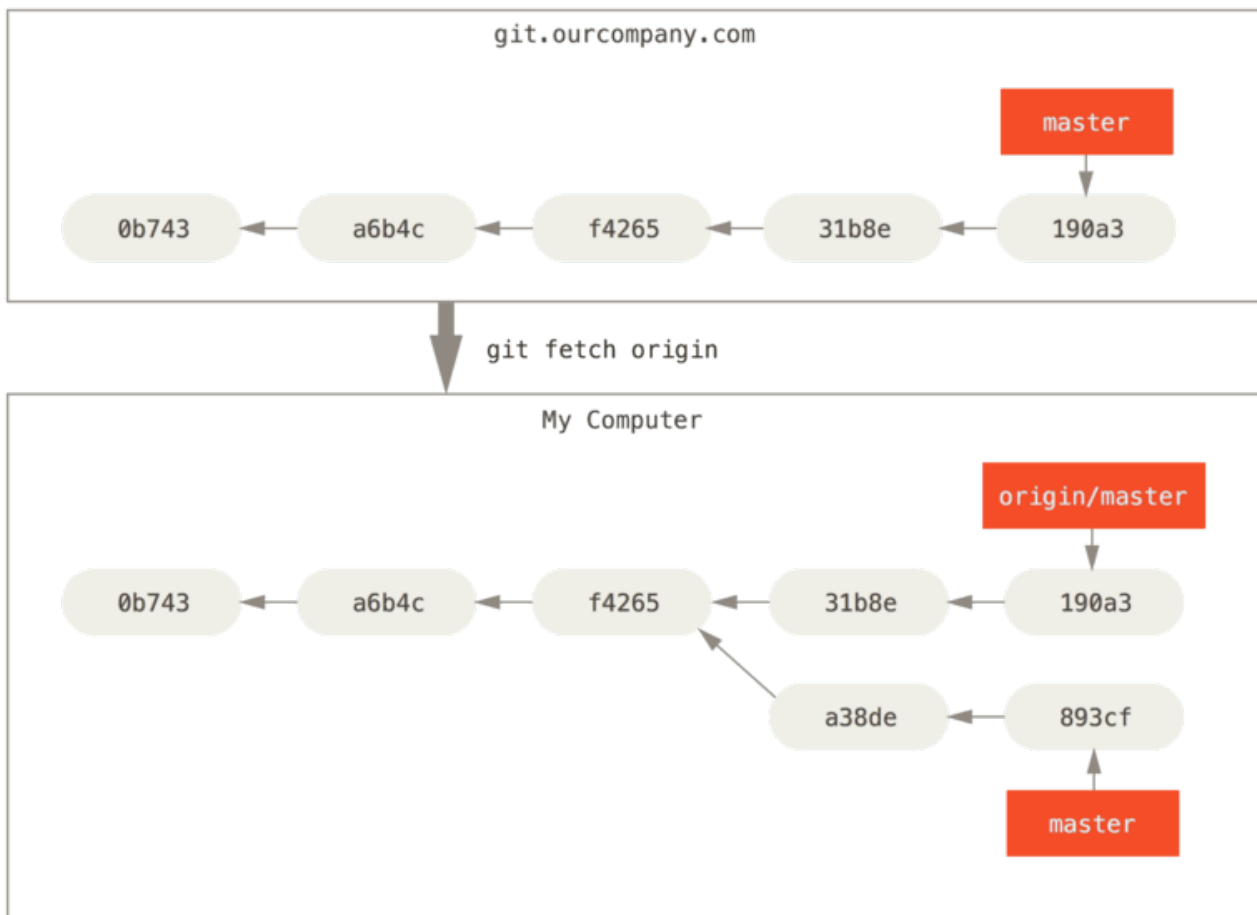


Figure 32. `git fetch` updates your remote references

Za demonstracijo imetja večih oddaljenih strežnikov in kako oddaljene veje za te oddaljene projekte izgledajo, predpostavimo, da imate drug interni strežnik Git, ki je uporabljen samo za razvoj s strani ene vaših šprintnih ekip. Ta strežnik je na `git.team1.ourcompany.com`. Lahko ga dodate kot novo oddaljeno referenco k projektu, kjer trenutno delate s pogonom ukaza `git remote add`, kot smo pokrili v [<Osnove Git](#). Poimenujte to daljavo `teamone`, ki bo vaša kratko ime za ta celotni URL.

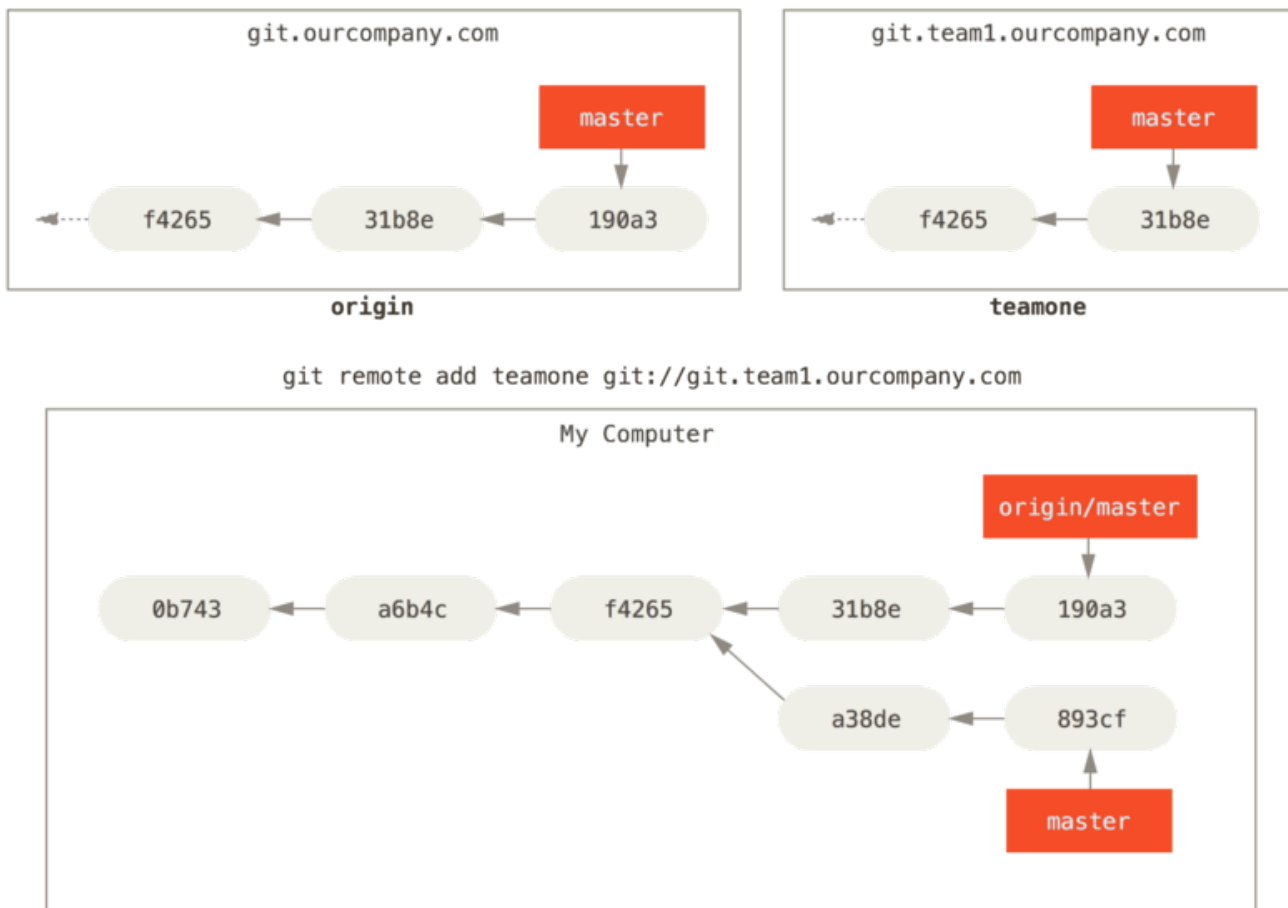


Figure 33. Adding another server as a remote

Sedaj lahko poženete `git fetch teamone`, da ujamete celotni oddaljeni `teamone` strežnik, ki ga še nimate. Ker ima ta strežnik podmnožico podatkov, ki jih ima vaš strežnik `origin` sedaj, Git ne ujame nobenih podatkov vendar skupke veje, ki sledi daljavi imenovana `teamone/master`, ki kaže na pošiljanje, ki ga ima `teamone` na njegovi veji `master`.

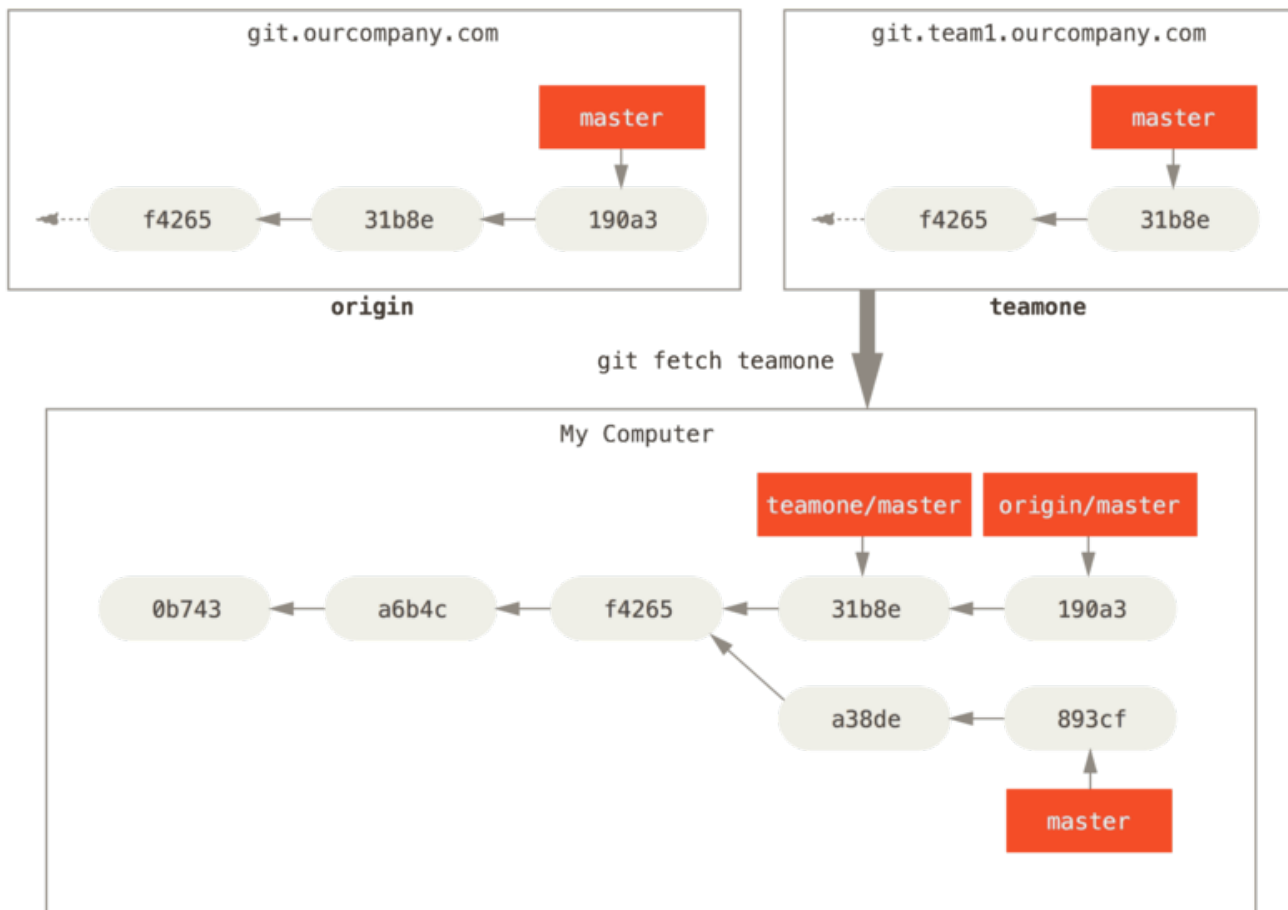


Figure 34. Remote tracking branch for `teamone/master`

Porivanje

Ko želite deliti vejo s svetom, jo morate poriniti na daljavo, za katero imate dostop pisanja. Vaše lokalne veje niso avtomatično sinhronizirane z daljavami, na katere pišete - morate eksplicitno potisniti na vejo, ki jo želite deliti. Na ta način lahko uporabite privatne veje za delo, ki ga ne želite deliti in porinete samo na tematske veje, s katerimi želite sodelovanje.

Če imate vejo imenovano `serverfix`, na kateri želite delati z drugimi, lahko porinete nanjo na enak način, kakor ste porinili na vašo prvo vejo. Poženite `git push (remote) (branch)`:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

To je nekakšna bližnjica. Git avtomatično razširi ime veje `serverfix` na `ref/heads/serverfix:refs/heads/serverfix`, kar pomeni, "Vzamimo mojo `serverfix` lokalno

vejo in porinimo nanjo, da se posodobi oddaljeno `serverfix` branch.” Šli bomo skozi `refs/heads/` del v podrobnosti v [Notranjost Git-a](#), vendar v splošnem lahko izpustite. Lahko tudi naredite `git push origin serverfix:serverfix`, ki naredi isto stvar - pove “Vzamite mojo `serverfix` in jo naredite oddaljeno `serverfix`.” To obliko lahko uporabite za porivanje lokalne veje na oddaljeno vejo, ki je poimenovana drugače. Če ne želite, da se imenuje `serverfix` na daljavi, lahko namesto tega pošete `git push origin serverfix:awesomebranch`, da porinete vašo lokalno vejo `serverfix` na vejo `awesomebranch` na oddaljenem projektu.

Don't type your password every time

Če uporabljate HTTPS URL za porivanje, vas bo strežnik Git vprašal za uporabniško ime in geslo za overitev. Privzeto vas bo vprašal za te informacije na terminalu, tako da strežnik lahko pove, če vam je dovoljeno porivati.

NOTE

Če ne želite vpisovati vsakič, ko porivate, lahko nastavite “credential cache”. Najenostavnejše je samo obdržati v spominu za nekaj minut, kar lahko enostavno nastavite s pogonom `git config --global credential.helper cache`.

Za več informacij o različnih opcij predpomnenja overilnic, ki so na voljo, glejte [Credential Storage](#).

Naslednjič, ko eden izmed vaših sodelavcev ujame iz strežnika, bo dobil referenco, kjer je verzija strežnika `serverfix` pod oddaljeno vejo `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
* [new branch]      serverfix    -> origin/serverfix
```

Pomembno je opomniti, da ko naredite ujetje, ki prinese novo vejo, ki sledi daljavi, vam ni treba avtoatično imeti njihove lokalne, urejevalne kopije. Z drugimi besedami v tem primeru, nimate nove veje `serverfix` - imate samo kazalec `origin/serverfix`, ki ga ne morete spremeniti.

Da združite to delo v vašo trenutno delovno vejo, lahko pošete `git merge origin/serverfix`. Če želite vašo lastno vejo `serverfix`, na kateri lahko delate, lahko naredi osnovo iz vaše veje, ki sledi daljavi:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

To vam da lokalno vejo, na kateri lahko delate in se začne, kjer je `origin/serverfix`.

Sledenje vej

Izpis lokalne veje iz veje, ki sledi daljavi avtomatično ustvari, kar se imenuje “sledena veja” oz. “tracking branch” (ali nekaj kot je “upstream branch”). Sledene veje so lokalne veje, ki imajo direktno relacijo z oddaljeno vejo. Če ste na sledeni veji in vpišete `git pull`, Git avtomatsko ve, iz katerega strežnika ujeti in vejo združiti.

Ko klonirate repozitorij, v splošnem avtomatično ustvari vejo `master`, ki sledi `origin/master`. Vendar lahko nastavite druge sledene veje, če želite - eno, ki sledi vejam na drugih daljavah ali ne sledi veji `master`. Enostaven primer je, kar ste ravnokar videli, pošete `git checkout -b [branch] [remotename]/[branch]`. To je dovolj pogosta operacija, ki jo Git ponuja na kratko `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

Da nastavite lokalno vejo z različnim imenom kot je oddaljena veja, lahko enostavno uporabite prvo verzijo z različnim imenom lokalne veje:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

Sedaj vaša lokalna veja `sf` bo avtomatično potegnila iz `origin/serverfix`.

Če že imate lokalno vejo in želite nastaviti oddaljeno vejo, ki ste jo ravnokar potegnili ali želite spremeniti gorvodno vejo, ki ji slediti, lahko uporabite opcijo `-u` ali `--set-upstream-to` k `git branch`, da jo eksplicitno nastavite kadarkoli.

```
$ git branch -u origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
```

Upstream shorthand

NOTE

Ko imate sledeno vejo nastavljeno, lahko naredite nanjo referenco z `@{upstream}` ali `@{u}` bližnjico. Torej če ste na veji `master` in sledi `origin/master` lahko rečete nekaj kot `git merge @{u}` namesto `git merge origin/master`, če želite.

Če želite videti, katero sledeno vejo imate nastavljeno, lahko uporabite opcijo `-vv` k `git branch`. To bo izpisalo vaše lokalne veje z več informacijami vključno, čemu vsaka veja sledi in če je vaša lokalna veja naprej, nazaj ali oboje.

```
$ git branch -vv
  iss53      7e424c3 [origin/iss53: ahead 2] forgot the brackets
  master     1ae2a45 [origin/master] deploying index fix
* serverfix  f8674d9 [teamone/server-fix-good: ahead 3, behind 1] this should do it
  testing    5ea463a trying something new
```

Torej tu lahko vidite, da naša veja `iss53` sledi `origin/iss533` in je naprej - “ahead” za dva, kar pomeni, da imate dve pošiljanji lokalno, ki nista porinjena na strežnik. Lahko tudi vidimo, da vaša veja `master` sledi `origin/master` in je posodobljena. Naslednje lahko vidimo, da vaša veja `serverfix` sledi `server-fix-good` veji na vašem `teamone` strežniku in je naprej za tri in nazaj za eno, kar pomeni, da je eno pošiljanje na strežniku, ki ga še nismo združili in tri pošiljanja lokalno, ki jih še nismo potisnili. Na koncu lahko vidimo, da naša veja `testing` ne sledi katerikoli oddaljeni veji.

Pomembno je opomniti, da te številke so samo od zadnjič, ko ste ujeli iz vsakega strežnika. Ta ukaz ne doseže strežnikov, pove vam o tem, kaj je ujel iz teh strežnikov lokalno. Če želite totalno posodobitev številčk naprej ali nazaj, boste morali ujeti iz vseh vaših daljav ravno preden to poženetete. To lahko naredite takole `$ git fetch --all; git branch --vv`

Porivanje

Medtem ko bo ukaz `git fetch` ujel vse spremembe na strežniku, ki jih še nimate, ne bo nikakor spremenil vašega delovnega direktorija. Enostavno bo dobil podatke za vas in vam omogočil, da jih združite sami. Vendar pa obstaja ukaz imenovan `git pull`, ki je v bistvu `git fetch`, ki mu takoj sledi `git merge` v večini primerov. Če imate nastavljeno sledeno vejo, kot je demonstrirano v zadnji sekciji, ali eksplicitno nastavljeno ali da imate ustvarjeno za vas z ukazoma `clone` ali `checkout`, bo `git pull` poiskal, kateremu strežniku in veji trenutna veja sledi, ujel iz tistega strežnika in nato poskusil združiti v to oddaljeno vejo.

V splošnem je bolje, da enostavno uporabite ukaza `fetch` in `merge` eksplicitno, saj je čarobnost `git pull` pogostokrat zmedena.

Izbris oddaljenih vej

Predpostavimo, da ste končali z oddaljeno vejo - recimo, da ste vi in vaši sodelavci končali z lastnostjo in jo imate združeno v vašo oddaljeno vejo `master` (ali katerokoli vejo, na kateri je vaša stabilna linija kode). Lahko izbrišete oddaljeno vejo z uporabo opcije `--delete` na `git push`. Če želite izbrisati vašo vejo `serverfix` iz strežnika, poženetite sledeče:

```
$ git push origin --delete serverfix
To https://github.com/schacon/simplegit
- [deleted]          serverfix
```

V osnovi vse kar naredi je, da odstrani kazalec iz strežnika. Git strežnik bo v

splošnem obdržal podatke tam za nekaj časa, dokler se poganja zbirka smeti, torej če je bila po nesreči izbrisana, je pogostokrat povrnitev enostavna.

Ponovno baziranje (rebasing)

V Git-u sta dva glavna načina za integracijo sprememb iz ene veje v drugo: `merge` in `rebase`. V tej sekciji se boste naučili, kaj je rebasing, kako ga narediti, zakaj je precej posebno orodje in v katerih primerih, ga ne boste uporabljali.

Osnovno ponovno baziranje

Če se vrnete na prejšnji primer iz [Basic Merging](#), lahko vidite, da ste se oddaljili od vašega dela in naredili pošiljanja na dveh različnih vejah.

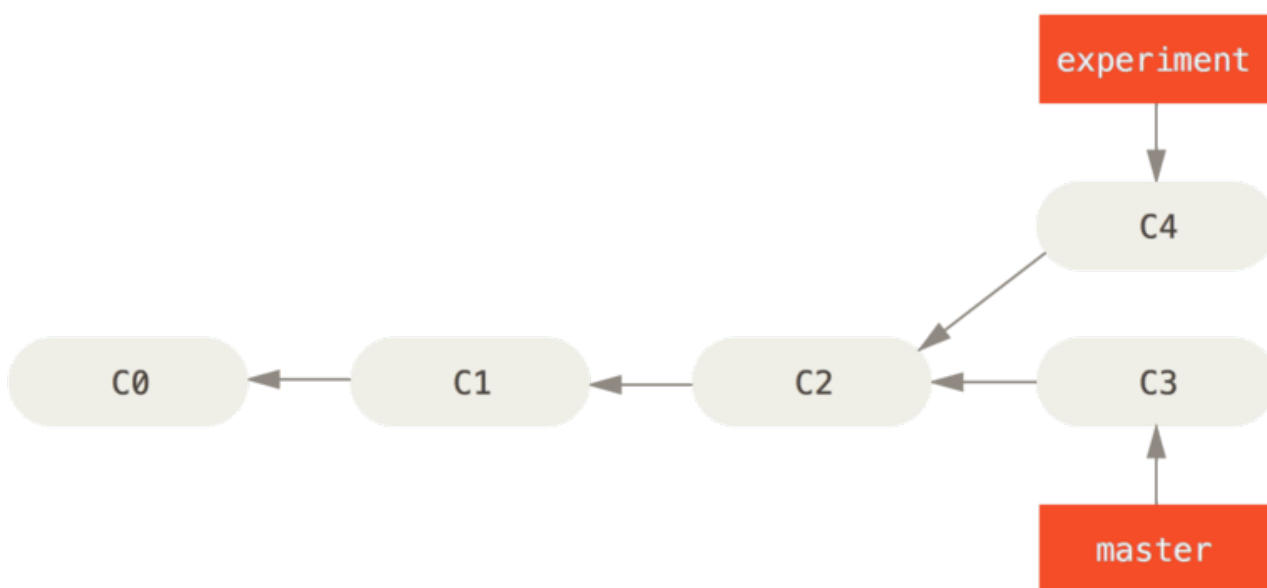


Figure 35. Simple divergent history

Najenostavnejši način za integracijo vej, kot smo to že pokrili je ukaz `merge`. Izvede tri-načinsko združevanje med dvema zadnjima posnetkoma vej (`C3` in `C4`) in najbolj zadnji skupen prednik obeh (`C2`), ustvarjanje novega posnetka (in pošiljanje).

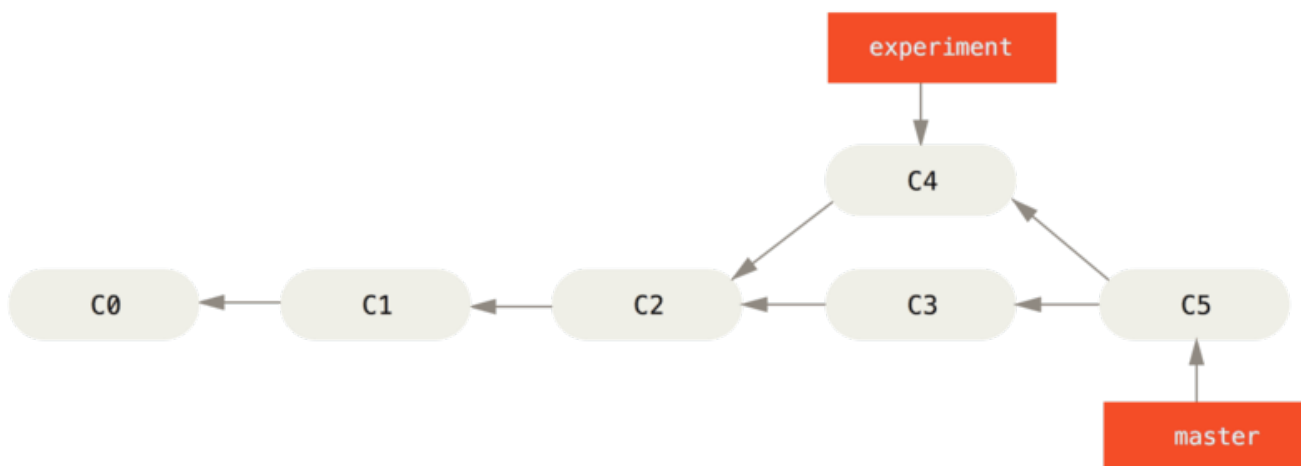


Figure 36. Merging to integrate diverged work history

Vendar obstaja še drug način: lahko vzamete popravek spremembe, ki je bil predstavljen v C4 in ga ponovno uporabite na vrhu C3. V Git-u se to imenuje *rebasing*. Z ukazom *rebase* lahko vzamete vse spremembe, ki so bile poslane na eni veji in jih ponovite na drugi.

V tem primeru bi pognali sledeče:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Deluje, če greste do skupnega prednika obeh vej (ene na kateri ste in druge katero ponovno bazirate), pridobitev diff-a predstavljenega z vsakim pošiljanjem veje na kateri ste, shranjevanje teh diff-ov v začasne datoteke, ponastavljanje trenutne veje na istem pošiljanju kot je veja, na katero bazirate in končno uporaba vsake spremembe v zameno.

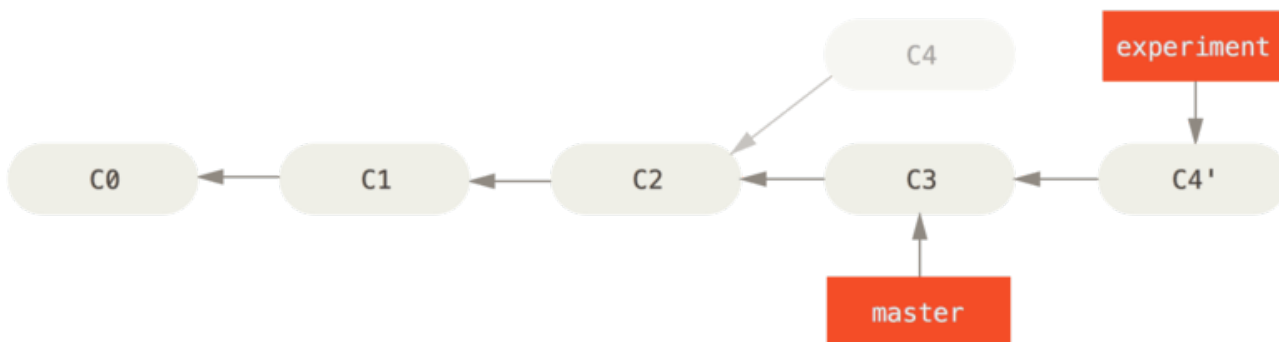


Figure 37. *Rebasing the change introduced in C4 onto C3*

Na tej točki lahko greste nazaj k master veji in naredite združevanje s hitrim pomikanjem naprej (fast-forward).

```
$ git checkout master
$ git merge experiment
```

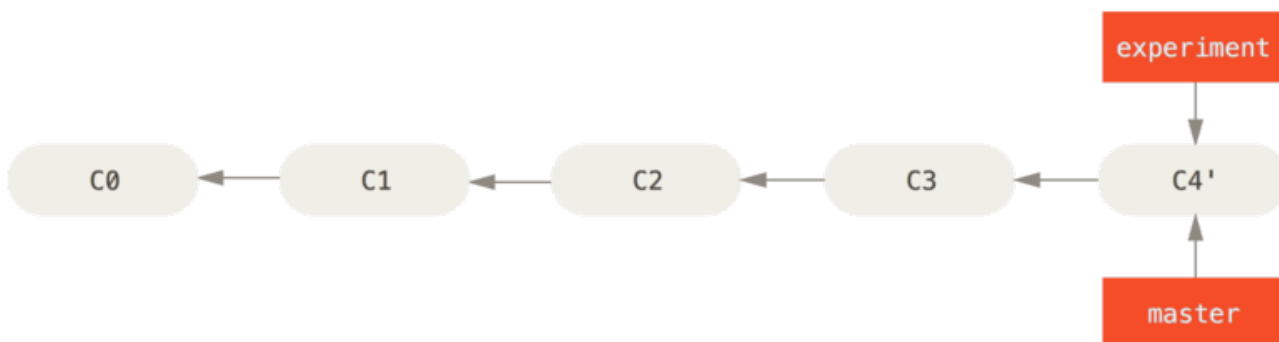


Figure 38. *Fast-forwarding the master branch*

Sedaj, ko posnetek kaže k C4 je točno enak kot tisti, ki je bil pokazan na s strani C5 v

primeru združevanja. Ni razlike v končnem produktu integracije vendar ponovno baziranje naredi čistejšo zgodovino. Če primerjate dnevnik ponovno baziranje veje izgleda kot linearna zgodovina: izgleda, da se je vso delo zgodilo v serijah tudi ko se je prvotno zgodilo vzporedno.

Pogostokrat boste to naredili, da zagotovite, da se vaša pošiljanja uporabijo čisto na oddaljeni veji - mogoče v projektu kateremu poskušate prispevati vendar ga ne vzdržujete. V tem primeru bi naredili vaše delo na veji in nato osnovali vaše delo glede na to `origin/master`, ko ste pripravljeni poslati vaše popravke glavnemu projektu. Na ta način vzdrževalcu ni treba narediti nikakršnega integracijskega dela - samo fast-forward ali čisto uporabo.

Bodite pozorni, saj posnetek na katerega kaže končno pošiljanje, s katerim ste končali, bodisi je zadnje ponovno baziranega pošiljanja za rebase ali končno pošiljanje združevanja po združevanju, gre za isti posnetek - samo zgodovina je, kar je drugačno. Rebasing ponovno predvaja spremembe iz ene vrstice dela v drugo v vrstnem redu, ki so bile predstajljene, medtem ko združevanje vzame končne točke in jih združi skupaj.

Bolj zanimivi rebasing

Lahko tudi imate vaše ponovno predvajanje rebase-a na nečem drugem od ciljne rebase veje. Vzamimo zgodovino, kot je [A history with a topic branch off another topic branch](#) za primer. Naredili ste tematsko vejo (`server`), da ste dodali nekaj funkcionalnosti strežniške strani vašemu projektu in naredili ste pošiljanje. Nato ste od tam naredili razvejanje, da ste naredili spremembe strani klienta (`client`) in nekajkrat poslali. Končno ste šli nazaj na vašo vejo server in naredili nekaj več pošiljanj.

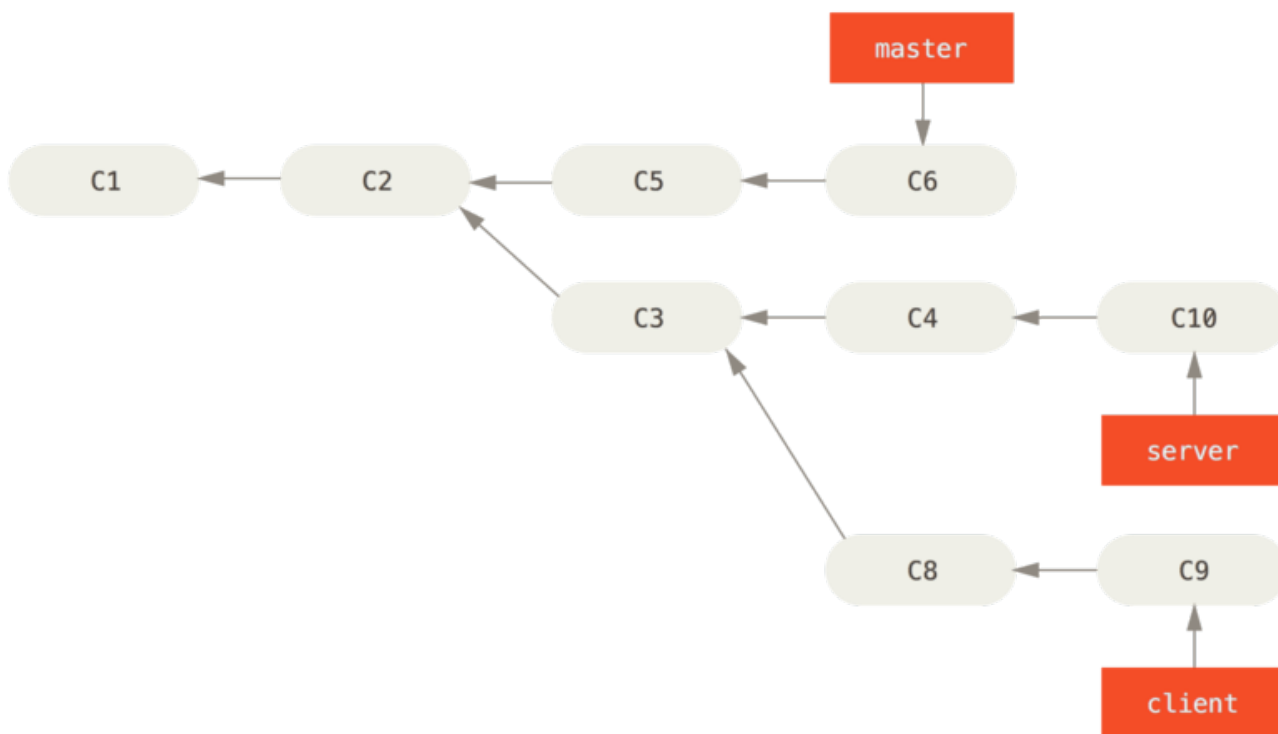


Figure 39. A history with a topic branch off another topic branch

Predpostavimo, da se odločite, da želite združiti vaše spremembe strani klienta v vašo

glavno izdajo, vendar se želite držati stran od sprememb strežniške strani dokler ni nadaljnje testirano. Lahko vzamete spremembe na klientu, ki niso na strežniku (C8 and C9) in jih ponovno predvajate na vaši veji master z uporabo opcije `--onto` na `git rebase`:

```
$ git rebase --onto master server client
```

To v osnovi pove, "Izpišite vejo client, ugotovite popravke iz skupnih prednikov vej `client` in `server` in jih nato ponovno predvajajte na ``master." Je nekoliko bolj kompleksno, vendar rezultat je precej cool.

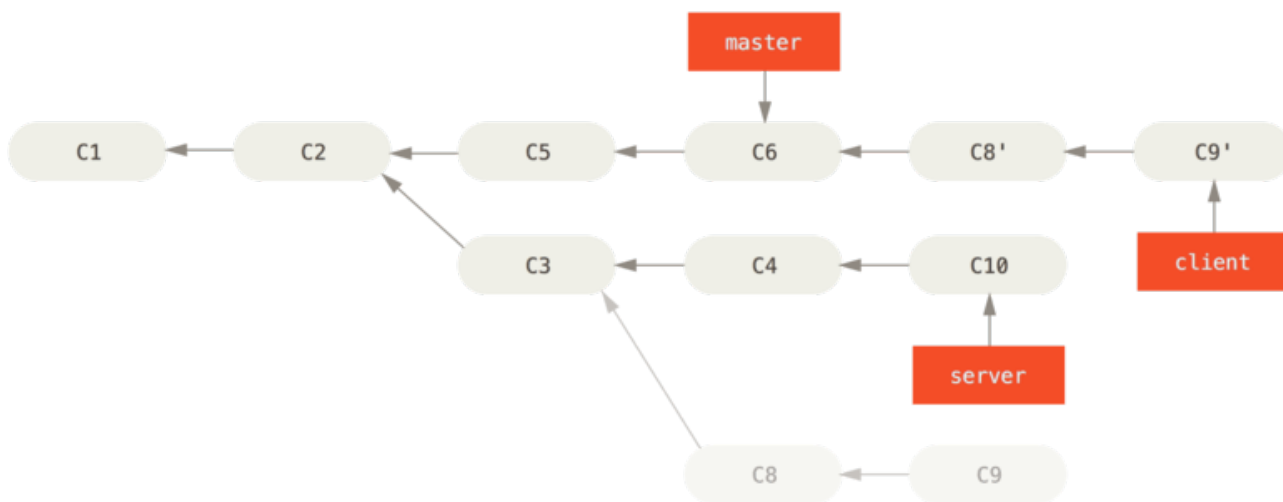


Figure 40. Rebasing a topic branch off another topic branch

Sedaj lahko naredite fast-forward na vaši veji master (glejte [Fast-forwarding your master branch to include the client branch changes](#)):

```
$ git checkout master
$ git merge client
```

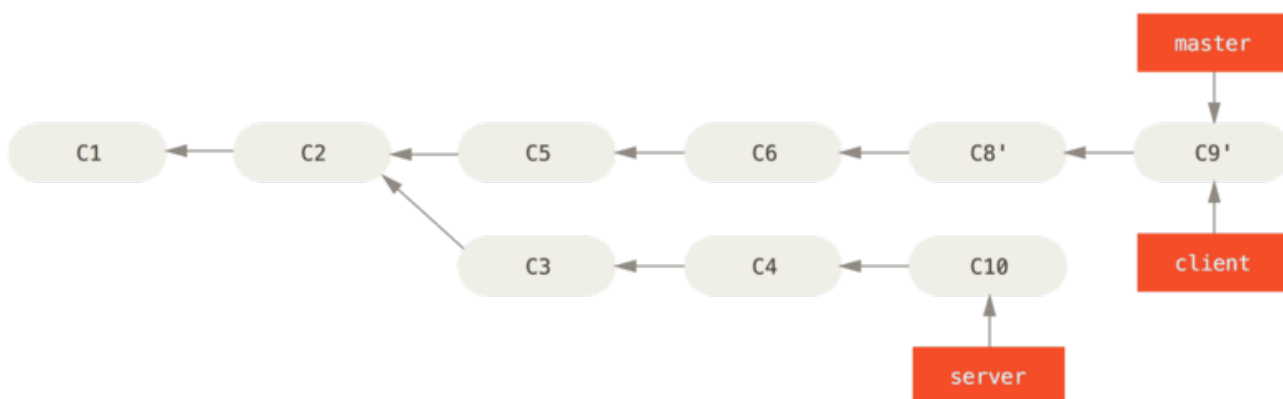


Figure 41. Fast-forwarding your master branch to include the client branch changes

Recimo, da se odločite potegniti tudi na vašo vejo server. Lahko naredite rebase na veji server glede na vejo master brez, da jo morate najprej izpisati s pogonom `git rebase [basebranch] [topicbranch]` - kar izpiše tematsko vejo (v tem primeru `server`) za vas in jo ponovno predvajate na osnovni veji (`master`):


```
$ git rebase master server
```

To ponovno predvaja vaše delo **server** na vrhu vašega dela **master**, kot je prikazano v [Rebasing your server branch on top of your master branch](#).

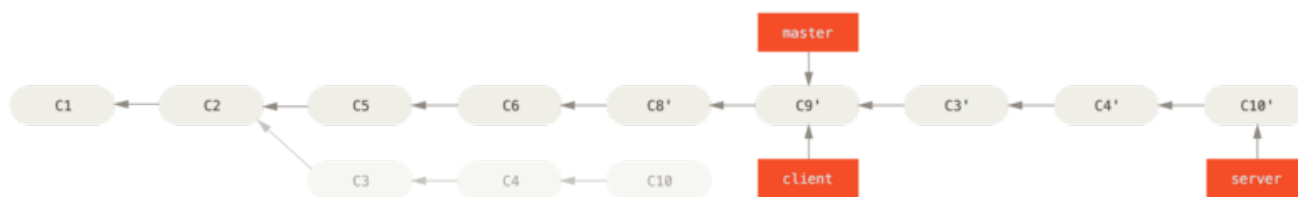


Figure 42. Rebasing your server branch on top of your master branch

Nato lahko naredite fast-forward na osnovni veji (**master**):

```
$ git checkout master  
$ git merge server
```

Lahko odstranite veji **client** in **server**, ker je celotno delo integrirano in ju ne potrebujete več, kar pusti vašo zgodovino za ta celotenn proces izgledati kot [Final commit history](#):

```
$ git branch -d client  
$ git branch -d server
```



Figure 43. Final commit history

Nevarnosti ponovnega baziranja

Ahh, vendar blagoslova ponovnega baziranja ni brez njegovih slabih strani, ki so lahko povzete v eni vrstici:

Pošiljanj, ki obstojajo izven vašega repozitorija ne ponovno bazirajte.

Če sledite tem smernicam, boste v redu. Če ne, vas bodo ljudje sovražili in zaničevani boste s strani prijateljev in družine.

Ko ponovno bazirate, opuščate obstoječa pošiljanja in ustvarjate nove, ki so podobna vendar drugačna. Če potisnete pošiljanja nekaj in jih ostali potegnejo in bazirajo svoje delo na njih in nato vi prepisete ta pošiljanja z **git rebase** in jih potisnete ponovno, bodo vaši sodelavci morali narediti ponovno združevanje njihovega dela in stvari bodo postale grde, ko poskušate potegniti njihovo delo nazaj v vaše.

Poglejmo primer, kako baziranje dela, ki ste naredili javno, lahko povzroča probleme.

Predpostavimo, da klonirate iz centralnega strežnika in nato naredite nekaj dela iz tega. Vaša zgodovina pošiljanja izgleda takole:

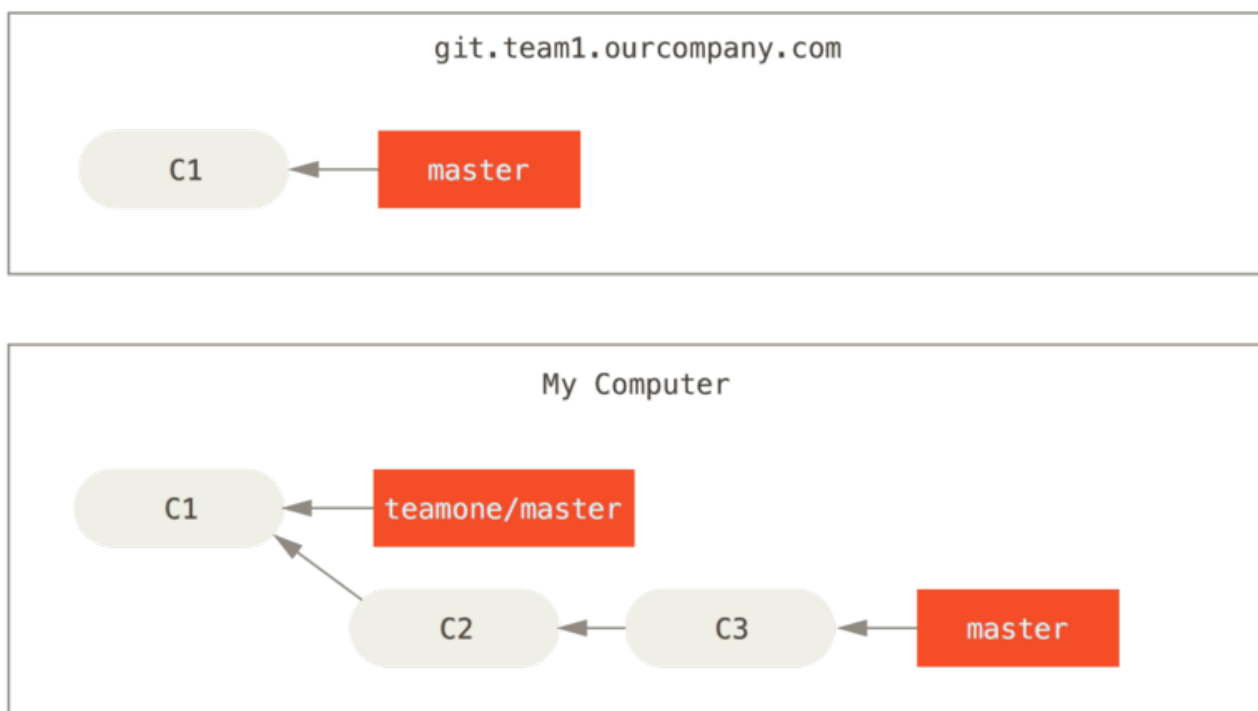


Figure 44. Clone a repository, and base some work on it

Sedaj, ko nekdo drug naredi delo, ki vključuje združitev in potiskanje, ki dela na centralnem strežniku. Ujamete ga in združite novo oddaljeno vejo v vaše delo, naredite, da vaša zgodovina izgleda nekakako takole:

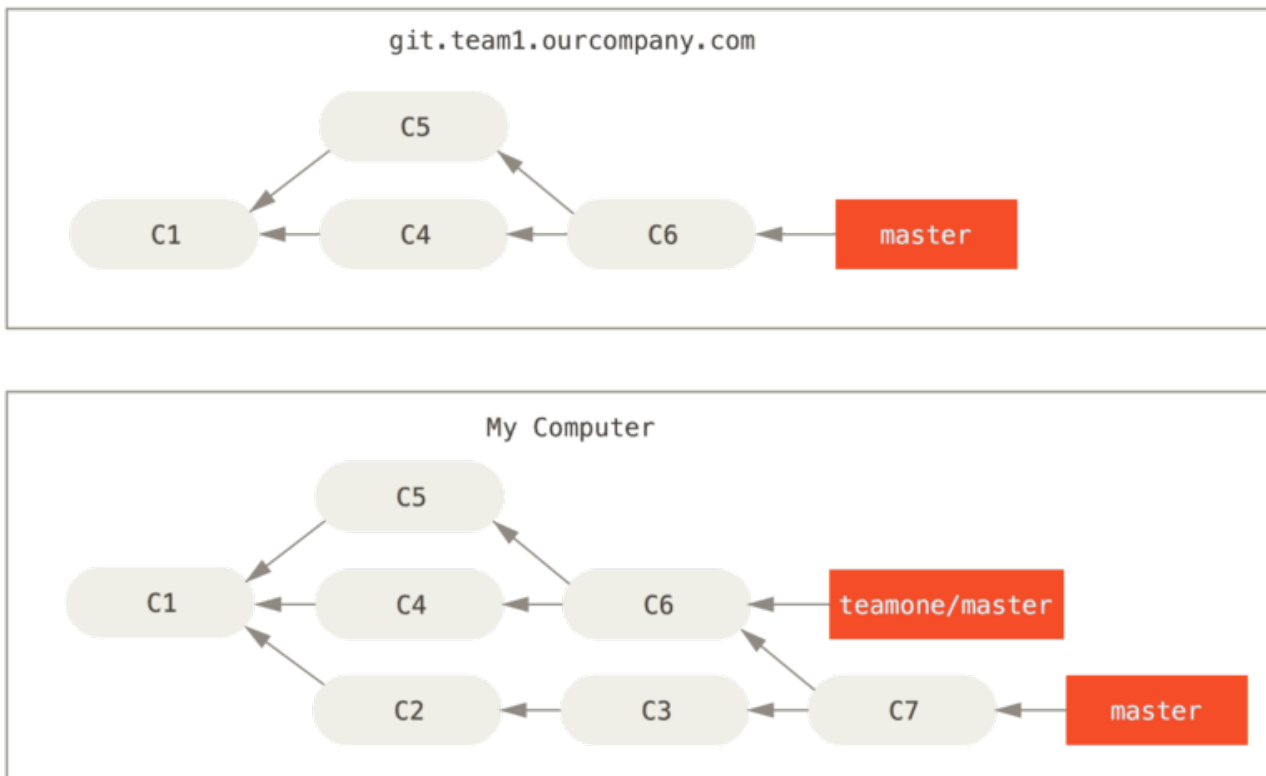


Figure 45. Fetch more commits, and merge them into your work

Naslednje, oseba, ki je potisnila združeno delo, se odloči iti nazaj in ponovno bazirati svoje delo namesto tega; naredi `git push --force`, da prepíše zgodovino na strežniku. Nato ujamete iz tega strežnika in prenesete nova pošiljanja.

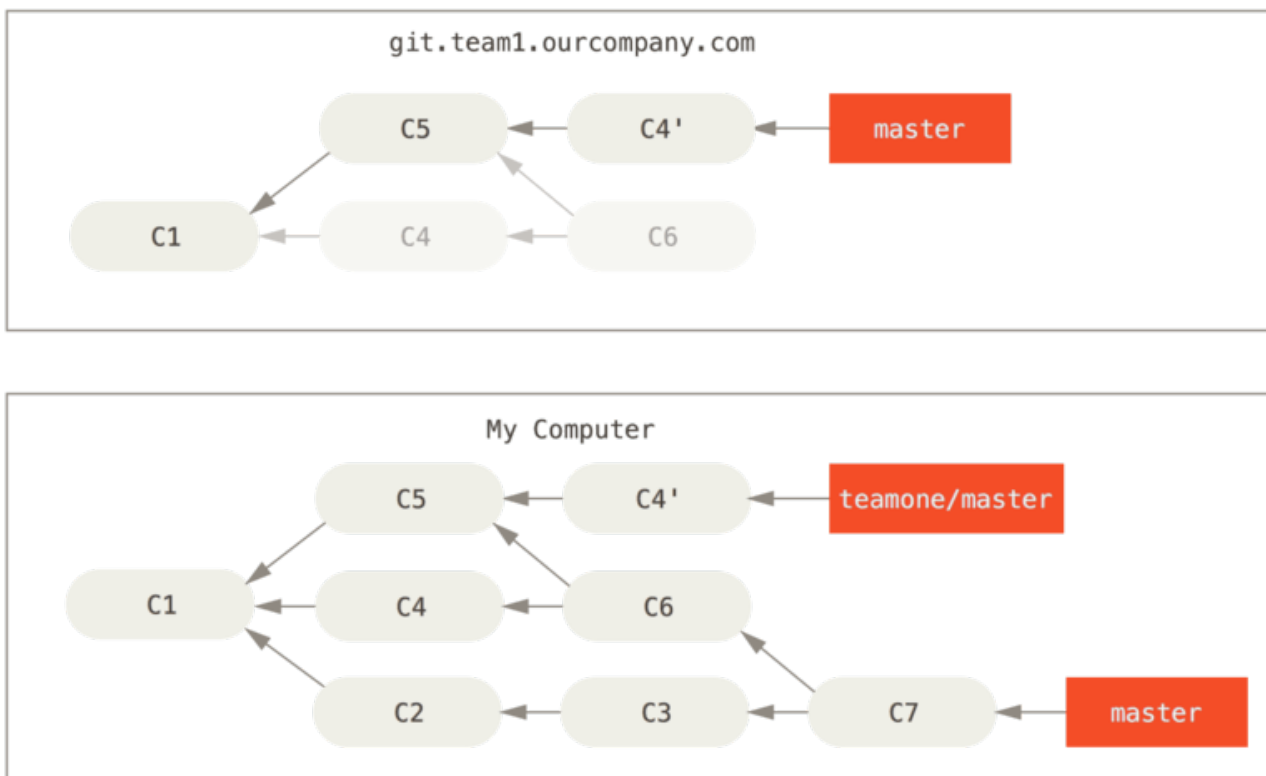


Figure 46. Someone pushes rebased commits, abandoning commits you've based your work on

Sedaj ste oboji v škripcih. Če naredite `git pull`, boste ustvarili pošiljanje združitve, ki vključuje tako vrstico zgodovine in vaš repozitorij bo izgledal takole:

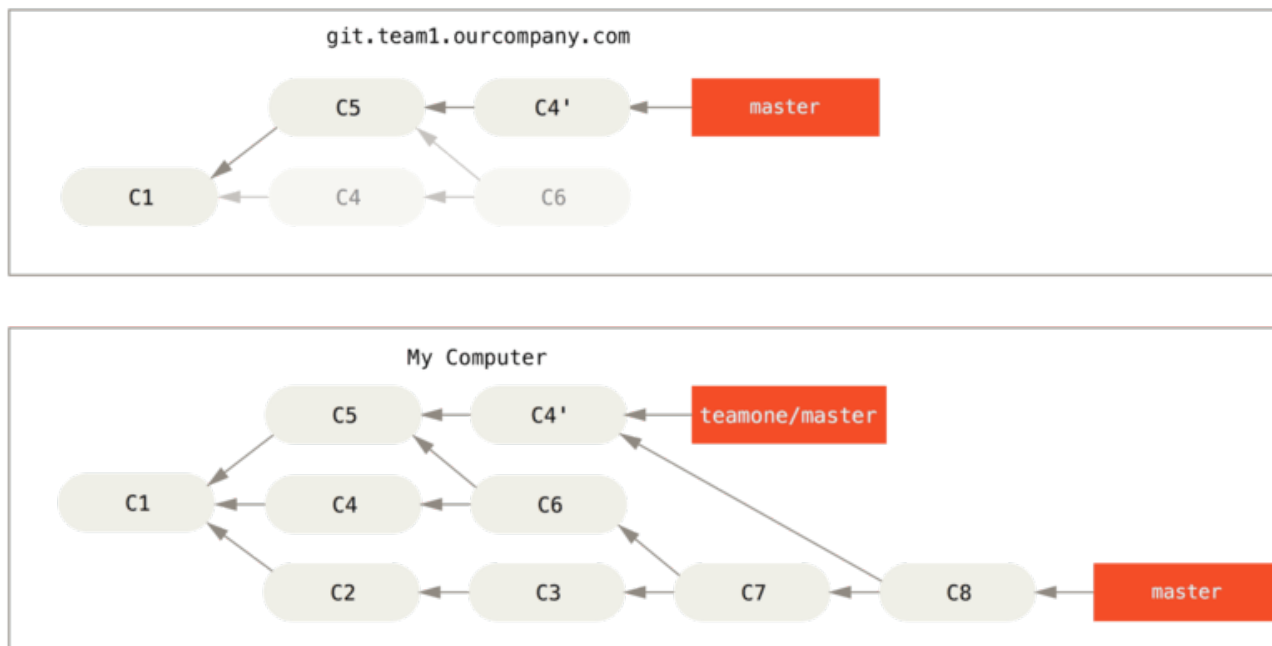


Figure 47. You merge in the same work again into a new merge commit

Če poženete `git log`, ko vaša zgodovina izgleda takole, boste videli dve pošiljanji, ki imata istega avtorja, datum in sporočilo, kar bo zmedeno. Nadalje, če potisnete to zgodovino nazaj na strežnik, boste ponovno predstavili vsa ta ponovno bazirana pošiljanja na centralnem strežniku, kar je lahko dalje zmede ljudi. Je precej varno predpostavljati, da drug razvijalec ne želi C4 in C6 v zgodovini; to je razlog, zakaj sploh ponovno baziranje.

Ponovno bazirajte ko ponovno bazirate

Če se najдете v situaciji, kot je ta, ima Git nekaj dodatne čarobnosti, ki vam lahko pomaga. Če nekdo v vaši ekipi porine spremembe, ki prepisujejo delo, na katerem ste osnovali vaše delo, je vaš iziv ugotoviti, kaj je vaše in kaj so drugi prepisali.

Izkaže se, da kot dodatek k pošiljanju preverjene vsote SHA-1, Git tudi preračuna preverjeno vsoto, ki je osnovana samo kot popravek predstavljen v pošiljanju. To se imenuje "patch-id".

Če potegneta delo, ki je bilo prepisano in osnovano na vrhu novega pošiljanja vašega partnerja lahko Git tudi pogostokrat uspešno ugotovi, kaj je unikatno vaše in jih uporabite nazaj na vrhu nove veje.

Na primer v prejšnjem scenariju če namesto, da delate združevanje, ko ste na [Someone pushes rebased commits, abandoning commits you've based your work on](#), poženemo `git rebase teamone/master`, bo Git:

- Določil, katero delo je unikatno za vašo vejo (C2, C3, C4, C6, C7)
- Določil, katera niso pošiljanja združevanja (C2, C3, C4)

- Določil, katera niso bila prepisana v ciljno vejo (samo C2 in C3, saj C4 je isti popravek kot C4')
- Uporabil ta pošiljanja na vrhu `teamone/master`

Torej namesto rezultata, ki ga vidimo v [You merge in the same work again into a new merge commit](#), bi končali z nečim bolj kot [Rebase on top of force-pushed rebase work](#).

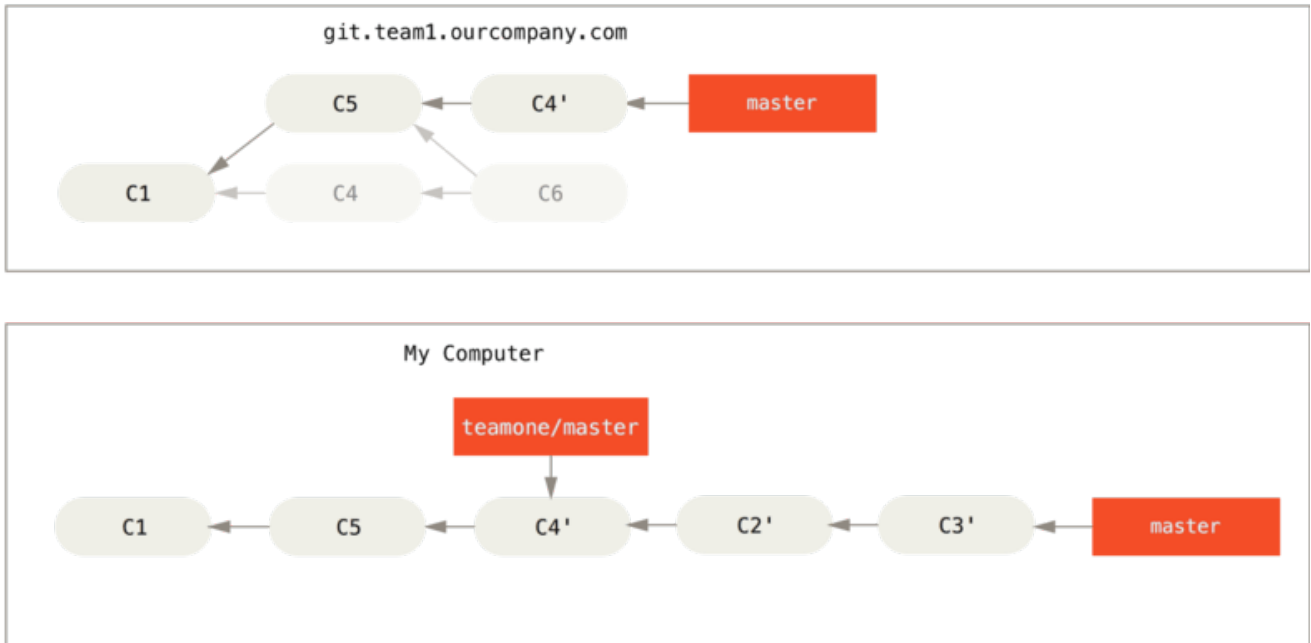


Figure 48. Rebase on top of force-pushed rebase work.

To samo deluje, če sta C4 in C4', ki ga je naredil vaš partner skoraj točno enak popravek. Drugače ponovno baziranje ne bo zmožno povedati, da je duplikat in bo dodal drug C4 podoben popravek (ki ga bo verjetno neuspešno uporabil čisto, saj bi spremembe bile vsaj nekako tam).

To lahko tudi poenostavite s pogonom `git pull --rebase` namesto normalnega `git pull`. Ali pa bi lahko naredili to ročno z `git fetch`, ki mu sledi `git rebase teamone/master` v tem primeru.

Če uporabljate `git pull` in želite narediti `--rebase` privzeto, lahko nastavite `pull.rebase` nastavitveno vrednost s nečim kot je `git config --global pull.rebase true`.

Če obravnavate ponovno baziranje kot način čiščenja in dela s pošiljanju preden jih potisnete in če samo ponovno bazirate pošiljanja, ki še nikoli niso bila na voljo javno potem boste v redu. Če ponovno bazirate pošiljanja, ki so že bila potisnjena javno in so ljudje lahko bazirali delo na teh pošiljanjih, potem ste lahko v neki frustrirajoči težavi in prezirate vaše člane ekipe.

Če vi ali partner najde to potrebno na neki točki, zagotovite, da vsi znajo pognati `git pull --rebase`, da poskusijo narediti problem potem, ko se zgodi, nekoliko bolj enostaven.

Ponovno baziranje proti združevanju

Sedaj, ko ste videli ponovno baziranje in združevanje v delovanju, se lahko sprašujete katero je boljše. Preden lahko to odgovorimo, pojdimo korak nazaj in povejmo o tem, kaj zgodovina pomeni.

Ena točka pogleda na to je, da je zgodovina pošiljanja vašega repozitorija **posnetek kaj se je dejansko zgodilo**. Je zgodovinski dokument, vreden svojega lastnega prav in ne bi smel biti ponarejen. Iz tega zornega kota spreminjanje zgodovine pošiljanja je skoraj bogolketno; *ležite*, kaj se je dejansko zgodilo. Torej kaj če je grda serijca pošiljanj združevanja? Tako se je zgodilo in repozitorij bi moral ohraniti to za zaznamce.

Nasprotno stališče je, da je zgodovina pošiljanja **zgodba, kako je bil vaš projekt narejen**. Ne bi objavili prvega osnutka knjige in navodila, kako vzdrževati programsko opremo, zaslužijo previdno urejanje. To je tabor, ki uporablja orodja, kot je rebase in filter-branch, da povejo zgodbo na način, ki je najboljši za bodoče bralce.

Sedaj vprašanje ali je združevanje ali ponovno baziranje boljše: upajmo, da boste videli, da ni tako enostavno. Git je močno orodje in vam omogoča narediti mnogo stvari in z vašo zgodovino, vendar vsaka ekipa in vsak projekt je drugačen. Sedaj ko veste kako obe stvari delujeta, je odvisno od vas, da se odločite, katera je najboljša za vašo določeno situacijo.

V splošnem način za dobiti najboljše obeh svetov je ponovno baziranje lokalnih sprememb, ki ste jih naredili vendar niste delili preden ste jih potisnili, da počistite vašo zgodbo, vendar nikoli ničesar ponovno bazirali, ste potisnili nekam.

Povzetek

Pokrili smo osnove vej in združevanja v Git-u. Morali bi se počutiti udobno z izdelavo in preklapanjem na nove veje, preklapanjem med vejami in združevanjem lokalnih vej skupaj. Morali bi tudi biti sposobni deliti vaše veje s potiskanjem le teh na deljeni strežnik, delo z ostalimi na deljenih vejah in ponovno baziranje (rebasing) vaših vej preden so deljene. Naslednje, bomo pokrili, kaj potrebujete, da poganjate vaš lastni Git strežnik gostujočih repozitorijev.

Git na strežniku

Na tej točki, bi morali biti sposobni narediti večino dnevnih opravil zaradi katerih boste uporabljali Git. Vendar, da naredite kakršno koli sodelovanje v Git-u, boste potrebovali imeti oddaljeni Git repozitorij. Čeprav lahko tehnično porinete spremembe v in potegnete spremembe iz posameznih repozitorijev, se to ne svetuje, ker lahko precej enostavno zamešate, na čemer se dela, če niste pazljivi. Poleg tega želite, da vaši sodelavci lahko dostopajo do repozitorija tudi če je vaš računalnik offline - imeti bolj zanesljiv skupni repozitorij je pogostokrat uporabno. Zato je željena metoda za sodelovanje z nekom nastaviti vmesni repozitorij, do katerega imata oba dostop ter porinete in potegneta iz njega.

Poganjanje Git strežnika je precej enostavno. Prvo izberete, katere protokole želite, da z njimi strežnik komunicira. Prva sekcija tega poglavja bo pokrila protokole, ki so na voljo ter prednosti in slabosti vsakega. Naslednja sekcija bo razložila nekatere tipične nastavitve z uporabo teh protokolov in kako pripravite vaš strežnik, da dela z njimi. Zadnje bomo šli skozi nekaj opcij gostovanja, če nimate težav z gostovanjem vaše kode na strežniku nekoga drugega in ne želite iti skozi težave nastavitvev in vzdrževanja vašega lastnega strežnika.

Če nimate nobenega interesa poganjati vaš lastni strežnik, lahko zadnjo sekcijo poglavja preskočite, da vidite nekaj opcij nastavitvev gostujočega računa in se nato premaknete na naslednje poglavje, kjer bomo govorili o različnih vhodih in izhodih dela v distribuiranem okolju upravljanje izvorne kode.

Oddaljeni repozitorij je v splošnem *goli repozitorij* - Git repozitorij, ki nima delujočega direktorija. Ker je repozitorij uporabljen samo kot točka sodelovanja, ni razloga imati posnetek izpisan na disk; gre samo za podatke Git. V najenostavnejših terminih, goli repozitorij je vsebina vašega projektnega direktorija `.git` in nič drugega.

Protokoli

Git lahko uporablja štiri glavne protokole za prenos podatkov: Local, HTTP, Secure Shell (SSH) in Git. Tu bomo govorili, kaj so in katere osnovne okoliščine bi želeli imeti (ali ne želeli), da jih uporabljate.

Lokalni protokoli

Najbolj osnovni je *lokalni protokol* v katerem je oddaljeni repozitorij v drugem direktoriju na disku. To je pogostokrat uporabljeno, če vsi v vaši ekipi imajo dostop do deljenega datotečnega sistema, kot je NFS mount ali v manj verjetnem primeru se vsi prijavijo v isti računalnik. Zadnje ne bi bilo idealno, ker vsa vaša instanca repozitorija kode bi domovala na istem računalniku, kar naredi katastrofične izgube bolj verjetne.

Če imate priklopljen deljeni datotečni sistem, potem lahko klonirate, porivate in potegneta iz lokalnega datotečno osnovanega repozitorija. Da tako klonirate repozitorij ali enega dodate kot oddaljenega k obstoječemu projektu, uporabite pot do repozitorija kot URL. Na primer, da klonirate lokalni repozitorij, lahko poženete nekaj takega:

```
$ git clone /opt/git/project.git
```

Or you can do this:

```
$ git clone file:///opt/git/project.git
```

Git operira malenkost drugače, če eksplicitno določite `file://` na začetku URL-ja. Če samo določite pot, Git poskuša uporabiti trde povezave ali direktno kopiranje datotek, ki jih potrebuje. Če določite `file://`, Git požene proces, ki ga običajno uporablja za prenos datotek podatkov preko omrežja, ki je v splošnem veliko manj učinkovita metoda prenosa podatkov. Glavni razlog za določanje predpone `file://` je, če želite čisto kopijo repozitorija z izpuščenimi tujimi referencami ali objekti - v splošnem po importiranju iz drugega sistema kontrole verzij ali nečesa podobnega (glejte [Notranjost Git-a](#) za opravila vzdrževanja). Tu bomo uporabili običajno pot, ker je to skoraj vedno hitrejše.

Da dodate lokalni repozitorij obstoječemu Git projektu, lahko poženete nekaj takega:

```
$ git remote add local_proj /opt/git/project.git
```

Nato lahko porinete ali potegnete iz tega oddaljenega, kot bi to naredili preko omrežja.

Prednosti

Prednosti datotečno osnovanih repozitorijev so, da so enostavni in da uporabljajo obstoječe pravice datotek in dostopa omrežja. Če že imate deljeni datotečni sistem, do katerega ima dostop celotna ekipa, je nastavev repozitorija zelo enostavna. Lahko se držite gole kopije repozitorija nekje, kjer ima vsakdo deljeni dostop in nastavite pravice pisanja/branja, kakor bi to naredili za katerikoli drugi deljeni direktorij. Govorili bomo, kako izvoziti golo kopijo repozitorija za ta razlog v [Pridobiti Git na strežnik](#).

To je tudi lepa opcija za hitro prijetje dela iz delovnega repozitorija nekoga drugega. Če vi in vaš sodelavec delate na istem projektu in želijo nekaj izpisati, je pogon ukaza, kot je `git pull /home/john/project`, pogostokrat enostavnejši kot porivanje v oddaljeni strežnik in da nato potegnete.

Slabosti

Slabosti te metode so, da je deljeni dostop v splošnem težji za nastaviti in doseči iz večih lokacij kot osnovni dostop omrežja. Če želite poriniti iz vašega prenosnika, ko ste doma, morate priklopiti oddaljeni disk, kar je lahko težko in počasno v primerjavi z dostopom na osnovi omrežja.

Pomembno je omeniti, da to ni nujno najhitrejša opcija, če uporabljate deljeni priklop neke vrste. Lokalni repozitorij je hiter samo, če imate hiter dostop do podatkov. Repozitorij na NFS je pogostokrat počasnejši kot repozitorij preko SSH na istem strežniku, kar omogoča Git-u, da poganja lokalne diske na vsakem sistemu.

In nazadnje, ta protokol ne ščiti repozitorijev proti škodi po nesreči. Vsak uporabnik ima polni lupinski dostop do oddaljenega direktorija in nič mu ne preprečuje spremeniti ali odstraniti notranje Git datoteke in poškodovati repozitorij.

HTTP protokoli

Git lahko komunicira preko HTTP v dveh različnih načinih. Pred Git 1.6.6 je bil samo en način, da to lahko naredi, kar je bilo zelo enostavno in v splošnem samo za branje. V verziji 1.6.6 je bil predstavljen nov pametni protokol, ki je vključeval Git, da je bil sposoben inteligentno pogajati prenos podatkov na način podoben, kakor to dela preko SSH. V zadnjih nekaj letih, je ta novi HTTP protokol postal zelo popularen, saj je enostavnejši za uporabnika in pametnejši kako komunicira. Novejša verzija je pogostokrat sklicana kot “Smart” HTTP protokol in starejši način kot “Dump” HTTP. Najprej bomo pokrili novejši “smart” HTTP protokol.

Pametni HTTP

Pametni oz. t.i. “smart” HTTP protokol operira zelo podobno kot SSH ali Git protokola vendar teče preko standardnih HTTP/S portov in lahko uporablja različne HTTP mehanizme avtentikacije, kar pomeni, da je enostavnejši na uporabniški strani kot SSH, odkar lahko uporabite stvari kot je osnovna avtentikacija z uporabniško imenom in geslom namesto, da morate nastaviti ključe SSH.

Verjetno je sedaj postal najpopularnejši način za uporabo Git-a, saj je lahko nastavljen, da ponuja tako anonimne, kot je protokol `git://`, in lahko je poslan tudi preko z overitvijo in ekripcijo kot je protokol SSH. Namesto, da morate nastavljati različne URL-je za te stvari, lahko sedaj uporabite en URL za oba. Če poskusite potisniti in repozitorij zahteva overitev (kar bi običajno moral), strežnika lahko vpraša za uporabniško ime in geslo. Isto gre za bralni dostop.

V bistvu za storitve kot je GitHub, je URL, ki ga uporabljate za ogled repozitorija na spletu (na primer “`https://github.com/schacon/simplegit[]`”) enak URL, ki ga lahko upraborte za kloniranje in, če imate dostop potiskanje.

Neumni HTTP

Če se strežnik ne odzove s pametno Git HTTP storitvijo, se bo Git klient poskušal vrniti k enostavnejšemu “dump” protokolu HTTP. Neumni protokol pričakuje, da je goli repozitorij Git ponujen kot običajne datoteke s spletnega strežnika. Lepota neumnega HTTP protokola je enostavnost nastavitve. V osnovi je vse kar morate narediti, dati goli repozitorij Git pod vaš vrhnji HTTP dokumenti direktorij in nastaviti določeno kjuko `post-update` in ste končali (Glejte [Git kljuke](#)). Na tej točki kdorkoli lahko dostopa do spletnega strežnika pod katerim ste dali repozitorij in lahko tudi klonira vaš repozitorij. Da omogočite bralni dostop do vašega repozitorija preko HTTP, naredite nekaj takega:

```
$ cd /var/www/htdocs/  
$ git clone --bare /path/to/git_project gitproject.git  
$ cd gitproject.git  
$ mv hooks/post-update.sample hooks/post-update  
$ chmod a+x hooks/post-update
```

To je vse. Kljuka `post-update`, ki prihaja z Git-om privzeto poganja ustrezeni ukaz (`git update-server-info`), da naredite HTTP ujetje in kloniranje ustrezno delujoče. Ta ukaz se izvede, ko potisnete v ta repozitorij (preko SSH); nato lahko ostali ljudje klonirajo preko tega nekako takole

```
$ git clone https://example.com/gitproject.git
```

V tem določenem primeru, uporabljamo pot `/var/www/htdocs`, ki je pogosta za nastavitve Apache, vendar lahko uporabite katerikoli statični spletni strežnik - samo dajte goli repozitorij v njegovo pot. Podatki Git so ponujeni kot osnovne statične datoteke (glejte [Notranjost Git-a](#) za podrobnosti o tem, kako točno je ponujen).

V splošnem bi ali izbrali, da se poganja bralno/pisalni pametni HTTP strežnik ali enostavno imate datoteke dostopne kot samo bralno v neumnem načinu. Redko je poganjati mešanico obeh storitev.

Prednosti

Osredotočili se bomo na prednosti pametne verzija HTTP protokola.

Enostavnost imetja enega URL-ja za vse tipe dostopov in imeti strežniški poziv samo ko je potrebna overitev, naredi stvari zelo enostavne za končnega uporabnika. Da ste zmožni overiti z uporabniškim imenom in geslom je tudi velika prednost preko SSH, saj uporabnikom ni treba generirati SSH ključev lokalno in naložiti njihovih javnih ključev strežniku preden so zmožni imeti interakcijo z njim. Za manj sofisticirane uporabnike ali uporabnike na sistemih, kjer je SSH manj pogost, je to glavna prednost v uporabnosti. Je tudi zelo hiter in učinkovit protokol, podobno kot je SSH.

Lahko tudi ponudite vaše repozitorije kot samo bralne preko HTTPS, kar pomeni, da lahko šifirate vsebino prenosa; ali lahko greste dalje in naredite, da klienti uporabljajo določene podpisane SSL certifikate.

Druga lepa stvar je, da so HTTP/S tako pogosto uporabljeni protokoli, da so korporacijski požarni zidovi pogostokrat nastavljeni, da omogočajo promet preko teh portov.

Slabosti

Git je lahko preko HTTP/S bolj zahtevak za nastaviti v primerjavi s SSH na nekaterih strežnikih. Razen tega je zelo malo prednosti, ki jih imajo ostali protokoli preko "pametnega" HTTP protokola za ponujanje Git-a.

Če uporabljate HTTP za overitveno porivanje, je ponujanje vaših poverilnic včasih bolj

komplicirano kot uporaba ključev preko SSH. Vendar so nekatera orodja predpomnjenja poverilnic, ki jih lahko uporabite, vključno s Keychain dostopom na OSX in Credential Manager-jem na Windows, da naredi to precej neboleče. Preberite [Credential Storage](#), da vidite kako nastaviti varno predpomnenje HTTP gesel na vašem sistemu.

Protokol SSH

Pogosti protokol prenosa za Git, ko sami gostujete je preko SSH. To je zato, ker je dostop SSH na strežnikih že večinoma nastavljen - in če ni, je to enostavno narediti. SSH je tudi overitveni omrežni protokol; in ker je vseposoten, je v splošnem enostaven za nastaviti in uporabljati.

Da klonirate repozitorij Git preko SSH, lahko določite URL ssh:// kot:

```
$ git clone ssh://user@server/project.git
```

Lahko pa uporabite kratko scp-podobno sintakso za protokol SSH:

```
$ git clone user@server:project.git
```

Lahko tudi ne določite uporabnika in Git predpostavlja, da je uporabnik trenutno prijavljen kot je.

Prednosti

Prednosti za uporabo SSH je mnogo. Najprej, SSH je relativno nastaviti - SSH prikriti procesi so pogosti, mnogi administratorji omrežij imajo izkušnje z njimi in mnoge OS distribucije so nastavljene z njimi ali imajo orodja za njihovo upravljanje. Naslednje, dostop preko SSH je varen - vsi poslani podatki so šifrirani in overjeni. Zadnje, kot HTTP/S, Git in lokalni protokoli je SSH učinkovit, kar naredi podatke kompaktne kolikor je možno pri prenašanju.

Slabosti

Negativni pogled SSH-ja je, da ne morete ponujati anonimnega dostopa vašega repozitorija preko tega. Ljudje morajo imeti dostop do vaše naprave preko SSH, tudi v zmožnosti samo branja, kar ne naredi SSH dostopa prevodnega za odprtokodne projekte. Če ga uporabljate samo znotraj vašega organizacijskega omrežja, je SSH lahko edini protokol s katerim se boste morali ukvarjati. Če želite dovoliti anonimno samo bralni dostop do vaših projektov in tudi želite uporabljati SSH, boste morali nastaviti SSH, da lahko potiskate preko njega, vendar nekaj drugega za ostale, da lahko ujemajo.

Protokol Git

Naslednji je protokol Git. To je posebni prikriti proces, ki prihaja v paketu z Git-om; posluča na namenskem portu (9418), ki ponuja storitev podobno protokolu SSH, vendar z absolutno brez overjanja. Da je lahko repozitorij ponujen preko protokola Git, morate

ustvariti datoteko `git-daemon-export-ok` - prikriti proces ne bo ponujal repozitorija brez te datoteke v njem - vendar razen tega ni nobene varnosti. Bodisi je repozitorij Git na voljo za vsakogar, da klonira ali pa ni. To pomeni, da v splošnem ni nobenega potiskanja preko tega protokola. Lahko omogočite dostop porivanja; vendar z danim manjkanjem overjanja, če vključite dostop potiskanja, kdorkoli na internetu, ki najde URL vašega projekta, lahko potisne v vaš projekt. Dovolj je reči, da je to redko.

Prednosti

Protokol Git je pogostokrat najhitrejši omrežni protokol na voljo. Če ponujate veliko prometa za javni projekt ali ponujate zelo velik projekt, ki ne zahteva uporabniškega overjanja za dostop branja, je verjetno, da boste želeli nastaviti Git prikriti proces, da ponuja vaš projekt. Uporablja enak mehanizem prenosa podatkov kot protokol SSH, vendar brez enkripcije in nadglavega overjanja.

Slabosti

Slabost protokola Git je pomanjkanje overjanja. V splošnem ni zaželeno za protokol Git, da je edini dostop do vašega projekta. V splošnem ga boste združili z dostopom SSH ali HTTPS za nekaj razvijalcev, ki imajo dostop potiskanja (branje) in za vse ostale uporabili `git://` za dostop samo branja. Je verjetno tudi najbolj težek protokol za nastaviti. Poganjati mora svoj lastni prikriti proces, ki zahteva `xinetd` nastavitve ali podobno, kar večinoma ni ravno sprehod po parku. Tudi zahteva dostop požarnega zidu za port 9418, kar ni standardni port, ki ga organizacijski požarni zidovi vedno dovoljujejo. Za velikimi organizacijskimi požarnimi zidovi je ta nejasen port pogostokrat blokiran.

Pridobiti Git na strežnik

Sedaj bomo šli skozi nastavitve Git storitve, ki poganja te protokole na vašem lastnem strežniku.

NOTE

Tu bomo demonstrirali ukaze in potrebne korake za osnovne, poenostavljene namestitve na Linux osnovani strežnik, čeprav je tudi možno poganjati te storitve tudi na Mac ali Windows strežnikih. V bistvu nastavitve produkcijskega strežnika znotraj vaše infrastrukture bo zagotovo povzročilo razlike v varnostnih ukrepih ali orodjih operacijskih sistemov, vendar upajmo, da vam bo to dalo splošno idejo, kaj je vključeno.

Da se začetno nastavi katerikoli Git strežnik, morate izvoziti obstoječi repozitorij v nov gol repozitorij - repozitorij, ki ne vsebuje delujočega direktorija. To je v splošnem precej enostavno narediti. Da klonirate vaš repozitorij, da ustvarite nov gol repozitorij, pošენite ukaz kloniranja z opcijo `--bare`. Po dogovoru se direktoriji golega repozitorija končajo z `.git`, sledeče:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

Sedaj bi morali imeti kopijo podatkov direktorija Git v vašem direktoriju `my_project`.

To je v grobem ekvivalentno nečemu takemu

```
$ cp -Rf my_project/.git my_project.git
```

Je nekaj manjših razlik v nastavitveni datoteki; vendar za vaše razloge je to blizu podobne stvari. Vzame sam repozitorij Git, brez delujočega direktorija in ustvari direktorij posebej zanj samega.

Dajanje golega repozitorija na strežnik

Sedaj, ko imate golo kopijo vašega repozitorija, vse kar potrebujete narediti je, da ga date na strežnik in nastavite vaše protokole. Recimo, da ste nastavili strežnik imenovan `git.example.com`, do katerega imate SSH dostop in želite shraniti vse vaše repozitorije Git pod direktorij `/opt/git`. Predpostavljamo, da `/opt/git` obstaja na tem strežniku, lahko nastavite vaš novi repozitorij s kopiranjem vašega golega repozitorija preko:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

Na tej točki, ostali uporabniki, ki imajo SSH dostop do istega strežnika, ki ima bralni dostop do direktorija `/opt/git` lahko klonirajo vaš repozitorij s pogonom

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

Če se uporabnik prijavi preko SSH v strežnik in ima pisalni dostop do direktorija `/opt/git/my_project.git`, bo tudi avtomatično imel dostop potiskanja.

Git bo avtomatično dodal skupino pravic pisanja k repozitoriju ustrezno, če poženete ukaz `git init` z opcijo `--shared`.

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

Vidite, kako enostavno je narediti repozitorij Git, ustvariti golo verzijo in ga postaviti na strežnik do katerega imate vi in vaši sodelavci dostop SSH. Sedaj ste pripravljeni na sodelovanje na istem projektu.

Pomembno je omeniti, da je to dobesedno vse, kar potrebujete pognati za uspešen

strežnik Git, do katerega ima več ljudi dostop - samo dodajte SSH-zmožne račune na strežnik in se držite golega repozitorija nekje, da vsi tisti uporabniki imajo bralni in pisalni dostop do njega. Ste pripravljeni za pogon - nič drugega ni potrebnega.

V naslednjih nekaj sekcija boste videli, kako razširiti na bolj sofisticirane nastavitve. Ta diskusija bo vključevala, da ni potrebno ustvariti uporabniških računov za vsakega uporabnika, dodajanje javnega bralnega dostopa k repozitorijem, nastavljanje spletnih uporabniških vmesnikov in več. Vendar pomnite, da za sodelovanje z večimi ljudmi na privatnem projektu je vse kar *morate* narediti_ SSH strežnik in goli repozitorij.

Majhne nastavitve

Če gre za manjše stvari ali samo poskušate Git v vaši organizaciji in imate samo nekaj razvijalcev, so stvari lahko enostavne za vas. Ena najbolj kompliciranih aspektov nastavljanja strežnika Git je upravljanje uporabnikov. Če želite, da je nekaj repozitorijev samo bralnih za določene uporabnike in bralno/pisalnih za ostale, so lahko dostop in pravice malo bolj zahtevne za poskrbeti.

Dostop SSH

Če imate strežnik do katerega vsi vaši razvijalci že imajo dostop SSH, je v splošnem najenostavnejše nastaviti vaš prvi repozitorij tam, ker nimate za narediti skoraj nič dela (kot smo pokrili v zadnji sekciji). Če želite bolj kompleksen tip kontrole dostopa pravic na vaših repozitorijih, jih lahko upravljate z običajnimi pravicami datotečnega sistema operacijskega sistema na katere teče vaš strežnik.

Če želite postaviti vaše repozitorije na strežnik, ki nima računov za vsakogar v vaši ekipi za katero želite imeti dostop pisanja, potem morate zanje nastaviti dostop SSH. Predpostavljamo, da če imate strežnik s katerim to naredite, že imate nameščen SSH strežnik in to je način, kako dostopate do strežnika.

Na voljo je nekaj načinov, na katere lahko date dostop za vse v vaši ekipi. Prvo je nastaviti račune za vsakogar, kar je enostavno vendar okorno. Morda ne želite poganjati **adduser** in nastavljanje začasnih gesel za vsakega uporabnika.

Drugi način je ustvariti enega uporabnika *git* na napravi, vprašati vsakega uporabnika, ki bo imel pisalni dostop, da vam pošlje SSH javni ključ in dodate ta ključ v datoteko **~/.ssh/authorized_keys** vašega novega *git* uporabnika. Na tej točki bo vsak lahko dostopa do te naprave preko uporabnika *git*. To ne vpliva na podatke pošiljanja na kakršenkoli način - SSH uporabnik, s katerim se povezujete, ne vpliva na pošiljanja, ki jih snemate.

Drug način je, da mora vaš strežnik SSH overiti iz strežnika LDAPali neke drugega centraliziranega overitvenega vira, ki ga že morda imate nastavljenega. Dokler vsak uporabnik lahko dobi lupinski dostop na napravi, katerikoli SSH overitveni mehanizem, ki si ga lahko izmislite, bi moral delati.

Generiranje vaših javnih ključev SSH

Tako kot je povedano, mnogi strežniki Git izvajajo avtentikacijo z uporabo javnih

ključev SSH. Za ponujanje javnega ključa, mora vsak uporabnik v vašem sistemu generirati enega, če ga še nimajo. Ta proces je podoben skozi mnoge operacijske sisteme. Najprej morate preveriti, da zagotovite, da nimate že ključa. Privzeto, uporabniki ključi SSH so shranjeni v direktoriju `~/.ssh` le tega uporabnika. Lahko enostavno preverite, da vidite, če že imate ključ, tako da greste v ta direktorij in izpišete vsebino:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa      known_hosts
config            id_dsa.pub
```

Iščete par datotek poimenovanih nekako kot `id_dsa` ali `id_rsa` in ujemanje datoteke s končnico `.pub`. Datoteka `.pub` je vaš javni ključ in druga datoteka je vaš privatni ključ. Če nimate teh datotek (ali nimate niti `.ssh` direktorij), jih lahko ustvarite s pogonom programa imenovanega `ssh-keygen`, ki je ponujen v paketu SSH na Linux/Mac sistemih in prihaja z MSysGit paketom na Windows:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/schacon/.ssh/id_rsa):
Created directory '/home/schacon/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/schacon/.ssh/id_rsa.
Your public key has been saved in /home/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
d0:82:24:8e:d7:f1:bb:9b:33:53:96:93:49:da:9b:e3  schacon@mylaptop.local
```

Najprej potrdi, kam želite shraniti ključ (`.ssh/id_rsa`) in nato dvakrat vpraša za geslo, ki ga lahko pustite prazno, če ga ne želite vpisovati, ko uporabljate ključ.

Sedaj vsak uporabnik, ki to naredi, mora poslati njegov javni ključ vam ali komurkoli, ki administrira Git strežnik (v predpostavki, da uporabljate SSH strežnik, ki zahteva javne ključe). Vse kar morajo narediti je kopirati vsebino datoteke `.pub` in jo poslati po e-pošti. Javni ključi izgledajo nekako takole:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEaklOUpkDHRfHY17SbrmTIpNLTKG9Tjom/BWDSU
GPL+nafzLHDTYW7hdI4yZ5ew18JH4JW9jbbUFRviQzM7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSLVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnTPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Za bolj poglobljen vodič o izdelavi SSH ključev na večih operacijskih sistemih, glejte GitHub-ov vodič o SSH ključih na <https://help.github.com/articles/generating-ssh-keys>.

Nastavitev strežnika

Pojdimo skozi nastavitve dostopa SSH na strežniški strani. V tem primeru boste uporabili metodo `authorized_keys` za overitev vaših uporabnikov. Predpostavljamo tudi, da poganjate standardno distribucijo Linux-a kot je Ubuntu. Najprej ustvarite uporabnika `git` in direktorij `.ssh` za tega uporabnika.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Naslednje morate dodati nekaj razvijalskih javnih ključev SSH v datoteko `authorized_keys` za uporabnika `git`. Predpostavimo, da imate nekaj zaupljivih javnih ključev in imate nekaj njih v začasnih datotekah. Ponovno, javni ključi izgledajo nekako takole:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPPK+4k
Yjh6541NYSnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIGS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtpofwFB1gc+myiv
07TCUSBdLQlGMVOFq1I2uPWQ0kOWQAHuKE0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Samo pripravite jih k `git` uporabnikovi datoteki `authorized_keys` v njegovem direktoriju `.ssh`:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

Sedaj lahko nastavite prazen repozitorij zanje s pogonom `git init` z opcijo `--bare`, ki inicializira repozitorij brez delovnega direktorija:

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git init --bare
Initialized empty Git repository in /opt/git/project.git/
```

Nato lahko John, Josie ali Jessica porinejo prvo verzijo njihovega projekta v ta repozitorij z njegovim dodajanjem kot daljave in porivanjem veje. Velja opomniti, da se mora nekdo prijaviti preko lupine v napravo in ustvariti goli repozitorij vsakič, ko želite dodati projekt. Uporabimo `gitserver` kot ime gostitelja strežnika na katerem ste nastavili vašega

uporabnika `git` in repozitorij. Če ga poganjate interno in ste nastavili DNS za `gitserver`, da kaže na ta strežnik, potem lahko uporabite ukaze precej kot so (ob predpostavki, da je `myproject` obstoječi projekt z datotekami v njem):

```
# on Johns computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

Na tej točki lahko ostali klonirajo in porivajo spremembe nazaj prav tako enostavno:

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

S to metodo lahko hitro dobite bralno/pisalni strežnik Git delujoč in poganjan za peščico razvijalcev.

Opomniti je potrebno, da trenutno se lahko vsi ti uporabniki lahko tudi prijavijo v strežnik in dobijo lupino kot uporabnik `git`. Če želite to omejiti, boste morali spremeniti lupino na nekaj drugega v datoteki `passwd`.

Enostavno lahko omejite uporabnika `git`, da dela samo aktivnosti Git z omejenim orodjem lupine imenovanim `git-shell`, ki prihaja z Git-om. Če ste to nastavili kot vašo `git` uporabniško prijavno lupino, potem uporabnik `git` ne more imeti običajnega dostopa lupine na vaš strežnik. Da to uporabite, določite `git-shell` namesto `bash` ali `csch` za vašo lopino uporabnikove prijave. Da to naredite, morate najprej dodati `git-shell` v `/etc/shells`, če še ni tam:

```
$ cat /etc/shells # see if `git-shell` is already in there. If not...
$ which git-shell # make sure git-shell is installed on your system.
$ sudo vim /etc/shells # and add the path to git-shell from last command
```

Sedaj lahko urejate lupino za uporabnika z uporabo `chsh <username>`:

```
$ sudo chsh git # and enter the path to git-shell, usually: /usr/bin/git-shell
```

Sedaj lahko uporabnik `git` uporablja samo povezave SSH, da poriva in potegne repozitorije Git in se ne more prijaviti preko lupine v napravo. Če poskusite, boste videli zavrnitev prijave sledeče:

```
$ ssh git@gitserver
fatal: Interactive git shell is not enabled.
hint: ~/git-shell-commands should exist and have read and execute access.
Connection to gitserver closed.
```

Sedaj bodo omrežni ukazi Git še vedno delovali popolnoma v redu, vendar uporabniki ne bodo morali dobiti lupine. Kot trdi izpis, lahko tudi nastavite direktorij v `git` uporabnikov domači direktorij, ki malenkost prilagaja ukaz `git-shell`. Na primer, lahko omejite ukaze Git, ki jih bo strežnik sprejemal ali pa lahko prilagodite sporočilo, ki ga uporabniki vidijo, če se tako poskusijo prijaviti preko SSH. Poženite `git help shell` za več informacij o prilagoditvi lupine.

Prikriti proces Git

Naslednje bomo nastavili prikriti proces (daemon), ki servira repozitorije preko “Git” protokola. To je pogosta izbira za hitro, neoverjen dostop do vaših Git podatkov. Pomnite, da ker ni overjena storitev, karkoli ponudite preko tega protokola je javno znotraj njegovega omrežja.

Če poganjate to na strežniku zunaj vašega požarnega zidu, bi moralo biti uporabljeno samo za projekte, ki so javno vidni svetu. Če je strežnik na katerem poganjate, znotraj vašega požarnega zidu, ga mogoče uporabljate za projekte, katerih veliko število ljudi ali računalnikov (stalna integracija ali zgrajeni strežniki) imajo samo bralni dostop, ko ne želite dodati ključa SSH za vsakega.

V kateremkoli primeru je protokol Git relativno enostavno nastaviti. V osnovi morate pognati ta ukaz v načinu prikritega procesa:

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr` omogoča strežniku ponovni zagon brez čakanja, da stare povezave pretečejo, opcija `--base-path` omogoča ljudem klonirati projekte brez določanja celotne poti in pot na koncu pove Git prikritemu procesu, da poišče repozitorije za izvoz. Če poganjate požarni zid, boste morali narediti vanj tudi luknjo na portu 9418 na napravi, kjer to nastavljate.

Ta proces lahko prikrijete na število načinov, odvisno od operacijskega sistema, na katerem poganjate. Na Ubuntu napravi lahko uporabite skripto Upstart. Torej v sledeči datoteki

```
/etc/event.d/local-git-daemon
```

you put this script:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
  --user=git --group=git \
  --reuseaddr \
  --base-path=/opt/git/ \
  /opt/git/
respawn
```

Zaradi varnostnih razlogov je močno priporočljivo, da se ta prikriti proces poganja kot uporabnik s samo bralnimi pravicami repozitorijev - to lahko naredite enostavno z ustvarjanjem novega uporabnika *git-ro* in pogonom prikritega procesa kot le-tega. Zaradi enostavnosti, ga bomo enostavno pognali kot istega *git* uporabnika, s katerim se poganja *git-shell*.

Ko poženete vašo napravo, se vaš Git prikriti proces začne avtomatično in ponovno odzove, če pade. Da se ga ponovno zažene brez potrebnega ponovnega zagova, lahko poženete to:

```
initctl start local-git-daemon
```

Na drugih sistemih, boste morda želeli uporabiti *xinetd*, skripto v vašem sistemu *sysvinit* ali nekaj drugega - dokler dobite tisti ukaz kot prikriti proces in nekako pregledovan.

Naslednje morate povedati Git-u, katerim repozitorijem se dovoljuje neoverjen Git strežniško osnovani dostop. To lahko naredite v vsakem repozitoriju z izdelavo datoteke poimenovane `'git-daemon-export-ok'`.

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

Prisotnost te datoteke pove Git-u, da je v redu ponuditi ta projekt brez overjanja.

Pametni HTTP

Sedaj imamo overjen dostop preko SSH in neoverjen dostop preko *git://*, vendar obstaja tudi protokol, ki lahko naredi oboje istočasno. Nastavitev pametnega HTTP je v osnovi samo omogočenje skripte CGI, ki je ponujena z Git-om imenovanim *git-http-backend* na strežniku. Ta CGI bo prebral pot in glave poslane, ki jih pošlje *git fetch* ali *git push* k HTTP UTL in določa, če klient lahko komunicira preko HTTP (kar je res za kateregakoli klienta od verzije 1.6.6). Če CGI vidi, da je klient pameten, bo z njim komuniciral pametno, drugače se bo vrnil k neumnemu obnašanju (tako, da je združljiv za nazaj za branje z starejšimi klienti).

Pojdimo skozi zelo osnovno nastavitev. To bomo nastavili z Apache-jem kot strežnikom CGI. Če nimate nastavitve Apache, lahko to naredite na Linux napravi nekako takole:

```
$ sudo apt-get install apache2 apache2-utils
$ a2enmod cgi alias env
```

To tudi omogoči module `mod_cgi`, `mod_alias` in `mod_env`, ki so potrebni, da to ustrezno deluje.

Naslednje moramo dodati nekaj stvari k Apache nastavitvi, da poganja `git-http-backend` kot handler za karkoli, kar pride v pot `/git` vašega spletnega strežnika.

```
SetEnv GIT_PROJECT_ROOT /opt/git
SetEnv GIT_HTTP_EXPORT_ALL
ScriptAlias /git/ /usr/lib/git-core/git-http-backend/
```

Če izpustite `GIT_HTTP_EXPORT_ALL` spremenljivko okolja, potem bo Git serviral samo neoverjenim klientom repozitorije z `git-daemon-export-ok` datoteko v njih, tako kot to dela Git prikriti proces (daemon).

Potem boste morali povedati Apache-ju, da omogoča zahteve k tej poti z nečim takim:

```
<Directory "/usr/lib/git-core*">
  Options ExecCGI Indexes
  Order allow,deny
  Allow from all
  Require all granted
</Directory>
```

Končno boste želeli narediti pisanja, da so nekako overjena, možno z Auth blokom takole:

```
<LocationMatch "^/git/.*/git-receive-pack$">
  AuthType Basic
  AuthName "Git Access"
  AuthUserFile /opt/git/.htpasswd
  Require valid-user
</LocationMatch>
```

To bo zahtevalo, da ustvarite `.htaccess` datoteko, ki vsebuje gesla za vse veljavne uporabnike. Tu je primer dodajanja uporabnika "schacon" datoteki:

```
$ htdigest -c /opt/git/.htpasswd "Git Access" schacon
```

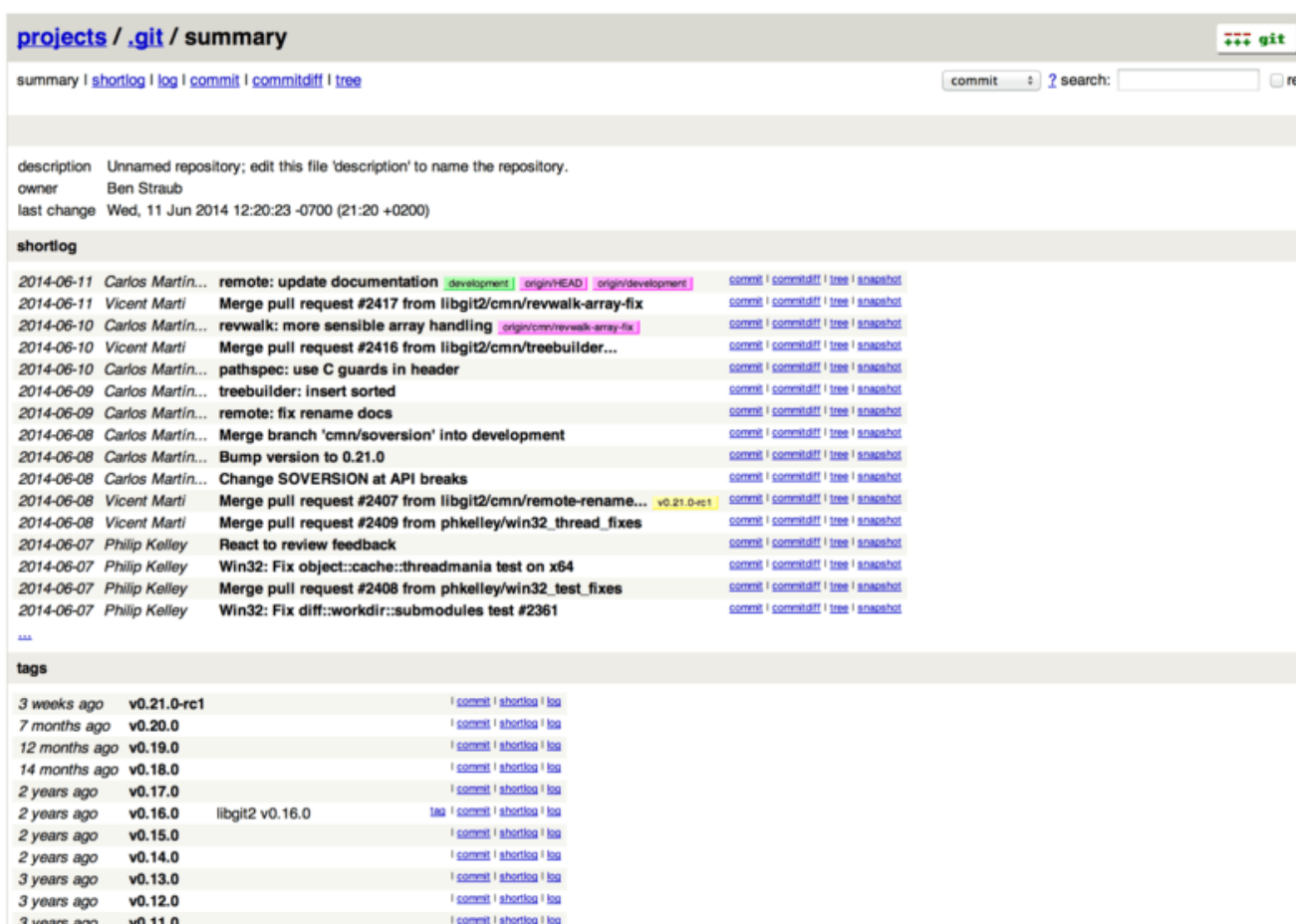
Na voljo je tona načinov imetja Apache overjenih uporabnikov, izbrati in implementirati boste morali enega od njih. To je samo enostaven primer, ki smo ga lahko prikazali. Skoraj zagotovo boste želeli to nastaviti tudi preko SSL, da so vsi podatki šifrirani.

Ne želimo iti predaleč po zajčevi luknji Apache nastavitvenih specifik, ker lahko boste uporabljali drugačni strežnik ali imeli različne overitvene potrebe. Ideja je, da Git prihaja s CGI-jem imenovanim `git-http-backend`, ki ko je sklican bo naredil vsa pogajanja, da pošiljajo in prejemaajo podatke preko HTTP. Ne implementira kakršnegakoli overjanja sam po sebi, vendar to je lahko enostavno kontrolirano na nivoju spletnega strežnika, ki je sklican. To lahko naredite s skoraj katerimkoli CGI-zmožnim spletnim strežnikom, torej pojdite s tistim, ki ga najbolj poznate.

NOTE Za več informacij o nastavitvah overjanja v Apache-ju, preverite dokumentacijo Apache tu: <http://httpd.apache.org/docs/current/howto/auth.html>

GitWeb

Sedaj, ko imate osnovno bralno/pisalni in samo-bralni dostop do vašega projekta, boste morda želeli nastaviti ensotaven spletno osnovani vizualizator. Git prihaja s skripto CGI imenovano GitWeb, ki je včasih uporabljena za to.



The screenshot shows the GitWeb interface for a repository. At the top, there's a navigation bar with 'projects / .git / summary' and a search box. Below that, there's a description of the repository: 'Unnamed repository; edit this file 'description' to name the repository.' The owner is 'Ben Straub' and the last change was on 'Wed, 11 Jun 2014 12:20:23 -0700 (21:20 +0200)'. The main part of the page is the 'shortlog' section, which lists recent commits with their dates, authors, and titles. For example, '2014-06-11 Carlos Martin... remote: update documentation' and '2014-06-11 Vicent Marti Merge pull request #2417 from libgit2/cmn/revwalk-array-fix'. Below the shortlog, there's a 'tags' section listing various versions like 'v0.21.0-rc1', 'v0.20.0', 'v0.19.0', etc., with links to their respective commit pages.

Figure 49. The GitWeb web-based user interface.

Če želite preveriti, kako bi GitWeb izgledal za vaš projekt, Git prihaja z ukazom za pogon začasne instance, če imate lahki strežnik na vašem sistemu, kot je `lighttpd` ali `webrick`. Na Linux napravah, `lighttpd` je pogostokrat nameščen, tako da ga lahko morda dobite, da poženete `git instaweb` v vašem projektne direktoriju. Če poganjate Mac, prihaja Leopard prenameščen z Ruby-jem, tako da `webrick` je lahko vaša najboljša stava. Da poženete `instaweb` z ne-lighttpd handler-jem, ga lahko poženete z opcijo `--httpd`.

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO WEBrick 1.3.1
[2009-02-21 10:02:21] INFO ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

Da poženete strežnik HTTPD na portu 1234 in nato avtomatično poženete spletni brskalnik, da se odpre na tej strani. Je precej enostavno na vašem koncu. Ko končate in želite zapreti strežnik, lahko poženete enak ukaz z opcijo **--stop**:

```
$ git instaweb --httpd=webrick --stop
```

Če želite pognati spletni vmesnik na strežniku ves čas za vso vašo ekipo ali za odprtokodni projekt, ki ga gostujete, boste morali nastaviti skripto CGI, ki je servirana na vašem običajnem spletnem strežniku. Nekatere Linux distribucije imajo paket **gitweb**, ki ga morda lahko namestite preko **apt** ali **yum**, tako da morda želite poskusiti to najprej. Šli bomo skozi namestitev GitWeb ročno zelo hitro. Najprej morate dobiti Git izvorno kodo, s katero GitWeb prihaja in generirati skripto CGI po meri:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" prefix=/usr gitweb
  SUBDIR gitweb
  SUBDIR ../
make[2]: `GIT-VERSION-FILE' is up to date.
  GEN gitweb.cgi
  GEN static/gitweb.js
$ sudo cp -Rf gitweb /var/www/
```

Bodite pozorni, ker morate povedati ukazu, kje najti vaše Git repozitorije s spremenljivko **GITWEB_PROJECTROOT**. Sedaj morate narediti Apache, da uporablja CGI za to skripto za katero lahko dodate VirtualHost:

```
<VirtualHost *:80>
  ServerName gitserver
  DocumentRoot /var/www/gitweb
  <Directory /var/www/gitweb>
    Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
    AllowOverride All
    order allow,deny
    Allow from all
    AddHandler cgi-script cgi
    DirectoryIndex gitweb.cgi
  </Directory>
</VirtualHost>
```

Ponovno, GitWeb je lahko serviran s katerimkoli CGI ali Perl zmožnim spletnim

strežnikom; če raje uporabljate kaj drugega, ga ne bi smelo biti pretežno nastaviti. Na tej točki, bi morali videti <http://gitserver/> za ogled vaših repozitorijev na spletu.

GitLab

GitWeb je precej enostaven konec koncev. Če iščete bolj moderni, strežnik Git s polno lastnostmi, je na voljo nekaj odprtokodnih rešitev, ki jih lahko namestite namesto tega. Ker je GitLab eden bolj popularnih, bomo šli skozi namestitev in uporabo kot primer. To je nekoliko bolj kompleksno kot opcija GitWeb in verjetno zahteva več vzdrževanja, vendar je veliko bolj opcija z več lastnostmi.

Namestitev

GitLab je spletna aplikacija podprta s podatkovno bazo, torej njegova namestitev vključuje nekoliko več vključevanja kot ostali strežniki git. Na srečo je ta proces zelo dobro dokumentiran in podprt.

Na voljo je nekaj metod, ki jih lahko uporabite za namestitev GitLab-a. Da postavite nekaj na hitro, lahko prenesete sliko virtualne naprave ali namestitveni program enega klika iz <https://bitnami.com/stack/gitlab> in prilagodite nastavitve, da se ujemajo z vašim določenim okoljem. Lep pristop, ki ga je Bitnami vključil, je prijavní zaslon (dostopen z vtipkanjem alt-→); pove vam naslov IP in privzeto uporabniško ime ter geslo za nameščeni GitLab.



