Handbook
and
Operations Manual

Institute for Intelligent Systems

THE UNIVERSITY OF MEMPHIS

IIS Handbook and Operations Manual

Table of Contents

IIS Handbook and Operations Manual

Preface by Andrew Olney

Welcome to the first version of the IIS Handbook and Operations Manual. The IIS has never had a book like this before, but as the IIS has grown in scope and complexity, so has the need for a handbook that documents our programs and procedures.

This first version (17.05 in year/month notation) reflects my personal views that have developed over the last 10 years as I've served as Associate Director and eventually Director of the IIS. Accordingly, the priorities and biases are mine and have not been endorsed by the IIS community.

My hope is that future versions will be voted into adoption or follow some similar consensus-building process. By hosting this handbook on GitHub and using AsciiDoc, I've made it as easy as I know how for this book to be improved by a distributed group of people, potentially across many years, such that all revisions are well documented. This approach is well-traveled by other open source books, and I've leaned heavily on the model provided by **Pro Git** (2nd edition). To make changes, please refer to the procedures in the README file in the root of this repository.

Getting Started

This chapter provides a common ground for everyone that is assumed for the following chapters. It gives a high-level view of what the IIS is, what it does, and how it works. Some of these topics are further developed for student and faculty concerns in later chapters.

About the IIS

What is the The Institute for Intelligent Systems?

The Institute for Intelligent Systems (IIS) is an interdisciplinary research center at the University of Memphis. Unlike many research centers, the IIS has historically pursued a research agenda set by participating researchers who **self-organize around research problems**, rather than a highly focused formal agenda. The breadth and flexibility of the IIS is reflected in our current mission statement:

> The Institute for Intelligent Systems is dedicated to advancing the state of knowledge and capabilities of intelligent systems, including psychological, biological, and artificial systems.
>
> By conducting cutting-edge research and publishing our findings in peer-reviewed venues, we contribute to the discipline and, ultimately, to the public. In doing so, we are also training the next generation of scientists.
>
> Our mission depends on an interdisciplinary approach that brings together researchers from many different backgrounds, including computer science, cognitive psychology, education, philosophy, linguistics, engineering, English, and biology.

TIP	Our "bottom-up" approach to forming research groups makes the IIS very accessible: if you can find some like-minded people interested in your project, you can form a group.

Although the IIS maintains a strong tradition of bottom-up research, our recent strategic plan adopted three focus areas:

- Language & Discourse
- Learning
- Artificial Intelligence

These focus areas represent the deepest areas of *current* expertise at the IIS, but they should not be regarded as an intention to exclude other areas of research or limit what the IIS can become in the future.

A Short History of the IIS

The IIS was formed in 1985 as a result of informal meetings between Dr. Stan Franklin (Computer Science), Dr. Art Graesser (Psychology), and Dr. Terry Horgan (Philosophy), with Graesser and Franklin initially serving as Co-Directors. Dr. Don Franceschetti (Physics) developed a five-year plan for the IIS which led to the recognition by the State of Tennessee of the IIS as an Institute. The Cognitive Science Seminar began in 1985 and continues to this day.

NOTE	The Cognitive Science Seminar has been in continuous operation since 1985. It is now a cross-listed course with Psychology, Philosophy, and Computer Science.

In 1998, as a result of multiple major grants to faculty participating in the IIS, the IIS received an operating budget of $25,000, 5% indirect cost recovery from grants, and two staff positions that were both filled by 2000.

In 2003, the IIS moved to the FedEx Institute of Technology (FIT) and began hiring two faculty with split appointments between the IIS and other departments.

Beginning in 2007, the IIS began hiring 100% faculty lines. These lines are fully within the IIS but have tenure housed in other departments. This development marked a number of administrative changes in the IIS as it continued to grow into a mini-department. An additional staff person was added to support grant submissions (i.e. pre-award support) in recognition of the importance of grant funding to the IIS mission.

In 2010, the IIS was allocated 10 graduate assistantships, additional faculty lines, additional staff, 20% indirect cost recovery, and the 4th floor of the FIT. The IIS Student Organization (IISSO) was formed and tasked with managing student travel awards. The IIS added its first degree program, the Cognitive Science Graduate Certificate.

In 2014, the IIS launched a multi-year strategic planning process. Many of the recommendations of the strategic planning process have already been implemented, including a new undergraduate Minor in Cognitive Science. However parts of the strategic plan will take years to unfold.

For more detailed history of the IIS, up to 2015, please refer to the IIS Strategic Plan Self-Study.

IIS Membership

For many years the IIS did not have a formal membership definition. [1: Stan Franklin had a retrospective method that calculated membership based on participation over a period of time. However this did not allow one to "sign up," which is a common attribute of memberships.]

Since 2008, the IIS has had a membership letter mechanism: faculty can join the IIS as "IIS Affiliates" by signing the letter and returning it to the IIS Director. The letter states what the Affiliate can expect from the IIS and vice versa. In a nutshell, the IIS expects participation and, ultimately, grant activity that funds IIS services. In return the Affiliate has access to the various resources the IIS provides.

There is not a similar membership letter for students. Students may join the IIS Student Organization (IISSO) according to the current policies and procedures of that organization.

However, being a member of the IISSO does not automatically grant access to IIS resources. To access IIS resources, students must be "claimed" by an IIS Affiliate faculty member as described later in this handbook.

IIS Student Organization (IISSO)

The IIS Student Organization (IISSO) is a self-organized group of students, most of whom work with IIS Affiliates. In addition to providing a community for students, the IISSO makes recommendations to the IIS regarding the allocation of various resources to students, including student travel awards and thesis/dissertation awards. The IISSO also organizes the poster session at the Speed Date, guest speaker lunches, and other social events.

Current information about the IISSO can be found at https://sites.google.com/site/iissomemphis2/

Students and faculty should refer to current information on the website.

A copy of the IISSO charter, **which may not be current**, follows.

Charter

Mission Statement

The IISSO is a student-led advocacy and coordinating group at the Institute for Intelligent Systems (IIS). The IISSO represents the interests of undergraduate and graduate students that are affiliated with the IIS. The IISSO actively pursues opportunities to improve student welfare through academic and social projects, and also strives to advance student research by providing opportunities for funding, networking, and promotion of research. IISSO meetings are scheduled once or twice a semester and all IIS-affiliated students are welcome to attend. The meeting minutes will be distributed through the IISSO email list within one week of the meeting. Student elections for the IISSO officers are held annually in May.