```
BitNami

*** Welcome to the BitNami Gitlab Stack ***
*** Built using Ubuntu 12.04 - Kernel 3.2.0-53-virtual (tty2). ***

*** You can access the application at http://10.0.1.17 ***
*** The default username and password is 'user@example.com' and 'bitnami1'. ***
*** Please refer to http://wiki.bitnami.com/Virtual_Machines for details. ***

linux login: _
```

Figure 50. The Bitnami GitLab virtual machine login screen.

Za karkoli drugega, sledite vodiču v datoteki readme GitLab izdaje skupnosti, ki se jo lahko najde na <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. Tam boste našli podporo za namestitev GitLab-a z uporabo Chef receptov, virtualno napravo na Digital Ocean in RPM ter DEB pakete (ki so v času tega pisanja v beta fazi). Na voljo je tudi “neuradni”

vodič, kako dobiti GitLab, ki se poganja z ne-standardnimi operacijskimi sistemi ter podatkovnimi bazami, v celoti ročno namestitveno skripto in mnogo ostalih tem.

Administracija

GitLab-ov administracijski vmesnik je dostopen preko spleta. Enostavno pokažite vaš brskalnik k imenu gostitelja ali naslovu IP, kjer je GitLab nameščen in se prijavite kot uporabnik admin. Privzeto uporabniško ime je `admin@local.host` in privzeto geslo je `5ivel!fe` (za katerega vas bo vprašalo, da spremenite kakor hitro ga vnesete). Ko se enkrat prijavite, kliknite na ikono “Admin area” v meniju zgoraj desno.

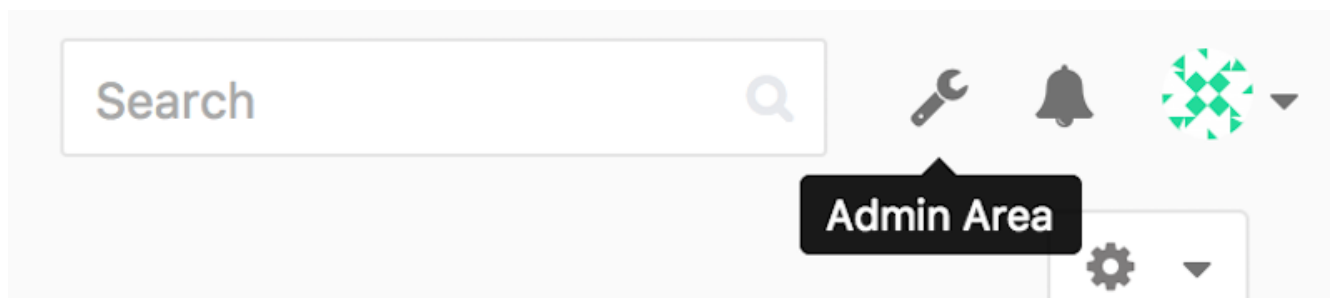


Figure 51. The “Admin area” item in the GitLab menu.

Uporabniki

Uporabniki v GitLab-u so računi, ki se ujemajo z ljudmi. Uporabniški računi nimajo veliko kompleksnosti; v celoti je zbirka osebnih informacij, pripetih k prijavnim podatkom. Vsak uporabniški račun prihaja z **imenskim prostorom**, kar je logično grupiranje projektov, ki pripadajo temu uporabniku. Če ima uporabnik `jane` projekt imenovan `projekt`, je url projekta `http://server/jane/projekt`.

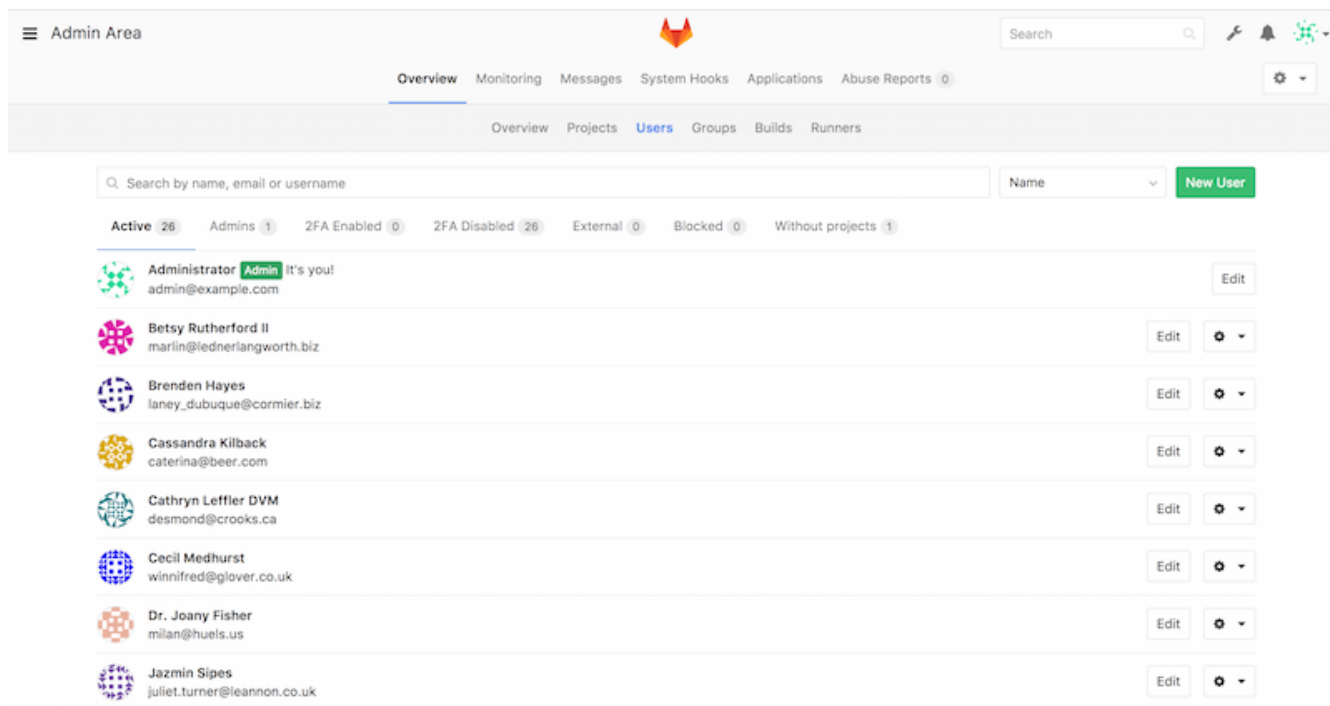


Figure 52. The GitLab user administration screen.

Odstranjevanje uporabnika je lahko narejeno na dva načina. “Blokiranje” uporabnika mu

prepreči prijavo v instanco GitLab, vendar vsi podatki pod uporabniškim imenskim prostorom so shranjeni in pošiljanja poslana s tem uporabniškim e-naslovom bodo še vedno povezovala njegov profil.

“Uničenje” uporabnika ga po drugi strani v celoti odstrani iz podatkovne baze in datotečnega sistema. Vsi projekti in podatki v njihovem imenskem prostoru so odstranjeni in katerekoli skupine, ki jih imajo v lasti bodo tudi odstranjene. To je očitno veliko bolj dokončna in destruktivna akcija in njene uporabe so redke.

Skupine

GitLab skupina je montaža projektov skupaj s podatki o tem, kako lahko uporabniki dostopajo do teh projektov. Vsaka skupina ima imenski prostor projekta (enak način, kakor to delajo uporabniki), torej če ima skupina `training` projekt `materials`, bi bil njen url `http://server/training/materials`.

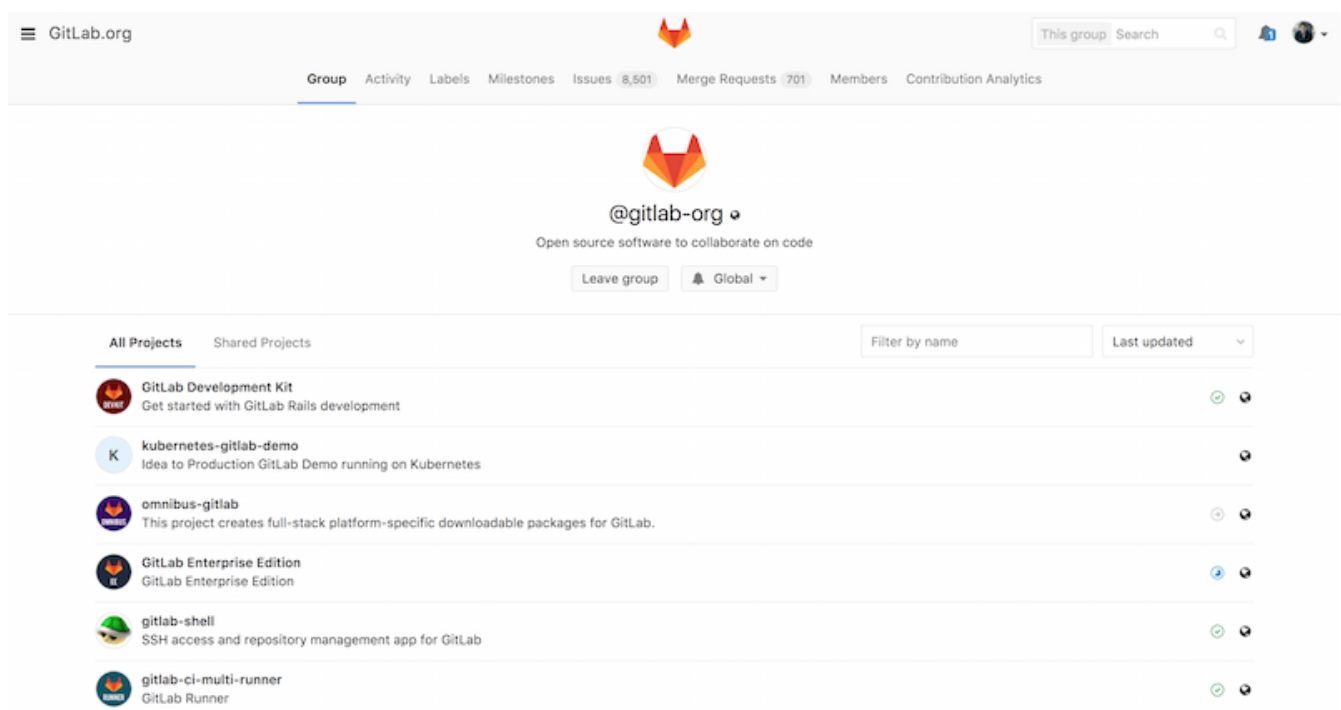


Figure 53. The GitLab group administration screen.

Vsaka skupina je povezana s številom uporabnikov, vsak od njih ima nivo pravic za skupinske projekte in samo skupino. To so območja od “Guest” (težave in samo pogovor) do “Owner” (polna kontrola skupina, njenih članov in njenih projektov). Tipi pravic so preveč številni za izpis tu, vendar GitLab je pomoči polna povezava na administracijskem zaslonu.

Projekti

GitLab projekt približno ustreza enemu repozitoriju git. Vsak projekt pripada enemu imenskemu prostoru ali uporabnika ali skupine. Če projekt pripada uporabniku, ima lastnik projekta direktno kontrolo nad tem, kdo ima dostop do projekta; če projekt pripada skupini, potem bodo pravice skupinskega nivoja uporabnika tudi imele učinek.

Vsak projekt ima tudi nivo vidnosti, kar kontrolira, kdo ima dostop branja do strani in

repozitorijev tega projekta. Če je projekt *Private*, mora lastnik projekta eksplicitno odobriti dostop določenem uporabniku. *Interni* projekt je viden katerikoli prijavljenem uporabniku in *javni* projekt je viden komurkoli. Bodite pozorni, da to krmili tako git “fetch” dostop kot tudi dostop do spletnega uporabniškega vmesnika za ta projekt.

Ključke

GitLab vključuje podporo za ključke tako na projektnem ali sistemskem nivoju. Za katerikoli od teh, bo GitLab strežnik izvedel HTTP POST z nekim opisnim JSON, kadarkoli se zgodi relevanten dogodek. To je odličen način za povezavo vaših repozitorijev git in instance GitLab z vašo preostalo avtomatizacijo, kot je CI strežnik, pogovorne sobe ali nalagalna orodja.

Osnovna uporaba

Prva stvar, ki jo boste želeli narediti z GitLab-om je ustvariti nov projekt. To se doseže s klikom na ikono “+” na orodni vrstici. Vprašani boste za ime projekta, kateremu imenskemu prostoru naj bi pripadal in kakšen nivo vidnosti bi moral imeti. Večino česar določite tu ni dokončno in se lahko ponovno prilagodi kasneje skozi vmesnik nastavitvev. Kliknite na “Create Project” in ste končali.

Enkrat, ko projekt obstaja, ga boste verjetno želeli povezati z lokalnim Git repozitorijem. Vsak projekt je dostopen preko HTTPS ali SSH, katerikoli od teh je lahko uporabljen za nastavitvev daljave Git. URL-ji so vidni na vrhu domače strani projekta. Za obstoječe lokalne repozitorije ta ukaz ustvari daljavo imenovano `gitlab` h gostovani lokaciji:

```
$ git remote add gitlab https://server/namespace/project.git
```

Če nimate lokalne kopije repozitorija, lahko enostavno naredite sledeče:

```
$ git clone https://server/namespace/project.git
```

Spletni UI ponuja dostop do mnogih uporabnik pogledov samega repozitorija. Vsaka domača stran projekta prikazuje zadnjo aktivnost in povezave na vrhu vas bodo peljale do pogledov projektnih datotek in dnevnika pošiljanja.

Delo skupaj

Najenostavnejši način dela skupaj na GitLab projektu je dajanje drugemu uporabniku direktne pravice porivanja v repozitorij git. Lahko dodate uporabnika projektu, da greste v sekcijo “Members” nastavitvev tega projekta in povežete novega uporabnika z nivojem dostopa (različni nivoji dostopa so predebatirani nekoliko v [Skupine](#)). Z dajanjem uporabniku nivo dostopa “Developer” ali več, ta uporabnik lahko potiska pošiljanja in veje direktno v repozitorij brez kaznovanja.

Drug bolj razvezan način sodelovanja je z uporabo zahtevkov združevanja. Ta lastnost

omogoča kateremukoli uporabniku, da vidit projekt za sodelovanje na kontroliran način. Uporabniki z direktnim dostopom lahko enostavno ustvarijo vejo, vanj pošljejo in odprejo zahtevek združevanja iz njihove veje nazaj v `master` ali katerokoli drugo vejo. Uporabniki, ki nimajo pravic porivanja za repozitorij lahko naredijo t.i. "fork" (ustvarijo njihovo lastno kopijo), pošljejo v to kopijo, odprejo zahtevek združevanja iz njihovega fork-a nazaj v glavni projekt. Ta model omogoča lastniku, da ima polno kontrolo nad tem, kaj gre v repozitorij in kdaj, medtem ko omogoča sodelovanje z nezaupanimi uporabniki.

Zahtevki združevanja in težave so glavna enota dolgo-živečih diskusij v GitLab-u. Vsak zahtevek združevanja omogoča debato tipa vrstica-za-vrstico predlagane spremembe (ki podpira lahek način pregleda kode), kot tudi splošno skupno nit diskusije. Oba sta lahko določena uporabnikom ali organizirana v mejnike.

Ta sekcija je osredotočena v glavnem na Git-povezane lastnosti GitLab-a, vendar kot zrel projekt ponuja mnoge ostale lastnosti, ki pomagajo vaši ekipi, da dela skupaj, kot so projektni wikiji in orodja vzdrževanja sistema. Ena korist GitLab-a je ta, da ko je enkrat strežnik nastavljen in v pogonu, boste redko morali prilagoditi nastavitveno datoteko ali dostop do strežnika preko SSH; večina administracije in splošne uporabe je lahko dosežena preko vmesnika znotraj-brskalnika.

Tretje osebne opcije gostovanja

Če ne želite iti skozi celotno delo, ki je vključeno v nastavljanju vašega lastnega Git strežnika, imate nekaj opcij za gostovanje vaših Git projektov na zunanji namenski gostovani strani. Tako delo ponuja število prednosti: gostovana stran je v splošnem hitreje za nastaviti in enostavnejše za začeti projekte in nobeno vzdrževanje strežnikov ali spremljanje je vključeno. Tudi če nastavite in poženete vaš lastni strežnik interno, lahko še vedno uporabite javno gostovano stran za vašo odprto kodo - je v splošnem enostavnejše za odprtokodno skupnost, da to najde in vam pri tem pomaga.

Te dni imate veliko število opcij gostovanja iz katerih izbirate, vsaka z različnimi prednostmi in slabostmi. Da vidite posodobljen seznam, preverite GitHosting stran na glavnem Git wiki-ju na <https://git.wiki.kernel.org/index.php/GitHosting>

Pokrili bomo uporabo GitHub-a v podrobnosti v [GitHub](#), saj je največji gostitelj Git-a na voljo in morda boste morali imeti interakcijo s projekti gostovanimi na njem v kateremkoli primeru, vendar je še ducat ostalih za izbiro za katere vam ni treba nastavljati lastnega strežnika Git.

Povzetek

Imate nekaj opcij, da dobite oddaljeni Git repozitorij v delovanje, tako da lahko sodelujete z ostalimi ali delite vaše delo.

Poganjanje vašega lastnega strežnika vam da veliko kontrole in vam omogoča, da poženete strežnik znotraj vašega lastnega požarnega zidu, vendar tak strežnik v splošnem zahteva precej vašega časa, da se ga nastavi in vzdržuje. Če date vaše podatke na gostovani strežnik, ga je enostavno nastaviti in vzdrževati; vendar morate biti zmožni

imeti kodo na strežniku nekoga drugega in nekatere organizacije tega ne omogočajo.

Moralo bi biti precej enostavno določiti, katera rešitev ali kombinacija rešitev je primerna za vas in vašo organizacijo.

Distribuirani Git

Sedaj, ko imate oddaljeni repozitorij Git nastavljen kot točko za vse razvijalce, da delijo svojo kodo in so vam poznani osnovni ukazi Git v lokalnem poteku dela, boste pogledali, kako uporabiti nekaj distribuiranih potekov dela, ki vam jih ponuja Git.

V tem poglavju, boste videli, kako delati z Git-om v distribuiranem okolju kot sodelavec in povezovalc. To pomeni, da se boste naučili, kako prispevati projektu kodo uspešno in narediti kakor enostavno za vas in vzdrževalca projekta je možno in tudi kako uspešno vzdrževati projekt s številnimi razvijalci, ki prispevajo.

Distribuirani poteki dela

Z razliko od centraliziranih sistemov za kontrolo verzija (CVCS-ji), vam narava distribucije Git-a omogoča, da ste veliko bolj fleksibilni v tem, kako razvijalci sodelujejo na projektih. V centraliziranih sistemih je vsak razvijalec vozlišče, ki bolj ali manj dela enako na centralnem središču. Vendar v Git-u vsak razvijalec je potencialno oboje vozlišče in središče - to je, vsak razvijalec lahko tako prispeva kodo k drugim repozitorijem in vzdržuje javen repozitorij na katerem ostali lahko osnujejo svoje delo in h katerem lahko prispevajo. To odpira širok spekter zmožnosti poteka dela za vaš projekt in/ali vašo ekipo, torej bomo pokrili nekaj pogostih paradig, ki izkoriščajo to fleksibilnost. Šli bomo skozi prednosti in možne slabosti vsakega modela; lahko izberete kateregakoli za uporabo ali mešate in ujamete lastnosti iz vsakega.

Centraliziran potek dela

V centraliziranih sistemih je v splošnem en model sodelovanja - centralizirani potek dela. Centralno središče ali repozitorij lahko sprejema kodo in vsakdo sinhronizira svoje delo nanj. Število razvijalcev so vozlišča - uporabniki tega središča - in sinhronizirajo na to lokacijo.

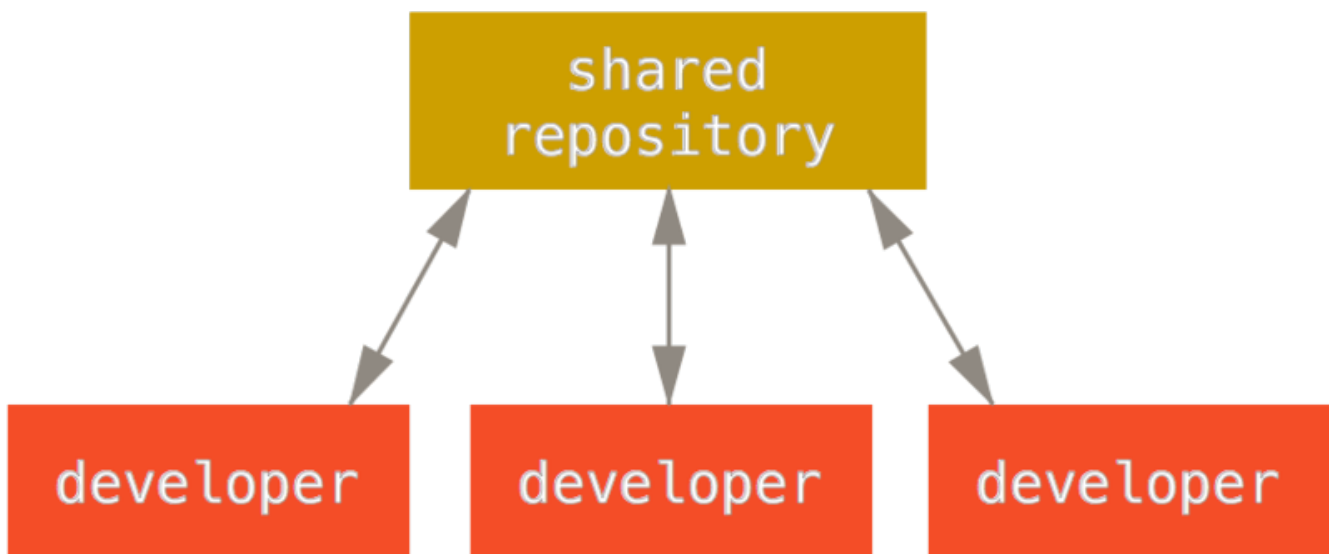


Figure 54. Centralized workflow.

To pomeni, da če dva razvijalca klonirata iz središča in oba naredita spremembe, prvi

razvijalec, ki porine svoje spremembe nazaj gor, lahko to naredi brez problemov. Drugi razvijalec mora združiti delo prvega preden poriva spremembe gor, da ne prepíše sprememb prvega razvijalca. Ta koncept je tako resničen v Git-u kot tudi v Subversion-u (ali kateremkoli drugem CVCS-ju) in ta model deluje odlično v Git-u.

Če ste že udobni s centraliziranim potekom dela v vašem podjetju ali ekipi, lahko enostavno nadaljujete z uporabo tega poteka dela v Git-u. Enostavno nastavite en repozitorij in dajte vsakomur v vaši ekipi dostop porivanja; Git ne bo dovolil uporabnikov prepisati drug drugega. Recimo, da John in Jessica oba začneta delati istočasno. John konča svoje spremembe in jih porine na strežnik. Nato Jessica poskuša poriniti njene spremembe, vendar jih strežnik zavrne. Povedano ji je, da poskuša poriniti t.i. non-fast-forward spremembe in da tega ne bo mogla narediti dokler ne ujame in združi. Ta potek dela je atraktiven za veliko ljudi, ker je paradigma, s katero so mnogi seznanjeni in udobni.

To tudi ni omejeno na majhne ekipe. Z Git modelom razvejanja, je možno za stotine razvijalcev, da uspešno delajo na enem projektu skozi ducat vej simultano.

Potek dela povezovalnega upravitelja

Ker vam Git omogoča imeti več oddaljenih repozitorijev, je možno imeti potek dela, kjer ima vsak razvijalec pisalni dostop do svojega lastnega javnega repozitorija in bralni dostop do vseh ostalih. Ta scenarij pogosto vključuje kanonični repozitorij, ki predstavlja “uradni” projekt. Da prispevate temu projektu, ustvarite vaš lastni javni klon projekta in porinete vaše spremembe vanj. Nato lahko pošljete zahtevek vzdrževalcu glavnega projekta, da potegne vaše spremembe. Vzdrževalec lahko nato doda vaš repozitorij kot daljavo, testira vaše spremembe lokalno, jih združi v svojo vejo in porine nazaj v svoj repozitorij. Proces deluje sledeče (glejte [Integration-manager workflow](#)):

1. Vzdrževalec projekta porine v svoj javni repozitorij.
2. Prispevalec klonira ta repozitorij in naredi spremembe.
3. Prispevalec porine v svojo lastno kopijo.
4. Prispevalec pošlje vzdrževalcu e-pošto, kjer ga prosi, da potegne spremembe.
5. Vzdrževalec doda repozitorij prispevalca kot daljavo in združi lokalno.
6. Vzdrževalec porine zdržene spremembe v glavni repozitorij.

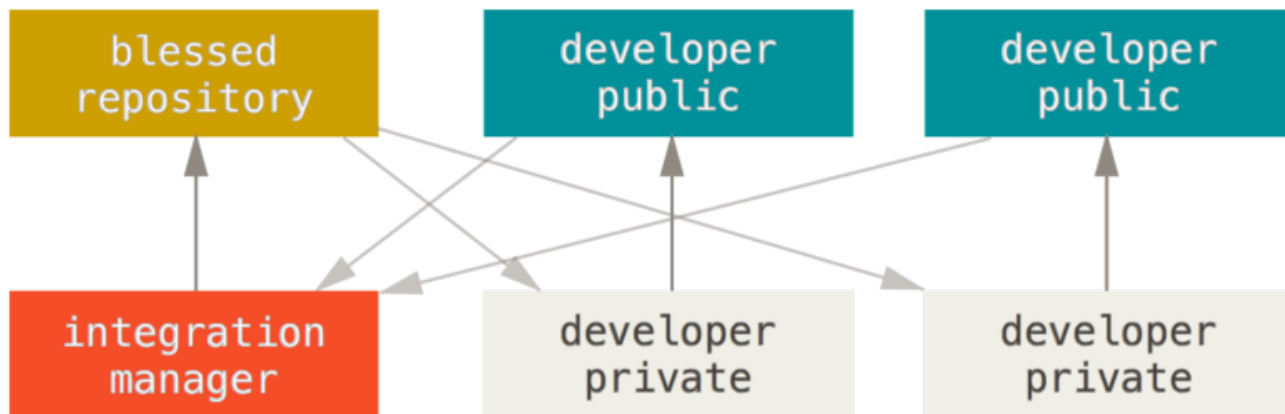


Figure 55. Integration-manager workflow.

To je zelo pogost potek dela z orodji s središčno-osnovo kot sta GitHub ali GitLab, kjer je enostavno "forkati" projekt in poriniti vaše spremembe v vaš fork, da jih vsakdo vidi. Ena glavnih prednosti tega pristopa je, da lahko nadaljujete delo in vzdrževalec glavnega repozitorija lahko potegne vaše spremembe kadarkoli. Prispevalcem ni treba čakati, da projekt vključi njihove spremembe - vsaka stran lahko dela po svojem lastnem ritmu.

Potek dela diktator in poročniki

To je različica poteka dela večih repozitorijev. V splošnem je uporabljen na velikih projektih s stotinami sodelavcev; eden znanih primerov je jedro Linux. Različni upravitelji integracij so zadolženi za določene dele repozitorija; imenujejo se poročniki. Vsi poročniki imajo enega upravitelja integracije znanega kot dobrohotni diktator. Repozitorij dobrohotnega diktatorja služi kot referenčni repozitorij iz katerega morajo vsi sodelavci potegniti. Proces deluje takole (glejte [Benevolent dictator workflow](#)):

1. Splošni razvijalci delajo na njihovih tematskih vejah in ponovno bazirajo svoje delo na glede na `master`. Veja `master` je veja diktatorja.
2. Poročniki združijo razvijalčeve tematske veje v svojo vejo `master`.
3. Diktator združi poročnikove veje `master` v diktatorjevo vejo `master`.
4. Diktator porine svojo `master` v referenčni repozitorij, da ostali razvijalci lahko ponovno bazirajo na njem.

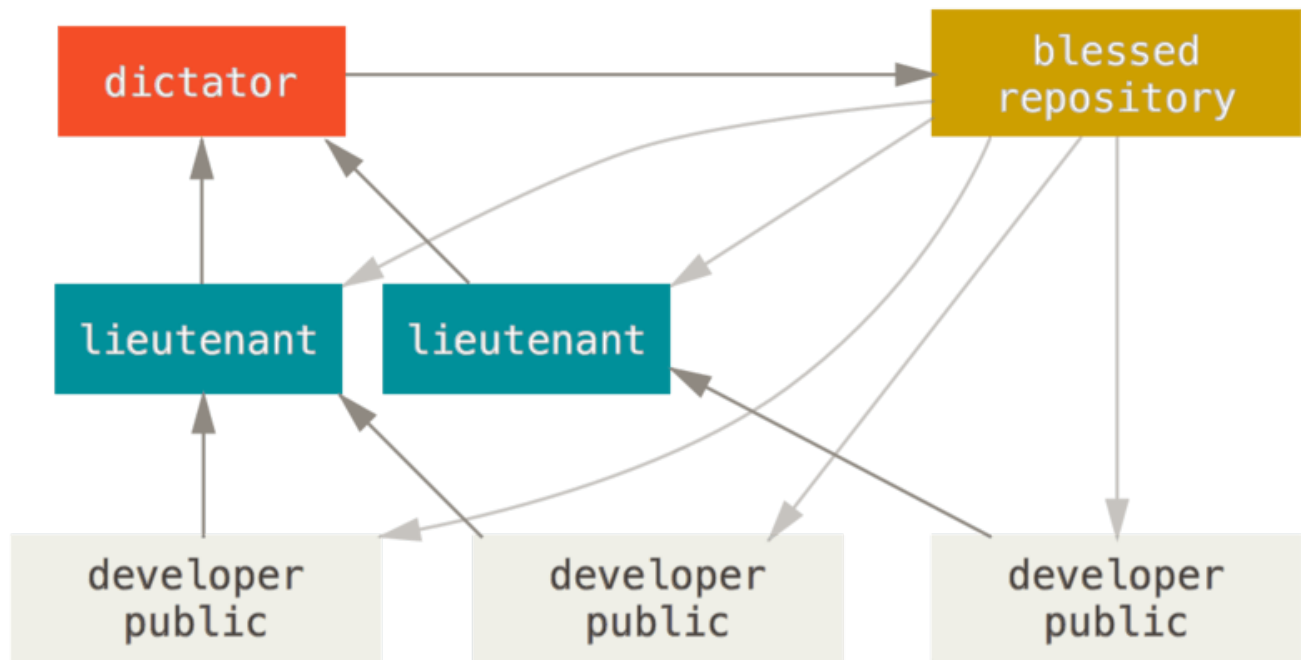


Figure 56. Benevolent dictator workflow.

Taka vrsta poteka dela ni pogosta, vendar je lahko uporabna v zelo velikih projektih ali v visoko hierarhičnih okoljih. Vodji projekta (diktatorju) omogoča delegirati večino dela in zbrati velike skupke koda na večih točkah preden jih integrira

Povzetek poteka dela

To so nekatere pogosto uporabljeni poteki dela, ki so možni v distribuiranih sistemih kot je Git, vendar lahko vidite, da so možne mnoge različice, ki ustrezajo vašem določenem realnem poteku dela. Sedaj ko lahko (upajmo) določite katera kombinacija poteka dela lahko deluje za vas, bomo pokrili nekaj določenih primerov kako doseči glavne vloge, ki naredijo različne poteke. V naslednji sekciji, se boste naučili o nekaterih pogostih vzorcih prispevanja projektu.

Prispevanje projektu

Glavna težava z opisovanjem, kako prispevati projektu je, da obstaja veliko število spremenljivk, kako je to narejeno. Ker je Git zelo fleksibilen, ljudje lahko in tudi res delajo skupaj na mnoge načine in problematično je opisovati, kako bi morali prispevati - vsak projekt je nekoliko drugačen. Nekatere spremenljivke vključene so število aktivnih ljudi, ki prispevajo, izbrani potek dela, vaš dostop pošiljanja in možno zunanja metoda prispevanja.

Prva spremenljivka je število aktivnih ljudi, ki prispevajo - koliko uporabnikov aktivno prispeva kodo temu projektu in kako pogosto? V mnogih primerih boste imeli dva ali tri razvijalce z nekaj pošiljanj na dan ali možno manj za projekte v nekako mirnem stanju. Za večja podjetja ali projekte bi lahko število razvijalcev bilo v tisočih, s stotinami ali tisoči pošiljanj, ki prihajajo vsak dan. To je pomembno, ker z več in več razvijalci naletite na več težav kako zagotavljati, da vaša koda uporablja čistočo ali je enostavno združljiva. Spremembe, ki jih pošljete lahko postanejo zastarele ali precej polomljene z

delom, ki je bilo združeno medtem ko ste delali ali medtem ko vaše spremembe čakajo na odobritev ali uporabo. Kako lahko obdržite vašo kodo konsistentno posodobljeno in vaša pošiljanja veljavna?

Naslednja spremenljivka je potek dela v uporabi za projekt. Je centralizirano z vsakim razvijalcem, ki ima enak dostop pisanja v glavno linijo kode? Ali ima projekt vzdrževalca ali integracijskega upravitelja, ki preveri vse popravke? So vsi popravki pregledani na stikih in odobreni? Ali ste vključeni v ta proces? Ali je sistem poročnika na mestu in ali jim morate poslati vaše delo prvotno?

Naslednja težava je vaš dostop pošiljanja. Potek dela, ki je zahtevan za prispevanje projektu je veliko bolj drugačen, če imate dostop pisanja k projektu, kot če ga nimate. Če nimate dostopa za pisanje, kako ima projekt raje, da sprejme prispevano delo? Ali ima politiko? Koliko dela prispevate na določen čas? Kako pogosto prispevate?

Vsa ta vprašanja lahko vplivajo, kako učinkovito prispevati projektu in katere poteke dela imate raje ali so na voljo za vas. Pokrili bomo aspekte za vsakega od teh v seriji primerov uporabe in se premaknili od enostavnega do bolj kompleksnega; morali bi biti sposobni skonstruirati določen potek dela, ki ga potrebujete v praksi iz teh primerov.

Smernice pošiljanja

Preden pričnemo gledati določen primer uporabe, je tu hitro obvestilo o sporočilih pošiljanja. Imeti dobre smernice za ustvarjanje pošiljanj in se jih držati naredi delo z Git-om in sodelovanjem z ostalimi veliko enostavnejše. Projekt Git ponuja dokument, ki začrta število dobrih nasvetov za ustvarjanje pošiljanj iz katerih se pošlje popravke - to lahko preberete v izvorni kodi Git v datoteki [Documentation/SubmittingPatches](#).

Najprej ne želite poslati kakršnih koli napak praznih znakov. Git ponuja enostaven način, da to preverite - preden pošljete, poženite `git diff --check`, ki identificira vse možne napake praznih znakov in jih izpiše za vas.

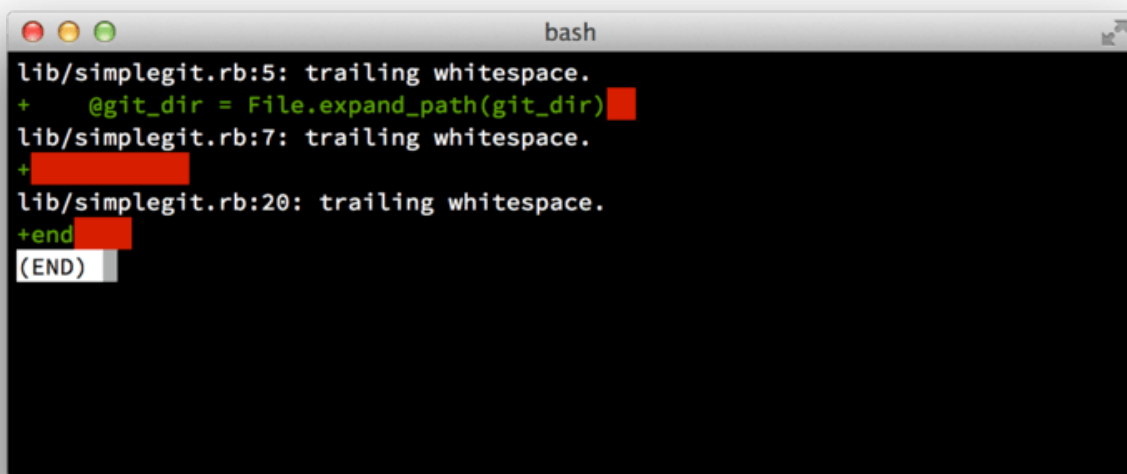
A screenshot of a terminal window titled 'bash'. The terminal shows the output of the command 'git diff --check'. The output consists of several lines of text: 'lib/simplegit.rb:5: trailing whitespace.', '+ @git_dir = File.expand_path(git_dir)', 'lib/simplegit.rb:7: trailing whitespace.', '+ [redacted]', 'lib/simplegit.rb:20: trailing whitespace.', '+end', and '(END)'. The redacted lines are represented by solid black boxes. The terminal background is black, and the text is white.

Figure 57. Output of `git diff --check`.

Če poženete ta ukaz preden pošljete, lahko poveste, če ste tik preden, da pošljete težave s praznimi znaki, ki lahko nagajajo ostalim razvijalcem.

Naslednej poskusite narediti vsako pošiljanje logično ločen skupek sprememb. Če lahko, poskusite narediti vaše spremembe prebavljive - ne kodirajte cel vikend na petih različnih težavah in nato pošljite vse kot eno masovno pošiljanje v ponedeljek. Tudi če ne pošljete med vikendom, uporabite vmesno fazo v ponedeljek, da se loči vaše delo v vsaj eno pošiljanje na težavo z uporabnim sporočilom na pošiljanje. Če nekatere spremembe spremenijo isto datoteko, poskusite uporabiti `git add --patch` za delno vmesne datoteke (pokrito v podrobnostih v [Interactive Staging](#)). Posnetek projekta pri nasvetu veje je identičen, če naredite eno pošiljanje ali pet, dokler so spremembe dodane na neki točki, torej poskusite narediti stvari enostavnejše za vaše kolege sodelavce, ko bodo morali pregledati vaše spremembe. Ta pristop tudi naredi enostavnejše potegniti ali povrniti eno izmed skupka sprememb, če to kasneje potrebujete. [Rewriting History](#) opisuje število uporabnih trikov Git za prepisovanje zgodovine in interaktivno dajanje datotek v vmesno fazo - uporabite ta orodja, da pomagajo izdelati čisto in razumljivo zgodovino preden pošljete delo nekemu drugemu.

Zadnja stvar za pomniti je sporočilo pošiljanja. Navaditi se ustvarjati kvalitetna sporočila pošiljanj naredi uporabo in sodelovanje z Git-om veliko enostavnejše. Kot splošno pravilo, bi se vaša sporočila morala začeti z eno vrstico, ki ni večja od 50 znakov in opisuje skupek sprememb jedrnato ter ji nato sledi prazna vrstica, ki ji sledi bolj podrobna razlaga. Projekt Git zahteva, da bolj podrobna razlaga vključuje vašo motivacijo za spremembe in kontrast njihove implementacije s prejšnjim obnašanjem - to je tudi dobra smernica za slediti. Dobra ideja je tudi uporabiti nujno prisotnos sedanjika v teh sporočilih. Z drugimi besedami, uporabite ukaze. Namesto "I added tests for" ali "Adding tests for," uporabite "Add tests for." Tu je predloga, ki jo je prvotno napisal Tim Pope:

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

Vsa vaša sporočila pošiljanj izgledajo nekako takole, stvari so veliko enostavnejše za vas in razvijalce s katerimi delate. Git projekt ima dobro oblikovana sporočila pošiljanj -

poskusite tam pognati `git log --no-merges`, da vidite, kako izgleda lepo oblikovana zgodovina pošiljanj projekta.

V naslednjem primeru in skozi večino te knjige zaradi kratkosti ta knjiga nima lepo oblikovanih sporočil, kot je ta; namesto tega, uporabljamo opcijo `-m` za `git commit`. Naredite kot pravimo in ne kot kot mi delamo.

Privatna majhna ekipa

Najenostavnejša nastavitvev, na katero boste verjetno naleteli je privatni projekt z enim ali dvema razvijalcema. "Private" v tem kontekstu pomeni zaprto kodo - ni dostopna za zunanji svet. Vi in ostali razvijalci imate vsi dostop potiskanja v repozitorij.

V tem okolju lahko sledite poteku dela, ki je podoben čemur ste morda delali, ko ste uporabljali Subversion ali drug centraliziran sistem. Še vedno dobite prednosti stvari kot so pošiljanje brez povezave in prostrano enostavno razvejanje in združevanje vendar potek dela je lahko zelo podoben; glavna razlika je, da se združevanje zgodi na strani klienta namesto na strežniku v času pošiljanja. Poglejmo, kako lahko izgleda, ko dva razvijalca začneta delati skupaj z deljenim repozitorijem. Prvi razvijalec, John, klonira repozitorij, naredi spremembe in jih pošlje lokalno. (Sporočila protokola so bila zamenjana z ... v teh primerih, da jih nekako skrajšajo.)

```
# John's Machine
$ git clone john@github:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
1 files changed, 1 insertions(+), 1 deletions(-)
```

Drug razvijalec, Jessica, naredi isto stvar - klonira repozitorij in pošlje spremembo:

```
# Jessica's Machine
$ git clone jessica@github:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
1 files changed, 1 insertions(+), 0 deletions(-)
```

Sedaj Jessica potisne njeno delo na strežnik:

```
# Jessica's Machine
$ git push origin master
...
To jessica@github.com:simplegit.git
 1edee6b..fbff5bc master -> master
```

John tudi poskuša potisniti svojo spremembo:

```
# John's Machine
$ git push origin master
To john@github.com:simplegit.git
 ! [rejected]          master -> master (non-fast forward)
error: failed to push some refs to 'john@github.com:simplegit.git'
```

John-u ni dovoljeno potisniti, ker je vmes potisnila Jessica. To je posebej pomembno za razumeti, če ste vajeni Subversion-a, ker boste opazili, da dva razvijalca nista uredila iste datoteke. Čeprav Subversion avtomatično naredi to združevanje na strežniku, če so urejene različne datoteke, morate v Git-u združevati pošiljanja lokalno. John mora ujeti spremembe Jessice in jih združiti preden mu je dovoljeno potiskati:

```
$ git fetch origin
...
From john@github.com:simplegit
 + 049d078...fbff5bc master    -> origin/master
```

Na tej točki John-ov lokalni repozitorij izgleda nekako takole:

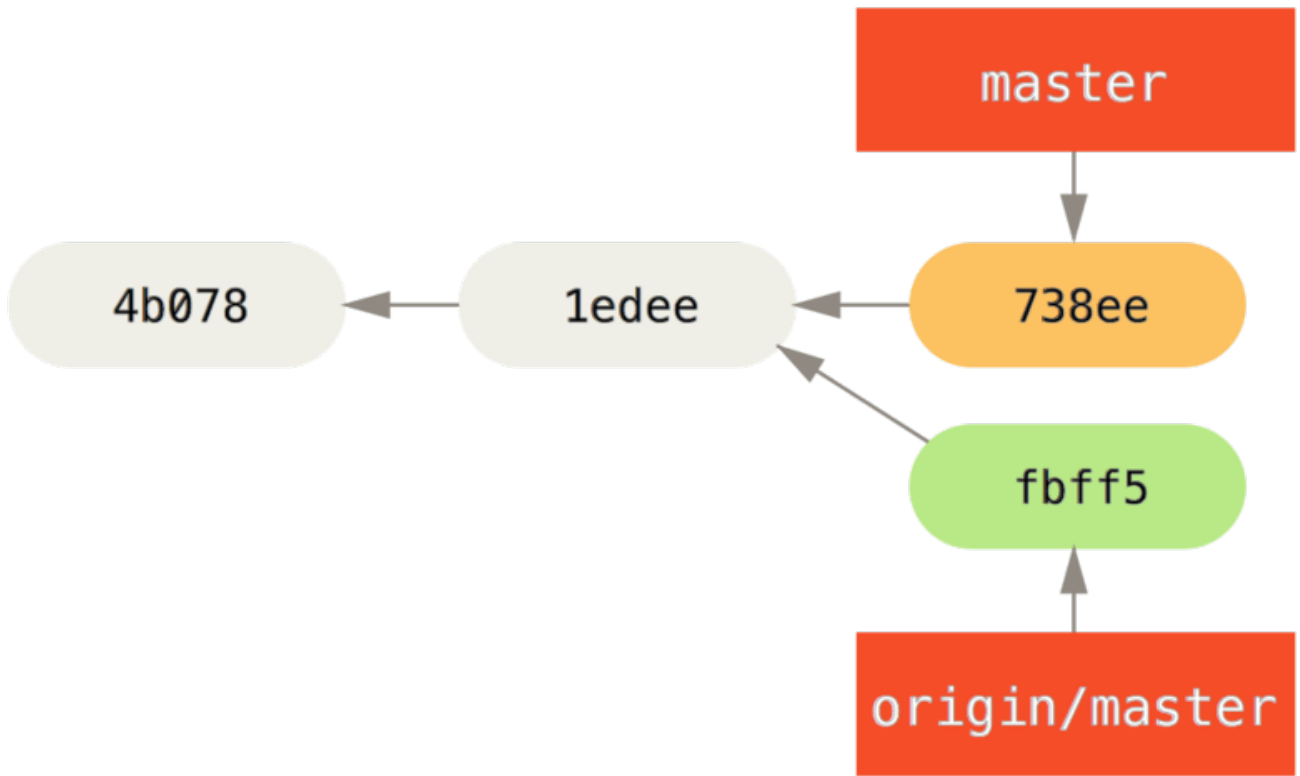


Figure 58. John's divergent history.

John ima referenco na spremembe, ki jih je potisnila Jessica, vendar jih mora združiti v svoje delo preden mu je dovoljeno potisniti:

```

$ git merge origin/master
Merge made by recursive.
TODO | 1 +
1 files changed, 1 insertions(+), 0 deletions(-)
  
```

Združevanje gre gladko - John-ova zgodovina pošiljanja sedaj izgleda nekako takole:

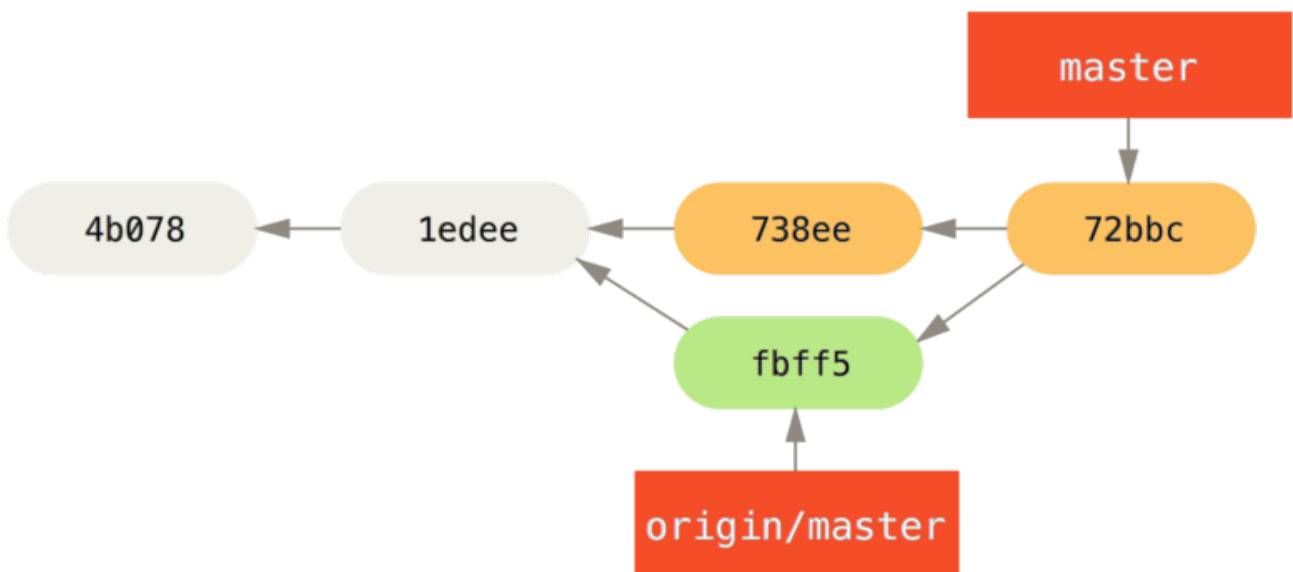


Figure 59. John's repository after merging `origin/master`.

Sedaj lahko John testira svojo kodo, da zagotovi, da še vedno ustrezno deluje in nato lahko potisne svoje novo združeno delo na strežnik:

```
$ git push origin master
...
To john@github.com:simplegit.git
 fbff5bc..72bbc59 master -> master
```

Končno, John-ova zgodovina pošiljanja izgleda nekako takole:

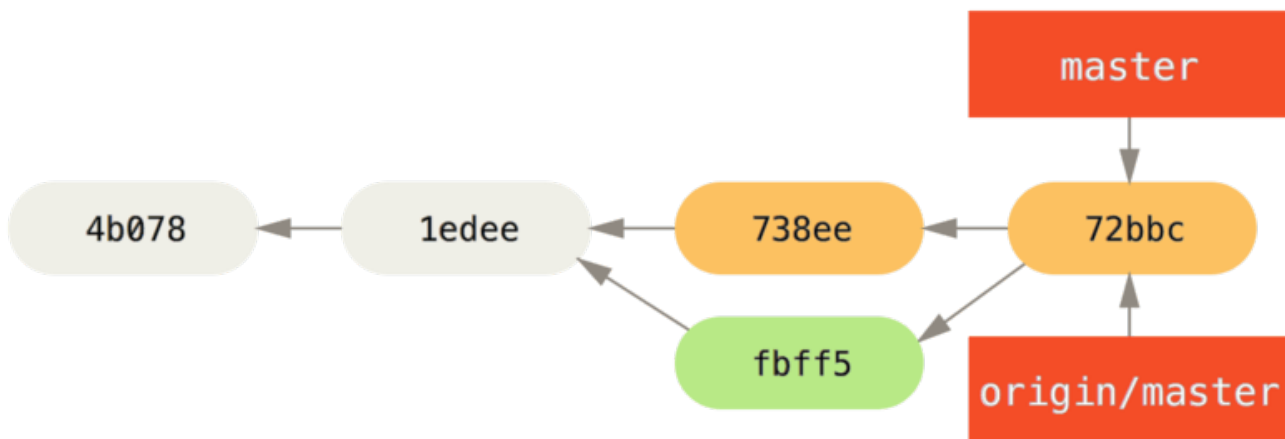


Figure 60. John's history after pushing to the `origin` server.

Vmes je Jessica delala na tematski veji. Ustvarila je tematsko vejo imenovano `issue54` in naredila tri pošiljanja na tej veji. Ni pa še ujela sprememb John-a, zato njena zgodovina pošiljanja izgleda nekako takole:

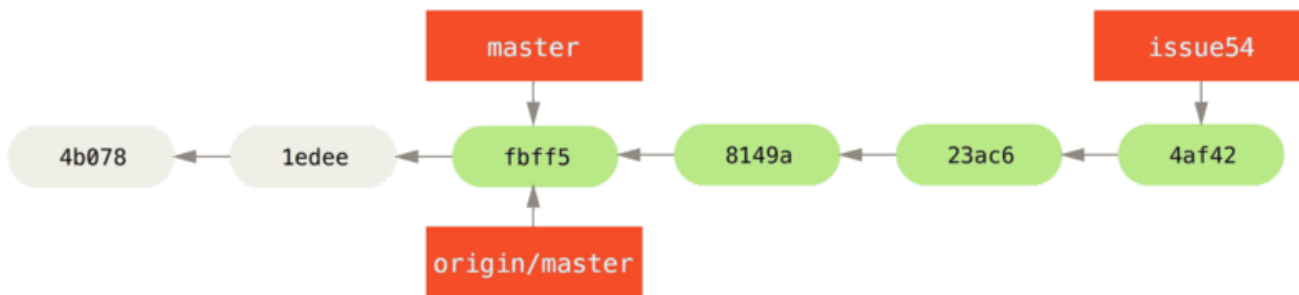


Figure 61. Jessica's topic branch.

Jessica se želi sinhronizirati z John-om, torej ujame:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@github.com:simplegit
 fbff5bc..72bbc59 master -> origin/master
```

To potegne delo, ki ga je vmes John potisnil. Zgodovina Jessice sedaj izgleda takole:

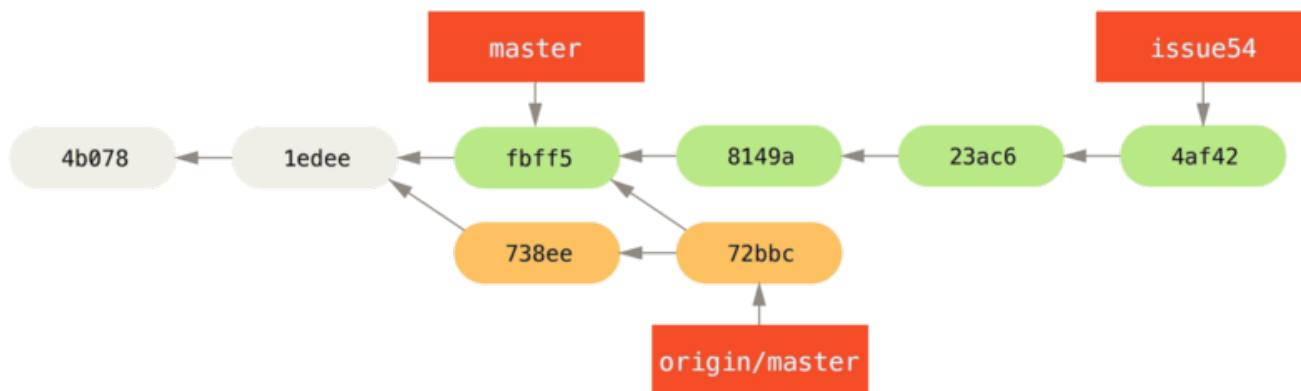


Figure 62. Jessica's history after fetching John's changes.

Jessica misli, da je njena tematska veja pripravljena, vendar želi vedeti, kaj mora združiti v njeno delo, da lahko potisne. Požene `git log`, da ugotovi:

```

$ git log --no-merges issue54..origin/master
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date: Fri May 29 16:01:27 2009 -0700

removed invalid default value
  
```

Sintaksa `issue54..origin/master` je dnevniški filter, ki vpraša Git, da prikaže samko seznam pošiljanj, ki so na kasnejši veji (v tem primeru `origin/master`), ki niso na prvi veji (v tem primeru `issue54`). Skozi to sintakso bomo šli v podrobnosti v [Commit Ranges](#).

Za sedaj, lahko vidimo izpis, da je eno pošiljanje, ki ga je naredil John in ga Jessica ni združila. Če Jessica združi `origin/master`, je to eno pošiljanje, ki bo spremenilo njeno lokalno delo.

Sedaj Jessica lahko združi njeno tematsko delo v njeno lokalno vejo, združi, John-ovo delo (`origin/master`) v njeno vejo `master` in nato potisne nazaj na strežnik. Najprej preklopi nazaj na njeno master vejo, da integrira vso to delo:

```

$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
  
```

Združi lahko najprej bodisi `origin/master` ali `issue54` - obe sta nazvgor vodni, torej vrstni red ni pomemben. Zadnji posnetek bi moral biti identičen ne glede na vrstni red, ki ga izbere, samo zgodovina bo nekoliko drugačna. Najprej izbere združiti `issue54`:

```

$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README          |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)

```

Ne pride do nobenih problemov; kot lahko vidite je šlo za enostaven fast-forward. Sedaj Jessica združi delo John-a (`origin/master`):

```

$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

```

Vse se združi čisto in zgodovina Jessice izgleda takole:

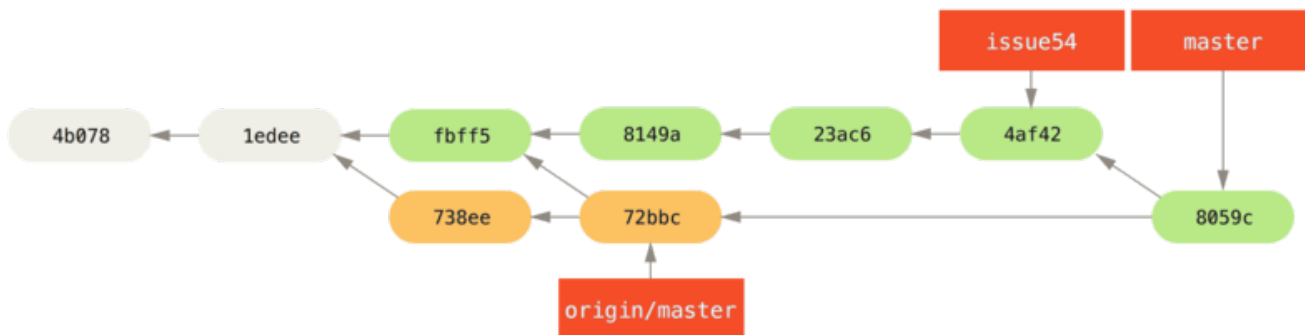


Figure 63. Jessica's history after merging John's changes.

Sedaj je `origin/master` dosegljiv iz Jessicine veje `master`, da lahko uspešno potiska (ob predpostavki, da John vmes ni ponovno potisnil):

```

$ git push origin master
...
To jessica@github:simplegit.git
 72bbc59..8059c15 master -> master

```

Vsak razvijalec je poslal nekajkrat in uspešno združil delo drug drugega.

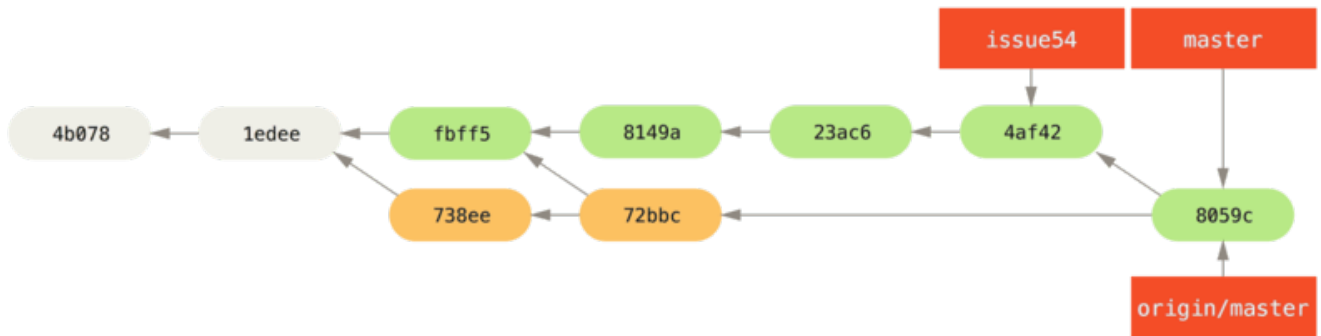


Figure 64. Jessica's history after pushing all changes back to the server.

To je eden najenostavnejših potekov dela. Delate nekaj časa, v splošnem na tematski veji in združite vašo vejo `master`, ko ste pripravljeni za integracijo. Ko želite deliti to delo, ga združite v vašo lastno vejo `master`, nato ujamite in združite `origin/master`, če se je spremenila in končno potisnite na vejo `master` na strežniku. Splošna sekvenca je nekaj takega:

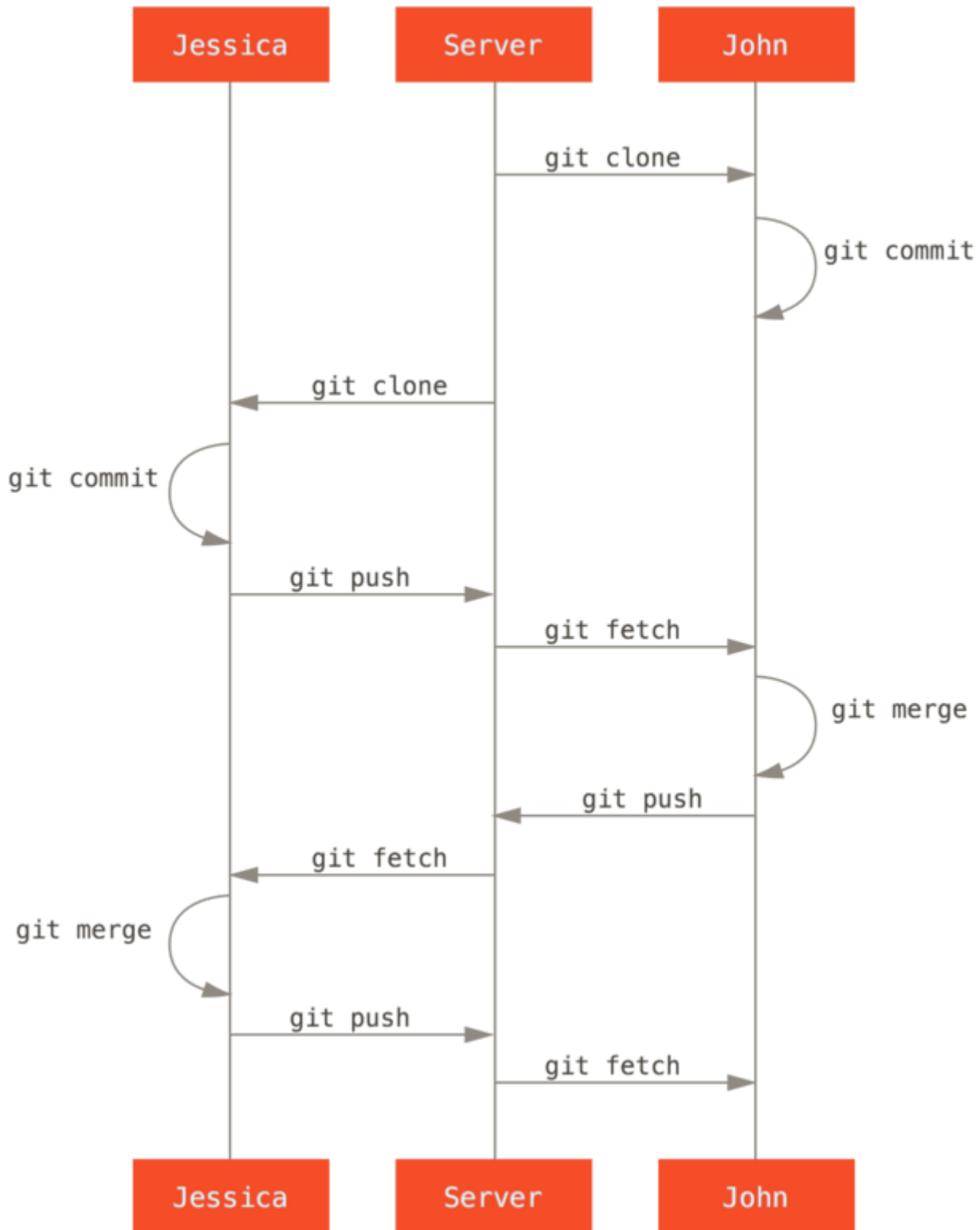


Figure 65. General sequence of events for a simple multiple-developer Git workflow.

Zasebne upravljane ekipe

V tem naslednjem scenariju, boste pogledali vloge prispevalcev v večji zasebni skupini. Naučili se boste, kako delati v okolju, kjer manjše skupine sodelujejo na lastnostih in nato so te prispevki na osnovi ekip integrirani s strani druge strani.

Recimo, da John in Jessica delata skupaj na lastnosti, medtem Jessica in Josie delata na drugi. V tem primeru podjetje uporablja tip poteka dela integracija-upravitelj, kjer je delo

posameznih skupin integrirano samo od določenih inženirjev in veja `master` glavnega repozitorija je lahko posodobljena samo s strani teh inženirjev. V tem scenariju je vso delo narejeno na vejah na osnovi ekipe in potegnjene skupaj s strani integratorjev kasneje.

Sledimo poteku dela Jessice kot dela na njenih dveh lastnostih, sodeluje vzporedno z dvema različnima razvijalcema v tem okolju. Predpostavimo, da že ima njen repozitorij kloniran in se odloči delati najprej na `featureA`. Ustvari novo vejo za lastnost in naredi nekaj dela na njej:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
1 files changed, 1 insertions(+), 1 deletions(-)
```

Na tej točki potrebuje deliti nekaj dela z John-om torej potisne njena pošiljanja veje `featureA` na strežnik. Jessica nima dostopa potiskanja na vejo `master` - samo integratorji imajo - torej mora potisniti na drugo vejo, da lahko sodeluje z John-om:

```
$ git push -u origin featureA
...
To jessica@github:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica sporoči po e-pošti Johnu, da je potisnila nekaj dela v vejo imenovano `featureA` in on lahko to sedaj pogleda. Medtem ko čaka za povratne informacije od John-a, se Jessica odloči začeti delati na `featureB` z Josie. Da začne, prične novo vejo lastnosti, ki je osnovana na strežniški veji `master`:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Sedaj, Jessica naredi nekaj pošiljanj na veji `featureB`:

```

$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
1 files changed, 5 insertions(+), 0 deletions(-)

```

Jessicin repozitorij izgleda takole:

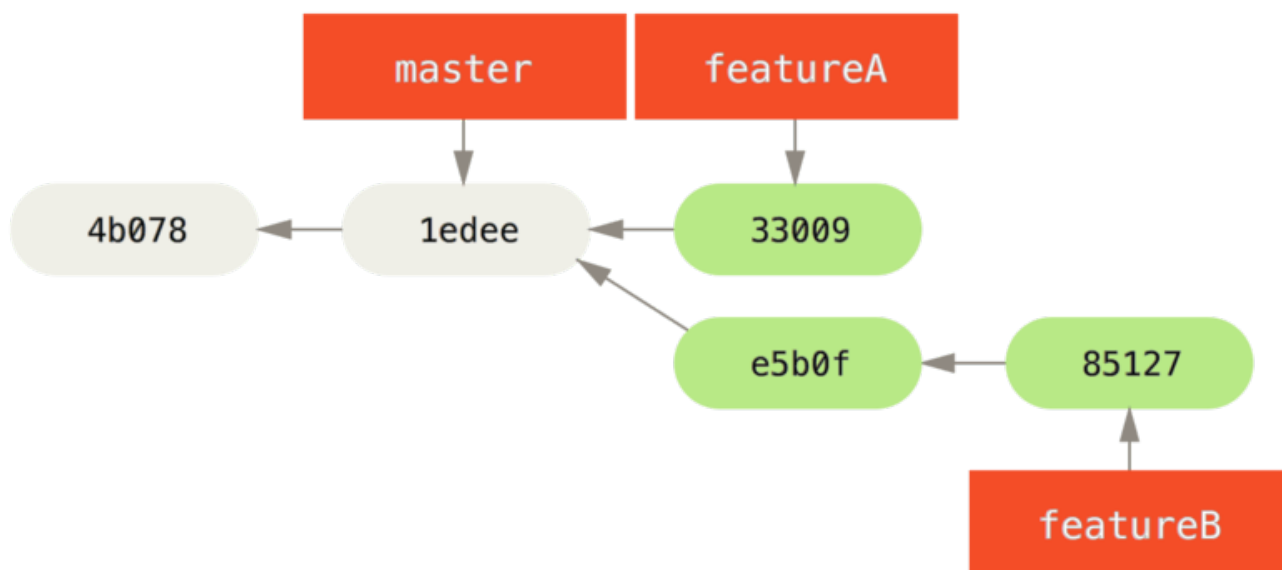


Figure 66. Jessica's initial commit history.

Pripravljena je potisniti njeno delo, vendar dobi e-pošto od Josie, da je veja z nekaj začetnega dela na njej že potisnjena na strežnik kot `featureBee`. Jessica najprej potrebuje združiti te spremembe v njeno lastno preden lahko potiska na strežnik. Nato lahko ujame spremembe Josie z `git fetch`:

```

$ git fetch origin
...
From jessica@github:simplegit
* [new branch]      featureBee -> origin/featureBee

```

Jessica lahko sedaj združi to v delo, ki ga je naredila z `git merge`:

```

$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
lib/simplegit.rb | 4 ++++
1 files changed, 4 insertions(+), 0 deletions(-)

```

Je pa manjši problem - potrebuje potisniti združeno delo v njeno vejo `featureB` v vejo

`featureBee` na strežniku. To lahko naredi z določanjem lokalne veje, ki ji sledi podpičje (:), ki mu sledi oddaljena veja ukazu `git push`:

```
$ git push -u origin featureB:featureBee
...
To jessica@github.com:simplegit.git
   fba9af8..cd685d1 featureB -> featureBee
```

To se imenuje *refspec*. Glejte [The Refspec](#) za bolj podrobno diskusijo respec Git-a in različnih stvari, ki jih lahko naredi z njimi. Opazite tudi zastavico `-u7`; to je kratica za `--set-upstream`, ki nastavi veje za enostavnejšo potiskanje in poteg kasneje.

Naslednje John pošlje e-pošto Jessici in ji pove, da je potisnil neke spremembe v vejo `featureA` in jo vpraša, če jih lahko potrdi. Požene `git fetch`, da potegne te spremembe:

```
$ git fetch origin
...
From jessica@github.com:simplegit
   3300904..aad881d featureA -> origin/featureA
```

Nato lahko vidi, kaj je bilo spremenjeno z `git log`:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

Končno združi delo John-a v njeno lastno vejo `featureA`:

```
$ git checkout featureA
Switched to branch 'featureA'
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb | 10 ++++++++
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica želi nekaj prilagoditi, torej ponovno pošlje in nato potisne to nazaj na strežnik:

```

$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
 3300904..774b3ed featureA -> featureA

```

Zgodovina pošiljanja Jessice sedaj izgleda nekako takole:

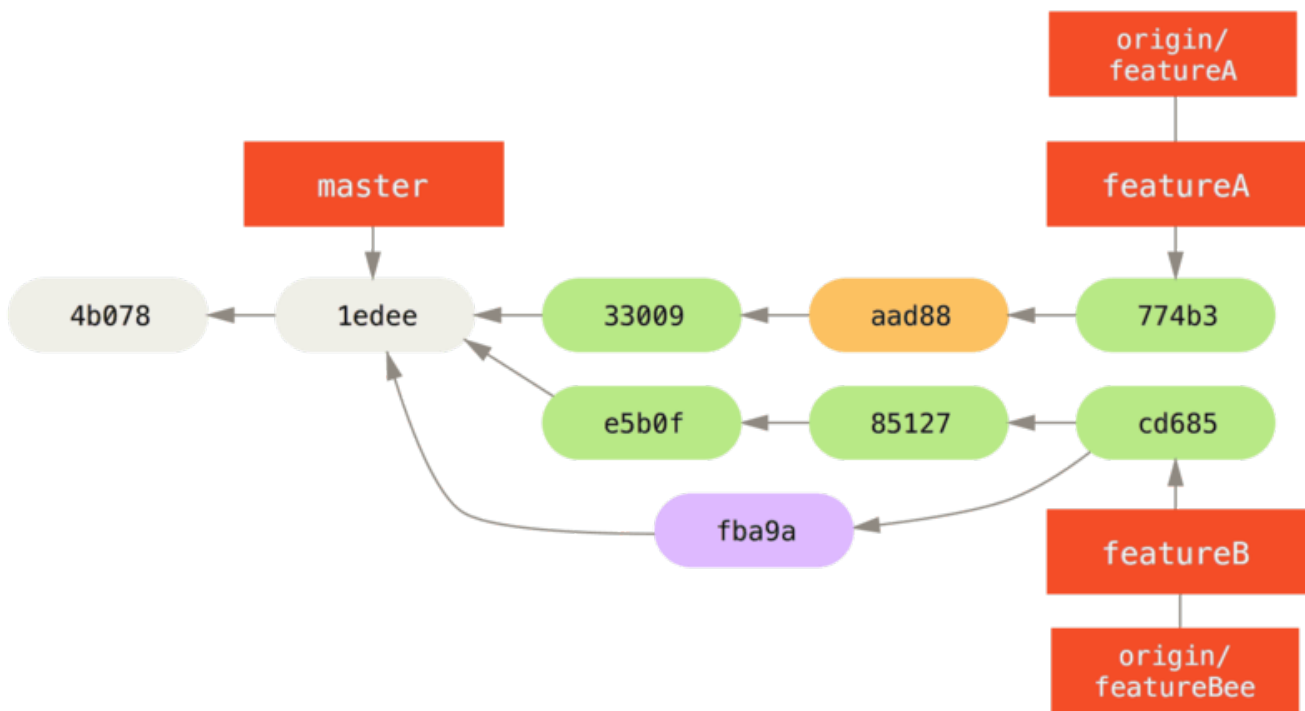


Figure 67. Jessica's history after committing on a feature branch.

Jessica, Josie in John obvestilo integratorje, da sta veji **featureA** in **featureBee** na strežniku pripravljeni za integracijo v glavno linijo. Ko integratorji združijo te veje v glavno linijo, bo ujetje preneslo novo pošiljanje združevanja in naredilo zgodovino, da izgleda nekako takole:

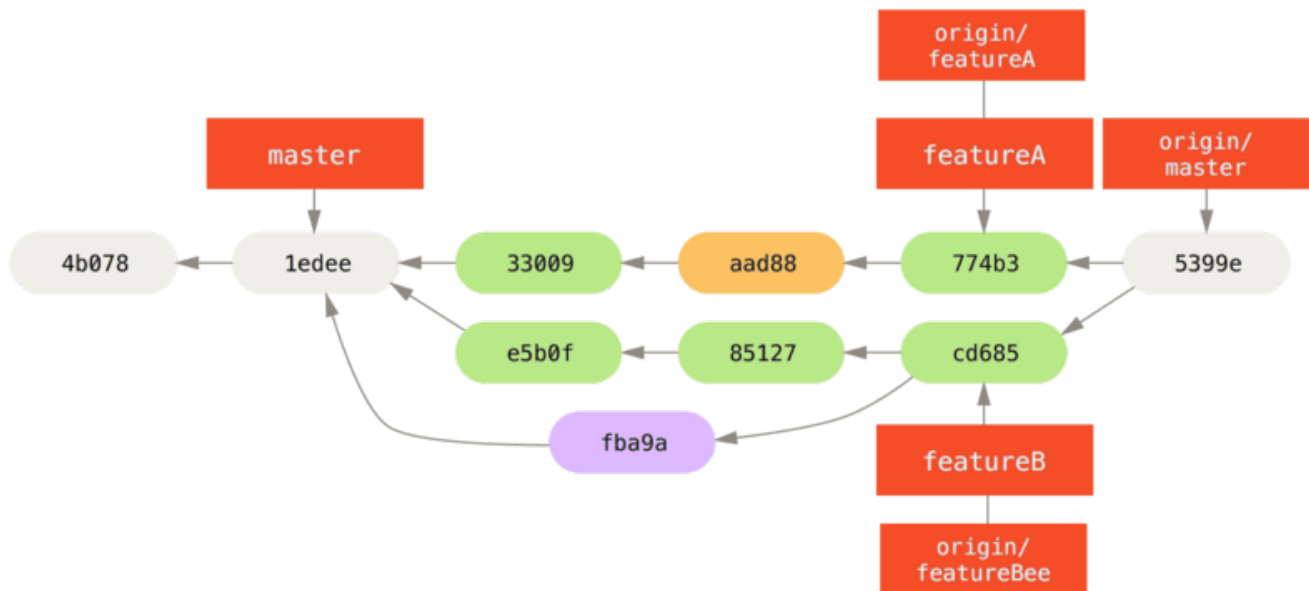


Figure 68. Jessica's history after merging both her topic branches.

Mnoge skupine preklopijo na Git zaradi te zmožnosti in imajo več ekip, ki delajo vzporedno, združujejo na različnih vrsticah dela kasneje v procesu. Zmožnost manjših podskupin ekipe, da sodelujejo preko oddaljenih vej brez potrebe po vključevanju ali oviri celotne ekipe je velika prednost Git-a. Sekvenca poteka dela, ki ste ga videli je nekaj takega:

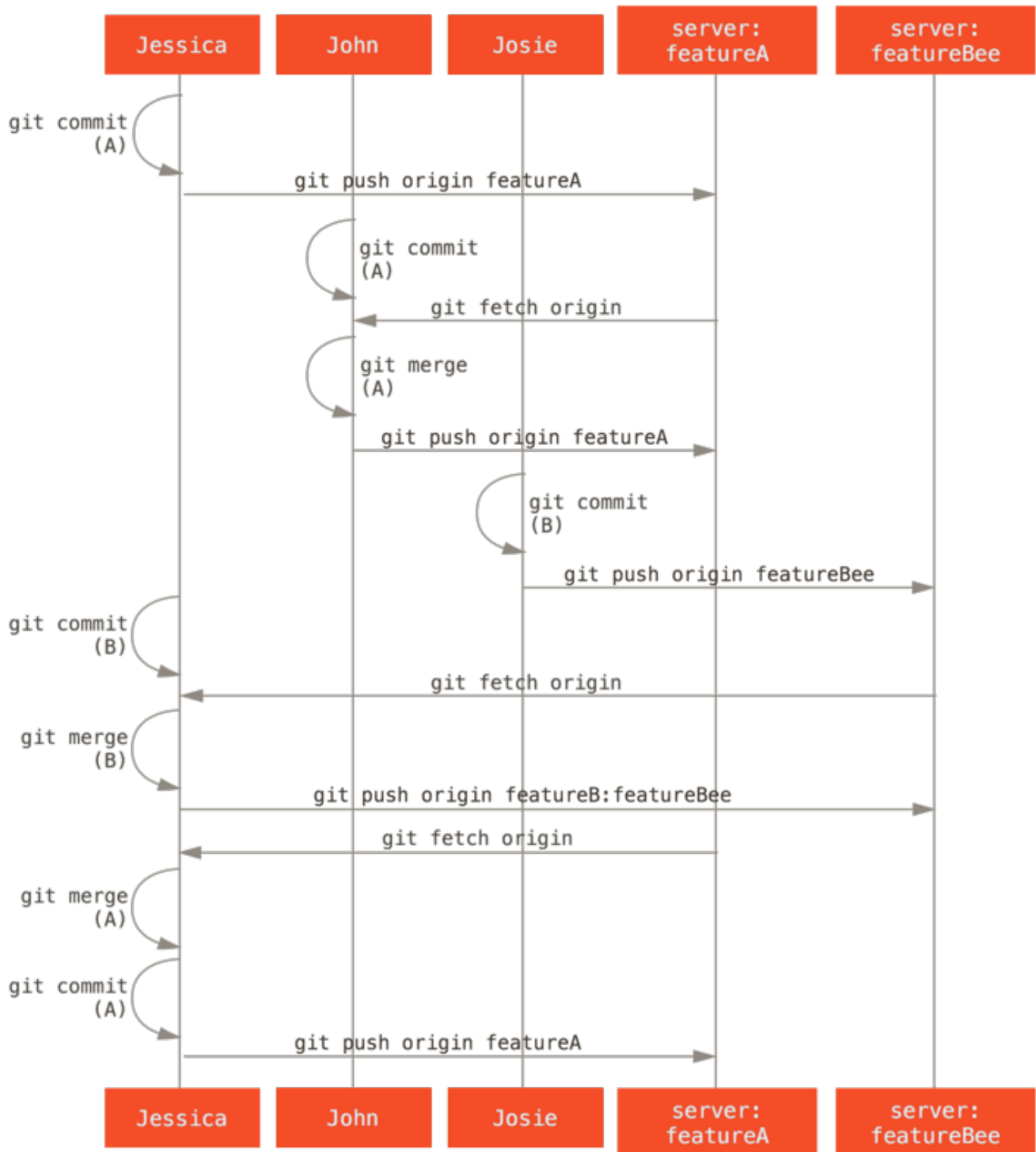


Figure 69. Basic sequence of this managed-team workflow.

Forkan javni projekt

Prispevanje javnim projektom je nekoliko drugačno. Ker nimate dovoljenja direktno posodobiti veje na projektu, morate nekako dati delo vzdrževalcem na nek drug način. Prvi primer opisuje prispevanje preko forkanja na Git gostiteljih, ki podpirajo enostavno forkanje. Mnogi strani gostiteljev to podpirajo (vključno GitHub, BitBucket, Google Code, repo.or.cz in ostali) in mnogi vzdrževalci projektov pričakujejo ta stil prispevanja. Naslednja sekcija se ukvarja s projekti, ki imajo raje sprejeti prispevane popravke preko e-pošte.

Najprej boste verjetno želeli klonirati glavni repozitorij, ustvariti tematsko vejo za

popravek ali serijo popravkov, ki jih planirate prispevati in narediti delo tam. Sekvenca izgleda v osnovi takole:

```
$ git clone (url)
$ cd project
$ git checkout -b featureA
# (work)
$ git commit
# (work)
$ git commit
```

NOTE

Lahko boste želeli uporabiti `rebase -i`, da vaše delo stisnete v eno pošiljanje ali delo preuredite v pošiljanja, da naredite popravek enostavnejši za pregled razvijalcev - glejte [Rewriting History](#) za več informacij o interaktivnem ponovnem baziranju.

Ko je delo vaše veje končano in ste pripravljeni prispevati nazaj vzdrževalcem, pojdite na prvotno stran projekta in kliknite na gumb “Fork”, kar ustvari vaš lasten zapisljiv fork projekta. Nato morate dodati ta novi URL repozitorija kot drugo daljavo v tem primeru imenovano `myfork`:

```
$ git remote add myfork (url)
```

Nato morate potisniti vaše delo gor. Najenostavnejše je potisniti tematsko vejo, na kateri delate na vaš strežnik namesto združevanja v vašo vejo master in potiskanje tega navzgor. Razlog je, da če delo ni sprejeto ali je izbrano (cherry pick), vam ni treba previti nazaj vaše veje master. Če vzdrževalci združijo, ponovno bazirajo ali izberejo vaše delo, ga boste eventuelno kakorkoli dobili nazaj iz njihovega repozitorija:

```
$ git push -u myfork featureA
```

Ko je bilo vaše delo potisnjeno gor na vaš fork, morate obvestiti vzdrževalca. To je pogostokrat imenovano zahtevek potega in ga lahko ali generirate preko spletne strani - GitHub ima svoj lastni mehanizem zahtevkov potega, ki jih bomo obravnavali v [GitHub](#) - ali lahko pošete ukaz `git request-pull` in izpišete e-pošto vzdrževalcu projekta ročno.

Ukaz `request-pull` vzame osnovno vejo v katero želite potegniti vašo tematsko vejo in URL repozitorija Git, iz katerega želite potegniti ter izpisati povzetek vseh sprememb za katere sprašujete, da se jih potegne. Na primer, če Jessica želi poslati John-u zahtevek potega in je končala dve pošiljanji na tematski veji, ki jo je ravnokar potisnila, lahko požene tole:

```

$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
    added a new function

are available in the git repository at:

  git://githost/simplegit.git featureA

Jessica Smith (2):
  add limit to log function
  change log output to 30 from 25

lib/simplegit.rb | 10 ++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)

```

Izpis se lahko pošlje vzdrževalcu - pove jim, iz kje je delo razvejano, povzame pošiljanja in pove, od kje potegniti to delo.

Na projektu za katerega niste vzdrževalec, je v splošnem enostavnejše imeti vejo kot je `master`, ki vedno sledi `origin/master` in narediti vaše delo v tematskih vejah, ki jih lahko enostavno zavržete, če so zavrjnene. Imeti delovne teme izolirane v tematske veje, tudi naredi enostavnejše za vas, da ponovno bazirate vaše delo, če se je konica glavnega repozitorija vmes premaknila in vaša pošiljanja niso več uporabljena čisto. Na primer, če želite poslati drugo temo dela projekta, ne nadaljujte z delom na tematski veji, samo potisnite - pričnite znova iz veje glavnega repozitorija `master`:

```

$ git checkout -b featureB origin/master
# (work)
$ git commit
$ git push myfork featureB
# (email maintainer)
$ git fetch origin

```

Za sedaj vsaka od vaših tem je vsebovana znotraj silosa - podobno kot čakalna vrsta popravka - ki jo lahko prepisete in spremenite brez tem, ki se vmešavajo ali so soodvisne druga od druge, takole:

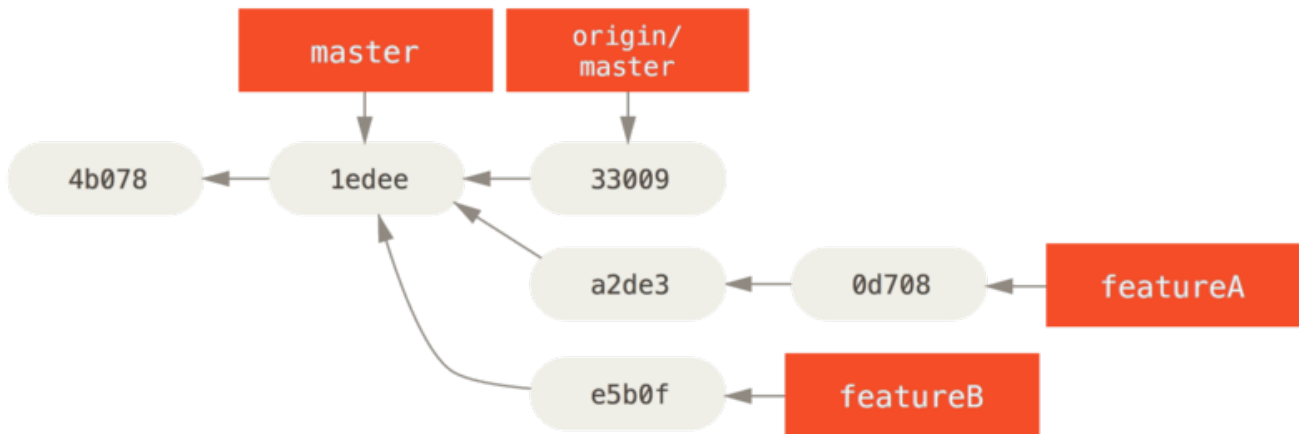


Figure 70. Initial commit history with `featureB` work.

Recimo, da je vzdrževalec projekta povlekel veliko ostalih popravkov in poskusil vašo prvo vejo, vendar ne združuje več čisto. V tem primeru, lahko poskusite ponovno bazirati to vejo na vrh `origin/master`, rešite konflikte za vzdrževalca in nato ponovno pošljete vaše spremembe:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

To prepíše vašo zgodovino, da sedaj izgleda kot [Commit history after featureA work](#).

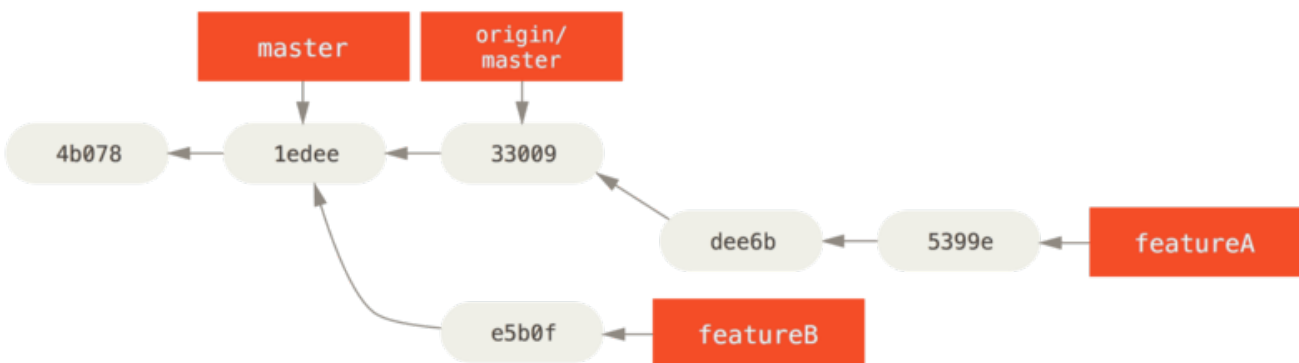


Figure 71. Commit history after `featureA` work.

Ker ste vejo ponovno bazirali, morate določiti `-f` za vaš ukaz potiskanja, da lahko zamenjate vejo `featureA` na strežniku s pošiljanji, ki niso njihovi potomci. Alternativa bi bila potisniti to novo delo na različno vejo na strežniku (mogoče imenovano `featureAv2`).

Poglejmo še en bolj možen scenarij: vzdrževalec je pogledal delo v vaši drugi veji in mu je koncept všeč, vendar bi rad, da spremenite podrobnost implementacije. Tudi vzeli boste to priložnost, da premaknete delo, da bo osnovano na trenutni veji projekta `master`. Začnete novo vejo, ki je osnovana na trenutno `origin/master`, tam stisnete spremembe `featureB`, rešite kakršnekoli konflikte, naredite implementacijo sprememb in nato to potisnite na novo vejo:

```

$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
# (change implementation)
$ git commit
$ git push myfork featureBv2

```

Opcija `--squash` vzame vso delo na združeni veji in jih stisne v en skupek sprememb, ki producira stanje repozitorija, kakor da bi se zgodilo realno združevanje, brez dejanskega združevanja pošiljanja. To pomeni, da bo vaše bodoče pošiljanje imelo samo enega starša in vam omogoča predstavitev vseh sprememb iz druge veje in nato naredite več sprememb pred snemanjem novega pošiljanja. Tudi opcija `--no-commit` je lahko uporabna za zakasnitev združevanja pošiljanja v primeru privzetega procesa združevanja.

Sedaj lahko pošljete vzdrževalcu sporočilo, da ste naredili zahtevane spremembe in te spremembe lahko najdejo v vaši veji `featureBv2`.

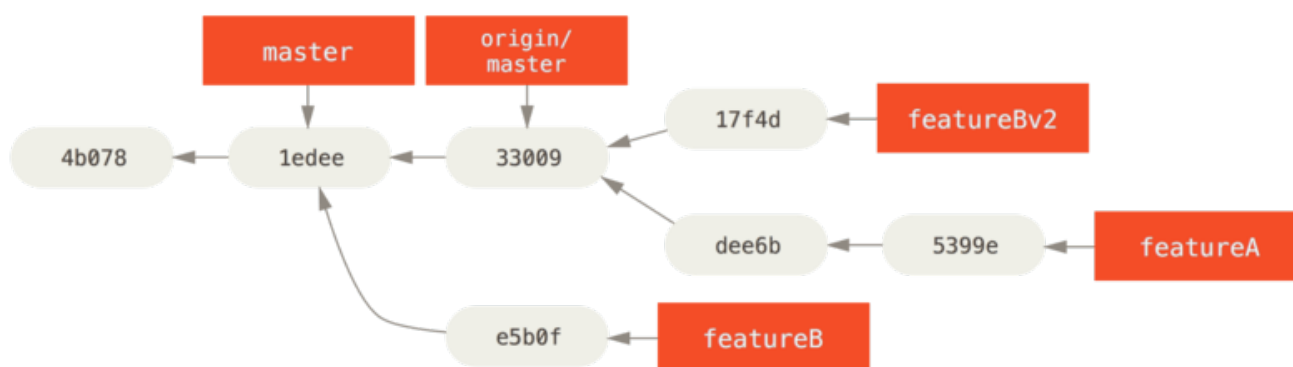


Figure 72. Commit history after `featureBv2` work.

Javni projekt preko e-pošte

Mnogi projekti so vzpostavili procedure za sprejemanje popravkov - preveriti boste morali določena pravila za vsak projekt, ker se razlikujejo. Odkar je na voljo nekaj starejših, večjih projektov, ki sprejemajo popravke preko razvijalskega e-poštnega seznama, bomo šli skozi primer tega sedaj.

Potek dela je podoben prejšnjemu primeru uporabe - ustvarite tematske veje za vsak popravek serije, na kateri delate. Razlika je, kako jih pošljete projektu. Namesto forkanja projekta in potiskanja v vašo lastno zapisljivo verzijo, generirate verzije e-pošte za vsako od serij pošiljanja in jih pošljete po e-pošti razvijalskemu e-poštnemu seznamu:

```

$ git checkout -b topicA
# (work)
$ git commit
# (work)
$ git commit

```

Sedaj imate dve pošiljanji, ki ju želite poslati na e-poštni seznam. Uporabili boste `git`

`format-patch`, da generirate mbox oblikovane datoteke, ki jih lahko pošljete preko e-pošte na seznam - vsako pošiljanje pretvori v sporočilo e-pošte s prvo vrstico sporočila pošiljanja kot zadevo in preostanek sporočila plus popravek, ki ga pošiljanje predstavlja kot telo. Lepa stvar pri tem je, da uporaba popravka iz e-pošte generirane z `format-patch` ohrani vse informacije pošiljanja ustrezno.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

Ukaz `format-patch` izpiše imena datotek popravka, ki jih ustvari. Preklop `-M` pove Git-u, da išče preimenovanja. Datoteke končno izgledajo takole:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Omejite dnevnik funkcionalnosti na prvih 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
2.1.0
```

Lahko tudi uredite te datoteke popravka, da dodate več informacij za seznam e-pošte, za katero ne želite, da se prikaže v sporočilu pošiljanja. Če dodate tekst med vrstico `---` in začetek popravka (vrstica `diff --git`), potem ga razvijalci lahko preberejo; vendar uporaba popravka ga izključuje.

Da to pošljete po e-pošti na e-poštni seznam, lahko ali prilepите datoteko v vaš e-poštni program ali pošljete preko programa ukazne vrstice. Prilepljenje teksta pogostokrat

povzročča težave oblikovanja, posebej s “pametnejšimi” klienti, ki ne ohranjajo novih vrstic in ostalih praznih znakov ustrezno. Na srečo, Git ponuja orodje, da vam ustrezno pomaga poslati oblikovane popravke preko IMAP, ki so lahko enostavnejši za vas. Prikazali bomo, kako poslati popravek preko Gmail-a, kar je e-poštni agent, ki ga najbolje poznamo; preberete lahko podrobna navodila za število poštnih programov na koncu zgoraj omenjene datoteke [Documentation/SubmittingPatches](#) v izvorni kodi Git.

Najprej morate nastaviti sekcijo `imap` v vašo datoteko `~/.gitconfig`. Lahko nastavite vsako vrednost ločeno s serijo ukazov `git config` ali jih lahko dodate ročno, vendar na koncu vaše nastavitvene datoteke bi morale izgledati nekaj takega:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

Če vaš strežnik IAMP ne uporablja SSL, zadnji dve vrstici verjetno nista potrebni in vrednost gostitelja bo `imap://` namesto `imaps://`. Ko je to nastavljeno, lahko uporabite `git send-email`, da dodate serije popravkov v mapo Drafts določenega strežnika IMAP:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

Na tej točki, bi morali imeti dostop do vaše mape osnutkov (Drafts), lahko spremenite polje `To` za seznam e-pošte, na katerega pošiljate popravek, opsijsko `CC` za vzdrževalca ali osebo odgovorno za to sekcijo in lahko pošiljate.

Lahko tudi pošljete popravek preko strežnika SMTP. Kot prej, lahko nastavite vsako vrednost ločeno s serijo ukazov `git config`, ali pa jih lahko dodate ročno v sekciji `sendemail` v vaši datoteki `~/.gitconfig`.

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

Kot je to narejeno, lahko uporabite `git send-email`, da pošljete vaše popravke:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Nato Git izpljune kopijo informacij dnevnika, kar izgleda nekako takole za vsak popravek, ki ga pošiljate:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

Povzetek

Ta sekcija je pokrila število pogostih potekov dela za ukvarjanje z mnogimi zelo različnimi tipi projektov Git, na katere boste verjetno naleteli in predstavila nekaj novih orogij, da vam pomagajo pri tem procesu. Naslednje boste videli, kako delati obratno: vzdrževanje projekta Git. Naučili se boste o dobrohotnem diktatorju ali integracijskem upravitelju.

Vzdrževanje projekta

Kot dodatek vedenju, kako učinkovito prispevati projektu, boste verjetno morali vedeti, kako ga vzdrževati. To lahko sestoji iz sprejemanja in uporabe popravkov generiranih preko `format-patch` in poslanih preko e-pošte k vam, ali integracije sprememb v oddaljenih vejah za repozitorije, ki ste jih dodali kot daljave v vaš projekt. Bodisi ali vzdržujete kanonični repozitorij ali želite pomagati s potrditvijo ali odobritvijo popravkov, morate vedeti, kako sprejeti delo na način, ki je najbolj jasen za ostale, ki prispevajo in da naredite trajnostno na dolgi rok.

Delo na tematskih vejah

Ko razmišljate o integraciji novega dela, je v splošnem dobra ideja poskusiti na tematski veji - začasni veji, posebej narejeni za preskušanje tega novega dela. Na ta način je enostavno prilagoditi popravek individualno ali ga pustiti, če ne deluje dokler nimate časa priti nazaj nanj. Če ustvarite enostavno ime veje na osnovi teme dela, katerega boste poskusili, kot je `ruby_client` ali nekaj podobno opisljivega, si lahko enostavno zapomnite, če jo morate opustiti za nekaj časa in se kasneje vrniti. Vzdrževalec projekta Git tudi stremi k prostorskemu poimenovanju teh vej - kot je `sc/ruby_client`, kjer je `sc` kratica za osebno, ki je prispevala delo. Kot se boste spomnili, lahko ustvarite vejo na osnovi vaše veje master takole:

```
$ git branch sc/ruby_client master
```

Ali če želite takoj nanjo tudi preklopiti, lahko uporabite opcijo `checkout -b`:

```
$ git checkout -b sc/ruby_client master
```

Sedaj ste pripravljeni dodati vaše prispevano delo v to tematsko vejo in določiti, če jo želite združiti v vaše veje na dolgi rok.

Uporaba popravkov iz e-pošte

Če prejmete popravek, ki ga morate integrirati v vaš projekt, preko e-pošte, morate uporabiti popravek na vaši tematski veji, da ga ocenite. Na voljo sta dva načina za uporabo e-poštnega popravka: z `git apply` ali z `git am`.

Uporaba popravka z apply

Če prejmete popravek od nekoga, ki ga je generiral z `git diff` ali Unix ukazom `diff` (kar ni priporočljivo; glejte naslednjo sekcijo), ga lahko uporabite z ukazom `git apply`. Predpostavljamo, da ste shranili popravek v `/tmp/patch-ruby-client.patch`, lahko uporabite popravek sledeče:

```
$ git apply /tmp/patch-ruby-client.patch
```

To spremeni datoteke v vašem delovnem direktoriju. Je skoraj identično pogonni ukaz `patch -p1`, da uporabite popravek, vendar je bolj paranoično in sprejema manj nejasna ujemanja kot popravek. Upravlja tudi dodajanja datotek, brisanja in preimenovanja, če so opisana v obliki `git diff`, kar `patch` ne naredi. Na koncu `git apply` je model "apply all or abort all", kjer je bodisi vse uporabljeno ali nič, kjer `patch` lahko delno uporablja datoteke popravkov, kar pusti vaš delovni direktorij v čudnem stanju. `git apply` je splošno veliko bolj konzervativen kot `patch`. Ne bo ustvaril pošiljanja za vas - po tem, ko ga poženate, morate dati v vmesno fazo in poslati spremembe predstavljene ročno.

Lahko uporabite tudi `git apply`, da vidite, če je popravek uporabljen čisto, preden ga

poskusite dejansko uporabiti - lahko poženete `git apply --check` s popravkom:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Če ni nobenega izpisa, potem bi moral biti popravek uporabljen čisto. Ta ukaz lahko obstaja z ne-ničelnim statusom, če preverjanje ni uspešno, da ga lahko uporabite v skriptah, če želite.

Uporaba popravka z `am`

Če je uporabnik, ki prispeva, Git uporabnik in je bilo dovolj dobro uporabiti ukaz `format-patch` za generiranje njegovega popravka, potem je vaše delo enostavnejše, ker popravek vsebuje informacije avtorja in sporočilo pošiljanja za vas. Če lahko, vzpodbudite vaše prispevalce, da uporabljajo `format-patch` namesto `diff` za generiranje popravkov za vas. Morali bi uporabiti samo `git apply` pri opuščeni popravkih in podobnih stvareh.

Da uporabite popravek generiran s `format-patch`, uporabite `git am`. Tehnično je `git am` zgrajen, da prebere datoteko mbox, ki je enostaven tekstovni format za shranjevanje enega ali več e-poštnih sporočil v eni tekstovni datoteki. Izgleda nekako sledeče:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

This is the beginning of the output of the `format-patch` command that you saw in the previous section. This is also a valid mbox e-mail format. If someone has e-mailed you the patch properly using `git send-email`, and you download that into an mbox format, then you can point `git am` to that mbox file, and it will start applying all the patches it sees. If you run a mail client that can save several e-mails out in mbox format, you can save entire patch series into a file and then use `git am` to apply them one at a time.

However, if someone uploaded a patch file generated via `format-patch` to a ticketing system or something similar, you can save the file locally and then pass that file saved on your disk to `git am` to apply it:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

You can see that it applied cleanly and automatically created the new commit for you. The author information is taken from the e-mail's `From` and `Date` headers, and the

message of the commit is taken from the **Subject** and body (before the patch) of the e-mail. For example, if this patch was applied from the mbox example above, the commit generated would look something like this:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:    Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:    Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700
```

```
    add limit to log function
```

```
    Limit log functionality to the first 20
```

The **Commit** information indicates the person who applied the patch and the time it was applied. The **Author** information is the individual who originally created the patch and when it was originally created.

But it's possible that the patch won't apply cleanly. Perhaps your main branch has diverged too far from the branch the patch was built from, or the patch depends on another patch you haven't applied yet. In that case, the **git am** process will fail and ask you what you want to do:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

This command puts conflict markers in any files it has issues with, much like a conflicted merge or rebase operation. You solve this issue much the same way – edit the file to resolve the conflict, stage the new file, and then run **git am --resolved** to continue to the next patch:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

If you want Git to try a bit more intelligently to resolve the conflict, you can pass a **-3** option to it, which makes Git attempt a three-way merge. This option isn't on by default because it doesn't work if the commit the patch says it was based on isn't in your repository. If you do have that commit – if the patch was based on a public

commit – then the `-3` option is generally much smarter about applying a conflicting patch:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In this case, this patch had already been applied. Without the `-3` option, it looks like a conflict.

If you're applying a number of patches from an mbox, you can also run the `am` command in interactive mode, which stops at each patch it finds and asks if you want to apply it:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

This is nice if you have a number of patches saved, because you can view the patch first if you don't remember what it is, or not apply the patch if you've already done so.

When all the patches for your topic are applied and committed into your branch, you can choose whether and how to integrate them into a longer-running branch.

Checking Out Remote Branches

If your contribution came from a Git user who set up their own repository, pushed a number of changes into it, and then sent you the URL to the repository and the name of the remote branch the changes are in, you can add them as a remote and do merges locally.

For instance, if Jessica sends you an e-mail saying that she has a great new feature in the `ruby-client` branch of her repository, you can test it by adding the remote and checking out that branch locally:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

If she e-mails you again later with another branch containing another great feature, you can fetch and check out because you already have the remote setup.

This is most useful if you're working with a person consistently. If someone only has a single patch to contribute once in a while, then accepting it over e-mail may be less time consuming than requiring everyone to run their own server and having to continually add and remove remotes to get a few patches. You're also unlikely to want to have hundreds of remotes, each for someone who contributes only a patch or two. However, scripts and hosted services may make this easier – it depends largely on how you develop and how your contributors develop.

The other advantage of this approach is that you get the history of the commits as well. Although you may have legitimate merge issues, you know where in your history their work is based; a proper three-way merge is the default rather than having to supply a `-3` and hope the patch was generated off a public commit to which you have access.

If you aren't working with a person consistently but still want to pull from them in this way, you can provide the URL of the remote repository to the `git pull` command. This does a one-time pull and doesn't save the URL as a remote reference:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch          HEAD      -> FETCH_HEAD
Merge made by recursive.
```

Determining What Is Introduced

Now you have a topic branch that contains contributed work. At this point, you can determine what you'd like to do with it. This section revisits a couple of commands so you can see how you can use them to review exactly what you'll be introducing if you merge this into your main branch.