Its major responsibilities include a) distributing travel funds to students that attend academic conferences and workshops*, b) organizing student lunches with visiting speakers, c) providing opportunities for students to present their research and/or lead workshops/seminars, d) providing opportunities to interact with professors interviewing for IIS-faculty positions, e) keeping students informed of IISSO and IIS activities, f) training the new IISSO officers-elect, and currently on pilot basis, g) distributing thesis and dissertation funds to students toward the completion of their degrees*.

*See eligibility requirements and submission guidelines on the Funding page.

Qualification for Membership

Any undergraduate or graduate student affiliated with the IIS or with a strong interest in cognitive science interdisciplinary research is able to join and participate in IISSO events.

Membership does not have any strict requirements. To join, simply email the President to have your name added to the mailing list (MUST be your U of M email). The IISSO strives to advance student research by providing opportunities for funding, networking, and promotion of research.

As such, members have access to the following benefits: participation in the IISSO research fair (usually toward the end of the Fall and Spring semesters), the opportunity to collaborate with researchers in different fields, as well as the chance to meet and network with experts from outside of the U of M in a variety of cognitive science fields (through the Cognitive Science Seminar lunches), attend and/or present workshops to further your academic career/CV.

While anyone is welcome to join and may benefit from any or all of the abovementioned benefits of membership, limited travel funding (and thesis/dissertation funding) can be awarded to members who meet certain criteria. Specifically, one must be an IISSO member collaborating on a project with an IIS-approved faculty member. Upon joining the IISSO, if you are interested in possibly applying for travel funding in the future, you should check with your advisor or any professor you are collaborating with to see if they are IIS-affiliated and if they have added their students to the list of IISSO-travel-funding-approved students. Even if you meet this requirement (and others detailed on the travel funding page), no one is guaranteed funding. For more information on travel funding, please see the IISSO Funding Page. There is also preliminary information on the Thesis/Dissertation funding on the Funding page. You may also contact any of the current officers with questions or concerns about membership and eligibility.

Events

Research Meetings

Individual projects and labs schedule weekly meetings in 405 and 407 FIT, the IIS main meeting rooms. Affiliates can book rooms with Renee Cogar. These meetings and descriptions are publicly posted to our website, and all IIS Affiliates and students are encouraged to drop in or participate in ongoing research meetings. There are approximately 25 regularly scheduled research meetings per week for externally funded projects.

Cognitive Science Seminar

The Cognitive Science Seminar is a hybrid course/public lecture. Students who register for the course (COMP/PHIL/PSYC 7514/8514) attend a course-only portion from 2:20pm to 4pm as well as a guest lecture portion, which is also open to the public, from 4pm to 5:20pm.

The topic of the seminar varies semester by semester, as do the faculty leading it. It is common for students to have lunch with guest speakers and for faculty to take them to dinner. More information about the seminar can be found in the faculty and student chapters.

Guest Speakers

IIS Affiliates will often bhost visitors who are willing to give a talk to the wider IIS community. Although the IIS does not provide material support for such ad-hoc speakers, we encourage Affiliates to make use of our meeting rooms and work with IIS staff to promote the talk.

Speed Date

At the end of each semester (typically 4pm to 7pm the Friday before Study Day), the IIS holds a Speed Date event in which 8-10 Affiliates give 5 minute presentations of their current research

interests. The presentation portion is followed by a wine and cheese poster session, organized by the IISSO, where students present their latest research. Affiliates who wish to volunteer to speak at the Speed Date are always welcome to do so, otherwise Affiliates are invited based on how much time has passed since their last presentation, with priority given to Affiliates who recently joined.

Communications

Email

The IIS maintains multiple mailing lists:

- IIS-Faculty: faculty affiliates.
- IIS-Students: **interested** students, some of which might not be working with IIS affiliates.
- IIS-Pro: grant-supported staff and postdocs
- IIS-Admin: president, provost, vice president of research, FIT administration, deans and chairs of all faculty affiliates, staff in communications

Web

The IIS website has the following elements:

- List of affiliates and staff
- Degree programs
- Calendar of research meetings
- Links to resources
- List of affiliate projects

The IIS Student Organization (IISSO) has their own website with specific information for students.

Social Media

Currently the IIS participates in:

- LinkedIn : there is a LinkedIn group for current and former students, staff, and faculty.
- Facebook

IIS social media accounts are managed by Leah Windsor.

Press Releases

In the past, faculty and students were largely on their own with respect to press releases, meaning that they had to write them, submit them to communications, and then wait and see if anything came of it.

The current process is that Mary Ann Dawson is on the IIS-Admin list and will proactively create press releases. If faculty or students have a particular story they think is worthy of a press release,

they are encouraged to contact Mary Ann Dawson directly.

Branding

Use of IIS logo and branding is encouraged on presentation slides and conference posters. Discretion and care should be taken to avoid using the IIS brand in such a way that it endorses an activity besides an individual's research. Sponsorships and similar promotion/endorsements must be approved by the IIS Director.

Logo art is available on the IIS website under Resources.

Signage

The IIS has several signs used primarily for IIS events like the Speed Date. These signs can be used for other purposes but need approval (like branding) and can be checked out from Renee Cogar.

Brochures

General brochures about the IIS are available. While the IIS can provide these in limited quantities (e.g. 100), it should be recognized that these are produced at cost and should be used strategically (e.g. to attract students). General brochures can be obtained from Renee Cogar.

IIS can also assist faculty affiliates with the creation of brochures specific to their lab or grant project. Faculty wishing to create custom brochures should contact Adam Remsen.

Outreach

The IIS engages in outreach and encourages students and faculty to participate. These activities help promote the IIS brand, attract future students, and help us connect with new collaborators. Outreach activities can take place on-campus or off-campus.

On-campus, IIS faculty and students are frequently asked to participate in lab tours and associated demonstrations. Often touring groups are middle or high school students, so the tours/demonstrations need to be tailored to a general audience. These are excellent opportunities for graduate students to practice their presentation skills.

Off-campus events include demonstrations at schools or community events, speaking engagements, competition judging, and similar activities. Often times off-campus outreach is spearheaded by a particular faculty member, but to the extent possible, the IIS can support these activities by providing signs or similar materials.

Resources

Resources specific to students and faculty are addressed in their respective chapters. The resources that follow are available to everyone. Additional resources, such as links to information or university resources, are listed on the IIS website Resource page.

Meeting space

Faculty and students can book meeting rooms with Renee Cogar. In general, booking is on a first-come, first-served basis.

Teleconference system

Faculty and students can also book the IIS teleconference system with Renee Cogar. The system is located in 407 FIT.

The system is an enterprise version of Google Hangouts with the following features

- Accomodates a large group with one screen, camera, and microphone
- Allows a large group of attendees (~20)
- Can connect participants via telephone in addition to Google Hangouts
- For recurring meetings with fixed participants, can automatically start meetings and invite participants based on a calendar event (managed by Renee Cogar).

Library

The IIS Library is open to faculty and students at any time. The library holds a large collection of books donated by faculty (approximately 2,000).

Each book is indexed and cataloged, and may be searched in two ways - Book metadata such as author and title - Full text search using Google Books search restricted to our library

Linguistic Data Consortium subscription (LDC)

The Linguistic Data Consortium (LDC) is perhaps the world's leading repository of linguistic data. LDC datasets are commonly used in shared tasks and to benchmark algorithms. They can cost thousands of dollars each.

The IIS has an institutional subscription, which gives us total access to each dataset released in a subscription year as well as a reduced price on non-covered datasets. Datasets in our subscription years are available on campus at http://psychiisnas.memphis.edu/LDC/start.html

Certain datasets require special license agreements. If a dataset appears to be missing, it may require a signature release. Contact Renee Cogar for such requests.

Dreamspark/Imagine Subscription

The IIS has a DreamSpark subscription. IIS Students, faculty, and staff may download Microsoft titles from our webstore.

The primary types of software offered are

- Windows operating systems
- SQL server and similar server titles
- Visual studio and similar development tools

Faculty and students can obtain Microsoft Office products directly from the University at http://www.memphis.edu/getoffice

Eyetracker

The IIS has a "single instrument" experiment room, which currently contains a Tobii T60 Eyetracker.

This room may be booked for single instrument experiments through Renee Cogar.

Equipment check out

Various equipment is available for checkout, including

- Projectors
- Google glass

Equipment may be checked out through Renee Cogar.

Degree programs

This IIS hosts two degree programs, the Cognitive Science graduate certificate (a graduate program) and the Cognitive Science Minor (an undergraduate program). Currently both programs consist of courses hosted in other departments, but often taught by IIS Affiliates. Information about both programs is available at http://www.memphis.edu/cognitive-science/

Importantly, that link lists the specific courses offered in the current/upcoming semester that apply to the certificate and minor, which should help students when registering for classes.

Cognitive Science Graduate Certificate

The certificate is available to all enrolled graduate students from any department. Additionally, students can be on the certificate as **non-degree seeking** and by doing so be eligible for GA positions. Similarly, non-degree seeking students on the certificate are eligible for financial aid. So a student between UG and a graduate program may find taking the certificate to be an effective use of their time while they are waiting to be admitted to a graduate program.

Ideally students interested in the certificate will apply for admission right away, but it is also possible to enter the certificate right before graduating and have previous classes retroactively count toward the certificate (as long as no more than 6 hours are "double counted" towards the certificate and another degree).

Students already enrolled in a graduate program (or who have previously been degree seeking) can apply to enter the certificate by filling out a change of status form:

http://www.memphis.edu/gradschool/pdfs/forms/changeofstatus.pdf

with degree program as "Graduate Certificate", Major: Cognitive Science.

and then filling out the form, printing/signing, and mailing to the Graduate School or walking over

to the Graduate School.

If the student is currently enrolled as non-degree and has never been degree seeking he or she must complete an online readmit application.

For non-degree students, the IIS graduate coordinator, Andrew Olney, is the advisor of record and must clear classes each semester. For degree-seeking students the regular advisor remains the advisor of record; however feel free to contact me for advice on course selection.

In addition to the listed courses that can count towards the certificate, the graduate coordinator may approve course substitutions. Faculty are encouraged to contact the graduate coordinator if they are teaching a "one off" seminar course that is aligned with the certificate or a new course that is similarly aligned. Students are encouraged to contact the graduate coordinator for advice regarding courses, particularly courses outside their area.

In the semester of graduation, students on the certificate must fill out the following forms

- Apply to graduate
- Graduate certificate candidacy form
- Course substitution forms (as needed)

Forms are available at http://www.memphis.edu/gradschool/resources/forms_index.php

Andrew Olney will sign off on the forms and route them for further approval.

The deadlines are listed here:

http://www.memphis.edu/gradschool/current_students/graduation_information/graduation_deadlines.php

Note that deadlines for graduation paperwork are early in the semester

Please contact Andrew Olney if you have any questions.

Cognitive Science Minor

The minor is available to all undergraduate students **except** Psychology majors. Psychology majors should instead take the Cognitive Science concentration within the Psychology major.

Unlike the graduate certificate, the minor is largely administered by the College of Arts and Sciences (CAS). Students wishing to declare the minor should follow the CAS procedure at http://www.memphis.edu/cas/advising/declare_major.php

A student wishing a course substitution or exception should contact the IIS undergraduate coordinator, {iis-undergraduate-coordinator}, who will contact graduation analysts in CAS to make the change.

Staff

Most staff are available during academic hours and on an as-needed basis. The major exceptions

are grant preparation and software development.

Grant preparation support is negotiated in advance and coordinated according to other grant submissions that are due at approximately the same time. This coordination encourages faculty to notify of intent to submit and work with our grant preparation staff well in advance of the submission deadline.

Software development likewise is time-intensive for any given project. Currently software development support is allocated using an RFP system. Faculty write a 1-2 page narrative describing how their project would benefit from software development support. Support is allocated in 2 month blocks per project on a 6 month cycle. During any 2 month block, our software development staff member is 100% committed to the assigned project.

Senior Administrative Secretary (Renee Cogar)

- Manage all meeting space (405/407)
- Manage all IIS email lists
- Manage library; order books and journals
- Handles travel bookings as requested
- Book events like the Speed Date
- Handles mail, faxing, and phone communications

Financial Services Associate (Vickie Middleton)

- Manage IIS Operating and IDCR accounts
- Purchasing
- GA contracts
- Reimbursement
- Travel authorization
- Inventory

Pre-Award Coordinator (Adam Remsen)

- Track and route funding opportunities to faculty
- Grant proposal preparation
- Budget, budget justification, advance account requests
- Edits documents and checks for sponsor compliance
- Maintains current and pending support database
- Edits faculty CVs
- Prepares proposal packages; coordinates submission with internal and external research offices
- Other grant-related activities as needed

Business Officer (Mattie Haynes)

- Manage all IIS grant accounts
- Monitor IIS Operating and IDCR accounts
- Manage program income
- Provide monthly reports on grant and IDCR accounts
- Rebudget as needed
- Manage summer salary, AY incentive, and check compliance
- HR functions; prepare contracts for research faculty
- Act as liaison with the office of research support services, accounts payable, financial planning,
- and grants accounting on behalf of the IIS (faculty, staff, students, etc.).