It's often helpful to get a review of all the commits that are in this branch but that aren't in your master branch. You can exclude commits in the master branch by adding the `--not` option before the branch name. This does the same thing as the `master..contrib` format that we used earlier. For example, if your contributor sends you two patches and you create a branch called `contrib` and applied those patches there, you can run this:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date: Fri Oct 24 09:53:59 2008 -0700
```

seeing if this helps the gem

```
commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date: Mon Oct 22 19:38:36 2008 -0700
```

updated the gemspec to hopefully work better

To see what changes each commit introduces, remember that you can pass the `-p` option to `git log` and it will append the diff introduced to each commit.

To see a full diff of what would happen if you were to merge this topic branch with another branch, you may have to use a weird trick to get the correct results. You may think to run this:

```
$ git diff master
```

This command gives you a diff, but it may be misleading. If your `master` branch has moved forward since you created the topic branch from it, then you'll get seemingly strange results. This happens because Git directly compares the snapshots of the last commit of the topic branch you're on and the snapshot of the last commit on the `master` branch. For example, if you've added a line in a file on the `master` branch, a direct comparison of the snapshots will look like the topic branch is going to remove that line.

If `master` is a direct ancestor of your topic branch, this isn't a problem; but if the two histories have diverged, the diff will look like you're adding all the new stuff in your topic branch and removing everything unique to the `master` branch.

What you really want to see are the changes added to the topic branch – the work you'll introduce if you merge this branch with master. You do that by having Git compare the last commit on your topic branch with the first common ancestor it has with the master branch.

Technically, you can do that by explicitly figuring out the common ancestor and then running your diff on it:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

However, that isn't convenient, so Git provides another shorthand for doing the

same thing: the triple-dot syntax. In the context of the `diff` command, you can put three periods after another branch to do a `diff` between the last commit of the branch you're on and its common ancestor with another branch:

```
$ git diff master...contrib
```

This command shows you only the work your current topic branch has introduced since its common ancestor with `master`. That is a very useful syntax to remember.

Integrating Contributed Work

When all the work in your topic branch is ready to be integrated into a more mainline branch, the question is how to do it. Furthermore, what overall workflow do you want to use to maintain your project? You have a number of choices, so we'll cover a few of them.

Merging Workflows

One simple workflow merges your work into your `master` branch. In this scenario, you have a `master` branch that contains basically stable code. When you have work in a topic branch that you've done or that someone has contributed and you've verified, you merge it into your `master` branch, delete the topic branch, and then continue the process. If we have a repository with work in two branches named `ruby_client` and `php_client` that looks like [History with several topic branches](#), and merge `ruby_client` first and then `php_client` next, then your history will end up looking like [After a topic branch merge](#).

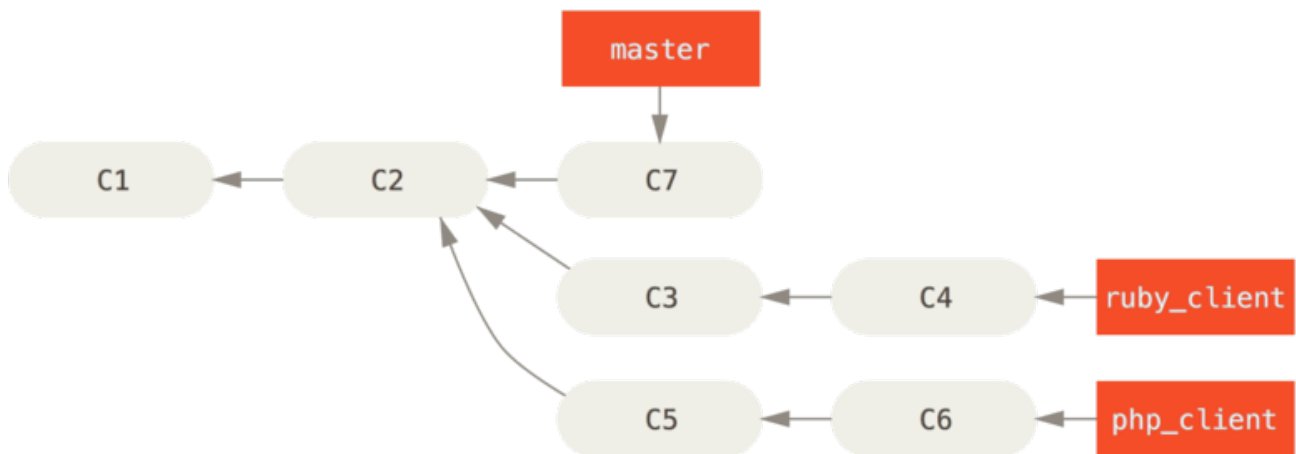


Figure 73. History with several topic branches.

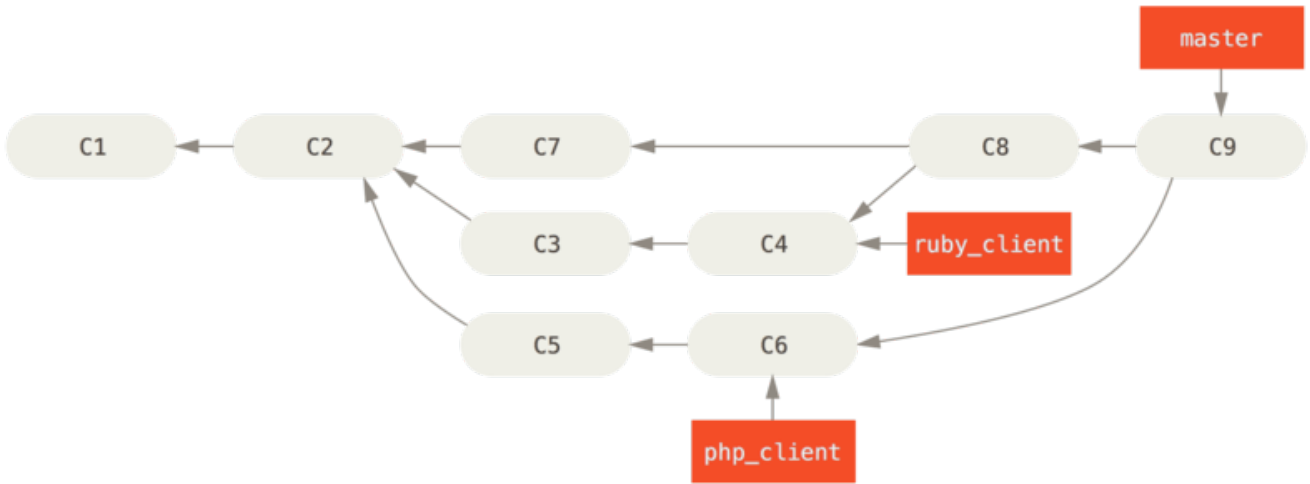


Figure 74. After a topic branch merge.

That is probably the simplest workflow, but it can possibly be problematic if you're dealing with larger or more stable projects where you want to be really careful about what you introduce.

If you have a more important project, you might want to use a two-phase merge cycle. In this scenario, you have two long-running branches, `master` and `develop`, in which you determine that `master` is updated only when a very stable release is cut and all new code is integrated into the `develop` branch. You regularly push both of these branches to the public repository. Each time you have a new topic branch to merge in ([Before a topic branch merge.](#)), you merge it into `develop` ([After a topic branch merge.](#)); then, when you tag a release, you fast-forward `master` to wherever the now-stable `develop` branch is ([After a project release.](#)).

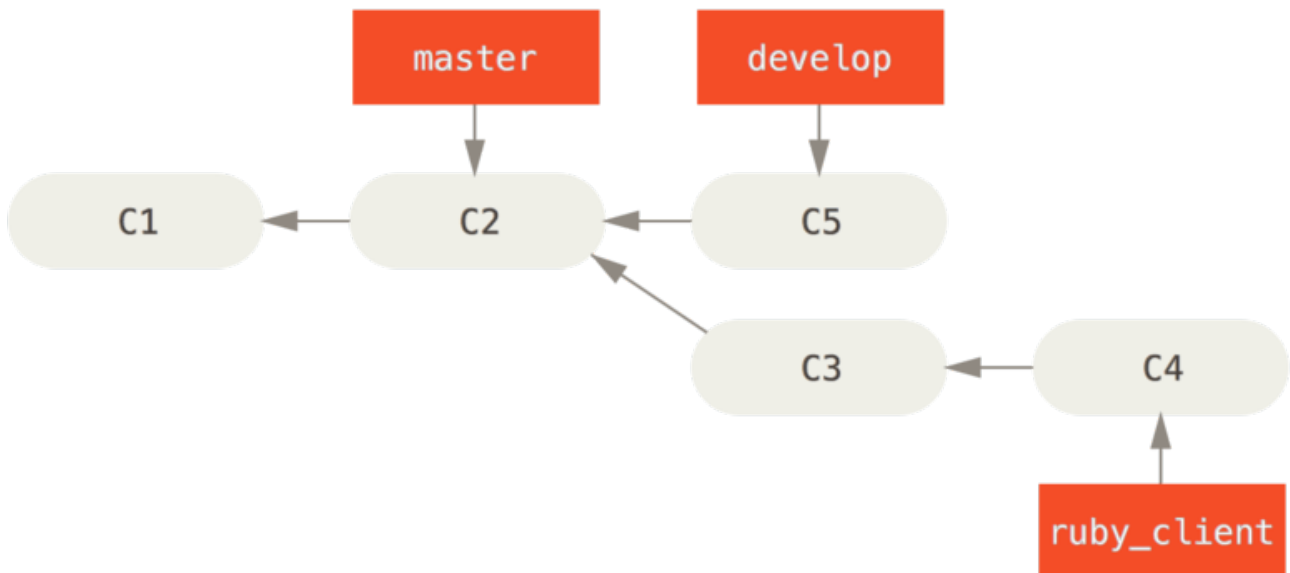


Figure 75. Before a topic branch merge.

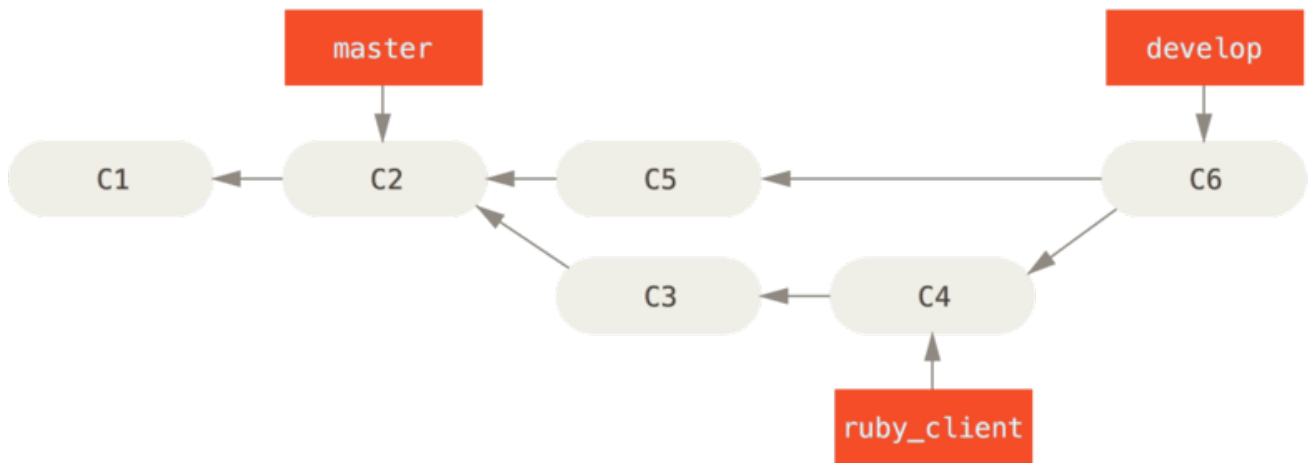


Figure 76. After a topic branch merge.

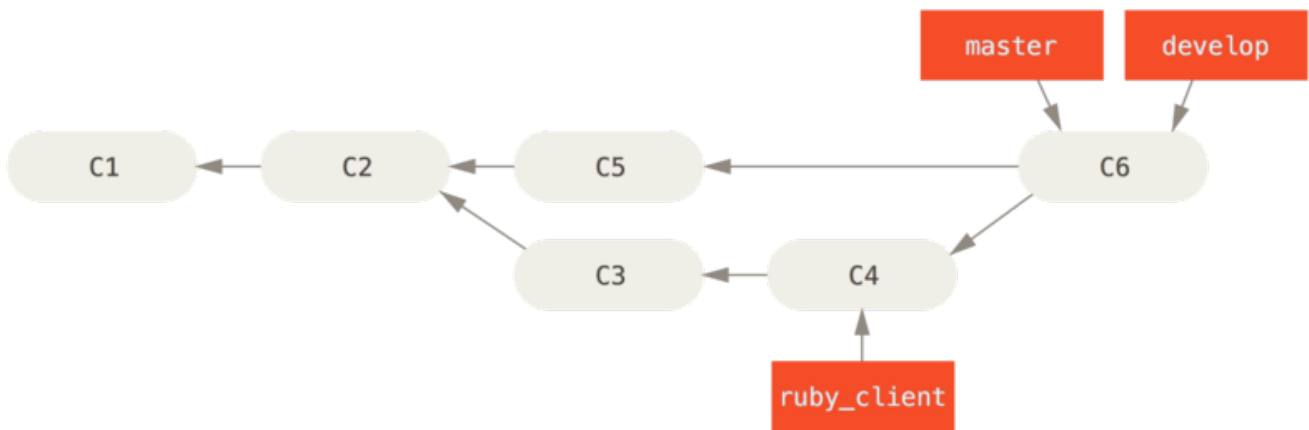


Figure 77. After a project release.

This way, when people clone your project’s repository, they can either check out `master` to build the latest stable version and keep up to date on that easily, or they can check out `develop`, which is the more cutting-edge stuff. You can also continue this concept, having an `integrate` branch where all the work is merged together. Then, when the codebase on that branch is stable and passes tests, you merge it into a `develop` branch; and when that has proven itself stable for a while, you fast-forward your `master` branch.

Large-Merging Workflows

The Git project has four long-running branches: `master`, `next`, and `pu` (proposed updates) for new work, and `maint` for maintenance backports. When new work is introduced by contributors, it’s collected into topic branches in the maintainer’s repository in a manner similar to what we’ve described (see [Managing a complex series of parallel contributed topic branches](#)). At this point, the topics are evaluated to determine whether they’re safe and ready for consumption or whether they need more work. If they’re safe, they’re merged into `next`, and that branch is pushed up so everyone can try the topics integrated together.

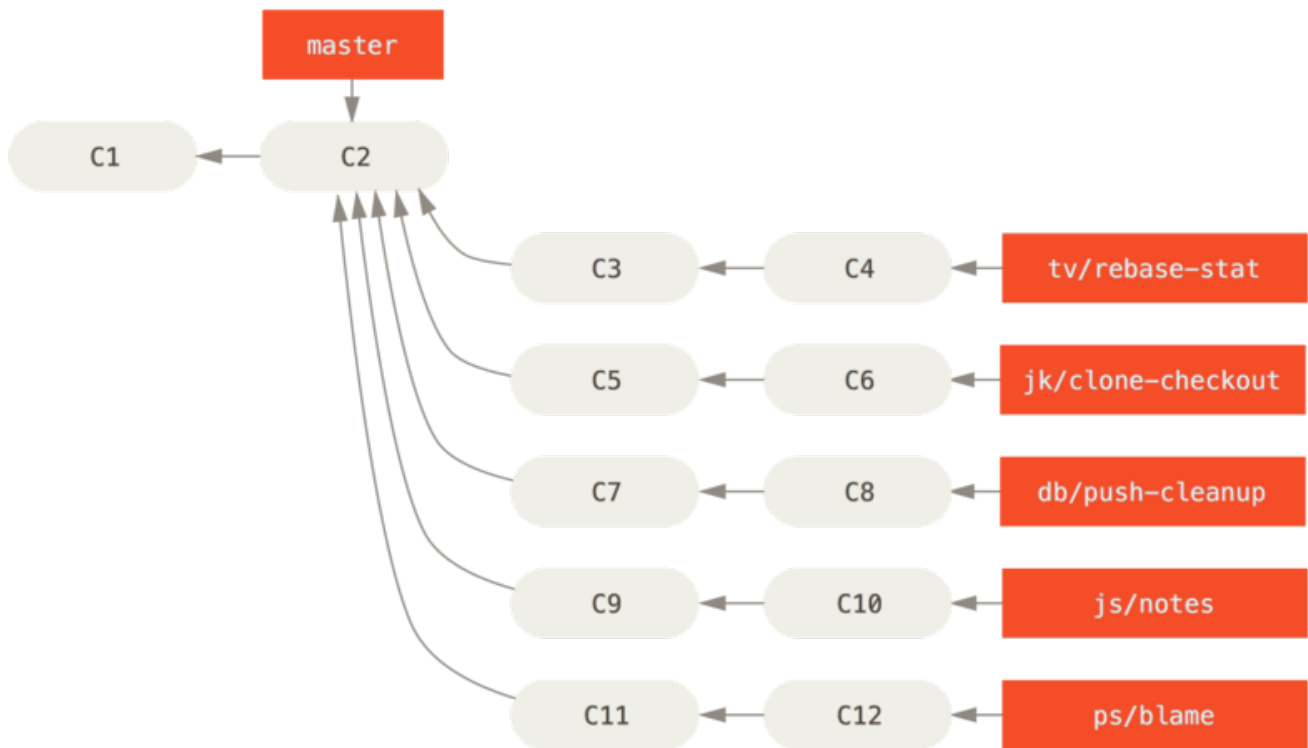


Figure 78. Managing a complex series of parallel contributed topic branches.

If the topics still need work, they're merged into `pu` instead. When it's determined that they're totally stable, the topics are re-merged into `master` and are then rebuilt from the topics that were in `next` but didn't yet graduate to `master`. This means `master` almost always moves forward, `next` is rebased occasionally, and `pu` is rebased even more often:

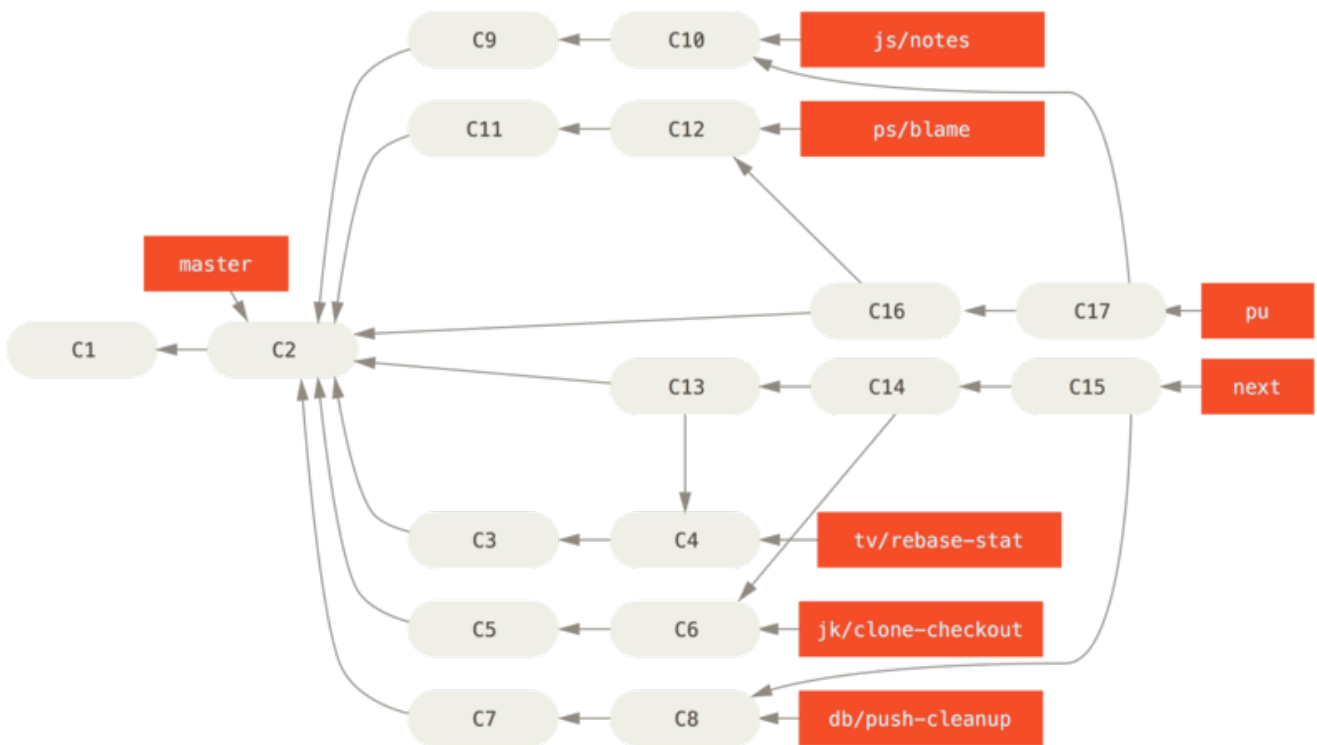


Figure 79. Merging contributed topic branches into long-term integration branches.

When a topic branch has finally been merged into `master`, it's removed from the repository. The Git project also has a `maint` branch that is forked off from the last

release to provide backported patches in case a maintenance release is required. Thus, when you clone the Git repository, you have four branches that you can check out to evaluate the project in different stages of development, depending on how cutting edge you want to be or how you want to contribute; and the maintainer has a structured workflow to help them vet new contributions.

Rebasing and Cherry Picking Workflows

Other maintainers prefer to rebase or cherry-pick contributed work on top of their master branch, rather than merging it in, to keep a mostly linear history. When you have work in a topic branch and have determined that you want to integrate it, you move to that branch and run the rebase command to rebuild the changes on top of your current master (or `develop`, and so on) branch. If that works well, you can fast-forward your `master` branch, and you'll end up with a linear project history.

The other way to move introduced work from one branch to another is to cherry-pick it. A cherry-pick in Git is like a rebase for a single commit. It takes the patch that was introduced in a commit and tries to reapply it on the branch you're currently on. This is useful if you have a number of commits on a topic branch and you want to integrate only one of them, or if you only have one commit on a topic branch and you'd prefer to cherry-pick it rather than run rebase. For example, suppose you have a project that looks like this:

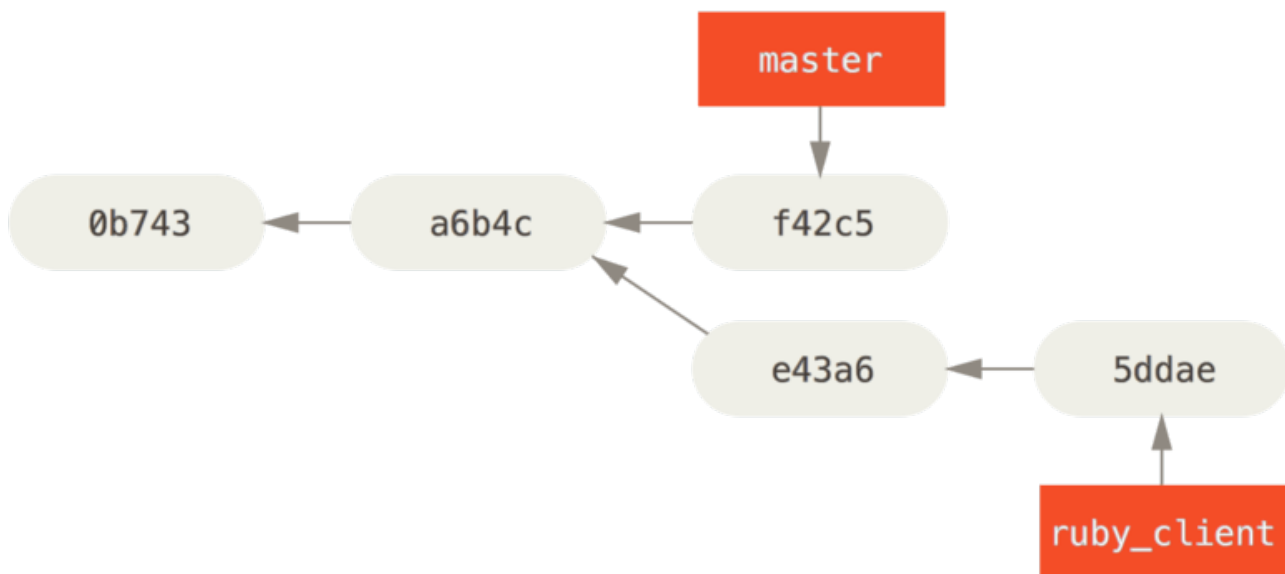


Figure 80. Example history before a cherry-pick.

If you want to pull commit `e43a6` into your master branch, you can run

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
```

This pulls the same change introduced in `e43a6`, but you get a new commit SHA-1

value, because the date applied is different. Now your history looks like this:

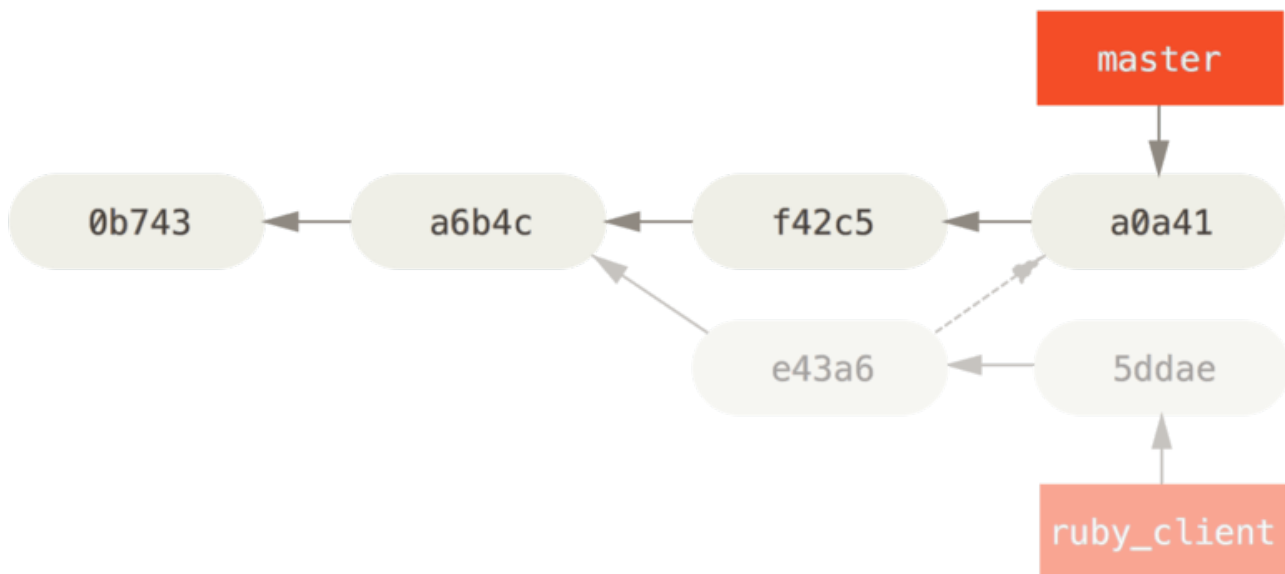


Figure 81. History after cherry-picking a commit on a topic branch.

Now you can remove your topic branch and drop the commits you didn't want to pull in.

Rerere

If you're doing lots of merging and rebasing, or you're maintaining a long-lived topic branch, Git has a feature called "rerere" that can help.

Rerere stands for "reuse recorded resolution" – it's a way of shortcutting manual conflict resolution. When rerere is enabled, Git will keep a set of pre- and post-images from successful merges, and if it notices that there's a conflict that looks exactly like one you've already fixed, it'll just use the fix from last time, without bothering you with it.

This feature comes in two parts: a configuration setting and a command. The configuration setting is `rerere.enabled`, and it's handy enough to put in your global config:

```
$ git config --global rerere.enabled true
```

Now, whenever you do a merge that resolves conflicts, the resolution will be recorded in the cache in case you need it in the future.

If you need to, you can interact with the rerere cache using the `git rerere` command. When it's invoked alone, Git checks its database of resolutions and tries to find a match with any current merge conflicts and resolve them (although this is done automatically if `rerere.enabled` is set to `true`). There are also subcommands to see what will be recorded, to erase specific resolution from the cache, and to clear the entire cache. We will cover rerere in more detail in [Rerere](#).

Tagging Your Releases

When you've decided to cut a release, you'll probably want to drop a tag so you can re-create that release at any point going forward. You can create a new tag as discussed in [Osnove Git](#). If you decide to sign the tag as the maintainer, the tagging may look something like this:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

If you do sign your tags, you may have the problem of distributing the public PGP key used to sign your tags. The maintainer of the Git project has solved this issue by including their public key as a blob in the repository and then adding a tag that points directly to that content. To do this, you can figure out which key you want by running `gpg --list-keys`:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
-----
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid           Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Then, you can directly import the key into the Git database by exporting it and piping that through `git hash-object`, which writes a new blob with those contents into Git and gives you back the SHA-1 of the blob:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Now that you have the contents of your key in Git, you can create a tag that points directly to it by specifying the new SHA-1 value that the `hash-object` command gave you:

```
$ git tag -a maintainer-gpg-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

If you run `git push --tags`, the `maintainer-gpg-pub` tag will be shared with everyone. If anyone wants to verify a tag, they can directly import your PGP key by pulling the blob directly out of the database and importing it into GPG:

```
$ git show maintainer-gpg-pub | gpg --import
```

They can use that key to verify all your signed tags. Also, if you include instructions in the tag message, running `git show <tag>` will let you give the end user more specific instructions about tag verification.

Generating a Build Number

Because Git doesn't have monotonically increasing numbers like `v123` or the equivalent to go with each commit, if you want to have a human-readable name to go with a commit, you can run `git describe` on that commit. Git gives you the name of the nearest tag with the number of commits on top of that tag and a partial SHA-1 value of the commit you're describing:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

This way, you can export a snapshot or build and name it something understandable to people. In fact, if you build Git from source code cloned from the Git repository, `git --version` gives you something that looks like this. If you're describing a commit that you have directly tagged, it gives you the tag name.

The `git describe` command favors annotated tags (tags created with the `-a` or `-s` flag), so release tags should be created this way if you're using `git describe`, to ensure the commit is named properly when described. You can also use this string as the target of a checkout or show command, although it relies on the abbreviated SHA-1 value at the end, so it may not be valid forever. For instance, the Linux kernel recently jumped from 8 to 10 characters to ensure SHA-1 object uniqueness, so older `git describe` output names were invalidated.

Preparing a Release

Now you want to release a build. One of the things you'll want to do is create an archive of the latest snapshot of your code for those poor souls who don't use Git. The command to do this is `git archive`:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

If someone opens that tarball, they get the latest snapshot of your project under a project directory. You can also create a zip archive in much the same way, but by passing the `--format=zip` option to `git archive`:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

You now have a nice tarball and a zip archive of your project release that you can upload to your website or e-mail to people.

The Shortlog

It's time to e-mail your mailing list of people who want to know what's happening in your project. A nice way of quickly getting a sort of changelog of what has been added to your project since your last release or e-mail is to use the `git shortlog` command. It summarizes all the commits in the range you give it; for example, the following gives you a summary of all the commits since your last release, if your last release was named `v1.0.1`:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
  Add support for annotated tags to Grit::Tag
  Add packed-refs annotated tag support.
  Add Grit::Commit#to_patch
  Update version and History.txt
  Remove stray `puts`
  Make ls_tree ignore nils

Tom Preston-Werner (4):
  fix dates in history
  dynamic version method
  Version bump to 1.0.2
  Regenerated gemspec for version 1.0.2
```

You get a clean summary of all the commits since `v1.0.1`, grouped by author, that you can e-mail to your list.

Povzetek

Počutiti se bi morali dokaj udobno pri prispevanju k projektu v Git-u kot tudi pri vzdrževanju vašega lastnega projekta in povezovanju prispevkov ostalih uporabnikov. Čestitke, ker ste učinkovit Git razvijalec! V naslednjem poglavju se boste naučili, kako uporabljati največjo in najbolj popularno storitev Git gostovanja, GitHub.

GitHub

GitHub je posamezni največji gostitelj za repozitorije Git in je centralna točka sodelovanja za milijone razvijalcev in projektov.

Velik delež vseh repozitorijev Git je gostovan na GitHub-u in mnogi odprto kodni projekti ga uporabljajo za gostovanje Git, sledenje težav, pregled kode in ostale stvari. Torej medtem ko ni direkten del odprto kodnega projekta Git, je dobra možnost, da boste želeli ali potrebovali interakcijo z GitHub-om na določeni točki, ko boste uporabljali Git profesionalno.

To poglavje je o efektivni uporabi GitHub-a. Pokrili bomo prijavo in upravljanje računa, izdelavo in uporabo repozitorijev Git, skupen potek dela za prispevanje k projektom in sprejemanje prispevkov k vašim, GitHub-ov programski vmesnik in veliko manjših nasvetov, ki bodo naredili vaše življenje enostavnejše v splošnem.

Če vas uporaba GitHub-a za gostovanje vaših lastnih projektov ne zanima ali za sodelovanje z ostalimi projekti, ki so gostovani na GitHub-u, lahko varno preskočite na [Orodja Git](#).

Interfaces Change

WARNING

Pomembno je opomniti, da kot mnoge aktivne spletne strani, so elementi UI v teh posnetkih povezani, da se spremenijo tekom časa. Upajmo, da je splošna ideja kaj poskušate doseči tu, še vedno tam, vendar če želite bolj posodobljene verzije teh posnetkov zaslonov, verzije na spletu te knjige lahko imajo novejšo posnetke zaslona.

Namestitev in konfiguracija računa

Prva stvar, ki jo morate narediti je nastaviti brezplačni uporabniški račun. Enostavno obiščite <https://github.com>, izberite uporabniško ime, ki še ni zasedeno, ponudite e-poštni naslov in geslo in kliknite na velik zeleni "Sign up for GitHub" gumb.

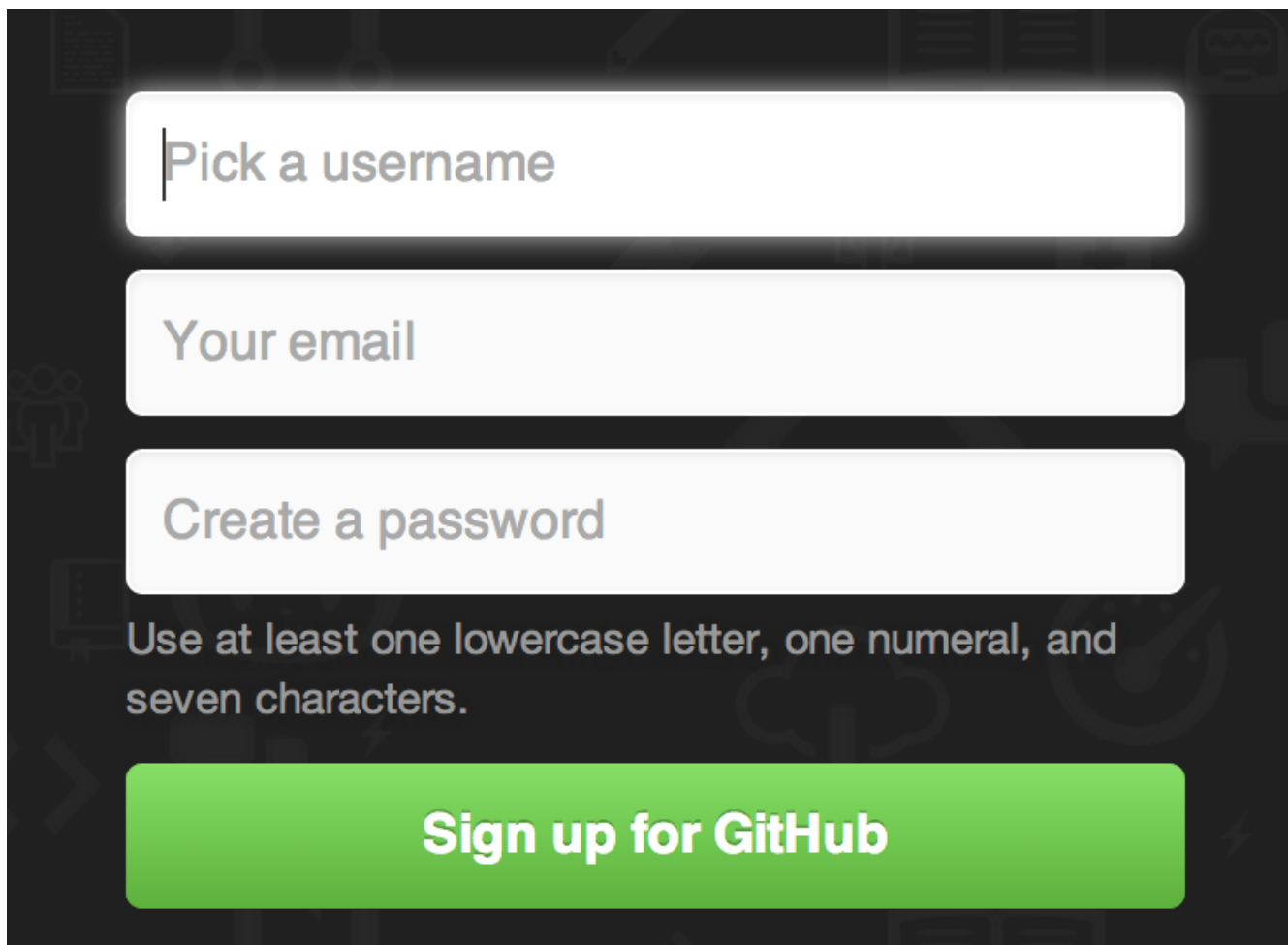


Figure 82. The GitHub sign-up form.

Naslednja stvar, ki jo vidite je stran cenika za nadgrajene plane, vendar je to varno ignorirati za sedaj. GitHub vam bo poslal e-pošto, da potrdite naslov, ki ste ga ponudili. Pojdite naprej in naredite to, je precej pomembno (kot boste videli kasneje).

NOTE

GitHub ponuja vso svojo funkcionalnost brezplačnim računom, z omejitvijo, da so vsi vaši projekti polno javni (vsi imajo bralni dostop). GitHub plačljivi plani vključujejo določeno število privatnih projektov, vendar tega tu ne bomo pokrivali v tej knjigi.

Klik na Octocat logotip levo zgoraj na zaslonu vas bo popeljal na vašo stran plošče. Sedaj ste pripravljeni uporabljati GitHub.

Dostop SSH

Od sedaj dalje, ste se polno sposobni povezati z repozitoriji Git z uporabo protokola <https://>, overitvijo z uporabniškim imenom in geslom, ki ste ga nastavili. Vendar za enostavno kloniranje javnih projektov, se vam niti ni treba prijaviti - račun, ki ste ga ravno ustvarili prihaja v igro, ko forkamo projekte in potiskamo v fork-e nekoliko kasneje.

Če želite uporabljati oddaljeni SSH, boste morali nastaviti javni ključ. (Če ga še nimate, glejte [Generiranje vaših javnih ključev SSH](#).) Odprite vaše nastavitve računa z uporabo povezave zgoraj desno v oknu:

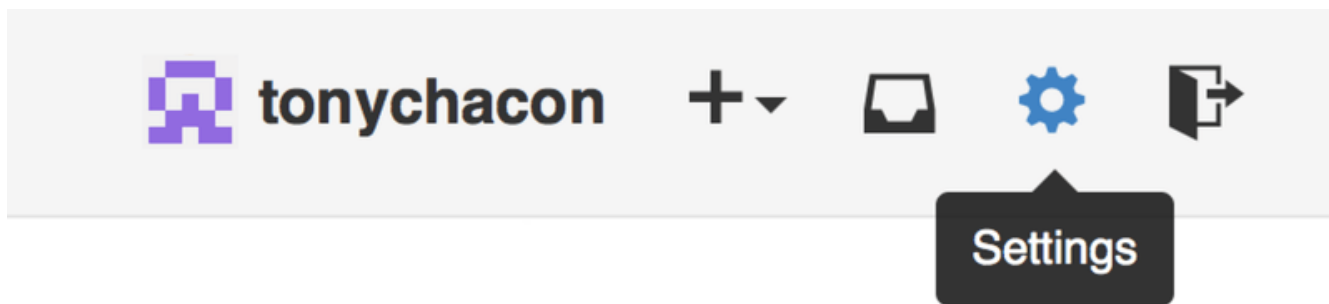


Figure 83. The “Account settings” link.

Nato izberite sekcijo “SSH keys” tekom leve strani.

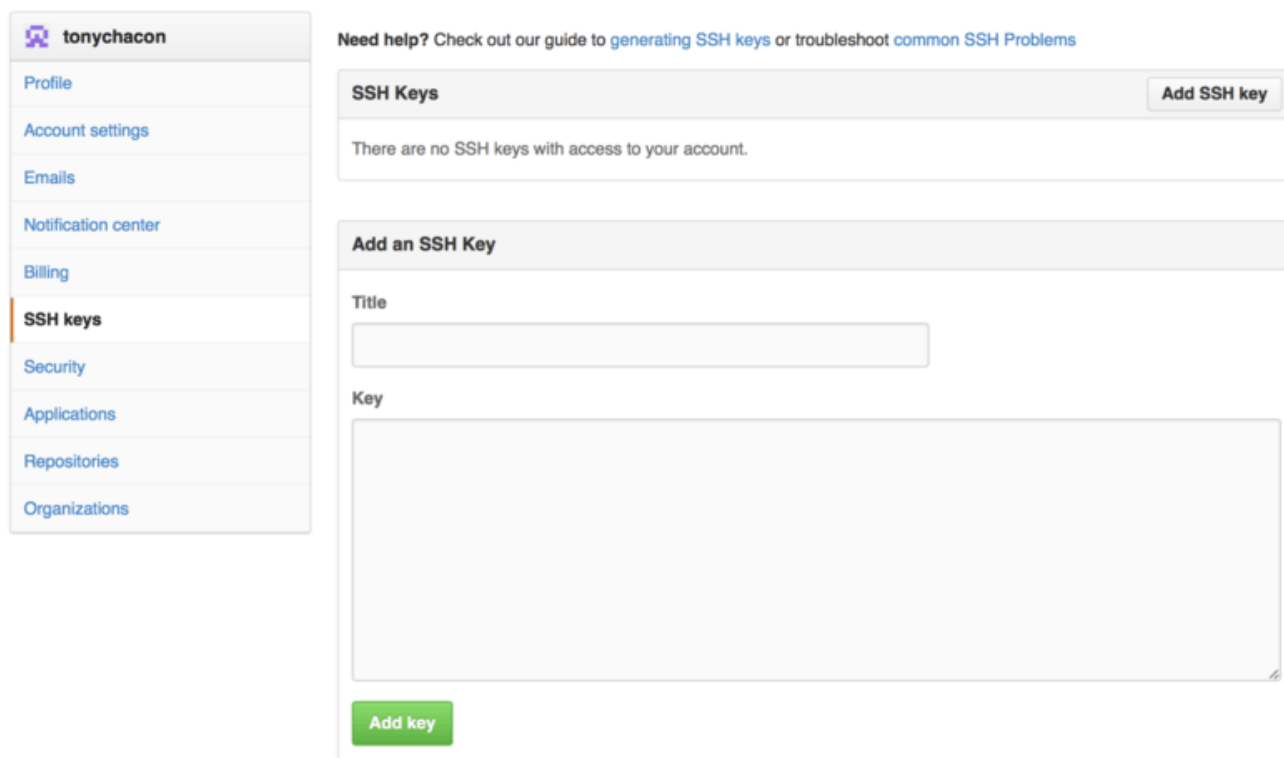


Figure 84. The “SSH keys” link.

Od tam kliknite gumb “Add an SSH key”, podajte ime vašega ključa, prilepite vsebino vaše datoteke javnega ključa `~/.ssh/id_rsa.pub` (ali karkoli ste jo poimenovali) v tekstovno polje in kliknite “Add key”.

NOTE

Zagotovite, da je ime vašega ključa SSH nekaj, kar si lahko zapomnite. Vsakega of vaših ključev lahko poimenujete (npr. “Moj prenosnik” ali “Delovni račun”), tako da če potrebujete povrniti ključ kasneje, lahko enostavno poveste, katerega iščete.

Vaš avatar

Naslednje, če želite, lahko zamenjate avatar, ki je generiran za vas s sliko vaše izbire. Najprej pojdite v zavihek “Profile” (na zavihkom SSH Keys) in kliknite “Upload new picture”.

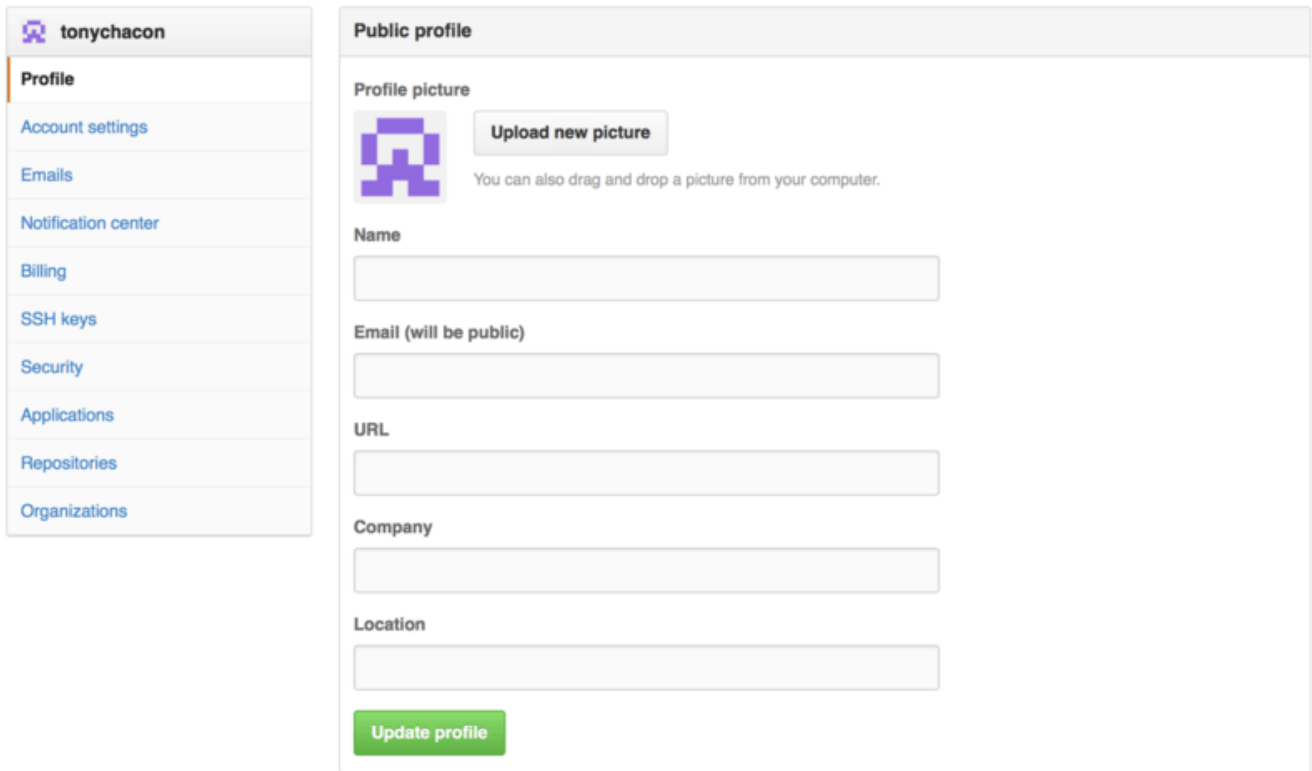


Figure 85. The “Profile” link.

Izbrali bomo kopijo logotipa Git, ki je na našem trdem distku in nato dobimo priložnost, da ga obrežemo.

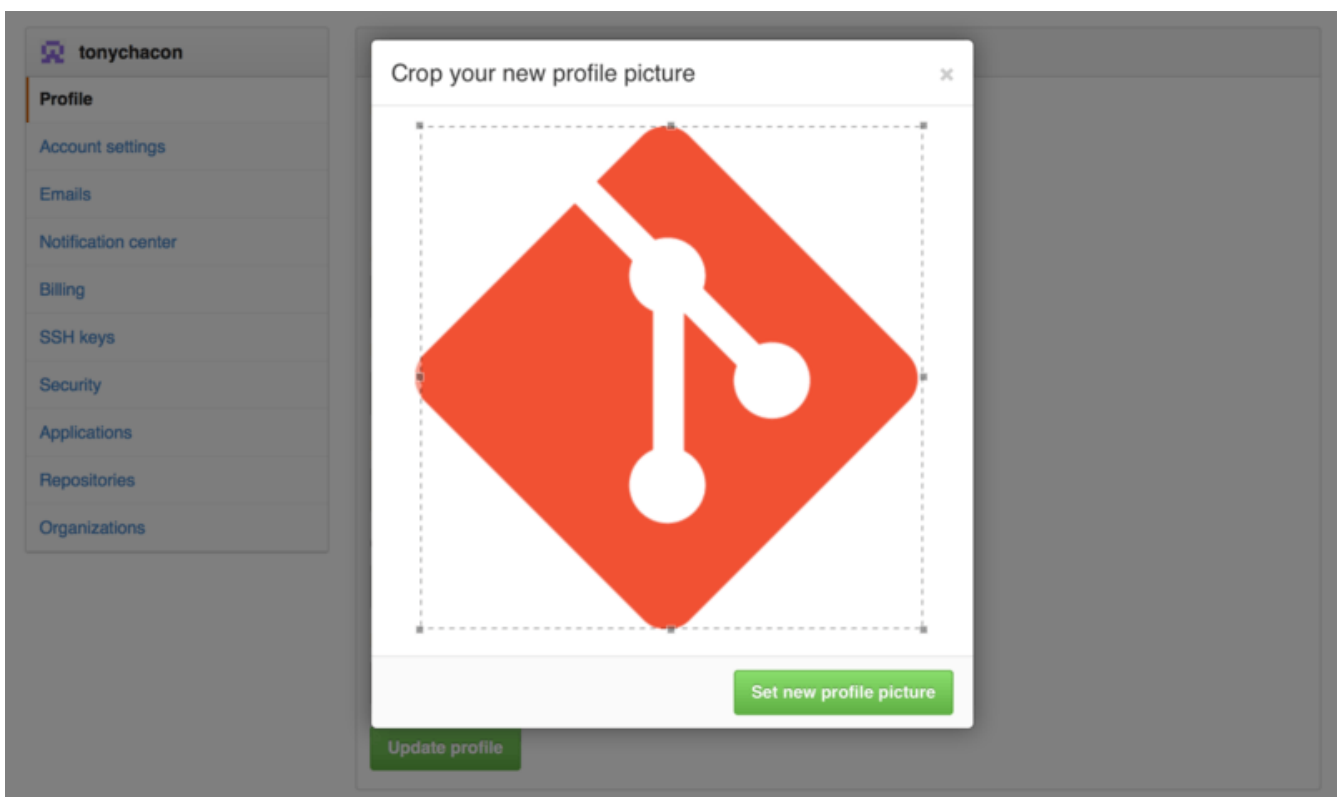


Figure 86. Crop your avatar

Sedaj kjerkoli imate interakcijo na strani, bodo ljudje videli vaš avatar ob vašem uporabniškem imenu.

Če se zgodi, da ste naložili avatar na popularno storitev Gravatar (pogosto uporabljeno za račune Wordpress) bo privzeto uporabljen ta avatar in tega koraka vam ni potrebno delati.

Naslov vaše e-pošte

Način, kako GitHub preslika vaša pošiljanja Git v vašega uporabnika je naslov e-pošte. Če uporabljate več naslovov e-pošte v vaših pošiljanjih in želite, da jih GitHub ustrezno poveže, morate dodati naslov e-pošte, ki ste jo uporabili, v sekciji Emails administracijske sekcije.

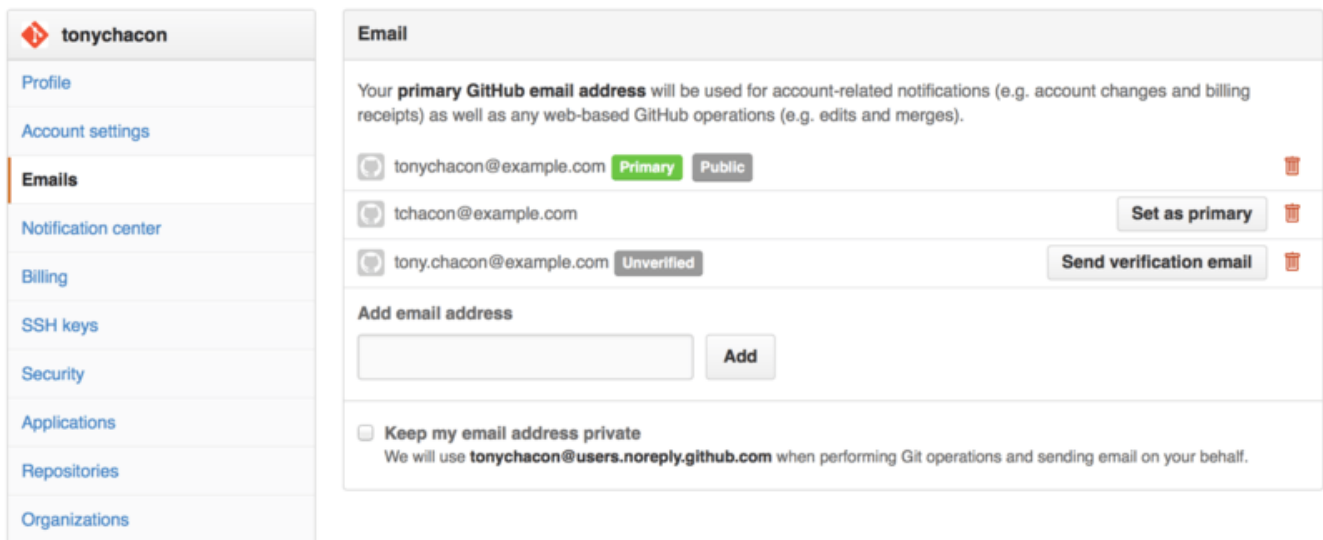


Figure 87. Add email addresses

V [Add email addresses](#) lahko vidimo nekaj različnih možnih stanj. Vrhnji naslov je preverjen in nastavljen na primarni naslov, kar pomeni, da je to tam, kjer boste dobili kakršnokoli obvestilo in račune. Drugi naslov je preverjen in tudi lahko nastavljen kot primaren, če ga želite zamenjati. Zadnji naslov je nepreverjen, kar pomeni, da ga ne morate narediti primarnega. Če GitHub vidi katerokoli od teh sporočil pošiljanj v kateremkoli repozitoriju na strani, ga bo povezal na vašega uporabnika.

Overitev dveh faktorjev

Končno za dodatno varnost, bi morali zagotovo nastaviti overitev dveh faktorjev ali "2FA". Overitev dveh faktorjev je mehanizem overitve, ki postaja zadnje čase bolj in bolj popularen, saj ublaži tveganje ogroženja vašega računa, če je vaše geslo kakorkoli ukradeno. Vključitev bo naredilo, da vas GitHub vpraša za dve različni metodi overitve, torej če je ena ogrožena, napadalec ne bo zmožen dostopati do vašega računa.

Nastavitev overitvije dveh faktorjev lahko najdete pod Security zavihkom nastavitve vašega računa.

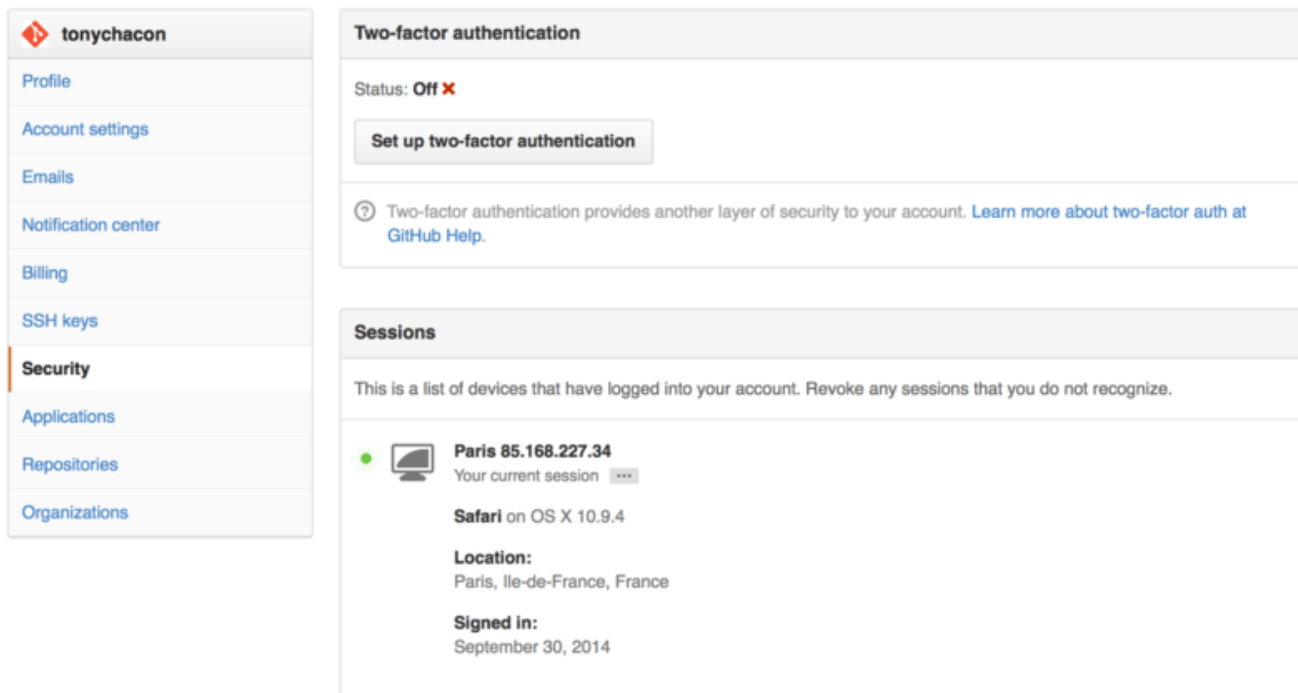


Figure 88. 2FA in the Security Tab

Če kliknete na gumb “Set up two-factor authentication”, vas bo popeljal na nastavitveno stran, kjer lahko izberete uporabo telefonske aplikacije, da generira vaše sekundarno kodo (“časovno osnovano enkratno geslo”) ali lahko naredite, da vam GitHub pošlje kodo preko SMS-a vsakič, ko se morate prijaviti.

Ko izberete katero metodo imate raje in sledite navodilom za nastavev 2FA, bo vaš račun potem malo bolj varen in morali boste ponuditi kodo kot dodatek k vašemu geslu vsakič ko se boste prijavili v GitHub.

Prispevanje k projektu

Sedaj, ko je vaš račun nastavljen, pojdimo skozi nekaj podrobnosti, ki bi vam bile lahko v pomoč pri prispevanju k obstoječem projektu.

Forkanje projektov

Če želite prispevati k obstoječem projektu, do katerega nimate pravic pošiljanja, lahko naredite “fork” projekta. Kaj to pomeni je, da bo GitHub naredil kopijo projekta, ki je popolnoma vaša; pusti vas v vašem imenskem prostoru in vanj lahko pošiljate.

NOTE

Zgodovinski izraz “fork” je bil nekoliko negativen v kontekstu in pomeni, da je nekdo vzel odprto kodni projekt v drugačno smer, včasih ustvaril konkurenčni projekt in razdvojil prispevalce. Na GitHub-u je “fork” enostavno isti projekt v vašem imenskem prostoru, kar vam omogoča, da naredite spremembe k projektu javno kot način prispevanja na bolj odprti način.

V tem primeru projektom ni treba skrbeti za dodajanje uporabnikov kot sodelavcev, da se jim da dostop potiskanja. Ljudje lahko forkajo projekt, vanj potiskajo in prispevajo

svoje spremembe nazaj v originalni repozitorij z ustvarjanjem zahtevka potega, ki ga bomo pokrili v nadaljevanju. To odpre temo diskusije s pregledom kode in lastnik in prispevalec lahko komunicirata o spremembi, dokler lastnik ni z njo zadovoljen in jo lahko združi v projekt.

Da forkate projekt, obiščite stran projekta, kliknite na gumb “Fork” na zgornji desni strani strani.



Figure 89. The “Fork” button.

Po nekaj sekundah, boste pripeljani na novo stran projekta, z vašo lasno kopijo kode.

The GitHub Flow

GitHub is designed around a particular collaboration workflow, centered on Pull Requests. This flow works whether you’re collaborating with a tightly-knit team in a single shared repository, or a globally-distributed company or network of strangers contributing to a project through dozens of forks. It is centered on the [Tematske veje](#) workflow covered in [Veje Git](#).

Here’s how it generally works:

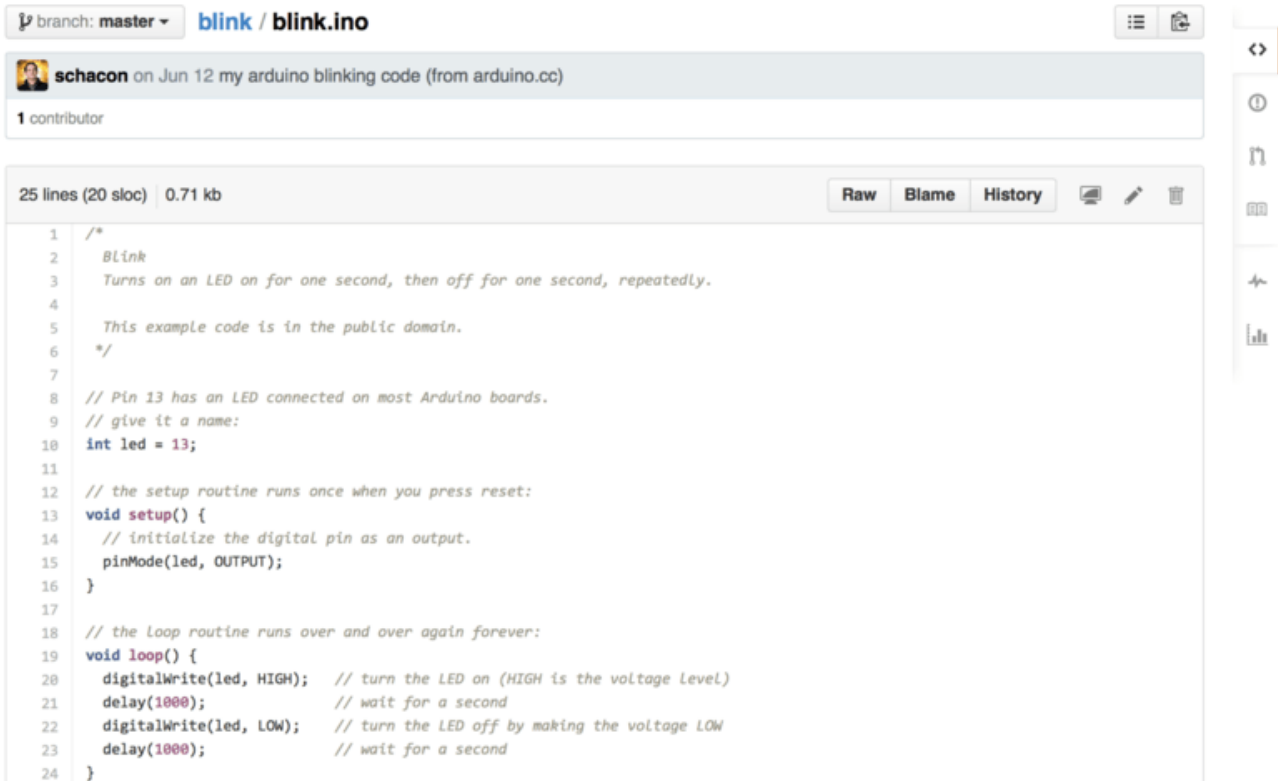
1. Create a topic branch from `master`.
2. Make some commits to improve the project.
3. Push this branch to your GitHub project.
4. Open a Pull Request on GitHub.
5. Discuss, and optionally continue committing.
6. The project owner merges or closes the Pull Request.

This is basically the Integration Manager workflow covered in [Potek dela povezovalnega upravitelja](#), but instead of using email to communicate and review changes, teams use GitHub’s web based tools.

Let’s walk through an example of proposing a change to an open source project hosted on GitHub using this flow.

Creating a Pull Request

Tony is looking for code to run on his Arduino programmable microcontroller and has found a great program file on GitHub at <https://github.com/schacon/blink>.



```
1 /*
2  * Blink
3  * Turns on an LED on for one second, then off for one second, repeatedly.
4
5  * This example code is in the public domain.
6  */
7
8  // Pin 13 has an LED connected on most Arduino boards.
9  // give it a name:
10 int led = 13;
11
12 // the setup routine runs once when you press reset:
13 void setup() {
14   // initialize the digital pin as an output.
15   pinMode(led, OUTPUT);
16 }
17
18 // the loop routine runs over and over again forever:
19 void loop() {
20   digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
21   delay(1000); // wait for a second
22   digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
23   delay(1000); // wait for a second
24 }
```

Figure 90. The project we want to contribute to.

The only problem is that the blinking rate is too fast, we think it's much nicer to wait 3 seconds instead of 1 in between each state change. So let's improve the program and submit it back to the project as a proposed change.

First, we click the *Fork* button as mentioned earlier to get our own copy of the project. Our user name here is "tonychacon" so our copy of this project is at <https://github.com/tonychacon/blink> and that's where we can edit it. We will clone it locally, create a topic branch, make the code change and finally push that change back up to GitHub.

```

$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
    [-delay(1000);-][+delay(3000);+] // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
    [-delay(1000);-][+delay(3000);+] // wait for a second
}

$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
* [new branch]      slow-blink -> slow-blink

```

- ① Clone our fork of the project locally
- ② Create a descriptive topic branch
- ③ Make our change to the code
- ④ Check that the change is good
- ⑤ Commit our change to the topic branch
- ⑥ Push our new topic branch back up to our GitHub fork

Now if we go back to our fork on GitHub, we can see that GitHub noticed that we pushed a new topic branch up and present us with a big green button to check out our changes and open a Pull Request to the original project.

You can alternatively go to the “Branches” page at <https://github.com/<user>/<project>/branches> to locate your branch and open a new Pull Request from there.

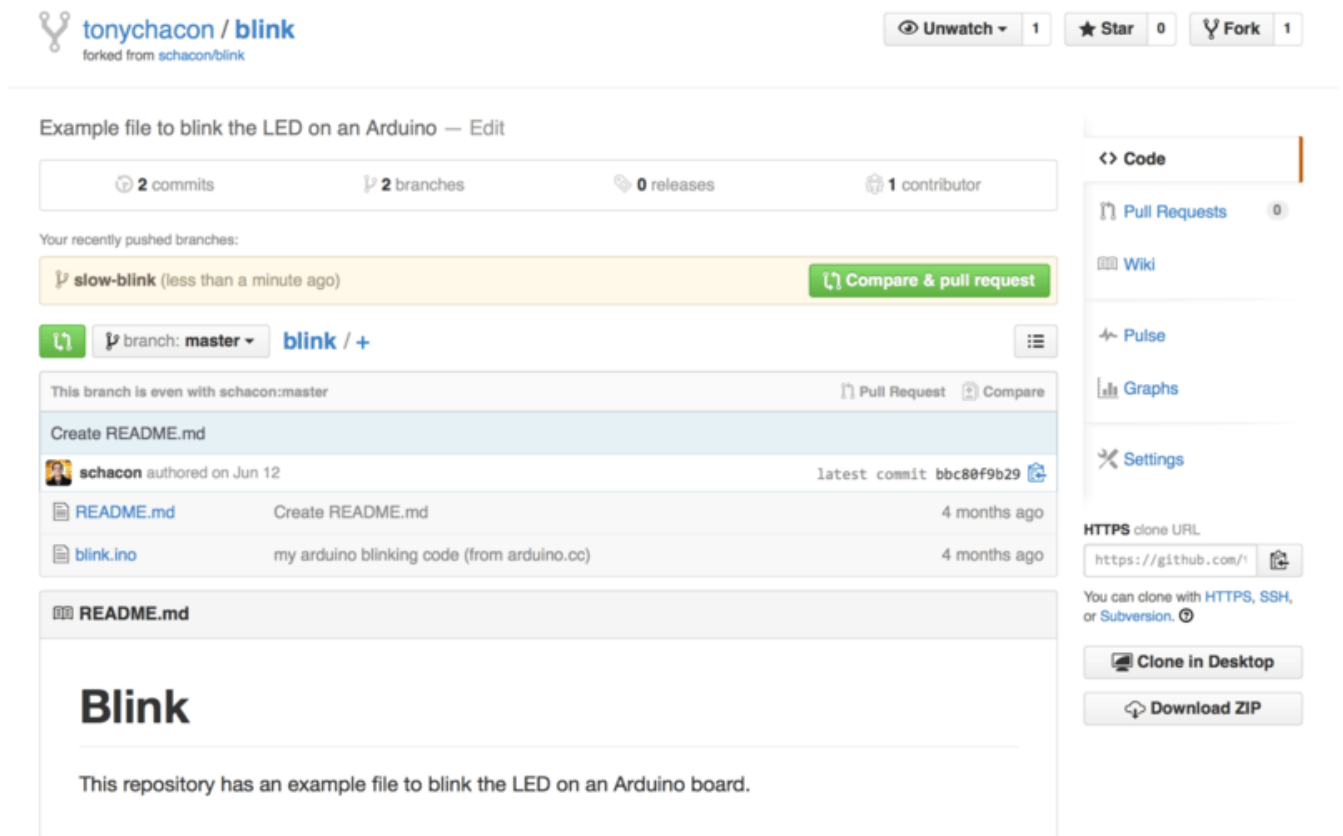


Figure 91. Pull Request button

If we click that green button, we’ll see a screen that allows us to create a title and description for the change we would like to request so the project owner has a good reason to consider it. It is generally a good idea to spend some effort making this description as useful as possible so the author knows why this is being suggested and why it would be a valuable change for them to accept.

We also see a list of the commits in our topic branch that are “ahead” of the `master` branch (in this case, just the one) and a unified diff of all the changes that will be made should this branch get merged by the project owner.

The screenshot shows the GitHub interface for creating a pull request. At the top, the repository is identified as 'tonychacon / blink' (forked from 'schacon/blink'). The current branch is 'tonychacon:slow-blink'.

The main content area has a title 'Three seconds is better' and a 'Write' tab selected. The description contains the text: 'Studies have shown that 3 seconds is a far better LED delay than 1 second.' followed by a URL: 'http://studies.example.com/optimal-led-delays.html'. A 'Create pull request' button is visible on the right side.

Below the description, it shows '1 commit', '1 file changed', '0 commit comments', and '1 contributor'. A commit history section shows a commit by 'tonychacon' titled 'three seconds is better' with hash 'db44c53' on Oct 01, 2014.

The diff view shows changes to 'blink.ino'. The diff highlights two lines that were added (green background):

```

21 + delay(3000); // wait for a second
22 + delay(3000); // wait for a second

```

Other lines show deletions (red background) and existing code.

Figure 92. Pull Request creation page

When you hit the *Create pull request* button on this screen, the owner of the project you forked will get a notification that someone is suggesting a change and will link to a page that has all of this information on it.

NOTE

Though Pull Requests are used commonly for public projects like this when the contributor has a complete change ready to be made, it's also often used in internal projects *at the beginning* of the development cycle. Since you can keep pushing to the topic branch even **after** the Pull Request is opened, it's often opened early and used as a way to iterate on work as a team within a context, rather than opened at the very end of the process.

Iterating on a Pull Request

At this point, the project owner can look at the suggested change and merge it, reject it or comment on it. Let's say that he likes the idea, but would prefer a slightly longer time for the light to be off than on.

Where this conversation may take place over email in the workflows presented in [Distribuirani Git](#), on GitHub this happens online. The project owner can review the unified diff and leave a comment by clicking on any of the lines.

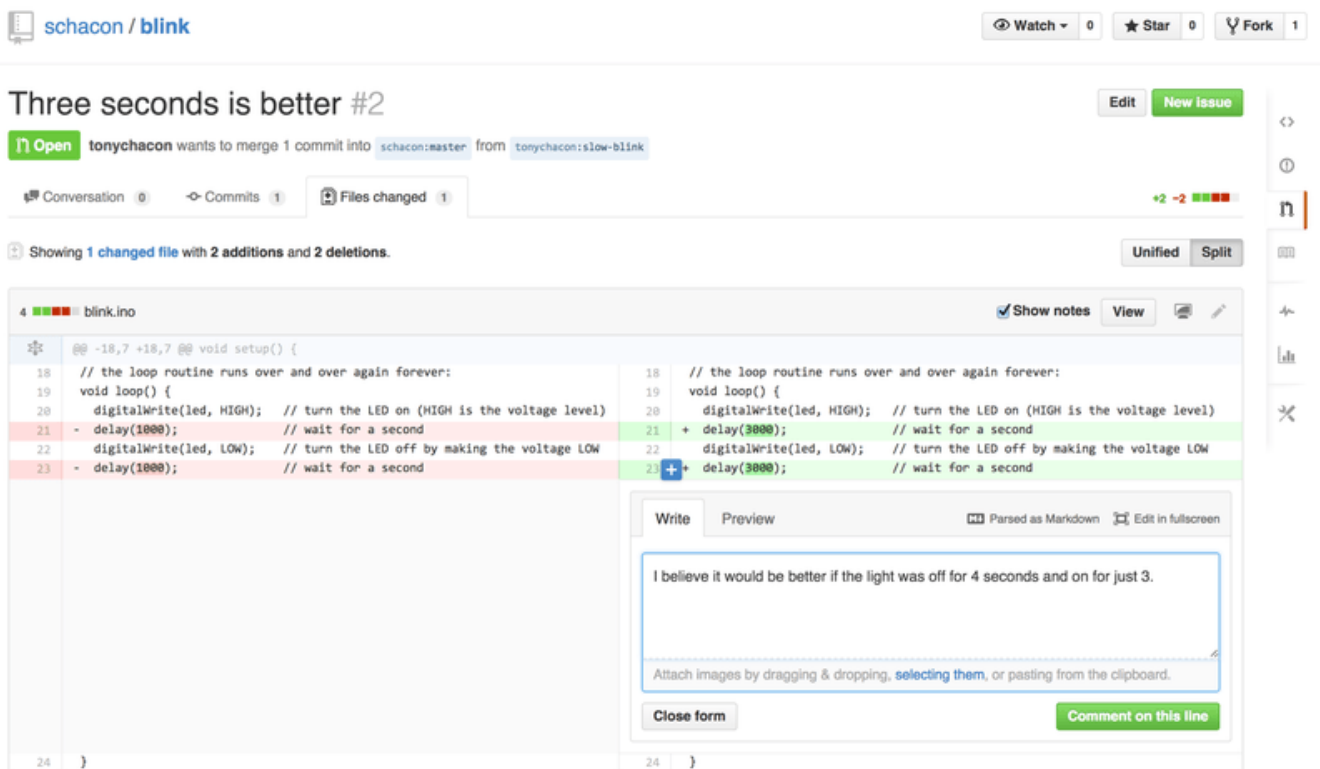


Figure 93. Comment on a specific line of code in a Pull Request

Once the maintainer makes this comment, the person who opened the Pull Request (and indeed, anyone else watching the repository) will get a notification. We'll go over customizing this later, but if he had email notifications turned on, Tony would get an email like this:



Figure 94. Comments sent as email notifications

Anyone can also leave general comments on the Pull Request. In [Pull Request discussion page](#) we can see an example of the project owner both commenting on a line of code and then leaving a general comment in the discussion section. You can see that the code comments are brought into the conversation as well.

Three seconds is better #2

Edit New Issue

Open tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1

+2 -2



tonychacon commented 6 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.

<http://studies.example.com/optimal-led-delays.html>

Labels

None yet

Milestone

No milestone

Assignee

No one—assign yourself

Notifications

Unsubscribe

You're receiving notifications because you commented.

2 participants



Lock pull request

three seconds is better

db44c53

schacon commented on the diff just now

blink.ino

View full changes

((6 lines not shown))		
22	22	digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
23	-	delay(1000); // wait for a second
	23	+ delay(3000); // wait for a second

schacon added a note just now

Owner

I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note



schacon commented just now

Owner


If you make that change, I'll be happy to merge this.

Figure 95. Pull Request discussion page


Now the contributor can see what they need to do in order to get their change accepted. Luckily this is also a very simple thing to do. Where over email you may have to re-roll your series and resubmit it to the mailing list, with GitHub you simply commit to the topic branch again and push.

If the contributor does that then the project owner will get notified again and when they visit the page they will see that it's been addressed. In fact, since a line of code changed that had a comment on it, GitHub notices that and collapses the outdated diff.

Three seconds is better #2


 **tonychacon** wants to merge 3 commits into `schacon:master` from `tonychacon:slow-blink`




Conversation 3 Commits 3 Files changed 1




 **tonychacon** commented 11 minutes ago

Studies have shown that 3 seconds is a far better LED delay than 1 second.


<http://studies.example.com/optimal-led-delays.html>


 three seconds is better db44c53


  **schacon** commented on an outdated diff 5 minutes ago  Show outdated diff




 **schacon** commented 5 minutes ago Owner  

If you make that change, I'll be happy to merge this.



 **tonychacon** added some commits 2 minutes ago

 longer off time 0c1f66f

 remove trailing whitespace ef4725c

 **tonychacon** commented 10 seconds ago  

I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?

 **This pull request can be automatically merged.**  **Merge pull request**

You can also merge branches on the [command line](#).

Figure 96. Pull Request final

An interesting thing to notice is that if you click on the “Files Changed” tab on this Pull Request, you’ll get the “unified” diff—that is, the total aggregate difference that would be introduced to your main branch if this topic branch was merged in. In `git diff` terms, it basically automatically shows you `git diff master...<branch>` for the branch this Pull Request is based on. See [Determining What Is Introduced](#) for more about this type of diff.

The other thing you’ll notice is that GitHub checks to see if the Pull Request merges cleanly and provides a button to do the merge for you on the server. This button only shows up if you have write access to the repository and a trivial merge is possible. If you click it GitHub will perform a “non-fast-forward” merge, meaning that even if the merge **could** be a fast-forward, it will still create a merge commit.

If you would prefer, you can simply pull the branch down and merge it locally. If you merge this branch into the `master` branch and push it to GitHub, the Pull Request will automatically be closed.

This is the basic workflow that most GitHub projects use. Topic branches are created, Pull Requests are opened on them, a discussion ensues, possibly more work is done on the branch and eventually the request is either closed or merged.

Not Only Forks

NOTE

It's important to note that you can also open a Pull Request between two branches in the same repository. If you're working on a feature with someone and you both have write access to the project, you can push a topic branch to the repository and open a Pull Request on it to the `master` branch of that same project to initiate the code review and discussion process. No forking necessary.

Advanced Pull Requests

Now that we've covered the basics of contributing to a project on GitHub, let's cover a few interesting tips and tricks about Pull Requests so you can be more effective in using them.

Pull Requests as Patches

It's important to understand that many projects don't really think of Pull Requests as queues of perfect patches that should apply cleanly in order, as most mailing list-based projects think of patch series contributions. Most GitHub projects think about Pull Request branches as iterative conversations around a proposed change, culminating in a unified diff that is applied by merging.

This is an important distinction, because generally the change is suggested before the code is thought to be perfect, which is far more rare with mailing list based patch series contributions. This enables an earlier conversation with the maintainers so that arriving at the proper solution is more of a community effort. When code is proposed with a Pull Request and the maintainers or community suggest a change, the patch series is generally not re-rolled, but instead the difference is pushed as a new commit to the branch, moving the conversation forward with the context of the previous work intact.

For instance, if you go back and look again at [Pull Request final](#), you'll notice that the contributor did not rebase his commit and send another Pull Request. Instead they added new commits and pushed them to the existing branch. This way if you go back and look at this Pull Request in the future, you can easily find all of the context of why decisions were made. Pushing the "Merge" button on the site purposefully creates a merge commit that references the Pull Request so that it's easy to go back and research the original conversation if necessary.

Keeping up with Upstream

If your Pull Request becomes out of date or otherwise doesn't merge cleanly, you will want to fix it so the maintainer can easily merge it. GitHub will test this for you and let you know at the bottom of every Pull Request if the merge is trivial or not.

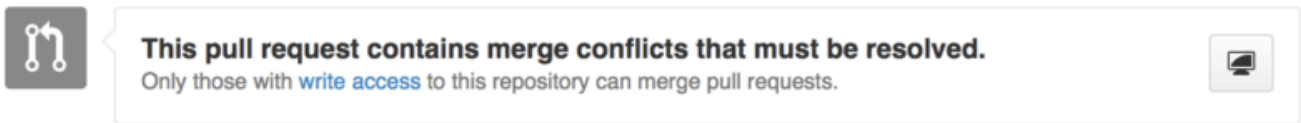


Figure 97. Pull Request does not merge cleanly

If you see something like [Pull Request does not merge cleanly](#), you'll want to fix your branch so that it turns green and the maintainer doesn't have to do extra work.

You have two main options in order to do this. You can either rebase your branch on top of whatever the target branch is (normally the `master` branch of the repository you forked), or you can merge the target branch into your branch.

Most developers on GitHub will choose to do the latter, for the same reasons we just went over in the previous section. What matters is the history and the final merge, so rebasing isn't getting you much other than a slightly cleaner history and in return is **far** more difficult and error prone.

If you want to merge in the target branch to make your Pull Request mergeable, you would add the original repository as a new remote, fetch from it, merge the main branch of that repository into your topic branch, fix any issues and finally push it back up to the same branch you opened the Pull Request on.

For example, let's say that in the "tonychacon" example we were using before, the original author made a change that would create a conflict in the Pull Request. Let's go through those steps.

```

$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
* [new branch]      master      -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
    into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
ef4725c..3c8d735  slower-blink -> slow-blink

```

- ① Add the original repository as a remote named “upstream”
- ② Fetch the newest work from that remote
- ③ Merge the main branch into your topic branch
- ④ Fix the conflict that occurred
- ⑤ Push back up to the same topic branch

Once you do that, the Pull Request will be automatically updated and re-checked to see if it merges cleanly.

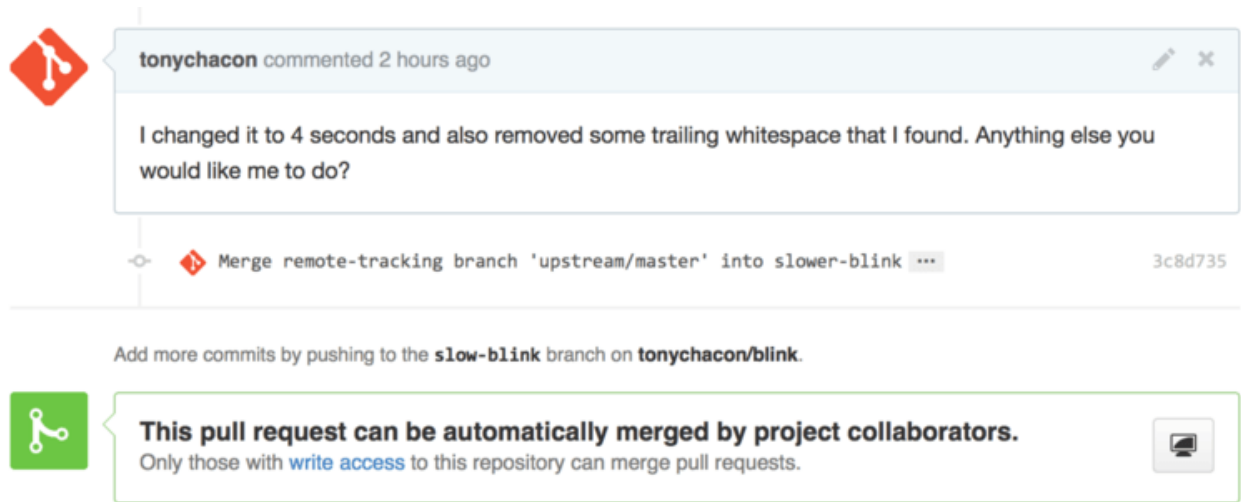


Figure 98. Pull Request now merges cleanly

One of the great things about Git is that you can do that continuously. If you have a very long-running project, you can easily merge from the target branch over and over again and only have to deal with conflicts that have arisen since the last time that you merged, making the process very manageable.

If you absolutely wish to rebase the branch to clean it up, you can certainly do so, but it is highly encouraged to not force push over the branch that the Pull Request is already opened on. If other people have pulled it down and done more work on it, you run into all of the issues outlined in [Nevarnosti ponovnega baziranja](#). Instead, push the rebased branch to a new branch on GitHub and open a brand new Pull Request referencing the old one, then close the original.

References

Your next question may be “How do I reference the old Pull Request?”. It turns out there are many, many ways to reference other things almost anywhere you can write in GitHub.

Let’s start with how to cross-reference another Pull Request or an Issue. All Pull Requests and Issues are assigned numbers and they are unique within the project. For example, you can’t have Pull Request #3 *and* Issue #3. If you want to reference any Pull Request or Issue from any other one, you can simply put #<num> in any comment or description. You can also be more specific if the Issue or Pull request lives somewhere else; write `username#<num>` if you’re referring to an Issue or Pull Request in a fork of the repository you’re in, or `username/repo#<num>` to reference something in another repository.

Let’s look at an example. Say we rebased the branch in the previous example, created a new pull request for it, and now we want to reference the old pull request from the new one. We also want to reference an issue in the fork of the repository and an issue in a completely different project. We can fill out the description just like [Cross references in a Pull Request](#).

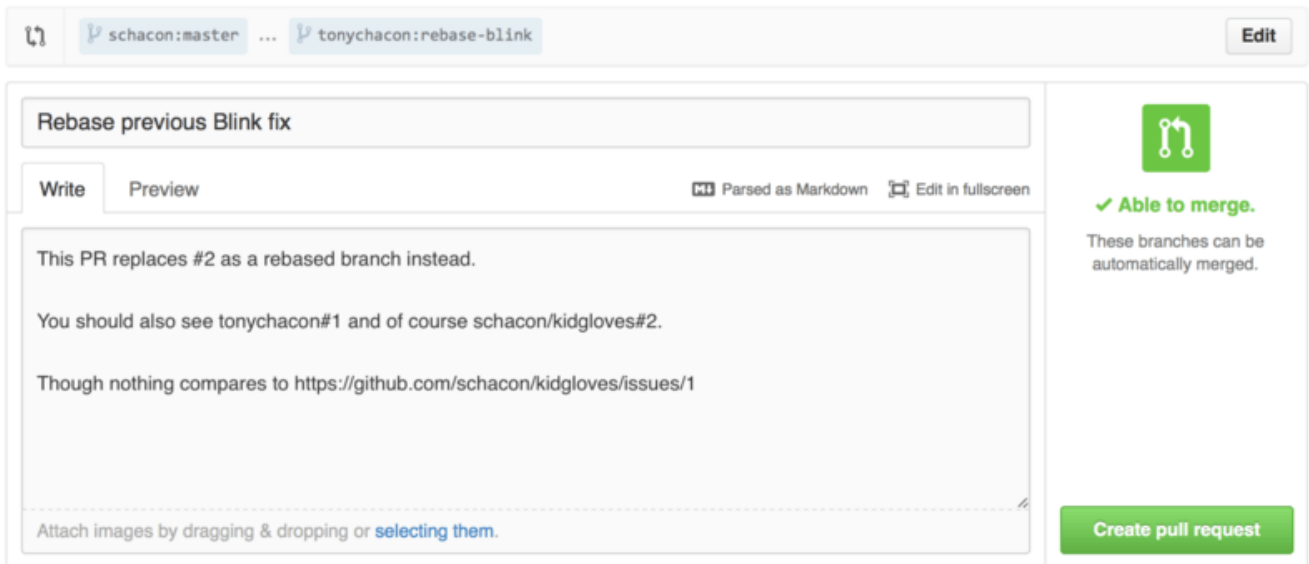


Figure 99. Cross references in a Pull Request.

When we submit this pull request, we'll see all of that rendered like [Cross references rendered in a Pull Request](#).

Rebase previous Blink fix #4

Open tonychacon wants to merge 2 commits into `schacon:master` from `tonychacon:rebase-blink`

Conversation 0 Commits 2 Files changed 1

tonychacon commented just now

This PR replaces [#2](#) as a rebase branch instead.

You should also see [tonychacon#1](#) and of course [schacon/kidgloves#2](#).

Though nothing compares to [schacon/kidgloves#1](#)

tonychacon added some commits 4 hours ago

- [three seconds is better](#) afe904a
- [remove trailing whitespace](#) a5a7751

Figure 100. Cross references rendered in a Pull Request.

Notice that the full GitHub URL we put in there was shortened to just the information needed.

Now if Tony goes back and closes out the original Pull Request, we can see that by mentioning it in the new one, GitHub has automatically created a traceback event in the Pull Request timeline. This means that anyone who visits this Pull Request and sees that it is closed can easily link back to the one that superseded it. The link will look something like [Cross references rendered in a Pull Request](#).

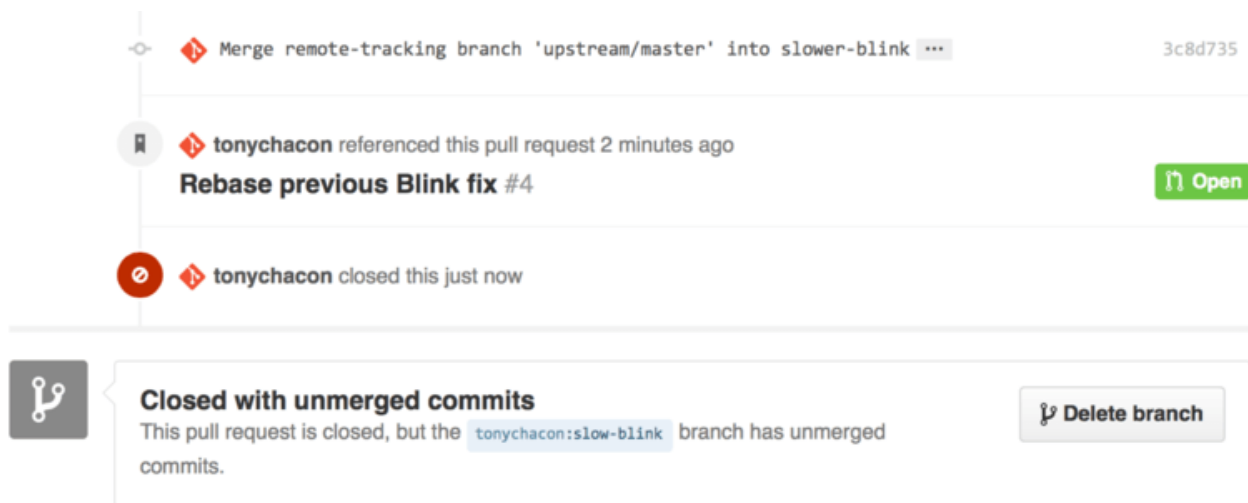


Figure 101. Cross references rendered in a Pull Request.

In addition to issue numbers, you can also reference a specific commit by SHA-1. You have to specify a full 40 character SHA, but if GitHub sees that in a comment, it will link directly to the commit. Again, you can reference commits in forks or other repositories in the same way you did with issues.

Markdown

Linking to other Issues is just the beginning of interesting things you can do with almost any text box on GitHub. In Issue and Pull Request descriptions, comments, code comments and more, you can use what is called “GitHub Flavored Markdown”. Markdown is like writing in plain text but which is rendered richly.

See [An example of Markdown as written and as rendered.](#) for an example of how comments or text can be written and then rendered using Markdown.

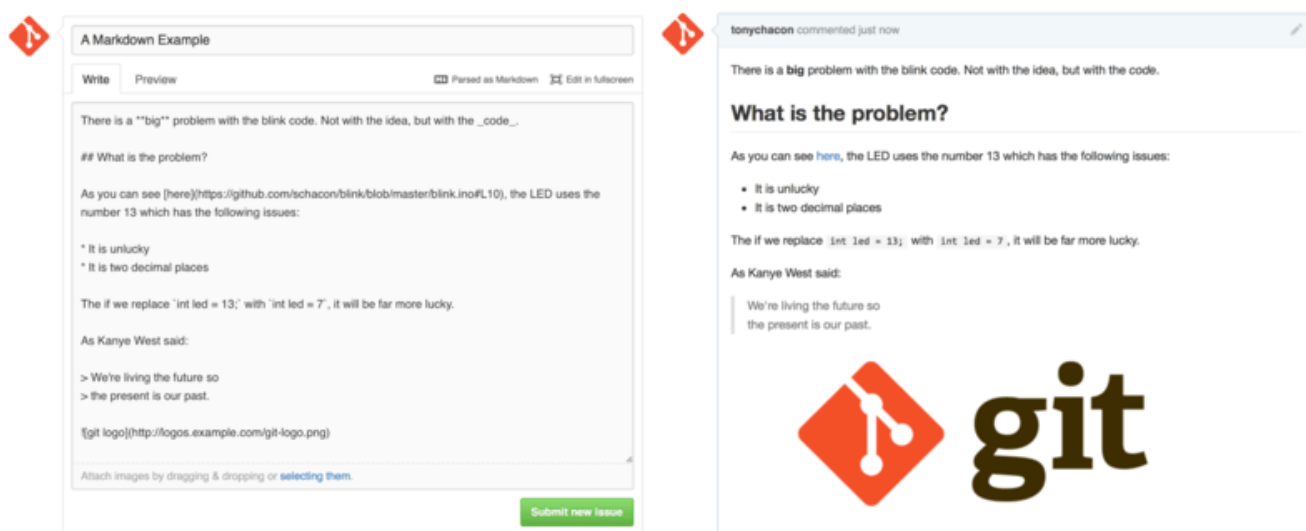


Figure 102. An example of Markdown as written and as rendered.

GitHub Flavored Markdown

The GitHub flavor of Markdown adds more things you can do beyond the basic Markdown syntax. These can all be really useful when creating useful Pull Request or

Issue comments or descriptions.

Task Lists

The first really useful GitHub specific Markdown feature, especially for use in Pull Requests, is the Task List. A task list is a list of checkboxes of things you want to get done. Putting them into an Issue or Pull Request normally indicates things that you want to get done before you consider the item complete.

You can create a task list like this:

- [X] Write the code
- [] Write all the tests
- [] Document the code

If we include this in the description of our Pull Request or Issue, we'll see it rendered like [Task lists rendered in a Markdown comment](#).



Figure 103. Task lists rendered in a Markdown comment.

This is often used in Pull Requests to indicate what all you would like to get done on the branch before the Pull Request will be ready to merge. The really cool part is that you can simply click the checkboxes to update the comment—you don't have to edit the Markdown directly to check tasks off.

What's more, GitHub will look for task lists in your Issues and Pull Requests and show them as metadata on the pages that list them out. For example, if you have a Pull Request with tasks and you look at the overview page of all Pull Requests, you can see how far done it is. This helps people break down Pull Requests into subtasks and helps other people track the progress of the branch. You can see an example of this in [Task list summary in the Pull Request list](#).



Figure 104. Task list summary in the Pull Request list.

These are incredibly useful when you open a Pull Request early and use it to track your progress through the implementation of the feature.

Code Snippets

You can also add code snippets to comments. This is especially useful if you want to present something that you *could* try to do before actually implementing it as a commit on your branch. This is also often used to add example code of what is not working or what this Pull Request could implement.

To add a snippet of code you have to “fence” it in backticks.

```
```java
for(int i=0 ; i < 5 ; i++)
{
 System.out.println("i is : " + i);
}
```
```

If you add a language name like we did there with *java*, GitHub will also try to syntax highlight the snippet. In the case of the above example, it would end up rendering like [Rendered fenced code example](#).



Figure 105. Rendered fenced code example.

Quoting

If you're responding to a small part of a long comment, you can selectively quote out of the other comment by preceding the lines with the `>` character. In fact, this is so common and so useful that there is a keyboard shortcut for it. If you highlight text in a comment that you want to directly reply to and hit the `r` key, it will quote that text in the comment box for you.

The quotes look something like this:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,

How big are these slings and in particular, these arrows?
```

Once rendered, the comment will look like [Rendered quoting example..](#)

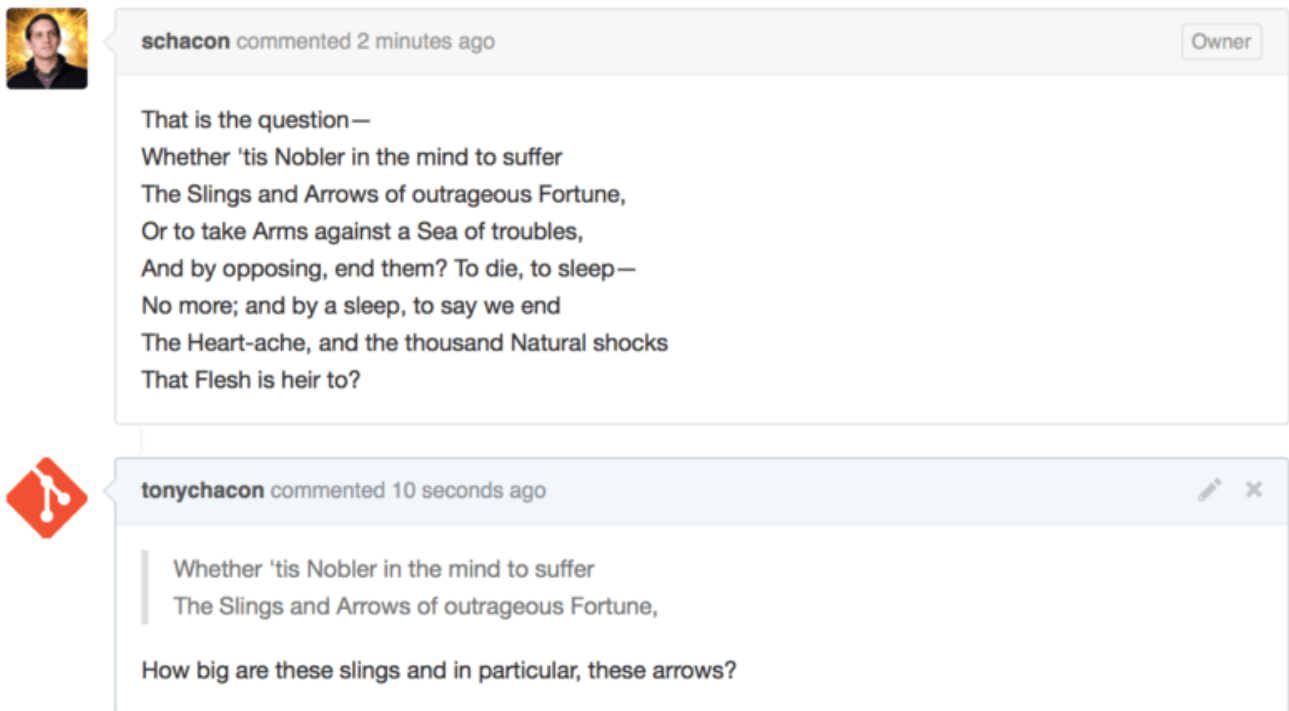


Figure 106. Rendered quoting example.

Emoji

Finally, you can also use emoji in your comments. This is actually used quite extensively in comments you see on many GitHub Issues and Pull Requests. There is even an emoji helper in GitHub. If you are typing a comment and you start with a `:` character, an autocompleter will help you find what you're looking for.

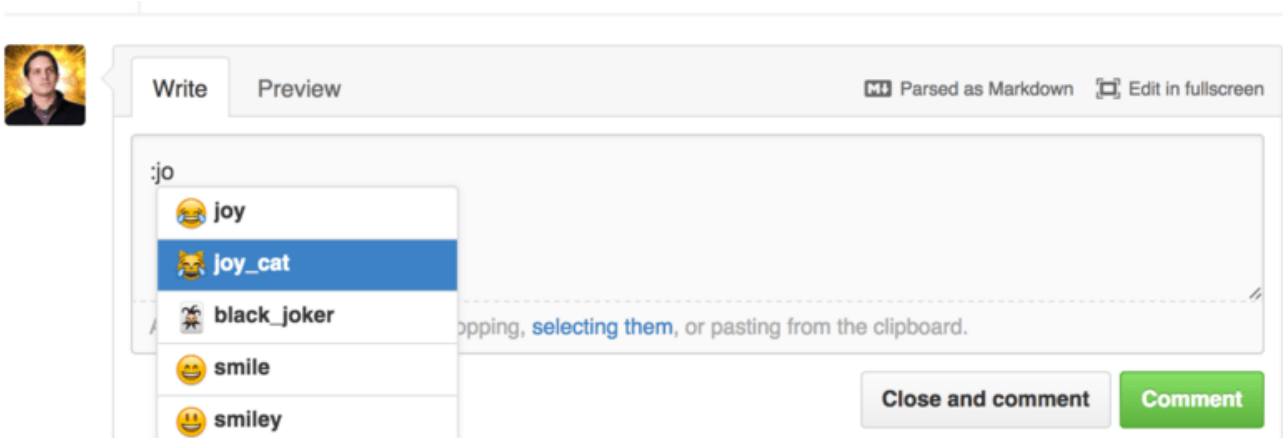


Figure 107. Emoji autocompleter in action.

Emojis take the form of `:<name>`: anywhere in the comment. For instance, you could write something like this:

```
I :eyes: that :bug: and I :cold_sweat:.  
:trophy: for :microscope: it.  
:+1: and :sparkles: on this :ship:, it's :fire::poop:!  
:clap::tada::panda_face:
```

When rendered, it would look something like [Heavy emoji commenting](#).

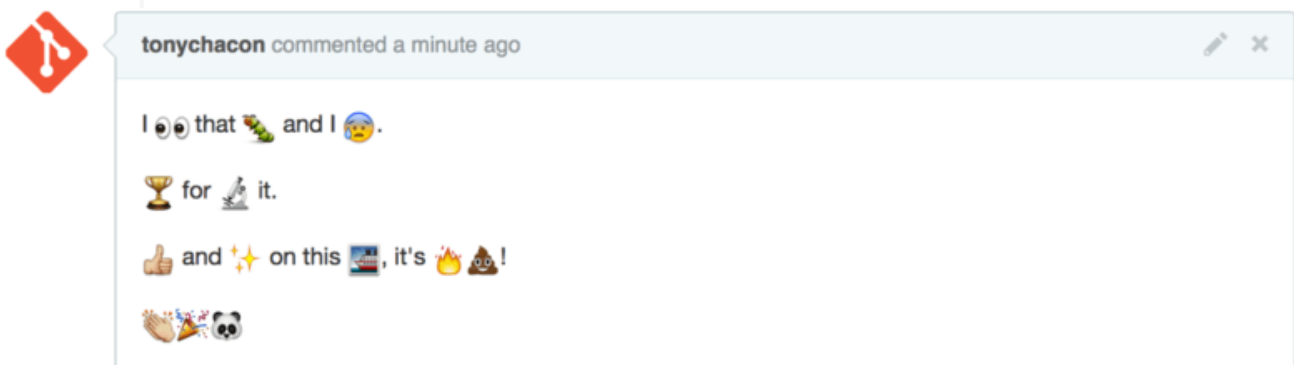


Figure 108. Heavy emoji commenting.

Not that this is incredibly useful, but it does add an element of fun and emotion to a medium that is otherwise hard to convey emotion in.

NOTE There are actually quite a number of web services that make use of emoji characters these days. A great cheat sheet to reference to find emoji that expresses what you want to say can be found at:

<http://www.emoji-cheat-sheet.com>

Images

This isn't technically GitHub Flavored Markdown, but it is incredibly useful. In addition to adding Markdown image links to comments, which can be difficult to find and embed URLs for, GitHub allows you to drag and drop images into text areas to embed them.

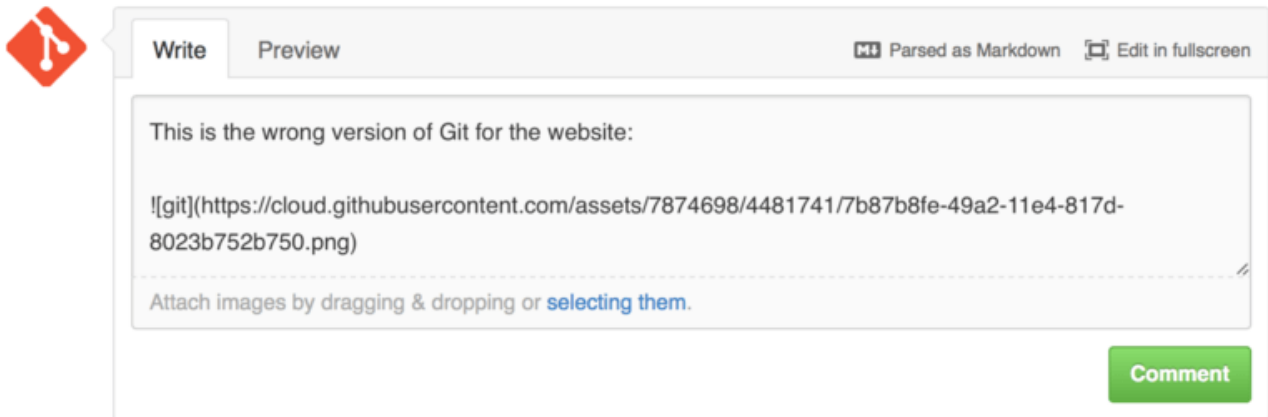
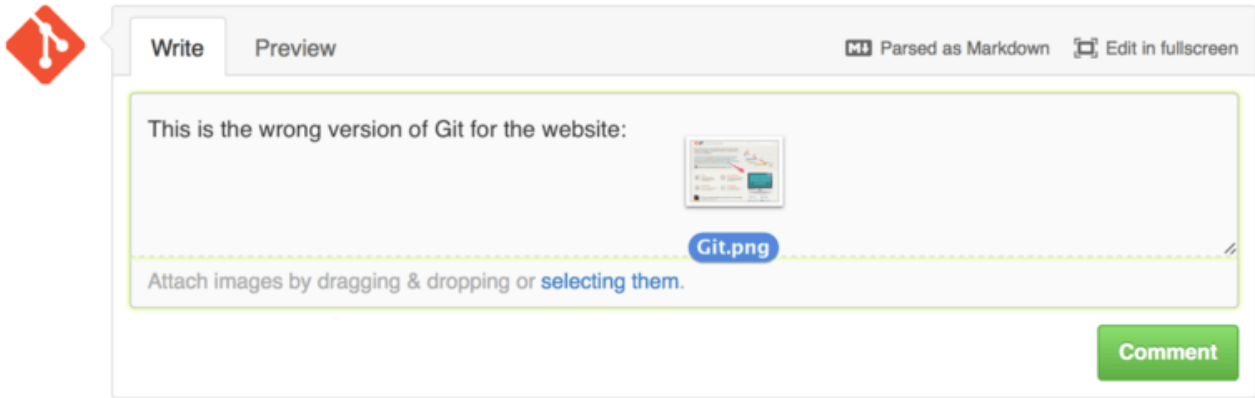


Figure 109. Drag and drop images to upload them and auto-embed them.

If you look back at [Cross references in a Pull Request.](#), you can see a small “Parsed as Markdown” hint above the text area. Clicking on that will give you a full cheat sheet of everything you can do with Markdown on GitHub.

Vzdrževanje projekta

Sedaj, ko smo domači s prispevanjem k projektu, pogledajmo na drugo stran: ustvarjanje, vzdrževanje in administriranje vašega lastnega projekta.

Ustvarjanje novega repozitorija

Ustvarimo nov repozitorij, da delite kodo vašega projekta. Začnete s klikom na gumb “New repository” na desni strani plošče ali z gumbom + na vrhni orodni vrstici zraven vašega uporabniškega imena kot je prikazano v [The “New repository” dropdown.](#)

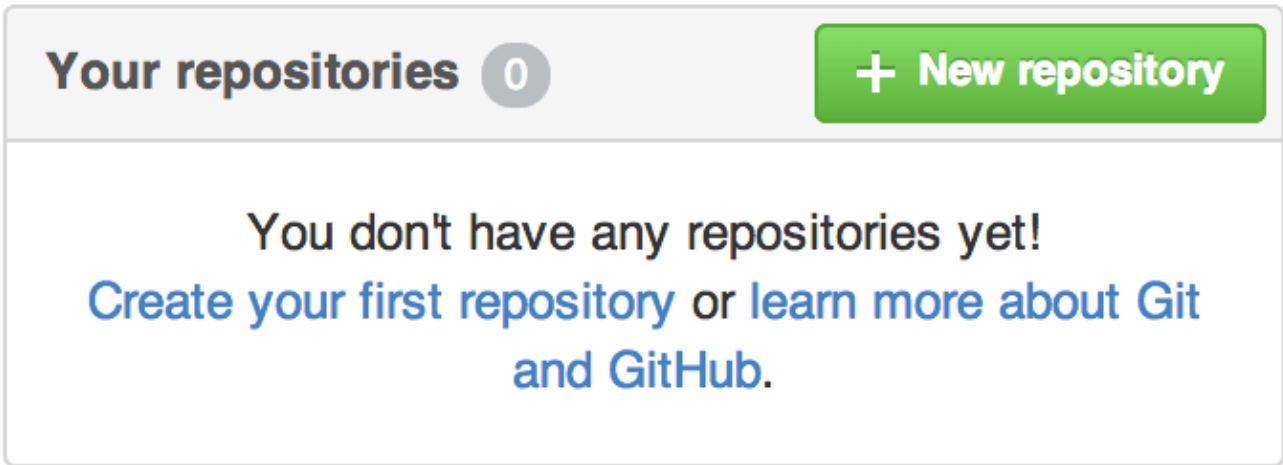


Figure 110. The “Your repositories” area.

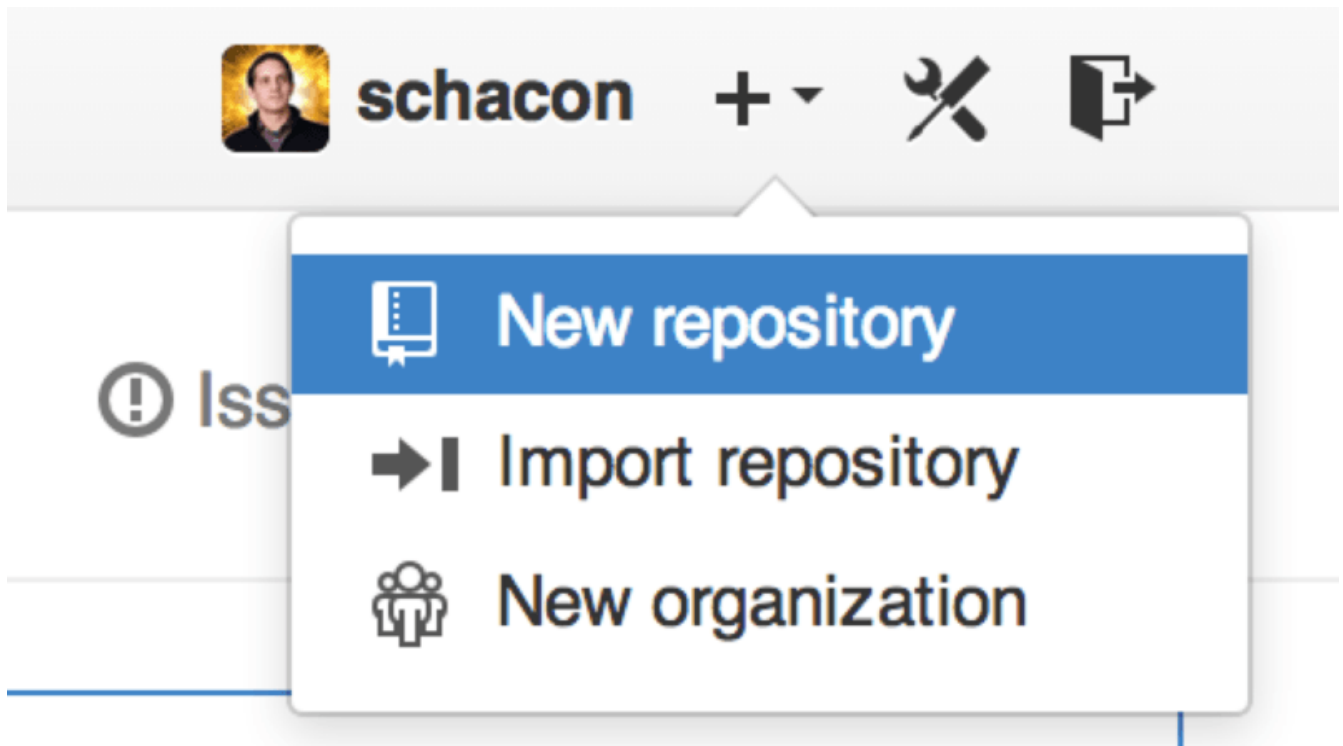


Figure 111. The “New repository” dropdown.

To vas popelje na obrazec “new repository”:

Owner **Repository name**

PUBLIC ben / iOSApp ✓

Great repository names are short and memorable. Need inspiration? How about **drunken-dubstep**.

Description (optional)

iOS project for our mobile group

Public
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with a README
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

Figure 112. The “new repository” form.

Vse kar morate resnično narediti tu je ponuditi ime projekta; preostanek polj je v celoti opcijski. Za sedaj, samo kliknite na gumb “Create Repository” in bum - imate nov repozitorij na GitHub-u imenovan `<user>/<project_name>`.

Odkar še nimate kode, vam bo GitHub prikazal navodila kako ustvariti popolnoma nov repozitorij Git ali povezavo z obstoječim projektom Git. Ne bomo poudarjali tega tu; če potrebujete osvežitev, preverite [Osnove Git](#).

Sedaj, ko je vaš projekt gostovan na GitHub-u, lahko date URL komurkoli s komur želite deliti vaš projekt. Vsak projekt na GitHub-u je dostopen preko HTTP kot `https://github.com/<user>/<project_name>` in preko SSH kot `git@github.com:<user>/<project_name>`. Git lahko ujame in potisne na oba od teh URL-jev, vendar sta osnovana na nadzoru dostopa na overilnicah uporabnika, ki se povezuje k njim.

NOTE

Pogostokrat je bolje deliti HTTP osnovan URL za javni projekt, saj uporabniku ni treba imeti računa GitHub, da dostopa do njega za kloniranje. Uporabniki bodo morali imeti račun in naložiti SSH ključ za dostop do vašega projekta, če jim date SSH URL. HTTP URL je tudi točno enak URL, ki bi ga prilepili v brskalnik, da tam pogledajo projekt.

Dodajanje sodelavcev

Če delate z drugimi ljudmi, katerim želite dati dostop pošiljanja, jih morate dodati kot “sodelavce”. Če se Ben, Jeff in Louise vsi prijavijo za račune na GitHub-u in jim želite dati dostop do vašega repozitorija, jih lahko dodate k vašemu projektu. To jim bo dalo dostop “push”, kar pomeni, da imajo tako pravice branja kot tudi pisanja projekta in repozitorija Git.

Kliknite na “Settings” povezavo na dnu orodne vrstice desne strani.

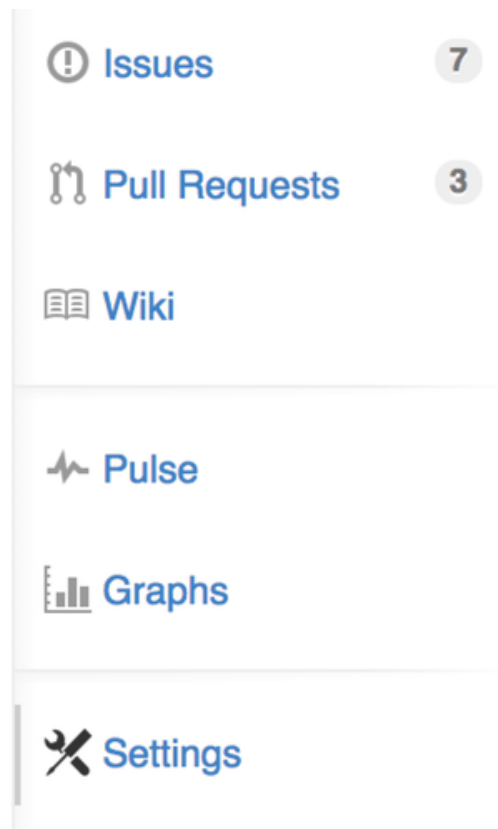


Figure 113. The repository settings link.

Nato izberite “Collaborators” iz menija na levi strani. Nato samo vpišite uporabniško ime v prosto in kliknite “Add collaborato.” To lahko ponovite kolikorokrat želite, da date dostop vsakomur želite. Če potrebujete odstraniti dostop, samo kliknite “X” na desni strani vrstice.

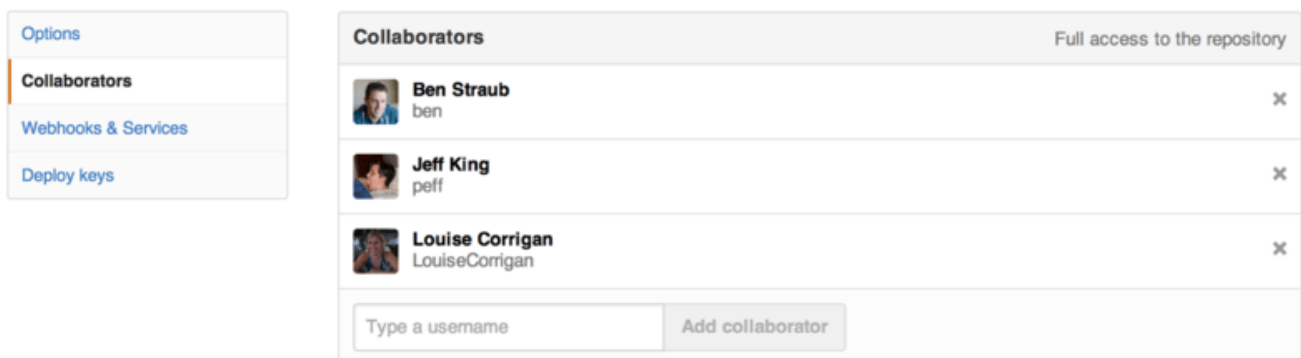


Figure 114. Repository collaborators.

Upravljanje zahtevkov potega

Sedaj, ko imate projekt z nekaj kode v njem in morda celo nekaj sodelavcev, ki imajo tudi dostop potiskanja, pojdimo skozi, kaj narediti, ko sami dobite zahtevek potega.

Zahtevki potegov lahko pridejo ali iz veje v fork-u vašega projekta ali lahko pridejo iz druge veje v istem repozitoriju. Edina razlika je, da tisti v fork-u so pogostokrat od ljudi, kjer ne morejo potiskati v svojo vejo in ne morejo potiskati v vašo, kjer z

internimi zahtevki potegov v splošnem obe strani lahko dostopata do veje.

Za te primere, predpostavimo, da ste “tonychacon” in ste ustvarili nov kodni projekt Arduino imenovan “fade”.

E-poštna obvestila

Nekdo pride zraven in naredi spremembe v vaši kodi in vam pošlje zahtevek potega. Morali bi dobiti e-pošto, ki vas obvesti o novem zahtevku potega in izgledati bi morali nekako kot [Email notification of a new Pull Request](#).

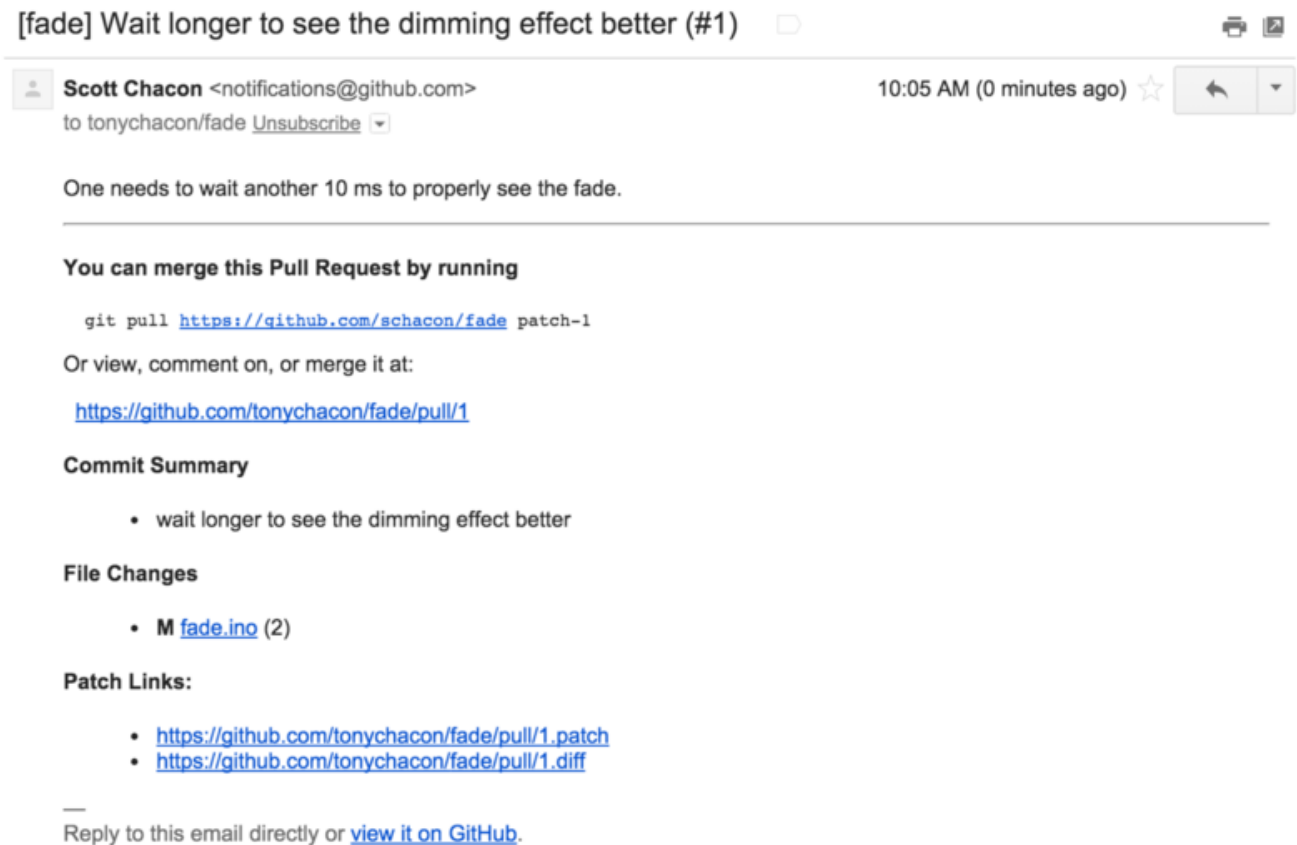


Figure 115. Email notification of a new Pull Request.

Tam je nekaj stvari za opaziti o tej e-pošti. Dalo vam bo majhen status razlik (diffstat) - seznam datotek, ki so se spremenile v zahtevku potega in za koliko. Da vam povežavo na zahtevek potega na GitHub-u. Da vam tudi nekaj URL-jev, ki jih lahko uporabite iz ukazne vrstice.

Če opazite vrstico, ki pravi `git pull <url> patch-1`, je to enostaven način za združevanje v oddaljeno vejo brez potrebe po dodajanju daljave. Šli smo skozi to hitro v [Checking Out Remote Branches](#). Če želite, lahko ustvarite in preklopite na tematsko vejo in nato pošete ta ukaz za združitev v spremembah zahtevka potega.

Drugi zanimivi URL-ji so `.diff` in `.patch`, ki so kot ste že ugotovili, ponujajo enoten diff in verzije popravka zahtevka potega. Tehnično bi lahko združili zahtevek potega dela z nečim takim:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

Sodelovanje na zahtevku potega

Kot smo pokrili v [The GitHub Flow](#), lahko imate sedaj pogovor z osebo, ki je odprla zahtevek potega. Lahko komentirate na določenih vrsticah kode, komentirate na celotnih pošiljanjih ali komentirate na samem celotnem zahtevku potega, povsod z uporabo GitHub Flavored Markdown-a.

Vsakič, ko nekdo drug komentira na zahtevku potega boste prejeli e-poštno obvestilo, da veste, da se dogaja aktivnost. Vsako bo imelo povezavo na zahtevek potega, kjer se aktivnost dogaja in lahko se tudi direktno odzovete na e-pošto, da komentirate na temi zahtevka potega.

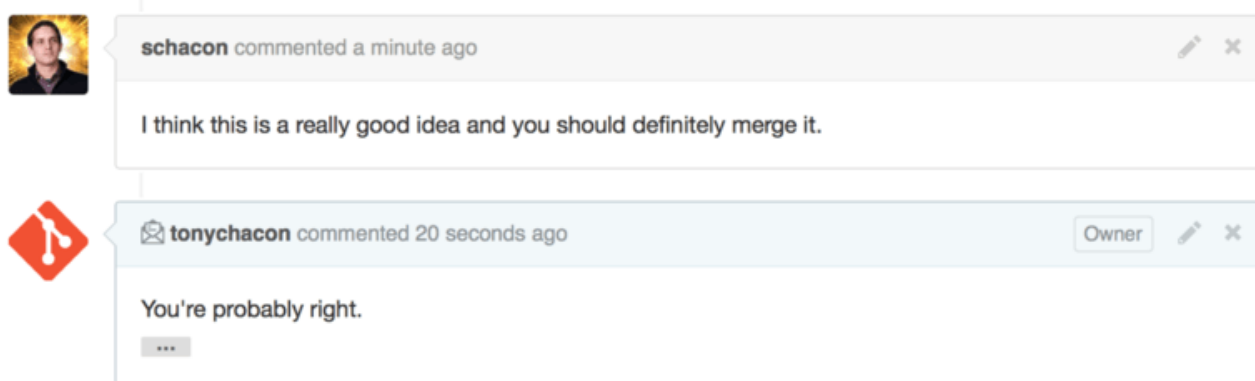


Figure 116. Responses to emails are included in the thread.

Ko je enkrat koda na mestu ki ga želite in želite združiti, lahko ali potegnete kodo ali jo združite lokalno, bodisi s sintakso `git pull <url> <branch>`, ki smo jo videli prej ali z dodajanjem fork-a kot daljave in ujeta ter združenja.

Če je združevanje trivialno, lahko samo tudi pritisnete gumb “Merge” na strani GitHub. To bo naredilo “non-fast-forward” združitev, ustvarilo pošiljanje združitve tudi če fast-forward združevanje ni bilo možno. To pomeni, da ne glede na kaj, vsakič ko pritisnete gumb združitev, se ustvari pošiljanje združitve. Kot lahko vidite v [Merge button and instructions for merging a Pull Request manually](#), vam GitHub da vse te informacije, če kliknete na povezavo namiga.

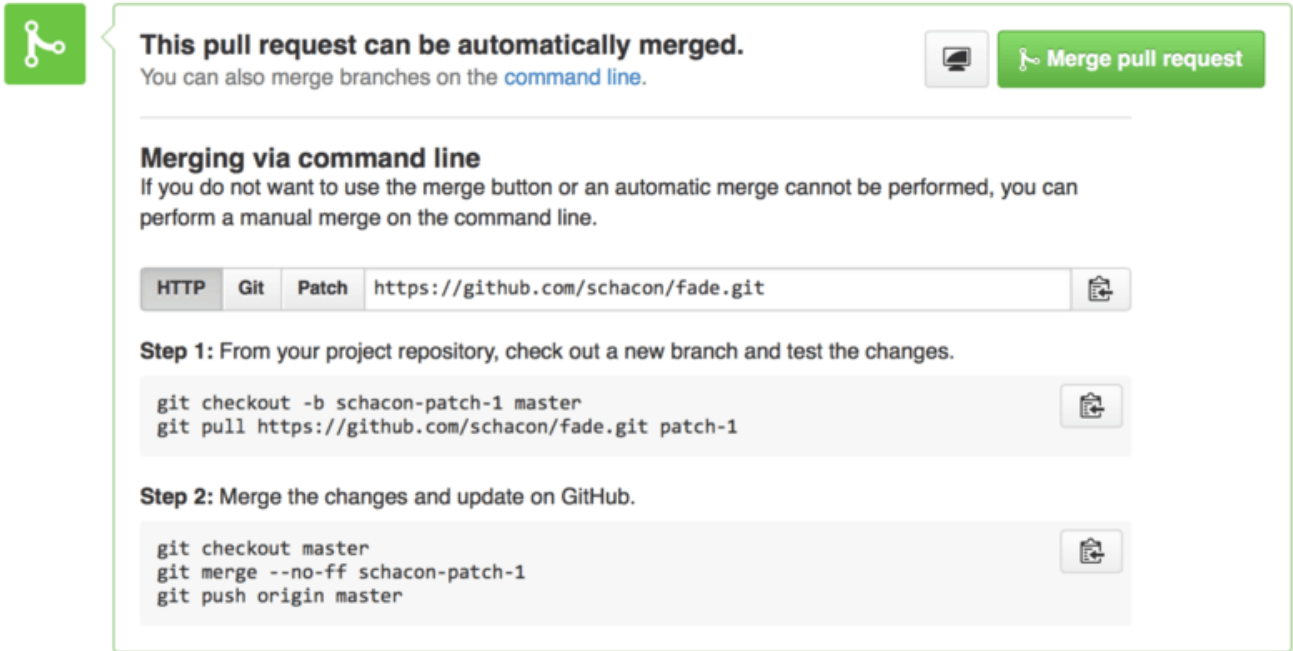


Figure 117. Merge button and instructions for merging a Pull Request manually.

Če se odločite, da ga ne želite združiti, lahko tudi samo zaprete zahtevek potega in oseba, ki je to odprta bo obveščena.

Pull Request Refs

If you're dealing with a **lot** of Pull Requests and don't want to add a bunch of remotes or do one time pulls every time, there is a neat trick that GitHub allows you to do. This is a bit of an advanced trick and we'll go over the details of this a bit more in [The Refspec](#), but it can be pretty useful.

GitHub actually advertises the Pull Request branches for a repository as sort of pseudo-branches on the server. By default you don't get them when you clone, but they are there in an obscured way and you can access them pretty easily.

To demonstrate this, we're going to use a low-level command (often referred to as a "plumbing" command, which we'll read about more in [Napeljava in porcelan](#)) called `ls-remote`. This command is generally not used in day-to-day Git operations but it's useful to show us what references are present on the server.

If we run this command against the "blink" repository we were using earlier, we will get a list of all the branches and tags and other references in the repository.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d HEAD
10d539600d86723087810ec636870a504f4fee4d refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3 refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1 refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c refs/pull/4/merge
```

Of course, if you're in your repository and you run `git ls-remote origin` or whatever remote you want to check, it will show you something similar to this.

If the repository is on GitHub and you have any Pull Requests that have been opened, you'll get these references that are prefixed with `refs/pull/`. These are basically branches, but since they're not under `refs/heads/` you don't get them normally when you clone or fetch from the server — the process of fetching ignores them normally.

There are two references per Pull Request - the one that ends in `/head` points to exactly the same commit as the last commit in the Pull Request branch. So if someone opens a Pull Request in our repository and their branch is named `bug-fix` and it points to commit `a5a775`, then in **our** repository we will not have a `bug-fix` branch (since that's in their fork), but we *will* have `pull/<pr#>/head` that points to `a5a775`. This means that we can pretty easily pull down every Pull Request branch in one go without having to add a bunch of remotes.

Now, you could do something like fetching the reference directly.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
* branch refs/pull/958/head -> FETCH_HEAD
```

This tells Git, “Connect to the `origin` remote, and download the ref named `refs/pull/958/head`.” Git happily obeys, and downloads everything you need to construct that ref, and puts a pointer to the commit you want under `.git/FETCH_HEAD`. You can follow that up with `git merge FETCH_HEAD` into a branch you want to test it in, but that merge commit message looks a bit weird. Also, if you're reviewing a **lot** of pull requests, this gets tedious.

There's also a way to fetch *all* of the pull requests, and keep them up to date whenever you connect to the remote. Open up `.git/config` in your favorite editor, and look for the `origin` remote. It should look a bit like this:

```
[remote "origin"]
url = https://github.com/libgit2/libgit2
fetch = +refs/heads/*:refs/remotes/origin/*
```

That line that begins with `fetch =` is a “refspec.” It’s a way of mapping names on the remote with names in your local `.git` directory. This particular one tells Git, “the things on the remote that are under `refs/heads` should go in my local repository under `refs/remotes/origin`.” You can modify this section to add another refspec:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

That last line tells Git, “All the refs that look like `refs/pull/123/head` should be stored locally like `refs/remotes/origin/pr/123`.” Now, if you save that file, and do a `git fetch`:

```
$ git fetch
# ...
* [new ref]      refs/pull/1/head -> origin/pr/1
* [new ref]      refs/pull/2/head -> origin/pr/2
* [new ref]      refs/pull/4/head -> origin/pr/4
# ...
```

Now all of the remote pull requests are represented locally with refs that act much like tracking branches; they’re read-only, and they update when you do a fetch. This makes it super easy to try the code from a pull request locally:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

The eagle-eyed among you would note the `head` on the end of the remote portion of the refspec. There’s also a `refs/pull/#/merge` ref on the GitHub side, which represents the commit that would result if you push the “merge” button on the site. This can allow you to test the merge before even hitting the button.

Pull Requests on Pull Requests

Not only can you open Pull Requests that target the `main` or `master` branch, you can actually open a Pull Request targeting any branch in the network. In fact, you can even target another Pull Request.

If you see a Pull Request that is moving in the right direction and you have an idea for a change that depends on it or you’re not sure is a good idea, or you just don’t have push access to the target branch, you can open a Pull Request directly to it.

When you go to open a Pull Request, there is a box at the top of the page that specifies which branch you’re requesting to pull to and which you’re requesting to pull from. If you hit the “Edit” button at the right of that box you can change not only the

branches but also which fork.

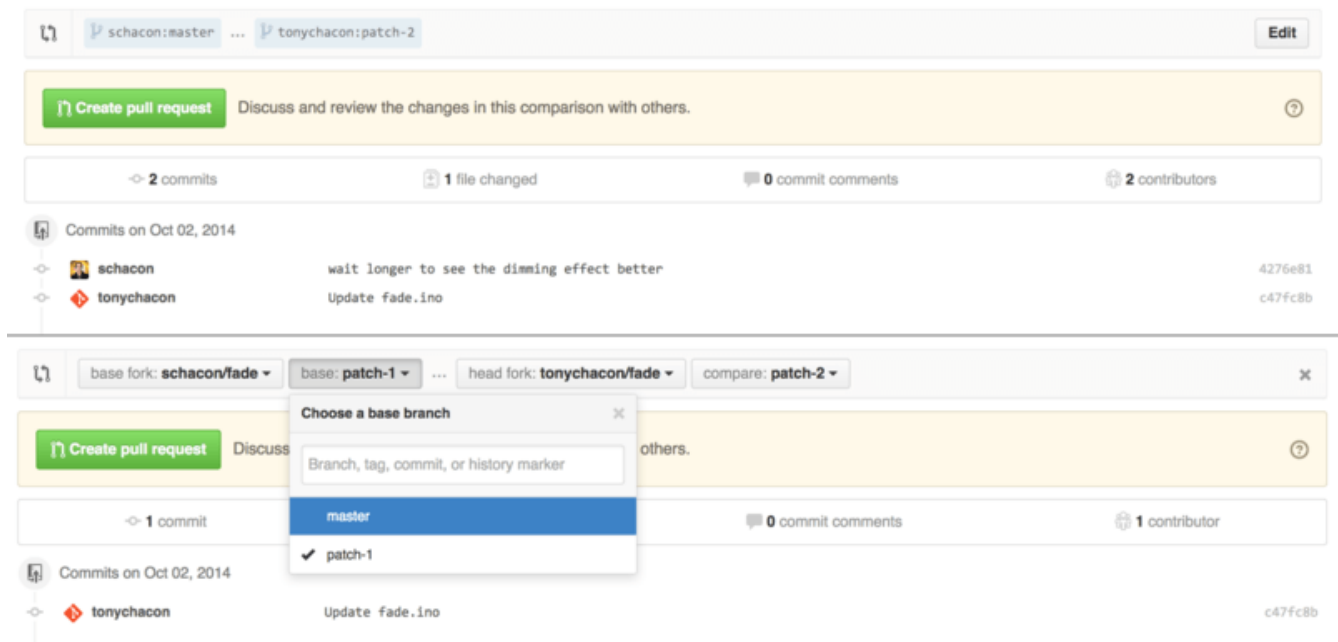


Figure 118. Manually change the Pull Request target fork and branch.

Here you can fairly easily specify to merge your new branch into another Pull Request or another fork of the project.

Mentions and Notifications

GitHub also has a pretty nice notifications system built in that can come in handy when you have questions or need feedback from specific individuals or teams.

In any comment you can start typing a @ character and it will begin to autocomplete with the names and usernames of people who are collaborators or contributors in the project.

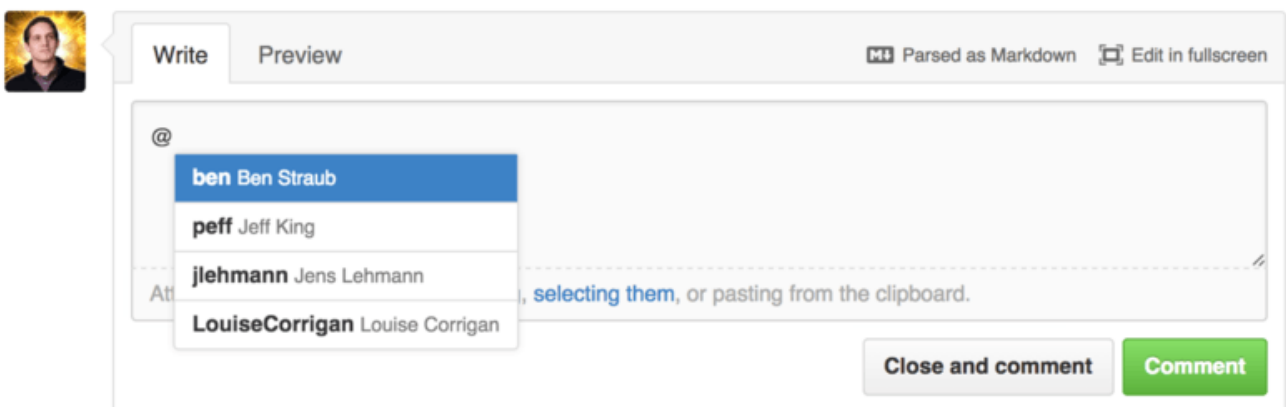


Figure 119. Start typing @ to mention someone.

You can also mention a user who is not in that dropdown, but often the autocompleter can make it faster.

Once you post a comment with a user mention, that user will be notified. This means that this can be a really effective way of pulling people into conversations rather than

making them poll. Very often in Pull Requests on GitHub people will pull in other people on their teams or in their company to review an Issue or Pull Request.

If someone gets mentioned on a Pull Request or Issue, they will be “subscribed” to it and will continue getting notifications any time some activity occurs on it. You will also be subscribed to something if you opened it, if you’re watching the repository or if you comment on something. If you no longer wish to receive notifications, there is an “Unsubscribe” button on the page you can click to stop receiving updates on it.

Notifications



You're receiving notifications
because you commented.

Figure 120. Unsubscribe from an Issue or Pull Request.

The Notifications Page

When we mention “notifications” here with respect to GitHub, we mean a specific way that GitHub tries to get in touch with you when events happen and there are a few different ways you can configure them. If you go to the “Notification center” tab from the settings page, you can see some of the options you have.

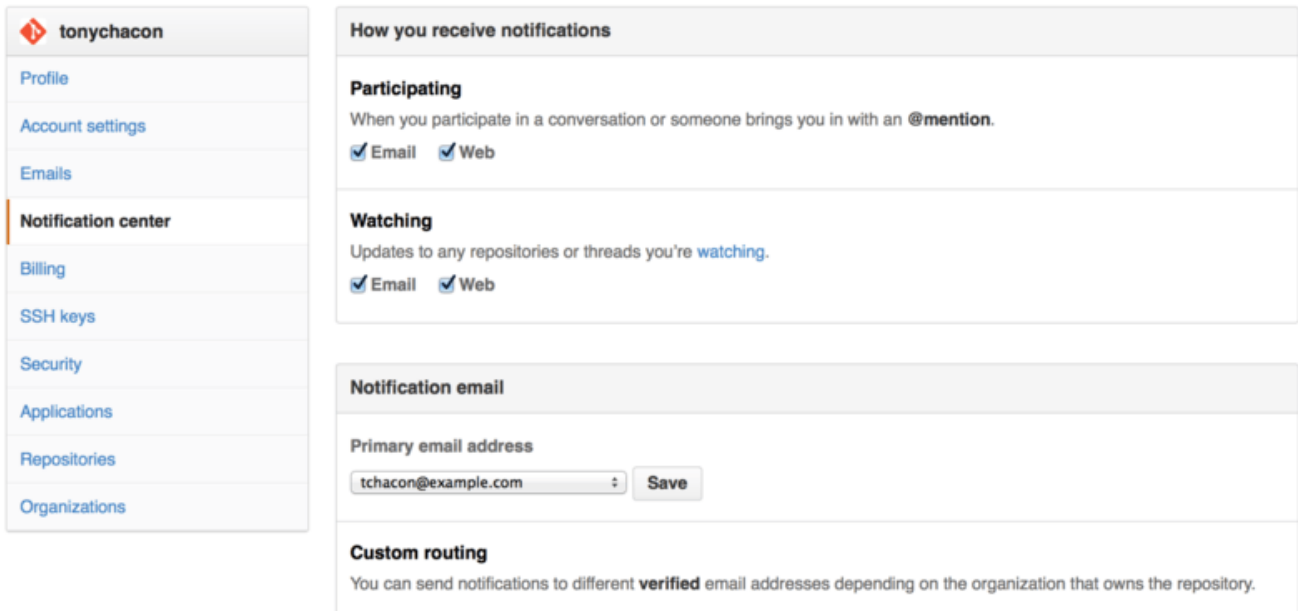


Figure 121. Notification center options.

The two choices are to get notifications over “Email” and over “Web” and you can choose either, neither or both for when you actively participate in things and for activity on repositories you are watching.

Web Notifications

Web notifications only exist on GitHub and you can only check them on GitHub. If you have this option selected in your preferences and a notification is triggered for you, you will see a small blue dot over your notifications icon at the top of your screen as seen in [Notification center](#).

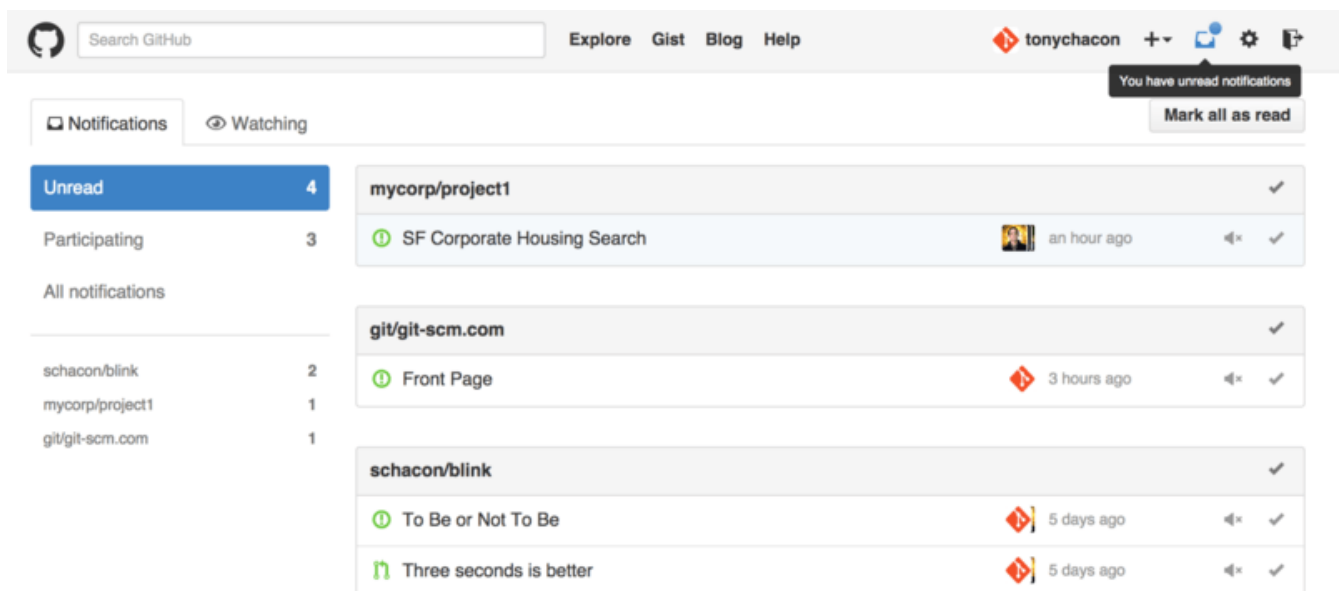


Figure 122. Notification center.

If you click on that, you will see a list of all the items you have been notified about, grouped by project. You can filter to the notifications of a specific project by clicking on

its name in the left hand sidebar. You can also acknowledge the notification by clicking the checkmark icon next to any notification, or acknowledge *all* of the notifications in a project by clicking the checkmark at the top of the group. There is also a mute button next to each checkmark that you can click to not receive any further notifications on that item.

All of these tools are very useful for handling large numbers of notifications. Many GitHub power users will simply turn off email notifications entirely and manage all of their notifications through this screen.

Email Notifications

Email notifications are the other way you can handle notifications through GitHub. If you have this turned on you will get emails for each notification. We saw examples of this in [Comments sent as email notifications](#) and [Email notification of a new Pull Request](#). The emails will also be threaded properly, which is nice if you're using a threading email client.

There is also a fair amount of metadata embedded in the headers of the emails that GitHub sends you, which can be really helpful for setting up custom filters and rules.

For instance, if we look at the actual email headers sent to Tony in the email shown in [Email notification of a new Pull Request](#), we will see the following among the information sent:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>, ...
X-GitHub-Recipient-Address: tchacon@example.com
```

There are a couple of interesting things here. If you want to highlight or re-route emails to this particular project or even Pull Request, the information in **Message-ID** gives you all the data in `<user>/<project>/<type>/<id>` format. If this were an issue, for example, the `<type>` field would have been “issues” rather than “pull”.

The **List-Post** and **List-Unsubscribe** fields mean that if you have a mail client that understands those, you can easily post to the list or “Unsubscribe” from the thread. That would be essentially the same as clicking the “mute” button on the web version of the notification or “Unsubscribe” on the Issue or Pull Request page itself.

It's also worth noting that if you have both email and web notifications enabled and you read the email version of the notification, the web version will be marked as read as well if you have images allowed in your mail client.

Special Files

There are a couple of special files that GitHub will notice if they are present in your repository.

README

The first is the `README` file, which can be of nearly any format that GitHub recognizes as prose. For example, it could be `README`, `README.md`, `README.asciidoc`, etc. If GitHub sees a `README` file in your source, it will render it on the landing page of the project.

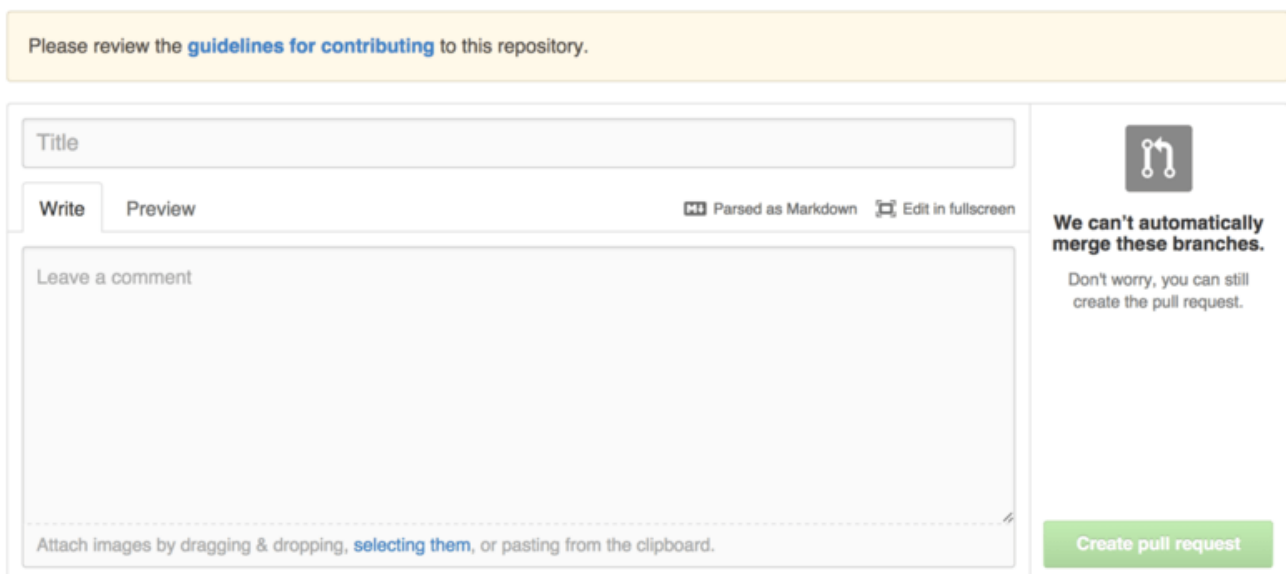
Many teams use this file to hold all the relevant project information for someone who might be new to the repository or project. This generally includes things like:

- What the project is for
- How to configure and install it
- An example of how to use it or get it running
- The license that the project is offered under
- How to contribute to it

Since GitHub will render this file, you can embed images or links in it for added ease of understanding.

CONTRIBUTING

The other special file that GitHub recognizes is the `CONTRIBUTING` file. If you have a file named `CONTRIBUTING` with any file extension, GitHub will show [Opening a Pull Request when a CONTRIBUTING file exists](#). when anyone starts opening a Pull Request.



The screenshot shows the GitHub interface for creating a pull request. At the top, a yellow banner contains the text: "Please review the [guidelines for contributing](#) to this repository." Below this is a form with a "Title" input field. Underneath the title field are two tabs: "Write" (selected) and "Preview". To the right of the tabs are two icons: "Parsed as Markdown" and "Edit in fullscreen". Below the tabs is a large text area with the placeholder "Leave a comment". At the bottom of the text area, there is a dashed line and the text: "Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard." On the right side of the form, there is a dark square icon with a white fork symbol. Below the icon, the text reads: "We can't automatically merge these branches. Don't worry, you can still create the pull request." At the bottom right of the form is a green button labeled "Create pull request".

Figure 123. Opening a Pull Request when a `CONTRIBUTING` file exists.

The idea here is that you can specify specific things you want or don't want in a Pull Request sent to your project. This way people may actually read the guidelines before

opening the Pull Request.

Project Administration

Generally there are not a lot of administrative things you can do with a single project, but there are a couple of items that might be of interest.

Changing the Default Branch

If you are using a branch other than “master” as your default branch that you want people to open Pull Requests on or see by default, you can change that in your repository’s settings page under the “Options” tab.

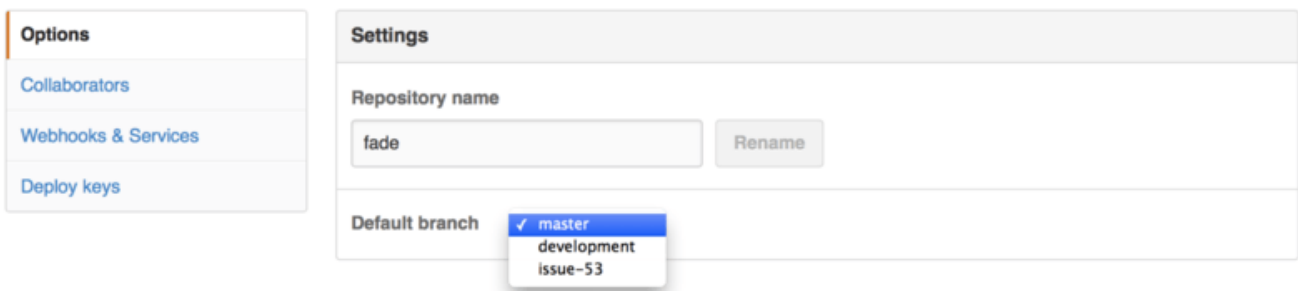


Figure 124. Change the default branch for a project.

Simply change the default branch in the dropdown and that will be the default for all major operations from then on, including which branch is checked out by default when someone clones the repository.

Transferring a Project

If you would like to transfer a project to another user or an organization in GitHub, there is a “Transfer ownership” option at the bottom of the same “Options” tab of your repository settings page that allows you to do this.

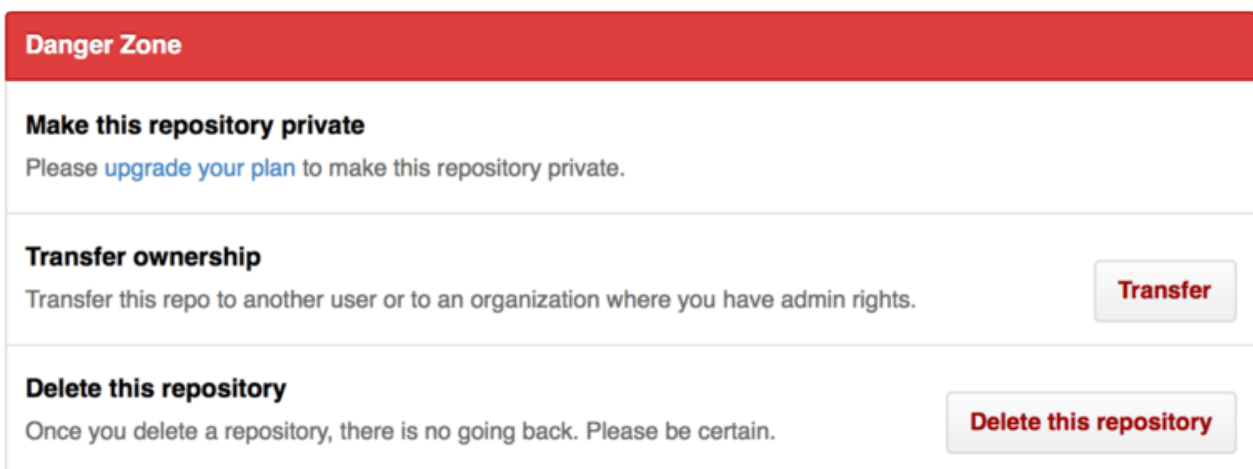


Figure 125. Transfer a project to another GitHub user or Organization.

This is helpful if you are abandoning a project and someone wants to take it over, or if your project is getting bigger and want to move it into an organization.

Not only does this move the repository along with all its watchers and stars to another place, it also sets up a redirect from your URL to the new place. It will also redirect clones and fetches from Git, not just web requests.

Upravljanje organizacije

Kot dodatek k računom enega uporabnika ima GitHub t.i. organizacije. Kot osebni računi, računi organizacij imajo imenski prostor, kjer vsi njihovi projekti obstajajo, vendar mnogo drugih stvari je različnih. Ti računi predstavljajo skupino ljudi z deljenim lastništvom projektov in na voljo je mnogo orodij za upravljanje podskupin teh ljudi. Običajno so te računi uporabljeni za skupine odprte kode (kot sta “perl” ali “rails”) ali podjetja (kot sta “google” ali “twitter”).

Osnove organizacij

Organizacija je precej enostavna za ustvariti; samo kliknite na ikono “+” na vrhu katerekoli GitHub strani in izberite “New organization” iz menija.

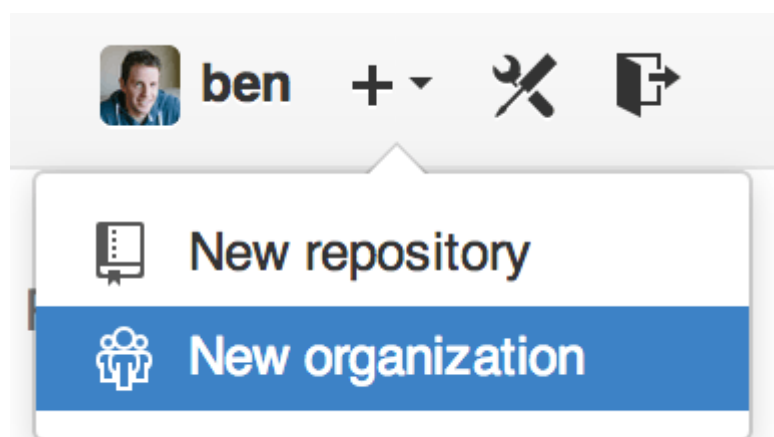


Figure 126. The “New organization” menu item.

Najprej boste morali poimenovati vašo organizacijo in ponuditi naslov e-pošte za glavno točko kontakta skupine. Nato lahko povabite uporabnike kot solastnike računa, če želite.

Sledite tem korakom in kmalu boste lastnik popolnoma nove organizacije. Kot osebni računi so organizacije brezplačne, če planirate tam vse shraniti kot odprto kodo.

Kot lastnik organizacije, ko forkate repozitorij, boste imeli izbiro forkanja v imenski prostor vaše organizacije. Ko ustvarite nove repozitorije, jih lahko ustvarite pod ali vašim osebnim računom ali pod katerokoli organizacijo, kjer ste lastnik. Tudi avtomatsko “watch” (gledate) katerikoli nov repozitorij ustvarjen pod temi organizacijami.

Kot v [Vaš avatar](#), lahko naložite avatar za vašo organizacijo, da jo nekoliko prilagodite po meri. Tudi kot osebni računi imate ciljno stran za organizacijo, ki izpisuje vse vaše repozitorije in je lahko vidna s strani ostalih ljudi.

Sedaj, pokrijmo nekatere stvari, ki so nekoliko drugačne z računom organizacije.

Ekipe

Organizacije so povezane z individualnimi ljudmi na način ekip, ki so enostavno skupine individualnih uporabniških računov in repozitorijev znotraj organizacije in kakršen način dostopa te ljudje imajo v teh repozitorijih.

Na primer, recimo, da ima vaše podjetje tri repozitorije: **frontend**, **backend** in **deployscripts**. Želite, da imajo vaši HTML/CSS/JavaScript razvijalci dostop do **frontend** in mogoče **backend** ter vaši operatorji imajo dostop do **backend** in **deployscripts**. Ekipe naredijo to enostavno brez potrebe po upravljanju sodelavcev za vsak posamezni repozitorij.

Stran organizacije prikazuje enostavno ploščo z vsemi repozitoriji, uporabniki in ekipami, ki so pod to organizacijo.

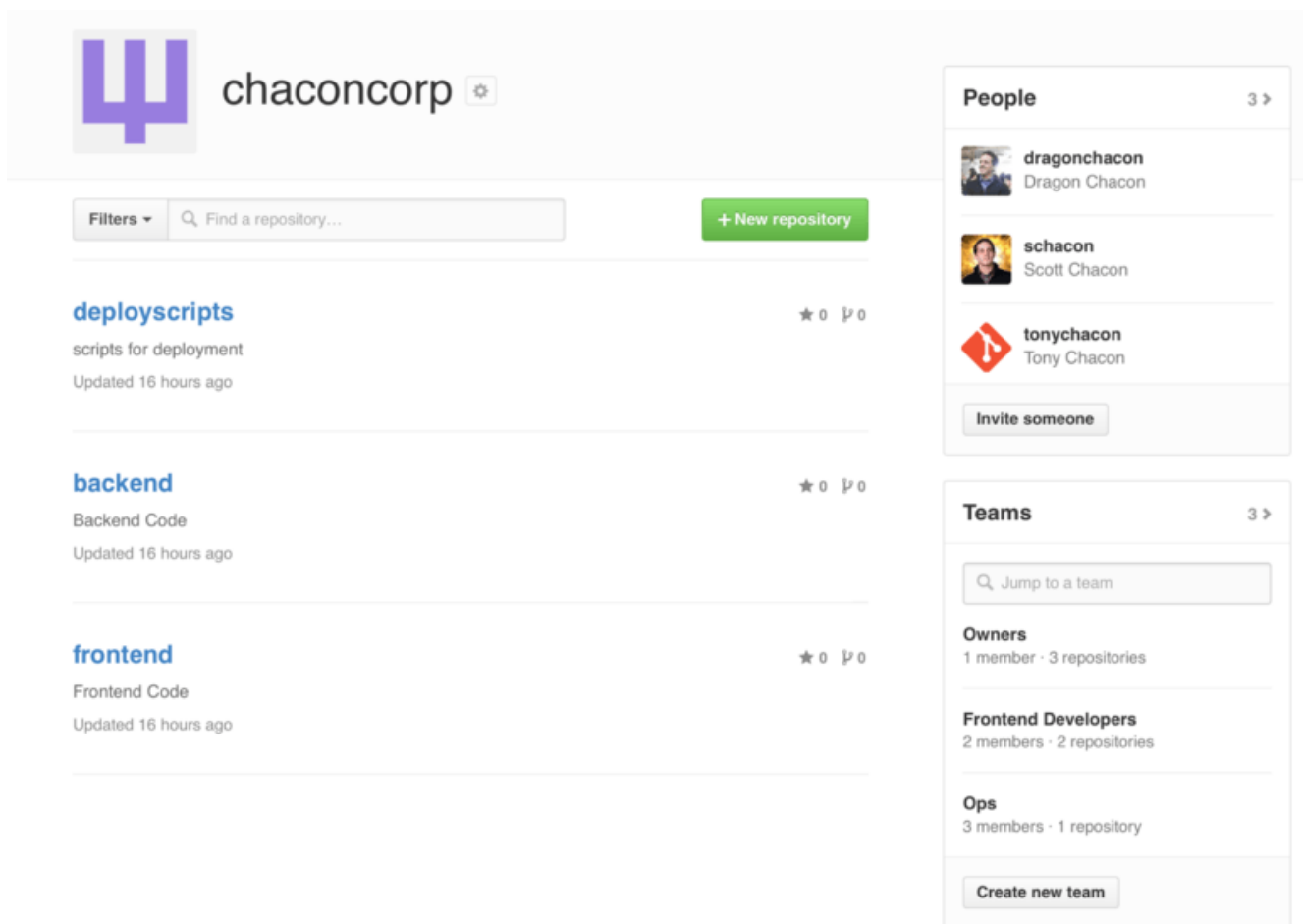


Figure 127. The Organization page.

Da upravljate vaše ekipe, lahko kliknete na stransko vrstico Teams na desni strani strani v [The Organization page](#). To vam bo prikazalo stran, kjer lahko dodate člane k ekipi, dodate repozitorije k ekipi ali upravljate nastavitve in kontrolo dostopa nivojev za ekipo. Vsaka ekipa ima lahko samo bralni, bralno/pisalni ali administrativni dostop do repozitorijev. Ta nivo lahko spremenite s klikom na gumb "Settings" v [The Team page](#).

The screenshot shows the GitHub interface for a team named 'Frontend Developers'. On the left, a summary box displays '2 MEMBERS' and '2 REPOSITORIES', with buttons for 'Leave' and 'Settings'. The main area is divided into 'Members' and 'Repositories' tabs. Under 'Members', two users are listed: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon), each with a 'Remove' button. A search bar at the top right says 'Invite or add users to team'. A note at the bottom states: 'This team grants Admin access: members can read from, push to, and add collaborators to the team's repositories.'

Figure 128. The Team page.

Ko povabite nekoga v ekipo, bo dobil e-pošto, ki ga obvešča, da je bil povabljen.

Kot dodatek ekipa `@mentions` (kot je `@acmecorp/frontend`) deluje precej podobno kot posamezni uporabniki, razen da so **vsi** člani ekipe potem naročeni na temo. To je uporabno, če želite pozornost od nekoga v ekipi vendar ne veste točno, kako vprašati.

Uporabnik lahko pripada kateremukoli številu ekip, tako da se omejite na samo kontrolo dostopa ekip. Posebne interesne ekipe kot `ux`, `css` ali `refactoring` so uporabne za določeno vrsto vprašanj in ostale kot so `legal` in `colorblind` za popolnoma različne vrste.

Revizijski dnevnik

Organizacije dajejo lastnikom tudi dostop do vseh informacij o tem, kaj se dogaja pod organizacijo. Lahko greste pod *Audit Log* zavihek in pogledate, kateri dogodki so se zgodili na nivoju organizacije, kdo jih je naredil in kje na svetu so bili narejeni.



| Recent events | | Filters | Search... |
|---------------|---|-------------------------|---|
| dragonchacon | added themselves to the chaconcorp/ops team | Organization membership | member 32 minutes ago |
| schacon | added themselves to the chaconcorp/ops team | Team management | member 33 minutes ago |
| tonychacon | invited dragonchacon to the chaconcorp organization | Repository management | member 16 hours ago |
| tonychacon | invited schacon to the chaconcorp organization | Billing updates | France org.invite_member 16 hours ago |
| tonychacon | gave chaconcorp/ops access to chaconcorp/backend | Hook activity | France team.add_repository 16 hours ago |
| tonychacon | gave chaconcorp/frontend-developers access to chaconcorp/backend | | France team.add_repository 16 hours ago |
| tonychacon | gave chaconcorp/frontend-developers access to chaconcorp/frontend | | France team.add_repository 16 hours ago |
| tonychacon | created the repository chaconcorp/deployscripts | | France repo.create 16 hours ago |
| tonychacon | created the repository chaconcorp/backend | | France repo.create 16 hours ago |

Figure 129. The Audit log.

Lahko tudi filtrirate na določen tip dogodkov, določena mesta ali določene ljudi.

Skriptni GitHub

Torej sedaj smo pokrili vse glavne lastnosti in poteke dela GitHub-a, vendar katerakoli večja skupina ali projekt bo imela prilagoditve po meri, ki jih želijo narediti ali zunanje storitve, ki jih želijo integrirati.

Na srečo za nas je GitHub resnično precej zmožen hekanja na mnoge načine. V tej sekciji bomo pokrili, kako uporabljati GitHub sistem kljuk in njegov API, da naredimo GitHub delati, kakor želimo.

Kljuke

Administracija sekcije kljuk in storitev repozitorija GitHub je najenostavnejši način, da ima GitHub interakcijo z zunanjimi sistemi.

Storitve

Najprej bomo pogledali storitve. Obe integraciji kljuka in storitve se lahko najde v sekciji Settings vašega repozitorija, kjer smo prej pogledali dodajanje sodelavcev in spreminjanje privzete veje za vaš projekt. Pod zavihkom “Webhooks and Services” boste videli nekaj kot je [Services and Hooks configuration section](#).

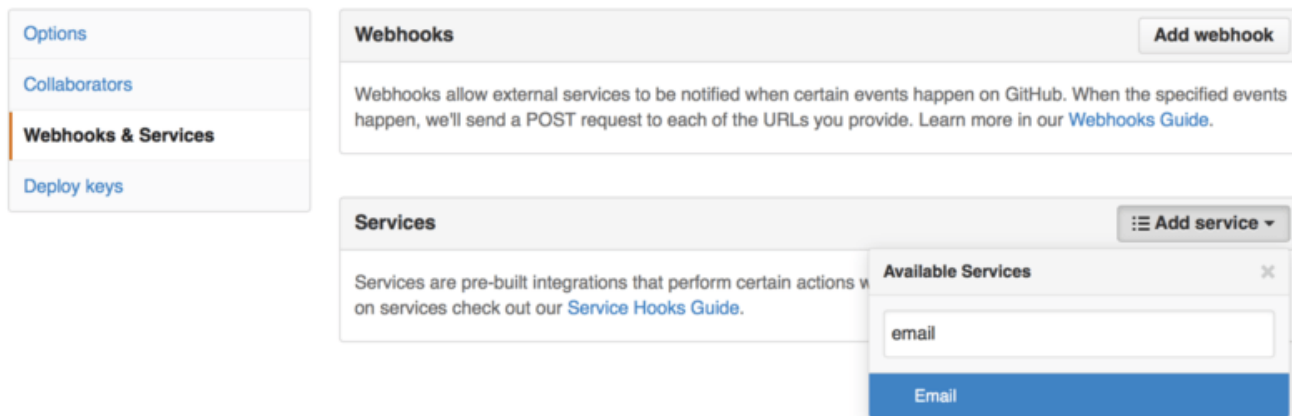


Figure 130. Services and Hooks configuration section.

Na voljo je ducat storitev, ki jih lahko izberete, večina od teh integracij v ostale komercialne in odprto kodne sisteme. Večina njih je za storitve stalne integracije, sledenja hroščev in težav, sisteme pogovornih sob in sisteme dokumentacije. Šli bomo skozi nastavitve zelo enostavne, e-poštne kljuka. Če izberete “email” iz padajočega menija “Add Service”, boste dobili nastavitveni zaslon, kot je [Email service configuration](#).

Options

Collaborators

Webhooks & Services

Deploy keys

Services / **Add Email**

Install Notes

1. `address` whitespace separated email addresses (at most two)
2. `secret` fills out the Approved header to automatically approve the message in a read-only or moderated mailing list.
3. `send_from_author` uses the commit author email address in the From address of the email.

Address

Secret

Send from author

Active
We will run this service when an event is triggered.

Add service

Figure 131. Email service configuration.

V tem primeru, če pritisnemo gumb “Add service”, bo naslov e-pošte, ki smo ga določili, dobil e-pošto vsakič, ko nekdo potisne v repozitorij. Storitve lahko poslušajo za veliko različnih tipov dogodkov, vendar večinoma samo poslušajo za dogodke potiskanja in nato naredijo nekaj s temi podatki.

Če je sistem, ki ga uporabljate in ga želite integrirati z GitHub-om, bi morali tu preveriti, da vidite, če je že obstoječa integracija storitev na voljo. Na primer, če uporabljate Jenkins za poganjanje testov na vaši bazi kode, lahko omogočite Jenkins vgrajeno storitveno integracijo za začetek poganjanja testov vsakič, kot nekdo potisne v vaš repozitorij.

Kljuje

Če potrebujete narediti nekaj bolj specifičnega ali želite integrirati storitev ali stran, ki ni vključena v ta seznam, lahko namesto tega uporabite bolj generične sisteme kljuk. Kljuje repozitorija GitHub so precej enostavne. Določite URL in GitHub bo poslal HTTP naročilo na ta URL na kateremkoli dogodku želite.

V splošnem je način, kako to deluje, da lahko nastavite majhno spletno storitev in da posluša za Github kljuko nalaganja in nato naredi nekaj s podatki, ko so prejeti.

Da omogočite kljuku, kliknite na gumb “Add webhook” v [Services and Hooks configuration section](#).. To vam bo prineslo stran, ki izgleda kot [Web hook configuration](#)..

Options

Collaborators

Webhooks & Services

Deploy keys

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

Payload URL *

Content type

Secret

Which events would you like to trigger this webhook?

Just the push event.

Send me **everything**.

Let me select individual events.

Active
We will deliver event details when this hook is triggered.

Add webhook

Figure 132. Web hook configuration.

Nastavitev za spletno kljuko je precej enostavna. V večini primerov enostavno vnesete URL in skriti ključ ter pritisnete “Add webhook”. Na voljo je nekaj opcij za katerimi dogodki želite, da vam GitHub pošlje nalaganje—privzeto je samo dobiti nalaganje za dogodek **push**, ko nekdo potisne novo kodo na katerokoli vejo vašega repozitorija.

Poglejmo majhen primer spletne storitve, ki jo morda želite nastaviti za upravljanje spletne kljuke. Uporabili bomo Ruby spletno ogrodje Sinatra, ker je precej jedrnato in vi bi morali uspeti enostavno pogledati, kaj delate.

Recimo, da želite dobiti e-pošto, če določena oseba potisne na določeno vejo vašega projekta in spremeni določeno datoteko. To bi lahko naredili precej enostavno s kodo takole:

```

require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
    branch == 'ref/heads/special-branch' &&
    files.include?('special-file.txt')




    Mail.deliver do
      from      'tchacon@example.com'
      to        'tchacon@example.com'
      subject   'Scott Changed the File'
      body      "ALARM"
    end
  end
end
end

```

Tu vzamemo JSON nalaganje, ki nam ga GitHub dostavi in pogledamo, kdo ga je potisnil, na katero vejo je potisnil in katere datoteke so bile dotaknjene v vseh pošiljanjih, ki so bila potisnjena. Nato to pregledamo napram našim kriterijem in pošljemo e-pošto, če se ujema.

Da razvijemo in pretestiramo nekaj takega imate lepo razvijalsko konzolo v istem zaslonu, kjer nastavite kljuko. Vidite lahko zadnjih nekaj dostav, ki jih je GitHub poskušal narediti za to spletno kljuko. Za vsako kljuko se lahko poglobite, ko je bila dostavljena, če je bila uspešna in telo ter glave za tako zahtevek in odziv. To naredi izjemno enostavno testirati in razhroščevati vaše kljuko.

Recent Deliveries

| | | | |
|---|--------------------------------------|---------------------|-----|
|  | 4aaee280-4e38-11e4-9bac-c130e992644b | 2014-10-07 17:40:41 | ... |
|  | aff20880-4e37-11e4-9089-35319435e08b | 2014-10-07 17:36:21 | ... |
|  | 90f37680-4e37-11e4-9508-227d13b2ccfc | 2014-10-07 17:35:29 | ... |

Request | Response **200** ⌚ Completed in 0.61 seconds. 🔄 Redeliver

Headers

```

Request URL: https://hooks.example.com/payload
Request method: POST
content-type: application/json
Expect:
User-Agent: GitHub-Hookshot/64a1910
X-GitHub-Delivery: 90f37680-4e37-11e4-9508-227d13b2ccfc
X-GitHub-Event: push

```

Payload

```

{
  "ref": "refs/heads/remove-whitespace",
  "before": "99d4fe5bfffaf827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bfffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonvchacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c"
    }
  ]
}

```

Figure 133. Web hook debugging information.

Za ostale odlične lastnosti tega je, da lahko ponovno dostavite katerokoli nalaganje za testiranje vaše storitve enostavno.

Za več informacij, kako ponovno pisati spletne kljuge in vse različne tipe dogodkov, ki jih lahko poslušate, pojdite na GitHub razvijalsko dokumentacijo na: <https://developer.github.com/webhooks/>

GitHub API

Spletne storitve in kljuge vam dajo način kako prejeti obvestila potiskanja o dogodkih, ki so se zgodili na vaših repozitorijih, vendar kaj pa če potrebujete več informacij o teh dogodkih? Kaj če potrebujete avtomatizirati nekaj kot je dodajanje sodelavcev ali

označevanje težav?

To je, kjer GitHub API pride prav. GitHub ima tona API končnih točk za delati skoraj karkoli lahko naredite na spletni strani na avtomatiziran način. V tej sekciji se bomo naučili, kako overiti in se povezati na API, kako komentirati težavo in kako spremeniti status zahtevka potega preko API-ja.

Osnovna uporaba

Najosnovnejša stvar, ki jo lahko naredite je enostaven GET zahtevek na končni točki, ki ne zahteva overitviije. To je lahko uporabnik ali samo bralne informacije na odprto kodnem projektu. Na primer, če želite vedeti več o uporabniku imenovanem "schacon", lahko pošemo nekaj kot je:

```
$ curl https://api.github.com/users/schacon
{
  "login": "schacon",
  "id": 70,
  "avatar_url": "https://avatars.githubusercontent.com/u/70",
  # ...
  "name": "Scott Chacon",
  "company": "GitHub",
  "following": 19,
  "created_at": "2008-01-27T17:19:28Z",
  "updated_at": "2014-06-10T02:37:23Z"
}
```

Na voljo je tona končnih točki kot je ta, da se dobi informacije o organizacijah, projektih, težavah, pošiljanjih—skoraj karkoli lahko javno vidite na GitHub-u. Lahko celo uporabite API za izpis arbitrarnega Markdown-a ali najdete predlogo [.gitignore](#).

```

$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}

```

Komentiranje na težavi

Vendar, če želite narediti akcijo na spletni strani kot je komentiranje na težavi ali zahtevku potega ali če želite pogledati ali imeti interakcijo z zasebno vsebino, boste morali narediti overitev.

Obstaja nekaj načinov za overjanje. Lahko uporabite osnovno overjanje s samo vašim uporabniškim imenom in geslom, vendar v splošnem je to slaba ideja uporabiti žeton zasebnega dostopa. To lahko generirate iz zavihka “Applications” strani vaših nastavitvev.

The screenshot displays the GitHub user settings page for 'tonychacon', specifically the 'Applications' tab. On the left is a navigation menu with options like Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications (highlighted), Repositories, and Organizations. The main content area is divided into several sections:

- Developer applications:** Includes a 'Register new application' button and text: 'Do you want to develop an application that uses the GitHub API? Register an application to generate OAuth tokens.'
- Personal access tokens:** Includes a 'Generate new token' button and text: 'Need an API token for scripts or testing? Generate a personal access token for quick access to the GitHub API.' Below this is a help icon and text: 'Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.'
- Authorized applications:** A section stating 'You have no applications authorized to access your account.'
- GitHub applications:** A section stating 'These are applications developed and owned by GitHub, Inc. They have full access to your GitHub account.' It lists one application: 'GitHub Team', with the text 'Last used on Oct 6, 2014' and a 'Revoke' button.

Figure 134. Generate your access token from the “Applications” tab of your settings page.

Vprašalo vas bo za kakšen obseg želite ta žeton in opis. Zagotovite, da uporabljate dober opis, da se počutite udobno, ko odstranjujete žeton, ko vaša skripta ali aplikacija ni več v uporabi.

GitHub vam bo samo prikazal žeton enkrat, torej bodite prepričani, da ga kopirate. Sedaj ga lahko uporabite za overitev v vaši skripti namesto uporabe uporabniškega imena in gesla. To je lepo, ker lahko omejite obseg česar želite narediti in žeton je možno umakniti.

To ima tudi dodano prednost povečanja vaše mejne stopnje. Brez overitvije boste omejeni na 60 zahtevkov na uro. Če naredite overitev lahko naredite do 5000 zahtevkov na uro.

Torej uporabimo, da naredimo komentar na eni izmed vaših težav. Predpostavimo, da želite pustiti komentar na določeni težavi, Issue #6. Da to naredite, moramo narediti HTTP POST zahtevek na `repos/<user>/<repo>/issues/<num>/comments` z žetonom, ki ste ga ravnokar generirali kot glavo overitvije.

```
$ curl -H "Content-Type: application/json" \
-H "Authorization: token TOKEN" \
--data '{"body":"A new comment, :+1:"}' \
https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}
```

Sedaj če greste na to težavo, lahko vidite komentar, ki smo ga ravnokar uspešno poslali kot v [A comment posted from the GitHub API](#).

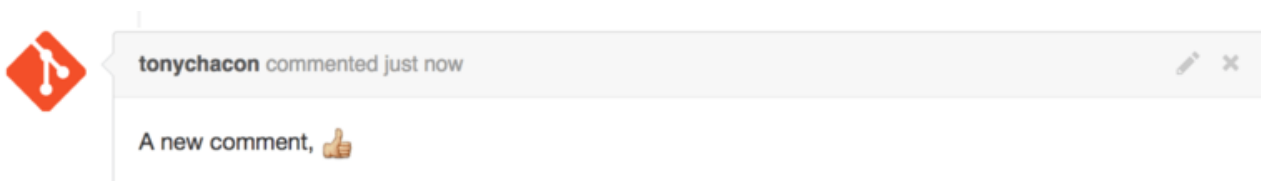


Figure 135. A comment posted from the GitHub API.

Uporabite lahko API, da naredite skoraj vse, kar lahko naredite na spletni strani—ustvarjanje in nastavitve mejnikov, določanje ljudi težavam in zahtevkom potegov,

ustvarjanje in spreminjanje oznak, dostopanje do podatkov pošiljanja, ustvarjanje novih pošiljanj in vej, odpiranje, zapiranje ali združevanje zahtevkov potega, ustvarjanje in urejanje ekip, komentiranje na vrsticah kode v zahtevku potega, iskanje na strani itd.

Sprememba statusa zahtevka potega

En zadnji primer, ki ga bomo pogledali, saj je resnično uporaben, če delate z zahtevki potegov. Vsako pošiljanje ima lahko en ali več statusov povezanih z njim in na voljo je API za dodajanje in poizvedbo tega statusa.

Večina storitev stalne integracije in testiranja uporabljajo ta API, da reagirajo in potiskajo testno kodo, ki je bila potiskana in nato poročajo nazaj, če je to pošiljanje šlo skozi vse teste. Lahko bi tudi uporabili to za preveriti, če je sporočilo pošiljanja ustrezno oblikovano, če je pošiljatelj sledil vsem vašim smernicam prispevanja, če je bilo pošiljanje veljavno podpisano — katerokoli število stvari.

Recimo, da ste nastavili spletno kljuko na vašem repozitoriju, ki doseže majhno spletno storitev, ki preveri za niz `Signed-off-by` v sporočilo pošiljanja.

```

require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"      => state,
      "description" => description,
      "target_url" => "http://example.com/how-to-signoff",
      "context"    => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type' => 'application/json',
        'User-Agent'   => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" }
    )
  end
end
end

```

Upajmo, da je to precej enostavno za slediti. V tej spletni kljuki handlerja bomo pogledali vsako pošiljanje, ki je bilo samo potisnjeno, pogledamo za niz `Signed-off-by` v sporočilu pošiljanja in končno naredimo POST preko HTTP na `/repos/<user>/<repo>/statuses/<commit_sha>` končno točko API-ja s statusom.

V tem primeru lahko pošljemo status (*success*, *failure*, *error*), opis, kaj se je zgodilo, ciljni URL, kamor gre lahko uporabnik po več informacij in “context” v primeru, da je več statusov za eno pošiljanje. Na primer testna storitev lahko ponuja status in storitev preverjanja kot tudi ponuja status - polje “context” je v čemer se razlikujeta.

Če nekdo odpre nov zahtevek potega na GitHub-u in je ta kljuka nastavljena, lahko vidite nekaj kot je [Commit status via the API](#).

schacon commented 33 minutes ago Collaborator

Removing whitespace in the files.

schacon added some commits 31 minutes ago

- properly signed off ... ✓ 9f48fd5
- forgot to sign off ✗ ee7aa38

Add more commits by pushing to the **remove-whitespace** branch on **tonychacon/fade**.

✗ Failed — No signoff found. · Details

Merge with caution!
You can also merge branches on the [command line](#).

Merge pull request

Figure 136. Commit status via the API.

Sedaj lahko vidite majhen zeleni izbirnik zraven pošiljanja, ki ima niz “Signed-off-by” v sporočilu in rdeči križec preko, kjer se je avtor pozabil podpisati. Vidite lahko tudi, da zahtevek potega vzame status zadnjega pošiljanja na veji in vas posvari, če gre za neuspeh. To je resnično uporabno, če uporabljate ta API za rezultate testov, da po nesreči ne združite nečesa, kjer je zadnje pošiljanje padlo na testih.

Octokit

Čeprav smo do sedaj delali skoraj vse preko `curl` in enostavnih HTTP zahtevkov v teh primerih, obstoja nekaj odprto-kodnih knjižnic, ki naredijo ta API na voljo na bolj idiomatski način. V času tega pisanja podprti jeziki vključujejo Go, Objective-C, Ruby in .NET. Preverite <http://github.com/octokit> za več informacij o tem, kako upravljajo večino HTTP-ja za vas.

Upajmo, da vam ta orodja lahko pomagajo prilagoditi in spremeniti GitHub, da dela boljše za vaš določen potek dela. Za celotno dokumentacijo o celotnem API-ju kot tudi vodičih za pogosta opravila, preverite <https://developer.github.com>.

Povzetek

Sedaj ste uporabnik GitHub-a. Veste, kako ustvariti račun, upravljati organizacijo, ustvariti in potisniti v repozitorije, prispevati projektom drugih ljudi in sprejeti prispevke od ostalih. V naslednjem poglavju se boste naučili več o močnih orodjih in nasvetih o ukvarjanju s kompleksnimi situacijami, ki vas bodo naredile pravega Git mojstra.

Orodja Git

Do sedaj ste se naučili večino dnevnih ukazov in potekov dela, ki jih boste morali upravljati ali vzdrževati v repozitoriju Git za vaš nadzor izvorne kode. Dosegli ste osnovna opravila sledenja in pošiljanja datotek in oprteni ste z močjo vmesne faze in lahkega tematskega razvejanja in združevanja.

Sedaj boste raziskali število zelo močnih stvari, ki jih Git lahko naredi in lahko jih morda ne nujno uporabili dnevno vendar jih boste potrebovali na neki točki.

Revision Selection

Git allows you to specify specific commits or a range of commits in several ways. They aren't necessarily obvious but are helpful to know.

Single Revisions

You can obviously refer to a commit by the SHA-1 hash that it's given, but there are more human-friendly ways to refer to commits as well. This section outlines the various ways you can refer to a single commit.

Short SHA-1

Git is smart enough to figure out what commit you meant to type if you provide the first few characters, as long as your partial SHA-1 is at least four characters long and unambiguous – that is, only one object in the current repository begins with that partial SHA-1.

For example, to see a specific commit, suppose you run a `git log` command and identify the commit where you added certain functionality:

```

$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

```

In this case, choose `1c002dd....`. If you `git show` that commit, the following commands are equivalent (assuming the shorter versions are unambiguous):

```

$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d

```

Git can figure out a short, unique abbreviation for your SHA-1 values. If you pass `--abbrev-commit` to the `git log` command, the output will use shorter values but keep them unique; it defaults to using seven characters but makes them longer if necessary to keep the SHA-1 unambiguous:

```

$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit

```

Generally, eight to ten characters are more than enough to be unique within a project.

As an example, the Linux kernel, which is a pretty large project with over 450k commits and 3.6 million objects, has no two objects whose SHA-1s overlap more than the first 11 characters.

A SHORT NOTE ABOUT SHA-1

A lot of people become concerned at some point that they will, by random happenstance, have two objects in their repository that hash to the same SHA-1 value. What then?

If you do happen to commit an object that hashes to the same SHA-1 value as a previous object in your repository, Git will see the previous object already in your Git database and assume it was already written. If you try to check out that object again at some point, you'll always get the data of the first object.

NOTE

However, you should be aware of how ridiculously unlikely this scenario is. The SHA-1 digest is 20 bytes or 160 bits. The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is $p = (n(n-1)/2) * (1/2^{160})$). 2^{80} is 1.2×10^{24} or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.

Here's an example to give you an idea of what it would take to get a SHA-1 collision. If all 6.5 billion humans on Earth were programming, and every second, each one was producing code that was the equivalent of the entire Linux kernel history (3.6 million Git objects) and pushing it into one enormous Git repository, it would take roughly 2 years until that repository contained enough objects to have a 50% probability of a single SHA-1 object collision. A higher probability exists that every member of your programming team will be attacked and killed by wolves in unrelated incidents on the same night.

Branch References

The most straightforward way to specify a commit requires that it has a branch reference pointed at it. Then, you can use a branch name in any Git command that expects a commit object or SHA-1 value. For instance, if you want to show the last commit object on a branch, the following commands are equivalent, assuming that the `topic1` branch points to `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

If you want to see which specific SHA-1 a branch points to, or if you want to see what any of these examples boils down to in terms of SHA-1s, you can use a Git plumbing tool called `rev-parse`. You can see [Notranjost Git-a](#) for more information about plumbing tools; basically, `rev-parse` exists for lower-level operations and isn't designed to be used in day-to-day operations. However, it can be helpful sometimes when you need to see what's really going on. Here you can run `rev-parse` on your branch.

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

RefLog Shortnames

One of the things Git does in the background while you're working away is keep a "reflog" – a log of where your HEAD and branch references have been for the last few months.

You can see your reflog by using `git reflog`:

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Every time your branch tip is updated for any reason, Git stores that information for you in this temporary history. And you can specify older commits with this data, as well. If you want to see the fifth prior value of the HEAD of your repository, you can use the `@{n}` reference that you see in the reflog output:

```
$ git show HEAD@{5}
```

You can also use this syntax to see where a branch was some specific amount of time ago. For instance, to see where your `master` branch was yesterday, you can type

```
$ git show master@{yesterday}
```

That shows you where the branch tip was yesterday. This technique only works for data that's still in your reflog, so you can't use it to look for commits older than a few months.

To see reflog information formatted like the `git log` output, you can run `git log -g`:


```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800
```

fixed refs handling, added gc auto, updated tests

```
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800
```

Merge commit 'phedders/rdocs'

It's important to note that the reflog information is strictly local – it's a log of what you've done in your repository. The references won't be the same on someone else's copy of the repository; and right after you initially clone a repository, you'll have an empty reflog, as no activity has occurred yet in your repository. Running `git show HEAD@{2.months.ago}` will work only if you cloned the project at least two months ago – if you cloned it five minutes ago, you'll get no results.

Ancestry References

The other main way to specify a commit is via its ancestry. If you place a `^` at the end of a reference, Git resolves it to mean the parent of that commit. Suppose you look at the history of your project:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
| \
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
| /
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

Then, you can see the previous commit by specifying `HEAD^`, which means “the parent of HEAD”:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 15:08:43 2008 -0800
```

```
Merge commit 'phedders/rdocs'
```

You can also specify a number after the `^` – for example, `d921970^2` means “the second parent of d921970.” This syntax is only useful for merge commits, which have more than one parent. The first parent is the branch you were on when you merged, and the second is the commit on the branch that you merged in:

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Dec 11 14:58:32 2008 -0800
```

```
added some blame and merge stuff
```

```
$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date: Wed Dec 10 22:22:03 2008 +0000
```

```
Some rdoc changes
```

The other main ancestry specification is the `~`. This also refers to the first parent, so `HEAD~` and `HEAD^` are equivalent. The difference becomes apparent when you specify a number. `HEAD~2` means “the first parent of the first parent,” or “the grandparent” – it traverses the first parents the number of times you specify. For example, in the history listed earlier, `HEAD~3` would be

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

```
ignore *.gem
```

This can also be written `HEAD^^^`, which again is the first parent of the first parent of the first parent:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500

ignore *.gem
```

You can also combine these syntaxes – you can get the second parent of the previous reference (assuming it was a merge commit) by using `HEAD~3^2`, and so on.

Commit Ranges

Now that you can specify individual commits, let's see how to specify ranges of commits. This is particularly useful for managing your branches – if you have a lot of branches, you can use range specifications to answer questions such as, “What work is on this branch that I haven't yet merged into my main branch?”

Double Dot

The most common range specification is the double-dot syntax. This basically asks Git to resolve a range of commits that are reachable from one commit but aren't reachable from another. For example, say you have a commit history that looks like [Example history for range selection](#).

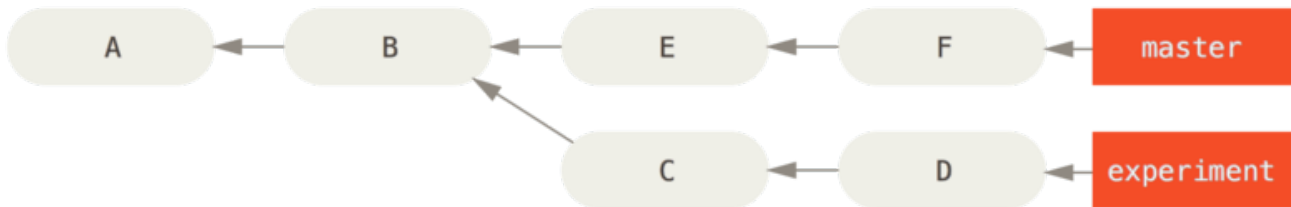


Figure 137. Example history for range selection.

You want to see what is in your experiment branch that hasn't yet been merged into your master branch. You can ask Git to show you a log of just those commits with `master..experiment` – that means “all commits reachable by experiment that aren't reachable by master.” For the sake of brevity and clarity in these examples, I'll use the letters of the commit objects from the diagram in place of the actual log output in the order that they would display:

```
$ git log master..experiment
D
C
```

If, on the other hand, you want to see the opposite – all commits in `master` that aren't in `experiment` – you can reverse the branch names. `experiment..master` shows you everything in `master` not reachable from `experiment`:

```
$ git log experiment..master
F
E
```

This is useful if you want to keep the `experiment` branch up to date and preview what you're about to merge in. Another very frequent use of this syntax is to see what you're about to push to a remote:

```
$ git log origin/master..HEAD
```

This command shows you any commits in your current branch that aren't in the `master` branch on your `origin` remote. If you run a `git push` and your current branch is tracking `origin/master`, the commits listed by `git log origin/master..HEAD` are the commits that will be transferred to the server. You can also leave off one side of the syntax to have Git assume HEAD. For example, you can get the same results as in the previous example by typing `git log origin/master..` – Git substitutes HEAD if one side is missing.

Multiple Points

The double-dot syntax is useful as a shorthand; but perhaps you want to specify more than two branches to indicate your revision, such as seeing what commits are in any of several branches that aren't in the branch you're currently on. Git allows you to do this by using either the `^` character or `--not` before any reference from which you don't want to see reachable commits. Thus these three commands are equivalent:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

This is nice because with this syntax you can specify more than two references in your query, which you cannot do with the double-dot syntax. For instance, if you want to see all commits that are reachable from `refA` or `refB` but not from `refC`, you can type one of these:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

This makes for a very powerful revision query system that should help you figure out what is in your branches.

Triple Dot

The last major range-selection syntax is the triple-dot syntax, which specifies all the commits that are reachable by either of two references but not by both of them. Look

back at the example commit history in [Example history for range selection](#).. If you want to see what is in `master` or `experiment` but not any common references, you can run

```
$ git log master...experiment
F
E
D
C
```

Again, this gives you normal `log` output but shows you only the commit information for those four commits, appearing in the traditional commit date ordering.

A common switch to use with the `log` command in this case is `--left-right`, which shows you which side of the range each commit is in. This helps make the data more useful:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

With these tools, you can much more easily let Git know what commit or commits you want to inspect.

Interactive Staging

Git comes with a couple of scripts that make some command-line tasks easier. Here, you'll look at a few interactive commands that can help you easily craft your commits to include only certain combinations and parts of files. These tools are very helpful if you modify a bunch of files and then decide that you want those changes to be in several focused commits rather than one big messy commit. This way, you can make sure your commits are logically separate changesets and can be easily reviewed by the developers working with you. If you run `git add` with the `-i` or `--interactive` option, Git goes into an interactive shell mode, displaying something like this:

```
$ git add -i
      staged      unstaged path
 1:  unchanged      +0/-1 TODO
 2:  unchanged      +1/-1 index.html
 3:  unchanged      +5/-1 lib/simplegit.rb

*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch      6: diff        7: quit        8: help
What now>
```

You can see that this command shows you a much different view of your staging area – basically the same information you get with `git status` but a bit more succinct and informative. It lists the changes you've staged on the left and unstaged changes on the right.

After this comes a Commands section. Here you can do a number of things, including staging files, unstaging files, staging parts of files, adding untracked files, and seeing diffs of what has been staged.

Staging and Unstaging Files

If you type `2` or `u` at the `What now>` prompt, the script prompts you for which files you want to stage:

```
What now> 2
      staged      unstaged path
1:   unchanged    +0/-1 TODO
2:   unchanged    +1/-1 index.html
3:   unchanged    +5/-1 lib/simplegit.rb
Update>>
```

To stage the `TODO` and `index.html` files, you can type the numbers:

```
Update>> 1,2
      staged      unstaged path
* 1:   unchanged    +0/-1 TODO
* 2:   unchanged    +1/-1 index.html
  3:   unchanged    +5/-1 lib/simplegit.rb
Update>>
```

The `*` next to each file means the file is selected to be staged. If you press Enter after typing nothing at the `Update>>` prompt, Git takes anything selected and stages it for you:

```
Update>>
updated 2 paths

*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit        8: help
What now> 1
      staged      unstaged path
1:     +0/-1      nothing TODO
2:     +1/-1      nothing index.html
3:   unchanged    +5/-1 lib/simplegit.rb
```

Now you can see that the `TODO` and `index.html` files are staged and the `simplegit.rb`

file is still unstaged. If you want to unstage the TODO file at this point, you use the **3** or **r** (for revert) option:

```
*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit       8: help
What now> 3
      staged      unstaged path
 1:      +0/-1      nothing TODO
 2:      +1/-1      nothing index.html
 3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> 1
      staged      unstaged path
* 1:      +0/-1      nothing TODO
 2:      +1/-1      nothing index.html
 3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Looking at your Git status again, you can see that you've unstaged the TODO file:

```
*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit       8: help
What now> 1
      staged      unstaged path
 1:      unchanged      +0/-1 TODO
 2:      +1/-1      nothing index.html
 3:      unchanged      +5/-1 lib/simplegit.rb
```

To see the diff of what you've staged, you can use the **6** or **d** (for diff) command. It shows you a list of your staged files, and you can select the ones for which you would like to see the staged diff. This is much like specifying `git diff --cached` on the command line:

```

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch       6: diff        7: quit       8: help
What now> 6
      staged      unstaged path
  1:      +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

With these basic commands, you can use the interactive add mode to deal with your staging area a little more easily.

Staging Patches

It's also possible for Git to stage certain parts of files and not the rest. For example, if you make two changes to your `simplegit.rb` file and want to stage one of them and not the other, doing so is very easy in Git. From the interactive prompt, type **5** or **p** (for patch). Git will ask you which files you would like to partially stage; then, for each section of the selected files, it will display hunks of the file diff and ask if you would like to stage them, one by one:

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
  end

  def log(treeish = 'master')
-   command("git log -n 25 #{treeish}")
+   command("git log -n 30 #{treeish}")
  end

  def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,]?

```

You have a lot of options at this point. Typing **?** shows a list of what you can do:


```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

Generally, you'll type `y` or `n` if you want to stage each hunk, but staging all of them in certain files or skipping a hunk decision until later can be helpful too. If you stage one part of the file and leave another part unstaged, your status output will look like this:

```
What now> 1
      staged      unstaged path
1:    unchanged  +0/-1 TODO
2:      +1/-1     nothing index.html
3:      +1/-1     +4/-0 lib/simplegit.rb
```

The status of the `simplegit.rb` file is interesting. It shows you that a couple of lines are staged and a couple are unstaged. You've partially staged this file. At this point, you can exit the interactive adding script and run `git commit` to commit the partially staged files.

You also don't need to be in interactive add mode to do the partial-file staging – you can start the same script by using `git add -p` or `git add --patch` on the command line.

Furthermore, you can use patch mode for partially resetting files with the `reset --patch` command, for checking out parts of files with the `checkout --patch` command and for stashing parts of files with the `stash save --patch` command. We'll go into more details on each of these as we get to more advanced usages of these commands.

Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can

reapply at any time.

Stashing Your Work

To demonstrate, you'll go into your project and start working on a couple of files and possibly stage one of the changes. If you run `git status`, you can see your dirty state:

```
$ git status
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

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

    modified:   lib/simplegit.rb
```

Now you want to switch branches, but you don't want to commit what you've been working on yet; so you'll stash the changes. To push a new stash onto your stack, run `git stash` or `git stash save`:

```
$ git stash
Saved working directory and index state \
"WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Your working directory is clean:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

At this point, you can easily switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, two stashes were done previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. If you want to apply

one of the older stashes, you can specify it by naming it, like this: `git stash apply stash@{2}`. If you don't specify a stash, Git assumes the most recent stash and tries to apply it:

```
$ git stash apply
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

You can see that Git re-modifies the files you reverted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from; but having a clean working directory and applying it on the same branch aren't necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash – Git gives you merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but the file you staged before wasn't restaged. To do that, you must run the `git stash apply` command with a `--index` option to tell the command to try to reapply the staged changes. If you had run that instead, you'd have gotten back to your original position:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
```

The `apply` option only tries to apply the stashed work – you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the `--keep-index` option to the `stash save` command. This tells Git to not stash anything that you've already staged with the `git add` command.

This can be really helpful if you've made a number of changes but want to only commit some of them and then come back to the rest of the changes at a later time.

```
$ git status -s
M index.html
M lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M index.html
```

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will only store files that are already in the index. If you specify `--include-untracked` or `-u`, Git will also stash any untracked files you have created.

```
$ git status -s
M index.html
M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Finally, if you specify the `--patch` flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
     return `#{git_cmd} 2>&1`.chomp
     end
   end
+
+   def show(treeish = 'master')
+     command("git show #{treeish}")
+   end
end
test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you've since modified, you'll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run `git stash branch`, which creates a new branch for you, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   index.html
#
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

Cleaning your Working Directory

Finally, you may not want to stash some work or files in your working directory, but simply get rid of them. The `git clean` command will do this for you.

Some common reasons for this might be to remove cruft that has been generated by merges or external tools or to remove build artifacts in order to run a clean build.

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with `git clean`. To remove all the untracked files in your working directory, you can run `git clean -f -d`, which removes any files and also any subdirectories that become empty as a result. The `-f` means *force* or "really do this".

If you ever want to see what it would do, you can run the command with the `-n` option, which means "do a dry run and tell me what you *would* have removed".

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

By default, the `git clean` command will only remove untracked files that are not ignored. Any file that matches a pattern in your `.gitignore` or other ignore files will not be removed. If you want to remove those files too, such as to remove all `.o` files generated from a build so you can do a fully clean build, you can add a `-x` to the clean command.

```
$ git status -s
 M lib/simplegit.rb
 ?? build.TMP
 ?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

If you don't know what the `git clean` command is going to do, always run it with a `-n`

first to double check before changing the `-n` to a `-f` and doing it for real. The other way you can be careful about the process is to run it with the `-i` or “interactive” flag.

This will run the clean command in an interactive mode.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
  1: clean          2: filter by pattern   3: select by numbers   4: ask
each             5: quit
  6: help
What now>
```

This way you can step through each file individually or specify patterns for deletion interactively.

Signing Your Work

Git is cryptographically secure, but it’s not foolproof. If you’re taking work from others on the internet and want to verify that commits are actually from a trusted source, Git has a few ways to sign and verify work using GPG.

GPG Introduction

First of all, if you want to sign anything you need to get GPG configured and your personal key installed.

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
-----
pub   2048R/0A46826A 2014-06-04
uid           Scott Chacon (Git signing key) <schacon@gmail.com>
sub   2048R/874529A9 2014-06-04
```

If you don’t have a key installed, you can generate one with `gpg --gen-key`.

```
gpg --gen-key
```

Once you have a private key to sign with, you can configure Git to use it for signing things by setting the `user.signingkey` config setting.

```
git config --global user.signingkey 0A46826A
```

Now Git will use your key by default to sign tags and commits if you want.

Signing Tags

If you have a GPG private key setup, you can now use it to sign new tags. All you have to do is use `-s` instead of `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
```

```
You need a passphrase to unlock the secret key for
user: "Ben Straub <ben@straub.cc>"
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

If you run `git show` on that tag, you can see your GPG signature attached to it:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date: Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAABAgAGBQJTZbQ1AAoJEF0+sviABDDrZbQH/09Pfe51KPVP1anr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kc3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4iXzQu7tupRihs1bNkfvcIMnSDeSvzCpWAH17h8Wj6hhqePmLm91AYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVdptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1Pb1GfHR4Xahu0qN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----

commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700

    changed the version number
```

Verifying Tags

To verify a signed tag, you use `git tag -v [tag-name]`. This command uses GPG to verify the signature. You need the signer's public key in your keyring for this to work properly:


```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700
```

GIT 1.4.2.1

```
Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:          aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A
```

If you don't have the signer's public key, you get something like this instead:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

Signing Commits

In more recent versions of Git (v1.7.9 and above), you can now also sign individual commits. If you're interested in signing commits directly instead of just the tags, all you need to do is add a `-S` to your `git commit` command.

```
$ git commit -a -S -m 'signed commit'
```

```
You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04
```

```
[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

To see and verify these signatures, there is also a `--show-signature` option to `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun  4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

signed commit
```

Additionally, you can configure `git log` to check any signatures it finds and list them in its output with the `%G?` format.

```
$ git log --pretty="format:%h %G? %aN %s"

5c3386c G Scott Chacon signed commit
ca82a6d N Scott Chacon changed the version number
085bb3b N Scott Chacon removed unnecessary test code
a11bef0 N Scott Chacon first commit
```

Here we can see that only the latest commit is signed and valid and the previous commits are not.

In Git 1.8.3 and later, "git merge" and "git pull" can be told to inspect and reject when merging a commit that does not carry a trusted GPG signature with the `--verify-signatures` command.

If you use this option when merging a branch and it contains commits that are not signed and valid, the merge will not work.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

If the merge contains only valid signed commits, the merge command will show you all the signatures it has checked and then move forward with the merge.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

You can also use the `-S` option with the `git merge` command itself to sign the resulting merge commit itself. The following example both verifies that every commit in the branch to be merged is signed and furthermore signs the resulting merge commit.

```
$ git merge --verify-signatures -S signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
```

```
You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04
```

```
Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 2 insertions(+)
```

Everyone Must Sign

Signing tags and commits is great, but if you decide to use this in your normal workflow, you'll have to make sure that everyone on your team understands how to do so. If you don't, you'll end up spending a lot of time helping people figure out how to rewrite their commits with signed versions. Make sure you understand GPG and the benefits of signing things before adopting this as part of your standard workflow.

Searching

With just about any size codebase, you'll often need to find where a function is called or defined, or find the history of a method. Git provides a couple of useful tools for looking through the code and commits stored in its database quickly and easily. We'll go through a few of them.

Git Grep

Git ships with a command called `grep` that allows you to easily search through any committed tree or the working directory for a string or regular expression. For these examples, we'll look through the Git source code itself.

By default, it will look through the files in your working directory. You can pass `-n` to print out the line numbers where Git has found matches.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:     return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:     ret = gmtime_r(timep, result);
compat/mingw.c:606:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:162:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:429:         if (gmtime_r(&now, &now_tm))
date.c:492:         if (gmtime_r(&time, tm)) {
git-compat-util.h:721:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:723:#define gmtime_r git_gmtime_r
```

There are a number of interesting options you can provide the `grep` command.

For instance, instead of the previous call, you can have Git summarize the output by just showing you which files matched and how many matches there were in each file with the `--count` option:

```
$ git grep --count gmtime_r
compat/gmtime.c:4
compat/mingw.c:1
compat/mingw.h:1
date.c:2
git-compat-util.h:2
```

If you want to see what method or function it thinks it has found a match in, you can pass `-p`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(unsigned long num, char c, const char *date, char
*end, struct tm *tm)
date.c:          if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int
*tm_gmt)
date.c:          if (gmtime_r(&time, tm)) {
```

So here we can see that `gmtime_r` is called in the `match_multi_number` and `match_digit` functions in the `date.c` file.

You can also look for complex combinations of strings with the `--and` flag, which makes sure that multiple matches are in the same line. For instance, let's look for any lines that define a constant with either the strings "LINK" or "BUF_MAX" in them in the Git codebase in an older 1.8.0 version.

Here we'll also use the `--break` and `--heading` options which help split up the output into a more readable format.

```

$ git grep --break --heading \
  -n -e '#define' --and \( -e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m)      (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATION_USES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */

```

The `git grep` command has a few advantages over normal searching commands like `grep` and `ack`. The first is that it's really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

Git Log Searching

Perhaps you're looking not for **where** a term exists, but **when** it existed or was introduced. The `git log` command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If we want to find out for example when the `ZLIB_BUF_MAX` constant was originally introduced, we can tell Git to only show us the commits that either added or removed that string with the `-S` option.

```

$ git log -SZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time

```

If we look at the diff of those commits we can see that in `ef49a7a` the constant was introduced and in `e01503b` it was modified.

If you need to be more specific, you can provide a regular expression to search for with the `-G` option.

Line Log Search

Another fairly advanced log search that is insanely useful is the line history search. This is a fairly recent addition and not very well known, but it can be really helpful. It is called with the `-L` option to `git log` and will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function `git_deflate_bound` in the `zlib.c` file, we could run `git log -L :git_deflate_bound:zlib.c`. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
 {
-    return deflateBound(strm, size);
+    return deflateBound(&strm->z, size);
 }

commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+    return deflateBound(strm, size);
+}
+
```

If Git can't figure out how to match a function or method in your programming language, you can also provide it a regex. For example, this would have done the same

thing: `git log -L '/unsigned long git_deflate_bound',/^}:/:zlib.c`. You could also give it a range of lines or a single line number and you'll get the same sort of output.

Rewriting History

Many times, when working with Git, you may want to revise your commit history for some reason. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn't mean to be working on something yet with the stash command, and you can rewrite commits that already happened so they look like they happened in a different way. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing together or splitting apart commits, or removing commits entirely – all before you share your work with others.

In this section, you'll cover how to accomplish these very useful tasks so that you can make your commit history look the way you want before you share it with others.

Changing the Last Commit

Changing your last commit is probably the most common rewriting of history that you'll do. You'll often want to do two basic things to your last commit: change the commit message, or change the snapshot you just recorded by adding, changing and removing files.

If you only want to modify your last commit message, it's very simple:

```
$ git commit --amend
```

That drops you into your text editor, which has your last commit message in it, ready for you to modify the message. When you save and close the editor, the editor writes a new commit containing that message and makes it your new last commit.

If you've committed and then you want to change the snapshot you committed by adding or changing files, possibly because you forgot to add a newly created file when you originally committed, the process works basically the same way. You stage the changes you want by editing a file and running `git add` on it or `git rm` to a tracked file, and the subsequent `git commit --amend` takes your current staging area and makes it the snapshot for the new commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It's like a very small rebase – don't amend your last commit if you've already pushed it.

Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the rebase

tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the `-i` option to `git rebase`. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to `git rebase -i` the parent of the last commit you want to edit, which is `HEAD~2^` or `HEAD~3`. It may be easier to remember the `~3` because you're trying to edit the last three commits; but keep in mind that you're actually designating four commits ago, the parent of the last commit you want to edit:

```
$ git rebase -i HEAD~3
```

Remember again that this is a rebasing command – every commit included in the range `HEAD~3..HEAD` will be rewritten, whether you change the message or not. Don't include any commit you've already pushed to a central server – doing so will confuse other developers by providing an alternate version of the same change.

Running this command gives you a list of commits in your text editor that looks something like this:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

It's important to note that these commits are listed in the opposite order than you normally see them using the `log` command. If you run a `log`, you see something like

this:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Notice the reverse order. The interactive rebase gives you a script that it's going to run. It will start at the commit you specify on the command line (`HEAD~3`) and replay the changes introduced in each of these commits from top to bottom. It lists the oldest at the top, rather than the newest, because that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word 'pick' to the word 'edit' for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

These instructions tell you exactly what to do. Type

```
$ git commit --amend
```

Change the commit message, and exit the editor. Then, run

```
$ git rebase --continue
```

This command will apply the other two commits automatically, and then you're done. If you change pick to edit on more lines, you can repeat these steps for each commit you

change to edit. Each time, Git will stop, let you amend the commit, and continue when you're finished.

Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the “added cat-file” commit and change the order in which the other two commits are introduced, you can change the rebase script from this

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

to this:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies `310154e` and then `f7f3f6d`, and then stops. You effectively change the order of those commits and remove the “added cat-file” commit completely.

Squashing Commits

It's also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

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

If, instead of “pick” or “edit”, you specify “squash”, Git applies both that change and the change directly before it and makes you merge the commit messages together.

So, if you want to make a single commit from these three commits, you make the script look like this:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

When you save that, you have a single commit that introduces the changes of all three previous commits.

Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of “updated README formatting and added blame”, you want to split it into two commits: “updated README formatting” for the first, and “added blame” for the second. You can do that in the `rebase -i` script by changing the instruction on the commit you want to split to “edit”:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Then, when the script drops you to the command line, you reset that commit, take the changes that have been reset, and create multiple commits out of them. When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (`f7f3f6d`), applies the second (`310154e`), and drops you to the console. There, you can do a mixed reset of that commit with `git reset HEAD^`, which effectively undoes that commit and leaves the modified files unstaged. Now you can stage and commit files until you have several commits, and run `git rebase --continue` when you’re done:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git applies the last commit ([a5f4a0d](#)) in the script, and your history looks like this:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Once again, this changes the SHA-1s of all the commits in your list, so make sure no commit shows up in that list that you've already pushed to a shared repository.

The Nuclear Option: `filter-branch`

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way – for instance, changing your e-mail address globally or removing a file from every commit. The command is `filter-branch`, and it can rewrite huge swaths of your history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses so you can get an idea of some of the things it's capable of.

Removing a File from Every Commit

This occurs fairly commonly. Someone accidentally commits a huge binary file with a thoughtless `git add .`, and you want to remove it everywhere. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `filter-branch` is the tool you probably want to use to scrub your entire history. To remove a file named `passwords.txt` from your entire history, you can use the `--tree-filter` option to `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called `passwords.txt` from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like `git filter-branch --tree-filter 'rm -f *~' HEAD`.

You'll be able to watch Git rewriting trees and commits and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your master branch after you've determined the outcome is what you really want. To run `filter-branch` on all your branches, you can pass `--all` to the command.

Making a Subdirectory the New Root

Suppose you've done an import from another source control system and have subdirectories that make no sense (trunk, tags, and so on). If you want to make the `trunk` subdirectory be the new project root for every commit, `filter-branch` can help you do that, too:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory each time. Git will also automatically remove commits that did not affect the subdirectory.