Senior Software Developer (Andrew Tackett)

- Develop and maintain grant-sponsored software
- Trains others in use and installation of software, prepared documentation
- Maintains versioning, code repositories, and releases of software
- Supervises and trains students in software development
- GitHub administrator

Director (Alistair Windsor)

- Immigration and sabbatical invitation letters
- Annual evaluations for faculty
- Annual evaluations for staff
- Assign research space as needed
- Assign computing resources as needed
- Assign IIS GAs on annual basis
- Monitor and update website as needed
- Coordinate Cognitive Science Graduate Certificate (admissions, matriculation, etc)
- Recruit and coordinate instructors for the CogSci seminar
- Recruit and coordinate speakers for the Speed Date
- Coordinate public demonstrations and outreach
- Serve as committee chair/member for faculty hires
- Recruit and retain IIS Affiliates, including counteroffers
- Work with IISSO to maintain student travel funding program
- Produce annual report and program reviews as needed
- DreamSpark, Google Apps for Education, and LinkedIn group administrator

- Monitor the IIS and intervene as needed

Summary

You should have a basic understanding of what the IIS is, what it does, and how it works.

Git Basics

If you can read only one chapter to get going with Git, this is it. This chapter covers every basic command you need to do the vast majority of the things you'll eventually spend your time doing with Git. By the end of the chapter, you should be able to configure and initialize a repository, begin and stop tracking files, and stage and commit changes. We'll also show you how to set up Git to ignore certain files and file patterns, how to undo mistakes quickly and easily, how to browse the history of your project and view changes between commits, and how to push and pull from remote repositories.

Getting a Git Repository

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

Initializing a Repository in an Existing Directory

If you're starting to track an existing project in Git, you need to go to the project's directory. If you've never done this, it looks a little different depending on which system you're running:

for Linux:

```
$ cd /home/user/your_repository
```

for Mac:

```
$ cd /Users/user/your_repository
```

for Windows:

```
$ cd /c/user/your_repository
```

and type:

```
$ git init
```

This creates a new subdirectory named `.git` that contains all of your necessary repository files – a Git repository skeleton. At this point, nothing in your project is tracked yet. (See [_git_internals] for more information about exactly what files are contained in the `.git` directory you just created.)

If you want to start version-controlling existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit. You can accomplish that with a few `git add` commands that specify the files you want to track, followed by a `git commit`:

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

We'll go over what these commands do in just a minute. At this point, you have a Git repository with tracked files and an initial commit.

Cloning an Existing Repository

If you want to get a copy of an existing Git repository – for example, a project you'd like to contribute to – the command you need is `git clone`. If you're familiar with other VCS systems such as Subversion, you'll notice that the command is "clone" and not "checkout". This is an important distinction – instead of getting just a working copy, Git receives a full copy of nearly all data that the server has. Every version of every file for the history of the project is pulled down by default when you run `git clone`. In fact, if your server disk gets corrupted, you can often use nearly any of the clones on any client to set the server back to the state it was in when it was cloned (you may lose some server-side hooks and such, but all the versioned data would be there – see Git on the Server for more details).

You clone a repository with `git clone [url]`. For example, if you want to clone the Git linkable library called libgit2, you can do so like this:

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

That creates a directory named "libgit2", initializes a `.git` directory inside it, pulls down all the data for that repository, and checks out a working copy of the latest version. If you go into the new `libgit2` directory, you'll see the project files in there, ready to be worked on or used. If you want to clone the repository into a directory named something other than "libgit2", you can specify that as the next command-line option:

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

That command does the same thing as the previous one, but the target directory is called `mylibgit`.

Git has a number of different transfer protocols you can use. The previous example uses the `https://` protocol, but you may also see `git://` or `user@server:path/to/repo.git`, which uses the SSH transfer protocol. Git on the Server will introduce all of the available options the server can set up to access your Git repository and the pros and cons of each.

Recording Changes to the Repository

You have a bona fide Git repository and a checkout or working copy of the files for that project. You need to make some changes and commit snapshots of those changes into your repository each time the project reaches a state you want to record.

Remember that each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven't edited anything.

As you edit files, Git sees them as modified, because you've changed them since your last commit. You stage these modified files and then commit all your staged changes, and the cycle repeats.

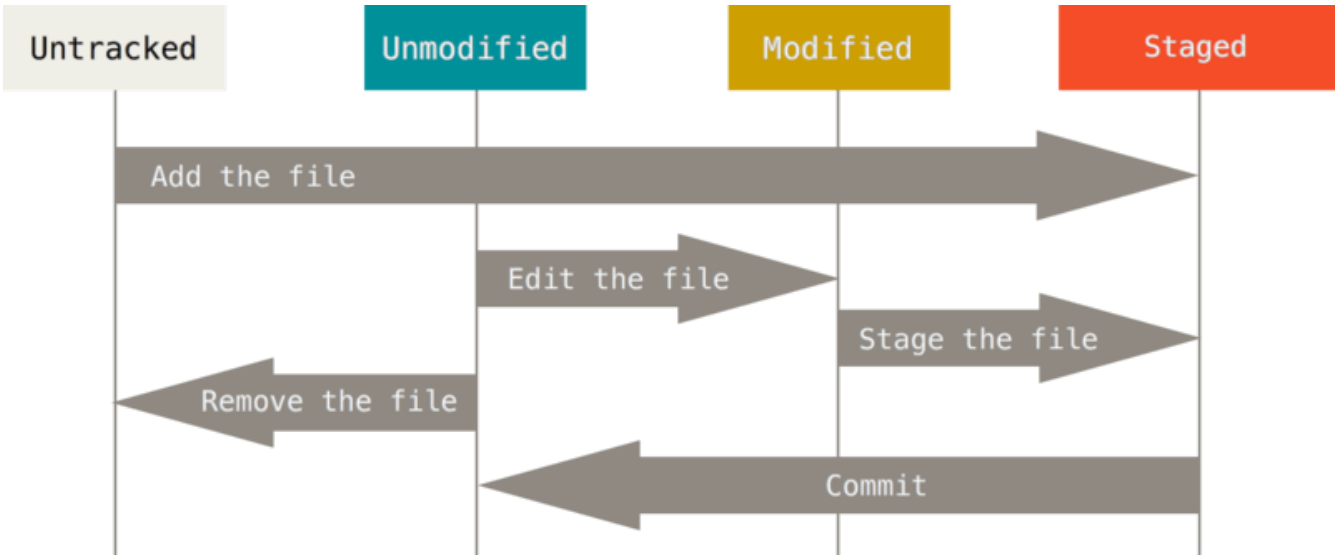

Figure 1. The lifecycle of the status of your files.