Changing E-Mail Addresses Globally

Another common case is that you forgot to run `git config` to set your name and e-mail address before you started working, or perhaps you want to open-source a project at work and change all your work e-mail addresses to your personal address. In any case, you can change e-mail addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the e-mail addresses that are yours, so you use `--commit-filter`:

```
$ git filter-branch --commit-filter '
    if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
    then
        GIT_AUTHOR_NAME="Scott Chacon";
        GIT_AUTHOR_EMAIL="schacon@example.com";
        git commit-tree "$@";
    else
        git commit-tree "$@";
    fi' HEAD
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA in your history, not just those that have the matching e-mail address.

Reset Demystified

Before moving on to more specialized tools, let's talk about `reset` and `checkout`. These commands are two of the most confusing parts of Git when you first encounter them. They do so many things, that it seems hopeless to actually understand them and

employ them properly. For this, we recommend a simple metaphor.

The Three Trees

An easier way to think about `reset` and `checkout` is through the mental frame of Git being a content manager of three different trees. By “tree” here we really mean “collection of files”, not specifically the data structure. (There are a few cases where the index doesn’t exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

| Tree | Role |
|-------------------|-----------------------------------|
| HEAD | Last commit snapshot, next parent |
| Index | Proposed next commit snapshot |
| Working Directory | Sandbox |

The HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It’s generally simplest to think of HEAD as the snapshot of **your last commit**.

In fact, it’s pretty easy to see what that snapshot looks like. Here is an example of getting the actual directory listing and SHA-1 checksums for each file in the HEAD snapshot:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

The `cat-file` and `ls-tree` commands are “plumbing” commands that are used for lower level things and not really used in day-to-day work, but they help us see what’s going on here.

The Index

The Index is your **proposed next commit**. We’ve also been referring to this concept as Git’s “Staging Area” as this is what Git looks at when you run `git commit`.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and `git commit` converts that into the tree for a new commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Again, here we're using `ls-files`, which is more of a behind the scenes command that shows you what your index currently looks like.

The index is not technically a tree structure – it's actually implemented as a flattened manifest – but for our purposes it's close enough.

The Working Directory

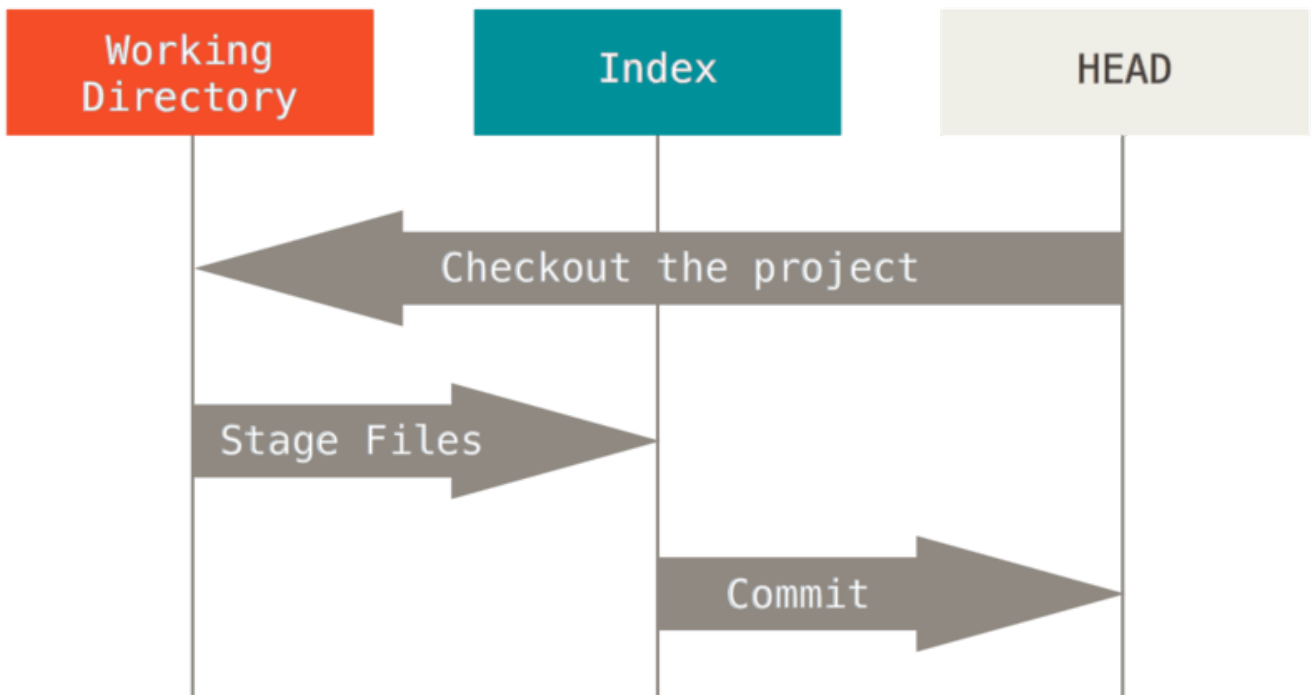
Finally, you have your working directory. The other two trees store their content in an efficient but inconvenient manner, inside the `.git` folder. The Working Directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the Working Directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

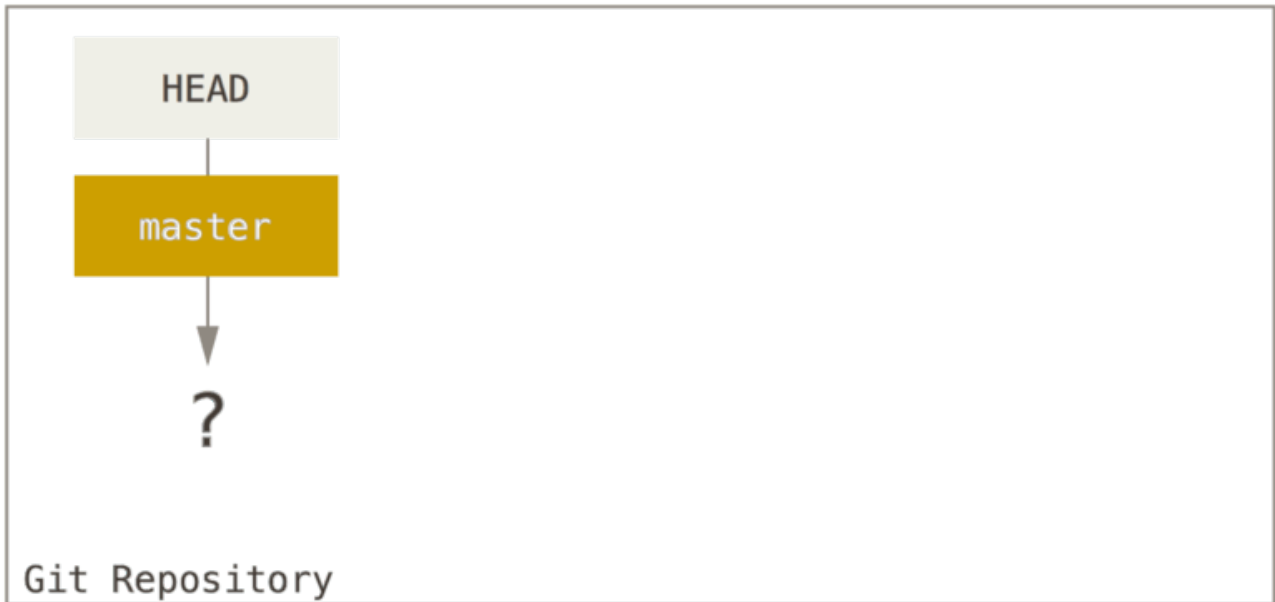
1 directory, 3 files
```

The Workflow

Git's main purpose is to record snapshots of your project in successively better states, by manipulating these three trees.

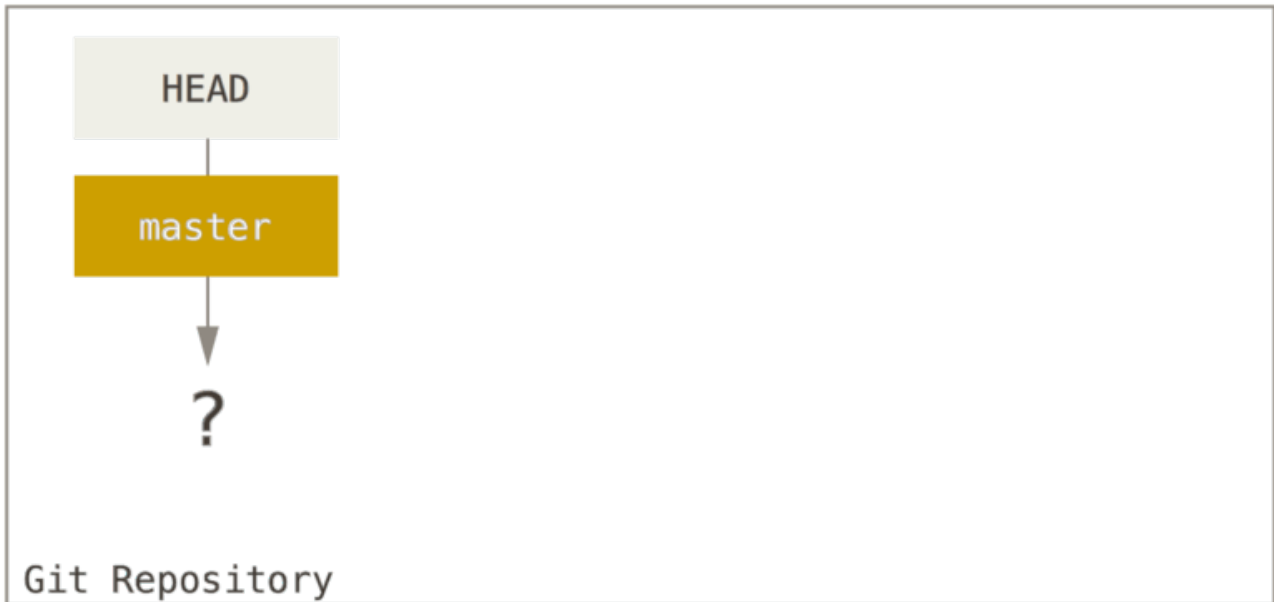


Let's visualize this process: say you go into a new directory with a single file in it. We'll call this `v1` of the file, and we'll indicate it in blue. Now we run `git init`, which will create a Git repository with a HEAD reference which points to an unborn branch (`master` doesn't exist yet).

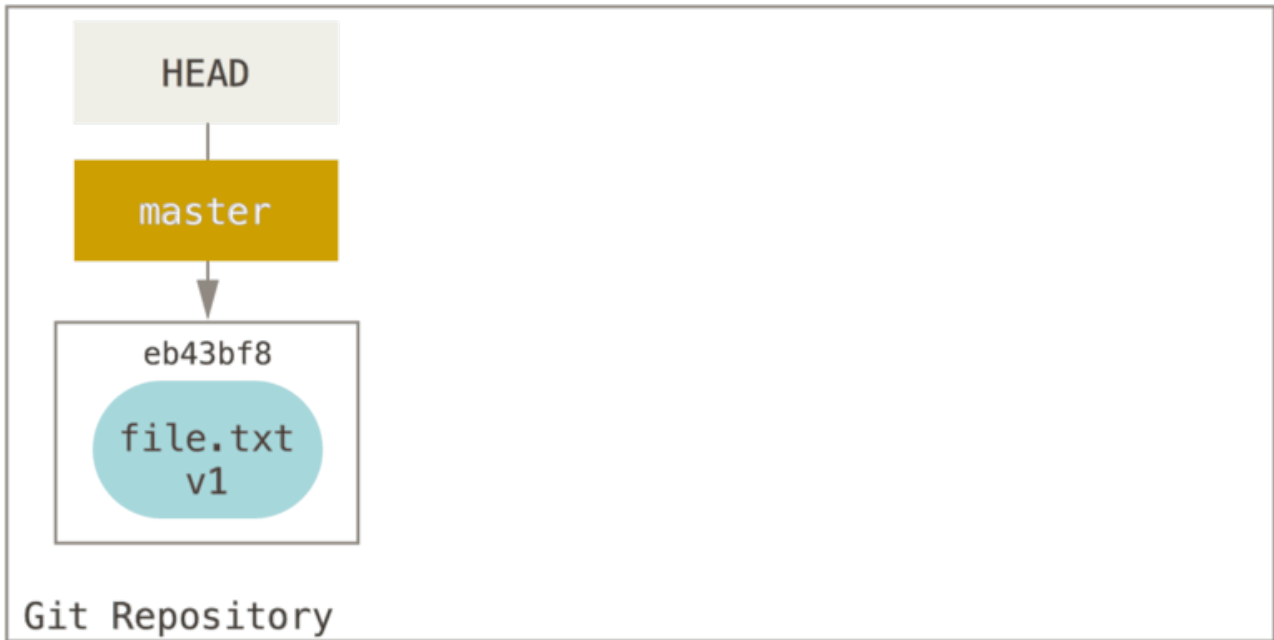


At this point, only the Working Directory tree has any content.

Now we want to commit this file, so we use `git add` to take content in the Working Directory and copy it to the Index.



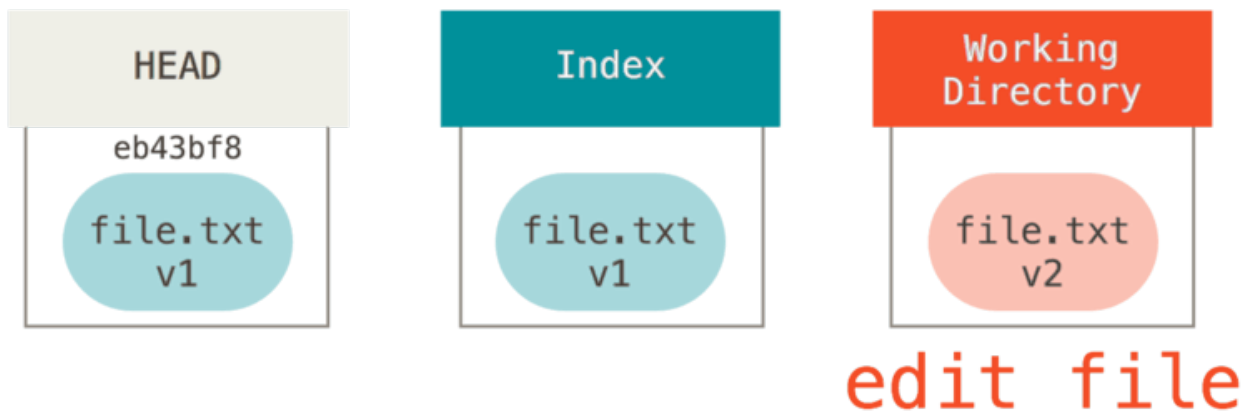
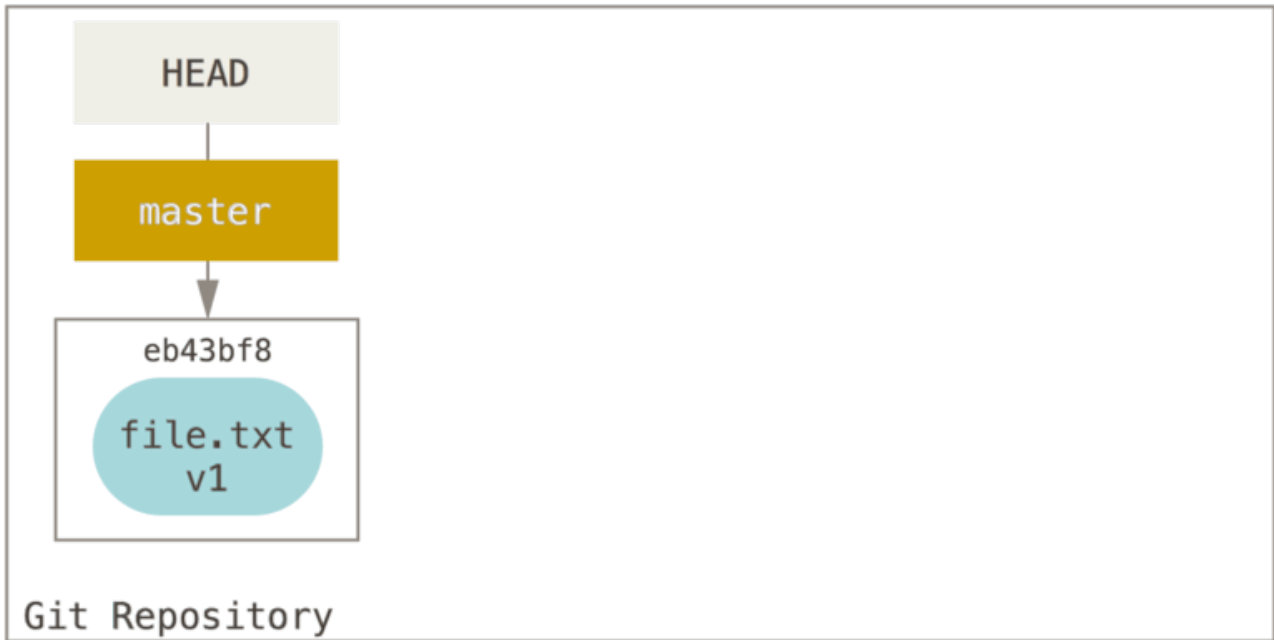
Then we run `git commit`, which takes the contents of the Index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates `master` to point to that commit.



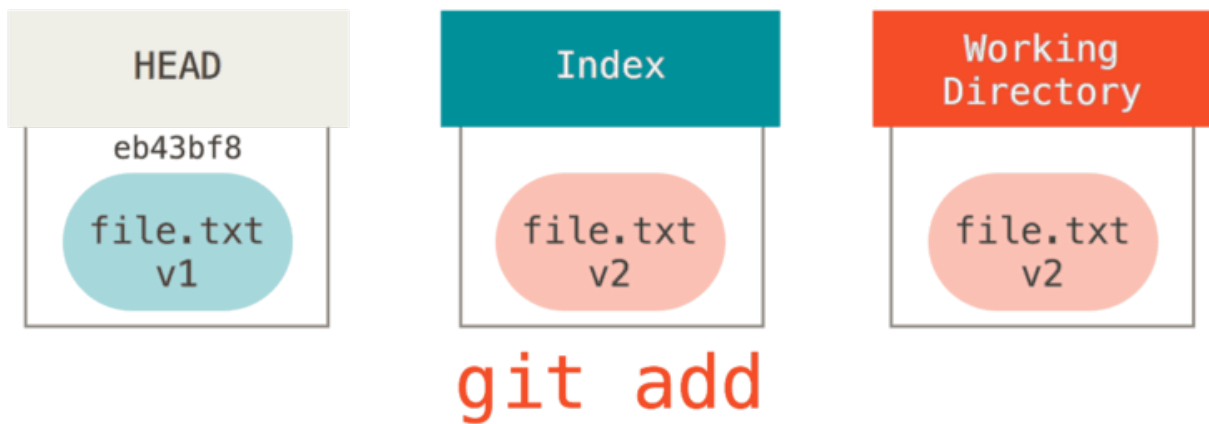
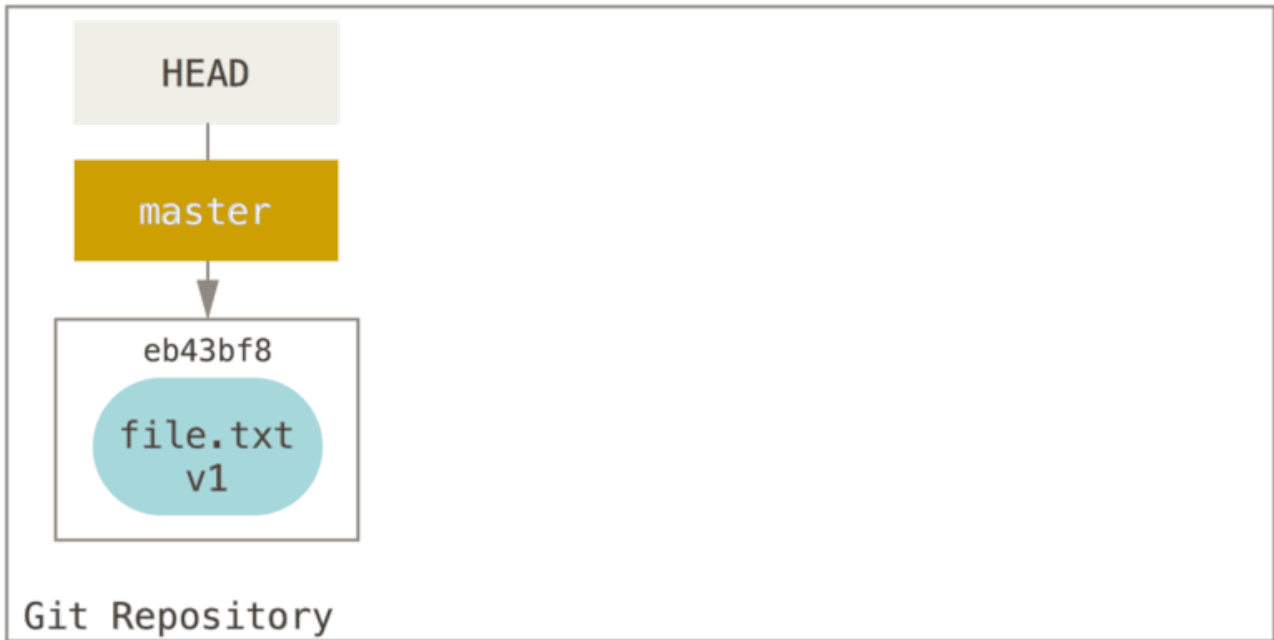
git commit

If we run `git status`, we'll see no changes, because all three trees are the same.

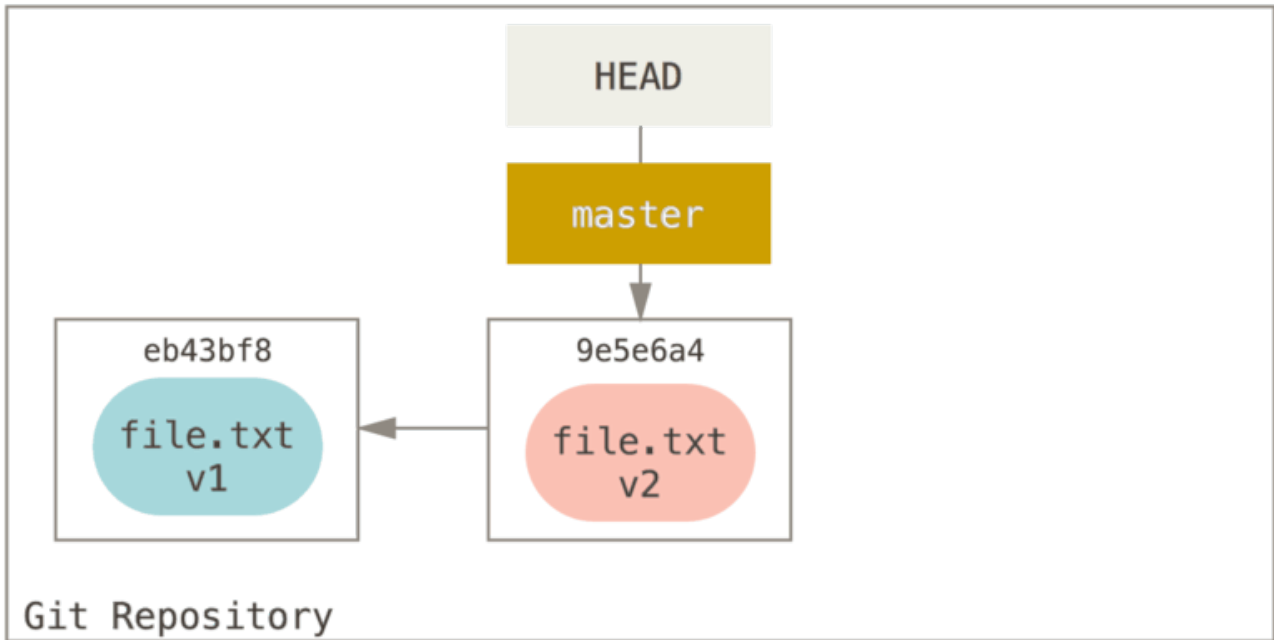
Now we want to make a change to that file and commit it. We'll go through the same process; first we change the file in our working directory. Let's call this `v2` of the file, and indicate it in red.



If we run `git status` right now, we'll see the file in red as "Changes not staged for commit," because that entry differs between the Index and the Working Directory. Next we run `git add` on it to stage it into our Index.



At this point if we run `git status` we will see the file in green under “Changes to be committed” because the Index and HEAD differ – that is, our proposed next commit is now different from our last commit. Finally, we run `git commit` to finalize the commit.



git commit

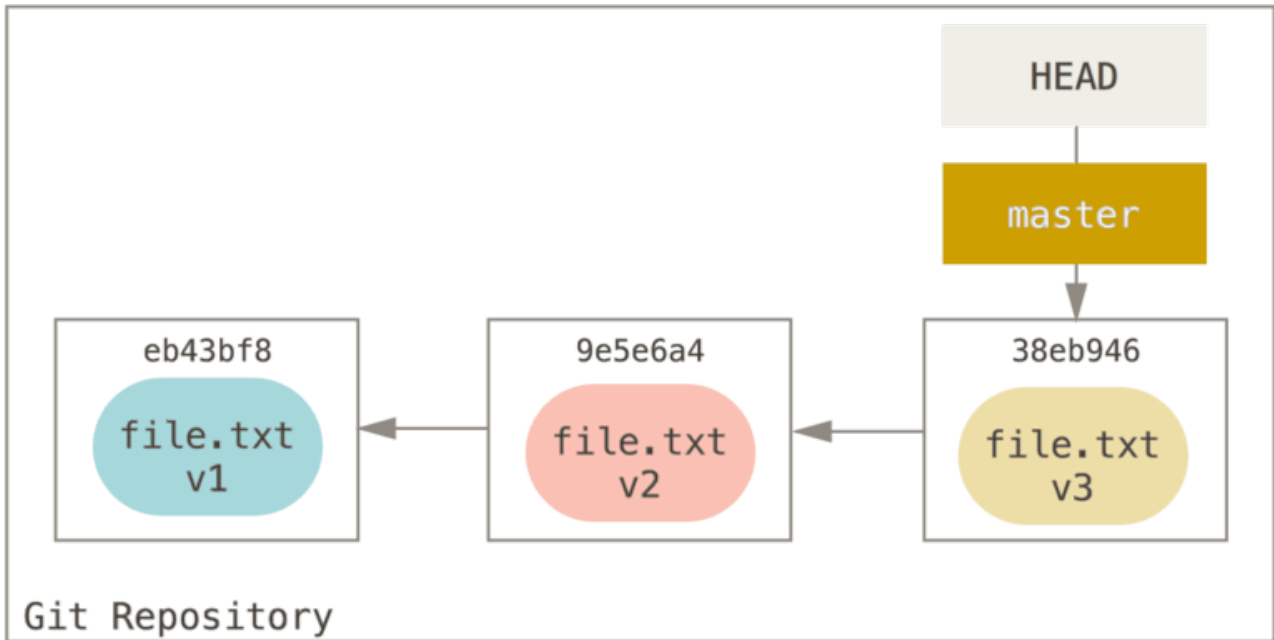
Now `git status` will give us no output, because all three trees are the same again.

Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **Index** with the snapshot of that commit, then copies the contents of the **Index** into your **Working Directory**.

The Role of Reset

The `reset` command makes more sense when viewed in this context.

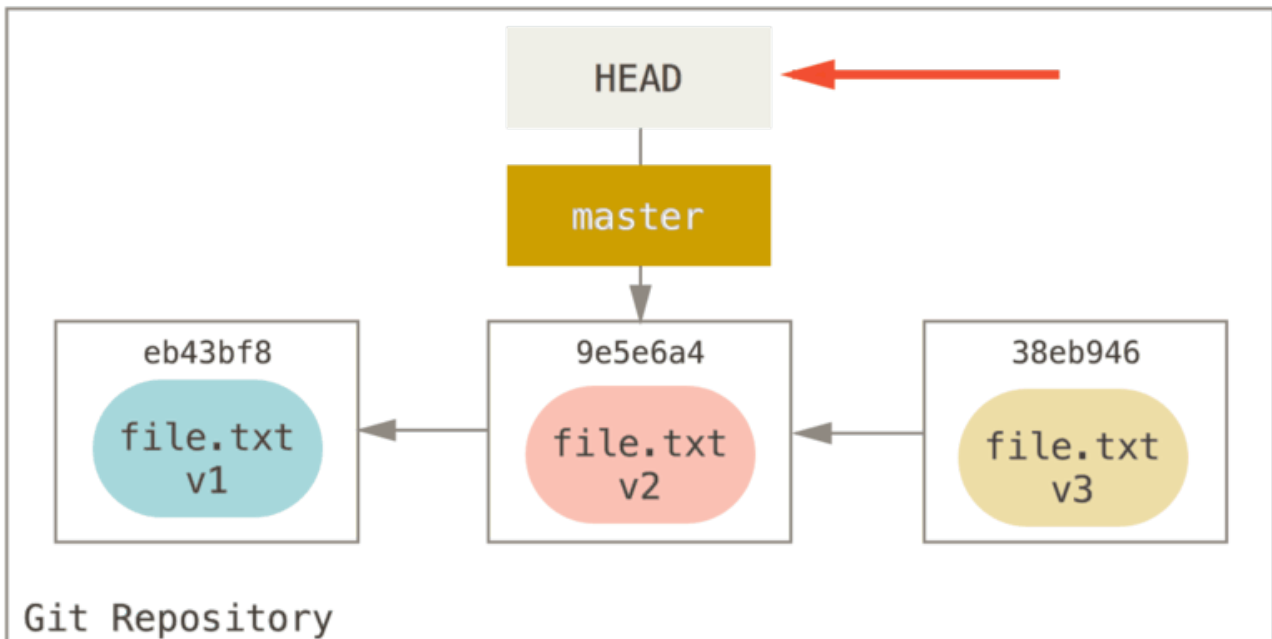
For the purposes of these examples, let's say that we've modified `file.txt` again and committed it a third time. So now our history looks like this:



Let's now walk through exactly what `reset` does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

Step 1: Move HEAD

The first thing `reset` will do is move what HEAD points to. This isn't the same as changing HEAD itself (which is what `checkout` does); `reset` moves the branch that HEAD is pointing to. This means if HEAD is set to the `master` branch (i.e. you're currently on the `master` branch), running `git reset 9e5e6a4` will start by making `master` point to `9e5e6a4`.



`git reset --soft HEAD~`

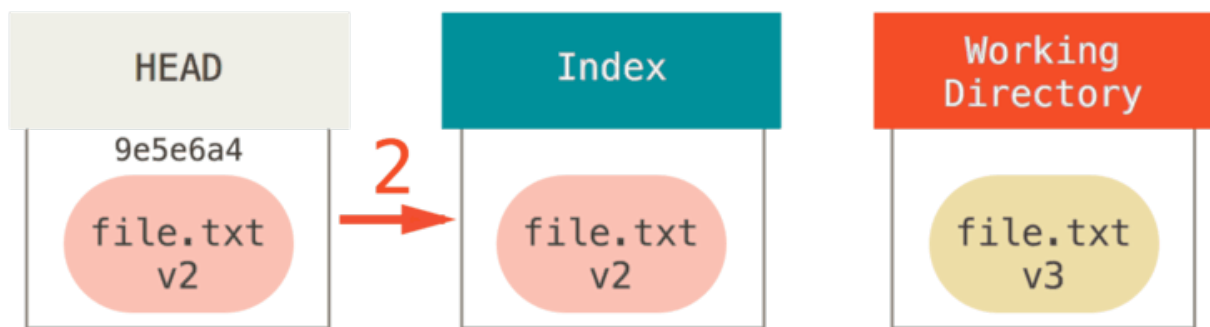
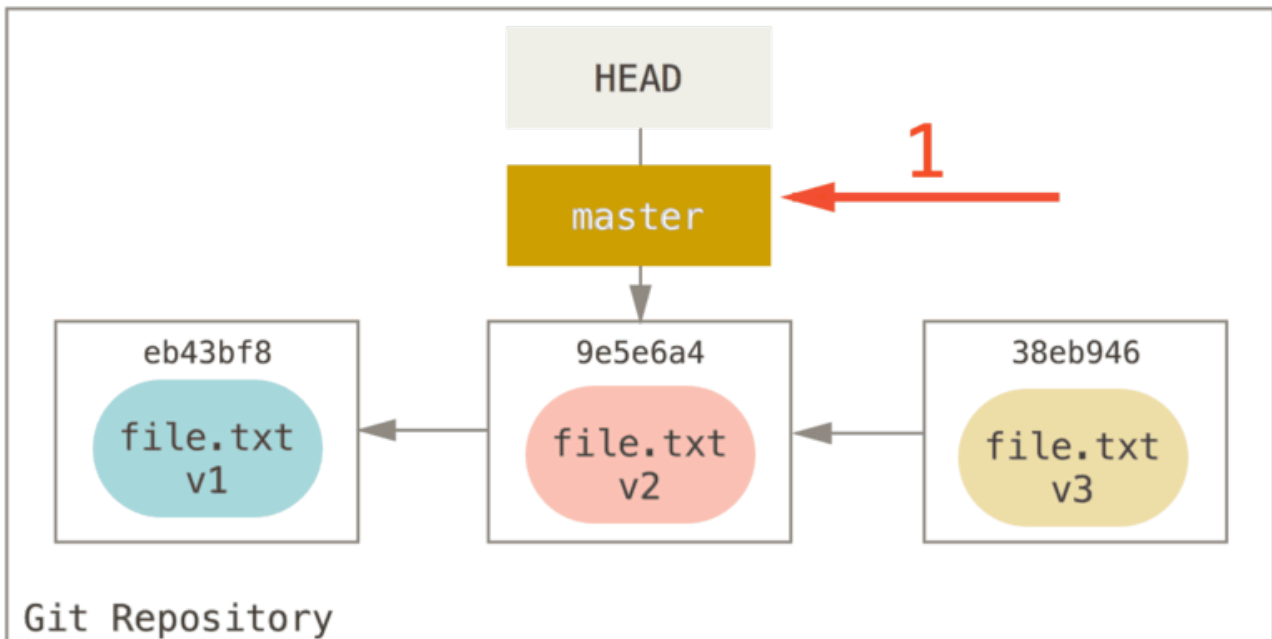
No matter what form of `reset` with a commit you invoke, this is the first thing it will always try to do. With `reset --soft`, it will simply stop there.

Now take a second to look at that diagram and realize what happened: it essentially undid the last `git commit` command. When you run `git commit`, Git creates a new commit and moves the branch that HEAD points to up to it. When you `reset` back to `HEAD~` (the parent of HEAD), you are moving the branch back to where it was, without changing the Index or Working Directory. You could now update the Index and run `git commit` again to accomplish what `git commit --amend` would have done (see [Changing the Last Commit](#)).

Step 2: Updating the Index (`--mixed`)

Note that if you run `git status` now you'll see in green the difference between the Index and what the new HEAD is.

The next thing `reset` will do is to update the Index with the contents of whatever snapshot HEAD now points to.



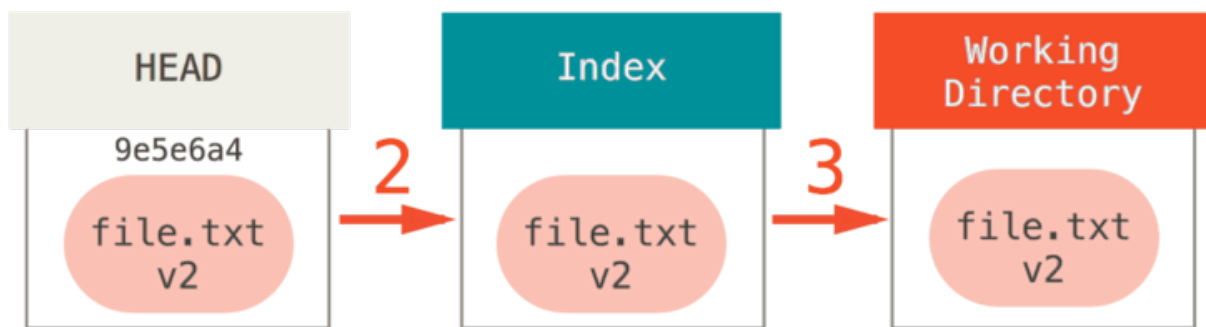
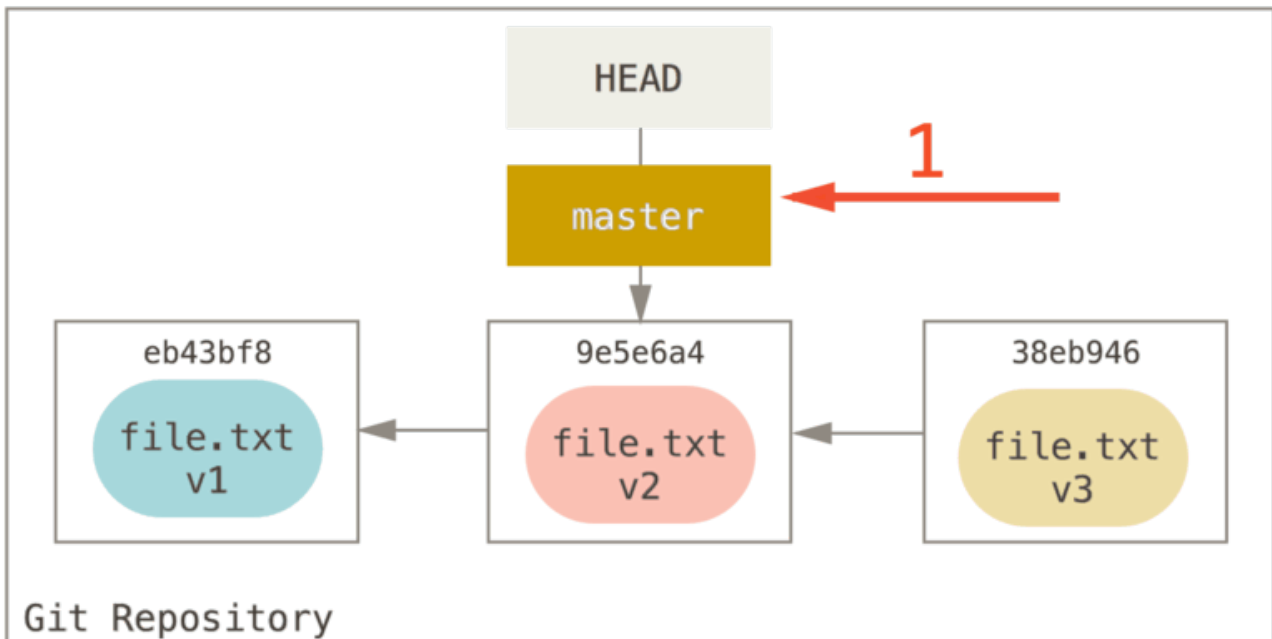
`git reset [--mixed] HEAD~`

If you specify the `--mixed` option, `reset` will stop at this point. This is also the default, so if you specify no option at all (just `git reset HEAD~` in this case), this is where the command will stop.

Now take another second to look at that diagram and realize what happened: it still undid your last `commit`, but also *unstaged* everything. You rolled back to before you ran all your `git add` and `git commit` commands.

Step 3: Updating the Working Directory (`--hard`)

The third thing that `reset` will do is to make the Working Directory look like the Index. If you use the `--hard` option, it will continue to this stage.



`git reset --hard HEAD~`

So let's think about what just happened. You undid your last commit, the `git add` and `git commit` commands, **and** all the work you did in your working directory.

It's important to note that this flag (`--hard`) is the only way to make the `reset` command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of `reset` can be pretty easily undone, but the `--hard` option cannot, since it forcibly overwrites files in the Working Directory. In this particular case, we still have the `v3` version of our file in a commit in our Git DB, and we could get it back by looking at our `reflog`, but if we had not committed it, Git still would have overwritten the file and it would be unrecoverable.

Recap

The `reset` command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch HEAD points to (*stop here if `--soft`*)
2. Make the Index look like HEAD (*stop here unless `--hard`*)
3. Make the Working Directory look like the Index

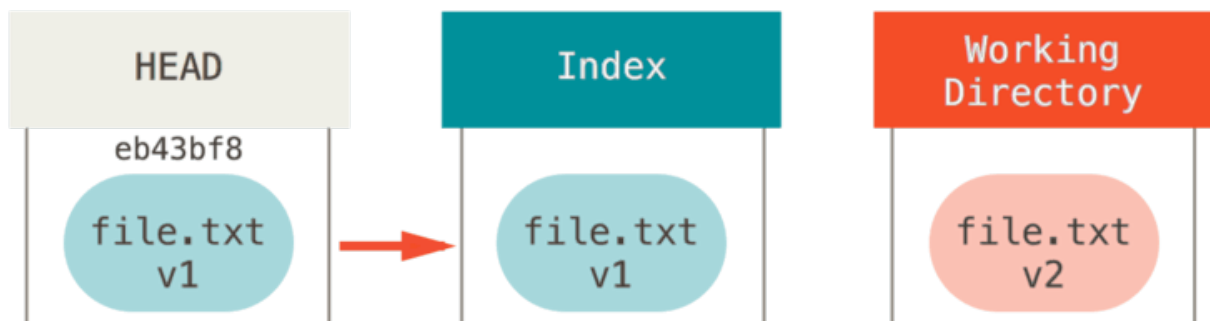
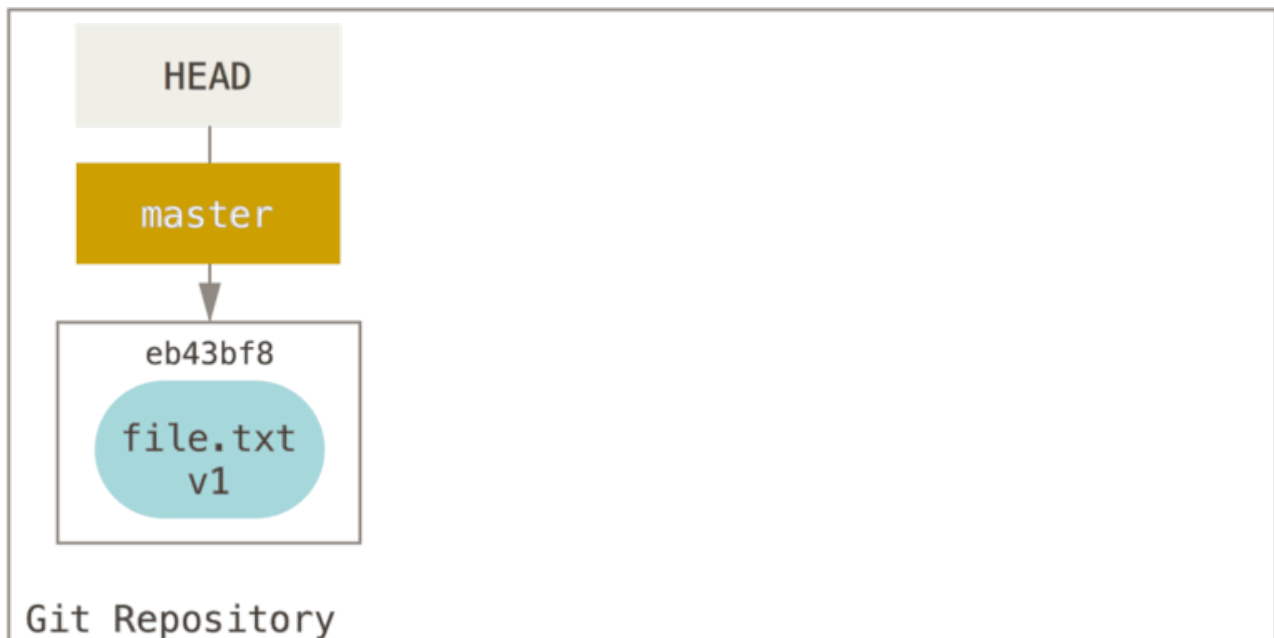
Reset With a Path

That covers the behavior of `reset` in its basic form, but you can also provide it with a path to act upon. If you specify a path, `reset` will skip step 1, and limit the remainder of its actions to a specific file or set of files. This actually sort of makes sense – HEAD is just a pointer, and you can't point to part of one commit and part of another. But the Index and Working directory *can* be partially updated, so `reset` proceeds with steps 2 and 3.

So, assume we run `git reset file.txt`. This form (since you did not specify a commit SHA-1 or branch, and you didn't specify `--soft` or `--hard`) is shorthand for `git reset --mixed HEAD file.txt`, which will:

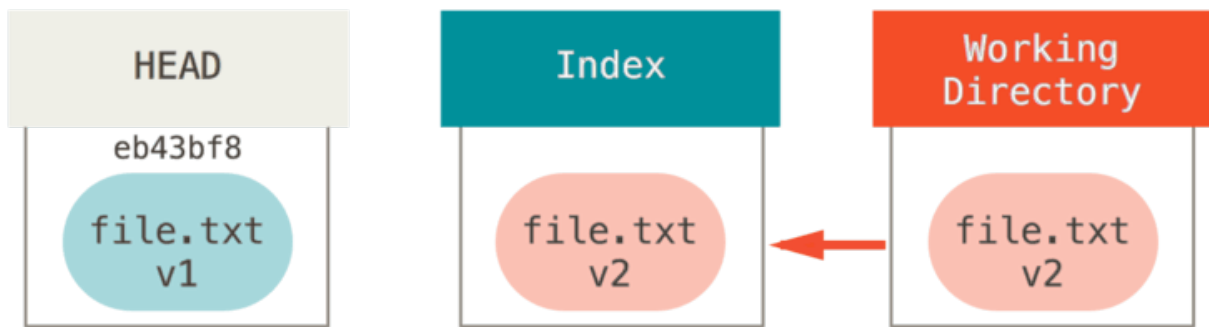
1. Move the branch HEAD points to (*skipped*)
2. Make the Index look like HEAD (*stop here*)

So it essentially just copies `file.txt` from HEAD to the Index.



git reset file.txt

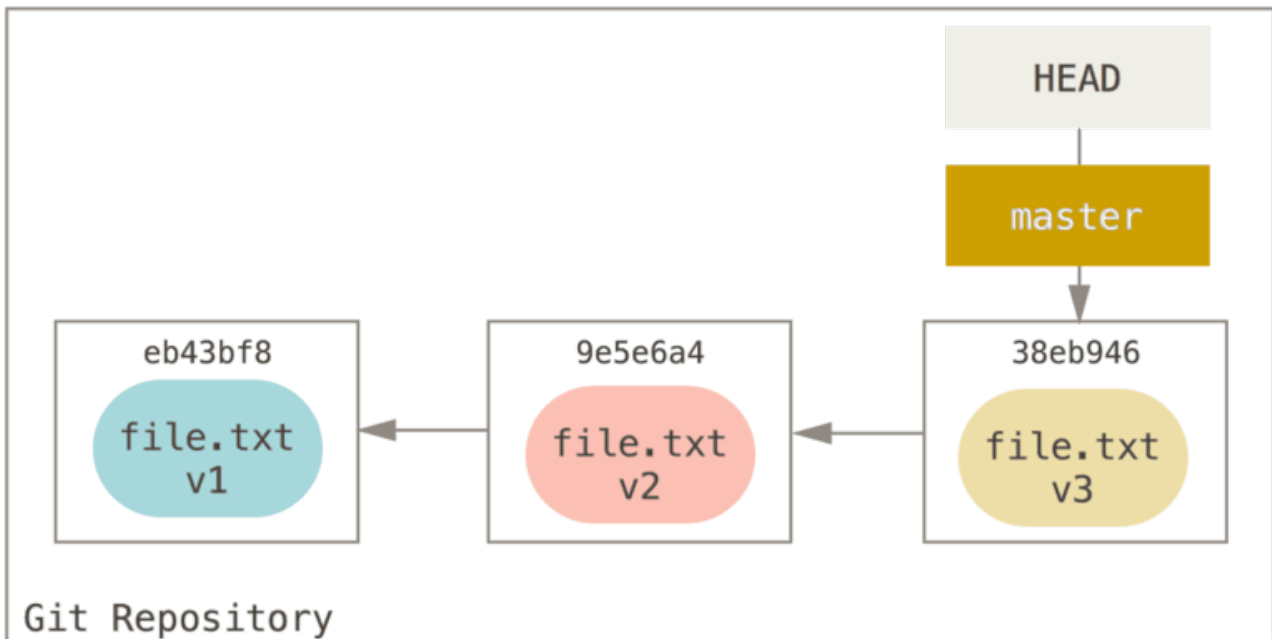
This has the practical effect of *unstaging* the file. If we look at the diagram for that command and think about what `git add` does, they are exact opposites.



`git add file.txt`

This is why the output of the `git status` command suggests that you run this to unstage a file. (See [Povrnitev datoteke iz vmesne faze](#) for more on this.)

We could just as easily not let Git assume we meant “pull the data from HEAD” by specifying a specific commit to pull that file version from. We would just run something like `git reset eb43bf file.txt`.



`git reset eb43 -- file.txt`

This effectively does the same thing as if we had reverted the content of the file to **v1** in the Working Directory, ran `git add` on it, then reverted it back to **v3** again (without actually going through all those steps). If we run `git commit` now, it will record a change that reverts that file back to **v1**, even though we never actually had it in our Working Directory again.

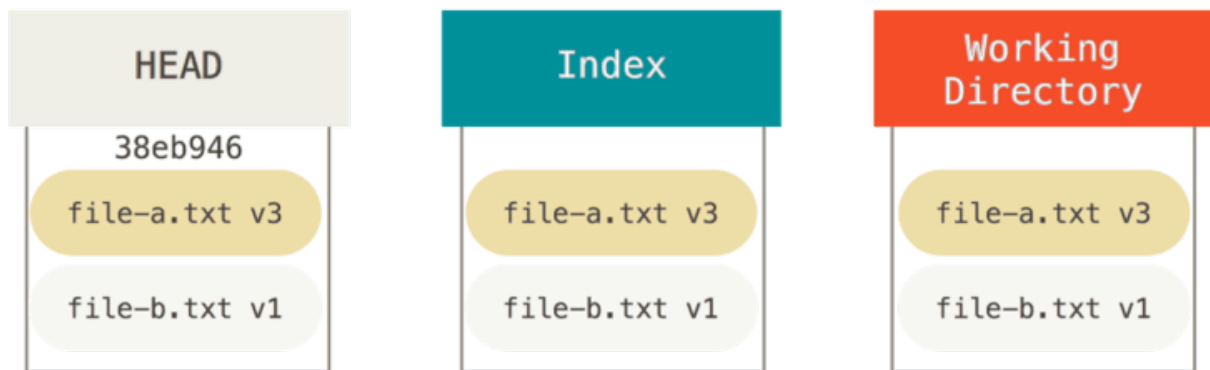
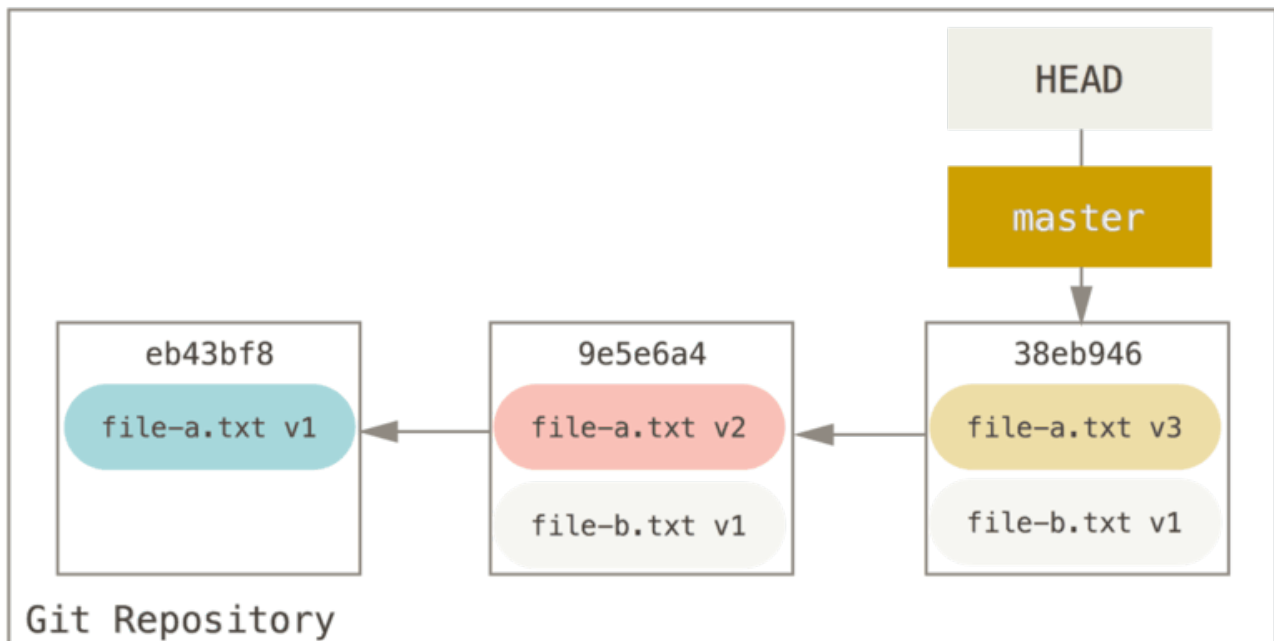
It's also interesting to note that like `git add`, the `reset` command will accept a `--patch` option to unstage content on a hunk-by-hunk basis. So you can selectively unstage or revert content.

Squashing

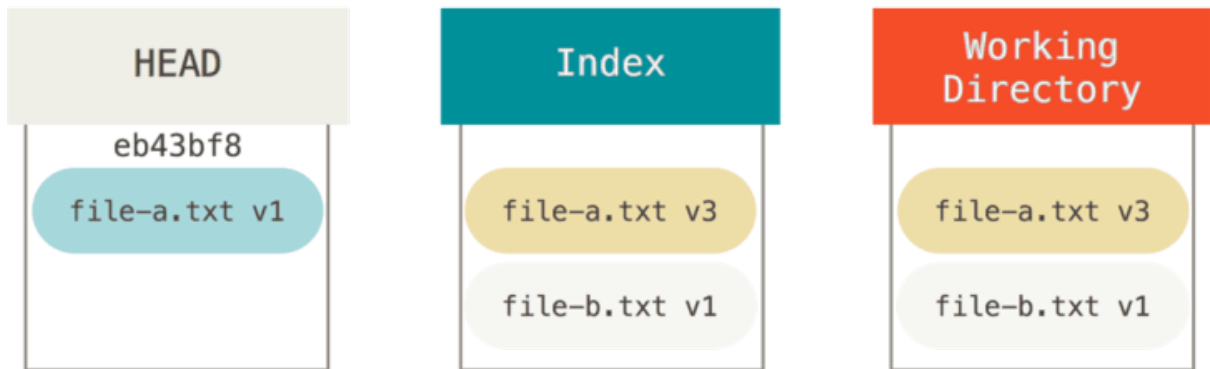
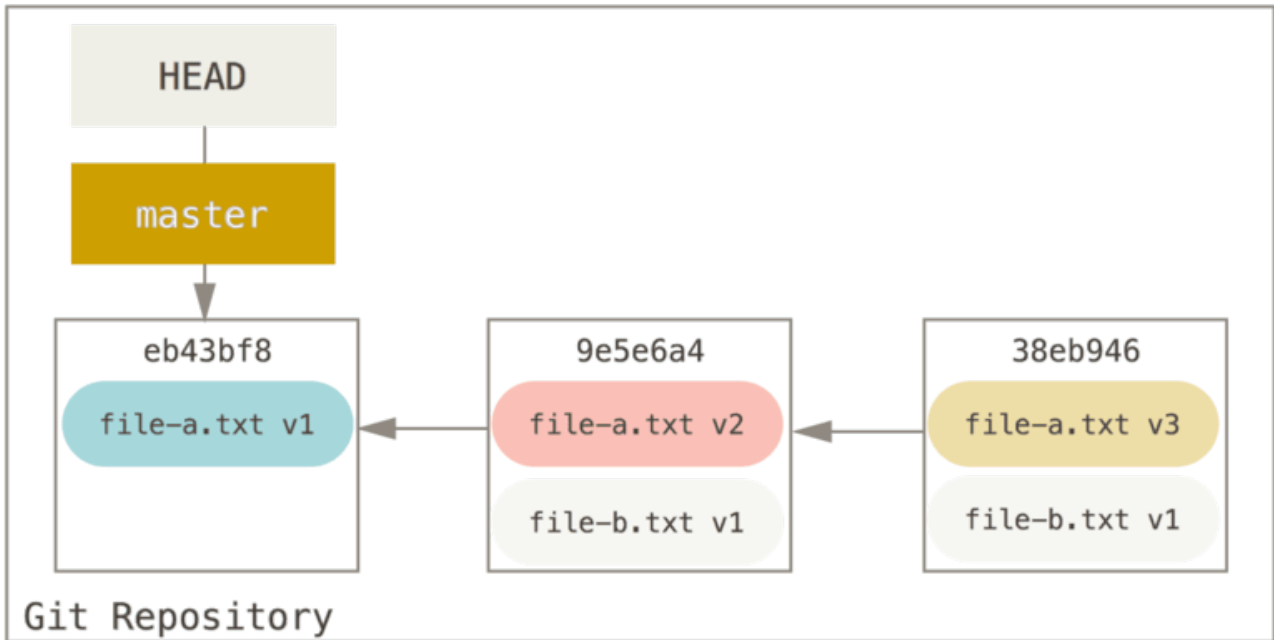
Let's look at how to do something interesting with this newfound power – squashing commits.

Say you have a series of commits with messages like “oops.”, “WIP” and “forgot this file”. You can use `reset` to quickly and easily squash them into a single commit that makes you look really smart. ([Squashing Commits](#) shows another way to do this, but in this example it's simpler to use `reset`.)

Let's say you have a project where the first commit has one file, the second commit added a new file and changed the first, and the third commit changed the first file again. The second commit was a work in progress and you want to squash it down.

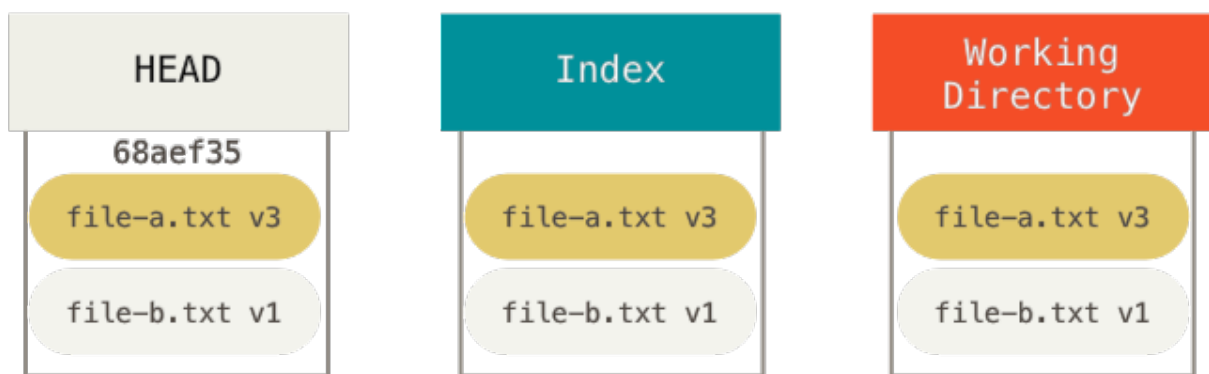
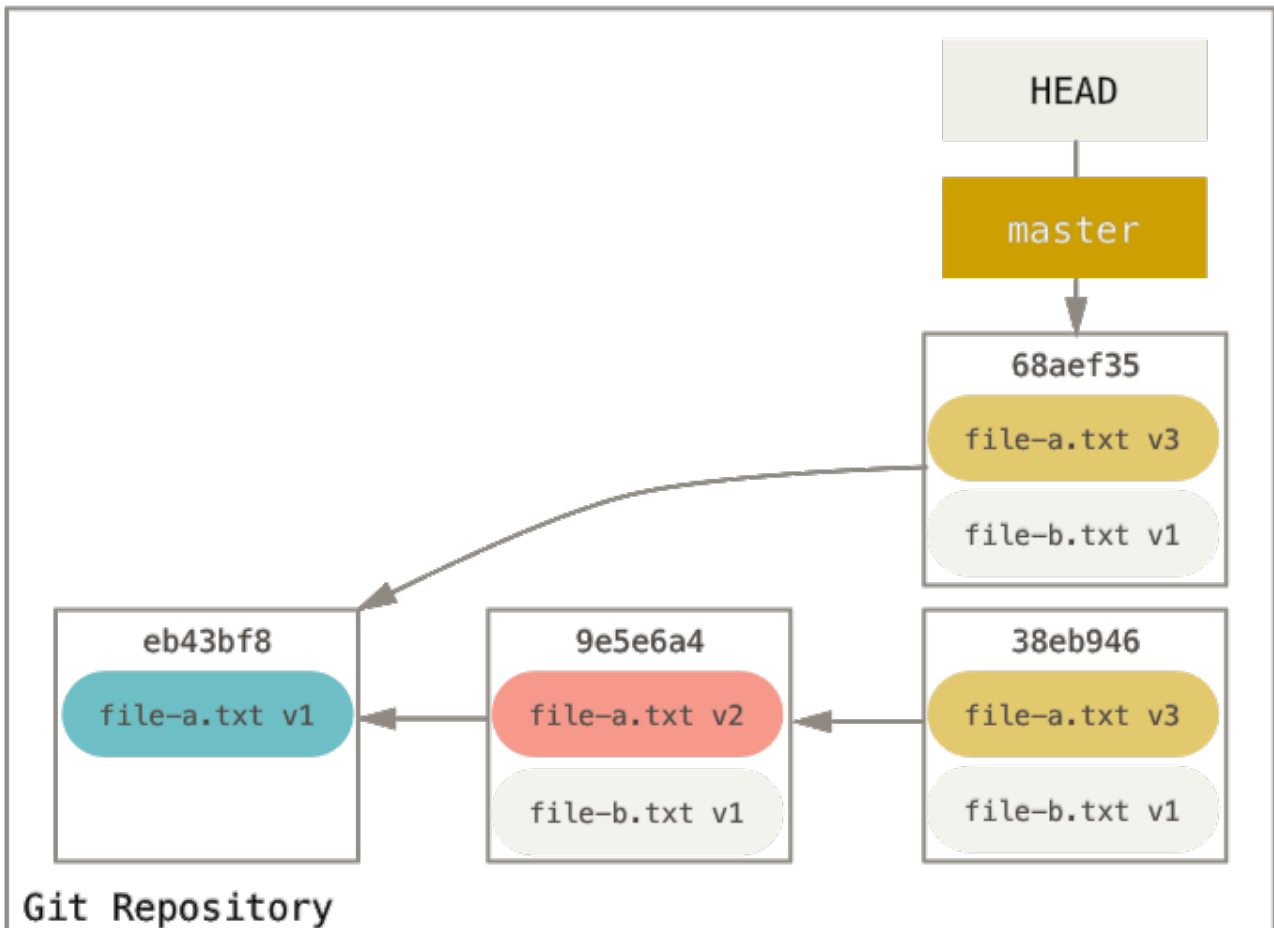


You can run `git reset --soft HEAD~2` to move the HEAD branch back to an older commit (the first commit you want to keep):



git reset --soft HEAD~2

And then simply run `git commit` again:



git commit

Now you can see that your reachable history, the history you would push, now looks like you had one commit with `file-a.txt v1`, then a second that both modified `file-a.txt` to v3 and added `file-b.txt`. The commit with the v2 version of the file is no longer in the history.

Check It Out

Finally, you may wonder what the difference between `checkout` and `reset` is. Like `reset`, `checkout` manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

Without Paths

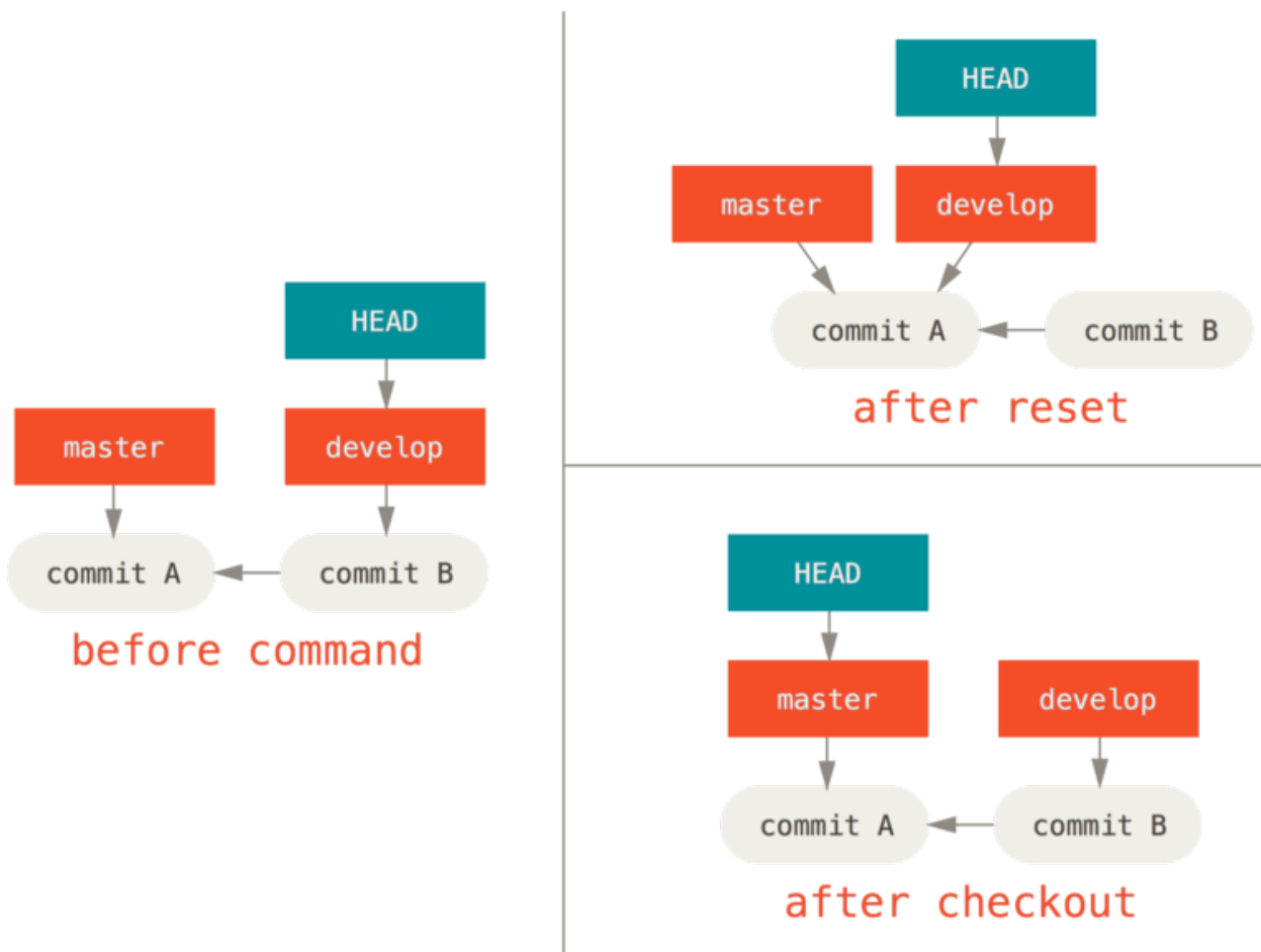
Running `git checkout [branch]` is pretty similar to running `git reset --hard [branch]` in that it updates all three trees for you to look like `[branch]`, but there are two important differences.

First, unlike `reset --hard`, `checkout` is working-directory safe; it will check to make sure it's not blowing away files that have changes to them. Actually, it's a bit smarter than that – it tries to do a trivial merge in the Working Directory, so all of the files you *haven't* changed in will be updated. `reset --hard`, on the other hand, will simply replace everything across the board without checking.

The second important difference is how it updates HEAD. Where `reset` will move the branch that HEAD points to, `checkout` will move HEAD itself to point to another branch.

For instance, say we have `master` and `develop` branches which point at different commits, and we're currently on `develop` (so HEAD points to it). If we run `git reset master`, `develop` itself will now point to the same commit that `master` does. If we instead run `git checkout master`, `develop` does not move, HEAD itself does. HEAD will now point to `master`.

So, in both cases we're moving HEAD to point to commit A, but *how* we do so is very different. `reset` will move the branch HEAD points to, `checkout` moves HEAD itself.



With Paths

The other way to run `checkout` is with a file path, which, like `reset`, does not move HEAD. It is just like `git reset [branch] file` in that it updates the index with that file at that commit, but it also overwrites the file in the working directory. It would be exactly like `git reset --hard [branch] file` (if `reset` would let you run that) – it’s not working-directory safe, and it does not move HEAD.

Also, like `git reset` and `git add`, `checkout` will accept a `--patch` option to allow you to selectively revert file contents on a hunk-by-hunk basis.

Summary

Hopefully now you understand and feel more comfortable with the `reset` command, but are probably still a little confused about how exactly it differs from `checkout` and could not possibly remember all the rules of the different invocations.

Here’s a cheat-sheet for which commands affect which trees. The “HEAD” column reads “REF” if that command moves the reference (branch) that HEAD points to, and “HEAD” if it moves HEAD itself. Pay especial attention to the *WD Safe?* column – if it says **NO**, take a second to think before running that command.

| | HEAD | Index | Workdir | WD Safe? |
|---------------------------------------|------|-------|---------|-----------|
| Commit Level | | | | |
| <code>reset --soft [commit]</code> | REF | NO | NO | YES |
| <code>reset [commit]</code> | REF | YES | NO | YES |
| <code>reset --hard [commit]</code> | REF | YES | YES | NO |
| <code>checkout [commit]</code> | HEAD | YES | YES | YES |
| File Level | | | | |
| <code>reset (commit) [file]</code> | NO | YES | NO | YES |
| <code>checkout (commit) [file]</code> | NO | YES | YES | NO |

Advanced Merging

Merging in Git is typically fairly easy. Since Git makes it easy to merge another branch multiple times, it means that you can have a very long lived branch but you can keep it up to date as you go, solving small conflicts often, rather than be surprised by one enormous conflict at the end of the series.

However, sometimes tricky conflicts do occur. Unlike some other version control systems, Git does not try to be overly clever about merge conflict resolution. Git’s philosophy is to be smart about determining when a merge resolution is unambiguous, but if there is a conflict, it does not try to be clever about automatically resolving it. Therefore, if you wait too long to merge two branches that diverge quickly, you can run into some issues.

In this section, we'll go over what some of those issues might be and what tools Git gives you to help handle these more tricky situations. We'll also cover some of the different, non-standard types of merges you can do, as well as see how to back out of merges that you've done.

Merge Conflicts

While we covered some basics on resolving merge conflicts in [Basic Merge Conflicts](#), for more complex conflicts, Git provides a few tools to help you figure out what's going on and how to better deal with the conflict.

First of all, if at all possible, try to make sure your working directory is clean before doing a merge that may have conflicts. If you have work in progress, either commit it to a temporary branch or stash it. This makes it so that you can undo **anything** you try here. If you have unsaved changes in your working directory when you try a merge, some of these tips may help you lose that work.

Let's walk through a very simple example. We have a super simple Ruby file that prints *hello world*.

```
#!/usr/bin/env ruby

def hello
  puts 'hello world'
end

hello()
```

In our repository, we create a new branch named `whitespace` and proceed to change all the Unix line endings to DOS line endings, essentially changing every line of the file, but just with whitespace. Then we change the line “hello world” to “hello mundo”.

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
  #! /usr/bin/env ruby

  def hello
-   puts 'hello world'
+   puts 'hello mundo'^M
  end

  hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
1 file changed, 1 insertion(+), 1 deletion(-)
```

Now we switch back to our `master` branch and add some documentation for the function.

```

$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
  #! /usr/bin/env ruby

+# prints out a greeting
  def hello
    puts 'hello world'
  end

$ git commit -am 'document the function'
[master bec6336] document the function
1 file changed, 1 insertion(+)

```

Now we try to merge in our `whitespace` branch and we'll get conflicts because of the whitespace changes.

```

$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.

```

Aborting a Merge

We now have a few options. First, let's cover how to get out of this situation. If you perhaps weren't expecting conflicts and don't want to quite deal with the situation yet, you can simply back out of the merge with `git merge --abort`.

```

$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master

```

The `git merge --abort` option tries to revert back to your state before you ran the merge. The only cases where it may not be able to do this perfectly would be if you had unstashed, uncommitted changes in your working directory when you ran it,

otherwise it should work fine.

If for some reason you just want to start over, you can also run `git reset --hard HEAD`, and your repository will be back to the last committed state. Remember that any uncommitted work will be lost, so make sure you don't want any of your changes.

Ignoring Whitespace

In this specific case, the conflicts are whitespace related. We know this because the case is simple, but it's also pretty easy to tell in real cases when looking at the conflict because every line is removed on one side and added again on the other. By default, Git sees all of these lines as being changed, so it can't merge the files.

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with `-Xignore-all-space` or `-Xignore-space-change`. The first option ignores whitespace **completely** when comparing lines, the second treats sequences of one or more whitespace characters as equivalent.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

Since in this case, the actual file changes were not conflicting, once we ignore the whitespace changes, everything merges just fine.

This is a lifesaver if you have someone on your team who likes to occasionally reformat everything from spaces to tabs or vice-versa.

Manual File Re-merging

Though Git handles whitespace pre-processing pretty well, there are other types of changes that perhaps Git can't handle automatically, but are scriptable fixes. As an example, let's pretend that Git could not handle the whitespace change and we needed to do it by hand.

What we really need to do is run the file we're trying to merge in through a `dos2unix` program before trying the actual file merge. So how would we do that?

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we're merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under "stages" which each have numbers associated with them. Stage 1 is

the common ancestor, stage 2 is your version and stage 3 is from the `MERGE_HEAD`, the version you're merging in ("theirs").

You can extract a copy of each of these versions of the conflicted file with the `git show` command and a special syntax.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

If you want to get a little more hard core, you can also use the `ls-files -u` plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1 hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2 hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3 hello.rb
```

The `:1:hello.rb` is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known `git merge-file` command which does just that.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
  hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

  +# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

At this point we have nicely merged the file. In fact, this actually works better than

the `ignore-space-change` option because this actually fixes the whitespace changes before merge instead of simply ignoring them. In the `ignore-space-change` merge, we actually ended up with a few lines with DOS line endings, making things mixed.

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask `git diff` to compare what is in your working directory that you're about to commit as the result of the merge to any of these stages. Let's go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@

# prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

So here we can easily see that what happened in our branch, what we're actually introducing to this file with this merge, is changing that single line.

If we want to see how the result of the merge differed from what was on their side, you can run `git diff --theirs`. In this and the following example, we have to use `-b` to strip out the whitespace because we're comparing it to what is in Git, not our cleaned up `hello.theirs.rb` file.


```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
  #! /usr/bin/env ruby

+# prints out a greeting
  def hello
    puts 'hello mundo'
  end
```

Finally, you can see how the file has changed from both sides with `git diff --base`.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
  #! /usr/bin/env ruby

+# prints out a greeting
  def hello
-   puts 'hello world'
+   puts 'hello mundo'
  end

  hello()
```

At this point we can use the `git clean` command to clear out the extra files we created to do the manual merge but no longer need.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

Checking Out Conflicts

Perhaps we're not happy with the resolution at this point for some reason, or maybe manually editing one or both sides still didn't work well and we need more context.

Let's change up the example a little. For this example, we have two longer lived

branches that each have a few commits in them but create a legitimate content conflict when merged.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|/
* b7dcc89 initial hello world code
```

We now have three unique commits that live only on the `master` branch and three others that live on the `mundo` branch. If we try to merge the `mundo` branch in, we get a conflict.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.
```

We would like to see what the merge conflict is. If we open up the file, we'll see something like this:

```
#!/usr/bin/env ruby

def hello
<<<<<<< HEAD
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>>>> mundo
end

hello()
```

Both sides of the merge added content to this file, but some of the commits modified the file in the same place that caused this conflict.

Let's explore a couple of tools that you now have at your disposal to determine how this conflict came to be. Perhaps it's not obvious how exactly you should fix this conflict. You need more context.

One helpful tool is `git checkout` with the `--conflict` option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass `--conflict` either `diff3` or `merge` (which is the default). If you pass it `diff3`, Git will use a slightly different version of conflict markers, not only giving you the “ours” and “theirs” versions, but also the “base” version inline to give you more context.

```
$ git checkout --conflict=diff3 hello.rb
```

Once we run that, the file will look like this instead:

```
#!/usr/bin/env ruby

def hello
  <<<<<<< ours
    puts 'hola world'
  ||||| base
    puts 'hello world'
  =====
    puts 'hello mundo'
  >>>>>> theirs
end

hello()
```

If you like this format, you can set it as the default for future merge conflicts by setting the `merge.conflictstyle` setting to `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

The `git checkout` command can also take `--ours` and `--theirs` options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing.

Merge Log

Another useful tool when resolving merge conflicts is `git log`. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in [Triple Dot](#).

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

That's a nice list of the six total commits involved, as well as which line of development each commit was on.

We can further simplify this though to give us much more specific context. If we add the `--merge` option to `git log`, it will only show the commits in either side of the merge that touch a file that's currently conflicted.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

If you run that with the `-p` option instead, you get just the diffs to the file that ended up in conflict. This can be **really** helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

Combined Diff Format

Since Git stages any merge results that are successful, when you run `git diff` while in a conflicted merge state, you only get what is currently still in conflict. This can be helpful to see what you still have to resolve.

When you run `git diff` directly after a merge conflict, it will give you information in a rather unique diff output format.

```

$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
  #! /usr/bin/env ruby

  def hello
++<<<<<<< HEAD
  + puts 'hola world'
++=====
  + puts 'hello mundo'
++>>>>>>> mundo
  end

  hello()

```

The format is called “Combined Diff” and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the “ours” branch and the file in your working directory and the second column does the same between the “theirs” branch and your working directory copy.

So in that example you can see that the <<<<<<< and >>>>>>> lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we’re expected to remove them.

If we resolve the conflict and run `git diff` again, we’ll see the same thing, but it’s a little more useful.

```

$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #! /usr/bin/env ruby

  def hello
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
  end

  hello()

```

This shows us that “hola world” was in our side but not in the working copy, that

“hello mundo” was in their side but not in the working copy and finally that “hola mundo” was not in either side but is now in the working copy. This can be useful to review before committing the resolution.

You can also get this from the `git log` for any merge to see how something was resolved after the fact. Git will output this format if you run `git show` on a merge commit, or if you add a `--cc` option to a `git log -p` (which by default only shows patches for non-merge commits).

```
$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200
```

```
    Merge branch 'mundo'
```

```
    Conflicts:
        hello.rb
```

```
diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
    #! /usr/bin/env ruby

    def hello
-   puts 'hola world'
-   puts 'hello mundo'
++  puts 'hola mundo'
    end

    hello()
```

Undoing Merges

Now that you know how to create a merge commit, you’ll probably make some by mistake. One of the great things about working with Git is that it’s okay to make mistakes, because it’s possible (and in many cases easy) to fix them.

Merge commits are no different. Let’s say you started work on a topic branch, accidentally merged it into `master`, and now your commit history looks like this:

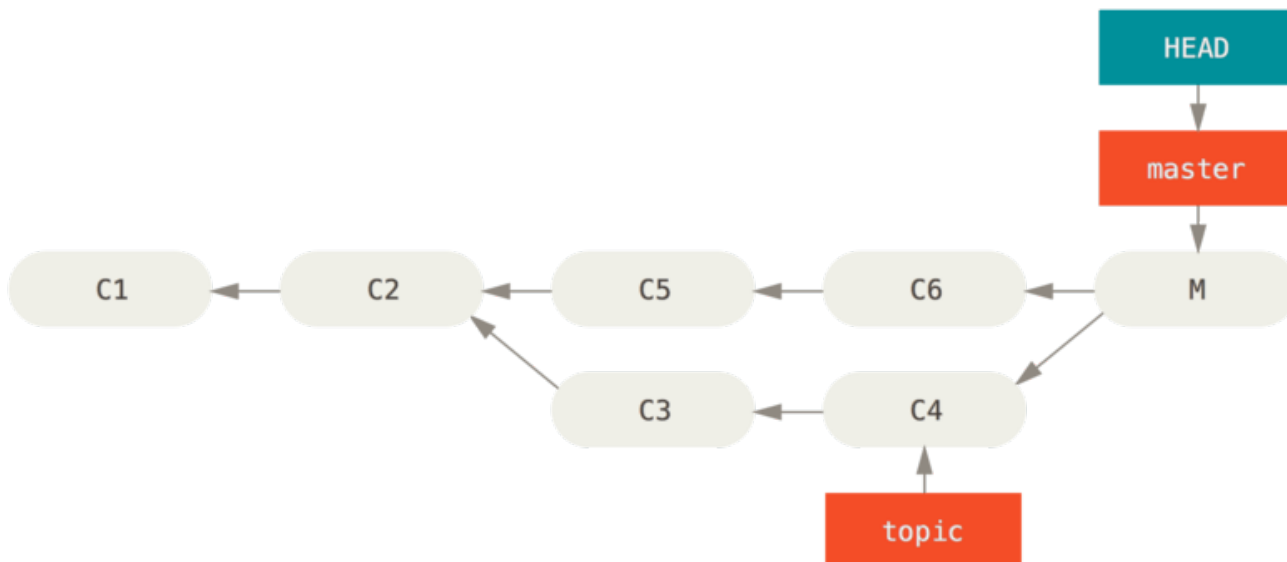


Figure 138. Accidental merge commit

There are two ways to approach this problem, depending on what your desired outcome is.

Fix the references

If the unwanted merge commit only exists on your local repository, the easiest and best solution is to move the branches so that they point where you want them to. In most cases, if you follow the errant `git merge` with `git reset --hard HEAD~`, this will reset the branch pointers so they look like this:

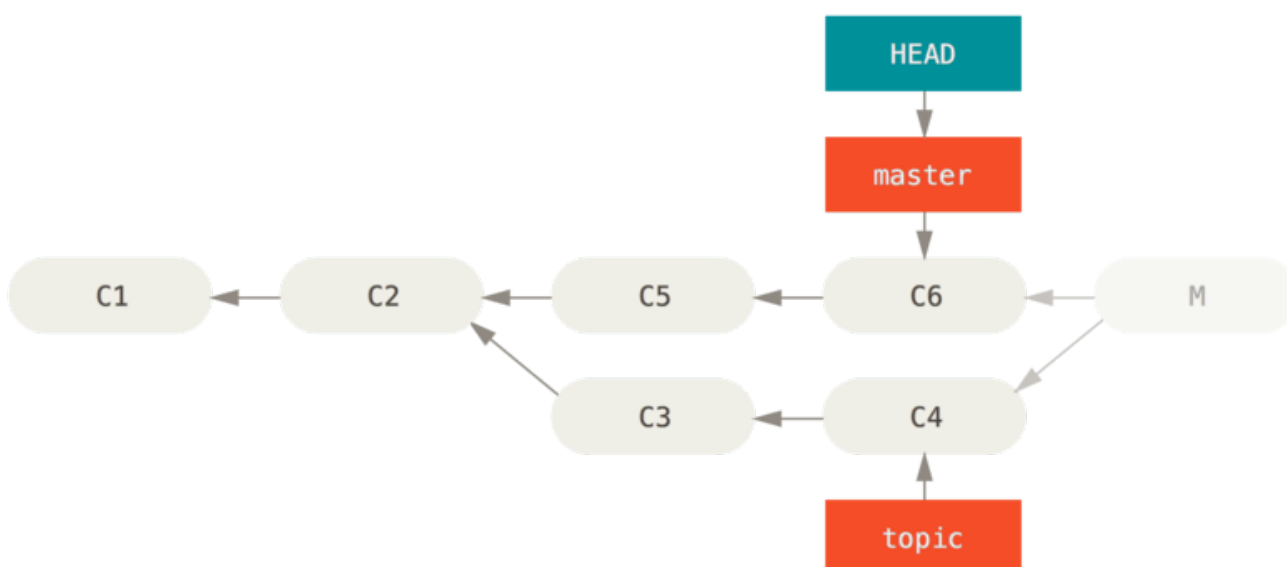


Figure 139. History after `git reset --hard HEAD~`

We covered `reset` back in [Reset Demystified](#), so it shouldn't be too hard to figure out what's going on here. Here's a quick refresher: `reset --hard` usually goes through three steps:

1. Move the branch HEAD points to. In this case, we want to move `master` to where it was before the merge commit (C6).

2. Make the index look like HEAD.
3. Make the working directory look like the index.

The downside of this approach is that it's rewriting history, which can be problematic with a shared repository. Check out [Nevarnosti ponovnega baziranja](#) for more on what can happen; the short version is that if other people have the commits you're rewriting, you should probably avoid `reset`. This approach also won't work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

Reverse the commit

If moving the branch pointers around isn't going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a "revert", and in this particular scenario, you'd invoke it like this:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

The `-m 1` flag indicates which parent is the "mainline" and should be kept. When you invoke a merge into HEAD (`git merge topic`), the new commit has two parents: the first one is HEAD (C6), and the second is the tip of the branch being merged in (C4). In this case, we want to undo all the changes introduced by merging in parent #2 (C4), while keeping all the content from parent #1 (C6).

The history with the revert commit looks like this:

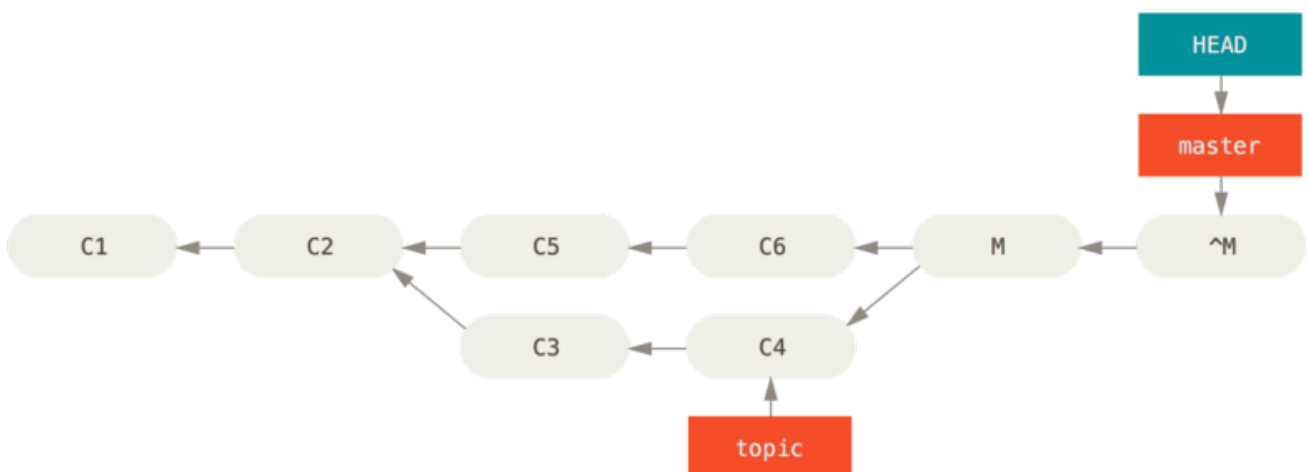


Figure 140. History after `git revert -m 1`

The new commit `^M` has exactly the same contents as `C6`, so starting from here it's as if the merge never happened, except that the now-unmerged commits are still in HEAD's history. Git will get confused if you try to merge `topic` into `master` again:


```
$ git merge topic
Already up-to-date.
```

There's nothing in `topic` that isn't already reachable from `master`. What's worse, if you add work to `topic` and merge again, Git will only bring in the changes *since* the reverted merge:

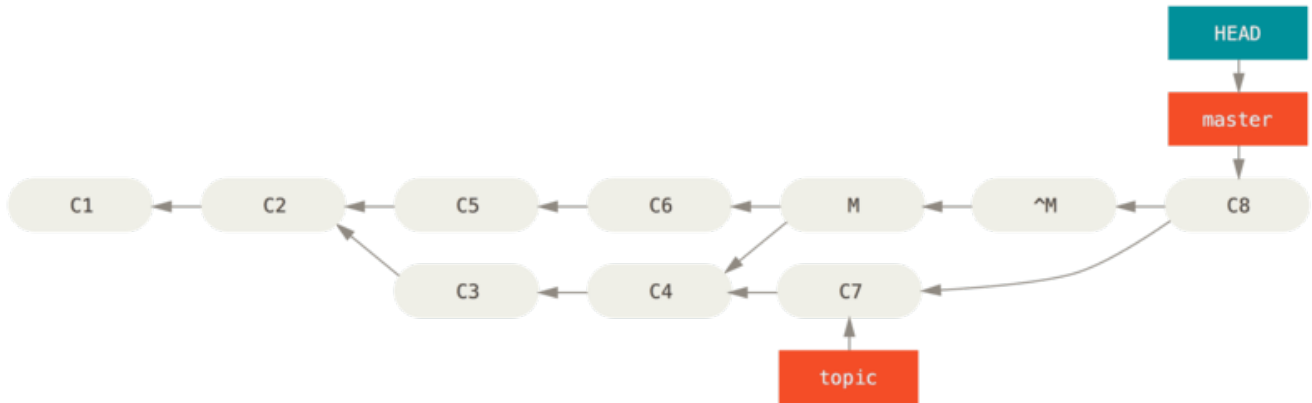


Figure 141. History with a bad merge

The best way around this is to un-revert the original merge, since now you want to bring in the changes that were reverted out, **then** create a new merge commit:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'"
$ git merge topic
```

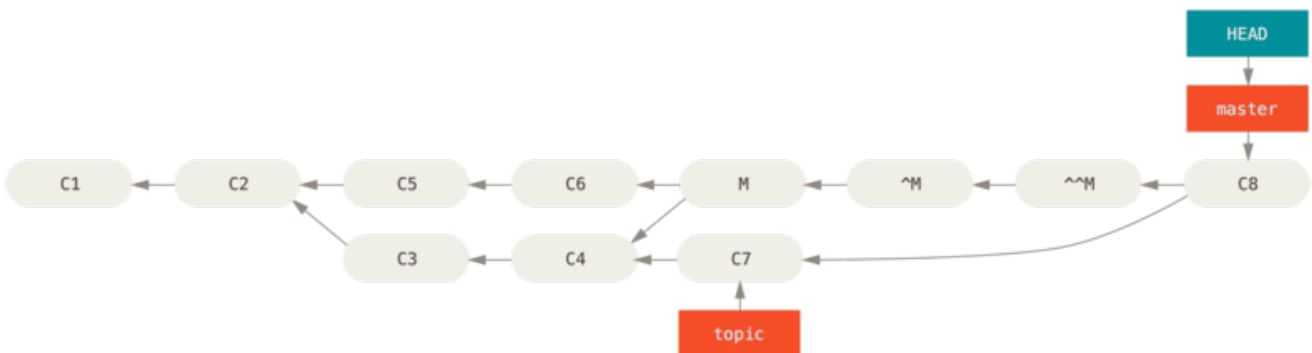


Figure 142. History after re-merging a reverted merge

In this example, `M` and `^M` cancel out. `^^M` effectively merges in the changes from `C3` and `C4`, and `C8` merges in the changes from `C7`, so now `topic` is fully merged.

Other Types of Merges

So far we've covered the normal merge of two branches, normally handled with what is called the "recursive" strategy of merging. There are other ways to merge branches together however. Let's cover a few of them quickly.

Our or Theirs Preference

First of all, there is another useful thing we can do with the normal “recursive” mode of merging. We’ve already seen the `ignore-all-space` and `ignore-space-change` options which are passed with a `-X` but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually resolve the conflict, you can pass the `merge` command either a `-Xours` or `-Xtheirs`.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

If we go back to the “hello world” example we were using before, we can see that merging in our branch causes conflicts.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

However if we run it with `-Xours` or `-Xtheirs` it does not.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
 hello.rb | 2 +-
 test.sh  | 2 ++
 2 files changed, 3 insertions(+), 1 deletion(-)
 create mode 100644 test.sh
```

In that case, instead of getting conflict markers in the file with “hello mundo” on one side and “hola world” on the other, it will simply pick “hola world”. However, all the other non-conflicting changes on that branch are merged successfully in.

This option can also be passed to the `git merge-file` command we saw earlier by running something like `git merge-file --ours` for individual file merges.

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the “ours” merge *strategy*. This is different from the “ours” recursive merge *option*.

This will basically do a fake merge. It will record a new merge commit with both

branches as parents, but it will not even look at the branch you're merging in. It will simply record as the result of the merge the exact code in your current branch.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

You can see that there is no difference between the branch we were on and the result of the merge.

This can often be useful to basically trick Git into thinking that a branch is already merged when doing a merge later on. For example, say you branched off a `release` branch and have done some work on it that you will want to merge back into your `master` branch at some point. In the meantime some bugfix on `master` needs to be backported into your `release` branch. You can merge the bugfix branch into the `release` branch and also `merge -s ours` the same branch into your `master` branch (even though the fix is already there) so when you later merge the `release` branch again, there are no conflicts from the bugfix.

Združevanje pod dreves

Ideja združevanja pod dreves (subtree merge) je, da imate dva projekta in eden izmed projektov je preslikan v poddirektorij drugega in obratno. Ko določite združitev pod dreves, je Git pogostokrat dovolj pameten, da ugotovi, da je en pod drevo drugega in ju ustrezno združi.

Šli bomo skozi primer dodajanja ločenega projekta v obstoječi projekt in nato združili kodo drugega v poddirektorij prvega.

Najprej bomo dodali aplikacijo Rack v naš projekt. Dodali bomo projekt Rack kot oddaljeno referenco v vaš lasten projekt in ga nato izpisali v svojo lastno vejo:

```

$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
* [new branch]      build      -> rack_remote/build
* [new branch]      master     -> rack_remote/master
* [new branch]      rack-0.4   -> rack_remote/rack-0.4
* [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"

```

Sedaj imate vrh projekta Rack v vaši veji `rack_branch` in vaš lastni projekt v veji `master`. Če izpišete enega in nato drugega lahko vidite, da imata različne vrhe projektov:

```

$ ls
AUTHORS      KNOWN-ISSUES  Rakefile     contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README

```

To je nekako čuden koncept. Ne vse veje v vašem repozitoriju dejansko morajo biti veje istega projekta. Ni pogosto, ker je redko v pomoč, vendar je precej enostavno imeti veje, ki vsebujejo popolnoma različne zgodovine.

V tem primeru želimo potegniti projekt Rack v naš projekt `master` kot poddirektorij. To lahko naredimo v Git-u z `git read-tree`. Več o `read-tree` in njegovih prijateljih se bomo naučili v [Notranjost Git-a](#), vendar za sedaj vedite, da prebere vrh drevesa ene veje v vašo trenutno vmesno fazo in delujoči direktorij. Smo samo preklopili nazaj na našo vejo `master` in potegnimo vejo `rack_branch` v poddirektorij `rack` naše veje `master` našega glavnega projekta:

```

$ git read-tree --prefix=rack/ -u rack_branch

```

Ko pošljemo izgleda kot da imamo vse datoteke Rack pod tem poddirektorijem - kakor če smo jih kopirali iz tarball-a. Kar postane zanimivo je, da lahko precej enostavno združimo spremembe iz ene veje v drugo. Torej če projekt Rack posodobi, lahko potegnemo spremembe navzgor s preklopom na to vejo in potegom:

```
$ git checkout rack_branch
$ git pull
```

Nato lahko združimo te spremembe nazaj v našo vejo `master`. Da potegnemo spremembe in vnaprej napolnimo sporočilo pošiljanja, uporabimo opcijo `--squash` kot tudi rekurzivno strateško združevalno opcijo `-Xsubtree`. (Rekurzivna strategija je tu privzeta, vendar jo vključujemo zaradi jasnosti.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Vse spremembe iz projekta Rack so združene in pripravljene za pošiljanje lokalno. Lahko tudi naredite nasprotno - naredite spremembe v poddirektoriju `rack` vaše veje `master` in jih nato kasneje združite v vašo vejo `rack_branch`, da jih pošljete vzdrževalcem ali potisnete navzgor.

To nam da način imeti potek dela nekako podoben poteku dela s submoduleli brez uporabe submodulelov (kar bomo pokrili v [Submodules](#)). Lahko obdržimo veje z ostalih povezanih projektov v našem repozitoriju in jih občasno pod drevesno združimo v naš projekt. Na neke načine je lepo, na primer vsa koda je poslana na eno mesto. Vendar ima ostale slabosti v tem, da je malo bolj kompleksna in enostavna za narediti napake in ponovno integrirati spremembe ali po nesreči potisniti vejo v nepovezani repozitorij.

Druga nekoliko čudna stvar je, da dobiti razliko med tem, kar imate v vašem poddirektoriju `rack` in kodo v vaši veji `rack_branch` - da vidite, če jih potrebujete združiti - ne morete normalno uporabiti ukaza `diff`. Namesto tega morate pognati `git diff-tree` z vejo, ki jo želite primerjati:

```
$ git diff-tree -p rack_branch
```

Ali primerjati, kaj je v vašem poddirektoriju `rack` s tem, kaje je bila veja `master` na strežniku zadnjič, ko ste ujeli, lahko poženete

```
$ git diff-tree -p rack_remote/master
```

Rerere

The `git rerere` functionality is a bit of a hidden feature. The name stands for “reuse recorded resolution” and as the name implies, it allows you to ask Git to remember how you’ve resolved a hunk conflict so that the next time it sees the same conflict, Git can automatically resolve it for you.

There are a number of scenarios in which this functionality might be really handy. One of the examples that is mentioned in the documentation is if you want to make sure a long lived topic branch will merge cleanly but don't want to have a bunch of intermediate merge commits. With `rerere` turned on you can merge occasionally, resolve the conflicts, then back out the merge. If you do this continuously, then the final merge should be easy because `rerere` can just do everything for you automatically.

This same tactic can be used if you want to keep a branch rebased so you don't have to deal with the same rebasing conflicts each time you do it. Or if you want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead - you likely won't have to do all the same conflicts again.

Another situation is where you merge a bunch of evolving topic branches together into a testable head occasionally, as the Git project itself often does. If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

To enable the `rerere` functionality, you simply have to run this config setting:

```
$ git config --global rerere.enabled true
```

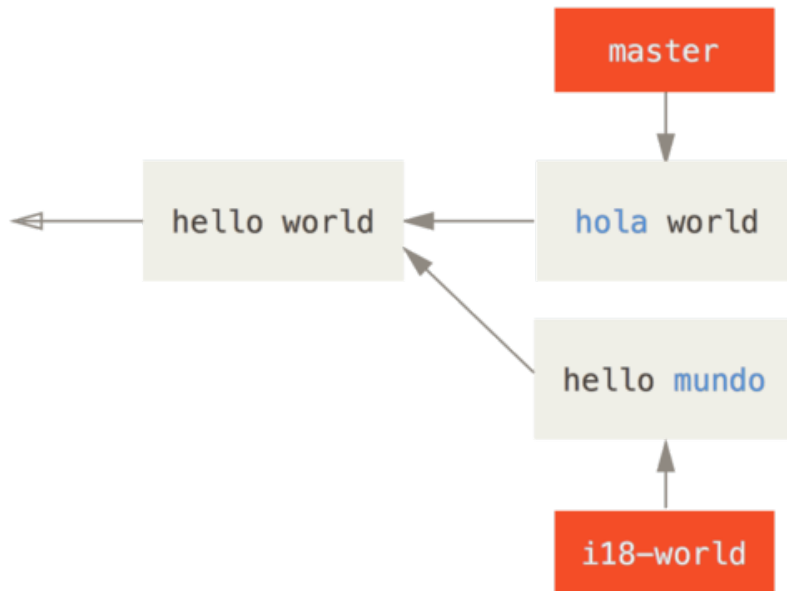
You can also turn it on by creating the `.git/rr-cache` directory in a specific repository, but the config setting is clearer and it can be done globally.

Now let's see a simple example, similar to our previous one. Let's say we have a file that looks like this:

```
#!/usr/bin/env ruby

def hello
  puts 'hello world'
end
```

In one branch we change the word "hello" to "hola", then in another branch we change the "world" to "mundo", just like before.



When we merge the two branches together, we'll get a merge conflict:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

You should notice the new line `Recorded preimage for FILE` in there. Otherwise it should look exactly like a normal merge conflict. At this point, `rerere` can tell us a few things. Normally, you might run `git status` at this point to see what all conflicted:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#   both modified:   hello.rb
#
```

However, `git rerere` will also tell you what it has recorded the pre-merge state for with `git rerere status`:

```
$ git rerere status
hello.rb
```

And `git rerere diff` will show the current state of the resolution - what you started with to resolve and what you've resolved it to.

```

$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
  #! /usr/bin/env ruby

  def hello
- <<<<<<<
- puts 'hello mundo'
- =====
+ <<<<<<< HEAD
+   puts 'hola world'
- >>>>>>>
+ =====
+ puts 'hello mundo'
+ >>>>>>> i18n-world
  end

```

Also (and this isn't really related to `rerere`), you can use `ls-files -u` to see the conflicted files and the before, left and right versions:

```

$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3 hello.rb

```

Now you can resolve it to just be `puts 'hola mundo'` and you can run the `rerere diff` command again to see what `rerere` will remember:

```

$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
  #! /usr/bin/env ruby

  def hello
- <<<<<<<
- puts 'hello mundo'
- =====
- puts 'hola world'
- >>>>>>>
+ puts 'hola mundo'
  end

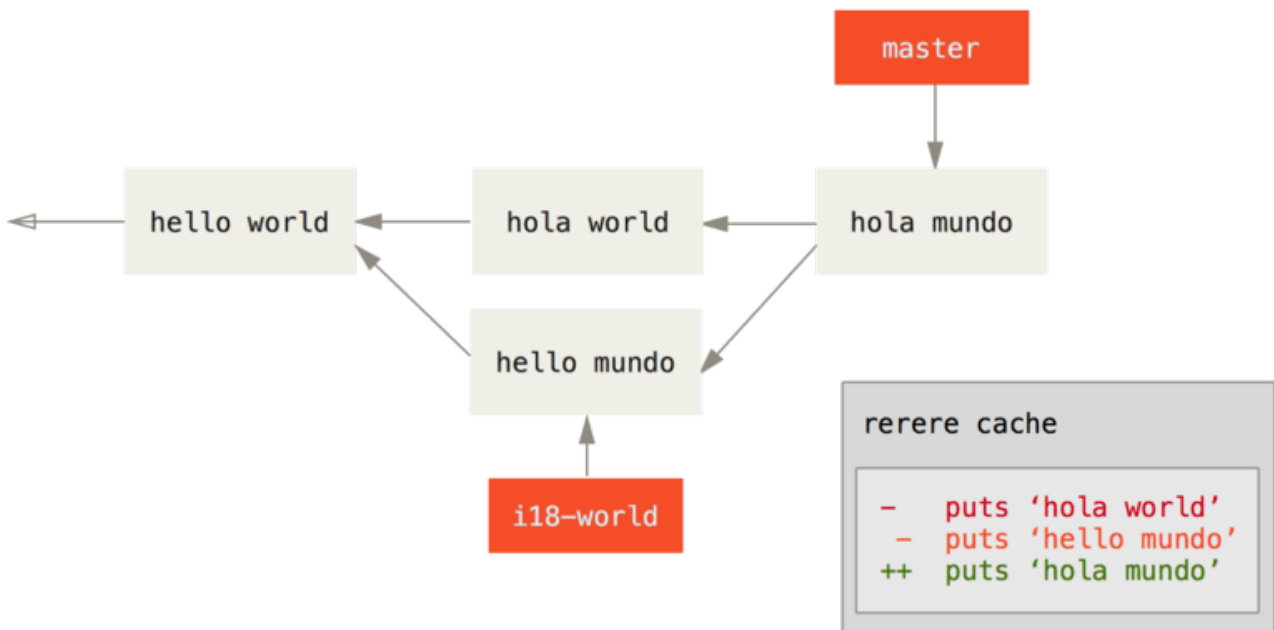
```

So that basically says, when Git sees a hunk conflict in a `hello.rb` file that has “hello mundo” on one side and “hola world” on the other, it will resolve it to “hola mundo”.

Now we can mark it as resolved and commit it:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

You can see that it "Recorded resolution for FILE".



Now, let's undo that merge and then rebase it on top of our master branch instead. We can move our branch back by using `reset` as we saw in [Reset Demystified](#).

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Our merge is undone. Now let's rebase the topic branch.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

Now, we got the same merge conflict like we expected, but take a look at the **Resolved FILE using previous resolution** line. If we look at the file, we'll see that it's already been resolved, there are no merge conflict markers in it.

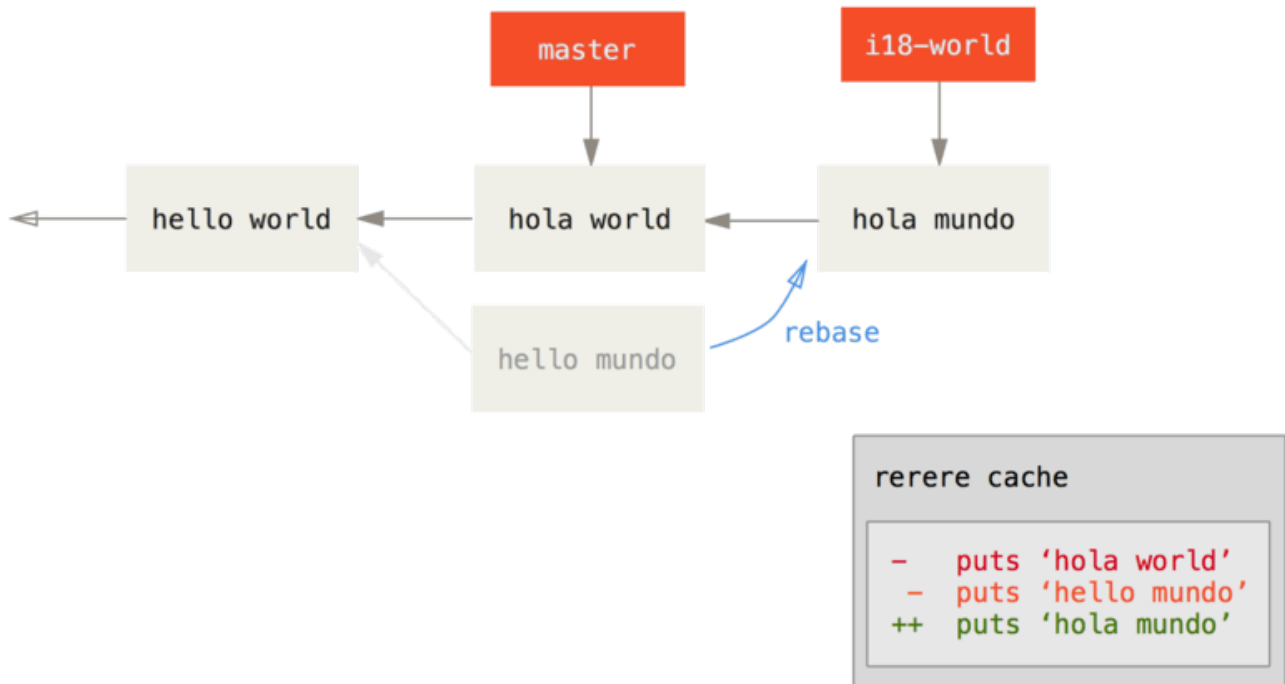
```
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Also, **git diff** will show you how it was automatically re-resolved:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
  #!/usr/bin/env ruby

  def hello
-   puts 'hola world'
-   puts 'hello mundo'
++  puts 'hola mundo'
  end
```



You can also recreate the conflicted file state with the `checkout` command:

```

$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<<<< ours
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>>> theirs
end

```

We saw an example of this in [Advanced Merging](#). For now though, let's re-resolve it by just running `rerere` again:

```

$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end

```

We have re-resolved the file automatically using the `rerere` cached resolution. You can now add and continue the rebase to complete it.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

So, if you do a lot of re-merges, or want to keep a topic branch up to date with your master branch without a ton of merges, or you rebase often, you can turn on `rerere` to help your life out a bit.

Debugging with Git

Git also provides a couple of tools to help you debug issues in your projects. Because Git is designed to work with nearly any type of project, these tools are pretty generic, but they can often help you hunt for a bug or culprit when things go wrong.

File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows you what commit was the last to modify each line of any file. So, if you see that a method in your code is buggy, you can annotate the file with `git blame` to see when each line of the method was last edited and by whom. This example uses the `-L` option to limit the output to lines 12 through 22:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

Notice that the first field is the partial SHA-1 of the commit that last modified that line. The next two fields are values extracted from that commit—the author name and the authored date of that commit – so you can easily see who modified that line and when. After that come the line number and the content of the file. Also note the `^4832fe2` commit lines, which designate that those lines were in this file’s original commit. That commit is when this file was first added to this project, and those lines have been unchanged since. This is a tad confusing, because now you’ve seen at least three different ways that Git uses the `^` to modify a commit SHA-1, but that is what it means here.

Another cool thing about Git is that it doesn't track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass `-C` to `git blame`, Git analyzes the file you're annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. For example, say you are refactoring a file named `GITServerHandler.m` into multiple files, one of which is `GITPackUpload.m`. By blaming `GITPackUpload.m` with the `-C` option, you can see where sections of the code originally came from:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 146) NSString *parentSha;
ad11ac80 GITPackUpload.m (Scott 2009-03-24 147) GITCommit *commit = [g
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151) if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152) [refDict setObject
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

This is really useful. Normally, you get as the original commit the commit where you copied the code over, because that is the first time you touched those lines in this file. Git tells you the original commit where you wrote those lines, even if it was in another file.

Binary Search

Annotating a file helps if you know where the issue is to begin with. If you don't know what is breaking, and there have been dozens or hundreds of commits since the last state where you know the code worked, you'll likely turn to `git bisect` for help. The `bisect` command does a binary search through your commit history to help you identify as quickly as possible which commit introduced an issue.

Let's say you just pushed out a release of your code to a production environment, you're getting bug reports about something that wasn't happening in your development environment, and you can't imagine why the code is doing that. You go back to your code, and it turns out you can reproduce the issue, but you can't figure out what is going wrong. You can bisect the code to find out. First you run `git bisect start` to get things going, and then you use `git bisect bad` to tell the system that the current commit you're on is broken. Then, you must tell bisect when the last known good state was, using `git bisect good [good_commit]`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccec5f9350d878ce677feb13d3b2] error handling on repo
```

Git figured out that about 12 commits came between the commit you marked as the last good commit (v1.0) and the current bad version, and it checked out the middle one for you. At this point, you can run your test to see if the issue exists as of this commit. If it does, then it was introduced sometime before this middle commit; if it doesn't, then the problem was introduced sometime after the middle commit. It turns out there is no issue here, and you tell Git that by typing `git bisect good` and continue your journey:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Now you're on another commit, halfway between the one you just tested and your bad commit. You run your test again and find that this commit is broken, so you tell Git that with `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

This commit is fine, and now Git has all the information it needs to determine where the issue was introduced. It tells you the SHA-1 of the first bad commit and show some of the commit information and which files were modified in that commit so you can figure out what happened that may have introduced this bug:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date: Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M config
```

When you're finished, you should run `git bisect reset` to reset your HEAD to where you were before you started, or you'll end up in a weird state:

```
$ git bisect reset
```

This is a powerful tool that can help you check hundreds of commits for an introduced bug in minutes. In fact, if you have a script that will exit 0 if the project is good or non-0 if the project is bad, you can fully automate `git bisect`. First, you again tell it the scope of the bisect by providing the known bad and good commits. You can do this by listing them with the `bisect start` command if you want, listing the known bad commit first and the known good commit second:

```
$ git bisect start HEAD v1.0  
$ git bisect run test-error.sh
```

Doing so automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. You can also run something like `make` or `make tests` or whatever you have that runs automated tests for you.

Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Here's an example. Suppose you're developing a web site and creating Atom feeds. Instead of writing your own Atom-generating code, you decide to use a library. You're likely to have to either include this code from a shared library like a CPAN install or Ruby gem, or copy the source code into your own project tree. The issue with including the library is that it's difficult to customize the library in any way and often more difficult to deploy it, because you need to make sure every client has that library available. The issue with vendoring the code into your own project is that any custom changes you make are difficult to merge when upstream changes become available.

Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

Starting with Submodules

We'll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let's start by adding an existing Git repository as a submodule of the repository that we're working on. To add a new submodule you use the `git submodule add` command with the absolute or relative URL of the project you would like to start tracking. In this example, we'll add a library called "DbConnector".

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

By default, submodules will add the subproject into a directory named the same as the repository, in this case “DbConnector”. You can add a different path at the end of the command if you want it to go elsewhere.

If you run `git status` at this point, you’ll notice a few things.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   new file:   .gitmodules
   new file:   DbConnector
```

First you should notice the new `.gitmodules` file. This is a configuration file that stores the mapping between the project’s URL and the local subdirectory you’ve pulled it into:

```
[submodule "DbConnector"]
  path = DbConnector
  url = https://github.com/chaconinc/DbConnector
```

If you have multiple submodules, you’ll have multiple entries in this file. It’s important to note that this file is version-controlled with your other files, like your `.gitignore` file. It’s pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.

NOTE

Since the URL in the `.gitmodules` file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would to pull from, use the one that others have access to. You can overwrite this value locally with `git config submodule.DbConnector.url PRIVATE_URL` for your own use. When applicable, a relative URL can be helpful.

The other listing in the `git status` output is the project folder entry. If you run `git diff` on that, you see something interesting:


```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Although `DbConnector` is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

If you want a little nicer diff output, you can pass the `--submodule` option to `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+    path = DbConnector
+    url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

When you commit, you see something like this:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 DbConnector
```

Notice the `160000` mode for the `DbConnector` entry. That is a special mode in Git that basically means you're recording a commit as a directory entry rather than a subdirectory or a file.

Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

```

$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x  9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x  7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--  1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x  2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--  1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x  3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$

```

The `DbConnector` directory is there, but empty. You must run two commands: `git submodule init` to initialize your local configuration file, and `git submodule update` to fetch all the data from that project and check out the appropriate commit listed in your superproject:

```

$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'

```

Now your `DbConnector` subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass `--recursive` to the `git clone` command, it will automatically initialize and update each submodule in the repository.

```
$ git clone --recursive https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Working on a Project with Submodules

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

Pulling in Upstream Changes

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let's walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run `git fetch` and `git merge` the upstream branch to update the local code.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
   c3f01dc..d0354fc master    -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
 scripts/connect.sh | 1 +
 src/db.c           | 1 +
 2 files changed, 2 insertions(+)
```

Now if you go back into the main project and run `git diff --submodule` you can see that the submodule was updated and get a list of commits that were added to it. If you don't want to type `--submodule` every time you run `git diff`, you can set it as the default format by setting the `diff.submodule` config value to "log".

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run `git submodule update --remote`, Git will go into your submodules and fetch and update for you.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 3f19983..d0354fc  master    -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

This command will by default assume that you want to update the checkout to the `master` branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the DbConnector submodule track that repository’s “stable” branch, you can set it in either your `.gitmodules` file (so everyone else also tracks it), or just in your local `.git/config` file. Let’s set it in the `.gitmodules` file:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 27cf5d3..c87d55d  stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

If you leave off the `-f .gitmodules` it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run `git status` at this point, Git will show us that we have “new commits” on the submodule.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

   modified:   .gitmodules
   modified:   DbConnector (new commits)

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

If you set the configuration setting `status.submodulesummary`, Git will also show you a short summary of changes to your submodules:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

   modified:   .gitmodules
   modified:   DbConnector (new commits)

Submodules changed but not updated:
* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

At this point if you run `git diff` we can see both that we have modified our `.gitmodules` file and also that there are a number of commits that we've pulled down and are ready to commit to our submodule project.

```

$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+   branch = stable
Submodule DbConnector c3f01dc..c87d55d:
 > catch non-null terminated lines
 > more robust error handling
 > more efficient db routine
 > better connection routine

```

This is pretty cool as we can actually see the log of commits that we're about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run `git log -p`.

```

$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+   branch = stable
Submodule DbConnector c3f01dc..c87d55d:
 > catch non-null terminated lines
 > more robust error handling
 > more efficient db routine
 > better connection routine

```

Git will by default try to update **all** of your submodules when you run `git submodule update --remote` so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

Working on a Submodule

It's quite likely that if you're using submodules, you're doing so because you really want to work on the code in the submodule at the same time as you're working on the code in the main project (or across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let's go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we've run the `git submodule update` command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what's called a "detached HEAD" state. This means that there is no local working branch (like "master", for example) tracking changes. So any changes you make aren't being tracked well.

In order to set up your submodule to be easier to go in and hack on, you need do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then `git submodule update --remote` pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let's go into our submodule directory and check out a branch.

```
$ git checkout stable
Switched to branch 'stable'
```

Let's try it with the "merge" option. To specify it manually, we can just add the `--merge` option to our `update` call. Here we'll see that there was a change on the server for this submodule and it gets merged in.

```
$ git submodule update --remote --merge
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
   c87d55d..92c7337  stable    -> origin/stable
Updating c87d55d..92c7337
Fast-forward
 src/main.c | 1 +
 1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

If we go into the DbConnector directory, we have the new changes already merged into

our local `stable` branch. Now let's see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

```
$ cd DbConnector/  
$ vim src/db.c  
$ git commit -am 'unicode support'  
[stable f906e16] unicode support  
1 file changed, 1 insertion(+)
```

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

```
$ git submodule update --remote --rebase  
First, rewinding head to replay your work on top of it...  
Applying: unicode support  
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If you forget the `--rebase` or `--merge`, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

```
$ git submodule update --remote  
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If this happens, don't worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase `origin/stable` (or whatever remote branch you want) manually.

If you haven't committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

```
$ git submodule update --remote  
remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 4 (delta 0), reused 4 (delta 0)  
Unpacking objects: 100% (4/4), done.  
From https://github.com/chaconinc/DbConnector  
5d60ef9..c75e92a stable -> origin/stable  
error: Your local changes to the following files would be overwritten by checkout:  
scripts/setup.sh  
Please, commit your changes or stash them before you can switch branches.  
Aborting  
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path  
'DbConnector'
```


If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

You can go into the submodule directory and fix the conflict just as you normally would.

Publishing Submodule Changes

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren't available to anyone else yet as we haven't pushed them yet.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
 > Merge from origin/stable
 > updated setup script
 > unicode support
 > remove unnecessary method
 > add new option for conn pooling
```

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn't happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The `git push` command takes the `--recurse-submodules` argument which can be set to either "check" or "on-demand". The "check" option will make `push` simply fail if any of the committed submodule changes haven't been pushed.

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that can
not be found on any remote:
  DbConnector
```

Please try

```
git push --recurse-submodules=on-demand
```

or cd to the path and use

```
git push
```

to push them to a remote.

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they're externally available and then try this push again.

The other option is to use the “on-demand” value, which will try to do this for you.

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'DbConnector'
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To https://github.com/chaconinc/DbConnector
   c75e92a..82d2ad3  stable -> stable
Counting objects: 2, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To https://github.com/chaconinc/MainProject
   3d6d338..9a377d1  master -> master
```

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail.

Merging Submodule Changes

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
 9a377d1..eb974f8 master    -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

So basically what has happened here is that Git has figured out that the two branches record points in the submodule's history that are divergent and need to be merged. It explains it as "merge following commits not found", which is confusing but we'll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn't really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it's simple to figure out. If you run `git diff` you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

So, in this case, `eb41d76` is the commit in our submodule that **we** had and `c771610` is the commit that upstream had. If we go into our submodule directory, it should already be on `eb41d76` as the merge would not have touched it. If for whatever reason it's not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you'll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch based on that second SHA-1 from `git diff` and manually merge.

```

$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.

```

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

```

$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
-Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes

```

- ① First we resolve the conflict
- ② Then we go back to the main project directory
- ③ We can check the SHA-1s again
- ④ Resolve the conflicted submodule entry
- ⑤ Commit our merge

It can be a bit confusing, but it's really not very hard.

Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone

merged branches containing these two commits, so maybe you'll want that one.

This is why the error message from before was “merge following commits not found”, because it could not do **this**. It's confusing because who would expect it to **try** to do this?

If it does find a single acceptable merge commit, you'll see something like this:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
  "DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

What it's suggesting that you do is to update the index like you had run `git add`, which clears the conflict, then commit. You probably shouldn't do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you're done.

Submodule Tips

There are a few things you can do to make working with submodules a little easier.

Submodule Foreach

There is a `foreach` submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same

project.

For example, let's say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

```
$ git submodule foreach 'git stash'
Entering 'CryptoLibrary'
No local changes to save
Entering 'DbConnector'
Saved working directory and index state WIP on stable: 82d2ad3 Merge from
origin/stable
HEAD is now at 82d2ad3 Merge from origin/stable
```

Then we can create a new branch and switch to it in all our submodules.

```
$ git submodule foreach 'git checkout -b featureA'
Entering 'CryptoLibrary'
Switched to a new branch 'featureA'
Entering 'DbConnector'
Switched to a new branch 'featureA'
```

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

```

$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

    commit_pager_choice();

+   url = url_decode(url_orig);
+
    /* build alias_argv */
    alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
    alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
    return url_decode_internal(&url, len, NULL, &out, 0);
}

+char *url_decode(const char *url)
+{
+   return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
    struct strbuf out = STRBUF_INIT;

```

Here we can see that we're defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

Useful Aliases

You may want to set up some aliases for some of these commands as they can be quite long and you can't set configuration options for most of them to make them defaults. We covered setting up Git aliases in [Git aliasi](#), but here is an example of what you may want to set up if you plan on working with submodules in Git a lot.

```

$ git config alias.sdiff '!git diff && git submodule foreach 'git diff''
$ git config alias.push 'push --recurse-submodules=on-demand'
$ git config alias.update 'submodule update --remote --merge'

```

This way you can simply run `git update` when you want to update your submodules, or `git push` to push with submodule dependency checking.

Issues with Submodules

Using submodules isn't without hiccups, however.

For instance switching branches with submodules in them can also be tricky. If you create a new branch, add a submodule there, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory:

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    CryptoLibrary/

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

Removing the directory isn't difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run `submodule update --init` to repopulate it.


```
$ git clean -fdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/

$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile    includes    scripts    src
```

Again, not really very difficult, but it can be a little confusing.

The other main caveat that many people run into involves switching from subdirectories to submodules. If you've been tracking files in your project and you want to move them out into a submodule, you must be careful or Git will get angry at you. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. If you delete the subdirectory and then run `submodule add`, Git yells at you:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

You have to unstage the `CryptoLibrary` directory first. Then you can add the submodule:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Now suppose you did that in a branch. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
  CryptoLibrary/Makefile
  CryptoLibrary/includes/crypto.h
  ...
Please move or remove them before you can switch branches.
Aborting
```

You can force it to switch with `checkout -f`, but be careful that you don't have unsaved changes in there as they could be overwritten with that command.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Then, when you switch back, you get an empty `CryptoLibrary` directory for some reason and `git submodule update` may not fix it either. You may need to go into your submodule directory and run a `git checkout .` to get all your files back. You could run this in a `submodule foreach` script to run it for multiple submodules.

It's important to note that submodules these days keep all their Git data in the top project's `.git` directory, so unlike much older versions of Git, destroying a submodule directory won't lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.

Bundling

Though we've covered the common ways to transfer Git data over a network (HTTP, SSH, etc), there is actually one more way to do so that is not commonly used but can actually be quite useful.

Git is capable of "bundling" its data into a single file. This can be useful in various scenarios. Maybe your network is down and you want to send changes to your co-workers. Perhaps you're working somewhere offsite and don't have access to the local network for security reasons. Maybe your wireless/ethernet card just broke. Maybe you don't have access to a shared server for the moment, you want to email someone updates and you don't want to transfer 40 commits via `format-patch`.

This is where the `git bundle` command can be helpful. The `bundle` command will package up everything that would normally be pushed over the wire with a `git push` command into a binary file that you can email to someone or put on a flash drive, then unbundle into another repository.

Let's see a simple example. Let's say you have a repository with two commits:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

If you want to send that repository to someone and you don't have access to a repository to push to, or simply don't want to set one up, you can bundle it with `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Now you have a file named `repo.bundle` that has all the data needed to re-create the repository's `master` branch. With the `bundle` command you need to list out every reference or specific range of commits that you want to be included. If you intend for this to be cloned somewhere else, you should add `HEAD` as a reference as well as we've done here.

You can email this `repo.bundle` file to someone else, or put it on a USB drive and walk it over.

On the other side, say you are sent this `repo.bundle` file and want to work on the project. You can clone from the binary file into a directory, much like you would from a URL.

```
$ git clone repo.bundle repo
Initialized empty Git repository in /private/tmp/bundle/repo/.git/
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

If you don't include `HEAD` in the references, you have to also specify `-b master` or whatever branch is included because otherwise it won't know what branch to check out.

Now let's say you do three commits on it and want to send the new commits back via a bundle on a USB stick or email.

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

First we need to determine the range of commits we want to include in the bundle. Unlike the network protocols which figure out the minimum set of data to transfer over the network for us, we'll have to figure this out manually. Now, you could just do the same thing and bundle the entire repository, which will work, but it's better to just bundle up the difference - just the three commits we just made locally.

In order to do that, you'll have to calculate the difference. As we described in [Commit Ranges](#), you can specify a range of commits in a number of ways. To get the three commits that we have in our master branch that weren't in the branch we originally cloned, we can use something like `origin/master..master` or `master ^origin/master`. You can test that with the `log` command.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

So now that we have the list of commits we want to include in the bundle, let's bundle them up. We do that with the `git bundle create` command, giving it a filename we want our bundle to be and the range of commits we want to go into it.

```
$ git bundle create commits.bundle master ^9a466c5
Counting objects: 11, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 775 bytes, done.
Total 9 (delta 0), reused 0 (delta 0)
```

Now we have a `commits.bundle` file in our directory. If we take that and send it to our partner, she can then import it into the original repository, even if more work has been done there in the meantime.

When she gets the bundle, she can inspect it to see what it contains before she imports it into her repository. The first command is the `bundle verify` command that will make sure the file is actually a valid Git bundle and that you have all the necessary ancestors to reconstitute it properly.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

If the bundler had created a bundle of just the last two commits they had done, rather than all three, the original repository would not be able to import it, since it is missing requisite history. The `verify` command would have looked like this instead:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

However, our first bundle is valid, so we can fetch in commits from it. If you want to see what branches are in the bundle that can be imported, there is also a command to just list the heads:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

The `verify` sub-command will tell you the heads as well. The point is to see what can be pulled in, so you can use the `fetch` or `pull` commands to import commits from this bundle. Here we'll fetch the `master` branch of the bundle to a branch named `other-master` in our repository:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
* [new branch]      master      -> other-master
```

Now we can see that we have the imported commits on the `other-master` branch as well as any commits we've done in the meantime in our own `master` branch.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

So, `git bundle` can be really useful for sharing or doing network-type operations when

you don't have the proper network or shared repository to do so.

Replace

Objekti v Git-u niso spremenljivi, vendar ponuja zanimiv način pretvarjanja zamenjave objektov v njegovi podatkovni bazi z drugimi objekti.

Ukaz `replace` vam omogoča določiti objekt v Git-u in povedati "vsakič ko to vidite, se pretvarjaj, da gre za drugo stvar". To je najpogostejše uporabno za zamenjavo enega pošiljanja v vaši zgodovini z drugim.

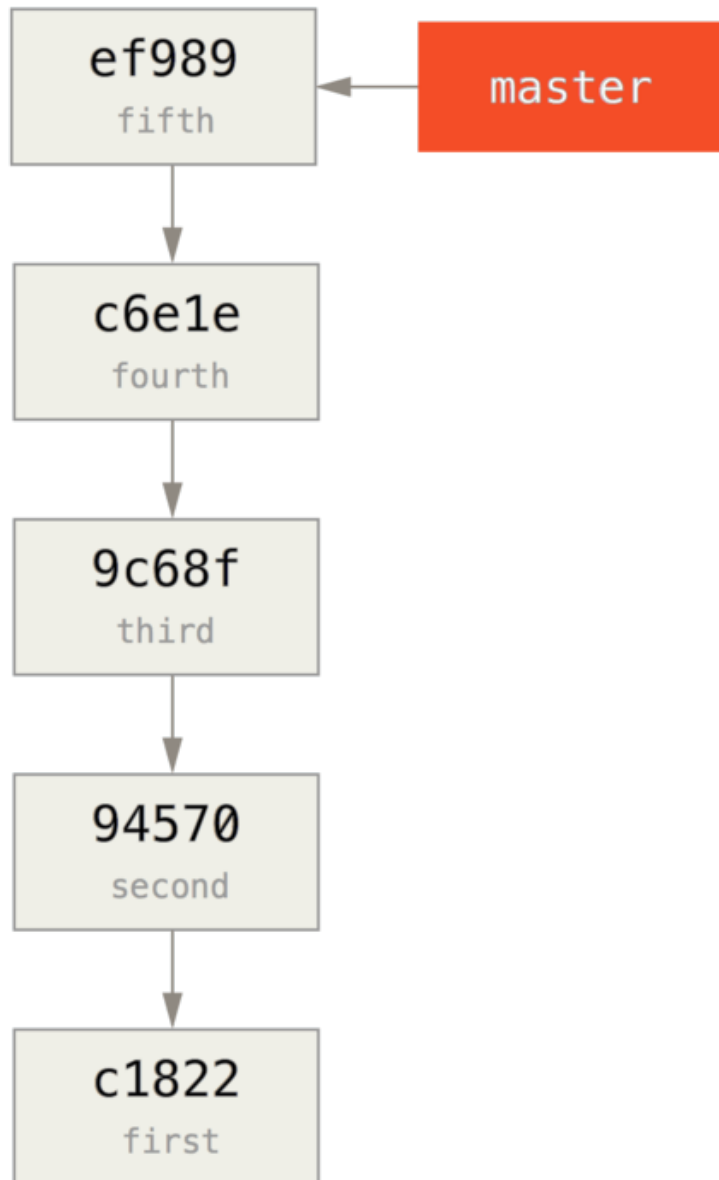
Na primer, predpostavimo, da imate veliko zgodovino kode in želite cepiti vaš repozitorij v eno hitro zgodovino za nove razvijalce in enega z veliko daljšo in večjo zgodovino za ljudi, ki jih zanima kopanje podatkov. Transplantirate lahko eno zgodovino v drugo z zamenjavo (`replace`) prejšnjih pošiljanj v novi vrstici z najnovejšim pošiljanjem na stari. To je lepo, ker pomeni, da vam ni potrebno dejansko prepisati vsakega pošiljanja v novi zgodovini, kot bi običajno morali, da ju združite skupaj (ker starševstvo vpliva na SHA-1).

Poskusimo to. Naredimo obstoječi repozitorij, ga cepimo v dva repozitorija, enega zadnjega in enega zgodovinskega in nato bomo videli, kako lahko ponovno kombiniramo oba brez sprememb zadnjih SHA-1 vrednosti repozitorija preko `replace`.

Uporabili bomo enostaven repozitorij s petimi enostavnimi pošiljanji:

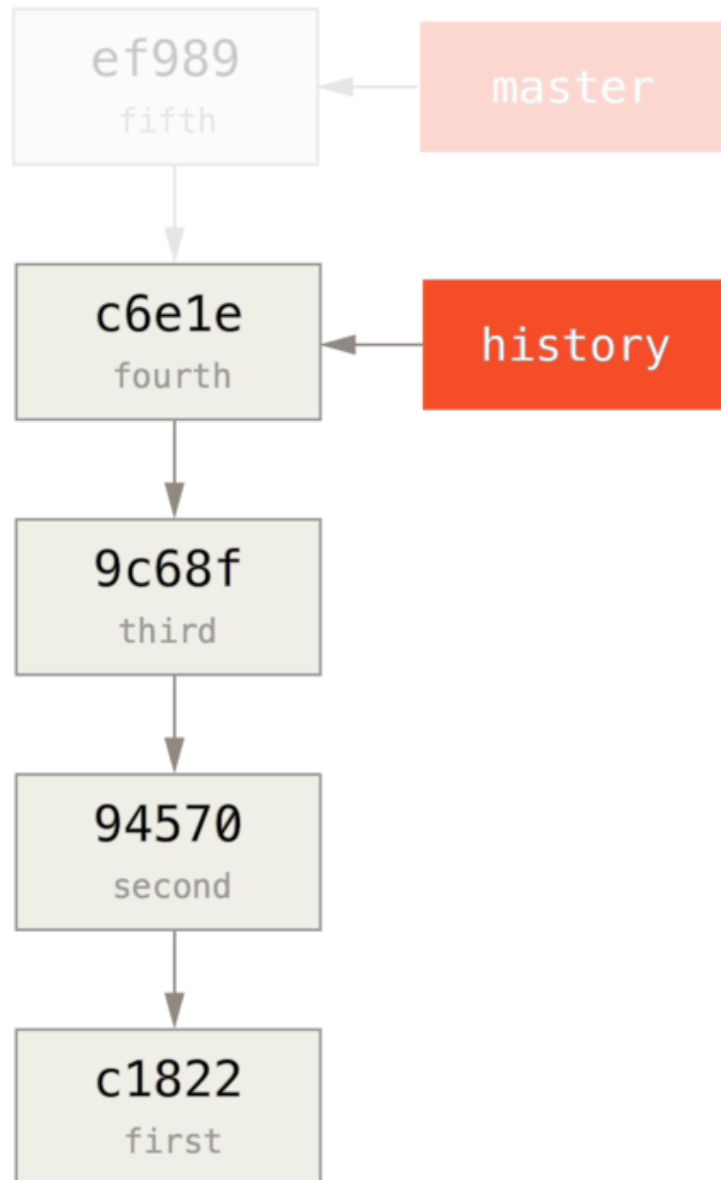
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Želimo to prelomiti v dve vrstici zgodovine. Ena vrstica gre iz prvega pošiljanja v četrto pošiljanje - ta bo zgodovinski. Druga vrstica bosta samo pošiljanja 4 in 5.



Torej ustvarjanje zgodovinske zgodovine je enostavno, lahko samo damo vejo v zgodovino in nato potisnemo to vejo v glavno vejo novega oddaljenega repozitorija.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



Sedaj lahko potisnemo novo vejo **history** v vejo **master** našega novega repozitorija:

```
$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]    history -> master
```

Dobro, sedaj je naša zgodovina objavljena. Sedaj je težji del krajšanje naše zadnje

zgodovine navzdol, da je manjša. Narediti moramo prekriavnje, da lahko zamenjamo pošiljanje v enem ekvivalentnem pošiljanju v drugem, torej bomo skrajšali to na samo pošiljanje 4 in 5 (torej se pošiljanje 4 prekrije).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

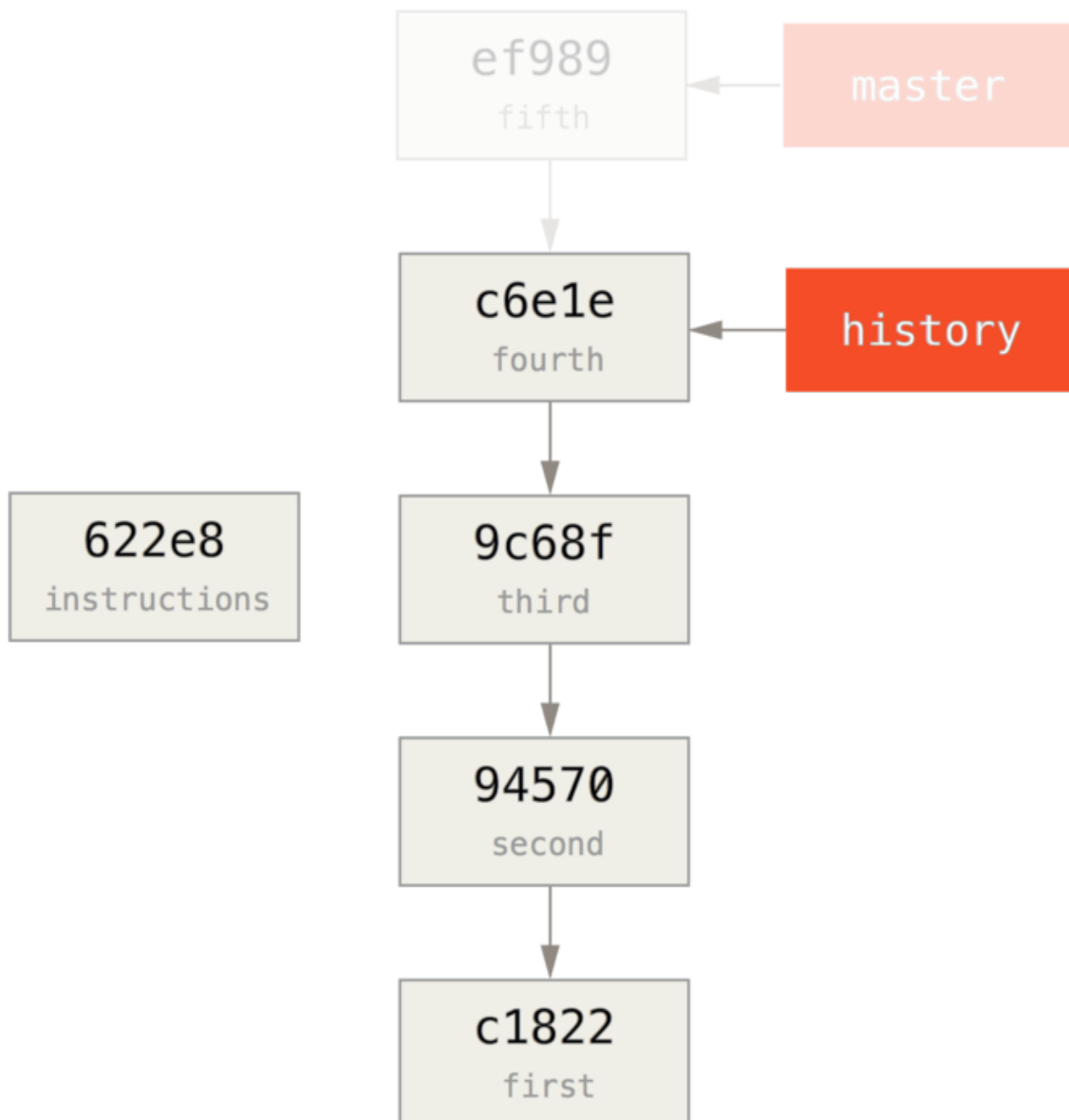
V tem primeru je uporabno ustvariti osnovno pošiljanje, da ima navodila, kako razširiti zgodovino, da drugi razvijalci vejo, kaj narediti, če pridejo do prvega pošiljanja v prekrito zgodovini in potrebujejo več. Torej, kar bomo naredili je ustvarjanje začetnega objekta pošiljanja kot našo osnovno točko z navodili, nato naredili rebase ostalih pošiljanj (4 in 5) na vrhu le tega.

Da to naredimo, potrebujemo izbrati točko za cepitev, ki bo za nas tretje pošiljanje, ki je `9c68fdc` v besedi SHA. Torej naše osnovno pošiljanje bo osnovano na tem drevesu. Lahko ustvarimo naše osnovno pošiljanje z uporabo ukaza `commit-tree`, ki samo vzame drevo in nam vrne popolnoma nov objekt pošiljanja SHA-1 brez starša.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfea10cf
```

NOTE

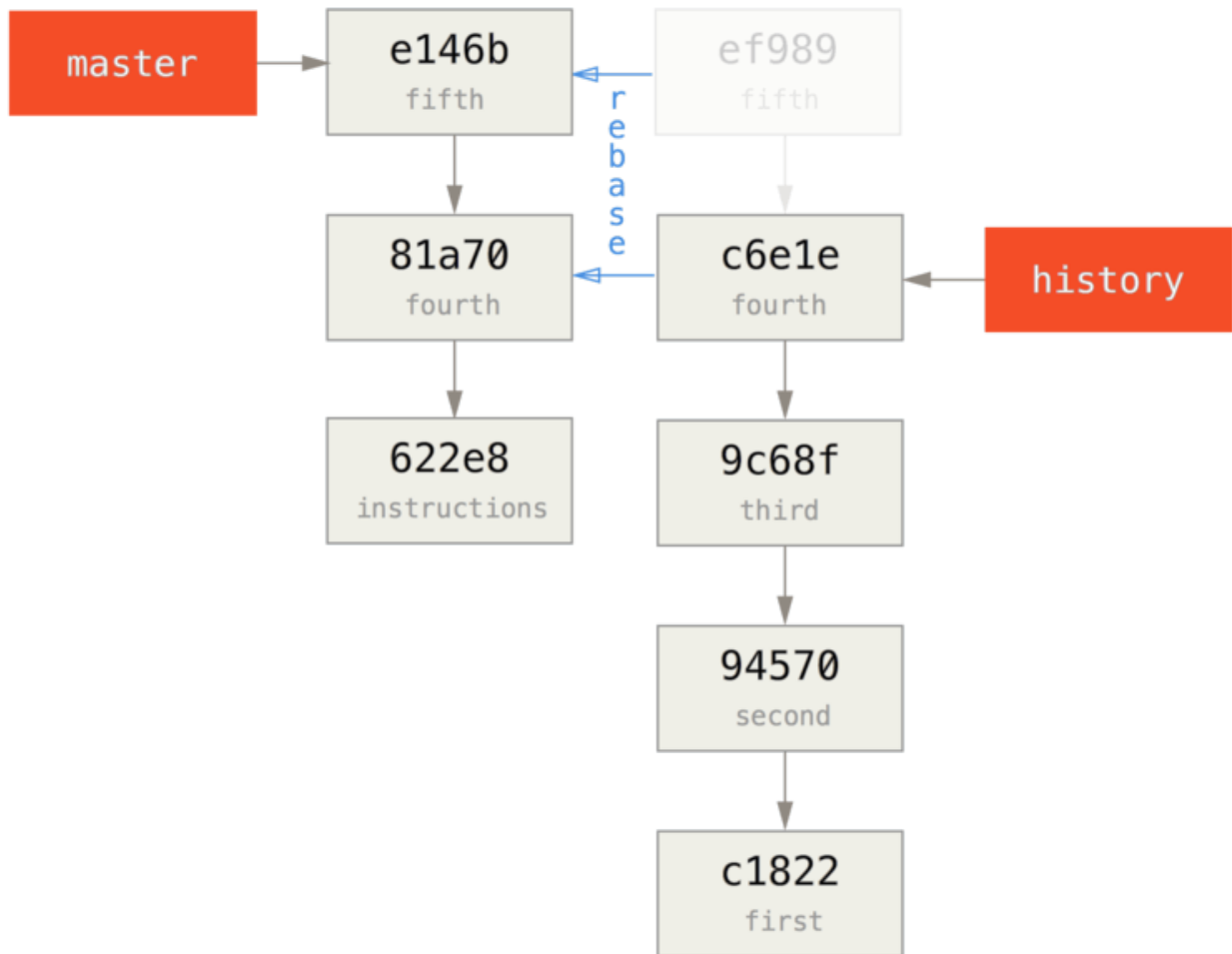
Ukaz `commit-tree` je eden izmed skupkov ukazov, ki so pogosto sklicevani kot ukazi napeljave (`plumbing`). Ti ukazi v splošnem niso mišljeni za direktno uporabo, vendar namesto tega so uporabljeni za **druge** Git ukaze, da naredijo manjše naloge. Občasno, ko delamo čudnejše stvari kot to, nam omogočajo narediti resnično nižje nivojske stvari, vendar niso mišljeni za dnevno uporabo. Več lahko preberete o ukazih napeljave v [Napeljava in porcelan](#)



Torej sedaj ko imamo osnovno pošiljanje, lahko naredimo t.i. rebase na naši preostali zgodovini na vrhu tega, z `git rebase --onto`. Argument `--onto` bo SHA-1, ki smo ga ravno dobili iz `commit-tree` in točke rebase-a bo tretje pošiljanje (starš prvega pošiljanja, ki ga želimo obdržati `9c68fdc`):

```

$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
  
```



Torej sedaj smo prepisali našo zadnjo zgodovino na vrhu ovrženeega osnovnega pošiljanja, ki ima sedaj navodila v njem, kako rekonstruirati celotno zgodovino, če to želimo. Lahko pošljemo to novo zgodovino v nov projekt in sedaj, ko ljudje klonirajo ta repozitorij, bodo videli samo zadnji dve pošiljanji in nato osnovno pošiljanje z navodili.

Sedaj preklopimo vloge nekomu, ki prvič klonira projekt in želi celotno zgodovino. Da dobimo podatke zgodovine po kloniranju tega skrajšanega repozitorija, bi nekdo moral dodati drugo daljavo za zgodovinski repozitorij in jo ujeti:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
* [new branch]      master      -> project-history/master
```

Sedaj bi sodelavec moral imeti svoja zadnja pošiljanja v veji `master` in zgodovinsko pošiljanje v veji `project-history/master`.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

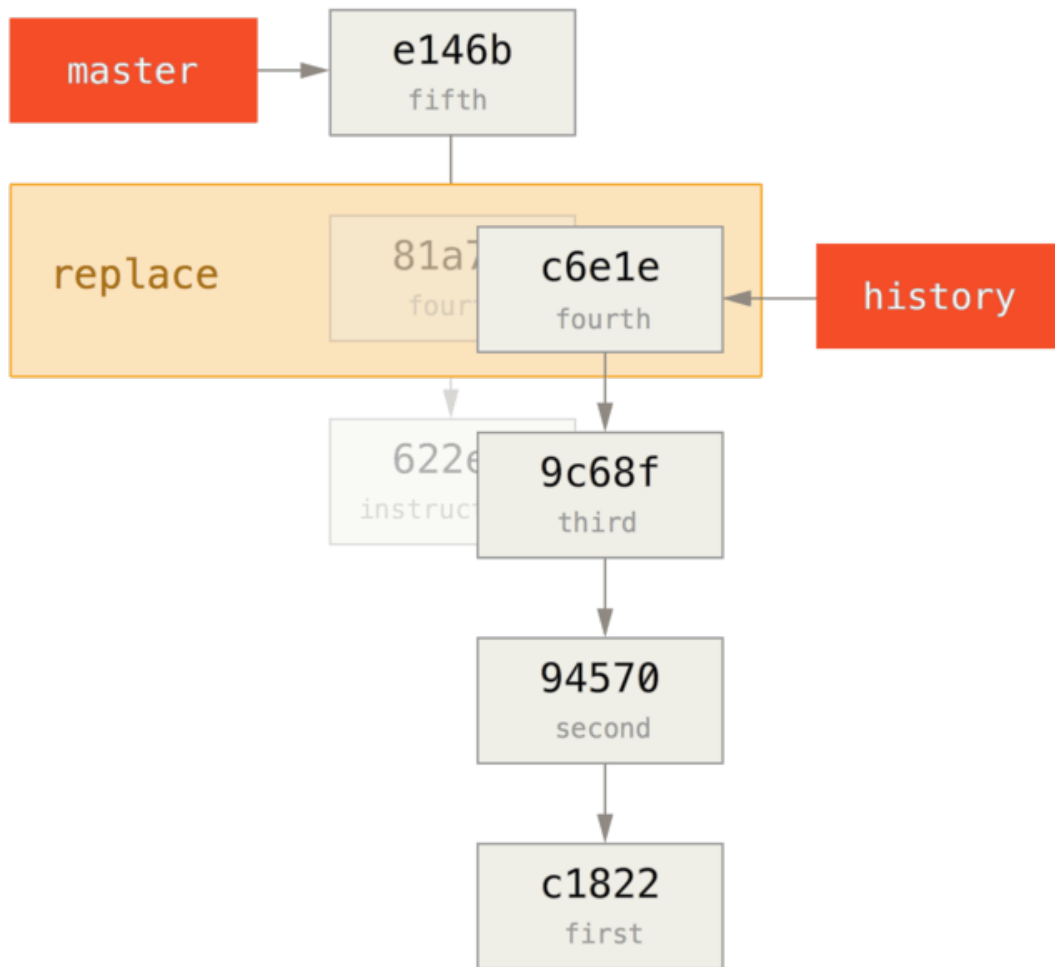
Da jih kombiniramo, lahko enostavno pokličete `git replace` s pošiljanjem, ki ga želite zamenjati in nato poslati, kar želite z njim zamenjati. Torej želimo zamenjati četrto pošiljanje v veji `master` s četrtim pošiljanjem v veji `project-history/master`:

```
$ git replace 81a708d c6e1e95
```

Sedaj, če pogledate zgodovino veje `master`, izgleda nekako takole:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Kul, kajne? Brez, da moramo spremeniti vse SHA-1 zgornjega toka, smo lahko zamenjali eno pošiljanje v naši zgodovini s popolnoma drugim pošiljanjem in vsa običajna orodja (`bisect`, `blame` itd) bodo delovala kakor pričakujemo.



Zanimivo še vedno kaže `81a708d` kot SHA-1, čeprav dejansko uporablja `c6e1e95`, podatke pošiljanja, s katerimi smo jih zamenjali. Tudi če poženete ukaz, kot je `cat-file`, vam bo pokazal zamenjane podatke:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Zapomniti si je dobro, da je bil dejanski starš `81a708d` ograda pošiljanja (`622e88e`) in ne `9c68fdce` kakor je tu zabeleženo.

Druga zanimiva stvar je, da so ti podatki obdržani v naši referenci:

```
$ git for-each-ref
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master
c6e1e95051d41771a649f3145423f8809d1a74d4 commit
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

To pomeni, da je enostavno deliti našo zamenjavo z drugimi, ker lahko potisnemo to na naš strežnik in drugi ljudje lahko to enostavno prenesejo. To ni v pomoč v zgodovini scenarija presadka, ko smo šli skozi tu (ker bi vsak prenesel obe zgodovini tako ali tako, torej zakaj ju ločiti?), vendar je lahko uporabno v drugih okoliščinah.

Credential Storage

If you use the SSH transport for connecting to remotes, it's possible for you to have a key without a passphrase, which allows you to securely transfer data without typing in your username and password. However, this isn't possible with the HTTP protocols – every connection needs a username and password. This gets even harder for systems with two-factor authentication, where the token you use for a password is randomly generated and unpronounceable.

Fortunately, Git has a credentials system that can help with this. Git has a few options provided in the box:

- The default is not to cache at all. Every connection will prompt you for your username and password.
- The “cache” mode keeps credentials in memory for a certain period of time. None of the passwords are ever stored on disk, and they are purged from the cache after 15 minutes.
- The “store” mode saves the credentials to a plain-text file on disk, and they never expire. This means that until you change your password for the Git host, you won't ever have to type in your credentials again. The downside of this approach is that your passwords are stored in cleartext in a plain file in your home directory.
- If you're using a Mac, Git comes with an “osxkeychain” mode, which caches credentials in the secure keychain that's attached to your system account. This method stores the credentials on disk, and they never expire, but they're encrypted with the same system that stores HTTPS certificates and Safari auto-fills.
- If you're using Windows, you can install a helper called “winstore.” This is similar to the “osxkeychain” helper described above, but uses the Windows Credential Store to control sensitive information. It can be found at <https://gitcredentialstore.codeplex.com>.

You can choose one of these methods by setting a Git configuration value:

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

Some of these helpers have options. The “store” helper can take a `--file <path>` argument, which customizes where the plaintext file is saved (the default is `~/.git-credentials`). The “cache” helper accepts the `--timeout <seconds>` option, which changes the amount of time its daemon is kept running (the default is “900”, or 15 minutes). Here’s an example of how you’d configure the “store” helper with a custom file name:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git even allows you to configure several helpers. When looking for credentials for a particular host, Git will query them in order, and stop after the first answer is provided. When saving credentials, Git will send the username and password to **all** of the helpers in the list, and they can choose what to do with them. Here’s what a `.gitconfig` would look like if you had a credentials file on a thumb drive, but wanted to use the in-memory cache to save some typing if the drive isn’t plugged in:

```
[credential]
  helper = store --file /mnt/thumbdrive/.git-credentials
  helper = cache --timeout 30000
```

Under the Hood

How does this all work? Git’s root command for the credential-helper system is `git credential`, which takes a command as an argument, and then more input through stdin.

This might be easier to understand with an example. Let’s say that a credential helper has been configured, and the helper has stored credentials for `mygithost`. Here’s a session that uses the “fill” command, which is invoked when Git is trying to find credentials for a host:

```

$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

```

```

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7

```

- ① This is the command line that initiates the interaction.
- ② Git-credential is then waiting for input on stdin. We provide it with the things we know: the protocol and hostname.
- ③ A blank line indicates that the input is complete, and the credential system should answer with what it knows.
- ④ Git-credential then takes over, and writes to stdout with the bits of information it found.
- ⑤ If credentials are not found, Git asks the user for the username and password, and provides them back to the invoking stdout (here they're attached to the same console).

The credential system is actually invoking a program that's separate from Git itself; which one and how depends on the `credential.helper` configuration value. There are several forms it can take:

| Configuration Value | Behavior |
|--|---|
| <code>foo</code> | Runs <code>git-credential-foo</code> |
| <code>foo -a --opt=bcd</code> | Runs <code>git-credential-foo -a --opt=bcd</code> |
| <code>/absolute/path/foo -xyz</code> | Runs <code>/absolute/path/foo -xyz</code> |
| <code>!f() { echo "password=s3cre7"; }; f</code> | Code after <code>!</code> evaluated in shell |

So the helpers described above are actually named `git-credential-cache`, `git-credential-store`, and so on, and we can configure them to take command-line arguments. The general form for this is “`git-credential-foo [args] <action>`.” The stdin/stdout protocol is the same as `git-credential`, but they use a slightly different set of actions:

- `get` is a request for a username/password pair.

- `store` is a request to save a set of credentials in this helper’s memory.
- `erase` purge the credentials for the given properties from this helper’s memory.

For the `store` and `erase` actions, no response is required (Git ignores it anyway). For the `get` action, however, Git is very interested in what the helper has to say. If the helper doesn’t know anything useful, it can simply exit with no output, but if it does know, it should augment the provided information with the information it has stored. The output is treated like a series of assignment statements; anything provided will replace what Git already knows.

Here’s the same example from above, but skipping `git-credential` and going straight for `git-credential-store`:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

- ① Here we tell `git-credential-store` to save some credentials: the username “bob” and the password “s3cre7” are to be used when `https://mygithost` is accessed.
- ② Now we’ll retrieve those credentials. We provide the parts of the connection we already know (`https://mygithost`), and an empty line.
- ③ `git-credential-store` replies with the username and password we stored above.

Here’s what the `~/git.store` file looks like:

```
https://bob:s3cre7@mygithost
```

It’s just a series of lines, each of which contains a credential-decorated URL. The `osxkeychain` and `winstore` helpers use the native format of their backing stores, while `cache` uses its own in-memory format (which no other process can read).

A Custom Credential Cache

Given that `git-credential-store` and friends are separate programs from Git, it’s not much of a leap to realize that *any* program can be a Git credential helper. The helpers provided by Git cover many common use cases, but not all. For example, let’s say your team has some credentials that are shared with the entire team, perhaps for deployment. These are stored in a shared directory, but you don’t want to copy them to your own credential store, because they change often. None of the existing

helpers cover this case; let's see what it would take to write our own. There are several key features this program needs to have:

1. The only action we need to pay attention to is `get`; `store` and `erase` are write operations, so we'll just exit cleanly when they're received.
2. The file format of the shared-credential file is the same as that used by `git-credential-store`.
3. The location of that file is fairly standard, but we should allow the user to pass a custom path just in case.

Once again, we'll write this extension in Ruby, but any language will work so long as Git can execute the finished product. Here's the full source code of our new credential helper:

```
#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
  opts.banner = 'USAGE: git-credential-read-only [options] <action>'
  opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
    path = File.expand_path argpath
  end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
  break if line.strip == ''
  k,v = line.strip.split '=', 2
  known[k] = v
end

File.readlines(path).each do |fileline| ④
  prot,user,pass,host = fileline.scan(/^(.*?):\\\/(.*?):(.*?)@(.*?)$/).first
  if prot == known['protocol'] and host == known['host'] then
    puts "protocol=#{prot}"
    puts "host=#{host}"
    puts "username=#{user}"
    puts "password=#{pass}"
    exit(0)
  end
end
```

- ① Here we parse the command-line options, allowing the user to specify the input file. The default is `~/.git-credentials`.

- ② This program only responds if the action is `get` and the backing-store file exists.
- ③ This loop reads from stdin until the first blank line is reached. The inputs are stored in the `known` hash for later reference.
- ④ This loop reads the contents of the storage file, looking for matches. If the protocol and host from `known` match this line, the program prints the results to stdout and exits.

We'll save our helper as `git-credential-read-only`, put it somewhere in our `PATH` and mark it executable. Here's what an interactive session looks like:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Since its name starts with “git-”, we can use the simple syntax for the configuration value:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

As you can see, extending this system is pretty straightforward, and can solve some common problems for you and your team.

Povzetek

Videli ste število naprednih orodij, ki vam omogočajo manipulacijo vaših pošiljanj in dajanja v vmesno fazo bolj točno. Ko opazite težave, bi morali biti sposobni enostavno določiti, katero pošiljanje jih je predstavilo, kdaj in kdo. Če želite uporabiti podprojekte v vašem projektu, ste se naučili, kako te potrebe sprejeti. Na tej točki, bi morali biti zmožni narediti večino stvari v Git-u, ki jih boste potrebovali v ukazni vrstici dan za dnem in se s tem počutite udobno.

Prilagoditev Git-a

Do sedaj smo pokrili osnove, kako Git deluje in kako ga uporabljati in predstavili smo število orodij, ki jih Git ponuja, da vam pomaga z enostavno in učinkovito uporabo. V tem poglavju bomo pogledali, kako lahko naredite, da Git operira na bolj prilagojeni način s predstavitvijo nekaj pomembnih konfiguracijskih nastavitev in sistema kljuk. S temi orodji je enostavno dobiti, da Git dela točno na način, ki ga vi, vaše podjetje ali vaša skupina potrebuje.

Git Configuration

As you briefly saw in [Pričetek](#), you can specify Git configuration settings with the `git config` command. One of the first things you did was set up your name and email address:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Now you'll learn a few of the more interesting options that you can set in this manner to customize your Git usage.

First, a quick review: Git uses a series of configuration files to determine non-default behavior that you may want. The first place Git looks for these values is in an `/etc/gitconfig` file, which contains values for every user on the system and all of their repositories. If you pass the option `--system` to `git config`, it reads and writes from this file specifically.

The next place Git looks is the `~/.gitconfig` (or `~/.config/git/config`) file, which is specific to each user. You can make Git read and write to this file by passing the `--global` option.

Finally, Git looks for configuration values in the configuration file in the Git directory (`.git/config`) of whatever repository you're currently using. These values are specific to that single repository.

Each of these "levels" (system, global, local) overwrites values in the previous level, so values in `.git/config` trump those in `/etc/gitconfig`, for instance.

NOTE Git's configuration files are plain-text, so you can also set these values by manually editing the file and inserting the correct syntax. It's generally easier to run the `git config` command, though.

Basic Client Configuration

The configuration options recognized by Git fall into two categories: client-side and server-side. The majority of the options are client-side – configuring your personal working preferences. Many, *many* configuration options are supported, but a large

fraction of them are only useful in certain edge cases. We'll only be covering the most common and most useful here. If you want to see a list of all the options your version of Git recognizes, you can run

```
$ man git-config
```

This command lists all the available options in quite a bit of detail. You can also find this reference material at <http://git-scm.com/docs/git-config.html>.

core.editor

By default, Git uses whatever you've set as your default text editor (`$VISUAL` or `$EDITOR`) or else falls back to the `vi` editor to create and edit your commit and tag messages. To change that default to something else, you can use the `core.editor` setting:

```
$ git config --global core.editor emacs
```

Now, no matter what is set as your default shell editor, Git will fire up Emacs to edit messages.

commit.template

If you set this to the path of a file on your system, Git will use that file as the default message when you commit. For instance, suppose you create a template file at `~/.gitmessage.txt` that looks like this:

```
subject line

what happened

[ticket: X]
```

To tell Git to use it as the default message that appears in your editor when you run `git commit`, set the `commit.template` configuration value:

```
$ git config --global commit.template ~/.gitmessage.txt
$ git commit
```

Then, your editor will open to something like this for your placeholder commit message when you commit:

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

If your team has a commit-message policy, then putting a template for that policy on your system and configuring Git to use it by default can help increase the chance of that policy being followed regularly.

core.pager

This setting determines which pager is used when Git pages output such as `log` and `diff`. You can set it to `more` or to your favorite pager (by default, it's `less`), or you can turn it off by setting it to a blank string:

```
$ git config --global core.pager ''
```

If you run that, Git will page the entire output of all commands, no matter how long they are.

user.signingkey

If you're making signed annotated tags (as discussed in [Signing Your Work](#)), setting your GPG signing key as a configuration setting makes things easier. Set your key ID like so:

```
$ git config --global user.signingkey <gpg-key-id>
```

Now, you can sign tags without having to specify your key every time with the `git tag` command:

```
$ git tag -s <tag-name>
```

core.excludesfile

You can put patterns in your project's `.gitignore` file to have Git not see them as untracked files or try to stage them when you run `git add` on them, as discussed in [Ignoriranje datotek](#).

But sometimes you want to ignore certain files for all repositories that you work with. If your computer is running Mac OS X, you're probably familiar with `.DS_Store` files. If your preferred editor is Emacs or Vim, you know about files that end with a `~`.

This setting lets you write a kind of global `.gitignore` file. If you create a `~/.gitignore_global` file with these contents:

```
*~
.DS_Store
```

...and you run `git config --global core.excludesfile ~/.gitignore_global`, Git will never again bother you about those files.

help.autocorrect

If you mistype a command, it shows you something like this:

```
$ git chekcout master
git: 'chekcout' is not a git command. See 'git --help'.

Did you mean this?
  checkout
```

Git helpfully tries to figure out what you meant, but it still refuses to do it. If you set `help.autocorrect` to 1, Git will actually run this command for you:

```
$ git chekcout master
WARNING: You called a Git command named 'chekcout', which does not exist.
Continuing under the assumption that you meant 'checkout'
in 0.1 seconds automatically...
```

Note that “0.1 seconds” business. `help.autocorrect` is actually an integer which represents tenths of a second. So if you set it to 50, Git will give you 5 seconds to change your mind before executing the autocorrected command.

Colors in Git

Git fully supports colored terminal output, which greatly aids in visually parsing command output quickly and easily. A number of options can help you set the coloring to your preference.

color.ui

Git automatically colors most of its output, but there's a master switch if you don't like this behavior. To turn off all Git's colored terminal output, do this:

```
$ git config --global color.ui false
```

The default setting is `auto`, which colors output when it's going straight to a terminal, but omits the color-control codes when the output is redirected to a pipe or a file.

You can also set it to `always` to ignore the difference between terminals and pipes. You'll rarely want this; in most scenarios, if you want color codes in your redirected output, you can instead pass a `--color` flag to the Git command to force it to use color codes. The default setting is almost always what you'll want.

color.*

If you want to be more specific about which commands are colored and how, Git provides verb-specific coloring settings. Each of these can be set to `true`, `false`, or `always`:

```
color.branch
color.diff
color.interactive
color.status
```

In addition, each of these has subsettings you can use to set specific colors for parts of the output, if you want to override each color. For example, to set the meta information in your diff output to blue foreground, black background, and bold text, you can run

```
$ git config --global color.diff.meta "blue black bold"
```

You can set the color to any of the following values: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, or `white`. If you want an attribute like `bold` in the previous example, you can choose from `bold`, `dim`, `ul` (underline), `blink`, and `reverse` (swap foreground and background).

External Merge and Diff Tools

Although Git has an internal implementation of diff, which is what we've been showing in this book, you can set up an external tool instead. You can also set up a graphical merge-conflict-resolution tool instead of having to resolve conflicts manually. We'll demonstrate setting up the Perforce Visual Merge Tool (P4Merge) to do your diffs and merge resolutions, because it's a nice graphical tool and it's free.

If you want to try this out, P4Merge works on all major platforms, so you should be

able to do so. We'll use path names in the examples that work on Mac and Linux systems; for Windows, you'll have to change `/usr/local/bin` to an executable path in your environment.

To begin, download P4Merge from [download P4Merge from Perforce](#). Next, you'll set up external wrapper scripts to run your commands. We'll use the Mac path for the executable; in other systems, it will be where your `p4merge` binary is installed. Set up a merge wrapper script named `extMerge` that calls your binary with all the arguments provided:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

The diff wrapper checks to make sure seven arguments are provided and passes two of them to your merge script. By default, Git passes the following arguments to the diff program:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Because you only want the `old-file` and `new-file` arguments, you use the wrapper script to pass the ones you need.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

You also need to make sure these tools are executable:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Now you can set up your config file to use your custom merge resolution and diff tools. This takes a number of custom settings: `merge.tool` to tell Git what strategy to use, `mergetool.<tool>.cmd` to specify how to run the command, `mergetool.<tool>.trustExitCode` to tell Git if the exit code of that program indicates a successful merge resolution or not, and `diff.external` to tell Git what command to run for diffs. So, you can either run four config commands

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
'extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"'
$ git config --global mergetool.extMerge.trustExitCode false
$ git config --global diff.external extDiff
```

or you can edit your `~/.gitconfig` file to add these lines:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

After all this is set, if you run diff commands such as this:

```
$ git diff 32d1776b1^ 32d1776b1
```

Instead of getting the diff output on the command line, Git fires up P4Merge, which looks something like this:

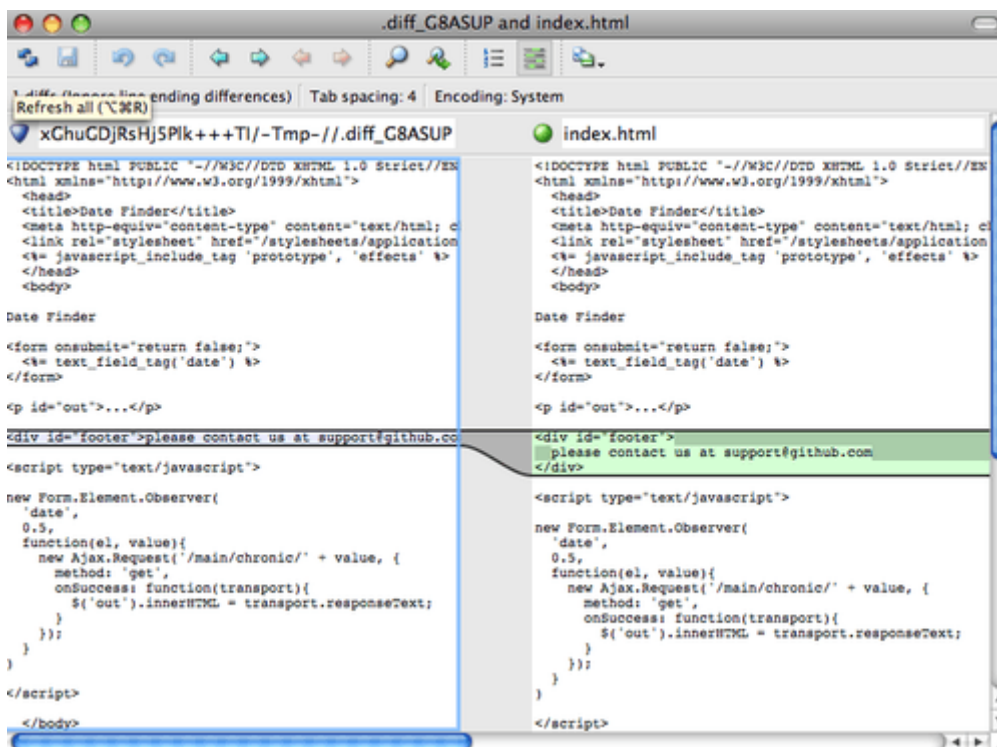


Figure 143. P4Merge.

If you try to merge two branches and subsequently have merge conflicts, you can run the command `git mergetool`; it starts P4Merge to let you resolve the conflicts through that GUI tool.

The nice thing about this wrapper setup is that you can change your diff and merge tools easily. For example, to change your `extDiff` and `extMerge` tools to run the KDiff3 tool instead, all you have to do is edit your `extMerge` file:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Now, Git will use the KDiff3 tool for diff viewing and merge conflict resolution.

Git comes preset to use a number of other merge-resolution tools without your having to set up the cmd configuration. To see a list of the tools it supports, try this:

```
$ git mergetool --tool-help
'git mergetool --tool=<tool>' may be set to one of the following:
  emerge
  gvimdiff
  gvimdiff2
  opendiff
  p4merge
  vimdiff
  vimdiff2
```

The following tools are valid, but not currently available:

```
  araxis
  bc3
  codecompare
  deltawalker
  diffmerge
  diffuse
  ecmmerge
  kdiff3
  meld
  tkdiff
  tortoisemerge
  xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

If you're not interested in using KDiff3 for diff but rather want to use it just for merge resolution, and the kdiff3 command is in your path, then you can run

```
$ git config --global merge.tool kdiff3
```

If you run this instead of setting up the `extMerge` and `extDiff` files, Git will use KDiff3 for merge resolution and the normal Git diff tool for diffs.

Formatting and Whitespace

Formatting and whitespace issues are some of the more frustrating and subtle problems

that many developers encounter when collaborating, especially cross-platform. It's very easy for patches or other collaborated work to introduce subtle whitespace changes because editors silently introduce them, and if your files ever touch a Windows system, their line endings might be replaced. Git has a few configuration options to help with these issues.

`core.autocrlf`

If you're programming on Windows and working with people who are not (or vice-versa), you'll probably run into line-ending issues at some point. This is because Windows uses both a carriage-return character and a linefeed character for newlines in its files, whereas Mac and Linux systems use only the linefeed character. This is a subtle but incredibly annoying fact of cross-platform work; many editors on Windows silently replace existing LF-style line endings with CRLF, or insert both line-ending characters when the user hits the enter key.

Git can handle this by auto-converting CRLF line endings into LF when you add a file to the index, and vice versa when it checks out code onto your filesystem. You can turn on this functionality with the `core.autocrlf` setting. If you're on a Windows machine, set it to `true` – this converts LF endings into CRLF when you check out code:

```
$ git config --global core.autocrlf true
```

If you're on a Linux or Mac system that uses LF line endings, then you don't want Git to automatically convert them when you check out files; however, if a file with CRLF endings accidentally gets introduced, then you may want Git to fix it. You can tell Git to convert CRLF to LF on commit but not the other way around by setting `core.autocrlf` to `input`:

```
$ git config --global core.autocrlf input
```

This setup should leave you with CRLF endings in Windows checkouts, but LF endings on Mac and Linux systems and in the repository.

If you're a Windows programmer doing a Windows-only project, then you can turn off this functionality, recording the carriage returns in the repository by setting the config value to `false`:

```
$ git config --global core.autocrlf false
```

`core.whitespace`

Git comes preset to detect and fix some whitespace issues. It can look for six primary whitespace issues – three are enabled by default and can be turned off, and three are disabled by default but can be activated.

The three that are turned on by default are `blank-at-eol`, which looks for spaces at the

end of a line; `blank-at-eof`, which notices blank lines at the end of a file; and `space-before-tab`, which looks for spaces before tabs at the beginning of a line.

The three that are disabled by default but can be turned on are `indent-with-non-tab`, which looks for lines that begin with spaces instead of tabs (and is controlled by the `tabwidth` option); `tab-in-indent`, which watches for tabs in the indentation portion of a line; and `cr-at-eol`, which tells Git that carriage returns at the end of lines are OK.

You can tell Git which of these you want enabled by setting `core.whitespace` to the values you want on or off, separated by commas. You can disable settings by either leaving them out of the setting string or prepending a `-` in front of the value. For example, if you want all but `cr-at-eol` to be set, you can do this:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git will detect these issues when you run a `git diff` command and try to color them so you can possibly fix them before you commit. It will also use these values to help you when you apply patches with `git apply`. When you're applying patches, you can ask Git to warn you if it's applying patches with the specified whitespace issues:

```
$ git apply --whitespace=warn <patch>
```

Or you can have Git try to automatically fix the issue before applying the patch:

```
$ git apply --whitespace=fix <patch>
```

These options apply to the `git rebase` command as well. If you've committed whitespace issues but haven't yet pushed upstream, you can run `git rebase --whitespace=fix` to have Git automatically fix whitespace issues as it's rewriting the patches.

Server Configuration

Not nearly as many configuration options are available for the server side of Git, but there are a few interesting ones you may want to take note of.

`receive.fsckObjects`

Git is capable of making sure every object received during a push still matches its SHA-1 checksum and points to valid objects. However, it doesn't do this by default; it's a fairly expensive operation, and might slow down the operation, especially on large repositories or pushes. If you want Git to check object consistency on every push, you can force it to do so by setting `receive.fsckObjects` to true:

```
$ git config --system receive.fsckObjects true
```

Now, Git will check the integrity of your repository before each push is accepted to make sure faulty (or malicious) clients aren't introducing corrupt data.

`receive.denyNonFastForwards`

If you rebase commits that you've already pushed and then try to push again, or otherwise try to push a commit to a remote branch that doesn't contain the commit that the remote branch currently points to, you'll be denied. This is generally good policy; but in the case of the rebase, you may determine that you know what you're doing and can force-update the remote branch with a `-f` flag to your push command.

To tell Git to refuse force-pushes, set `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

The other way you can do this is via server-side receive hooks, which we'll cover in a bit. That approach lets you do more complex things like deny non-fast-forwards to a certain subset of users.

`receive.denyDeletes`

One of the workarounds to the `denyNonFastForwards` policy is for the user to delete the branch and then push it back up with the new reference. To avoid this, set `receive.denyDeletes` to true:

```
$ git config --system receive.denyDeletes true
```

This denies any deletion of branches or tags – no user can do it. To remove remote branches, you must remove the ref files from the server manually. There are also more interesting ways to do this on a per-user basis via ACLs, as you'll learn in [An Example Git-Enforced Policy](#).

Git Attributes

Some of these settings can also be specified for a path, so that Git applies those settings only for a subdirectory or subset of files. These path-specific settings are called Git attributes and are set either in a `.gitattributes` file in one of your directories (normally the root of your project) or in the `.git/info/attributes` file if you don't want the attributes file committed with your project.

Using attributes, you can do things like specify separate merge strategies for individual files or directories in your project, tell Git how to diff non-text files, or have Git filter content before you check it into or out of Git. In this section, you'll learn about some of the attributes you can set on your paths in your Git project and see

a few examples of using this feature in practice.

Binary Files

One cool trick for which you can use Git attributes is telling Git which files are binary (in cases it otherwise may not be able to figure out) and giving Git special instructions about how to handle those files. For instance, some text files may be machine generated and not diffable, whereas some binary files can be diffed. You'll see how to tell Git which is which.

Identifying Binary Files

Some files look like text files but for all intents and purposes are to be treated as binary data. For instance, Xcode projects on the Mac contain a file that ends in `.pbxproj`, which is basically a JSON (plain-text JavaScript data format) dataset written out to disk by the IDE, which records your build settings and so on. Although it's technically a text file (because it's all UTF-8), you don't want to treat it as such because it's really a lightweight database – you can't merge the contents if two people change it, and diffs generally aren't helpful. The file is meant to be consumed by a machine. In essence, you want to treat it like a binary file.

To tell Git to treat all `pbxproj` files as binary data, add the following line to your `.gitattributes` file:

```
*.pbxproj binary
```

Now, Git won't try to convert or fix CRLF issues; nor will it try to compute or print a diff for changes in this file when you run `git show` or `git diff` on your project.

Diffing Binary Files

You can also use the Git attributes functionality to effectively diff binary files. You do this by telling Git how to convert your binary data to a text format that can be compared via the normal diff.

First, you'll use this technique to solve one of the most annoying problems known to humanity: version-controlling Microsoft Word documents. Everyone knows that Word is the most horrific editor around, but oddly, everyone still uses it. If you want to version-control Word documents, you can stick them in a Git repository and commit every once in a while; but what good does that do? If you run `git diff` normally, you only see something like this:

```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 88839c4..4afcb7c 100644
Binary files a/chapter1.docx and b/chapter1.docx differ
```

You can't directly compare two versions unless you check them out and scan them manually, right? It turns out you can do this fairly well using Git attributes. Put the following line in your `.gitattributes` file:

```
*.docx diff=word
```

This tells Git that any file that matches this pattern (`.docx`) should use the “word” filter when you try to view a diff that contains changes. What is the “word” filter? You have to set it up. Here you'll configure Git to use the `docx2txt` program to convert Word documents into readable text files, which it will then diff properly.

First, you'll need to install `docx2txt`; you can download it from <http://docx2txt.sourceforge.net>. Follow the instructions in the `INSTALL` file to put it somewhere your shell can find it. Next, you'll write a wrapper script to convert output to the format Git expects. Create a file that's somewhere in your path called `docx2txt`, and add these contents:

```
#!/bin/bash
docx2txt.pl $1 -
```

Don't forget to `chmod a+x` that file. Finally, you can configure Git to use this script:

```
$ git config diff.word.textconv docx2txt
```

Now Git knows that if it tries to do a diff between two snapshots, and any of the files end in `.docx`, it should run those files through the “word” filter, which is defined as the `docx2txt` program. This effectively makes nice text-based versions of your Word files before attempting to diff them.

Here's an example: Chapter 1 of this book was converted to Word format and committed in a Git repository. Then a new paragraph was added. Here's what `git diff` shows:


```
$ git diff
diff --git a/chapter1.docx b/chapter1.docx
index 0b013ca..ba25db5 100644
--- a/chapter1.docx
+++ b/chapter1.docx
@@ -2,6 +2,7 @@
```

This chapter will be about getting started with Git. We will begin at the beginning by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it setup to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all setup to do so.

1.1. About Version Control

What is "version control", and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

+Testing: 1, 2, 3.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

1.1.1. Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

Git successfully and succinctly tells us that we added the string "Testing: 1, 2, 3.", which is correct. It's not perfect – formatting changes wouldn't show up here – but it certainly works.

Another interesting problem you can solve this way involves diffing image files. One way to do this is to run image files through a filter that extracts their EXIF information – metadata that is recorded with most image formats. If you download and install the `exiftool` program, you can use it to convert your images into text about the metadata, so at least the diff will show you a textual representation of any changes that happened:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

If you replace an image in your project and run `git diff`, you see something like this:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
 ExifTool Version Number      : 7.74
-File Size                    : 70 kB
-File Modification Date/Time  : 2009:04:21 07:02:45-07:00
+File Size                    : 94 kB
+File Modification Date/Time  : 2009:04:21 07:02:43-07:00
 File Type                    : PNG
 MIME Type                    : image/png
-Image Width                  : 1058
-Image Height                 : 889
+Image Width                  : 1056
+Image Height                 : 827
 Bit Depth                    : 8
 Color Type                   : RGB with Alpha
```

You can easily see that the file size and image dimensions have both changed.

Keyword Expansion

SVN- or CVS-style keyword expansion is often requested by developers used to those systems. The main problem with this in Git is that you can't modify a file with information about the commit after you've committed, because Git checksums the file first. However, you can inject text into a file when it's checked out and remove it again before it's added to a commit. Git attributes offers you two ways to do this.

First, you can inject the SHA-1 checksum of a blob into an `Id` field in the file automatically. If you set this attribute on a file or set of files, then the next time you check out that branch, Git will replace that field with the SHA-1 of the blob. It's important to notice that it isn't the SHA-1 of the commit, but of the blob itself:

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

The next time you check out this file, Git injects the SHA-1 of the blob:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

However, that result is of limited use. If you've used keyword substitution in CVS or Subversion, you can include a datestamp – the SHA-1 isn't all that helpful, because it's fairly random and you can't tell if one SHA-1 is older or newer than another just by

looking at them.

It turns out that you can write your own filters for doing substitutions in files on commit/checkout. These are called “clean” and “smudge” filters. In the `.gitattributes` file, you can set a filter for particular paths and then set up scripts that will process files just before they’re checked out (“smudge”, see [The “smudge” filter is run on checkout.](#)) and just before they’re staged (“clean”, see [The “clean” filter is run when files are staged.](#)). These filters can be set to do all sorts of fun things.

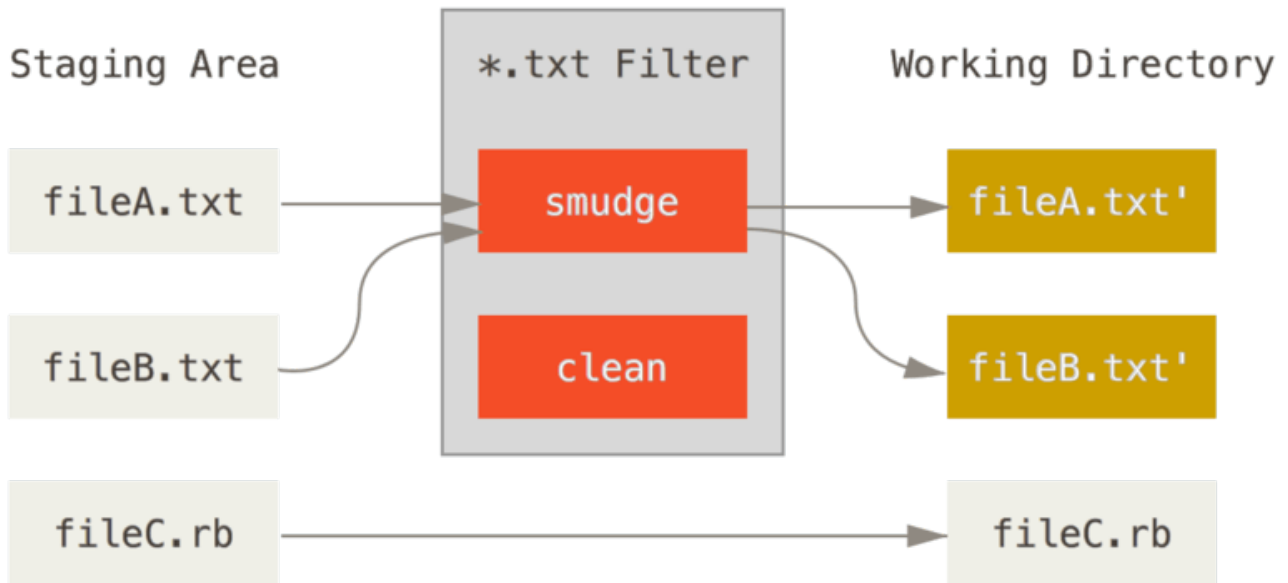


Figure 144. The “smudge” filter is run on checkout.

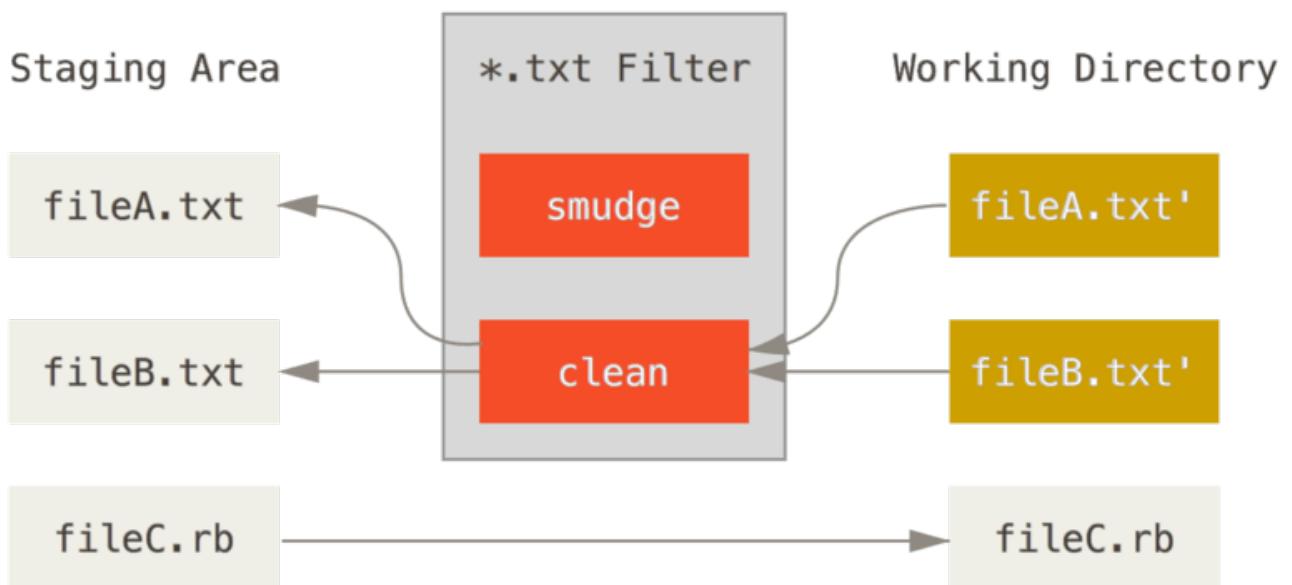


Figure 145. The “clean” filter is run when files are staged.

The original commit message for this feature gives a simple example of running all your C source code through the `indent` program before committing. You can set it up by setting the filter attribute in your `.gitattributes` file to filter `*.c` files with the “indent” filter:

```
*.c filter=indent
```

Then, tell Git what the “indent” filter does on smudge and clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In this case, when you commit files that match `*.c`, Git will run them through the `indent` program before it stages them and then run them through the `cat` program before it checks them back out onto disk. The `cat` program does essentially nothing: it spits out the same data that it comes in. This combination effectively filters all C source code files through `indent` before committing.

Another interesting example gets `$Date$` keyword expansion, RCS style. To do this properly, you need a small script that takes a filename, figures out the last commit date for this project, and inserts the date into the file. Here is a small Ruby script that does that:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:@"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

All the script does is get the latest commit date from the `git log` command, stick that into any `$Date$` strings it sees in stdin, and print the results – it should be simple to do in whatever language you’re most comfortable in. You can name this file `expand_date` and put it in your path. Now, you need to set up a filter in Git (call it `dater`) and tell it to use your `expand_date` filter to smudge the files on checkout. You’ll use a Perl expression to clean that up on commit:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\$]*\\$/\\\$Date\\$/'"
```

This Perl snippet strips out anything it sees in a `$Date$` string, to get back to where you started. Now that your filter is ready, you can test it by setting up a file with your `$Date$` keyword and then setting up a Git attribute for that file that engages the new filter:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

If you commit those changes and check out the file again, you see the keyword properly substituted:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

You can see how powerful this technique can be for customized applications. You have to be careful, though, because the `.gitattributes` file is committed and passed around with the project, but the driver (in this case, `dater`) isn't, so it won't work everywhere. When you design these filters, they should be able to fail gracefully and have the project still work properly.

Exporting Your Repository

Git attribute data also allows you to do some interesting things when exporting an archive of your project.

`export-ignore`

You can tell Git not to export certain files or directories when generating an archive. If there is a subdirectory or file that you don't want to include in your archive file but that you do want checked into your project, you can determine those files via the `export-ignore` attribute.

For example, say you have some test files in a `test/` subdirectory, and it doesn't make sense to include them in the tarball export of your project. You can add the following line to your Git attributes file:

```
test/ export-ignore
```

Now, when you run `git archive` to create a tarball of your project, that directory won't be included in the archive.

`export-subst`

When exporting files for deployment you can apply `git log`'s formatting and keyword-expansion processing to selected portions of files marked with the `export-subst` attribute.

For instance, if you want to include a file named `LAST_COMMIT` in your project, and have metadata about the last commit automatically injected into it when `git archive` runs, you can for example set up the file like this:

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

When you run `git archive`, the contents of the archived file will look like this:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

The substitutions can include for example the commit message and any git notes, and git log can do simple word wrapping:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log's custom formatter

git archive uses git log's `pretty=format:` processor
directly, and strips the surrounding `$Format:` and `$`
markup from the output.
`

$ git archive @ | tar xf0 - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
    export-subst uses git log's custom formatter

    git archive uses git log's `pretty=format:` processor directly, and
    strips the surrounding `$Format:` and `$` markup from the output.
```

The resulting archive is suitable for deployment work, but like any exported archive it isn't suitable for further development work.

Merge Strategies

You can also use Git attributes to tell Git to use different merge strategies for specific files in your project. One very useful option is to tell Git to not try to merge specific files when they have conflicts, but rather to use your side of the merge over someone else's.

This is helpful if a branch in your project has diverged or is specialized, but you want to be able to merge changes back in from it, and you want to ignore certain files. Say you have a database settings file called `database.xml` that is different in two branches, and you want to merge in your other branch without messing up the database file. You can set up an attribute like this:

```
database.xml merge=ours
```

And then define a dummy `ours` merge strategy with:

```
$ git config --global merge.ours.driver true
```

If you merge in the other branch, instead of having merge conflicts with the `database.xml` file, you see something like this:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In this case, `database.xml` stays at whatever version you originally had.

Git kljuka

Kot mnogi drugi sistemi nadzora različic ima Git način zagona skript po meri, ko se zgodijo določene pomembne akcije. Na voljo sta dve skupini teh kljuk: na strani klienta in na strani strežnika. Kljuka na strani klienta so sprožene z operacijami, kot so pošiljanje in združevanje, medtem ko so kljuka strežniške strani gnane na omrežju. Client-side hooks are triggered by operations such as committing and merging, while server-side hooks run on network operations such as receiving pushed commits. You can use these hooks for all sorts of reasons.

Installing a Hook

The hooks are all stored in the `hooks` subdirectory of the Git directory. In most projects, that's `.git/hooks`. When you initialize a new repository with `git init`, Git populates the hooks directory with a bunch of example scripts, many of which are useful by themselves; but they also document the input values of each script. All the examples are written as shell scripts, with some Perl thrown in, but any properly named executable scripts will work fine – you can write them in Ruby or Python or what have you. If you want to use the bundled hook scripts, you'll have to rename them; their file names all end with `.sample`.

To enable a hook script, put a file in the `hooks` subdirectory of your Git directory that is named appropriately and is executable. From that point forward, it should be called. We'll cover most of the major hook filenames here.

Client-Side Hooks

There are a lot of client-side hooks. This section splits them into committing-workflow hooks, e-mail-workflow scripts, and everything else.

NOTE

It's important to note that client-side hooks are **not** copied when you clone a repository. If your intent with these scripts is to enforce a policy, you'll probably want to do that on the server side; see the example in [An Example Git-Enforced Policy](#).

Committing-Workflow Hooks

The first four hooks have to do with the committing process.

The `pre-commit` hook is run first, before you even type in a commit message. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests run, or to examine whatever you need to inspect in the code. Exiting non-zero from this hook aborts the commit, although you can bypass it with `git commit --no-verify`. You can do things like check for code style (run `lint` or something equivalent), check for trailing whitespace (the default hook does exactly this), or check for appropriate documentation on new methods.

The `prepare-commit-msg` hook is run before the commit message editor is fired up but after the default message is created. It lets you edit the default message before the commit author sees it. This hook takes a few parameters: the path to the file that holds the commit message so far, the type of commit, and the commit SHA-1 if this is an amended commit. This hook generally isn't useful for normal commits; rather, it's good for commits where the default message is auto-generated, such as templated commit messages, merge commits, squashed commits, and amended commits. You may use it in conjunction with a commit template to programmatically insert information.

The `commit-msg` hook takes one parameter, which again is the path to a temporary file that contains the commit message written by the developer. If this script exits non-zero, Git aborts the commit process, so you can use it to validate your project state or commit message before allowing a commit to go through. In the last section of this chapter, we'll demonstrate using this hook to check that your commit message is conformant to a required pattern.

After the entire commit process is completed, the `post-commit` hook runs. It doesn't take any parameters, but you can easily get the last commit by running `git log -1 HEAD`. Generally, this script is used for notification or something similar.

E-mail Workflow Hooks

You can set up three client-side hooks for an e-mail-based workflow. They're all invoked by the `git am` command, so if you aren't using that command in your workflow, you can safely skip to the next section. If you're taking patches over e-mail prepared by `git format-patch`, then some of these may be helpful to you.

The first hook that is run is `applypatch-msg`. It takes a single argument: the name of the temporary file that contains the proposed commit message. Git aborts the patch if this script exits non-zero. You can use this to make sure a commit message is properly formatted, or to normalize the message by having the script edit it in place.

The next hook to run when applying patches via `git am` is `pre-applypatch`. Somewhat confusingly, it is run *after* the patch is applied but before a commit is made, so you can use it to inspect the snapshot before making the commit. You can run tests or otherwise inspect the working tree with this script. If something is missing or the tests don't pass, exiting non-zero aborts the `git am` script without committing the patch.

The last hook to run during a `git am` operation is `post-applypatch`, which runs after the commit is made. You can use it to notify a group or the author of the patch you pulled in that you've done so. You can't stop the patching process with this script.

Other Client Hooks

The `pre-rebase` hook runs before you rebase anything and can halt the process by exiting non-zero. You can use this hook to disallow rebasing any commits that have already been pushed. The example `pre-rebase` hook that Git installs does this, although it makes some assumptions that may not match with your workflow.

The `post-rewrite` hook is run by commands that replace commits, such as `git commit --amend` and `git rebase` (though not by `git filter-branch`). Its single argument is which command triggered the rewrite, and it receives a list of rewrites on `stdin`. This hook has many of the same uses as the `post-checkout` and `post-merge` hooks.

After you run a successful `git checkout`, the `post-checkout` hook runs; you can use it to set up your working directory properly for your project environment. This may mean moving in large binary files that you don't want source controlled, auto-generating documentation, or something along those lines.

The `post-merge` hook runs after a successful `merge` command. You can use it to restore data in the working tree that Git can't track, such as permissions data. This hook can likewise validate the presence of files external to Git control that you may want copied in when the working tree changes.

The `pre-push` hook runs during `git push`, after the remote refs have been updated but before any objects have been transferred. It receives the name and location of the remote as parameters, and a list of to-be-updated refs through `stdin`. You can use it to validate a set of ref updates before a push occurs (a non-zero exit code will abort the push).

Git occasionally does garbage collection as part of its normal operation, by invoking `git gc --auto`. The `pre-auto-gc` hook is invoked just before the garbage collection takes place, and can be used to notify you that this is happening, or to abort the collection if now isn't a good time.

Server-Side Hooks

In addition to the client-side hooks, you can use a couple of important server-side hooks as a system administrator to enforce nearly any kind of policy for your project. These scripts run before and after pushes to the server. The pre hooks can exit non-zero at any time to reject the push as well as print an error message back to the client; you can set up a push policy that's as complex as you wish.

`pre-receive`

The first script to run when handling a push from a client is `pre-receive`. It takes a list of references that are being pushed from `stdin`; if it exits non-zero, none of them are accepted. You can use this hook to do things like make sure none of the updated references are non-fast-forwards, or to do access control for all the refs and files they're modifying with the push.

update

The `update` script is very similar to the `pre-receive` script, except that it's run once for each branch the pusher is trying to update. If the pusher is trying to push to multiple branches, `pre-receive` runs only once, whereas `update` runs once per branch they're pushing to. Instead of reading from stdin, this script takes three arguments: the name of the reference (branch), the SHA-1 that reference pointed to before the push, and the SHA-1 the user is trying to push. If the `update` script exits non-zero, only that reference is rejected; other references can still be updated.

post-receive

The `post-receive` hook runs after the entire process is completed and can be used to update other services or notify users. It takes the same stdin data as the `pre-receive` hook. Examples include e-mailing a list, notifying a continuous integration server, or updating a ticket-tracking system – you can even parse the commit messages to see if any tickets need to be opened, modified, or closed. This script can't stop the push process, but the client doesn't disconnect until it has completed, so be careful if you try to do anything that may take a long time.

An Example Git-Enforced Policy

In this section, you'll use what you've learned to establish a Git workflow that checks for a custom commit message format, and allows only certain users to modify certain subdirectories in a project. You'll build client scripts that help the developer know if their push will be rejected and server scripts that actually enforce the policies.

The scripts we'll show are written in Ruby; partly because of our intellectual inertia, but also because Ruby is easy to read, even if you can't necessarily write it. However, any language will work – all the sample hook scripts distributed with Git are in either Perl or Bash, so you can also see plenty of examples of hooks in those languages by looking at the samples.

Server-Side Hook

All the server-side work will go into the `update` file in your `hooks` directory. The `update` hook runs once per branch being pushed and takes three arguments:

- The name of the reference being pushed to
- The old revision where that branch was
- The new revision being pushed

You also have access to the user doing the pushing if the push is being run over SSH. If you've allowed everyone to connect with a single user (like "git") via public-key authentication, you may have to give that user a shell wrapper that determines which user is connecting based on the public key, and set an environment variable accordingly. Here we'll assume the connecting user is in the `$USER` environment variable, so your `update` script begins by gathering all the information you need:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies..."
puts "(#{ $refname }) (#{ $oldrev[0,6] }) (#{ $newrev[0,6] })"
```

Yes, those are global variables. Don't judge – it's easier to demonstrate this way.

Enforcing a Specific Commit-Message Format

Your first challenge is to enforce that each commit message adheres to a particular format. Just to have a target, assume that each message has to include a string that looks like “ref: 1234” because you want each commit to link to a work item in your ticketing system. You must look at each commit being pushed up, see if that string is in the commit message, and, if the string is absent from any of the commits, exit non-zero so the push is rejected.

You can get a list of the SHA-1 values of all the commits that are being pushed by taking the `$newrev` and `$oldrev` values and passing them to a Git plumbing command called `git rev-list`. This is basically the `git log` command, but by default it prints out only the SHA-1 values and no other information. So, to get a list of all the commit SHA-1s introduced between one commit SHA and another, you can run something like this:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

You can take that output, loop through each of those commit SHA-1s, grab the message for it, and test that message against a regular expression that looks for a pattern.

You have to figure out how to get the commit message from each of these commits to test. To get the raw commit data, you can use another plumbing command called `git cat-file`. We'll go over all these plumbing commands in detail in [Notranjost Git-a](#); but for now, here's what that command gives you:

```
$ git cat-file commit ca82a6
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

A simple way to get the commit message from a commit when you have the SHA-1 value is to go to the first blank line and take everything after that. You can do so with the `sed` command on Unix systems:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

You can use that incantation to grab the commit message from each commit that is trying to be pushed and exit if you see anything that doesn't match. To exit the script and reject the push, exit non-zero. The whole method looks like this:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^\$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
end
check_message_format
```

Putting that in your `update` script will reject updates that contain commits that have messages that don't adhere to your rule.

Enforcing a User-Based ACL System

Suppose you want to add a mechanism that uses an access control list (ACL) that specifies which users are allowed to push changes to which parts of your projects. Some people have full access, and others can only push changes to certain subdirectories or specific files. To enforce this, you'll write those rules to a file named `acl` that lives in your bare Git repository on the server. You'll have the `update` hook look at those rules, see what files are being introduced for all the commits being pushed, and determine whether the user doing the push has access to update all those files.

The first thing you'll do is write your ACL. Here you'll use a format very much like the CVS ACL mechanism: it uses a series of lines, where the first field is `avail` or `unavail`, the next field is a comma-delimited list of the users to which the rule applies, and the last field is the path to which the rule applies (blank meaning open access). All of these fields are delimited by a pipe (`|`) character.

In this case, you have a couple of administrators, some documentation writers with access to the `doc` directory, and one developer who only has access to the `lib` and `tests` directories, and your ACL file looks like this:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

You begin by reading this data into a structure that you can use. In this case, to keep the example simple, you'll only enforce the `avail` directives. Here is a method that gives you an associative array where the key is the user name and the value is an array of paths to which the user has write access:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

On the ACL file you looked at earlier, this `get_acl_access_data` method returns a data structure that looks like this:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Now that you have the permissions sorted out, you need to determine what paths the commits being pushed have modified, so you can make sure the user who's pushing has access to all of them.

You can pretty easily see what files have been modified in a single commit with the `--name-only` option to the `git log` command (mentioned briefly in Chapter 2):

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

If you use the ACL structure returned from the `get_acl_access_data` method and check it against the listed files in each of the commits, you can determine whether the user has access to push all of their commits:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path # user has access to everything
          || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

You get a list of new commits being pushed to your server with `git rev-list`. Then, for each of those commits, you find which files are modified and make sure the user who's pushing has access to all the paths being modified.

Now your users can't push any commits with badly formed messages or with modified

files outside of their designated paths.

Testing It Out

If you run `chmod u+x .git/hooks/update`, which is the file into which you should have put all this code, and then try to push a commit with a non-compliant message, you get something like this:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

There are a couple of interesting things here. First, you see this where the hook starts running.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Remember that you printed that out at the very beginning of your update script. Anything your script echoes to `stdout` will be transferred to the client.

The next thing you'll notice is the error message.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

The first line was printed out by you, the other two were Git telling you that the update script exited non-zero and that is what is declining your push. Lastly, you have this:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

You'll see a remote rejected message for each reference that your hook declined, and it tells you that it was declined specifically because of a hook failure.

Furthermore, if someone tries to edit a file they don't have access to and push a commit containing it, they will see something similar. For instance, if a documentation author tries to push a commit modifying something in the `lib` directory, they see

```
[POLICY] You do not have access to push to lib/test.rb
```

From now on, as long as that `update` script is there and executable, your repository will never have a commit message without your pattern in it, and your users will be sandboxed.

Client-Side Hooks

The downside to this approach is the whining that will inevitably result when your users' commit pushes are rejected. Having their carefully crafted work rejected at the last minute can be extremely frustrating and confusing; and furthermore, they will have to edit their history to correct it, which isn't always for the faint of heart.

The answer to this dilemma is to provide some client-side hooks that users can run to notify them when they're doing something that the server is likely to reject. That way, they can correct any problems before committing and before those issues become more difficult to fix. Because hooks aren't transferred with a clone of a project, you must distribute these scripts some other way and then have your users copy them to their `.git/hooks` directory and make them executable. You can distribute these hooks within the project or in a separate project, but Git won't set them up automatically.

To begin, you should check your commit message just before each commit is recorded, so you know the server won't reject your changes due to badly formatted commit messages. To do this, you can add the `commit-msg` hook. If you have it read the message from the file passed as the first argument and compare that to the pattern, you can force Git to abort the commit if there is no match:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

If that script is in place (in `.git/hooks/commit-msg`) and executable, and you commit with a message that isn't properly formatted, you see this:


```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

No commit was completed in that instance. However, if your message contains the proper pattern, Git allows you to commit:

```
$ git commit -am 'test [ref: 132]'
```

```
[master e05c914] test [ref: 132]
 1 file changed, 1 insertions(+), 0 deletions(-)
```

Next, you want to make sure you aren't modifying files that are outside your ACL scope. If your project's `.git` directory contains a copy of the ACL file you used previously, then the following `pre-commit` script will enforce those constraints for you:

```
#!/usr/bin/env ruby

$user = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
      if !access_path || (path.index(access_path) == 0)
        has_file_access = true
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms
```

This is roughly the same script as the server-side part, but with two important differences. First, the ACL file is in a different place, because this script runs from your working directory, not from your `.git` directory. You have to change the path to the ACL file from this

```
access = get_acl_access_data('acl')
```

to this:

```
access = get_acl_access_data('.git/acl')
```

The other important difference is the way you get a listing of the files that have been changed. Because the server-side method looks at the log of commits, and, at this point, the commit hasn't been recorded yet, you must get your file listing from the staging area instead. Instead of

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

you have to use

```
files_modified = `git diff-index --cached --name-only HEAD`
```

But those are the only two differences – otherwise, the script works the same way. One caveat is that it expects you to be running locally as the same user you push as to the remote machine. If that is different, you must set the `$user` variable manually.

One other thing we can do here is make sure the user doesn't push non-fast-forwarded references. To get a reference that isn't a fast-forward, you either have to rebase past a commit you've already pushed up or try pushing a different local branch up to the same remote branch.

Presumably, the server is already configured with `receive.denyDeletes` and `receive.denyNonFastForwards` to enforce this policy, so the only accidental thing you can try to catch is rebasing commits that have already been pushed.

Here is an example pre-rebase script that checks for that. It gets a list of all the commits you're about to rewrite and checks whether they exist in any of your remote references. If it sees one that is reachable from one of your remote references, it aborts the rebase.

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
end
```

This script uses a syntax that wasn't covered in the Revision Selection section of Chapter 6. You get a list of commits that have already been pushed up by running this:

```
`git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
```

The `SHA^@` syntax resolves to all the parents of that commit. You're looking for any commit that is reachable from the last commit on the remote and that isn't reachable from any parent of any of the SHA-1s you're trying to push up – meaning it's a fast-forward.

The main drawback to this approach is that it can be very slow and is often unnecessary – if you don't try to force the push with `-f`, the server will warn you and not accept the push. However, it's an interesting exercise and can in theory help you avoid a rebase that you might later have to go back and fix.

Povzetek

Pokrili smo večino glavnih načinov, s katerimi lahko prilagodite vašega klienta Git in strežnik, da se najboljše ujema z vašim potekom dela in projekti. Naučili ste se o vseh vrstah konfiguracijskih nastavitev, atributih datotečne osnove in kljukah dogodkov ter zgradili ste primer strežnika z vsiljevanjem politike. Sedaj bi morali biti sposobni narediti, da se Git prilagaja s skoraj katerimkoli potekom dela, ki si ga lahko zamislite.

Git in drugi sistemi

Svet ni perfekten. Običajno ne morete takoj preklopiti vsakega projekta, s katerim pridete v kontakt z Git-om. Včasih obtičite na projektu, ki uporablja drug VCS in želite, da bi bil Git. Prvi del tega poglavja bomo porabili za učenje o načinih, kako uporabiti Git kot klient, ko je projekt na katerem delate gostovan na različnem sistemu.

Na neki točki, boste morda želeli pretvoriti vaš obstoječi projekt v Git. Drugi del tega poglavja pokriva, kako migrirati vaš projekt v Git iz večih določenih sistemov kot tudi metodo, ki bo delovala, če ne obstaja nobeno vnaprej zgrajeno orodje za uvažanje.

Git kot klient

Git ponuja tako lepo izkušnjo za razvijalce, ki so jo mnogi ljudje ugotovili, kako jo uporabljati na njihovih delovnih postajah, tudi če preostanek njihove ekipe uporablja v celoti različen VCS. Obstaja število teh pretvornikov na voljo imenovanih "bridges,". Tu bomo pokrili tiste, na katere boste najverjetneje naleteli tam zunaj.

Git and Subversion

A large fraction of open source development projects and a good number of corporate projects use Subversion to manage their source code. It's been around for more than a decade, and for most of that time was the *de facto* VCS choice for open-source projects. It's also very similar in many ways to CVS, which was the big boy of the source-control world before that.

One of Git's great features is a bidirectional bridge to Subversion called `git svn`. This tool allows you to use Git as a valid client to a Subversion server, so you can use all the local features of Git and then push to a Subversion server as if you were using Subversion locally. This means you can do local branching and merging, use the staging area, use rebasing and cherry-picking, and so on, while your collaborators continue to work in their dark and ancient ways. It's a good way to sneak Git into the corporate environment and help your fellow developers become more efficient while you lobby to get the infrastructure changed to support Git fully. The Subversion bridge is the gateway drug to the DVCS world.

`git svn`

The base command in Git for all the Subversion bridging commands is `git svn`. It takes quite a few commands, so we'll show the most common while going through a few simple workflows.

It's important to note that when you're using `git svn`, you're interacting with Subversion, which is a system that works very differently from Git. Although you **can** do local branching and merging, it's generally best to keep your history as linear as possible by rebasing your work, and avoiding doing things like simultaneously interacting with a Git remote repository.

Don't rewrite your history and try to push again, and don't push to a parallel Git repository to collaborate with fellow Git developers at the same time. Subversion can have only a single linear history, and confusing it is very easy. If you're working with a team, and some are using SVN and others are using Git, make sure everyone is using the SVN server to collaborate – doing so will make your life easier.

Setting Up

To demonstrate this functionality, you need a typical SVN repository that you have write access to. If you want to copy these examples, you'll have to make a writeable copy of my test repository. In order to do that easily, you can use a tool called `svnsync` that comes with Subversion. For these tests, we created a new Subversion repository on Google Code that was a partial copy of the `protobuf` project, which is a tool that encodes structured data for network transmission.

To follow along, you first need to create a new local Subversion repository:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Then, enable all users to change revprops – the easy way is to add a `pre-revprop-change` script that always exits 0:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

You can now sync this project to your local machine by calling `svnsync init` with the to and from repositories.

```
$ svnsync init file:///tmp/test-svn \
http://progit-example.googlecode.com/svn/
```

This sets up the properties to run the sync. You can then clone the code by running

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Transmitting file data .....[...]
Committed revision 2.
Copied properties for revision 2.
[...]
```

Although this operation may take only a few minutes, if you try to copy the original repository to another remote repository instead of a local one, the process will take

nearly an hour, even though there are fewer than 100 commits. Subversion has to clone one revision at a time and then push it back into another repository – it’s ridiculously inefficient, but it’s the only easy way to do this.

Getting Started

Now that you have a Subversion repository to which you have write access, you can go through a typical workflow. You’ll start with the `git svn clone` command, which imports an entire Subversion repository into a local Git repository. Remember that if you’re importing from a real hosted Subversion repository, you should replace the `file:///tmp/test-svn` here with the URL of your Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcbfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
  A   m4/acx_pthread.m4
  A   m4/stl_hash.m4
  A   java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
  A   java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
file:///tmp/test-svn/trunk r75
```

This runs the equivalent of two commands – `git svn init` followed by `git svn fetch` – on the URL you provide. This can take a while. The test project has only about 75 commits and the codebase isn’t that big, but Git has to check out each version, one at a time, and commit it individually. For a project with hundreds or thousands of commits, this can literally take hours or even days to finish.

The `-T trunk -b branches -t tags` part tells Git that this Subversion repository follows the basic branching and tagging conventions. If you name your trunk, branches, or tags differently, you can change these options. Because this is so common, you can replace this entire part with `-s`, which means standard layout and implies all those options. The following command is equivalent:

```
$ git svn clone file:///tmp/test-svn -s
```

At this point, you should have a valid Git repository that has imported your branches and tags:

```
$ git branch -a
* master
remotes/origin/my-calc-branch
remotes/origin/tags/2.0.2
remotes/origin/tags/release-2.0.1
remotes/origin/tags/release-2.0.2
remotes/origin/tags/release-2.0.2rc1
remotes/origin/trunk
```

Note how this tool manages Subversion tags as remote refs. Let's take a closer look with the Git plumbing command `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doesn't do this when it clones from a Git server; here's what a repository with tags looks like after a fresh clone:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebcd695e2e4193db5e refs/tags/v1.0.0
```

Git fetches the tags directly into `refs/tags`, rather than treating them remote branches.

Committing Back to Subversion

Now that you have a working repository, you can do some work on the project and push your commits back upstream, using Git effectively as a SVN client. If you edit one of the files and commit it, you have a commit that exists in Git locally that doesn't exist on the Subversion server:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
1 file changed, 5 insertions(+)
```

Next, you need to push your change upstream. Notice how this changes the way you work with Subversion – you can do several commits offline and then push them all at once to the Subversion server. To push to a Subversion server, you run the `git svn dcommit` command:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r77
M README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

This takes all the commits you've made on top of the Subversion server code, does a Subversion commit for each, and then rewrites your local Git commit to include a unique identifier. This is important because it means that all the SHA-1 checksums for your commits change. Partly for this reason, working with Git-based remote versions of your projects concurrently with a Subversion server isn't a good idea. If you look at the last commit, you can see the new `git-svn-id` that was added:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date: Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Notice that the SHA-1 checksum that originally started with `4af61fd` when you committed now begins with `95e0222`. If you want to push to both a Git server and a Subversion server, you have to push (`dcommit`) to the Subversion server first, because that action changes your commit data.

Pulling in New Changes

If you're working with other developers, then at some point one of you will push, and then the other one will try to push a change that conflicts. That change will be rejected until you merge in their work. In `git svn`, it looks like this:


```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcd218730ddf3a2da4eb39138 M README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

To resolve this situation, you can run `git svn rebase`, which pulls down any changes on the server that you don't have yet and rebases any work you have on top of what is on the server:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 65536c6e30d263495c17d781962cfff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Now, all your work is on top of what is on the Subversion server, so you can successfully `dcommit`:

```

$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M   README.txt
Committed r85
M   README.txt
r85 = 9c29704cc0bbbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk

```

Note that unlike Git, which requires you to merge upstream work you don't yet have locally before you can push, `git svn` makes you do that only if the changes conflict (much like how Subversion works). If someone else pushes a change to one file and then you push a change to another file, your `dcommit` will work fine:

```

$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M   configure.ac
Committed r87
M   autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
M   configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fef2b1d92806 and refs/remotes/origin/trunk differ,
using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M   autogen.sh
First, rewinding head to replay your work on top of it...