Checking the Status of Your Files

The main tool you use to determine which files are in which state is the `git status` command. If you run this command directly after a clone, you should see something like this:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

This means you have a clean working directory – in other words, none of your tracked files are modified. Git also doesn't see any untracked files, or they would be listed here. Finally, the command tells you which branch you're on and informs you that it has not diverged from the same branch on the server. For now, that branch is always "master", which is the default; you won't worry about it here. Git Branching will go over branches and references in detail.

Let's say you add a new file to your project, a simple README file. If the file didn't exist before, and you run `git status`, you see your untracked file like so:

```
$ echo 'My Project' > README
$ 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)

    README

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

You can see that your new README file is untracked, because it's under the "Untracked files" heading in your status output. Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit); Git won't start including it in your commit snapshots until you explicitly tell it to do so. It does this so you don't accidentally begin including generated binary files or other files that you did not mean to include. You do want to start including README, so let's start tracking the file.

Tracking New Files

In order to begin tracking a new file, you use the command `git add`. To begin tracking the README file, you can run this:

```
$ git add README
```

If you run your status command again, you can see that your README file is now tracked and staged to be committed:

```
$ 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:   README
```

You can tell that it's staged because it's under the "Changes to be committed" heading. If you commit at this point, the version of the file at the time you ran `git add` is what will be in the historical snapshot. You may recall that when you ran `git init` earlier, you then ran `git add (files)` – that was to begin tracking files in your directory. The `git add` command takes a path name for either a file or a directory; if it's a directory, the command adds all the files in that directory recursively.

Staging Modified Files

Let's change a file that was already tracked. If you change a previously tracked file called

`CONTRIBUTING.md` and then run your `git status` command again, you get something that looks like this:

```
$ 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:   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
```

The `CONTRIBUTING.md` file appears under a section named “Changes not staged for commit” – which means that a file that is tracked has been modified in the working directory but not yet staged. To stage it, you run the `git add` command. `git add` is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add this content to the next commit” rather than “add this file to the project”. Let’s run `git add` now to stage the `CONTRIBUTING.md` file, and then run `git status` again:

```
$ git add CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md
```

Both files are staged and will go into your next commit. At this point, suppose you remember one little change that you want to make in `CONTRIBUTING.md` before you commit it. You open it again and make that change, and you’re ready to commit. However, let’s run `git status` one more time:

```
$ vim CONTRIBUTING.md
$ 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:   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
```

What the heck? Now `CONTRIBUTING.md` is listed as both staged *and* unstaged. How is that possible? It turns out that Git stages a file exactly as it is when you run the `git add` command. If you commit now, the version of `CONTRIBUTING.md` as it was when you last ran the `git add` command is how it will go into the commit, not the version of the file as it looks in your working directory when you run `git commit`. If you modify a file after you run `git add`, you have to run `git add` again to stage the latest version of the file:

```
$ git add CONTRIBUTING.md
$ 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:   README
    modified:   CONTRIBUTING.md
```

Short Status

While the `git status` output is pretty comprehensive, it's also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run `git status -s` or `git status --short` you get a far more simplified output from the command:

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

New files that aren't tracked have a `??` next to them, new files that have been added to the staging area have an `A`, modified files have an `M` and so on. There are two columns to the output - the left-hand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the `README` file is modified in the working directory but not yet staged, while the `lib/simplegit.rb` file is modified and staged. The `Rakefile` was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

Ignoring Files

Often, you'll have a class of files that you don't want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named `.gitignore`. Here is an example `.gitignore` file:

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

The first line tells Git to ignore any files ending in ".o" or ".a" – object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (`~`), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a `.gitignore` file before you get going is generally a good idea so you don't accidentally commit files that you really don't want in your Git repository.

The rules for the patterns you can put in the `.gitignore` file are as follows:

- Blank lines or lines starting with `#` are ignored.
- Standard glob patterns work.
- You can start patterns with a forward slash (`/`) to avoid recursivity.
- You can end patterns with a forward slash (`/`) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (`!`).

Glob patterns are like simplified regular expressions that shells use. An asterisk (`*`) matches zero or more characters; `[abc]` matches any character inside the brackets (in this case a, b, or c); a question mark (`?`) matches a single character; and brackets enclosing characters separated by a hyphen (`[0-9]`) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; `a/**/z` would match `a/z`, `a/b/z`, `a/b/c/z`, and so on.

Here is another example .gitignore file:

```
# 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 .pdf files in the doc/ directory
doc/**/*.pdf
```

TIP	GitHub maintains a fairly comprehensive list of good `.gitignore` file examples for dozens of projects and languages at https://github.com/github/gitignore if you want a starting point for your project.

Viewing Your Staged and Unstaged Changes

If the `git status` command is too vague for you – you want to know exactly what you changed, not just which files were changed – you can use the `git diff` command. We'll cover `git diff` in more detail later, but you'll probably use it most often to answer these two questions: What have you changed but not yet staged? And what have you staged that you are about to commit? Although `git status` answers those questions very generally by listing the file names, `git diff` shows you the exact lines added and removed – the patch, as it were.

Let's say you edit and stage the `README` file again and then edit the `CONTRIBUTING.md` file without staging it. If you run your `git status` command, you once again see something like this:

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

    modified:   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
```

To see what you've changed but not yet staged, type `git diff` with no other arguments:

```
$ 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
```

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you've made that you haven't yet staged.

If you want to see what you've staged that will go into your next commit, you can use `git diff --staged`. This command compares your staged changes to your last commit:

```
$ 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
```

It's important to note that `git diff` by itself doesn't show all changes made since your last commit – only changes that are still unstaged. This can be confusing, because if you've staged all of your changes, `git diff` will give you no output.