```

This is important to remember, because the outcome is a project state that didn't exist on either of your computers when you pushed. If the changes are incompatible but don't conflict, you may get issues that are difficult to diagnose. This is different than using a Git server – in Git, you can fully test the state on your client system before publishing it, whereas in SVN, you can't ever be certain that the states immediately before commit and after commit are identical.

You should also run this command to pull in changes from the Subversion server, even if you're not ready to commit yourself. You can run `git svn fetch` to grab the new data, but `git svn rebase` does the fetch and then updates your local commits.

```

$ git svn rebase
M   autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.

```

Running `git svn rebase` every once in a while makes sure your code is always up to

date. You need to be sure your working directory is clean when you run this, though. If you have local changes, you must either stash your work or temporarily commit it before running `git svn rebase` – otherwise, the command will stop if it sees that the rebase will result in a merge conflict.

Git Branching Issues

When you've become comfortable with a Git workflow, you'll likely create topic branches, do work on them, and then merge them in. If you're pushing to a Subversion server via `git svn`, you may want to rebase your work onto a single branch each time instead of merging branches together. The reason to prefer rebasing is that Subversion has a linear history and doesn't deal with merges like Git does, so `git svn` follows only the first parent when converting the snapshots into Subversion commits.

Suppose your history looks like the following: you created an `experiment` branch, did two commits, and then merged them back into `master`. When you `dcommit`, you see output like this:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
  M   CHANGES.txt
Committed r89
  M   CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
  M   COPYING.txt
  M   INSTALL.txt
Committed r90
  M   INSTALL.txt
  M   COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Running `dcommit` on a branch with merged history works fine, except that when you look at your Git project history, it hasn't rewritten either of the commits you made on the `experiment` branch – instead, all those changes appear in the SVN version of the single merge commit.

When someone else clones that work, all they see is the merge commit with all the work squashed into it, as though you ran `git merge --squash`; they don't see the commit data about where it came from or when it was committed.

Subversion Branching

Branching in Subversion isn't the same as branching in Git; if you can avoid using it much, that's probably best. However, you can create and commit to branches in Subversion using `git svn`.

Creating a New SVN Branch

To create a new branch in Subversion, you run `git svn branch [branchname]:`

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

This does the equivalent of the `svn copy trunk branches/opera` command in Subversion and operates on the Subversion server. It's important to note that it doesn't check you out into that branch; if you commit at this point, that commit will go to `trunk` on the server, not `opera`.

Switching Active Branches

Git figures out what branch your `dcommit`s go to by looking for the tip of any of your Subversion branches in your history – you should have only one, and it should be the last one with a `git-svn-id` in your current branch history.

If you want to work on more than one branch simultaneously, you can set up local branches to `dcommit` to specific Subversion branches by starting them at the imported Subversion commit for that branch. If you want an `opera` branch that you can work on separately, you can run

```
$ git branch opera remotes/origin/opera
```

Now, if you want to merge your `opera` branch into `trunk` (your `master` branch), you can do so with a normal `git merge`. But you need to provide a descriptive commit message (via `-m`), or the merge will say “Merge branch opera” instead of something useful.

Remember that although you're using `git merge` to do this operation, and the merge likely will be much easier than it would be in Subversion (because Git will automatically detect the appropriate merge base for you), this isn't a normal Git merge commit. You have to push this data back to a Subversion server that can't handle a commit that tracks more than one parent; so, after you push it up, it will look like a single commit that squashed in all the work of another branch under a single commit. After you merge one branch into another, you can't easily go back and continue working on that branch, as you normally can in Git. The `dcommit` command that you run erases any information that says what branch was merged in, so subsequent merge-base calculations will be wrong – the `dcommit` makes your `git merge` result look like you ran `git merge --squash`. Unfortunately, there's no good way to avoid this situation – Subversion can't store this information, so you'll always be crippled by

its limitations while you're using it as your server. To avoid issues, you should delete the local branch (in this case, `opera`) after you merge it into trunk.

Subversion Commands

The `git svn` toolset provides a number of commands to help ease the transition to Git by providing some functionality that's similar to what you had in Subversion. Here are a few commands that give you what Subversion used to.

SVN Style History

If you're used to Subversion and want to see your history in SVN output style, you can run `git svn log` to view your commit history in SVN formatting:

```
$ git svn log
-----
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines

autogen change

-----
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines

Merge branch 'experiment'

-----
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines

updated the changelog
```

You should know two important things about `git svn log`. First, it works offline, unlike the real `svn log` command, which asks the Subversion server for the data. Second, it only shows you commits that have been committed up to the Subversion server. Local Git commits that you haven't dcommitted don't show up; neither do commits that people have made to the Subversion server in the meantime. It's more like the last known state of the commits on the Subversion server.

SVN Annotation

Much as the `git svn log` command simulates the `svn log` command offline, you can get the equivalent of `svn annotate` by running `git svn blame [FILE]`. The output looks like this:

```

$ git svn blame README.txt
2   temporal Protocol Buffers - Google's data interchange format
2   temporal Copyright 2008 Google Inc.
2   temporal http://code.google.com/apis/protocolbuffers/
2   temporal
22  temporal C++ Installation - Unix
22  temporal =====
2   temporal
79  schacon Committing in git-svn.
78  schacon
2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2   temporal Buffer compiler (protoc) execute the following:
2   temporal

```

Again, it doesn't show commits that you did locally in Git or that have been pushed to Subversion in the meantime.

SVN Server Information

You can also get the same sort of information that `svn info` gives you by running `git svn info`:

```

$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)

```

This is like `blame` and `log` in that it runs offline and is up to date only as of the last time you communicated with the Subversion server.

Ignoring What Subversion Ignores

If you clone a Subversion repository that has `svn:ignore` properties set anywhere, you'll likely want to set corresponding `.gitignore` files so you don't accidentally commit files that you shouldn't. `git svn` has two commands to help with this issue. The first is `git svn create-ignore`, which automatically creates corresponding `.gitignore` files for you so your next commit can include them.

The second command is `git svn show-ignore`, which prints to stdout the lines you need to put in a `.gitignore` file so you can redirect the output into your project exclude file:

```
$ git svn show-ignore > .git/info/exclude
```

That way, you don't litter the project with `.gitignore` files. This is a good option if you're the only Git user on a Subversion team, and your teammates don't want `.gitignore` files in the project.

Git-Svn Summary

The `git svn` tools are useful if you're stuck with a Subversion server, or are otherwise in a development environment that necessitates running a Subversion server. You should consider it crippled Git, however, or you'll hit issues in translation that may confuse you and your collaborators. To stay out of trouble, try to follow these guidelines:

- Keep a linear Git history that doesn't contain merge commits made by `git merge`. Rebase any work you do outside of your mainline branch back onto it; don't merge it in.
- Don't set up and collaborate on a separate Git server. Possibly have one to speed up clones for new developers, but don't push anything to it that doesn't have a `git-svn-id` entry. You may even want to add a `pre-receive` hook that checks each commit message for a `git-svn-id` and rejects pushes that contain commits without it.

If you follow those guidelines, working with a Subversion server can be more bearable. However, if it's possible to move to a real Git server, doing so can gain your team a lot more.

Git and Mercurial

The DVCS universe is larger than just Git. In fact, there are many other systems in this space, each with their own angle on how to do distributed version control correctly. Apart from Git, the most popular is Mercurial, and the two are very similar in many respects.

The good news, if you prefer Git's client-side behavior but are working with a project whose source code is controlled with Mercurial, is that there's a way to use Git as a client for a Mercurial-hosted repository. Since the way Git talks to server repositories is through remotes, it should come as no surprise that this bridge is implemented as a remote helper. The project's name is `git-remote-hg`, and it can be found at <https://github.com/felipec/git-remote-hg>.

`git-remote-hg`

First, you need to install `git-remote-hg`. This basically entails dropping its file somewhere in your path, like so:

```
$ curl -o ~/bin/git-remote-hg \  
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg  
$ chmod +x ~/bin/git-remote-hg
```

...assuming `~/bin` is in your `$PATH`. `Git-remote-hg` has one other dependency: the `mercurial` library for Python. If you have Python installed, this is as simple as:

```
$ pip install mercurial
```

(If you don't have Python installed, visit <https://www.python.org/> and get it first.)

The last thing you'll need is the Mercurial client. Go to <https://mercurial-scm.org/> and install it if you haven't already.

Now you're ready to rock. All you need is a Mercurial repository you can push to. Fortunately, every Mercurial repository can act this way, so we'll just use the "hello world" repository everyone uses to learn Mercurial:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

Getting Started

Now that we have a suitable "server-side" repository, we can go through a typical workflow. As you'll see, these two systems are similar enough that there isn't much friction.

As always with Git, first we clone:

```
$ git clone hg::/tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard "hello, world" program
```

You'll notice that working with a Mercurial repository uses the standard `git clone` command. That's because `git-remote-hg` is working at a fairly low level, using a similar mechanism to how Git's HTTP/S protocol is implemented (remote helpers). Since Git and Mercurial are both designed for every client to have a full copy of the repository history, this command makes a full clone, including all the project's history, and does it fairly quickly.

The `log` command shows two commits, the latest of which is pointed to by a whole slew of refs. It turns out some of these aren't actually there. Let's take a look at what's actually in the `.git` directory:


```

$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   ├── origin
│   │   ├── bookmarks
│   │   │   └── master
│   │   └── branches
│   │       └── default
├── notes
│   └── hg
├── remotes
│   └── origin
│       └── HEAD
└── tags

```

9 directories, 5 files

Git-remote-hg is trying to make things more idiomatically Git-esque, but under the hood it's managing the conceptual mapping between two slightly different systems. The `refs/hg` directory is where the actual remote refs are stored. For example, the `refs/hg/origin/branches/default` is a Git ref file that contains the SHA-1 starting with "ac7955c", which is the commit that `master` points to. So the `refs/hg` directory is kind of like a fake `refs/remotes/origin`, but it has the added distinction between bookmarks and branches.

The `notes/hg` file is the starting point for how git-remote-hg maps Git commit hashes to Mercurial changeset IDs. Let's explore a bit:

```

$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f... 65bb417...
100644 blob 485e178... ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9

```

So `refs/notes/hg` points to a tree, which in the Git object database is a list of other

objects with names. `git ls-tree` outputs the mode, type, object hash, and filename for items inside a tree. Once we dig down to one of the tree items, we find that inside it is a blob named “ac9117f” (the SHA-1 hash of the commit pointed to by `master`), with contents “0a04b98” (which is the ID of the Mercurial changeset at the tip of the `default` branch).

The good news is that we mostly don’t have to worry about all of this. The typical workflow won’t be very different from working with a Git remote.

There’s one more thing we should attend to before we continue: ignores. Mercurial and Git use a very similar mechanism for this, but it’s likely you don’t want to actually commit a `.gitignore` file into a Mercurial repository. Fortunately, Git has a way to ignore files that’s local to an on-disk repository, and the Mercurial format is compatible with Git, so you just have to copy it over:

```
$ cp .hgignore .git/info/exclude
```

The `.git/info/exclude` file acts just like a `.gitignore`, but isn’t included in commits.

Workflow

Let’s assume we’ve done some work and made some commits on the `master` branch, and you’re ready to push it to the remote repository. Here’s what our repository looks like right now:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Our `master` branch is two commits ahead of `origin/master`, but those two commits exist only on our local machine. Let’s see if anyone else has been doing important work at the same time:

```

$ git fetch
From hg::/tmp/hello
   ac7955c..df85e87  master      -> origin/master
   ac7955c..df85e87  branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program

```

Since we used the `--all` flag, we see the “notes” refs that are used internally by `git-remote-hg`, but we can ignore them. The rest is what we expected; `origin/master` has advanced by one commit, and our history has now diverged. Unlike the other systems we work with in this chapter, Mercurial is capable of handling merges, so we’re not going to do anything fancy.

```

$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
* 0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program

```

Perfect. We run the tests and everything passes, so we’re ready to share our work with the rest of the team:

```

$ git push
To hg::/tmp/hello
   df85e87..0c64627  master -> master

```

That’s it! If you take a look at the Mercurial repository, you’ll see that this did what

we'd expect:

```
$ hg log -G --style compact
o 5[tip]:4,2 dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
| |
| o 4 64f27bcefc35 2014-08-14 19:27 -0700 ben
| | Update makefile
| |
| o 3:1 4256fc29598f 2014-08-14 19:27 -0700 ben
| | Goodbye
| |
@ | 2 7db0b4848b3c 2014-08-14 19:30 -0700 ben
|/ Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

The changeset numbered 2 was made by Mercurial, and the changesets numbered 3 and 4 were made by git-remote-hg, by pushing commits made with Git.

Branches and Bookmarks

Git has only one kind of branch: a reference that moves when commits are made. In Mercurial, this kind of a reference is called a “bookmark,” and it behaves in much the same way as a Git branch.

Mercurial’s concept of a “branch” is more heavyweight. The branch that a changeset is made on is recorded *with the changeset*, which means it will always be in the repository history. Here’s an example of a commit that was made on the `develop` branch:

```
$ hg log -l 1
changeset: 6:8f65e5e02793
branch:    develop
tag:      tip
user:     Ben Straub <ben@straub.cc>
date:     Thu Aug 14 20:06:38 2014 -0700
summary:  More documentation
```

Note the line that begins with “branch”. Git can’t really replicate this (and doesn’t need to; both types of branch can be represented as a Git ref), but git-remote-hg needs to understand the difference, because Mercurial cares.

Creating Mercurial bookmarks is as easy as creating Git branches. On the Git side:

```

$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
* [new branch]      featureA -> featureA

```

That's all there is to it. On the Mercurial side, it looks like this:

```

$ hg bookmarks
featureA          5:bd5ac26f11f9
$ hg log --style compact -G
@ 6[tip] 8f65e5e02793 2014-08-14 20:06 -0700 ben
| More documentation
|
o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
||
| o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | update makefile
| |
| o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | goodbye
| |
o | 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
|/ Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
  Create a standard "hello, world" program

```

Note the new `[featureA]` tag on revision 5. These act exactly like Git branches on the Git side, with one exception: you can't delete a bookmark from the Git side (this is a limitation of remote helpers).

You can work on a “heavyweight” Mercurial branch also: just put a branch in the `branches` namespace:

```

$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg::/tmp/hello
* [new branch]      branches/permanent -> branches/permanent

```

Here's what that looks like on the Mercurial side:

```
$ hg branches
permanent          7:a4529d07aad4
develop            6:8f65e5e02793
default            5:bd5ac26f11f9 (inactive)
$ hg log -G
o changeset: 7:a4529d07aad4
| branch:    permanent
| tag:       tip
| parent:    5:bd5ac26f11f9
| user:      Ben Straub <ben@straub.cc>
| date:      Thu Aug 14 20:21:09 2014 -0700
| summary:   A permanent change
|
| @ changeset: 6:8f65e5e02793
|/ branch:    develop
| user:      Ben Straub <ben@straub.cc>
| date:      Thu Aug 14 20:06:38 2014 -0700
| summary:   More documentation
|
o changeset: 5:bd5ac26f11f9
|\ bookmark:  featureA
| | parent:   4:0434aaa6b91f
| | parent:   2:f098c7f45c4f
| | user:     Ben Straub <ben@straub.cc>
| | date:     Thu Aug 14 20:02:21 2014 -0700
| | summary:  Merge remote-tracking branch 'origin/master'
[...]
```

The branch name “permanent” was recorded with the changeset marked 7.

From the Git side, working with either of these branch styles is the same: just checkout, commit, fetch, merge, pull, and push as you normally would. One thing you should know is that Mercurial doesn't support rewriting history, only adding to it. Here's what our Mercurial repository looks like after an interactive rebase and a force-push:

```

$ hg log --style compact -G
o 10[tip] 99611176cbc9 2014-08-14 20:21 -0700 ben
|   A permanent change
|
o 9 f23e12f939c3 2014-08-14 20:01 -0700 ben
|   Add some documentation
|
o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben
|   goodbye
|
| o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben
| |   A permanent change
| |
| | @ 6 8f65e5e02793 2014-08-14 20:06 -0700 ben
| | /   More documentation
| |
| | o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
| | \   Merge remote-tracking branch 'origin/master'
| | |
| | | o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | |   update makefile
| | |
+---o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| |   goodbye
| |
| o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
| /   Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
|   Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
|   Create a standard "hello, world" program

```

Changesets 8, 9, and 10 have been created and belong to the **permanent** branch, but the old changesets are still there. This can be **very** confusing for your teammates who are using Mercurial, so try to avoid it.

Mercurial Summary

Git and Mercurial are similar enough that working across the boundary is fairly painless. If you avoid changing history that's left your machine (as is generally recommended), you may not even be aware that the other end is Mercurial.

Git and Perforce

Perforce is a very popular version-control system in corporate environments. It's been around since 1995, which makes it the oldest system covered in this chapter. As such, it's designed with the constraints of its day; it assumes you're always connected to a

single central server, and only one version is kept on the local disk. To be sure, its features and constraints are well-suited to several specific problems, but there are lots of projects using Perforce where Git would actually work better.

There are two options if you'd like to mix your use of Perforce and Git. The first one we'll cover is the "Git Fusion" bridge from the makers of Perforce, which lets you expose subtrees of your Perforce depot as read-write Git repositories. The second is git-p4, a client-side bridge that lets you use Git as a Perforce client, without requiring any reconfiguration of the Perforce server.

Git Fusion

Perforce provides a product called Git Fusion (available at <http://www.perforce.com/git-fusion>), which synchronizes a Perforce server with Git repositories on the server side.

Setting Up

For our examples, we'll be using the easiest installation method for Git Fusion, which is downloading a virtual machine that runs the Perforce daemon and Git Fusion. You can get the virtual machine image from <http://www.perforce.com/downloads/Perforce/20-User>, and once it's finished downloading, import it into your favorite virtualization software (we'll use VirtualBox).

Upon first starting the machine, it asks you to customize the password for three Linux users (`root`, `perforce`, and `git`), and provide an instance name, which can be used to distinguish this installation from others on the same network. When that has all completed, you'll see this:

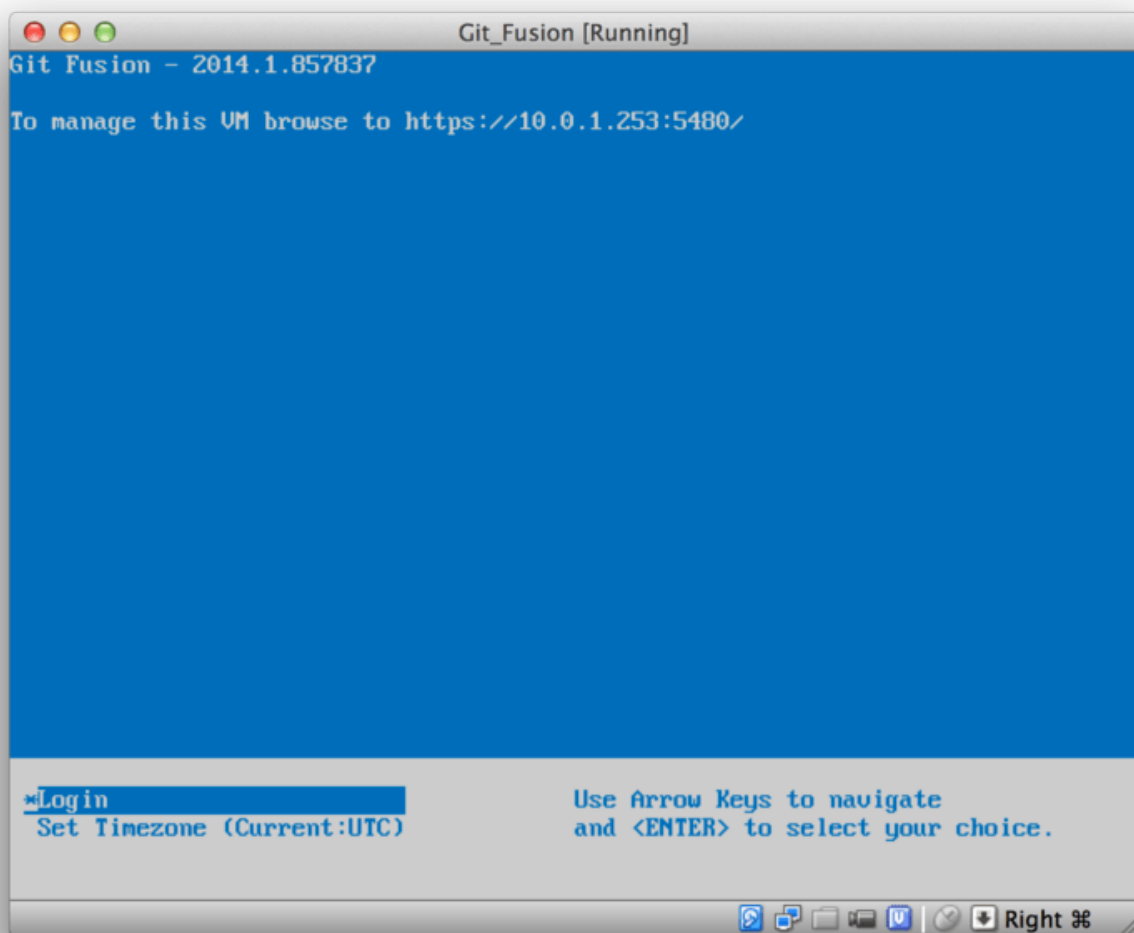


Figure 146. The Git Fusion virtual machine boot screen.

You should take note of the IP address that's shown here, we'll be using it later on. Next, we'll create a Perforce user. Select the "Login" option at the bottom and press enter (or SSH to the machine), and log in as `root`. Then use these commands to create a user:

```
$ p4 -p localhost:1666 -u super user -f john
$ p4 -p localhost:1666 -u john passwd
$ exit
```

The first one will open a VI editor to customize the user, but you can accept the defaults by typing `:wq` and hitting enter. The second one will prompt you to enter a password twice. That's all we need to do with a shell prompt, so exit out of the session.

The next thing you'll need to do to follow along is to tell Git not to verify SSL certificates. The Git Fusion image comes with a certificate, but it's for a domain that won't match your virtual machine's IP address, so Git will reject the HTTPS connection. If this is going to be a permanent installation, consult the Perforce Git

Fusion manual to install a different certificate; for our example purposes, this will suffice:

```
$ export GIT_SSL_NO_VERIFY=true
```

Now we can test that everything is working.

```
$ git clone https://10.0.1.254/Talkhouse
Cloning into 'Talkhouse'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
remote: Counting objects: 630, done.
remote: Compressing objects: 100% (581/581), done.
remote: Total 630 (delta 172), reused 0 (delta 0)
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.
Resolving deltas: 100% (172/172), done.
Checking connectivity... done.
```

The virtual-machine image comes equipped with a sample project that you can clone. Here we're cloning over HTTPS, with the `john` user that we created above; Git asks for credentials for this connection, but the credential cache will allow us to skip this step for any subsequent requests.

Fusion Configuration

Once you've got Git Fusion installed, you'll want to tweak the configuration. This is actually fairly easy to do using your favorite Perforce client; just map the `/.git-fusion` directory on the Perforce server into your workspace. The file structure looks like this:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
├── p4gf_config
├── repos
│   └── Talkhouse
│       └── p4gf_config
├── users
│   └── p4gf_usermap
└──
```

498 directories, 287 files

The `objects` directory is used internally by Git Fusion to map Perforce objects to

Git and vice versa, you won't have to mess with anything in there. There's a global `p4gf_config` file in this directory, as well as one for each repository – these are the configuration files that determine how Git Fusion behaves. Let's take a look at the file in the root:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

We won't go into the meanings of these flags here, but note that this is just an INI-formatted text file, much like Git uses for configuration. This file specifies the global options, which can then be overridden by repository-specific configuration files, like `repos/Talkhouse/p4gf_config`. If you open this file, you'll see a `[@repo]` section with some settings that are different from the global defaults. You'll also see sections that look like this:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ..
```

This is a mapping between a Perforce branch and a Git branch. The section can be named whatever you like, so long as the name is unique. `git-branch-name` lets you convert a depot path that would be cumbersome under Git to a more friendly name. The `view` setting controls how Perforce files are mapped into the Git repository, using the standard view mapping syntax. More than one mapping can be

specified, like in this example:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

This way, if your normal workspace mapping includes changes in the structure of the directories, you can replicate that with a Git repository.

The last file we'll discuss is `users/p4gf_usermap`, which maps Perforce users to Git users, and which you may not even need. When converting from a Perforce changeset to a Git commit, Git Fusion's default behavior is to look up the Perforce user, and use the email address and full name stored there for the author/commmitter field in Git. When converting the other way, the default is to look up the Perforce user with the email address stored in the Git commit's author field, and submit the changeset as that user (with permissions applying). In most cases, this behavior will do just fine, but consider the following mapping file:

```
john john@example.com "John Doe"
john johnny@appleseed.net "John Doe"
bob employeeX@example.com "Anon X. Mouse"
joe employeeY@example.com "Anon Y. Mouse"
```

Each line is of the format `<user> <email> "<full name>"`, and creates a single user mapping. The first two lines map two distinct email addresses to the same Perforce user account. This is useful if you've created Git commits under several different email addresses (or change email addresses), but want them to be mapped to the same Perforce user. When creating a Git commit from a Perforce changeset, the first line matching the Perforce user is used for Git authorship information.

The last two lines mask Bob and Joe's actual names and email addresses from the Git commits that are created. This is nice if you want to open-source an internal project, but don't want to publish your employee directory to the entire world. Note that the email addresses and full names should be unique, unless you want all the Git commits to be attributed to a single fictional author.

Workflow

Perforce Git Fusion is a two-way bridge between Perforce and Git version control. Let's have a look at how it feels to work from the Git side. We'll assume we've mapped in the "Jam" project using a configuration file as shown above, which we can clone like this:

```

$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on
Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

The first time you do this, it may take some time. What's happening is that Git Fusion is converting all the applicable changesets in the Perforce history into Git commits. This happens locally on the server, so it's relatively fast, but if you have a lot of history, it can still take some time. Subsequent fetches do incremental conversion, so it'll feel more like Git's native speed.

As you can see, our repository looks exactly like any other Git repository you might work with. There are three branches, and Git has helpfully created a local **master** branch that tracks **origin/master**. Let's do a bit of work, and create a couple of new commits:

```

# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the
Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

We have two new commits. Now let's check if anyone else has been working:

```

$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
   d254865..6afeb15  master    -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]

```

It looks like someone was! You wouldn't know it from this view, but the `6afeb15` commit was actually created using a Perforce client. It just looks like another commit from Git's point of view, which is exactly the point. Let's see how the Perforce server deals with a merge commit:

```

$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
   6afeb15..89cba2b  master -> master

```

Git thinks it worked. Let's take a look at the history of the `README` file from Perforce's point of view, using the revision graph feature of `p4v`:

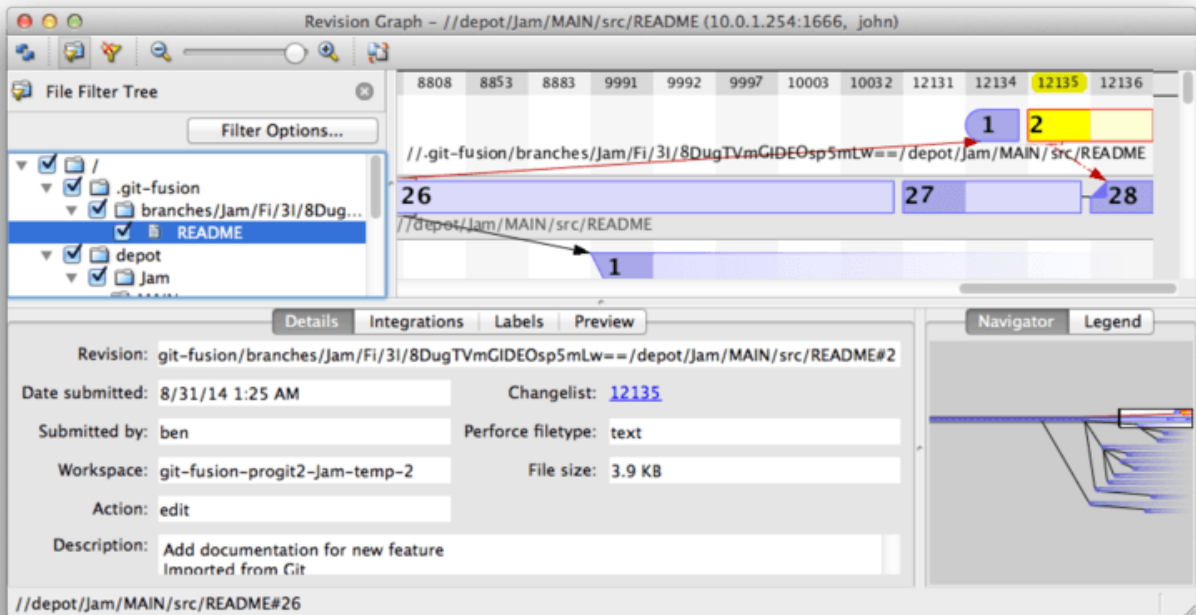


Figure 147. Perforce revision graph resulting from Git push.

If you've never seen this view before, it may seem confusing, but it shows the same concepts as a graphical viewer for Git history. We're looking at the history of the `README` file, so the directory tree at top left only shows that file as it surfaces in various branches. At top right, we have a visual graph of how different revisions of the file are related, and the big-picture view of this graph is at bottom right. The rest of the view is given to the details view for the selected revision (2 in this case).

One thing to notice is that the graph looks exactly like the one in Git's history. Perforce didn't have a named branch to store the 1 and 2 commits, so it made an "anonymous" branch in the `.git-fusion` directory to hold it. This will also happen for named Git branches that don't correspond to a named Perforce branch (and you can later map them to a Perforce branch using the configuration file).

Most of this happens behind the scenes, but the end result is that one person on a team can be using Git, another can be using Perforce, and neither of them will know about the other's choice.

Git-Fusion Summary

If you have (or can get) access to your Perforce server, Git Fusion is a great way to make Git and Perforce talk to each other. There's a bit of configuration involved, but the learning curve isn't very steep. This is one of the few sections in this chapter where cautions about using Git's full power will not appear. That's not to say that Perforce will be happy with everything you throw at it – if you try to rewrite history that's already been pushed, Git Fusion will reject it – but Git Fusion tries very hard to feel native. You can even use Git submodules (though they'll look strange to Perforce users), and merge branches (this will be recorded as an integration on the Perforce side).

If you can't convince the administrator of your server to set up Git Fusion, there is still a way to use these tools together.

Git-p4

Git-p4 is a two-way bridge between Git and Perforce. It runs entirely inside your Git repository, so you won't need any kind of access to the Perforce server (other than user credentials, of course). Git-p4 isn't as flexible or complete a solution as Git Fusion, but it does allow you to do most of what you'd want to do without being invasive to the server environment.

NOTE

You'll need the `p4` tool somewhere in your `PATH` to work with `git-p4`. As of this writing, it is freely available at <http://www.perforce.com/downloads/Perforce/20-User>.

Setting Up

For example purposes, we'll be running the Perforce server from the Git Fusion OVA as shown above, but we'll bypass the Git Fusion server and go directly to the Perforce version control.

In order to use the `p4` command-line client (which `git-p4` depends on), you'll need to set a couple of environment variables:

```
$ export P4PORT=10.0.1.254:1666
$ export P4USER=john
```

Getting Started

As with anything in Git, the first command is to clone:

```
$ git p4 clone //depot/www/live www-shallow
Importing from //depot/www/live into www-shallow
Initialized empty Git repository in /private/tmp/www-shallow/.git/
Doing initial import of //depot/www/live/ from revision #head into
refs/remotes/p4/master
```

This creates what in Git terms is a "shallow" clone; only the very latest Perforce revision is imported into Git; remember, Perforce isn't designed to give every revision to every user. This is enough to use Git as a Perforce client, but for other purposes it's not enough.

Once it's finished, we have a fully-functional Git repository:


```
$ cd myproject
$ git log --oneline --all --graph --decorate
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from
the state at revision #head
```

Note how there's a "p4" remote for the Perforce server, but everything else looks like a standard clone. Actually, that's a bit misleading; there isn't actually a remote there.

```
$ git remote -v
```

No remotes exist in this repository at all. Git-p4 has created some refs to represent the state of the server, and they look like remote refs to `git log`, but they're not managed by Git itself, and you can't push to them.

Workflow

Okay, let's do some work. Let's assume you've made some progress on a very important feature, and you're ready to show it to the rest of your team.

```
$ git log --oneline --all --graph --decorate
* 018467c (HEAD, master) Change page title
* c0fb617 Update link
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at
revision #head
```

We've made two new commits that we're ready to submit to the Perforce server. Let's check if anyone else was working today:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Looks like they were, and `master` and `p4/master` have diverged. Perforce's branching system is *nothing* like Git's, so submitting merge commits doesn't make any sense. Git-p4 recommends that you rebase your commits, and even comes with a shortcut to do so:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
index.html | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

You can probably tell from the output, but `git p4 rebase` is a shortcut for `git p4 sync` followed by `git rebase p4/master`. It's a bit smarter than that, especially when working with multiple branches, but this is a good approximation.

Now our history is linear again, and we're ready to contribute our changes back to Perforce. The `git p4 submit` command will try to create a new Perforce revision for every Git commit between `p4/master` and `master`. Running it drops us into our favorite editor, and the contents of the file look something like this:

```
# A Perforce Change Specification.
#
# Change:      The change number. 'new' on a new changelist.
# Date:       The date this specification was last modified.
# Client:     The client on which the changelist was created.  Read-only.
# User:      The user who created the changelist.
# Status:    Either 'pending' or 'submitted'.  Read-only.
# Type:      Either 'public' or 'restricted'.  Default is 'public'.
# Description: Comments about the changelist.  Required.
# Jobs:      What opened jobs are to be closed by this changelist.
#            You may delete jobs from this list.  (New changelists only.)
# Files:     What opened files from the default changelist are to be added
#            to this changelist.  You may delete files from this list.
#            (New changelists only.)
```

Change: new

Client: john_bens-mbp_8487

User: john

Status: new

Description:
Update link

Files:
//depot/www/live/index.html # edit

```
##### git author ben@straub.cc does not match your p4 account.
##### Use option --preserve-user to modify authorship.
##### Variable git-p4.skipUserNameCheck hides this message.
##### everything below this line is just the diff #####
--- //depot/www/live/index.html 2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html 2014-
08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
</td>
<td valign=top>
Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
Jam/MR</a>,
a software build tool.
</td>
```

This is mostly the same content you'd see by running `p4 submit`, except the stuff at the end which git-p4 has helpfully included. Git-p4 tries to honor your Git and Perforce

settings individually when it has to provide a name for a commit or changeset, but in some cases you want to override it. For example, if the Git commit you're importing was written by a contributor who doesn't have a Perforce user account, you may still want the resulting changeset to look like they wrote it (and not you).

Git-p4 has helpfully imported the message from the Git commit as the content for this Perforce changeset, so all we have to do is save and quit, twice (once for each commit). The resulting shell output will look something like this:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

The result is as though we just did a `git push`, which is the closest analogy to what actually did happen.

Note that during this process every Git commit is turned into a Perforce changeset; if you want to squash them down into a single changeset, you can do that with an interactive rebase before running `git p4 submit`. Also note that the SHA-1 hashes of all the commits that were submitted as changesets have changed; this is because git-p4 adds a line to the end of each commit it converts:

```

$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date: Sun Aug 31 10:31:44 2014 -0800

    Change page title

[git-p4: depot-paths = "//depot/www/live/": change = 12144]

```

What happens if you try to submit a merge commit? Let's give it a try. Here's the situation we've gotten ourselves into:

```

$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
* 1dcbf21 Merge remote-tracking branch 'p4/master'
|\
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head

```

The Git and Perforce history diverge after 775a46f. The Git side has two commits, then a merge commit with the Perforce head, then another commit. We're going to try to submit these on top of a single changeset on the Perforce side. Let's see what would happen if we tried to submit now:

```

$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
    b4959b6 Trademark
    cbacd0a Table borders: yes please
    3be6fd8 Correct email address

```

The `-n` flag is short for `--dry-run`, which tries to report what would happen if the submit command were run for real. In this case, it looks like we'd be creating three Perforce changesets, which correspond to the three non-merge commits that don't yet exist on the Perforce server. That sounds like exactly what we want, let's see how it turns out:

```
$ git p4 submit
[...]
```

```
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Our history became linear, just as though we had rebased before submitting (which is in fact exactly what happened). This means you can be free to create, work on, throw away, and merge branches on the Git side without fear that your history will somehow become incompatible with Perforce. If you can rebase it, you can contribute it to a Perforce server.

Branching

If your Perforce project has multiple branches, you're not out of luck; git-p4 can handle that in a way that makes it feel like Git. Let's say your Perforce depot is laid out like this:

```
//depot
├── project
│   ├── main
│   └── dev
```

And let's say you have a `dev` branch, which has a view spec that looks like this:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 can automatically detect that situation and do the right thing:

```

$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
  Importing new branch project/dev

  Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init

```

Note the “@all” specifier in the depot path; that tells git-p4 to clone not just the latest changeset for that subtree, but all changesets that have ever touched those paths. This is closer to Git’s concept of a clone, but if you’re working on a project with a long history, it could take a while.

The `--detect-branches` flag tells git-p4 to use Perforce’s branch specs to map the branches to Git refs. If these mappings aren’t present on the Perforce server (which is a perfectly valid way to use Perforce), you can tell git-p4 what the branch mappings are, and you get the same result:

```

$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .

```

Setting the `git-p4.branchList` configuration variable to `main:dev` tells git-p4 that “main” and “dev” are both branches, and the second one is a child of the first one.

If we now `git checkout -b dev p4/project/dev` and make some commits, git-p4 is smart enough to target the right branch when we do `git p4 submit`. Unfortunately, git-p4 can’t mix shallow clones and multiple branches; if you have a huge project and want to work on more than one branch, you’ll have to `git p4 clone` once for each branch you want to submit to.

For creating or integrating branches, you’ll have to use a Perforce client. Git-p4 can only sync and submit to existing branches, and it can only do it one linear changeset at a time. If you merge two branches in Git and try to submit the new changeset, all that will be recorded is a bunch of file changes; the metadata about which branches are involved in the integration will be lost.

Git and Perforce Summary

Git-p4 makes it possible to use a Git workflow with a Perforce server, and it's pretty good at it. However, it's important to remember that Perforce is in charge of the source, and you're only using Git to work locally. Just be really careful about sharing Git commits; if you have a remote that other people use, don't push any commits that haven't already been submitted to the Perforce server.

If you want to freely mix the use of Perforce and Git as clients for source control, and you can convince the server administrator to install it, Git Fusion makes using Git a first-class version-control client for a Perforce server.

Git and TFS

Git is becoming popular with Windows developers, and if you're writing code on Windows, there's a good chance you're using Microsoft's Team Foundation Server (TFS). TFS is a collaboration suite that includes defect and work-item tracking, process support for Scrum and others, code review, and version control. There's a bit of confusion ahead: **TFS** is the server, which supports controlling source code using both Git and their own custom VCS, which they've dubbed **TFVC** (Team Foundation Version Control). Git support is a somewhat new feature for TFS (shipping with the 2013 version), so all of the tools that predate that refer to the version-control portion as "TFS", even though they're mostly working with TFVC.

If you find yourself on a team that's using TFVC but you'd rather use Git as your version-control client, there's a project for you.

Which Tool

In fact, there are two: git-tf and git-tfs.

Git-tfs (found at <https://github.com/git-tfs/git-tfs>) is a .NET project, and (as of this writing) it only runs on Windows. To work with Git repositories, it uses the .NET bindings for libgit2, a library-oriented implementation of Git which is highly performant and allows a lot of flexibility with the guts of a Git repository. Libgit2 is not a complete implementation of Git, so to cover the difference git-tfs will actually call the command-line Git client for some operations, so there are no artificial limits on what it can do with Git repositories. Its support of TFVC features is very mature, since it uses the Visual Studio assemblies for operations with servers. This does mean you'll need access to those assemblies, which means you need to install a recent version of Visual Studio (any edition since version 2010, including Express since version 2012), or the Visual Studio SDK.

Git-tf (whose home is at <https://gittf.codeplex.com>) is a Java project, and as such runs on any computer with a Java runtime environment. It interfaces with Git repositories through JGit (a JVM implementation of Git), which means it has virtually no limitations in terms of Git functions. However, its support for TFVC is limited as compared to git-tfs – it does not support branches, for instance.

So each tool has pros and cons, and there are plenty of situations that favor one over

the other. We'll cover the basic usage of both of them in this book.

NOTE

You'll need access to a TFVC-based repository to follow along with these instructions. These aren't as plentiful in the wild as Git or Subversion repositories, so you may need to create one of your own. Codeplex (<https://www.codeplex.com>) or Visual Studio Online (<http://www.visualstudio.com>) are both good choices for this.

Getting Started: `git-tf`

The first thing you do, just as with any Git project, is clone. Here's what that looks like with `git-tf`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

The first argument is the URL of a TFVC collection, the second is of the form `$/project/branch`, and the third is the path to the local Git repository that is to be created (this last one is optional). `Git-tf` can only work with one branch at a time; if you want to make checkins on a different TFVC branch, you'll have to make a new clone from that branch.

This creates a fully functional Git repository:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

This is called a *shallow* clone, meaning that only the latest changeset has been downloaded. TFVC isn't designed for each client to have a full copy of the history, so `git-tf` defaults to only getting the latest version, which is much faster.

If you have some time, it's probably worth it to clone the entire project history, using the `--deep` option:

```

$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard

```

Notice the tags with names like `TFS_C35189`; this is a feature that helps you know which Git commits are associated with TFVC changesets. This is a nice way to represent it, since you can see with a simple log command which of your commits is associated with a snapshot that also exists in TFVC. They aren't necessary (and in fact you can turn them off with `git config git-tf.tag false`) – git-tf keeps the real commit-changeset mappings in the `.git/git-tf` file.

Getting Started: `git-tfs`

Git-tfs cloning behaves a bit differently. Observe:

```

PS> git tfs clone --with-branches \
  https://username.visualstudio.com/DefaultCollection \
  $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeefb961958b674

```

Notice the `--with-branches` flag. Git-tfs is capable of mapping TFVC branches to Git branches, and this flag tells it to set up a local Git branch for every TFVC branch. This is highly recommended if you've ever branched or merged in TFS, but it won't work with a server older than TFS 2010 – before that release, “branches” were just folders, so git-tfs can't tell them from regular folders.

Let's take a look at the resulting Git repository:

```
PS> git log --oneline --graph --decorate --all
* 44cd729 (tfs/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfs/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date: Fri Aug 1 03:41:59 2014 +0000
```

Hello

```
git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]/myproject/Trunk;C16
```

There are two local branches, `master` and `featureA`, which represent the initial starting point of the clone (`Trunk` in TFVC) and a child branch (`featureA` in TFVC). You can also see that the `tfs` “remote” has a couple of refs too: `default` and `featureA`, which represent TFVC branches. Git-tfs maps the branch you cloned from to `tfs/default`, and others get their own names.

Another thing to notice is the `git-tfs-id:` lines in the commit messages. Instead of tags, git-tfs uses these markers to relate TFVC changesets to Git commits. This has the implication that your Git commits will have a different SHA-1 hash before and after they have been pushed to TFVC.

Git-tf[s] Workflow

Regardless of which tool you’re using, you should set a couple of Git configuration values to avoid running into issues.

NOTE

```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

The obvious next thing you’re going to want to do is work on the project. TFVC and TFS have several features that may add complexity to your workflow:

1. Feature branches that aren’t represented in TFVC add a bit of complexity. This has to do with the **very** different ways that TFVC and Git represent branches.
2. Be aware that TFVC allows users to “checkout” files from the server, locking them so nobody else can edit them. This obviously won’t stop you from editing them in your local repository, but it could get in the way when it comes time to push your changes up to the TFVC server.
3. TFS has the concept of “gated” checkins, where a TFS build-test cycle has to complete successfully before the checkin is allowed. This uses the “shelve” function in TFVC, which we don’t cover in detail here. You can fake this in a manual fashion

with git-tf, and git-tfs provides the `checkintool` command which is gate-aware.

In the interest of brevity, what we'll cover here is the happy path, which sidesteps or avoids most of these issues.

Workflow: `git-tf`

Let's say you've done some work, made a couple of Git commits on `master`, and you're ready to share your progress on the TFVC server. Here's our Git repository:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

We want to take the snapshot that's in the `4178a82` commit and push it up to the TFVC server. First things first: let's see if any of our teammates did anything since we last connected:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Looks like someone else is working, too, and now we have divergent history. This is where Git shines, but we have two choices of how to proceed:

1. Making a merge commit feels natural as a Git user (after all, that's what `git pull` does), and `git-tf` can do this for you with a simple `git tf pull`. Be aware, however, that TFVC doesn't think this way, and if you push merge commits your history will start to look different on both sides, which can be confusing. However, if you plan on submitting all of your changes as one changeset, this is probably the

easiest choice.

2. Rebasing makes our commit history linear, which means we have the option of converting each of our Git commits into a TFVC changeset. Since this leaves the most options open, we recommend you do it this way; git-tf even makes it easy for you with `git tf pull --rebase`.

The choice is yours. For this example, we'll be rebasing:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Now we're ready to make a checkin to the TFVC server. Git-tf gives you the choice of making a single changeset that represents all the changes since the last one (`--shallow`, which is the default) and creating a new changeset for each Git commit (`--deep`). For this example, we'll just create one changeset:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

There's a new `TFS_C35348` tag, indicating that TFVC is storing the exact same snapshot as the `5a0e25e` commit. It's important to note that not every Git commit needs to have an exact counterpart in TFVC; the `6eb3eb5` commit, for example, doesn't exist anywhere on the server.

That's the main workflow. There are a couple of other considerations you'll want to keep in mind:

- There is no branching. Git-tf can only create Git repositories from one TFVC branch at a time.
- Collaborate using either TFVC or Git, but not both. Different git-tf clones of the same TFVC repository may have different commit SHA-1 hashes, which will cause no end of headaches.
- If your team's workflow includes collaborating in Git and syncing periodically with TFVC, only connect to TFVC with one of the Git repositories.

Workflow: `git-tfs`

Let's walk through the same scenario using `git-tfs`. Here are the new commits we've made to the `master` branch in our Git repository:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 (tfs/default) Hello
* b75da1a New project
```

Now let's see if anyone else has done work while we were hacking away:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfs/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|/
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Yes, it turns out our coworker has added a new TFVC changeset, which shows up as the new `aea74a0` commit, and the `tfs/default` remote branch has moved.

As with `git-tf`, we have two fundamental options for how to resolve this divergent history:

1. `Rebase to preserve a linear history.`

2. Merge to preserve what actually happened.

In this case, we're going to do a "deep" checkin, where every Git commit becomes a TFVC changeset, so we want to rebase.

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Now we're ready to complete our contribution by checking in our code to the TFVC server. We'll use the `rcheckin` command here to create a TFVC changeset for each Git commit in the path from HEAD to the first `tfs` remote branch found (the `checkin` command would only create one changeset, sort of like squashing Git commits).

```

PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
  add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-
commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
  edit .git\tfs\default\workspace\ConsoleApplication1\ConsoleApplication1/Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-
commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project

```

Notice how after every successful checkin to the TFVC server, git-tfs is rebasing the remaining work onto what it just did. That's because it's adding the `git-tfs-id` field to the bottom of the commit messages, which changes the SHA-1 hashes. This is exactly as designed, and there's nothing to worry about, but you should be aware that it's happening, especially if you're sharing Git commits with others.

TFS has many features that integrate with its version control system, such as work items, designated reviewers, gated checkins, and so on. It can be cumbersome to work with these features using only a command-line tool, but fortunately git-tfs lets you launch a graphical checkin tool very easily:

```

PS> git tfs checkintool
PS> git tfs ct

```

It looks a bit like this:

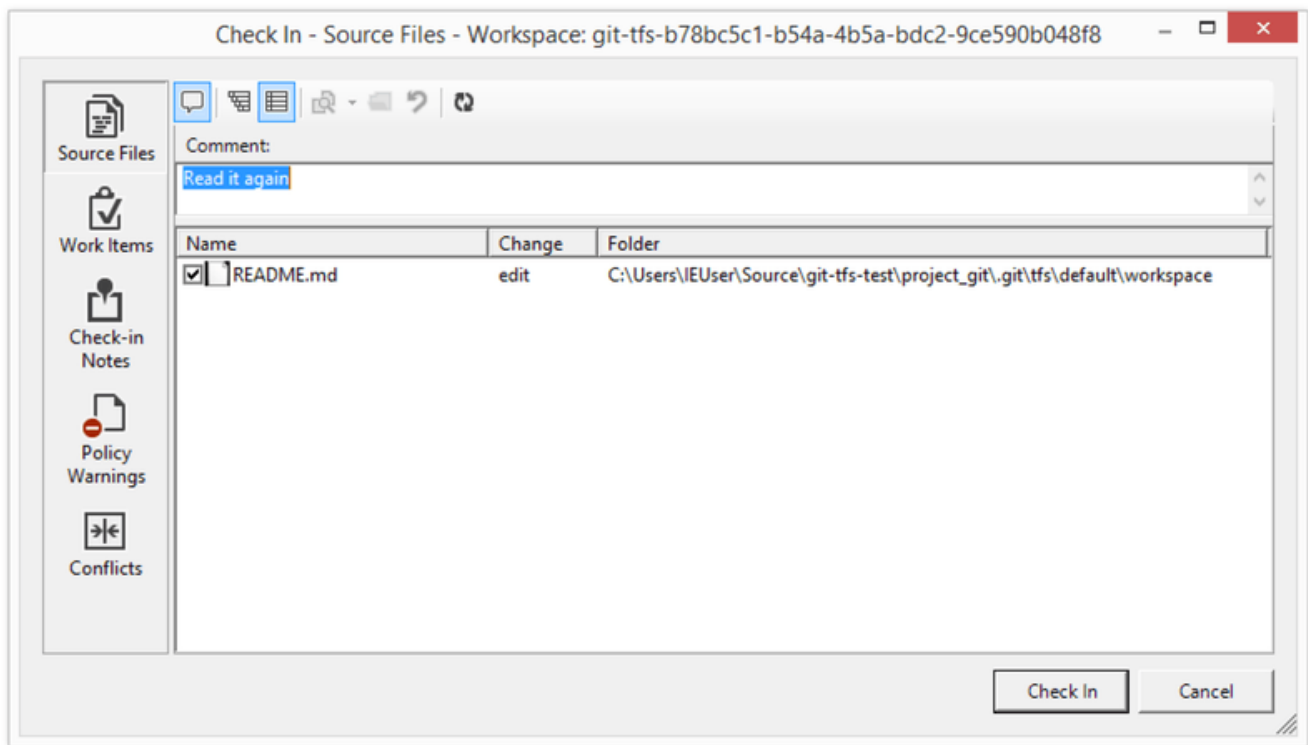


Figure 148. The git-tfs checkin tool.

This will look familiar to TFS users, as it's the same dialog that's launched from within Visual Studio.

Git-tfs also lets you control TFVC branches from your Git repository. As an example, let's create one:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfs/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Creating a branch in TFVC means adding a changeset where that branch now exists, and this is projected as a Git commit. Note also that git-tfs **created** the `tfs/featureBee` remote branch, but `HEAD` is still pointing to `master`. If you want to work on the newly-minted branch, you'll want to base your new commits on the `1d54865` commit, perhaps by creating a topic branch from that commit.

Git and TFS Summary

Git-tf and Git-tfs are both great tools for interfacing with a TFVC server. They allow you to use the power of Git locally, avoid constantly having to round-trip to the central TFVC server, and make your life as a developer much easier, without forcing your entire team to migrate to Git. If you're working on Windows (which is likely if your team is using TFS), you'll probably want to use git-tfs, since its feature set is more complete, but if you're working on another platform, you'll be using git-tf, which is more limited. As with most of the tools in this chapter, you should choose one of these version-control systems to be canonical, and use the other one in a subordinate fashion – either Git or TFVC should be the center of collaboration, but not both.

Migracija na Git

Če imate obstoječo bazo kode v drugem VCS-ju, vendar ste se odločili začetni uporabljati Git, morate migrirati vaš projekt na en ali drug način. Ta sekcija gre skozi nekaj uvoznikov za pogoste sisteme in nato demonstrira, kako razvijati vašega lastnega uvoznika. Naučili se boste, kako uvažati podatke iz nekaj največjih profesionalnih uporabljenih SCM sistemov, ker delajo glavnino uporabnikov, ki preklaplajo in ker visoko kvalitetna orodja zanje so enostavna za dobiti.

Subversion

Če ste prebrali prejšnjo sekcijo o uporabi `git svn` lahko enostavno uporabite ta navodila za `git svn clone` repozitorija; nato prenehajte uporabljati strežnik Subversion, potisnite na novi strežnik Git in ga začnete uporabljati. Če želite zgodovino, lahko to dosežete kakor hitro lahko potegnete podatke iz strežnika Subversion (kar lahko vzame nekaj časa).

Vendar import ni popoln; in ker bo vzel nekaj časa, lahko tudi naredite, kakor je prav. Prvi problem so informacije avtorja. V Subversion-u, vsaka oseba, ki pošilja, ima uporabnika na sistemu, ki je posnet v informacija pošiljanja. Primeri v prejšnji sekciji prikazani `schacon` na nekaterih mestih, kot je izpis `blame` in `git svn log`. Če želite preslikati to na boljše podatke Git avtorja, morate preslika iz Subversion uporabnikov na avtorje Git. Ustvarite datoteko imenovano `users.txt`, ki ima to preslikavo v sledeči obliki:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Da dobite seznam imen avtorja, ki jih uporablja SVN, lahko pošete to:

```
$ svn log --xml | grep author | sort -u | \
perl -pe 's/.*>(.*?)<.*$/1 = /'
```

To generira izpis dnevnika v XML formatu, nato obdrži samo vrstice z informacijami

avtorja, opusti duplikate, izpusti značke XML. (Očitno to deluje samo na napravi z nameščenimi `grep`, `sort` in `perl`.) Nato preusmerite ta izpis v vašo datoteko `users.txt`, da lahko dodate ekvivalentne podatke Git uporabnika zraven vsakega vnosa.

To datoteko lahko ponudite `git svn`, da pomaga preslikati podatke avtorja bolj točno. Poveste lahko tudi, da `git svn` ne vključuje meta podatkov, ki jih Subversion običajno uvažava s podajanjem `--no-metadata` k ukazoma `clone` ali `init`. To naredi vaš ukaz `import`, da izgleda sledeče:

```
$ git svn clone http://my-project.googlecode.com/svn/ \  
  --authors-file=users.txt --no-metadata -s my_project
```

Sedaj morate imeti lepši uvoz Subversion-a v vaš direktorij `my_project`. Namesto pošiljanj, ki izgledajo takole

```
commit 37efa680e8473b615de980fa935944215428a35a  
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>  
Date: Sun May 3 00:12:22 2009 +0000  
  
    fixed install - go to trunk  
  
git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-  
be05-5f7a86268029
```

izgledajo takole:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2  
Author: Scott Chacon <schacon@geemail.com>  
Date: Sun May 3 00:12:22 2009 +0000  
  
    fixed install - go to trunk
```

Ne samo, da polje `Author` izgleda veliko boljše, ampak tudi `git-svn-id` ni več tam.

Sedaj bi morali narediti tudi nekaj post-import čiščenja. Za eno stvar, bi morali počistiti čudne reference, ki jih je nastavil `git svn`. Najprej boste premaknili oznake, da so dejansko oznake namesto čudnih oddaljenih vej in nato boste premaknili preostanek vej, da so lokalne.

Da premaknete oznake, da so ustrezne Git oznake, poženite

```
$ cp -Rf .git/refs/remotes/origin/tags/* .git/refs/tags/  
$ rm -Rf .git/refs/remotes/origin/tags
```

To vzame reference, ki so oddaljene veje in se začnejo z `remotes/origin/tags/` in jih naredi realne (lightweight) oznake.

Naslednje premaknite preostanek referenc pod `refs/remotes`, da so lokalne veje:

```
$ cp -Rf .git/refs/remotes/* .git/refs/heads/  
$ rm -Rf .git/refs/remotes
```

Sedaj so vse stare veje prave Git veje in vse stare oznake so prave Git oznake. Zadnja stvar za narediti je dodati vaš novi strežnik Git kot daljavo in potisniti nanj. Tu je primer dodajanja vašega strežnika kot daljavo:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Ker želite vse vaše veje in oznake dodati gor, lahko sedaj pošenetete to:

```
$ git push origin --all
```

Vse vaše veje in oznake bi morale biti na vašem novem Git strežniku z lepim, čistim uvozom.

Mercurial

Ker imate Mercurial in Git precej podobna modela za predstavitev verzij in ker je Git nekoliko bolj fleksibilen, je pretvorba repozitorija iz Mercurial na Git precej enostavna z uporabo orodja imenovanega "hg-fast-export", ki ga boste potrebovali kopirati:

```
$ git clone http://repo.or.cz/r/fast-export.git /tmp/fast-export
```

Prvi korak je pretvorba dobiti polni klon repozitorija Mercurial, ki ga želite pretvoriti:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

Naslednji korak je ustvariti datoteko preslikave avtorja. Mercurial je nekoliko bolj odpustljiv kot Git zaradi česar bo dal polje avtorja za skupke sprememb, torej je to dober čas za počistiti hišo. Generiranje tega je ukaz ene vrstice v lupini `bash`:

```
$ cd /tmp/hg-repo  
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

To bo vzelo nekaj sekund, odvisno od tega kako dolga je zgodovina vašega projekta in potem bo datoteka `/tmp/authors` izgledala nekako takole:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

V tem primeru je ista oseba (Bob) ustvarila skupek sprememb pod štirimi različnimi imeni, eno izmed njih dejansko izgleda v redu in eno od njih bi bilo popolnoma neveljavno za pošiljanje Git-a. Hg-fast-export vam omogoča to popraviti z dodajanjem ``={novo ime in e-pošta}` na koncu vsake vrstice, ki jo želimo spremeniti in odstraniti vrstice za katerokoli uporabniško ime, ki ga želimo pustiti pri miru. Če vsa uporabniška imena izgledajo v redu te datoteke sploh ne bomo potrebovali. V tem primeru želimo, da naša datoteka izgleda takole:

```
bob=Bob Jones <bob@company.com>
bob@localhost=Bob Jones <bob@company.com>
bob jones <bob <AT> company <DOT> com>=Bob Jones <bob@company.com>
bob <bob@company.com>=Bob Jones <bob@company.com>
```

Naslednji korak je ustvariti naš novi repozitorij Git in pognati izvozno skripto:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

Zastavica `-r` pove hg-fast-export, kje najti repozitorij Mercurial, ki ga želimo pretvoriti in zastavica `-A` mu pove, kje najti datoteko author-mapping. Skripta prevede skupke sprememb Mercurial-a in jih pretvori v skripto za Git-ovo lastnost "fast-import" (o kateri bomo govorili v podrobnosti nekoliko kasneje). To vzame nekaj (čeprav je *veliko* hitreje kot bi bilo preko omrežja) in izpis je precej opisen:

```

$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[...]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed
files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed
files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[...]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
-----
Alloc'd objects:      120000
Total objects:       115032 (    208171 duplicates          )
   blobs  :          40504 (    205320 duplicates      26117 deltas of    39602
attempts)
   trees  :          52320 (     2851 duplicates      47467 deltas of    47599
attempts)
   commits:          22208 (         0 duplicates         0 deltas of         0
attempts)
   tags   :              0 (         0 duplicates         0 deltas of         0
attempts)
Total branches:       109 (         2 loads          )
   marks:      1048576 (    22208 unique          )
   atoms:           1952
Memory total:         7860 KiB
   pools:           2235 KiB
   objects:          5625 KiB
-----
pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 90430
pack_report: pack_mmap_calls = 46771
pack_report: pack_open_windows = 1 / 1
pack_report: pack_mapped = 340852700 / 340852700
-----

$ git shortlog -sn
 369 Bob Jones
 365 Joe Smith

```

To je večinoma vse, kar je. Vse oznake Mercurial-a so bile pretvorjene v oznake Git in veje Mercurial in zaznamki so bili pretvorjeni v veje Git. Sedaj ste pripravljeni potisniti repozitorij na njegov novi strežniški dom:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

Perforce

Naslednji sistem, ki ga boste pogledali pri uvažanju je Perforce. Kot smo govorili zgoraj, sta dva načina, da omogočimo Git-u in Perforce-u govoriti drug z drugim: git-p4 in Perforce Git Fusion.

Perforce Git Fusion

Git Fusion naredi ta proces precej neboleč. Samo nastavite nastavitve vašega projekta, preslikave uporabnika in veje, ki uporabljajo nastavitveno datoteko (kot je povedano v [Git Fusion](#) in klonirajte repozitorij. Git Fusion vas pusti z nečim, kar izgleda kot materni repozitorij Git, ki je nato pripravljen za potiskanje na materni gostitelj Gi, če želite. Lahko bi celo uporabili Perforce kot vašega gostitelja Git-a, če želite.

Git-p4

Git-p4 se lahko obnaša tudi kot uvozno orodje. Kot primer, bomo uvozili projekt Jam iz Perforce javnega depot-a. Da nastavite vašega klienta, morate izvoziti okoljsko spremenljivko P4PORT, da kaže na Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```

NOTE

Da zrave sledite, boste potrebovali Perforce depot za povezavo. Uporabljali bomo javni depot na public.perforce.com za naš primer, vendar lahko uporabite katerikoli drugi depot, do katerega imate dostop.

Poženite ukaz `git p4 clone`, da uvozite projekt Jam iz strežnika Perforce, kar dobavlja depot in pot projekta ter pot v katero želite uvoziti projekt:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

Ta določen projekt ima samo eno vejo vendar če imate veje, ki so nastavljive s pogledi

vej (ali samo skupkom direktorijev), lahko uporabite zastavico `--detect-branches` na `git p4 clone`, da uvozite tudi vse veje projekta. Glejte [Branching](#) za nekoliko več podrobnosti o tem.

Na tej točki ste že skoraj končali. Če greste v direktorij `p4import` in poženete `git log`, lahko vidite vaše uvoženo delo:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

[git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Vidite lahko, da je `git-p4` pustil identifikator v vsakem sporočilu pošiljanja. V redu je obdržati ta identifikator tam v primeru, če se potrebujete sklicevati na Perforce število spremembe kasneje. Vendar, če želite odstraniti identifikator je to sedaj čas, da naredite - preden začnete delati delo na novem repozitoriju. Lahko uporabite `git filter-branch`, da odstranite nize identifikatorja v celoti:

```
$ git filter-branch --msg-filter 'sed -e "/^\[git-p4:/d"'
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Če poženete `git log` lahko vidite, da so bile vse preverjene vsote SHA-1 za pošiljanja spremenjene vendar nizi `git-p4` niso več v sporočilih pošiljanja:


```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

Correction to line 355; change to .

```
commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800
```

Fix spelling error on Jam doc page (cummulative -> cumulative).

Vaš uvoz je sedaj pripravljen potisniti na vaš novi strežnik Git.

TFS

Če vaša ekipa pretvarja svoj nadzor izvirne kode iz TFVC na Git, boste želeli najvišjo zvestobo pretvorbe, ki jo lahko dobite. To pomeni, da medtem ko smo pokrili tako git-tfv in git-tf za sekcijo interoperabilnosti, bomo pokrili samo git-tfs za ta del, ker git-tfs podpira veje in to je nedopustno težko z uporabo git-tf.

NOTE

To je enosmerna pretvorba. Rezultirajoči repozitorij Git se ne bo zmožen povezati z originalnim projektom TFVC.

Prva stvar, ki jo morate narediti je preslikati uporabniška imena. TFVC je precej liberalen s tem, kar gre v polje avtorja za skupke sprememb, vendar Git želi človeku bralno ime in naslov e-pošte. Te informacije lahko dobite iz klienta ukazne vrstice **tf**, sledeče:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

To vzame vse skupke sprememb v zgodovini projekta in jih da v datoteko **AUTHORS_TMP**, ki ga bomo procesirali za razširitev podatkov stolpca *User* (2. stolpec). Odprite datoteko in najdite kateri znaki se začnejo na koncu stolpca in zamenjajo v sledeči ukazni vrstici, parametri **11-20** ukaza **cut** s tistimi najdenimi:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | uniq | sort > AUTHORS
```

Ukaz **cut** obdrži samo znake med 11 in 20 iz vsake vrstice. Ukaz **tail** preskoči prvi dve vrstici, ki sta glavi polj in podčrtaji ASCII-art. Rezultat vsega tega je preusmerjen na **uniq**, da eliminira duplikate in shrani datoteko imenovano **AUTHORS**. Naslednji korak je ročen; da je git-tfs efektiven uporabite to datoteko, vsaka vrstica mora biti tega formata:

```
DOMAIN\username = User Name <email@address.com>
```

Del na levi je polje “User” iz TFVC in del na desni strani znaka za enakost je uporabniško ime, ki bo uporabljeno za pošiljanja Git.

Ko imate enkrat to datoteko, je naslednja stvar narediti polno kloniranje projekta TFVC, za katerega ste zainteresirani:

```
PS> git tfs clone --with-branches --authors=AUTHORS  
https://username.visualstudio.com/DefaultCollection $/project/Trunk project_git
```

Naslednje boste želeli počistiti sekcije `git-tfs-id` iz dna sporočila pošiljanja. Sledeči ukaz bo to naredil:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$/g"' -- --all
```

To uporablja ukaz `sed` iz okolja Git-bash, da zamenja katerokoli vrstico, ki se začne z “git-tfs-id:” s praznino, ki jo bo Git nato ignoriral.

Ko je enkrat to narejeno, ste pripravljeni, da dodate novo daljavo, potisnete navzgor vse vaše veje in vaša ekipa prične delati iz Git-a.

A Custom Importer

If your system isn't one of the above, you should look for an importer online – quality importers are available for many other systems, including CVS, Clear Case, Visual Source Safe, even a directory of archives. If none of these tools works for you, you have a more obscure tool, or you otherwise need a more custom importing process, you should use `git fast-import`. This command reads simple instructions from stdin to write specific Git data. It's much easier to create Git objects this way than to run the raw Git commands or try to write the raw objects (see [Notranjost Git-a](#) for more information). This way, you can write an import script that reads the necessary information out of the system you're importing from and prints straightforward instructions to stdout. You can then run this program and pipe its output through `git fast-import`.

To quickly demonstrate, you'll write a simple importer. Suppose you work in `current`, you back up your project by occasionally copying the directory into a time-stamped `back_YYYY_MM_DD` backup directory, and you want to import this into Git. Your directory structure looks like this:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

In order to import a Git directory, you need to review how Git stores its data. As you may remember, Git is fundamentally a linked list of commit objects that point to a snapshot of content. All you have to do is tell `fast-import` what the content snapshots are, what commit data points to them, and the order they go in. Your strategy will be to go through the snapshots one at a time and create commits with the contents of each directory, linking each commit back to the previous one.

As we did in [An Example Git-Enforced Policy](#), we'll write this in Ruby, because it's what we generally work with and it tends to be easy to read. You can write this example pretty easily in anything you're familiar with – it just needs to print the appropriate information to `stdout`. And, if you are running on Windows, this means you'll need to take special care to not introduce carriage returns at the end your lines – `git fast-import` is very particular about just wanting line feeds (LF) not the carriage return line feeds (CRLF) that Windows uses.

To begin, you'll change into the target directory and identify every subdirectory, each of which is a snapshot that you want to import as a commit. You'll change into each subdirectory and print the commands necessary to export it. Your basic main loop looks like this:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

You run `print_export` inside each directory, which takes the manifest and mark of the previous snapshot and returns the manifest and mark of this one; that way, you can link them properly. “Mark” is the `fast-import` term for an identifier you give to a commit; as you create commits, you give each one a mark that you can use to link to it from other commits. So, the first thing to do in your `print_export` method is generate a mark from the directory name:

```
mark = convert_dir_to_mark(dir)
```

You'll do this by creating an array of directories and using the index value as the mark, because a mark must be an integer. Your method looks like this:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Now that you have an integer representation of your commit, you need a date for the commit metadata. Because the date is expressed in the name of the directory, you'll parse it out. The next line in your `print_export` file is

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

That returns an integer value for the date of each directory. The last piece of meta-information you need for each commit is the committer data, which you hardcode in a global variable:

```
$author = 'John Doe <john@example.com>'
```

Now you're ready to begin printing out the commit data for your importer. The initial information states that you're defining a commit object and what branch it's on, followed by the mark you've generated, the committer information and commit message, and then the previous commit, if any. The code looks like this:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word `data`, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

All that's left is to specify the file contents for each snapshot. This is easy, because you have each one in a directory – you can print out the `deleteall` command followed by the contents of each file in the directory. Git will then record each snapshot appropriately:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Note: Because many systems think of their revisions as changes from one commit to another, `fast-import` can also take commands with each commit to specify which files have been added, removed, or modified and what the new contents are. You could calculate the differences between snapshots and provide only this data, but doing so is more complex – you may as well give Git all the data and let it figure it out. If this is better suited to your data, check the `fast-import` man page for details about how to provide your data in this manner.

The format for listing the new file contents or specifying a modified file with the new contents is as follows:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Here, 644 is the mode (if you have executable files, you need to detect and specify 755 instead), and inline says you'll list the contents immediately after this line. Your `inline_data` method looks like this:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

You reuse the `export_data` method you defined earlier, because it's the same as the way you specified your commit message data.

The last thing you need to do is to return the current mark so it can be passed to the next iteration:

```
return mark
```

NOTE

If you are running on Windows you'll need to make sure that you add one extra step. As mentioned before, Windows uses CRLF for new line characters while git fast-import expects only LF. To get around this problem and make git fast-import happy, you need to tell ruby to use LF instead of CRLF:

```
$stdout.binmode
```

That's it. Here's the script in its entirety:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir)+1).to_s
end
```

```

def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end

def export_data(string)
  print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end

def print_export(dir, last_mark)
  date = convert_dir_to_date(dir)
  mark = convert_dir_to_mark(dir)

  puts 'commit refs/heads/master'
  puts "mark :#{mark}"
  puts "committer #{ENV['author']} #{date} -0700"
  export_data("imported from #{dir}")
  puts "from :#{last_mark}" if last_mark

  puts 'deleteall'
  Dir.glob("**/*").each do |file|
    next if !File.file?(file)
    inline_data(file)
  end
  mark
end

# Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end

```

```
end
end
```

If you run this script, you'll get content that looks something like this:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

To run the importer, pipe this output through `git fast-import` while in the Git directory you want to import into. You can create a new directory and then run `git init` in it for a starting point, and then run your script:


```

$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:       13 (      6 duplicates      )
  blobs :            5 (      4 duplicates      3 deltas of      5
attempts)
  trees :            4 (      1 duplicates      0 deltas of      4
attempts)
  commits:           4 (      1 duplicates      0 deltas of      0
attempts)
  tags :             0 (      0 duplicates      0 deltas of      0
attempts)
Total branches:      1 (      1 loads      )
  marks:            1024 (      5 unique      )
  atoms:             2
Memory total:        2344 KiB
  pools:            2110 KiB
  objects:           234 KiB
-----
pack_report: getpagesize() =      4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr =      10
pack_report: pack_mmap_calls =      5
pack_report: pack_open_windows =      2 /      2
pack_report: pack_mapped =      1457 /      1457
-----

```

As you can see, when it completes successfully, it gives you a bunch of statistics about what it accomplished. In this case, you imported 13 objects total for 4 commits into 1 branch. Now, you can run `git log` to see your new history:

```

$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date: Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date: Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03

```

There you go – a nice, clean Git repository. It's important to note that nothing is checked out – you don't have any files in your working directory at first. To get them, you must reset your branch to where `master` is now:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

You can do a lot more with the `fast-import` tool – handle different modes, binary data, multiple branches and merging, tags, progress indicators, and more. A number of examples of more complex scenarios are available in the `contrib/fast-import` directory of the Git source code.

Povzetek

Morali se bi počutiti udobno z uporabo Git-a kot klienta za ostale sisteme nadzora različic ali uvoziti skoraj kateregakoli obstoječega repozitorija v Git brez izgube podatkov. V naslednjem poglavju bomo pokrili surove notranjosti Git-a, tako da lahko izdelujete praktično vsak bajt, če je potreba.

Notranjost Git-a

Morda ste preskočili na to poglavje iz prejšnjega poglavja ali ste morda prišli sem po branju preostanka knjige - v katerem koli primeru, to je, kjer bomo pokrili notranje delovanje in implementacijo Git-a. Ugotovili smo, da je učenje teh informacij v osnovi pomembno za razumevanje, kako uporaben in močan Git je, vendar ostali so nam trdili, da je lahko zmedeno in nepotrebno kompleksno za začetnike. Vseeno, smo naredili to diskusijo v zadnjem poglavju knjige, da ga lahko preberete prej ali kasneje v vašem procesu učenja. Odločitev prepuščamo vam.

Sedaj ko ste tu, pričnimo. Najprej, če ni še dovolj jasno, Git je v bistvu vsebinsko naslavljajoč datotečni sistem z VCS uporabniškim vmesnikom napisanim na vrhu njega. Nekoliko se boste o tem naučili, kaj to pomeni.

V zgodnjih dneva Git-a (večinoma pred 1.5) je bil uporabniški vmesnik veliko bolj kompleksen, ker je povdarjal ta datotečni sistem namesto poliranega VCS-ja. V zadnjih nekaj letih je bil UI rafiniran dokler ni postal kakor čist in enostaven za uporabo od katerega koli sistema na voljo; vendar pogostokrat stereotip ostaja o zgodnjem UI Git-a, da je kompleksen in težek se za naučiti.

Datotečni sistem naslavljanja vsebine je izjemno cool, torej bomo pokrili prvo to v tem poglavju; nato se boste naučili o mehanizmih transporta in opravilih vzdrževanja repozitorija, s katerimi se boste eventuelno morali ukvarjati.

Napeljava in porcelan

Ta knjiga pokriva, kako uporabljati Git s 30 ali nekaj glagoli, kot so `checkout`, `branch`, `remote` itd. Vendar ker je bil Git prvotno orodje za VCS, ima namesto polnega uporabniku prijaznega VCS-ja kopico glagolov, ki delajo nižje nivojsko delo in so bila načrtovana za veriženje skupaj v stilu UNIX-a oz. klicana iz skript. Ti ukazi so v splošnem imenovani kot ukazi "plumbing" in bolj prijazni ukazi so imenovani ukazi "porcelain".

Prvih devet poglavij knjige se ukvarjajo skoraj ekskluzivno z ukazi porcelana. Vendar v tem poglavju se boste ukvarjali večinoma z nižje nivojskimi ukazi napeljave, ker vam dajo dostop do notranjega delovanja Git-a in pomagajo demonstrirati, kako in zakaj Git dela, kar dela. Mnogo teh ukazov ni mišljenih, da so uporabljeni ročno na ukazni vrstici vendar so bolj uporabljeni za gradnjo blokov za nova orodja in skripte po meri.

Ko poženete ukaz `git init` v novem obstoječem direktoriju, Git ustvari direktorij `.git`, kjer je locirano skoraj vse, kar Git shranjuje in manipulira. Če želite iti nazaj ali klonirati vaš repozitorij, kopirati ta posamezni direktorij nekam drugam, vam da skoraj vse, kar potrebujete. To celotno poglavje se v osnovi ukvarja s stvarmi v tem direktoriju. Tu je, kako izgleda:

```
$ ls -F1
HEAD
config*
description
hooks/
info/
objects/
refs/
```

Tam lahko vidite nekatere ostale datoteke, vendar to je sveži `git init` repozitorij - to je, kar vidite privzeto. Datoteka `description` je samo uporabljena s programom GitWeb, tako da ne skrbite o njem. Datoteka `config` vsebuje vaše nastavitvene opcije določenega projekta in direktorij `info` drži globalno izključitveno datoteko za ignorirane vzorce, ki jim ne želite slediti v datoteki `.gitignore`. Direktorij `hooks` vsebuje vaše skripte kljuk klientne ali strežniške strani, ki so diskutirane v [podrobnostih v Git kljuka](#).

To pusti štiri pomembne vnose: `HEAD` in (še za ustvariti) datoteko `index` in direktorija `objects` in `refs`. Te so deli jedra Git-a. Direktorij `objects` shranjuje vso vsebino za vašo podatkovno bazo, direktorij `refs` shranjuje kazalce v objekte pošiljanja v teh podatkih (branches) v datoteki `HEAD`, ki kaže na vejo, ki ste jo trenutno izpisali in datoteko `index`, kjer Git shranjuje informacije vaše vmesne faze. Sedaj boste pogledali vsako od teh sekcij v podrobnosti, da vidite, kako Git operira.

Git Objects

Git is a content-addressable filesystem. Great. What does that mean? It means that at the core of Git is a simple key-value data store. You can insert any kind of content into it, and it will give you back a key that you can use to retrieve the content again at any time. To demonstrate, you can use the plumbing command `hash-object`, which takes some data, stores it in your `.git` directory, and gives you back the key the data is stored as. First, you initialize a new Git repository and verify that there is nothing in the `objects` directory:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git has initialized the `objects` directory and created `pack` and `info` subdirectories in it, but there are no regular files. Now, store some text in your Git database:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

The `-w` tells `hash-object` to store the object; otherwise, the command simply tells you what the key would be. `--stdin` tells the command to read the content from stdin; if you don't specify this, `hash-object` expects a file path at the end. The output from the command is a 40-character checksum hash. This is the SHA-1 hash – a checksum of the content you're storing plus a header, which you'll learn about in a bit. Now you can see how Git has stored your data:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

You can see a file in the `objects` directory. This is how Git stores the content initially – as a single file per piece of content, named with the SHA-1 checksum of the content and its header. The subdirectory is named with the first 2 characters of the SHA-1, and the filename is the remaining 38 characters.

You can pull the content back out of Git with the `cat-file` command. This command is sort of a Swiss army knife for inspecting Git objects. Passing `-p` to it instructs the `cat-file` command to figure out the type of content and display it nicely for you:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Now, you can add content to Git and pull it back out again. You can also do this with content in files. For example, you can do some simple version control on a file. First, create a new file and save its contents in your database:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Then, write some new content to the file, and save it again:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Your database contains the two new versions of the file as well as the first content you stored there:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Now you can revert the file back to the first version

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

or the second version:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

But remembering the SHA-1 key for each version of your file isn't practical; plus, you aren't storing the filename in your system – just the content. This object type is called a blob. You can have Git tell you the object type of any object in Git, given its SHA-1 key, with `cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Tree Objects

The next type we'll look at is the tree, which solves the problem of storing the filename and also allows you to store a group of files together. Git stores content in a manner similar to a UNIX filesystem, but a bit simplified. All the content is stored as tree and blob objects, with trees corresponding to UNIX directory entries and blobs corresponding more or less to inodes or file contents. A single tree object contains one or more tree entries, each of which contains a SHA-1 pointer to a blob or subtree with its associated mode, type, and filename. For example, the most recent tree in a project may look something like this:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859    README
100644 blob 8f94139338f9404f26296befa88755fc2598c289    Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0    lib
```

The `master^{tree}` syntax specifies the tree object that is pointed to by the last commit on your `master` branch. Notice that the `lib` subdirectory isn't a blob but a pointer to another tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Conceptually, the data that Git is storing is something like this:

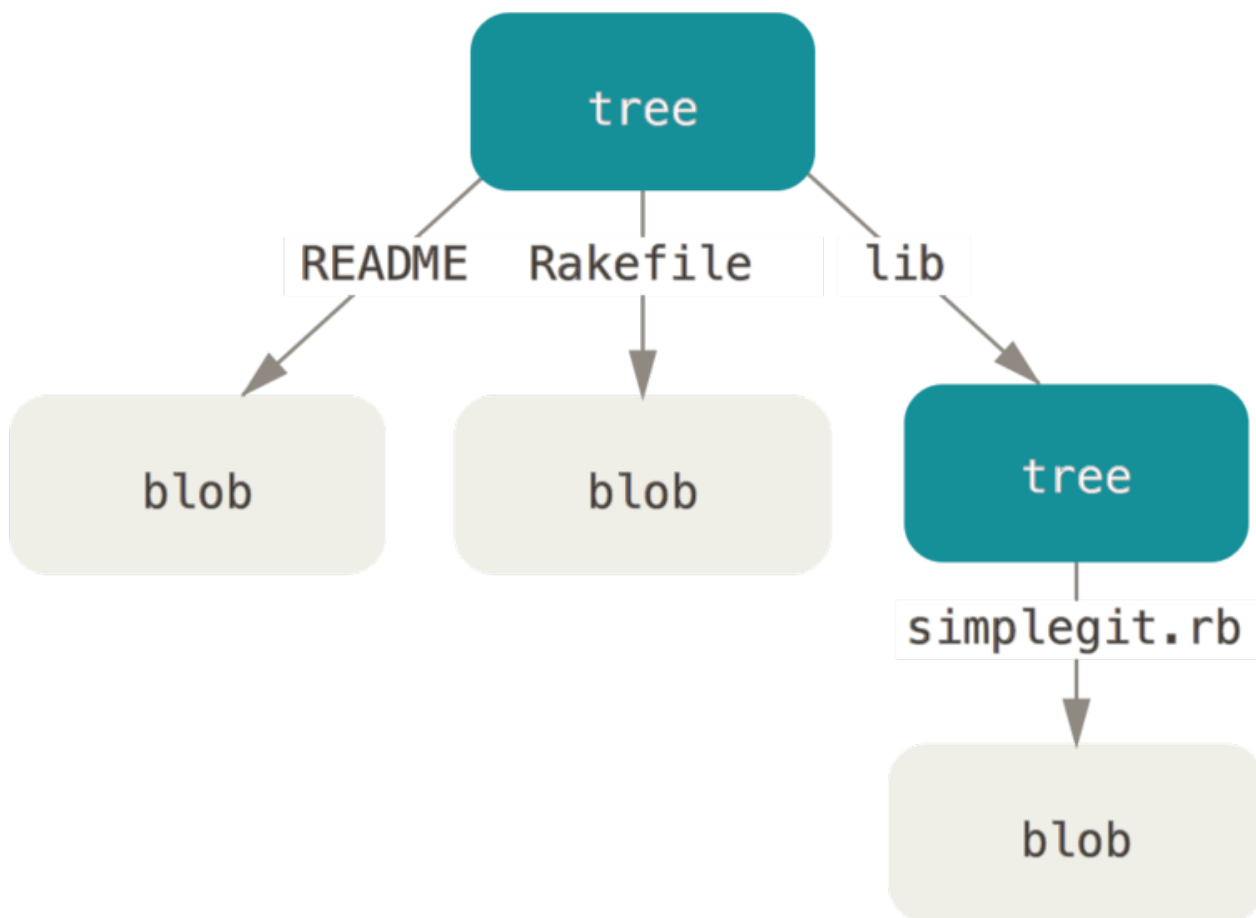


Figure 149. Simple version of the Git data model.

You can fairly easily create your own tree. Git normally creates a tree by taking the state of your staging area or index and writing a series of tree objects from it. So, to create a tree object, you first have to set up an index by staging some files. To create an index with a single entry – the first version of your `test.txt` file – you can use the plumbing command `update-index`. You use this command to artificially add the earlier version of the `test.txt` file to a new staging area. You must pass it the `--add` option because the file doesn't yet exist in your staging area (you don't even have a staging area set up yet) and `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, you specify the mode, SHA-1, and filename:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In this case, you're specifying a mode of `100644`, which means it's a normal file. Other options are `100755`, which means it's an executable file; and `120000`, which specifies a symbolic link. The mode is taken from normal UNIX modes but is much less flexible –

these three modes are the only ones that are valid for files (blobs) in Git (although other modes are used for directories and submodules).

Now, you can use the `write-tree` command to write the staging area out to a tree object. No `-w` option is needed – calling `write-tree` automatically creates a tree object from the state of the index if that tree doesn't yet exist:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

You can also verify that this is a tree object:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

You'll now create a new tree with the second version of `test.txt` and a new file as well:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Your staging area now has the new version of `test.txt` as well as the new file `new.txt`. Write out that tree (recording the state of the staging area or index to a tree object) and see what it looks like:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt
```

Notice that this tree has both file entries and also that the `test.txt` SHA-1 is the “version 2” SHA-1 from earlier (`1f7a7a`). Just for fun, you'll add the first tree as a subdirectory into this one. You can read trees into your staging area by calling `read-tree`. In this case, you can read an existing tree into your staging area as a subtree by using the `--prefix` option to `read-tree`:


```

$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92    new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a    test.txt

```

If you created a working directory from the new tree you just wrote, you would get the two files in the top level of the working directory and a subdirectory named `bak` that contained the first version of the `test.txt` file. You can think of the data that Git contains for these structures as being like this:

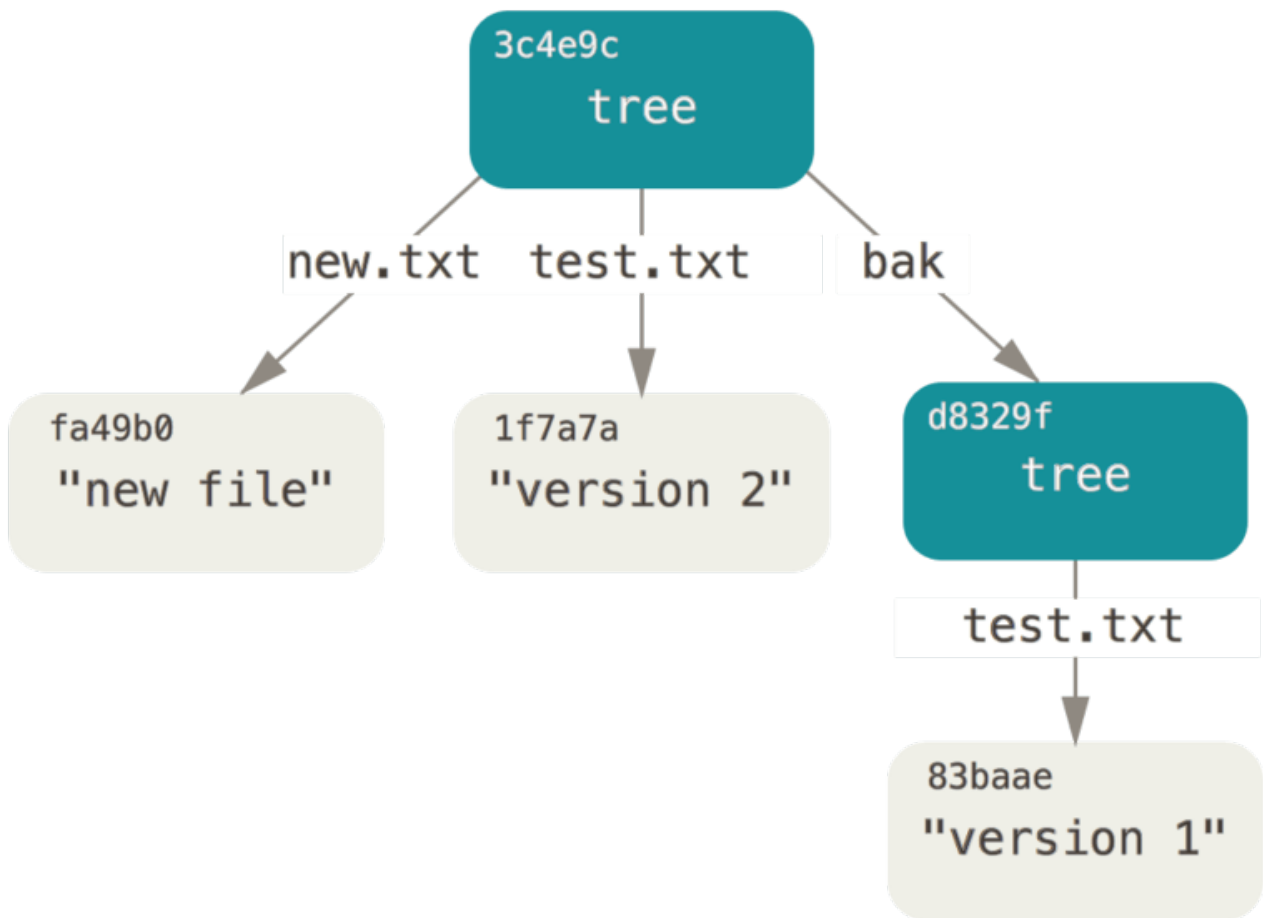


Figure 150. The content structure of your current Git data.

Commit Objects

You have three trees that specify the different snapshots of your project that you want to track, but the earlier problem remains: you must remember all three SHA-1 values in order to recall the snapshots. You also don't have any information about who saved the snapshots, when they were saved, or why they were saved. This is the basic information that the commit object stores for you.

To create a commit object, you call `commit-tree` and specify a single tree SHA-1 and

which commit objects, if any, directly preceded it. Start with the first tree you wrote:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Now you can look at your new commit object with `cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

The format for a commit object is simple: it specifies the top-level tree for the snapshot of the project at that point; the author/committer information (which uses your `user.name` and `user.email` configuration settings and a timestamp); a blank line, and then the commit message.

Next, you'll write the other two commit objects, each referencing the commit that came directly before it:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Each of the three commit objects points to one of the three snapshot trees you created. Oddly enough, you have a real Git history now that you can view with the `git log` command, if you run it on the last commit SHA-1:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700
```

third commit

```
bak/test.txt | 1 +
1 file changed, 1 insertion(+)
```

```
commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:14:29 2009 -0700
```

second commit

```
new.txt | 1 +
test.txt | 2 +-
2 files changed, 2 insertions(+), 1 deletion(-)
```

```
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:09:34 2009 -0700
```

first commit

```
test.txt | 1 +
1 file changed, 1 insertion(+)
```

Amazing. You've just done the low-level operations to build up a Git history without using any of the front end commands. This is essentially what Git does when you run the `git add` and `git commit` commands – it stores blobs for the files that have changed, updates the index, writes out trees, and writes commit objects that reference the top-level trees and the commits that came immediately before them. These three main Git objects – the blob, the tree, and the commit – are initially stored as separate files in your `.git/objects` directory. Here are all the objects in the example directory now, commented with what they store:

```

$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1

```

If you follow all the internal pointers, you get an object graph something like this:

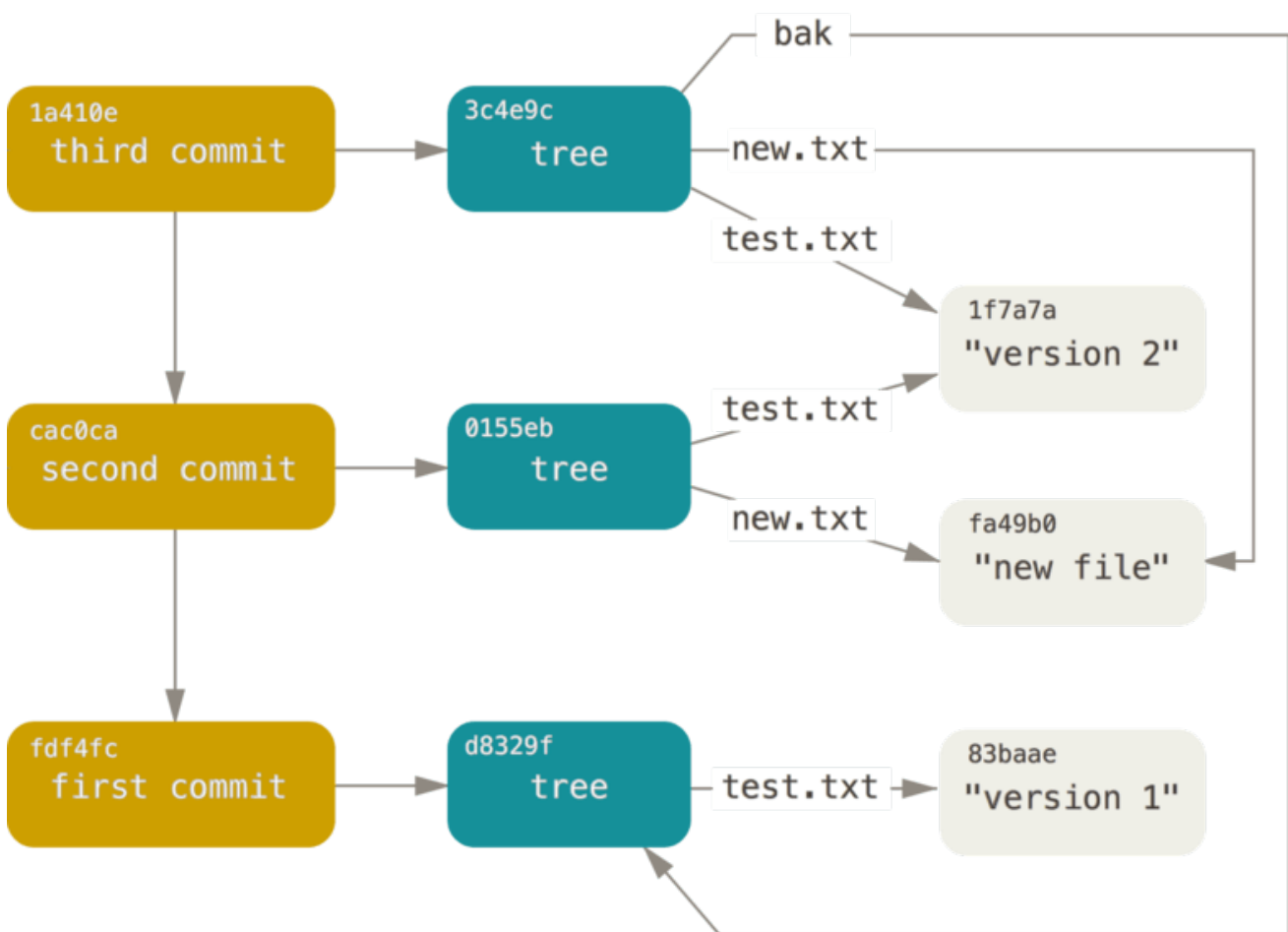


Figure 151. All the objects in your Git directory.

Object Storage

We mentioned earlier that a header is stored with the content. Let's take a minute to look at how Git stores its objects. You'll see how to store a blob object – in this case, the string “what is up, doc?” – interactively in the Ruby scripting language.

You can start up interactive Ruby mode with the `irb` command:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git constructs a header that starts with the type of the object, in this case a blob. Then, it adds a space followed by the size of the content and finally a null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Git concatenates the header and the original content and then calculates the SHA-1 checksum of that new content. You can calculate the SHA-1 value of a string in Ruby by including the SHA1 digest library with the `require` command and then calling `Digest::SHA1.hexdigest()` with the string:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git compresses the new content with zlib, which you can do in Ruby with the zlib library. First, you need to require the library and then run `Zlib::Deflate.deflate()` on the content:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "\x9CK\xCA\xC90R04c(\xCFH,Q\xC8,V(-\xD0QH\xC90\xB6\a\x00_\x1C\a\x9D"
```

Finally, you'll write your zlib-deflated content to an object on disk. You'll determine the path of the object you want to write out (the first two characters of the SHA-1 value being the subdirectory name, and the last 38 characters being the filename within that directory). In Ruby, you can use the `FileUtils.mkdir_p()` function to create the subdirectory if it doesn't exist. Then, open the file with `File.open()` and write out the previously zlib-compressed content to the file with a `write()` call on the resulting file handle:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

That's it – you've created a valid Git blob object. All Git objects are stored the same way, just with different types – instead of the string blob, the header will begin with commit or tree. Also, although the blob content can be nearly anything, the commit and tree content are very specifically formatted.

Git References

You can run something like `git log 1a410e` to look through your whole history, but you still have to remember that `1a410e` is the last commit in order to walk that history to find all those objects. You need a file in which you can store the SHA-1 value under a simple name so you can use that pointer rather than the raw SHA-1 value.

In Git, these are called “references” or “refs”; you can find the files that contain the SHA-1 values in the `.git/refs` directory. In the current project, this directory contains no files, but it does contain a simple structure:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

To create a new reference that will help you remember where your latest commit is, you can technically do something as simple as this:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Now, you can use the head reference you just created instead of the SHA-1 value in your Git commands:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You aren't encouraged to directly edit the reference files. Git provides a safer

command to do this if you want to update a reference called `update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

That's basically what a branch in Git is: a simple pointer or reference to the head of a line of work. To create a branch back at the second commit, you can do this:

```
$ git update-ref refs/heads/test cac0ca
```

Your branch will contain only work from that commit down:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, your Git database conceptually looks something like this:

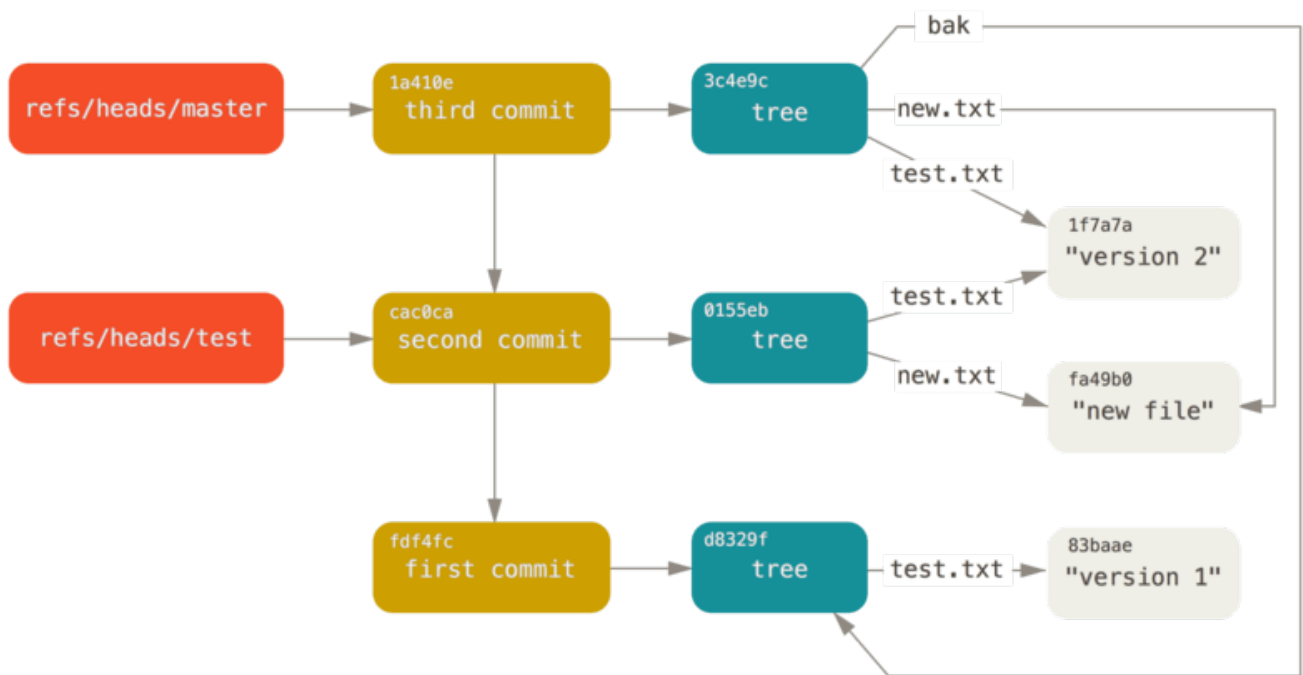


Figure 152. Git directory objects with branch head references included.

When you run commands like `git branch (branchname)`, Git basically runs that `update-ref` command to add the SHA-1 of the last commit of the branch you're on into whatever new reference you want to create.

The HEAD

The question now is, when you run `git branch (branchname)`, how does Git know the SHA-1 of the last commit? The answer is the HEAD file.

The HEAD file is a symbolic reference to the branch you're currently on. By symbolic

reference, we mean that unlike a normal reference, it doesn't generally contain a SHA-1 value but rather a pointer to another reference. If you look at the file, you'll normally see something like this:

```
$ cat .git/HEAD
ref: refs/heads/master
```

If you run `git checkout test`, Git updates the file to look like this:

```
$ cat .git/HEAD
ref: refs/heads/test
```

When you run `git commit`, it creates the commit object, specifying the parent of that commit object to be whatever SHA-1 value the reference in HEAD points to.

You can also manually edit this file, but again a safer command exists to do so: `symbolic-ref`. You can read the value of your HEAD via this command:

```
$ git symbolic-ref HEAD
refs/heads/master
```

You can also set the value of HEAD:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

You can't set a symbolic reference outside of the refs style:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

We just finished discussing Git's three main object types, but there is a fourth. The tag object is very much like a commit object – it contains a tagger, a date, a message, and a pointer. The main difference is that a tag object generally points to a commit rather than a tree. It's like a branch reference, but it never moves – it always points to the same commit but gives it a friendlier name.

As discussed in [Osnove Git](#), there are two types of tags: annotated and lightweight. You can make a lightweight tag by running something like this:


```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

That is all a lightweight tag is – a reference that never moves. An annotated tag is more complex, however. If you create an annotated tag, Git creates a tag object and then writes a reference to point to it rather than directly to the commit. You can see this by creating an annotated tag (-a specifies that it's an annotated tag):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Here's the object SHA-1 value it created:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Now, run the `cat-file` command on that SHA-1 value:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

Notice that the object entry points to the commit SHA-1 value that you tagged. Also notice that it doesn't need to point to a commit; you can tag any Git object. In the Git source code, for example, the maintainer has added their GPG public key as a blob object and then tagged it. You can view the public key by running this in a clone of the Git repository:

```
$ git cat-file blob junio-gpg-pub
```

The Linux kernel repository also has a non-commit-pointing tag object – the first tag created points to the initial tree of the import of the source code.

Remotes

The third type of reference that you'll see is a remote reference. If you add a remote and push to it, Git stores the value you last pushed to that remote for each branch in the `refs/remotes` directory. For instance, you can add a remote called `origin` and push your `master` branch to it:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
a11bef0..ca82a6d master -> master
```

Then, you can see what the `master` branch on the `origin` remote was the last time you communicated with the server, by checking the `refs/remotes/origin/master` file:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote references differ from branches (`refs/heads` references) mainly in that they're considered read-only. You can `git checkout` to one, but Git won't point HEAD at one, so you'll never update it with a `commit` command. Git manages them as bookmarks to the last known state of where those branches were on those servers.

Packfiles

Let's go back to the objects database for your test Git repository. At this point, you have 11 objects – 4 blobs, 3 trees, 3 commits, and 1 tag:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git compresses the contents of these files with `zlib`, and you're not storing much, so all these files collectively take up only 925 bytes. You'll add some larger content to the repository to demonstrate an interesting feature of Git. To demonstrate, we'll add the `repo.rb` file from the Grit library – this is about a 22K source code file:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb >
repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
3 files changed, 709 insertions(+), 2 deletions(-)
delete mode 100644 bak/test.txt
create mode 100644 repo.rb
rewrite test.txt (100%)
```

If you look at the resulting tree, you can see the SHA-1 value your `repo.rb` file got for the blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

You can then use `git cat-file` to see how big that object is:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Now, modify that file a little, and see what happens:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master 2431da6] modified repo.rb a bit
1 file changed, 1 insertion(+)
```

Check the tree created by that commit, and you see something interesting:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e    repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b    test.txt
```

The blob is now a different blob, which means that although you added only a single line to the end of a 400-line file, Git stored that new content as a completely new object:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

You have two nearly identical 22K objects on your disk. Wouldn't it be nice if Git could store one of them in full but then the second object only as the delta between it and the first?

It turns out that it can. The initial format in which Git saves objects on disk is called a “loose” object format. However, occasionally Git packs up several of these objects into a single binary file called a “packfile” in order to save space and be more efficient. Git does this if you have too many loose objects around, if you run the `git gc` command manually, or if you push to a remote server. To see what happens, you can manually ask Git to pack up the objects by calling the `git gc` command:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

If you look in your objects directory, you'll find that most of your objects are gone, and a new pair of files has appeared:

```
$ find .git/objects -type f
.git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.idx
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack
```

The objects that remain are the blobs that aren't pointed to by any commit – in this case, the “what is up, doc?” example and the “test content” example blobs you created earlier. Because you never added them to any commits, they're considered dangling and aren't packed up in your new packfile.

The other files are your new packfile and an index. The packfile is a single file containing the contents of all the objects that were removed from your filesystem. The index is a file that contains offsets into that packfile so you can quickly seek to a specific object. What is cool is that although the objects on disk before you ran the `gc` were collectively about 22K in size, the new packfile is only 7K. You've cut your disk usage by by packing your objects.

How does Git do this? When Git packs objects, it looks for files that are named and sized similarly, and stores just the deltas from one version of the file to the next. You can look into the packfile and see what Git did to save space. The `git verify-pack` plumbing command allows you to see what was packed up:

```

$ git verify-pack -v .git/objects/pack/pack-
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
  deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \
  b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok

```

Here, the `033b4` blob, which if you remember was the first version of your `repo.rb` file, is referencing the `b042a` blob, which was the second version of the file. The third column in the output is the size of the object in the pack, so you can see that `b042a` takes up 22K of the file, but that `033b4` only takes up 9 bytes. What is also interesting is that the second version of the file is the one that is stored intact, whereas the original version is stored as a delta – this is because you’re most likely to need faster access to the most recent version of the file.

The really nice thing about this is that it can be repacked at any time. Git will occasionally repack your database automatically, always trying to save more space, but you can also manually repack at any time by running `git gc` by hand.

The Refspec

Skozi to knjigo smo uporabljali enostavne preslikave iz oddaljenih vej na lokalne reference, vendar lahko so bolj kompleksne. Predpostavimo, da dodate daljavo sledeče:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Doda sekcijo v vašo datoteko `.git/config`, ki določa ime daljave (`origin`), URL oddaljenega

repozitorija in refs spec za ujemanje:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Oblika refs spec-a je opsijski `+`, ki mu sledi `<src>:<dst>`, kjer je `<src>` vzorec za reference na oddaljeni strani in `<dst>` je lokacija, kamor bodo te reference zapisane lokalno. `+` pove Git-u, da posodobi referenco tudi če ni t.i. fast-forward.

V privzetem primeru, je to avtomatsko prepisano z ukazom `git remote add`, Git ujame vse reference pod `refs/heads/` na strežniku in jih zapiše v `refs/remotes/origins/` lokalno. Torej, če je na strežniku veja `master`, lahko dostopate do dnevnika te veje lokalno preko

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Vsi ukazi so ekvivalentni, ker Git vsakega razširi v `refs/remotes/origins/master`.

Če želite, da Git namesto tega vsakič potegne samo vejo `master` in ne vseh ostalih vej na oddaljenem strežniku, lahko spremenite vrstico ujetja na

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

To je samo privzeti refs spec za `git fetch` za to daljavo. Če želite narediti nekaj enkrat, lahko določite refs spec tudi v ukazni vrstici. Da potegnete vejo `master` na daljavi dol v `origin/mymaster` lokalno, lahko poženete

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Lahko določite tudi več refs spec-ov. V ukazni vrstici lahko potegnete dol več vej sledeče:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
  topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
! [rejected]      master    -> origin/mymaster (non fast forward)
* [new branch]   topic     -> origin/topic
```

V tem primeru je bil poteg veje `master` zavržen, ker ni bil referenca fast-forward. To lahko prepišete z določanjem `+` na začetku refs spec.

V vaši nastavitveni datoteki lahko določite tudi več refs spec za ujetje. Če želite vedno ujeti veje `master` in `experiment`, dodajte dve vrstici:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/master:refs/remotes/origin/master
fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Ne morete uporabiti več globov v vzorcu, torej to bi bilo neveljavno:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Vendar lahko uporabite imenski prostor (ali direktorije) za doseg nečesa takega. Če imate ekipo QA, ki potiska serijo vej in želite dobiti vejo master in katerokoli vejo od ekipe QA vendar ničesar več, lahko uporabite sledečo sekcijo nastavitvev:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/master:refs/remotes/origin/master
fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Če imate kompleksen potek dela, ki ima potiskalne veje ekipe QA, potiskalne veje razvijalcev in ekipe integracije in sodelujete na oddaljenih vejah, lahko naredite imenski prostor enostavno na ta način.

Refspec-i za potiskanje

Lepo je, če lahko ujamete reference imenskega prostora na ta način, vendar kako ekipa QA dobi svoje veje v imenski prostor `qa/` na prvem mestu? To lahko dosežete z uporabo refspec-ov za potiskanje.

Če ekipa QA želi potisniti svojo vejo `master` v `qa/master` na oddaljenem strežniku, lahko poženejo

```
$ git push origin master:refs/heads/qa/master
```

Če želijo, da Git to naredi avtomatsko vsakič, ko poženejo `git push origin`, lahko dodajo vrednost `push` v svojo nastavitveno datoteko:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
push = refs/heads/master:refs/heads/qa/master
```

Ponovno, bo to povzročilo, da `git push origin` potisne lokalno vejo `master` na daljavo `qa/master` vejo privzeto.

Brisanje referenc

Lahko uporabite tudi refspec za izbris referenc iz oddaljenega strežnika s pogonom nečesa takega:

```
$ git push origin :topic
```

Ker je refspec `<src>:<dst>`, z opuščanjem `<src>` dela, to v osnovi pravi, da naredite tematsko vejo na nobeni daljavi, kar jo izbriše.

Transfer Protocols

Git can transfer data between two repositories in two major ways: the “dumb” protocol and the “smart” protocol. This section will quickly cover how these two main protocols operate.

The Dumb Protocol

If you’re setting up a repository to be served read-only over HTTP, the dumb protocol is likely what will be used. This protocol is called “dumb” because it requires no Git-specific code on the server side during the transport process; the fetch process is a series of HTTP `GET` requests, where the client can assume the layout of the Git repository on the server.

NOTE

The dumb protocol is fairly rarely used these days. It’s difficult to secure or make private, so most Git hosts (both cloud-based and on-premises) will refuse to use it. It’s generally advised to use the smart protocol, which we describe a bit further on.

Let’s follow the `http-fetch` process for the `simplegit` library:

```
$ git clone http://server/simplegit-progit.git
```

The first thing this command does is pull down the `info/refs` file. This file is written by the `update-server-info` command, which is why you need to enable that as a `post-receive` hook in order for the HTTP transport to work properly:

```
=> GET info/refs  
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Now you have a list of the remote references and SHA-1s. Next, you look for what the HEAD reference is so you know what to check out when you’re finished:


```
=> GET HEAD
ref: refs/heads/master
```

You need to check out the `master` branch when you've completed the process. At this point, you're ready to start the walking process. Because your starting point is the `ca82a6` commit object you saw in the `info/refs` file, you start by fetching that:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

You get an object back – that object is in loose format on the server, and you fetched it over a static HTTP GET request. You can `zlib-uncompress` it, strip off the header, and look at the commit content:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfd3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

Next, you have two more objects to retrieve – `cfd3b3`, which is the tree of content that the commit we just retrieved points to; and `085bb3`, which is the parent commit:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

That gives you your next commit object. Grab the tree object:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Oops – it looks like that tree object isn't in loose format on the server, so you get a 404 response back. There are a couple of reasons for this – the object could be in an alternate repository, or it could be in a packfile in this repository. Git checks for any listed alternates first:

```
=> GET objects/info/http-alternates
(empty file)
```

If this comes back with a list of alternate URLs, Git checks for loose files and packfiles there – this is a nice mechanism for projects that are forks of one another to

share objects on disk. However, because no alternates are listed in this case, your object must be in a packfile. To see what packfiles are available on this server, you need to get the `objects/info/packs` file, which contains a listing of them (also generated by `update-server-info`):

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

There is only one packfile on the server, so your object is obviously in there, but you'll check the index file to make sure. This is also useful if you have multiple packfiles on the server, so you can see which packfile contains the object you need:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

Now that you have the packfile index, you can see if your object is in it – because the index lists the SHA-1s of the objects contained in the packfile and the offsets to those objects. Your object is there, so go ahead and get the whole packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

You have your tree object, so you continue walking your commits. They're all also within the packfile you just downloaded, so you don't have to do any more requests to your server. Git checks out a working copy of the `master` branch that was pointed to by the HEAD reference you downloaded at the beginning.

The Smart Protocol

The dumb protocol is simple but a bit inefficient, and it can't handle writing of data from the client to the server. The smart protocol is a more common method of transferring data, but it requires a process on the remote end that is intelligent about Git – it can read local data, figure out what the client has and needs, and generate a custom packfile for it. There are two sets of processes for transferring data: a pair for uploading data and a pair for downloading data.

Uploading Data

To upload data to a remote process, Git uses the `send-pack` and `receive-pack` processes. The `send-pack` process runs on the client and connects to a `receive-pack` process on the remote side.

SSH

For example, say you run `git push origin master` in your project, and `origin` is defined as a URL that uses the SSH protocol. Git fires up the `send-pack` process, which

initiates a connection over SSH to your server. It tries to run a command on the remote server via an SSH call that looks something like this:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs
0000
```

The `git-receive-pack` command immediately responds with one line for each reference it currently has – in this case, just the `master` branch and its SHA-1. The first line also has a list of the server's capabilities (here, `report-status`, `delete-refs`, and some others, including the client identifier).

Each line starts with a 4-character hex value specifying how long the rest of the line is. Your first line starts with `00a5`, which is hexadecimal for 165, meaning that 165 bytes remain on that line. The next line is `0000`, meaning the server is done with its references listing.

Now that it knows the server's state, your `send-pack` process determines what commits it has that the server doesn't. For each reference that this push will update, the `send-pack` process tells the `receive-pack` process that information. For instance, if you're updating the `master` branch and adding an `experiment` branch, the `send-pack` response may look something like this:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
\
  refs/heads/master report-status
006c0000000000000000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
\
  refs/heads/experiment
0000
```

Git sends a line for each reference you're updating with the line's length, the old SHA-1, the new SHA-1, and the reference that is being updated. The first line also has the client's capabilities. The SHA-1 value of all '0's means that nothing was there before – because you're adding the `experiment` reference. If you were deleting a reference, you would see the opposite: all '0's on the right side.

Next, the client sends a packfile of all the objects the server doesn't have yet. Finally, the server responds with a success (or failure) indication:

```
0000eunpack ok
```

HTTP(S)

This process is mostly the same over HTTP, though the handshaking is a bit different. The connection is initiated with this request:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack
001f# service=git-receive-pack
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master␣report-status \
    delete-refs side-band-64k quiet ofs-delta \
    agent=git/2:2.1.1~vmg-bitmaps-bugaloo-608-g116744e
0000
```

That's the end of the first client-server exchange. The client then makes another request, this time a **POST**, with the data that **git-upload-pack** provides.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

The **POST** request includes the **send-pack** output and the packfile as its payload. The server then indicates success or failure with its HTTP response.

Downloading Data

When you download data, the **fetch-pack** and **upload-pack** processes are involved. The client initiates a **fetch-pack** process that connects to an **upload-pack** process on the remote side to negotiate what data will be transferred down.

SSH

If you're doing the fetch over SSH, **fetch-pack** runs something like this:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

After **fetch-pack** connects, **upload-pack** sends back something like this:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD␣multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

This is very similar to what **receive-pack** responds with, but the capabilities are different. In addition, it sends back what HEAD points to (**symref=HEAD:refs/heads/master**) so the client knows what to check out if this is a clone.

At this point, the **fetch-pack** process looks at what objects it has and responds with the

objects that it needs by sending “want” and then the SHA-1 it wants. It sends all the objects it already has with “have” and then the SHA-1. At the end of this list, it writes “done” to initiate the `upload-pack` process to begin sending the packfile of the data it needs:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

HTTP(S)

The handshake for a fetch operation takes two HTTP requests. The first is a `GET` to the same endpoint used in the dumb protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD multi_ack thin-pack \
    side-band side-band-64k ofs-delta shallow no-progress include-tag \
    multi_ack_detailed no-done symref=HEAD:refs/heads/master \
    agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

This is very similar to invoking `git-upload-pack` over an SSH connection, but the second exchange is performed as a separate request:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Again, this is the same format as above. The response to this request indicates success or failure, and includes the packfile.

Protocols Summary

This section contains a very basic overview of the transfer protocols. The protocol includes many other features, such as `multi_ack` or `side-band` capabilities, but covering them is outside the scope of this book. We’ve tried to give you a sense of the general back-and-forth between client and server; if you need more knowledge than this, you’ll probably want to take a look at the Git source code.

Maintenance and Data Recovery

Occasionally, you may have to do some cleanup – make a repository more compact,

clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

Maintenance

Occasionally, Git automatically runs a command called “auto gc”. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The “gc” stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren’t reachable from any commit and are a few months old.

You can run auto gc manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real gc command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you’ll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ cat .git/packed-refs
# pack-refs with: peeled fully-peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

If you update a reference, Git doesn’t edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can’t find a reference in the `refs` directory, it’s probably in your

packed-refs file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in [Git References](#). You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date: Fri May 22 18:15:24 2009 -0700

    modified repo.rb a bit
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Cool – now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```


Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In this case, you can see your missing commit after the string “dangling commit”. You can recover it the same way, by adding a branch that points to that SHA-1.

Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a `git clone` downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

Be warned: this technique is destructive to your commit history. It rewrites every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oops – you didn’t want to add a huge tarball to your project. Better get rid of it:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Now, `gc` your database and see how much space you’re using:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

You can run the `count-objects` command to quickly see how much space you’re using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you’re using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn’t remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let’s get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn’t; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by

running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ git verify-pack -v .git/objects/pack/pack-29...69.idx \  
  | sort -k 3 -n \  
  | tail -3  
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12  
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 22044 5792 4977696  
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob 4975916 4976258 1438
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in [Enforcing a Specific Commit-Message Format](#). If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 82c99a3  
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --oneline --branches -- git.tgz  
dadf725 oops - removed large tarball  
7b30847 add git tarball
```

You must rewrite all the commits downstream from `7b30847` to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in [Rewriting History](#):

```
$ git filter-branch --index-filter \  
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..  
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'  
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)  
Ref 'refs/heads/master' was rewritten
```

The `--index-filter` option is similar to the `--tree-filter` option used in [Rewriting History](#), except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` – you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to

`git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the `7b30847` commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (15/15), done.
Total 15 (delta 1), reused 12 (delta 0)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Environment Variables

Git always runs inside a `bash` shell, and uses a number of shell environment variables to determine how it behaves. Occasionally, it comes in handy to know what these are, and how they can be used to make Git behave the way you want it to. This isn't an exhaustive list of all the environment variables Git pays attention to, but we'll cover the most useful.

Global Behavior

Some of Git's general behavior as a computer program depends on environment variables.

`GIT_EXEC_PATH` determines where Git looks for its sub-programs (like `git-commit`, `git-diff`, and others). You can check the current setting by running `git --exec-path`.

`HOME` isn't usually considered customizable (too many other things depend on it), but it's where Git looks for the global configuration file. If you want a truly portable Git installation, complete with global configuration, you can override `HOME` in the portable Git's shell profile.

`PREFIX` is similar, but for the system-wide configuration. Git looks for this file at `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, if set, disables the use of the system-wide configuration file. This is useful if your system config is interfering with your commands, but you don't have access to change or remove it.

`GIT_PAGER` controls the program used to display multi-page output on the command line. If this is unset, `PAGER` will be used as a fallback.

`GIT_EDITOR` is the editor Git will launch when the user needs to edit some text (a commit message, for example). If unset, `EDITOR` will be used.

Repository Locations

Git uses several environment variables to determine how it interfaces with the

current repository.

`GIT_DIR` is the location of the `.git` folder. If this isn't specified, Git walks up the directory tree until it gets to `~` or `/`, looking for a `.git` directory at every step.

`GIT_CEILING_DIRECTORIES` controls the behavior of searching for a `.git` directory. If you access directories that are slow to load (such as those on a tape drive, or across a slow network connection), you may want to have Git stop trying earlier than it might otherwise, especially if Git is invoked when building your shell prompt.

`GIT_WORK_TREE` is the location of the root of the working directory for a non-bare repository. If not specified, the parent directory of `$GIT_DIR` is used.

`GIT_INDEX_FILE` is the path to the index file (non-bare repositories only).

`GIT_OBJECT_DIRECTORY` can be used to specify the location of the directory that usually resides at `.git/objects`.

`GIT_ALTERNATE_OBJECT_DIRECTORIES` is a colon-separated list (formatted like `/dir/one:/dir/two:...`) which tells Git where to check for objects if they aren't in `GIT_OBJECT_DIRECTORY`. If you happen to have a lot of projects with large files that have the exact same contents, this can be used to avoid storing too many copies of them.

Pathspeccs

A “pathspecc” refers to how you specify paths to things in Git, including the use of wildcards. These are used in the `.gitignore` file, but also on the command-line (`git add *.c`).

`GIT_GLOB_PATHSPECS` and `GIT_NOGLOB_PATHSPECS` control the default behavior of wildcards in pathspeccs. If `GIT_GLOB_PATHSPECS` is set to 1, wildcard characters act as wildcards (which is the default); if `GIT_NOGLOB_PATHSPECS` is set to 1, wildcard characters only match themselves, meaning something like `*.c` would only match a file *named* “*.c”, rather than any file whose name ends with `.c`. You can override this in individual cases by starting the pathspecc with `:(glob)` or `:(literal)`, as in `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` disables both of the above behaviors; no wildcard characters will work, and the override prefixes are disabled as well.

`GIT_ICASE_PATHSPECS` sets all pathspeccs to work in a case-insensitive manner.

Committing

The final creation of a Git commit object is usually done by `git-commit-tree`, which uses these environment variables as its primary source of information, falling back to configuration values only if these aren't present.

`GIT_AUTHOR_NAME` is the human-readable name in the “author” field.

`GIT_AUTHOR_EMAIL` is the email for the “author” field.

`GIT_AUTHOR_DATE` is the timestamp used for the “author” field.

`GIT_COMMITTER_NAME` sets the human name for the “committer” field.

`GIT_COMMITTER_EMAIL` is the email address for the “committer” field.

`GIT_COMMITTER_DATE` is used for the timestamp in the “committer” field.

`EMAIL` is the fallback email address in case the `user.email` configuration value isn’t set. If *this* isn’t set, Git falls back to the system user and host names.

Networking

Git uses the `curl` library to do network operations over HTTP, so `GIT_CURL_VERBOSE` tells Git to emit all the messages generated by that library. This is similar to doing `curl -v` on the command line.

`GIT_SSL_NO_VERIFY` tells Git not to verify SSL certificates. This can sometimes be necessary if you’re using a self-signed certificate to serve Git repositories over HTTPS, or you’re in the middle of setting up a Git server but haven’t installed a full certificate yet.

If the data rate of an HTTP operation is lower than `GIT_HTTP_LOW_SPEED_LIMIT` bytes per second for longer than `GIT_HTTP_LOW_SPEED_TIME` seconds, Git will abort that operation. These values override the `http.lowSpeedLimit` and `http.lowSpeedTime` configuration values.

`GIT_HTTP_USER_AGENT` sets the user-agent string used by Git when communicating over HTTP. The default is a value like `git/2.0.0`.

Diffing and Merging

`GIT_DIFF_OPTS` is a bit of a misnomer. The only valid values are `-u<n>` or `--unified=<n>`, which controls the number of context lines shown in a `git diff` command.

`GIT_EXTERNAL_DIFF` is used as an override for the `diff.external` configuration value. If it’s set, Git will invoke this program when `git diff` is invoked.

`GIT_DIFF_PATH_COUNTER` and `GIT_DIFF_PATH_TOTAL` are useful from inside the program specified by `GIT_EXTERNAL_DIFF` or `diff.external`. The former represents which file in a series is being diffed (starting with 1), and the latter is the total number of files in the batch.

`GIT_MERGE_VERBOSITY` controls the output for the recursive merge strategy. The allowed values are as follows:

- 0 outputs nothing, except possibly a single error message.
- 1 shows only conflicts.
- 2 also shows file changes.
- 3 shows when files are skipped because they haven’t changed.
- 4 shows all paths as they are processed.

- 5 and above show detailed debugging information.

The default value is 2.

Debugging

Want to *really* know what Git is up to? Git has a fairly complete set of traces embedded, and all you need to do is turn them on. The possible values of these variables are as follows:

- “true”, “1”, or “2” – the trace category is written to stderr.
- An absolute path starting with / – the trace output will be written to that file.

GIT_TRACE controls general traces, which don't fit into any specific category. This includes the expansion of aliases, and delegation to other sub-programs.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554          trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341 trace: run_command: 'git-lga'
20:12:49.879529 git.c:282          trace: alias expansion: lga => 'log' '--graph'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349          trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341 trace: run_command: 'less'
20:12:49.899675 run-command.c:192 trace: exec: 'less'
```

GIT_TRACE_PACK_ACCESS controls tracing of packfile access. The first field is the packfile being accessed, the second is the offset within that file:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088   .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088   .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088   .git/objects/pack/pack-c3fa...291e.pack 35175
# [...]
20:10:12.087398 sha1_file.c:2088   .git/objects/pack/pack-e80e...e3d2.pack
56914983
20:10:12.087419 sha1_file.c:2088   .git/objects/pack/pack-e80e...e3d2.pack
14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

GIT_TRACE_PACKET enables packet-level tracing for network operations.


```

$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46          packet:          git< # service=git-upload-
pack
20:15:14.867071 pkt-line.c:46          packet:          git< 0000
20:15:14.867079 pkt-line.c:46          packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-band side-
band-64k ofs-delta shallow no-progress include-tag multi_ack_detailed no-done
symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46          packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46          packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# [...]

```

GIT_TRACE_PERFORMANCE controls logging of performance data. The output shows how long each particular git invocation takes.

```

$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414          performance: 0.374835000 s: git command: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414          performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414          performance: 3.715349000 s: git command: 'git'
'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--
reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414          performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414          performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414          performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414          performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414          performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414          performance: 6.112217000 s: git command: 'git'
'gc'

```

GIT_TRACE_SETUP shows information about what Git is discovering about the repository and environment it's interacting with.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315          setup: git_dir: .git
20:19:47.087184 trace.c:316          setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317          setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318          setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Miscellaneous

GIT_SSH, if specified, is a program that is invoked instead of `ssh` when Git tries to connect to an SSH host. It is invoked like `$GIT_SSH [username@]host [-p <port>] <command>`. Note that this isn't the easiest way to customize how `ssh` is invoked; it won't support extra command-line parameters, so you'd have to write a wrapper script and set `GIT_SSH` to point to it. It's probably easier just to use the `~/.ssh/config` file for that.

GIT_ASKPASS is an override for the `core.askpass` configuration value. This is the program invoked whenever Git needs to ask the user for credentials, which can expect a text prompt as a command-line argument, and should return the answer on `stdout`. (See [Credential Storage](#) for more on this subsystem.)

GIT_NAMESPACE controls access to namespaced refs, and is equivalent to the `--namespace` flag. This is mostly useful on the server side, where you may want to store multiple forks of a single repository in one repository, only keeping the refs separate.

GIT_FLUSH can be used to force Git to use non-buffered I/O when writing incrementally to `stdout`. A value of 1 causes Git to flush more often, a value of 0 causes all output to be buffered. The default value (if this variable is not set) is to choose an appropriate buffering scheme depending on the activity and the output mode.

GIT_REFLOG_ACTION lets you specify the descriptive text written to the reflog. Here's an example:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'
[master 9e3d55a] my message
$ git reflog -1
9e3d55a HEAD@{0}: my action: my message
```

Povzetek

Morali bi imeti precej dobro razumevanje, kaj Git dela v ozadju in do neke mere, kako je implementiran. To poglavje je pokrilo število ukazov vodovoda - ukazi, ki so na nižjem nivoju in enostavnejši od ukazov porcelana, o katerih ste se naučili v preostanku knjige. Razumevanje delovanja Git-a na nižjem nivoju bi moralo narediti razumevanje zakaj dela kar dela enostavnejše in tudi pisati vaša lastna orodja in pomagalne skripte, da naredijo vaš določen potek dela za vas.

Git kot datotečni sistem naslavljanja vsebine je zelo močno orodje, ki ga lahko enostavno uporabite kot nekaj več kot samo VCS. Upamo, da lahko uporabljate vaše novo pridobljeno znanje notranjosti Git-a za implementacijo vaše lastne cool aplikacije te tehnologije in se počutite bolj udobno z uporabo Git-a na bolj naprednih nivojih.

Appendix A: Git v drugih okoljih

Če ste prebrali celotno knjigo, ste se naučili veliko o tem, kako uporabljati Git v ukazni vrstici. Lahko delate z lokalnimi datotekami, povežete vaš repozitorij z drugimi preko omrežja in delate učinkovito z ostalimi. Vendar zgodba se tu ne konča; Git je običajno uporabljen kot del večjih ekosistemov in terminal ni vedno najboljši način za delo z njim. Sedaj bomo pogledali nekaj drugim vrst okolj, kjer je Git lahko uporaben in kako druge aplikacij (vključno z vašo) delajo skupaj z Git-om.

Grafični vmesniki

Git materno okolje je v terminalu. Nove lastnosti se najprej prikažejo tam in samo ukazna vrstica je polna moč Git-a v celoti na razpolago. Vendar osnovni tekst ni najboljša izbira za vsa opravila; včasih je vizualna predstavitev to, kar potrebujete in nekateri uporabniki so veliko bolj udobni z vmesnikom točke in klika.

Pomembno je omeniti, da razlike med vmesniki so prilagojene z različnimi poteki dela. Nekateri klienti izpostavijo samo pazljive kuste razstave funkcionalnosti Git, da podpirajo določen način dela, ki ga avtor smatra za učinkovitega. Ko se pogleda to v tej luči, nobeno izmed teh orodij ne more biti poimenovano kot “boljše” od ostalih, saj so enostavno samo prilagojeni njihovim predvidenim namenom. Pomnite tudi, da ni ničesar v teh grafičnih klientih, kar klient ukazne vrstice ne more narediti; ukazna vrstica je še vedno, kjer boste imeli največ moči in kontrole, ko delate z vašimi repozitoriji.

gitk in git-gui

Ko namestite Git, dobite tudi njegovi vizualni orodji `gitk` in `git-gui`.

`gitk` je grafični pregledovalnik zgodovine. O njem razmišljajte kot o močni GUI lupini preko `git log` in `git grep`. To je orodje za uporabo, ko poskušate ugotoviti, kaj se je zgodilo v preteklosti ali vizualizirati zgodovino vašega projekta.

Gitk je najenostavnejši za sklicevanje iz ukazne vrstice. Naredite samo `cd` v repozitorij Git in vpišite:

```
$ gitk [git log options]
```

Gitk sprejema mnoge opcije ukazne vrstice, večina od njih je podanih preko podležeče akcije `git log`. Verjetno ena najbolj uporabnih je zastavica `--all`, ki pove gitk, da prikaže pošiljanja, ki so dosegljiva iz *kateregakoli* ref-a ne samo HEAD. Gitk-jev vmesnik izgleda takole:

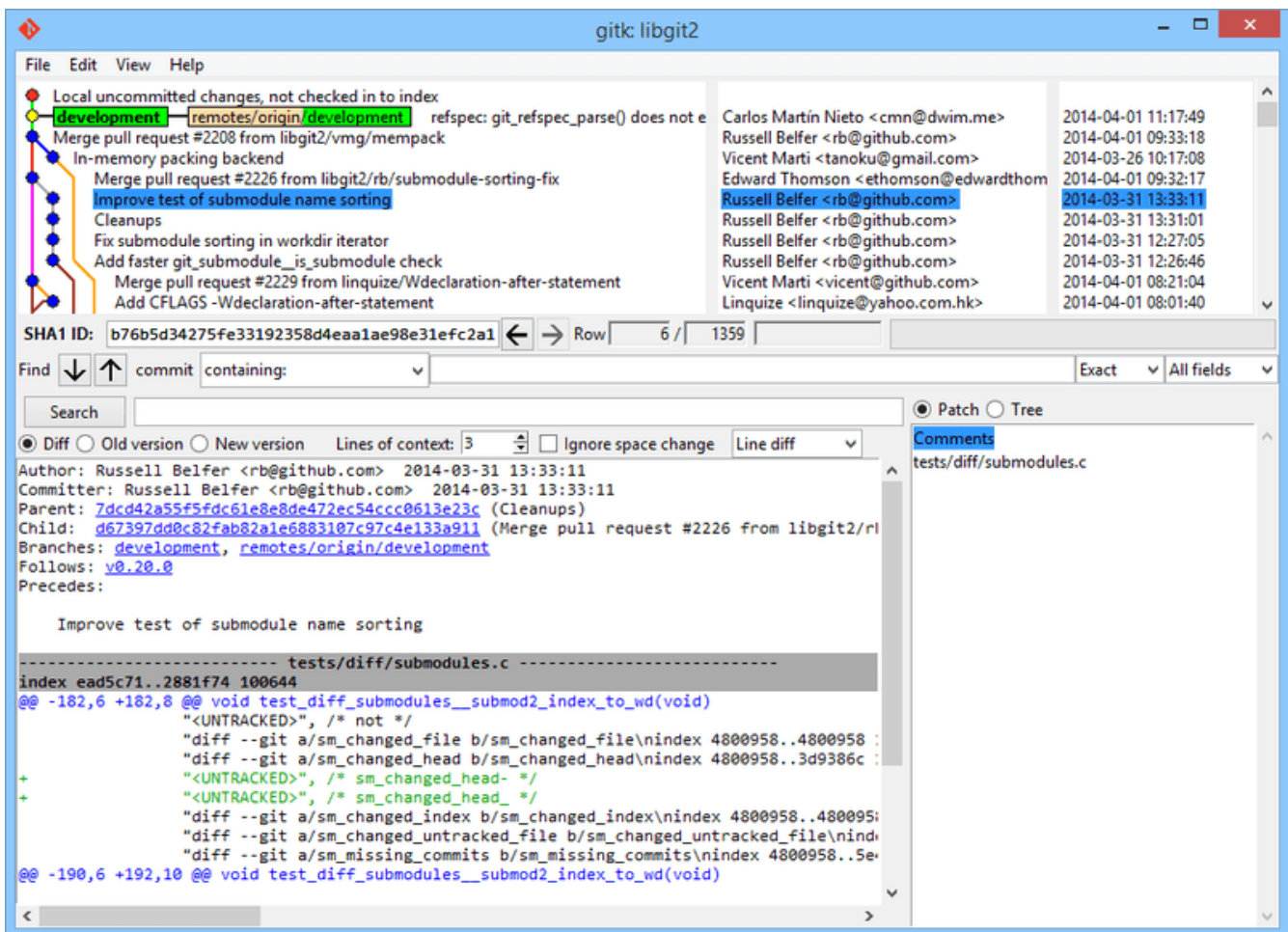


Figure 153. The `gitk` history viewer.

Na vrhu je nekaj, kar izgleda kot nek izpis `git log --graph`; vsaka pika predstavlja pošiljanje, vrstice predstavljajo starševska razmerja in ref-i so prikazani kot obarvane škatlice. Rumena pika predstavlja HEAD in rdeča pika predstavlja spremembe, ki še bodo postale pošiljanja. Na dnu je pogled izbranega pošiljanja; komentarji in popravki na levi in pogled povzetka na desni. Vmes je zbirka kontrol uporabljenih za iskanje zgodovine.

`git-gui` na drugi strani je primarno orodje za obdelovanje pošiljanj. To je tudi najenostavnejše za sklic iz ukazne vrstice:

```
$ git gui
```

In izgleda nekako takole:

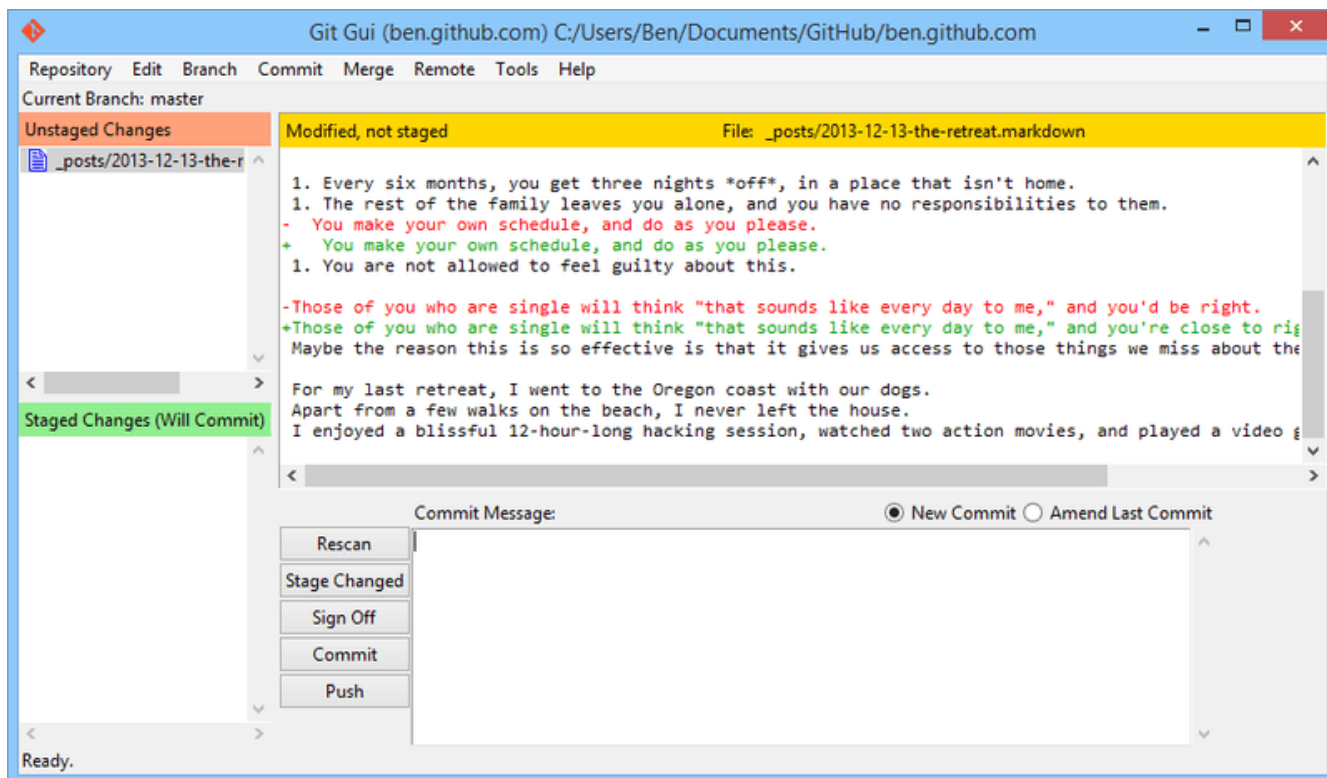


Figure 154. The `git-gui` commit tool.

Na levi je kazalo; spremembe, ki niso v vmesni fazi na vrhu; spremembe dane v vmesno fazo na dnu. Premikate lahko celotne datoteke med dvema stanji s klikom na njihove ikone ali pa lahko izberete datoteko za pogled s klikom na njeno ime.

Desno zgoraj je pogled diff, ki prikazuje spremembe za trenutno izbrano datoteko. Posamezne kupe lahko date v vmesno fazo (ali posamezne vrstice) z desnim klikom v tem področju.

Desno spodaj je sporočilo in področje akcije. Vpišite vaše sporočilo v tekstovno polje in kliknite "Commit", da naredite nekaj podobnega `git commit`. Lahko tudi izberete za spreminjanje zadnjega pošiljanja z izbiro "Amend" izbirnega gumba, ki bo posodobil področje "Staged Changes" z vsebino zadnjega pošiljanja. Nato lahko enostavno date v vmesno fazo ali date izved vmesne faze nekatere spremembe, spremenite sporočilo pošiljanja in kliknite "Commit" ponovno, da zamenjate starejše pošiljanje z novim.

`gitk` in `git-gui` so primeri orodij orientiranih na naloge. Vsako od njih je prilagojeno za določen namen (ogledovanje zgodovine in ustvarjanje pošiljanj zaporedno) in izpuščanju lastnosti, ki niso potrebne za to opravilo.

GitHub za Mac in Windows

GitHub je ustvaril dva poteku dela orientirana klienta Git: enega za Windows in enega za Mac. Ta klienta sta dober primer poteku dela orientiranih orodij - namesto izpostavljanju *vseh* funkcionalnosti Git-a, se namesto tega fokusirata na kuriran skupek pogosto uporabljenih lastnosti, ki skupaj dobro delajo. Izgledata nekako takole:

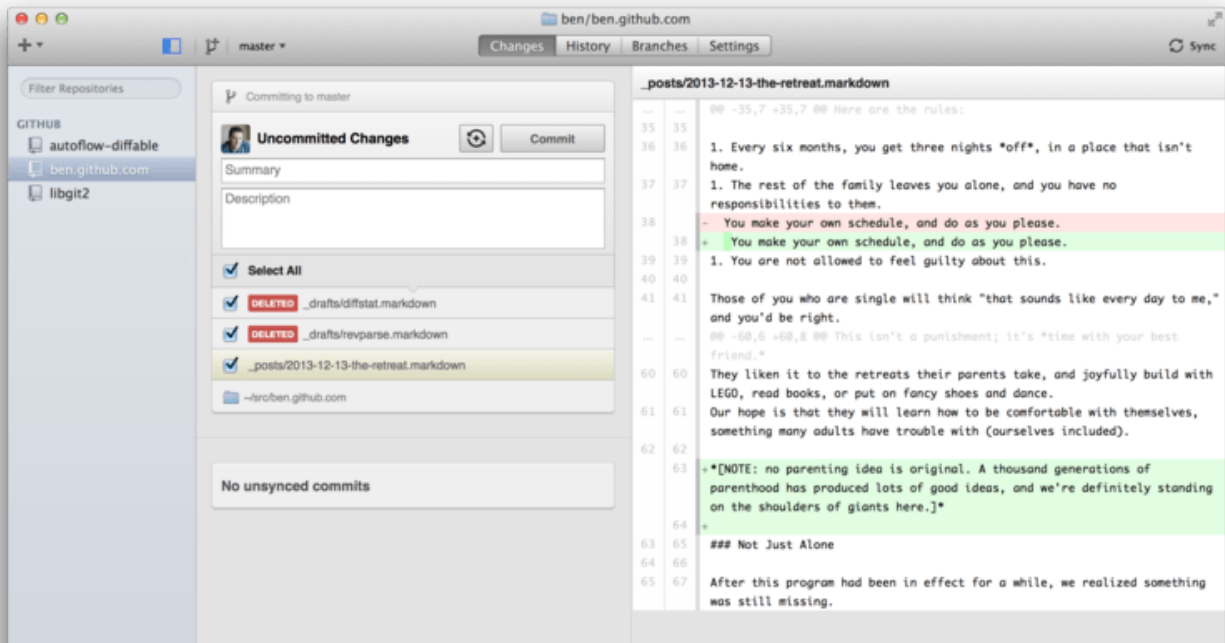


Figure 155. GitHub for Mac.

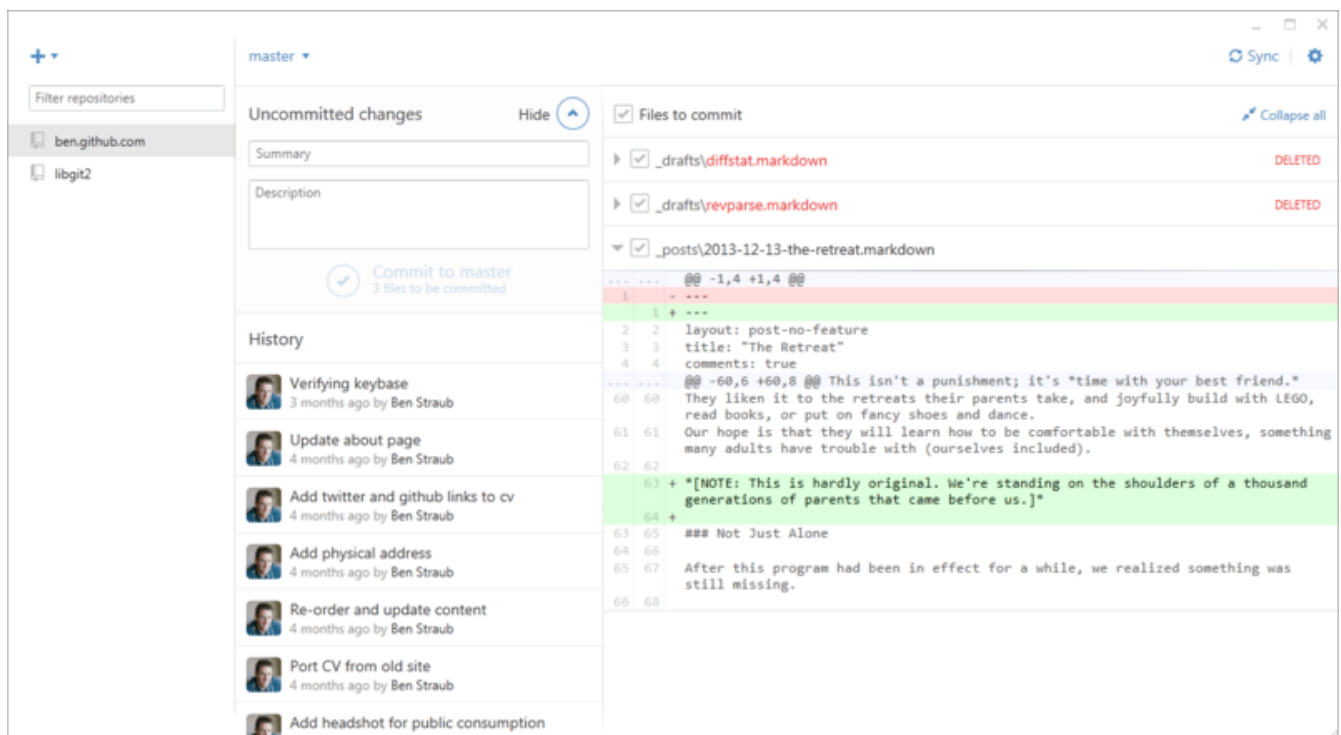


Figure 156. GitHub for Windows.

Oblikovana sta, da izgledata in delujeta zelo podobno, torej ju bomo tretirali kot en produkt v tem poglavju. Ne bomo delali podrobnostnega poteka teh orodij (imata svojo lastno dokumentacijo), vendar na hitro iti skozi pogled sprememb (kjer boste porabili večino vašega časa) je v redu.

- Na levi strani je seznam repozitorijev, katerim klient sledi; repozitorij lahko dodate (bodisi s kloniranjem ali pripetjem lokalno) s klikom na ikono “+” tega področja.
- V centru je področje vnosa pošiljanj, ki vam omoga vnesti sporočilo pošiljanja in

izbrati, katere datoteke naj bodo vključene. (Na Windows-u je zgodovina pošiljanj prikazana direktno pod tem; na Mac-u je na ločenem zavihku.)

- Na desni je pogled diff, ki prikazuje, kaj se je spremenilo v vašem delovnem direktoriju, ali katere spremembe so bile vključene v izbranem pošiljanje.
- Zadnja stvar za opaziti je gumb “Sync” zgoraj desno, ki je primarni način za interakcijo preko omrežja.

NOTE

Ne potrebujete računa GitHub za uporabo teh orodij. Medtem ko sta načrtovani posebej za storitev GitHub-a in priporočeni potek dela, bosta tudi veselo delovali s katerimkoli drugim repozitorijem in delali operacije omrežja s katerimkoli gostiteljem Git-a.

Namestitev

GitHub za Windows se lahko prenese iz <https://windows.github.com> in GitHub za Mac iz <https://mac.github.com>. Ko se aplikaciji prvič požene, vas pelje skozi vse prve nastavitve Git, kot je nastavitev vašega imena in naslova e-pošte in tako nastavi smiselne privzete vrednosti za mnogo pogostih nastavitvenih opcij, kot je predpomnenje overilnic in obnašanje CRLF.

Oboji sta “evergreen” - posodobitve so prenesene in nameščene v ozadju medtem kot so aplikacije odprte. To pomagajoče vključuje zapakirano verzijo Git-a, kar pomeni, da vam verjetno ne bo treba skrbeti o ročnem posodabljanju ponovno. Na Windows klient vključuje bližnjico za zagon Powershell-a s Posh-git, o katerem bomo govorili več kasneje v tem poglavju.

Naslednji korak je dati orodju neke repozitorije za delo. Klient vam prikaže seznam repozitorijev do katerih imate dostop na GitHub-u in jih lahko klonirate v enem koraku. Če že imate lokalni repozitorij, samo potegnite njegov direktorij iz Finder-ja ali Windows Explorer-ja v okno klienta GitHub in vključen bo v seznam repozitorijev na levi.

Priporočljiv potek dela

Enkrat ko je nameščen in nastavljen, lahko uporabite klienta GitHub za mnogo pogostih opravil Git. Namenjen potek dela za to orodje je včasih imenovan “GitHub Flow.” To pokrivamo v več podrobnostih v [The GitHub Flow](#), vendar splošno bistvo je, da (a) pošiljali boste na vejo in (b) sinhronizirali boste z oddaljenim repozitorijev dokaj redno.

Upravljanje vej je eno področij, kjer se ti dve orodji razhajata. Na Mac-u je gump na vrhu okna za ustvarjanje nove veje:

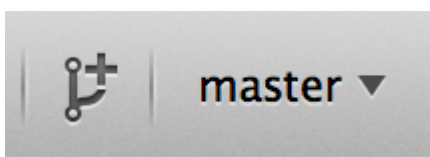


Figure 157. “Create Branch” button on Mac.

Na Windows-u je to narejeno tako, da vpišete ime nove veje v gradnik preklapljanja vej:

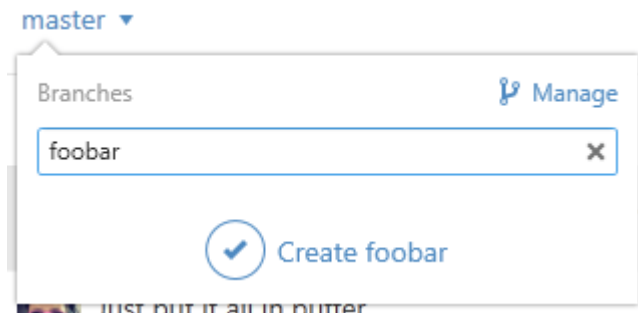


Figure 158. Creating a branch on Windows.

Ko je enkrat veja ustvarjena, je ustvarjanje novih pošiljanj dokaj enostavno. Naredite nekaj sprememb v vašem delovnem direktoriju in ko preklopite na okno klienta GitHub, vam bo prikazal katere datoteke so se spremenile. Vpišite sporočilo pošiljanja, izberite datoteke, ki bi jih želeli vključiti in kliknite na gumb “Commit” (ctrl-enter ali -enter).

Glavni način za interakcijo z ostalimi repozitoriji preko omrežja je skozi lastnost “Sync”. Git ima v notranjosti ločene operacije za porivanje, ujemanje, združevanje in ponovno baziranje vendar GitHub klienti strnejo vse te v eno več-koračno lastnost. Tu je, kaj se zgodi, ko kliknete na gumb Sync:

1. `git pull --rebase`. If this fails because of a merge conflict, fall back to `git pull --no-rebase`.
2. `git push`.

To je najbolj pogosta sekvenca ukazov omrežja, ko delate v tem stilu, tako da mečkanje v en ukaz vam prihrani veliko časa.

Povzetek

Ta orodja so zelo dobro primerna za potek dela za katerega so načrtovana. Podobni razvijalci in ne-razvijalci lahko sodelujejo na projektu v nekaj minutah in mnoge od najboljših praks za ta potek dela so zapečene v orodja. Vendar, če je vaš potek dela različen ali želite več kontrole nad tem kako in kdaj so operacije omrežja narejene, priporočamo, da uporabite drugega klienta ali ukazno vrstico.

Ostali GUI-ji

Na voljo je število ostalih grafičnih Git klientov in imajo celoten rang od specializiranih orodij z enim razlogom vse do aplikacij, ki poskušajo izpostaviti vse kar Git lahko naredi. Uradna spletna stran Git ima točen seznam najbolj popularnih klientov na <http://git-scm.com/downloads/guis>. Bolj celovit seznam je na voljo na spletni strani Git wiki na https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces.

Git v Visual Studiu

Z začetko Visual Studio 2013 Update 1, imajo uporabniki Visual Studia klient Git vgrajen direktno v njihov IDE. Visual Studio ima tudi integracijo kontrole izvirne kode

že kar nekaj časa, vendar so bili orientirani proti centraliziranim sistemom z zaklepom datotek in Git ni bila dobra opcija za ta potek dela. Podpora Git-a v Visual Studio 2013 je bila ločena iz te stare lastnosti in rezultat je veliko boljše ujemanje med Gitom in Studiem.

Da locirate lastnost, odprite projekt, ki je krmiljen z Git-om (ali samo `git init` obstoječega projekta) in izberite View > Team Explorer iz menija. Videli boste pogled "Connect", ki izgleda nekako takole:

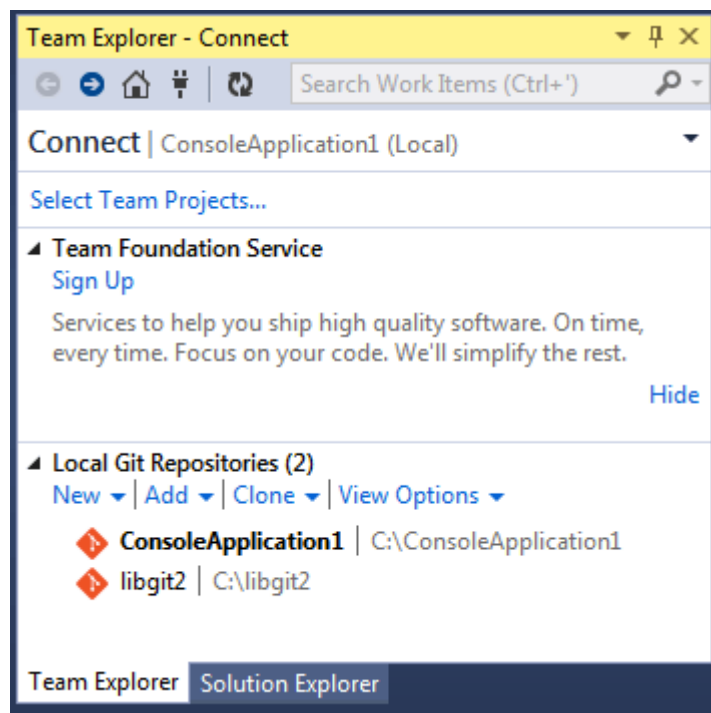


Figure 159. Connecting to a Git repository from Team Explorer.

Visual Studio si zapomni vse projekte, ki ste jih odprli in so krmiljeni z Git-om ter so na voljo v seznamu na dnu. Če tam ne vidite tistega, ki ga želite, kliknite na povezavo "Add" in vpišite pot do delovnega direktorija. Dvojni klik na enega izmed lokalnih repozitorijev Git vas popelje v pogled Home, ki izgleda kot [The "Home" view for a Git repository in Visual Studio..](#) To je središče za izvajanje akcij Git; ko *pišete* kodo, boste verjetno porabili večino vašega časa v pogledu "Changes", vendar ko pride čas za potegniti spremembe narejene s strani vaših sodelavcev, boste uporabili "Unsynced Commits" in "Branches" poglede.

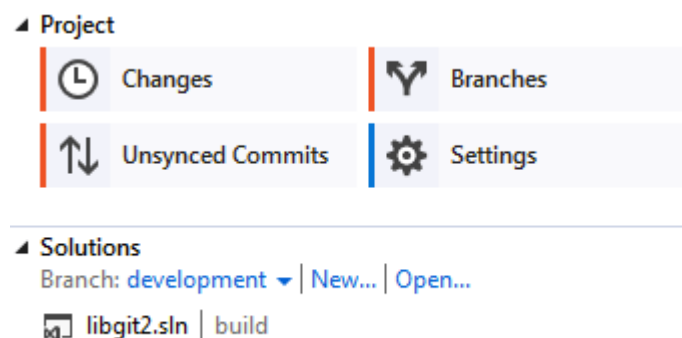


Figure 160. The "Home" view for a Git repository in Visual Studio.

Visual Studio ima seda močno nalagam-osredotočeni UI za Git. Vključuje pogled linearne

zgodovine, diff pregledovalnik, oddaljene ukaze in mnoge ostale zmožnosti. Za celotno dokumentacijo te lastnosti (ki se ne ujema tu), pojdite na <http://msdn.microsoft.com/en-us/library/hh850437.aspx>.

Git v Eclipse

Eclipse prihaja z vtičnikom imenovanim Egit, kar ponuja dokaj-celovit vmesnik k operacijam Git. Dostopano je s preklpom v perspektivo Git (Window > Open Perspective > Other... in izbiro "Git").

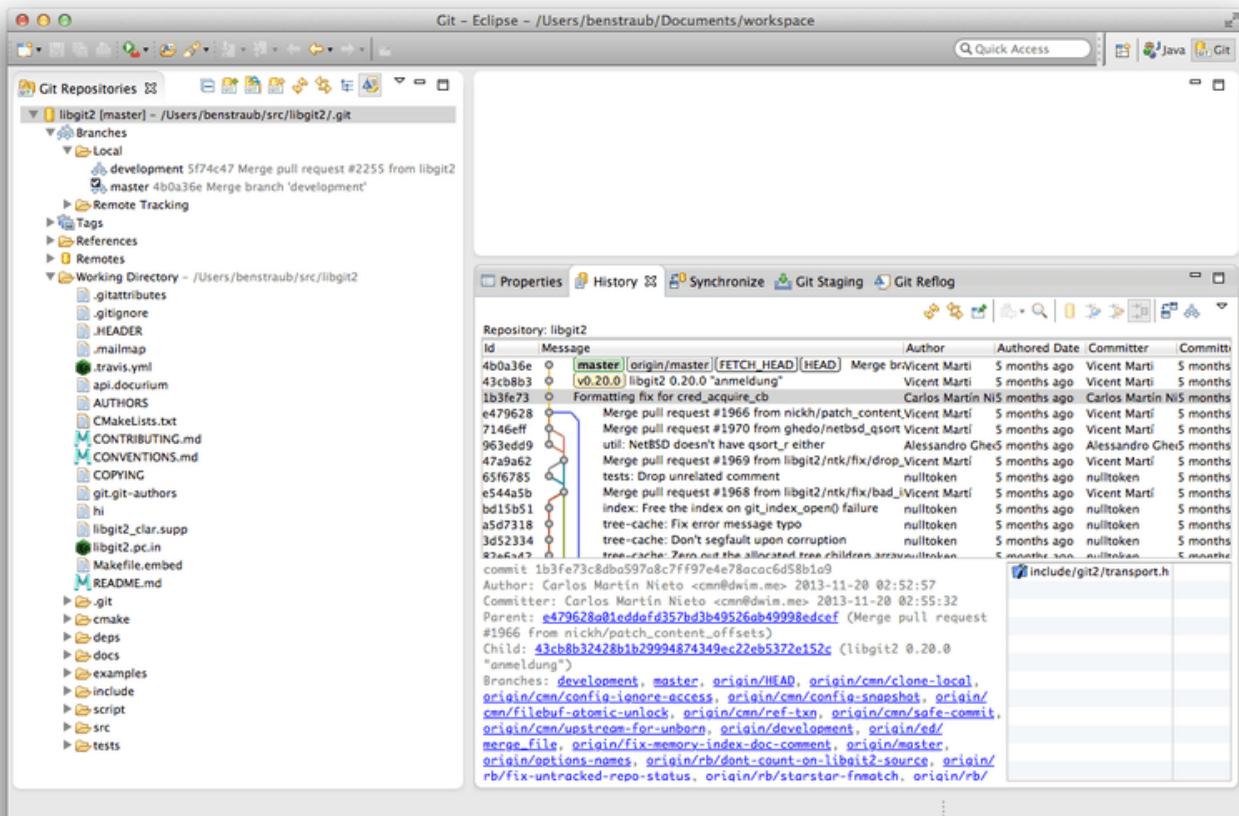


Figure 161. Eclipse's EGit environment.

Egit prihaja z veliko odlične dokumentacije, ki jo lahko najdete, če greste na Help > Help Contents in izberete "EGit Documentation" vozlišče iz seznama vsebin.

Git V Bash-u

Če ste uporabnika Bash-a, lahko izkoristite nekaj lastnosti vaše lupine, da naredite vašo izkušnjo z Git-om veliko bolj prijazno. Git dejansko prihaja z vtičniki za nekaj lupin, vendar privzeto niso vključeni.

Najprej potrebujete dobiti kopijo `contrib/completion/git-completion.bash` datoteke iz izvorne kode Git. Kopirajte jo nekam priročno, kot je vaš domači direktorij in dodajte to v vaš `.bashrc`:

```
. ~/git-completion.bash
```

Ko ste enkrat opravili, spremenite vaš direktorij v git repozitorij in vpišite:

```
$ git chec<tab>
```

... in Bash bo avtomatsko zaključeval na `git checkout`. To deluje z vsemi podukazi Git-a, parametri ukazne vrstice in daljavami ter t.i. ref imeni, kjer je ustrezno.

Uporabno je tudi prilagoditi vaš poziv ali terminal, da prikazuje informacije o trenutnem direktoriju Git. To je lahko enostavno ali kompleksno kakor želite, vendar so v splošnem nekateri deli informacij, ki jih večina ljudi želi, kot je trenutna veja in status delovnega direktorija. Da to dodate k vašemu pozivu, samo kopirajte `contrib/completion/git-prompt.sh` datoteko iz izvirnega repozitorija Git v vaš domači direktorij in dodajte nekaj takega v vaš `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(__git_ps1 " (%s)")\$ '
```

`\w` pomeni izpis trenutnega delovnega direktorija `\$` izpiše del `$` poziva in `__git_ps1 " (%s)"` pokliče funkcijo, ki je ponujena z `git-prompt.sh` z argumentom oblikovanja. Sedaj bo vaš bash poziv izgledal kot to, ko ste kjerkoli znotraj Git-kontroliranega projekta:



Figure 162. Customized `bash` prompt.

Obe teh skript prihajajo s pomagalno dokumentacijo; poglejte vsebino `git-completion.bash` in `git-prompt.sh` za več informacij.

Git v Zsh

Git tudi prihaja s knjižnico zaključevanja s tab-om za Zsh. Samo kopirajte `contrib/completion/git-completion.zsh` v vaš domači direktorij in podajte izvor v vaš `.zshrc`. Vmesnik Zsh je malenkost močnejši kot Bash-ev:

```
$ git che<tab>
check-attr      -- display gitattributes information
check-ref-format -- ensure that a reference name is well formed
checkout        -- checkout branch or paths to working tree
checkout-index  -- copy files from index to working directory
cherry          -- find commits not merged upstream
cherry-pick     -- apply changes introduced by some existing commits
```

Dvoumna zaključevanja s tabi niso samo izpisana; imajo pomagalne opise in lahko grafično navigirate po seznamu, da ponovno pritisnete tab. To deluje z ukazi Git, njegovimi argumenti in imeni stvari znotraj repozitorija (kot so ref-i in daljave) kot tudi imeni datotek in vsemi ostalimi stvarmi za katere Zsh ve, kako jih zaključiti s tabulatorjem.

Zsh je precej kompatibilen z Bash-em, kar se tiče prilagoditve poziva, vendar vam omogoča, da imate tudi poziv desne strani. Da vključite ime veje na desno stran, dodajte te vrstice v vašo datoteko `~/.zshrc`:

```
setopt prompt_subst
. ~/git-prompt.sh
export RPPROMPT='${__git_ps1 "%s"}'
```

To ima za rezultat prikaz trenutne veje na desni strani okna terminala, kadarkoli je vaša lupina znotraj repozitorija Git. Izgleda nekako takole:

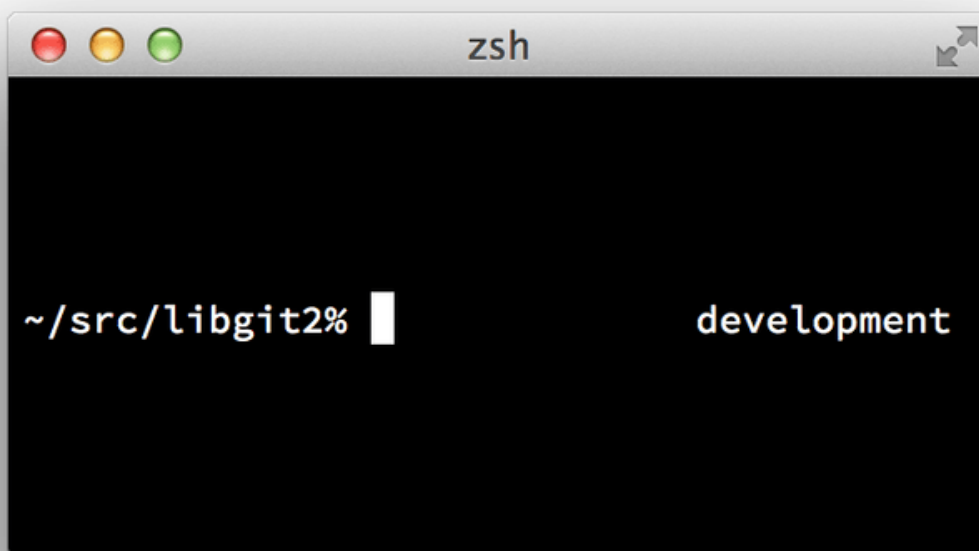


Figure 163. Customized `zsh` prompt.

Zsh je dovolj močen, saj so mu posvečena celotna ogrodja, da ga naredi boljšega. Eno izmed njih se imenuje "oh-my-zsh" in najdete ga lahko na <https://github.com/robbyrussell/oh-my-zsh>. Sistem vtičnika oh-my-zsh prihaja z močnim zaključevanjem git-a s tabulatorjem in ima sorto "tem", mnogo od njih prikazujejo podatke kontrole verzije. [An example of an oh-my-zsh theme.](#) je samo en primer, kar se lahko naredi s tem sistemom.

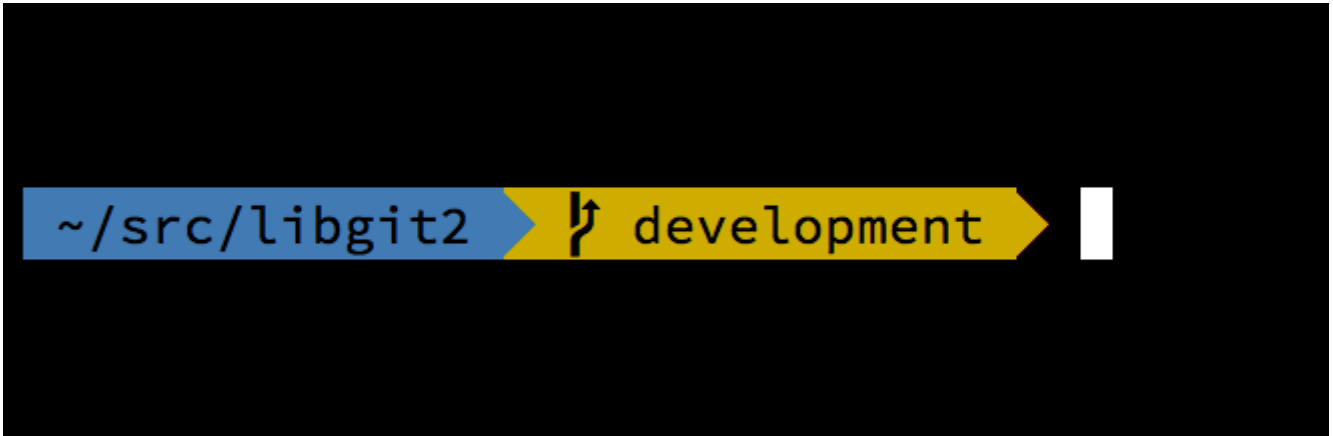


Figure 164. An example of an oh-my-zsh theme.

Git v Powershell-u

Standardni terminal ukazne vrstice na Windows-u (`cmd.exe`) ni resnično zmožen prilagojene izkušnje Git, vendar če uporabljate Powershell, imate srečo. Paket imenovan Posh-Git (<https://github.com/dahlbyk/posh-git>) ponuja močne lastnosti zaključevanja s tab-om, kot tudi okrepljeni poziv, da vam pomaga ostati na vrhu vašega statusa repozitorija. Izgleda nekako takole:

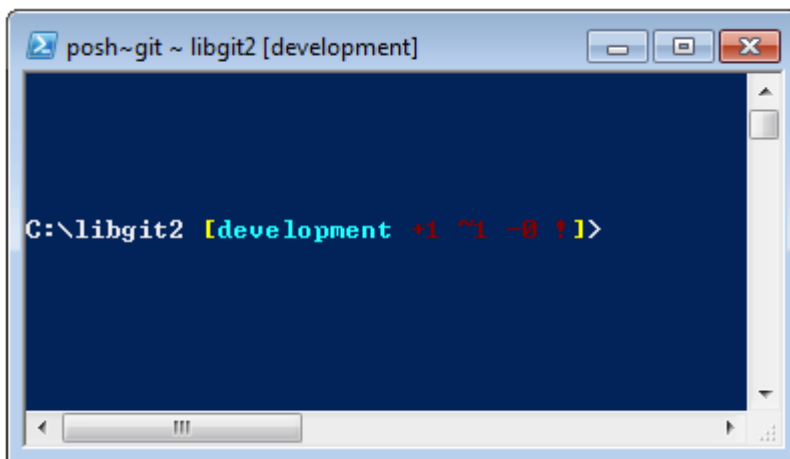


Figure 165. Powershell with Posh-git.

Če ste namestili GitHub za Windows, je Posh-Git vključen privzeto in vse kar morate narediti je dodati te vrstice v vaš `profile.ps1` (ki je običajno lociran v `C:\Users\\Documents\WindowsPowerShell`):

```
. (Resolve-Path "$env:LOCALAPPDATA\GitHub\shell.ps1")  
. $env:github_posh_git\profile.example.ps1
```

Če niste Windows uporabnik GitHub-a, samo prenesite Posh-Git izdajo iz (<https://github.com/dahlbyk/posh-git>) in jo razširite v direktorij `WindowsPowerShell`. Nato odprite Powershell poziv kot administrator in naredite sledeče:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser -Confirm  
> cd ~\Documents\WindowsPowerShell\posh-git  
> .\install.ps1
```

To bo dodalo ustrezno vrstico v vašo datoteko `profile.ps1` in posh-git bo aktiven naslednjič, ko odprete vaš poziv.

Povzetek

Naučili ste se, kako izkoristiti moč Git-a znotraj orodij, ki jih uporabljate med vašim vsakodnevnim delom in tudi kako dostopati do repozitorijev Git iz vaših lastnih programov.

Appendix B: Vključevanje Git-a v vašo aplikacijo

Če je vaša aplikacija za razvijalce, so dobre možnosti, da lahko pridobi koristi iz integracije s kontrolo izvirne kode. Tudi ne-razvijalske aplikacije kot so urejevalniki dokumentov, lahko potencialno koristijo iz lastnosti kontrole verzij in Git-ov model deluje zelo dobro za mnoge različne scenarije.

Če potrebujete integrirati Git z vašo aplikacijo imate v osnovi tri opcije: drstenje lupine in uporabo orodja Git ukazne vrstice; Libgit2; in JGit.

Git v ukazni vrstici

Ena opcija je drstenje procesa lupine in uporaba orodja Git ukazne vrstice za opravljanje dela. To ima prednosti biti kanončno in imeti vse lastnosti Git-a podprte. To je tudi precej enostavno, saj večina okolij pognanih v zagonu ima relativno enostavne objekte za klicanje procesa z argumenti ukazne vrstice. Vendar ta pristop ima nekaj slabosti.

Ena izmed teh je izpis preprostega teksta. To pomeni, da boste morali prevajati Git-ov občasno spreminjajočo se obliko izpisa za branje napredka in informacij rezultata, kar je lahko neefektivno in lahko prihaja do napak.

Druga je pomanjkanje okrevanja od napak. Če je repozitorij nekako pokvarjen ali ima uporabnik napačne vrednosti nastavitev, bo Git enostavno zavrnil izvajati mnogo operacij.

Še ena je upravljanje procesa. Git zahteva, da vzdržujete lupino okolja na ločenem procesu, kar lahko doda nezaželeno kompleksnost. Poskušanje koordinacije mnogo teh procesov (posebej ko potencialno dostopanje do istega repozitorija iz nekaj procesov) je lahko precej zahtevno.

Libgit2

© Druga opcija na vašem dosegu je uporabiti Libgit2. Libgit2 je neodvisna implementacija Git-a s fokusom na imetju lepega API-ja za uporabo znotraj ostalih programov. Lahko ga najdete na <http://libgit2.github.com>.

Najprej pogledjmo, kako C API izgleda.

Tu je viharen ogled:


```

// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);

```

Prvih nekaj vrstic odpre repozitorij Git. Tip `git_repository` predstavlja ravnanje repozitorija s predpomnilnikom v spominu. To je najenostavnejša metoda kot veste točno pot delovnega direktorija repozitorija ali direktorij `.git`. Obstaja tudi `git_repository_open_ext`, ki vključuje opcije za iskanje, `git_clone` in prijatelje za izdelavo lokalnega klona oddaljenega repozitorija in `git_repository_init` za izdelavo celotnega novega repozitorija.

Drug kos kode uporablja `rev-parse` sintakso (glejte [Branch References](#) za več o tem), da dobi pošiljanje na katerega HEAD eventuelno kaže. Vrnjeni tip je kazalec `git_object`, ki predstavlja nekaj, kar obstaja v objektni podatkovni bazi Git-a za repozitorij. `git_object` je dejansko “parent” tip za nekaj različnih vrst objekta; postavitve spomina za vsakega od tipov “child” je enak za `git_object`, tako da lahko varno oddate pravega. V tem primeru bi `git_object_type(commit)` vrnil `GIT_OBJ_COMMIT`, torej je varno oddati `git_commit` kazalec.

Naslednji kos prikazuje dostop do lastnosti pošiljanja. Zadnja vrstica tu uporablja tip `git_oid`; to je predstavitev Libgit2 za razpršitev SHA-1.

Iz tega primera se je pričelo pojavljati nekaj vzorcev:

- Če določite kazalec in podate referenco nanj v klicu Libgit2, bo ta klic verjetno vrnil kodo napake celega števila. Vrednost `0` indicira uspeh; karkoli manjšega je napaka.
- Če Libgit2 zapolni kazalec za vas, ste odgovorni za njegovo izpustitev.
- Če Libgit2 vrne kazalec `const` iz klica, vam ga ni treba izpustiti, vendar bo postal neveljaven, ko je objekt, kateremu pripada, izpuščen.
- Pisanje C-ja je nekoliko mučno.

Zadnja pomeni, da ni zelo verjetno, da boste pisali C, ko uporabljate Libgit2. Na srečo je na voljo število jezikom-specifičnih vezav, ki naredijo precej enostavno za delo z

rezpozitoriji Git iz vašega določenega jezika in okolja. Poglejmo zgornji primer napisan z vezavami Ruby za Libgit2, ki so poimenovane Rugged in lahko najdene na <https://github.com/libgit2/rugged>.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

Kot lahko vidite, je koda veliko manj v neredu. Najprej Rugged uporablja izjeme; lahko vrne stvari kot so `ConfigError` ali `ObjectError` za signalizacijo pogojev napak. Drugič ni nobenega eksplicitnega izpiščanja izvorov, odkar Ruby zbira odpadke. Poglejmo nekoliko bolj kompliciran primer: obdelovanje pošiljanja od začetka

```
blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
  :email => "bob@example.com",
  :name => "Bob User",
  :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
  :tree => index.write_tree(repo), ③
  :author => sig,
  :committer => sig, ④
  :message => "Add newfile.txt", ⑤
  :parents => repo.empty? ? [] : [ repo.head.target ].compact, ⑥
  :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧
```

- ① Ustvarite nov blog, ki vsebuje vsebno nove datoteke.
- ② Zapolnite index z drevesom pošiljanja glave in dodajte novo datoteko v pot `newfile.txt`.
- ③ To ustvari novo drevo v ODB in ga uporablja za novo pošiljanje.
- ④ Uporabljamo enak podpis za tako avtorja kot tudi za polja pošiljanja.
- ⑤ Sporočilo pošiljanja.
- ⑥ Ko se ustvarja pošiljanje, morate določiti nove starše pošiljanja. To uporablja nasvet od HEAD-a za enega starša.
- ⑦ Rugged (in Libgit2) lahko opcijsko posodobi referenco, ko dela pošiljanje.

- ⑧ Vrtnjena vrednost je zgostitev SHA-1 novega objekta pošiljanja, kar lahko potem uporabite, da dobite objekt `Commit`.

Koda Ruby je lepa in čista vendar odkar Libgit2 dela težko dvigovanje, se bo ta koda tudi poganjala hitro. Če niste rubist, se bomo dotaknili nekaterih ostalih povezav v [Ostale vezave](#).

Napredna funkcionalnost

Libgit2 ima nekaj zmožnosti, ki so izven obsega jedra Git. En primer je možnost vtičnikov: Libgit2 vam omogoča ponujati prilagojena “ozadja” za nekaj tipov operacij, tako da lahko shranite stvari na različne načine, ki jih goli Git počne. Libgit2 omogoča prilagojena ozadja za nastavitve, ref shranjevanje in podatkovno bazo objektov med drugimi stvarmi.

Poglejmo, kako to deluje. Koda spodaj je sposojena iz skupka primerov ozadja ponujenih s strani ekipe Libgit2 (kar je moč najti na <https://github.com/libgit2/libgit2-backends>). Tu je, kako je nastavljeno prilagojeno ozadje za objektno podatkovno bazo:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*...*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(odb); ④
```

(Pomnite, da napake so ujete vendar niso upravljane. Upamo, da je vaša koda boljša od naše.)

- ① Inicializacija prazne objektno podatkovne baze (ODB) “ospredje,” ki se bo obnašalo kot kontejner za “ozadja”, ki so tista, ki delajo pravo delo.
- ② Inicializacija prilagojenega ODB ozadja.
- ③ Dodajanje ozadnja k ospredju.
- ④ odpiranje repozitorija in njegova nastavitve, da uporablja našo ODB za iskanje objektov.

Vendar kaj je ta stvar `git_odb_backend_mine`? Torej, konstruktor za vašo lastno ODB implementacijo in lahko delate karkoli želite tam, dokler ustrezno zapolnjujete strukturo v `git_odb_backend`. Tu je, kako *bi* lahko izgledal:

```

typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*...*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = ...;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // ...

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}

```

Subtilna omejitev tu je, da mora biti prvi član `my_backend_struct`-a struktura `git_odb_backend`; to zagotavlja, da je postavitve spomina to, kar Libgit2 pričakuje, da je. Preostanek je arbitraren; ta struktura je lahko tako velika ali majhna, kakor jo potrebujete.

Funkcija inicializacije dodeli nekaj spomina za strukturo, nastavi kontekst po meri in nato zapolni člane strukture `parent`, ki jo podpira. Poglejmo datoteko `include/git2/sys/odb_backend.h` v izvoru Libgit2 za celoten skupek podpisov klica; vaš določen primer uporabe vam bo pomagal določiti, katerega od teh boste želeli podpirati.

Ostale vezave

Libgit2 ima vezave za mnogo jezikov. Tu bomo pokazali majhen primer, ki uporablja nekaj od bolj celovitih vezav paketov od tega pisanja; knjižnjice obstojajo za mnoge ostale jezike, vključno C++, Go, Node.js, Erlang in JVM vse v različnih faza zrelosti. Uradno zbirko vezav se lahko najde z brskanjem po repozitorijih na <https://github.com/libgit2>. Koda, ki jo boste pisali, bo vrnila sporočilo pošiljanja iz pošiljanja, ki eventuelno kaže na HEAD (neke vrste `git log -1`).

LibGit2Sharp

Če pišete .NET ali Mono aplikacijo, je Libgit2Sharp (<https://github.com/libgit2/libgit2sharp>) to, kar iščete. Vezave so napisane v C# in veliko skrbnosti je bilo dane za ovitje surovih klicev Libgit2 z nativnim občutkom CLR API-jev. Tu je, kako izgleda naš primer

programa:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Za namizne Windows aplikacije je celo paket NuGet, ki vam bo pomagal pričeti hitro.

objective-git

Če se vaša aplikacija poganja na platformi Apple, verjetno uporabljate objektivni C kot vaš jezik implementacije. Objektivni Git (<https://github.com/libgit2/objective-git>) je ime vezave Libgit2 za to okolje. Primer programa izgleda takole:

```
GTRepository *repo =  
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]  
    error:NULL];  
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objektivni git je polno interoperabilen s Swift-om, torej se ne bojte, če ste izpustili objektni C zadaj.

pygit2

Vezave za Libgit2 v Python-u so imenovane Pygit2 in se jih lahko najde na <http://www.pygit2.org/>. Naš primer programa:

```
pygit2.Repository("/path/to/repo") # open repository  
.head # get the current branch  
.peel(pygit2.Commit) # walk down to the commit  
.message # read the message
```

Nadaljnje branje

Seveda polno obravnavanje zmožnosti Libgit2 je izven obsega te knjige. Če želite več informacij o samem Libgit2, je na voljo dokumentacija API na <https://libgit2.github.com/libgit2> in skupek vodičev na <https://libgit2.github.com/docs>. Za ostale povezave preverite zapakiran README in teste; tam so pogostokrat na voljo majhni vodiči in kazalci k nadaljnem branju.

JGit

Če želite uporabljati Git znotraj programa Java, je na voljo lastnosti polna knjižnica Git imenovana JGit. JGit je relativno lastnosti polna implementacija Git-a napisanega izvorno v Javi in je široko uporabljena v skupnosti Java. Projekt JGit je pod okriljem Eclipse in njegov dom je moč najti na <http://www.eclipse.org/jgit>.

Nastavitve

Na voljo je število načinov za povezavo vašega projekta z JGit in začetkom pisanja kode pod njim. Verjetno najenostavnejši je uporaba Maven-a - integracija je dosežena z dodajanjem sledečih odrezkov znački ``<dependencies>`` v vaši datoteki `pom.xml`:

```
<dependency>
  <groupId>org.eclipse.jgit</groupId>
  <artifactId>org.eclipse.jgit</artifactId>
  <version>3.5.0.201409260305-r</version>
</dependency>
```

`version` se bo najverjetneje povečevala tekom časa, ko to berete; preverite <http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit> za nalaganje informacij repozitorija. Ko je to enkrat narejeno bo Maven avtomatično zahteval in uporabljal knjižnice JGit, ki jih boste potrebovali.

Če bi raje upravljali binarne odvisnosti sami, so vnaprej zgrajene zagonske datoteke JGit na voljo iz <http://www.eclipse.org/jgit/download>. Zgradite jih lahko v vaš projekt s pogonom ukaza sledeče:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

Vodovod

JGit ima dva osnovna nivoja API-ja: vodovod in keramika. Terminologija za to dvojje prihaja iz samega Git-a in JGit je razdeljen v groben iste vrste področij: porcelan API-ji so prijazno ospredje za pogoste akcije na nivoju uporabnika (vrsta stvari, ki bi jih običajni uporabnik uporabil za orodje ukazne vrstice Git) medtem ko vodovod API-ji so za interakcijo z objekti repozitorija nižjega nivoja direktno.

Začetna točka za večino sej JGit je razred `Repository` in prva stvar, ki jo boste želeli narediti, da ustvarite instanco iz njega. Za repozitorij na osnovi datotečnega sistema (da, JGit omoogoča ostale modele shranjevanja) je to urejeno z uporabo `FileRepositoryBuilder`:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

Graditelj ima učinkovit API za ponujanje vseh stvari, ki jih potrebuje, da najde repozitorij Git, bodisi ali ne vaš program točno ve, kje je lociran. Uporablja lahko spremenljivke okolja (`.readEnvironment()`), začne iz mesta v delovnem direktoriju in išče (`.setWorkTree(...).findGitDir()`), ali pa samo odpre znani direktorij `.git` kot zgoraj.

Enkrat ko imate instanco `Repository`, lahko z njim naredite vse vrste stvari. Tu je hitro vzorčenje:

```
// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");
```

Tu se kar veliko dogaja, torej pojdemo skozi po eno sekcijo na enkrat.

Prva vrstica dobi kazalec na referenco `master`. JGit avtomatično vzame *dejanski* master ref, ki domuje v `refs/heads/master` in vrne objekt, ki vam omogoča ujeti informacije o referenci. Dobite lahko ime (`getName()`) in bodisi tarčo objekta direktne reference (`getObjectId()`) ali referenco, ki kaže na simbolični ref (`getTarget()`). Objekti ref so tudi uporabljeni, da predstavljajo reference značk in objektov, tako da lahko vprašate, če je značka “peeled” - olupljena, kar pomeni, da kaže na končno tarčo (potencialno dolgega) niza objektov značke.

Druga vrstica dobi tarčo reference `master`, ki je vrnjena kot instanca `ObjectId`. `ObjectId` predstavlja SHA-1 razpršitev objekta, ki lahko ali ne obstaja v podatkovni bazi objekta Git. Tretja vrstica je podobna, vendar prikazuje kako JGit upravlja s sintakso `rev-parse` (za več o tem, glejte [Branch References](#)); lahko podate katerokoli določilo objekta, ki ga

Git razume in JGit bo vrnil bodisi veljavni `ObjectId` za ta objekt ali `null`.

Naslednji dve vrstici prikazujeta kako naložiti golo vsebino objekta. V tem primeru kličemo `ObjectLoader.copyTo()`, da pretok vsebine objekta direktno v `stdout`, vendar `ObjectLoader` ima tudi metode za pisanje tipa in velikosti objekta kot tudi vrne bajtno polje. Za velike objekte (kjer `.isLarge()` vrne `true`), lahko kličete `.openStream()`, da dobite `InputStream`-u podoben objekt, ki lahko prebere surovi objekt podatkov brez, da potegne vse v spomin naenkrat.

Naslednjih nekaj vrstic prikazuje, kaj vzame, da ustvari novo vejo. Ustvarimo instanco `RefUpdate`, nastavimo nekaj parametrov in kličemo `.update()`, da sprožimo spremembo. Direktno temu kar sledi, je koda za izbris te iste veje. Pomnite, da je `.setForceUpdate(true)` obvezen, da to deluje; drugače bo `.delete()` klic vrnil `REJECTED` in nič se ne bo zgodilo.

Zadnji primer prikazuje, kako ujeti `user.name` vrednost iz nastavitvenih datotek Git. Instanca `Config` uporablja repozitorij, ki smo ga odprli prej za lokalne nastavitve vendar bo avtomatično odkril globalne in sistemske nastavitvene datoteke in prebral vrednosti tudi iz njih.

To je samo majhen primer celotnega vodovodnega API-ja; na voljo je veliko več metod in razredov. Tudi kar ni prikazano tu, je način, kako JGit upravlja z napakami, kar je skozi uporabo izjem. API-ji JGit-a včasih vržejo standardne Java izjeme (kot je `IOException`), vendar so gostitelji JGIT določenih tipov izjem, ki so tudi ponujene (kot so `NoRemoteRepositoryException`, `CorruptObjectException` in `NoMergeBaseException`).

Porcelan

API-ji vodovoda so nekoliko zaključeni, vendar jih je lahko težavno nízati skupaj, da se doseže skupne cilje, kot je dodajanje datoteke indeks-u ali ustvarjanje novega pošiljanja. JGit ponuja skupek višje-nivojskih API-jev, da vam pri tem pomaga in vnosna točka tem API-je je razred `Git`:

```
Repository repo;  
// construct repo...  
Git git = new Git(repo);
```

Razred `Git` ima lep skupek visoko-nivojskih *builder*-stilskih metod, ki so lahko uporabljene za konstrukcijo nekega lepega kompleksnega obnašanja. Poglejmo primer - narediti nekaj kot je `git ls-remote`:


```

CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",
"p4ssw0rd");
Collection<Ref> remoteRefs = git.lsRemote()
    .setCredentialsProvider(cp)
    .setRemote("origin")
    .setTags(true)
    .setHeads(false)
    .call();
for (Ref ref : remoteRefs) {
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());
}

```

To je pogosti vzorec z razredom `Git`; metode vrnejo ukaz objektov, ki vam omogoča verižiti klice metod, da nastavijo parametre, ki so izvršene, ko kličete `.call()`. V tem primeru sprašujemo daljavo `origin` za oznage, vendar ne glave. Opazite tudi uporabo objekta `CredentialsProvider` za overitev.

Mnogi ostali ukazi so na voljo preko razreda `Git`, vključno vendar ne omejeno na `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert` in `reset`.

Nadaljnje branje

To je samo majhen primer JGit-ove polne zmožnosti. Če vas zanima in želite izvedeti več, pogledjte tu za informacije in inspiracijo:

- Uradna JGit API dokumentacija je na voljo na spletu na <https://www.eclipse.org/jgit/documentation>. To so standardni Javadoc, tako da vaš priljubljeni JVM IDE jih bo tudi zmožen namestiti lokalno.
- JGit knjiga receptov na <https://github.com/centic9/jgit-cookbook> ima mnoge primere, kako narediti določena opravila z JGit-om.

Appendix C: Git Commands

Throughout the book we have introduced dozens of Git commands and have tried hard to introduce them within something of a narrative, adding more commands to the story slowly. However, this leaves us with examples of usage of the commands somewhat scattered throughout the whole book.

In this appendix, we'll go through all the Git commands we addressed throughout the book, grouped roughly by what they're used for. We'll talk about what each command very generally does and then point out where in the book you can find us having used it.

Setup and Config

There are two commands that are used quite a lot, from the first invocations of Git to common every day tweaking and referencing, the `config` and `help` commands.

`git config`

Git has a default way of doing hundreds of things. For a lot of these things, you can tell Git to default to doing them a different way, or set your preferences. This involves everything from telling Git what your name is to specific terminal color preferences or what editor you use. There are several files this command will read from and write to so you can set values globally or down to specific repositories.

The `git config` command has been used in nearly every chapter of the book.

In [Prva namestitev Git-a](#) we used it to specify our name, email address and editor preference before we even got started using Git.

In [Git aliasi](#) we showed how you could use it to create shorthand commands that expand to long option sequences so you don't have to type them every time.

In [Ponovno baziranje \(rebasing\)](#) we used it to make `--rebase` the default when you run `git pull`.

In [Credential Storage](#) we used it to set up a default store for your HTTP passwords.

In [Keyword Expansion](#) we showed how to set up smudge and clean filters on content coming in and out of Git.

Finally, basically the entirety of [Git Configuration](#) is dedicated to the command.

`git help`

The `git help` command is used to show you all the documentation shipped with Git about any command. While we're giving a rough overview of most of the more popular ones in this appendix, for a full listing of all of the possible options and flags for every command, you can always run `git help <command>`.

We introduced the `git help` command in [Pridobitev pomoči](#) and showed you how to use it to find more information about the `git shell` in [Nastavitev strežnika](#).

Getting and Creating Projects

There are two ways to get a Git repository. One is to copy it from an existing repository on the network or elsewhere and the other is to create a new one in an existing directory.

`git init`

To take a directory and turn it into a new Git repository so you can start version controlling it, you can simply run `git init`.

We first introduce this in [Pridobitev repozitorija Git](#), where we show creating a brand new repository to start working with.

We talk briefly about how you can change the default branch from “master” in [Oddaljene veje](#).

We use this command to create an empty bare repository for a server in [Dajanje golega repozitorija na strežnik](#).

Finally, we go through some of the details of what it actually does behind the scenes in [Napeljava in porcelan](#).

`git clone`

The `git clone` command is actually something of a wrapper around several other commands. It creates a new directory, goes into it and runs `git init` to make it an empty Git repository, adds a remote (`git remote add`) to the URL that you pass it (by default named `origin`), runs a `git fetch` from that remote repository and then checks out the latest commit into your working directory with `git checkout`.

The `git clone` command is used in dozens of places throughout the book, but we’ll just list a few interesting places.

It’s basically introduced and explained in [Kloniranje obstoječega repozitorija](#), where we go through a few examples.

In [Pridobiti Git na strežnik](#) we look at using the `--bare` option to create a copy of a Git repository with no working directory.

In [Bundling](#) we use it to unbundle a bundled Git repository.

Finally, in [Cloning a Project with Submodules](#) we learn the `--recursive` option to make cloning a repository with submodules a little simpler.

Though it’s used in many other places through the book, these are the ones that are somewhat unique or where it is used in ways that are a little different.

Basic Snapshotting

For the basic workflow of staging content and committing it to your history, there are only a few basic commands.

git add

The `git add` command adds content from the working directory into the staging area (or “index”) for the next commit. When the `git commit` command is run, by default it only looks at this staging area, so `git add` is used to craft what exactly you would like your next commit snapshot to look like.

This command is an incredibly important command in Git and is mentioned or used dozens of times in this book. We’ll quickly cover some of the unique uses that can be found.

We first introduce and explain `git add` in detail in [Tracking New Files](#).

We mention how to use it to resolve merge conflicts in [Basic Merge Conflicts](#).

We go over using it to interactively stage only specific parts of a modified file in [Interactive Staging](#).

Finally, we emulate it at a low level in [Tree Objects](#), so you can get an idea of what it’s doing behind the scenes.

git status

The `git status` command will show you the different states of files in your working directory and staging area. Which files are modified and unstaged and which are staged but not yet committed. In its normal form, it also will show you some basic hints on how to move files between these stages.

We first cover `status` in [Preverjanje status vaših datotek](#), both in its basic and simplified forms. While we use it throughout the book, pretty much everything you can do with the `git status` command is covered there.

git diff

The `git diff` command is used when you want to see differences between any two trees. This could be the difference between your working environment and your staging area (`git diff` by itself), between your staging area and your last commit (`git diff --staged`), or between two commits (`git diff master branchB`).

We first look at the basic uses of `git diff` in [Ogled vaših sprememb v vmesni fazi in izven vmesne faze](#), where we show how to see what changes are staged and which are not yet staged.

We use it to look for possible whitespace issues before committing with the `--check` option in [Smernice pošiljanja](#).

We see how to check the differences between branches more effectively with the `git diff A...B` syntax in [Determining What Is Introduced](#).

We use it to filter out whitespace differences with `-b` and how to compare different stages of conflicted files with `--theirs`, `--ours` and `--base` in [Advanced Merging](#).

Finally, we use it to effectively compare submodule changes with `--submodule` in [Starting with Submodules](#).

git difftool

The `git difftool` command simply launches an external tool to show you the difference between two trees in case you want to use something other than the built in `git diff` command.

We only briefly mention this in [Ogled vaših sprememb v vmesni fazi in izven vmesne faze](#).

git commit

The `git commit` command takes all the file contents that have been staged with `git add` and records a new permanent snapshot in the database and then moves the branch pointer on the current branch up to it.

We first cover the basics of committing in [Pošiljanje vaših sprememb](#). There we also demonstrate how to use the `-a` flag to skip the `git add` step in daily workflows and how to use the `-m` flag to pass a commit message in on the command line instead of firing up an editor.

In [Razveljavljanje stvari](#) we cover using the `--amend` option to redo the most recent commit.

In [Veje na kratko](#), we go into much more detail about what `git commit` does and why it does it like that.

We looked at how to sign commits cryptographically with the `-S` flag in [Signing Commits](#).

Finally, we take a look at what the `git commit` command does in the background and how it's actually implemented in [Commit Objects](#).

git reset

The `git reset` command is primarily used to undo things, as you can possibly tell by the verb. It moves around the `HEAD` pointer and optionally changes the `index` or staging area and can also optionally change the working directory if you use `--hard`. This final option makes it possible for this command to lose your work if used incorrectly, so make sure you understand it before using it.

We first effectively cover the simplest use of `git reset` in [Povrnitev datoteke iz vmesne](#)

[faze](#), where we use it to unstage a file we had run `git add` on.

We then cover it in quite some detail in [Reset Demystified](#), which is entirely devoted to explaining this command.

We use `git reset --hard` to abort a merge in [Aborting a Merge](#), where we also use `git merge --abort`, which is a bit of a wrapper for the `git reset` command.

git rm

The `git rm` command is used to remove files from the staging area and working directory for Git. It is similar to `git add` in that it stages a removal of a file for the next commit.

We cover the `git rm` command in some detail in [Odstranjevanje datotek](#), including recursively removing files and only removing files from the staging area but leaving them in the working directory with `--cached`.

The only other differing use of `git rm` in the book is in [Removing Objects](#) where we briefly use and explain the `--ignore-unmatch` when running `git filter-branch`, which simply makes it not error out when the file we are trying to remove doesn't exist. This can be useful for scripting purposes.

git mv

The `git mv` command is a thin convenience command to move a file and then run `git add` on the new file and `git rm` on the old file.

We only briefly mention this command in [Premikanje datotek](#).

git clean

The `git clean` command is used to remove unwanted files from your working directory. This could include removing temporary build artifacts or merge conflict files.

We cover many of the options and scenarios in which you might use the clean command in [Cleaning your Working Directory](#).

Branching and Merging

There are just a handful of commands that implement most of the branching and merging functionality in Git.

git branch

The `git branch` command is actually something of a branch management tool. It can list the branches you have, create a new branch, delete branches and rename branches.

Most of [Veje Git](#) is dedicated to the `branch` command and it's used throughout the entire chapter. We first introduce it in [Ustvarjanje nove veje](#) and we go through most

of its other features (listing and deleting) in [Upravljanje vej](#).

In [Sledenje vej](#) we use the `git branch -u` option to set up a tracking branch.

Finally, we go through some of what it does in the background in [Git References](#).

git checkout

The `git checkout` command is used to switch branches and check content out into your working directory.

We first encounter the command in [Switching Branches](#) along with the `git branch` command.

We see how to use it to start tracking branches with the `--track` flag in [Sledenje vej](#).

We use it to reintroduce file conflicts with `--conflict=diff3` in [Checking Out Conflicts](#).

We go into closer detail on its relationship with `git reset` in [Reset Demystified](#).

Finally, we go into some implementation detail in [The HEAD](#).

git merge

The `git merge` tool is used to merge one or more branches into the branch you have checked out. It will then advance the current branch to the result of the merge.

The `git merge` command was first introduced in [Osnove vej](#). Though it is used in various places in the book, there are very few variations of the `merge` command—generally just `git merge <branch>` with the name of the single branch you want to merge in.

We covered how to do a squashed merge (where Git merges the work but pretends like it's just a new commit without recording the history of the branch you're merging in) at the very end of [Forkan javni projekt](#).

We went over a lot about the merge process and command, including the `-Xignore-space-change` command and the `--abort` flag to abort a problem merge in [Advanced Merging](#).

We learned how to verify signatures before merging if your project is using GPG signing in [Signing Commits](#).

Finally, we learned about Subtree merging in [Združevanje pod dreves](#).

git mergetool

The `git mergetool` command simply launches an external merge helper in case you have issues with a merge in Git.

We mention it quickly in [Basic Merge Conflicts](#) and go into detail on how to implement your own external merge tool in [External Merge and Diff Tools](#).

git log

The `git log` command is used to show the reachable recorded history of a project from the most recent commit snapshot backwards. By default it will only show the history of the branch you're currently on, but can be given different or even multiple heads or branches from which to traverse. It is also often used to show differences between two or more branches at the commit level.

This command is used in nearly every chapter of the book to demonstrate the history of a project.

We introduce the command and cover it in some depth in [Pregled zgodovine pošiljanja](#). There we look at the `-p` and `--stat` option to get an idea of what was introduced in each commit and the `--pretty` and `--oneline` options to view the history more concisely, along with some simple date and author filtering options.

In [Ustvarjanje nove veje](#) we use it with the `--decorate` option to easily visualize where our branch pointers are located and we also use the `--graph` option to see what divergent histories look like.

In [Privatna majhna ekipa](#) and [Commit Ranges](#) we cover the `branchA..branchB` syntax to use the `git log` command to see what commits are unique to a branch relative to another branch. In [Commit Ranges](#) we go through this fairly extensively.

In [Merge Log](#) and [Triple Dot](#) we cover using the `branchA...branchB` format and the `--left-right` syntax to see what is in one branch or the other but not in both. In [Merge Log](#) we also look at how to use the `--merge` option to help with merge conflict debugging as well as using the `--cc` option to look at merge commit conflicts in your history.

In [RefLog Shortnames](#) we use the `-g` option to view the Git relog through this tool instead of doing branch traversal.

In [Searching](#) we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In [Signing Commits](#) we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch.

This is basically entirely covered in [Stashing and Cleaning](#).

git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

This command is introduced and covered in detail in [Označevanje](#) and we use it in practice in [Tagging Your Releases](#).

We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in [Signing Your Work](#).

Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

We first look at this command in [Pridobivanje in poteg iz vaših daljav](#) and we continue to see examples of it use in [Oddaljene veje](#).

We also use it in several of the examples in [Prispevanje projektu](#).

We use it to fetch a single specific reference that is outside of the default space in [Pull Request Refs](#) and we see how to fetch from a bundle in [Bundling](#).

We set up highly custom refsspecs in order to make `git fetch` do something a little different than the default in [The Refspec](#).

git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you're on.

We introduce it quickly in [Pridobivanje in poteg iz vaših daljav](#) and show how to see what it will merge if you run it in [Preverjanje daljave](#).

We also see how to use it to help with rebasing difficulties in [Ponovno bazirajte ko ponovno bazirate](#).

We show how to use it with a URL to pull in changes in a one-off fashion in [Checking Out Remote Branches](#).

Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in [Signing Commits](#).

git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

We first look at the `git push` command in [Potiskanje v vaše daljave](#). Here we cover the basics of pushing a branch to a remote repository. In [Porivanje](#) we go a little deeper into pushing specific branches and in [Sledenje vej](#) we see how to set up tracking branches to automatically push to. In [Izbris oddaljenih vej](#) we use the `--delete` flag to delete a branch on the server with `git push`.

Throughout [Prispevanje projektu](#) we see several examples of using `git push` to share work on branches through multiple remotes.

We see how to use it to share tags that you have made with the `--tags` option in [Deljenej oznak](#).

In [Publishing Submodule Changes](#) we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

In [Other Client Hooks](#) we talk briefly about the `pre-push` hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.

Finally, in [Refspec-i za potiskanje](#) we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as “origin” so you don’t have to type them out all the time. You can have several of these and the `git remote` command is used to add, change and delete them.

This command is covered in detail in [Delo z daljavami](#), including listing, adding, removing and renaming them.

It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote add <name> <url>` format.

git archive

The `git archive` command is used to create an archive file of a specific snapshot of the

project.

We use `git archive` to create a tarball of a project for sharing in [Preparing a Release](#).

git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The `submodule` command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources.

This command is only mentioned and entirely covered in [Submodules](#).

Inspection and Comparison

git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

We first use it to show annotated tag information in [Anotirane oznake](#).

Later we use it quite a bit in [Revision Selection](#) to show the commits that our various revision selections resolve to.

One of the more interesting things we do with `git show` is in [Manual File Re-merging](#) to extract specific file contents of various stages during a merge conflict.

git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

We showed how to use it to create a nice changelog in [The Shortlog](#).

git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It's a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.

We use `git describe` in [Generating a Build Number](#) and [Preparing a Release](#) to get a string to name our release file after.

Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from figuring out where something was introduced to figuring out who

introduced it.

git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

It is fully covered in [Binary Search](#) and is only mentioned in that section.

git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

It is covered in [File Annotation](#) and is only mentioned in that section.

git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

It is covered in [Git Grep](#) and is only mentioned in that section.

Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as though the commit series is a series of patches. These commands help you manage your branches in this manner.

git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you're currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

Cherry picking is described and demonstrated in [Rebasing and Cherry Picking Workflows](#).

git rebase

The `git rebase` command is basically an automated `cherry-pick`. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

Rebasing is covered in detail in [Ponovno baziranje \(rebasing\)](#), including covering the collaborative issues involved with rebasing branches that are already public.

We use it in practice during an example of splitting your history into two separate repositories in [Replace](#), using the `--onto` flag as well.

We go through running into a merge conflict during rebasing in [Rerere](#).

We also use it in an interactive scripting mode with the `-i` option in [Changing Multiple Commit Messages](#).

git revert

The `git revert` command is essentially a reverse `git cherry-pick`. It creates a new commit that applies the exact opposite of the change introduced in the commit you're targeting, essentially undoing or reverting it.

We use this in [Reverse the commit](#) to undo a merge commit.

Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

git apply

The `git apply` command applies a patch created with the `git diff` or even GNU diff command. It is similar to what the `patch` command might do with a few small differences.

We demonstrate using it and the circumstances in which you might do so in [Uporaba popravkov iz e-pošte](#).

git am

The `git am` command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

We covered usage and workflow around `git am` in [Uporaba popravka z am](#) including using the `--resolved`, `-i` and `-3` options.

There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in [E-mail Workflow Hooks](#).

We also use it to apply patch formatted GitHub Pull Request changes in [E-poštna obvestila](#).

git format-patch

The `git format-patch` command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted.

We go through an example of contributing to a project using the `git format-patch` tool in [Javni projekt preko e-pošte](#).

git imap-send

The `git imap-send` command uploads a mailbox generated with `git format-patch` into an IMAP drafts folder.

We go through an example of contributing to a project by sending patches with the `git imap-send` tool in [Javni projekt preko e-pošte](#).

git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email.

We go through an example of contributing to a project by sending patches with the `git send-email` tool in [Javni projekt preko e-pošte](#).

git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in [Forkan javni projekt](#).

External Systems

Git comes with a few commands to integrate with other version control systems.

git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

This command is covered in depth in [Git and Subversion](#).

git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record.

This command is covered in depth in [A Custom Importer](#).

Administration

If you're administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

git gc

The `git gc` command runs “garbage collection” on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in [Maintenance](#).

git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies.

We only quickly use this once in [Data Recovery](#) to search for dangling objects.

git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in [RefLog Shortnames](#), where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.

We also go through a practical example of recovering such a lost branch in [Data Recovery](#).

git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

In [Removing a File from Every Commit](#) we explain the command and explore several different options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.

In [Git-p4](#) and [TFS](#) we use it to fix up imported external repositories.

Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

The first one we encounter is `ls-remote` in [Pull Request Refs](#) which we use to look at the raw references on the server.

We use `ls-files` in [Manual File Re-merging](#), [Rerere](#) and [The Index](#) to take a more raw look at what your staging area looks like.

We also mention `rev-parse` in [Branch References](#) to take just about any string and turn it into an object SHA-1.

However, most of the low level plumbing commands we cover are in [Notranjost Git-a](#), which is more or less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.

Index

@

\$EDITOR, 336

\$VISUAL

see \$EDITOR, 336

.NET, 487

.gitignore, 338

0 , 86

1 , 86

©, 483

A

Apache, 110

Apple, 488

aliases, 56

archiving, 352

attributes, 345

autocorrect, 338

B

BitKeeper, 12

bash, 478

binary files, 346

bitnami, 114

branches, 58

basic workflow, 65

creating, 60

deleting remote, 87

diffing, 151

long-running, 75

managing, 73

merging, 69

remote, 78, 150

switching, 61

topic, 76, 147

tracking, 86

upstream, 86

build numbers, 160

C

C#, 487

CRLF, 18

CVS, 9

Cocoa, 488

color, 338

commit templates, 336

contributing, 123

private managed team, 133

private small team, 126

public large project, 143

public small project, 139

credential caching, 18

credentials, 329

crLf, 343

D

diffTool, 339

distributed git, 120

E

Eclipse, 478

editor

changing default, 33

email, 145

applying patches from, 147

excludes, 338, 431

F

files

moving, 36

removing, 35

forking, 122, 167

G

GPG, 337

GUIs, 471

Git as a client, 367

GitHub, 162

API, 209

Flow, 168

organizations, 201

pull requests, 171

user accounts, 162

GitHub for Mac, 473

GitHub for Windows, 473

GitLab, 114

GitWeb, 112

Graphical tools, 471

git commands

add, 26, 26, 27

am, 148

- apply, 147
- archive, 160
- branch, 60, 73
- checkout, 61
- cherry-pick, 157
- clone, 24
 - bare, 103
- commit, 33, 58
- config, 20, 21, 33, 56, 145, 335
- credential, 329
- daemon, 109
- describe, 160
- diff, 30
 - check, 124
- fast-import, 421
- fetch, 49
- fetch-pack, 455
- filter-branch, 419
- format-patch, 143
- gitk, 471
- gui, 471
- help, 22, 109
- http-backend, 110
- init, 23, 26
 - bare, 104, 107
- instaweb, 112
- log, 37
- merge, 67
 - squash, 142
- mergetool, 72
- p4, 395, 418
- pull, 49
- push, 49, 55, 84
- rebase, 89
- receive-pack, 453
- remote, 47, 48, 50, 51
- request-pull, 140
- rerere, 158
- send-pack, 453
- shortlog, 161
- show, 54
- show-ref, 370
- status, 25, 33
- svn, 367
- tag, 52, 53, 54

- upload-pack, 455
- git-svn, 367
- git-tf, 403
- git-tfs, 403
- gitk, 471

H

- hooks, 354
 - post-update, 101

I

- IRC, 22

- Importing

- from Mercurial, 415
- from Perforce, 418
- from Subversion, 413
- from TFS, 420
- from others, 421

- Interoperation with other VCSs

- Mercurial, 378
- Perforce, 386
- Subversion, 367
- TFS, 403

- ignoring files, 29

- integrating work, 153

J

- java, 488

- jgit, 488

K

- keyword expansion, 349

L

- Linus Torvalds, 12

- Linux, 12

- installing, 16

- libgit2, 483

- line endings, 343

- log filtering, 44

- log formatting, 40

M

- Mac

- installing, 17

Mercurial, [378](#), [415](#)
Migrating to Git, [413](#)
Mono, [487](#)
maintaining a project, [146](#)
master, [59](#)
mergetool, [339](#)
merging, [69](#)
 conflicts, [71](#)
 strategies, [353](#)
 vs. rebasing, [97](#)

O

Objective-C, [488](#)
origin, [79](#)

P

Perforce, [9](#), [12](#), [386](#), [418](#)
 Git Fusion, [387](#)
Powershell, [18](#)
Python, [488](#)
pager, [337](#)
policy example, [357](#)
posh-git, [481](#)
powershell, [481](#)
protocols
 SSH, [102](#)
 dumb HTTP, [100](#)
 git, [102](#)
 local, [98](#)
 smart HTTP, [100](#)
pulling, [87](#)
pushing, [84](#)

R

Ruby, [484](#)
rebasing, [88](#)
 perils of, [92](#)
 vs. merging, [97](#)
references
 remote, [78](#)
releasing, [160](#)
rerere, [158](#)

S

SHA-1, [14](#)

SSH keys, [105](#)
 with GitHub, [163](#)
Subversion, [9](#), [12](#), [121](#), [367](#), [413](#)
serving repositories, [98](#)
 GitLab, [114](#)
 GitWeb, [112](#)
 HTTP, [110](#)
 SSH, [105](#)
 git protocol, [109](#)
shell prompts
 bash, [478](#)
 powershell, [481](#)
 zsh, [479](#)
staging area
 skipping, [34](#)

T

TFS, [403](#), [420](#)
TFVC
 see=TFS, [403](#)
tab completion
 bash, [478](#)
 powershell, [481](#)
 zsh, [479](#)
tags, [52](#), [159](#)
 annotated, [53](#)
 lightweight, [53](#)
 signing, [159](#)

V

Visual Studio, [476](#)
version control, [8](#)
 centralized, [9](#)
 distributed, [10](#)
 local, [8](#)

W

Windows
 installing, [18](#)
whitespace, [342](#)
workflows, [120](#)
 centralized, [120](#)
 dictator and lieutenants, [122](#)
 integration manager, [121](#)
 merging, [153](#)

merging (large), [155](#)
rebasing and cherry-picking, [157](#)

X

Xcode, [17](#)

Z

zsh, [479](#)