For another example, if you stage the `CONTRIBUTING.md` file and then edit it, you can use `git diff` to see the changes in the file that are staged and the changes that are unstaged. If our environment looks like this:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   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
```

Now you can use `git diff` to see what is still unstaged:

```
$ 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
```

and `git diff --cached` to see what you've staged so far (`--staged` and `--cached` are synonyms):

```
$ 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
```

NOTE

Git Diff in an External Tool

We will continue to use the `git diff` command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run `git difftool` instead of `git diff`, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run `git difftool --tool-help` to see what is available on your system.

Committing Your Changes

Now that your staging area is set up the way you want it, you can commit your changes. Remember that anything that is still unstaged – any files you have created or modified that you haven't run `git add` on since you edited them – won't go into this commit. They will stay as modified files on your disk. In this case, let's say that the last time you ran `git status`, you saw that everything was staged, so you're ready to commit your changes. The simplest way to commit is to type `git commit`:

```
$ git commit
```

Doing so launches your editor of choice. (This is set by your shell's `$EDITOR` environment variable – usually vim or emacs, although you can configure it with whatever you want using the `git config --global core.editor` command as you saw in Getting Started).

The editor displays the following text (this example is a Vim screen):

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

You can see that the default commit message contains the latest output of the `git status` command commented out and one empty line on top. You can remove these comments and type your commit message, or you can leave them there to help you remember what you're committing. (For an even more explicit reminder of what you've modified, you can pass the `-v` option to `git commit`. Doing so also puts the diff of your change in the editor so you can see exactly what changes you're committing.) When you exit the editor, Git creates your commit with that commit message (with the comments and diff stripped out).

Alternatively, you can type your commit message inline with the `commit` command by specifying it

after a `-m` flag, like this:

```
$ 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
```

Now you've created your first commit! You can see that the commit has given you some output about itself: which branch you committed to (`master`), what SHA-1 checksum the commit has (`463dc4f`), how many files were changed, and statistics about lines added and removed in the commit.

Remember that the commit records the snapshot you set up in your staging area. Anything you didn't stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you're recording a snapshot of your project that you can revert to or compare to later.

Skipping the Staging Area

Although it can be amazingly useful for crafting commits exactly how you want them, the staging area is sometimes a bit more complex than you need in your workflow. If you want to skip the staging area, Git provides a simple shortcut. Adding the `-a` option to the `git commit` command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the `git add` part:

```
$ 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:   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(-)
```

Notice how you don't have to run `git add` on the `CONTRIBUTING.md` file in this case before you commit. That's because the `-a` flag includes all changed files. This is convenient, but be careful; sometimes this flag will cause you to include unwanted changes.

Removing Files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The `git rm` command does that, and also removes the file

from your working directory so you don't see it as an untracked file the next time around.

If you simply remove the file from your working directory, it shows up under the "Changed but not updated" (that is, *unstaged*) area of your `git status` output:

```
$ 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")
```

Then, if you run `git rm`, it stages the file's removal:

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

    deleted:    PROJECTS.md
```

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the index already, you must force the removal with the `-f` option. This is a safety feature to prevent accidental removal of data that hasn't yet been recorded in a snapshot and that can't be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your `.gitignore` file and accidentally staged it, like a large log file or a bunch of `.a` compiled files. To do this, use the `--cached` option:

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

You can pass files, directories, and file-glob patterns to the `git rm` command. That means you can do things such as:

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

Note the backslash (`\`) in front of the `*`. This is necessary because Git does its own filename expansion in addition to your shell's filename expansion. This command removes all files that have the `.log` extension in the `log/` directory. Or, you can do something like this:

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

This command removes all files whose names end with a `~`.

Moving Files

Unlike many other VCS systems, Git doesn't explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact – we'll deal with detecting file movement a bit later.

Thus it's a bit confusing that Git has a `mv` command. If you want to rename a file in Git, you can run something like:

```
$ git mv file_from file_to
```

and it works fine. In fact, if you run something like this and look at the status, you'll see that Git considers it a renamed file:

```
$ git mv README.md README
$ 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)

    renamed:    README.md -> README
```

However, this is equivalent to running something like this:

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

Git figures out that it's a rename implicitly, so it doesn't matter if you rename a file that way or with the `mv` command. The only real difference is that `git mv` is one command instead of three – it's a convenience function. More importantly, you can use any tool you like to rename a file, and address the add/rm later, before you commit.

Viewing the Commit History

After you have created several commits, or if you have cloned a repository with an existing commit

history, you'll probably want to look back to see what has happened. The most basic and powerful tool to do this is the `git log` command.

These examples use a very simple project called "simplegit". To get the project, run

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

When you run `git log` in this project, you should get output that looks something like this:

```
$ 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
```

By default, with no arguments, `git log` lists the commits made in that repository in reverse chronological order – that is, the most recent commits show up first. As you can see, this command lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

A huge number and variety of options to the `git log` command are available to show you exactly what you're looking for. Here, we'll show you some of the most popular.

One of the more helpful options is `-p`, which shows the difference introduced in each commit. You can also use `-2`, which limits the output to only the last two entries:

```
$ 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
```

This option displays the same information but with a diff directly following each entry. This is very helpful for code review or to quickly browse what happened during a series of commits that a collaborator has added. You can also use a series of summarizing options with `git log`. For example, if you want to see some abbreviated stats for each commit, you can use the `--stat` option:

```
$ 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(+)
```

As you can see, the `--stat` option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is `--pretty`. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The `oneline` option prints each commit on a single line, which is useful if you're looking at a lot of commits. In addition, the `short`, `full`, and `fuller` options show the output in roughly the same format but with less or more information, respectively:

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

The most interesting option is `format`, which allows you to specify your own log output format. This is especially useful when you're generating output for machine parsing – because you specify the format explicitly, you know it won't change with updates to Git:

```
$ 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` lists some of the more useful options that `format` takes.

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

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

You may be wondering what the difference is between *author* and *committer*. The author is the person who originally wrote the work, whereas the committer is the person who last applied the work. So, if you send in a patch to a project and one of the core members applies the patch, both of you get credit – you as the author, and the core member as the committer. We'll cover this distinction a bit more in [_distributed_git].

The `oneline` and `format` options are particularly useful with another `log` option called `--graph`. This option adds a nice little ASCII graph showing your branch and merge history:

```
$ 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
```

This type of output will become more interesting as we go through branching and merging in the next chapter.

Those are only some simple output-formatting options to `git log` – there are many more. Common options to `git log` lists the options we've covered so far, as well as some other common formatting options that may be useful, along with how they change the output of the log command.

Table 2. Common options to `git log`

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

Limiting Log Output

In addition to output-formatting options, `git log` takes a number of useful limiting options – that is, options that let you show only a subset of commits. You've seen one such option already – the `-2` option, which show only the last two commits. In fact, you can do `-<n>`, where `n` is any integer to show the last `n` commits. In reality, you're unlikely to use that often, because Git by default pipes all output through a pager so you see only one page of log output at a time.

However, the time-limiting options such as `--since` and `--until` are very useful. For example, this command gets the list of commits made in the last two weeks:

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

This command works with lots of formats – you can specify a specific date like `"2008-01-15"`, or a relative date such as `"2 years 1 day 3 minutes ago"`.

You can also filter the list to commits that match some search criteria. The `--author` option allows you to filter on a specific author, and the `--grep` option lets you search for keywords in the commit messages. (Note that if you want to specify both author and grep options, you have to add `--all-match` or the command will match commits with either.)

Another really helpful filter is the `-S` option which takes a string and only shows the commits that introduced a change to the code that added or removed that string. For instance, if you wanted to find the last commit that added or removed a reference to a specific function, you could call:

```
$ git log -Sfunction_name
```

The last really useful option to pass to `git log` as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (`--`) to separate the paths from the options.

In Options to limit the output of `git log` we'll list these and a few other common options for your reference.

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

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

For example, if you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano in the month of October 2008 and are not merge commits, you can run something like this:

```
$ 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
```

Of the nearly 40,000 commits in the Git source code history, this command shows the 6 that match those criteria.

Undoing Things

At any stage, you may want to undo something. Here, we'll review a few basic tools for undoing changes that you've made. Be careful, because you can't always undo some of these undos. This is one of the few areas in Git where you may lose some work if you do it wrong.

One of the common undos takes place when you commit too early and possibly forget to add some files, or you mess up your commit message. If you want to try that commit again, you can run commit with the `--amend` option:

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

This command takes your staging area and uses it for the commit. If you've made no changes since your last commit (for instance, you run this command immediately after your previous commit), then your snapshot will look exactly the same, and all you'll change is your commit message.

The same commit-message editor fires up, but it already contains the message of your previous commit. You can edit the message the same as always, but it overwrites your previous commit.

As an example, if you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

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

You end up with a single commit – the second commit replaces the results of the first.

Unstaging a Staged File

The next two sections demonstrate how to work with your staging area and working directory changes. The nice part is that the command you use to determine the state of those two areas also reminds you how to undo changes to them. For example, let's say you've changed two files and

want to commit them as two separate changes, but you accidentally type `git add *` and stage them both. How can you unstage one of the two? The `git status` command reminds you:

```
$ 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
```

Right below the “Changes to be committed” text, it says use `git reset HEAD <file>...` to unstage. So, let’s use that advice to unstage the `CONTRIBUTING.md` file:

```
$ 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
```

The command is a bit strange, but it works. The `CONTRIBUTING.md` file is modified but once again unstaged.

NOTE It’s true that `git reset` can be a dangerous command, especially if you provide the `--hard` flag. However, in the scenario described above, the file in your working directory is not touched, so it’s relatively safe.

For now this magic invocation is all you need to know about the `git reset` command. We’ll go into much more detail about what `reset` does and how to master it to do really interesting things in [_git_reset].

Unmodifying a Modified File

What if you realize that you don’t want to keep your changes to the `CONTRIBUTING.md` file? How can you easily unmodify it – revert it back to what it looked like when you last committed (or initially cloned, or however you got it into your working directory)? Luckily, `git status` tells you how to do

that, too. In the last example output, the unstaged area looks like this:

```
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
```

It tells you pretty explicitly how to discard the changes you've made. Let's do what it says:

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

    renamed:    README.md -> README
```

You can see that the changes have been reverted.

IMPORTANT	It's important to understand that `git checkout -- <file>` is a dangerous command. Any changes you made to that file are gone – Git just copied another file over it. Don't ever use this command unless you absolutely know that you don't want the file.

If you would like to keep the changes you've made to that file but still need to get it out of the way for now, we'll go over stashing and branching in Git Branching; these are generally better ways to go.

Remember, anything that is *committed* in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an `--amend` commit can be recovered (see [_data_recovery] for data recovery). However, anything you lose that was never committed is likely never to be seen again.

Working with Remotes

To be able to collaborate on any Git project, you need to know how to manage your remote repositories. Remote repositories are versions of your project that are hosted on the Internet or network somewhere. You can have several of them, each of which generally is either read-only or read/write for you. Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work. Managing remote repositories includes knowing how to add remote repositories, remove remotes that are no longer valid, manage various remote branches and define them as being tracked or not, and more. In this section, we'll cover some of these remote-management skills.

Showing Your Remotes

To see which remote servers you have configured, you can run the `git remote` command. It lists the shortnames of each remote handle you've specified. If you've cloned your repository, you should at least see origin – that is the default name Git gives to the server you cloned from:

```
$ 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
```

You can also specify `-v`, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

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

If you have more than one remote, the command lists them all. For example, a repository with multiple remotes for working with several collaborators might look something like this.

```
$ 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)
```

This means we can pull contributions from any of these users pretty easily. We may additionally have permission to push to one or more of these, though we can't tell that here.

Notice that these remotes use a variety of protocols; we'll cover more about this in Git on the Server.

Adding Remote Repositories

We've mentioned and given some demonstrations of how the *clone* command implicitly adds the `origin` remote for you. Here's how to add a new remote explicitly. To add a new remote Git repository as a shortname you can reference easily, run `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)
```

Now you can use the string `pb` on the command line in lieu of the whole URL. For example, if you want to fetch all the information that Paul has but that you don't yet have in your repository, you can run `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
```

Paul's master branch is now accessible locally as `pb/master` – you can merge it into one of your branches, or you can check out a local branch at that point if you want to inspect it. (We'll go over what branches are and how to use them in much more detail in Git Branching.)

Fetching and Pulling from Your Remotes

As you just saw, to get data from your remote projects, you can run:

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

The command goes out to that remote project and pulls down all the data from that remote project that you don't have yet. After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time.

If you clone a repository, the command automatically adds that remote repository under the name "origin". So, `git fetch origin` fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It's important to note that the `git fetch` command only downloads the data to your local repository – it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to merge it manually into your work when you're

ready.

If your current branch is set up to track a remote branch (see the next section and Git Branching for more information), you can use the `git pull` command to automatically fetch and then merge that remote branch into your current branch. This may be an easier or more comfortable workflow for you; and by default, the `git clone` command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running `git pull` generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you're currently working on.

Pushing to Your Remotes

When you have your project at a point that you want to share, you have to push it upstream. The command for this is simple: `git push [remote-name] [branch-name]`. If you want to push your master branch to your `origin` server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you've done back up to the server:

```
$ git push origin master
```

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You'll have to fetch their work first and incorporate it into yours before you'll be allowed to push. See Git Branching for more detailed information on how to push to remote servers.

Inspecting a Remote

If you want to see more information about a particular remote, you can use the `git remote show [remote-name]` command. If you run this command with a particular shortname, such as `origin`, you get something like this:

```
$ 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)
```

It lists the URL for the remote repository as well as the tracking branch information. The command helpfully tells you that if you're on the master branch and you run `git pull`, it will automatically merge in the master branch on the remote after it fetches all the remote references. It also lists all

the remote references it has pulled down.

That is a simple example you're likely to encounter. When you're using Git more heavily, however, you may see much more information from `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)
```

This command shows which branch is automatically pushed to when you run `git push` while on certain branches. It also shows you which remote branches on the server you don't yet have, which remote branches you have that have been removed from the server, and multiple local branches that are able to merge automatically with their remote-tracking branch when you run `git pull`.

Removing and Renaming Remotes

You can run `git remote rename` to change a remote's shortname. For instance, if you want to rename `pb` to `paul`, you can do so with `git remote rename`:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

It's worth mentioning that this changes all your remote-tracking branch names, too. What used to be referenced at `pb/master` is now at `paul/master`.

If you want to remove a remote for some reason – you've moved the server or are no longer using a

particular mirror, or perhaps a contributor isn't contributing anymore – you can either use `git remote remove` or `git remote rm`:

```
$ git remote remove paul
$ git remote
origin
```

Tagging

Like most VCSs, Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on). In this section, you'll learn how to list the available tags, how to create new tags, and what the different types of tags are.

Listing Your Tags

Listing the available tags in Git is straightforward. Just type `git tag`:

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

This command lists the tags in alphabetical order; the order in which they appear has no real importance.

You can also search for tags with a particular pattern. The Git source repo, for instance, contains more than 500 tags. If you're only interested in looking at the 1.8.5 series, you can run this:

```
$ 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
```

Creating Tags

Git uses two main types of tags: lightweight and annotated.

A lightweight tag is very much like a branch that doesn't change – it's just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. They're checksummed; contain the tagger name, email, and date; have a tagging message; and can be signed and verified with GNU Privacy Guard (GPG). It's generally recommended that you create annotated tags so you can have all this information; but if you want a temporary tag or for some reason don't want to keep the other information, lightweight tags are available too.

Annotated Tags

Creating an annotated tag in Git is simple. The easiest way is to specify `-a` when you run the `tag` command:

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

The `-m` specifies a tagging message, which is stored with the tag. If you don't specify a message for an annotated tag, Git launches your editor so you can type it in.

You can see the tag data along with the commit that was tagged by using the `git show` command:

```
$ 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
```

That shows the tagger information, the date the commit was tagged, and the annotation message before showing the commit information.

Lightweight Tags

Another way to tag commits is with a lightweight tag. This is basically the commit checksum stored in a file – no other information is kept. To create a lightweight tag, don't supply the `-a`, `-s`, or `-m` option:

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

This time, if you run `git show` on the tag, you don't see the extra tag information. The command just shows the commit:

```
$ 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
```

Tagging Later

You can also tag commits after you've moved past them. Suppose your commit history looks like this:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Now, suppose you forgot to tag the project at v1.2, which was at the "updated rakefile" commit. You can add it after the fact. To tag that commit, you specify the commit checksum (or part of it) at the end of the command:

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

You can see that you've tagged the commit:

```
$ 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
...
```

Sharing Tags

By default, the `git push` command doesn't transfer tags to remote servers. You will have to explicitly push tags to a shared server after you have created them. This process is just like sharing remote branches – you can run `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
```

If you have a lot of tags that you want to push up at once, you can also use the `--tags` option to the `git push` command. This will transfer all of your tags to the remote server that are not already there.

```
$ 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
```

Now, when someone else clones or pulls from your repository, they will get all your tags as well.

Checking out Tags

You can't really check out a tag in Git, since they can't be moved around. If you want to put a version of your repository in your working directory that looks like a specific tag, you can create a new branch at a specific tag with `git checkout -b [branchname] [tagname]`:

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

Of course if you do this and do a commit, your `version2` branch will be slightly different than your `v2.0.0` tag since it will move forward with your new changes, so do be careful.

Unresolved directive in book/02-students/1-students.asc - include::sections/aliases.asc[]

Summary

At this point, you can do all the basic local Git operations – creating or cloning a repository, making changes, staging and committing those changes, and viewing the history of all the changes the repository has been through. Next, we'll cover Git's killer feature: its branching model.

Git Branching

Nearly every VCS has some form of branching support. Branching means you diverge from the main line of development and continue to do work without messing with that main line. In many VCS tools, this is a somewhat expensive process, often requiring you to create a new copy of your source code directory, which can take a long time for large projects.

Some people refer to Git's branching model as its "killer feature," and it certainly sets Git apart in the VCS community. Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Unlike many other VCSs, Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop.

Branches in a Nutshell

To really understand the way Git does branching, we need to take a step back and examine how Git stores its data.

As you may remember from Getting Started, Git doesn't store data as a series of changesets or differences, but instead as a series of snapshots.

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author's name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.

To visualize this, let's assume that you have a directory containing three files, and you stage them all and commit. Staging the files computes a checksum for each one (the SHA-1 hash we mentioned in Getting Started), stores that version of the file in the Git repository (Git refers to them as blobs), and adds that checksum to the staging area:

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

When you create the commit by running `git commit`, Git checksums each subdirectory (in this case, just the root project directory) and stores those tree objects in the Git repository. Git then creates a commit object that has the metadata and a pointer to the root project tree so it can re-create that snapshot when needed.

Your Git repository now contains five objects: one blob for the contents of each of your three files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.

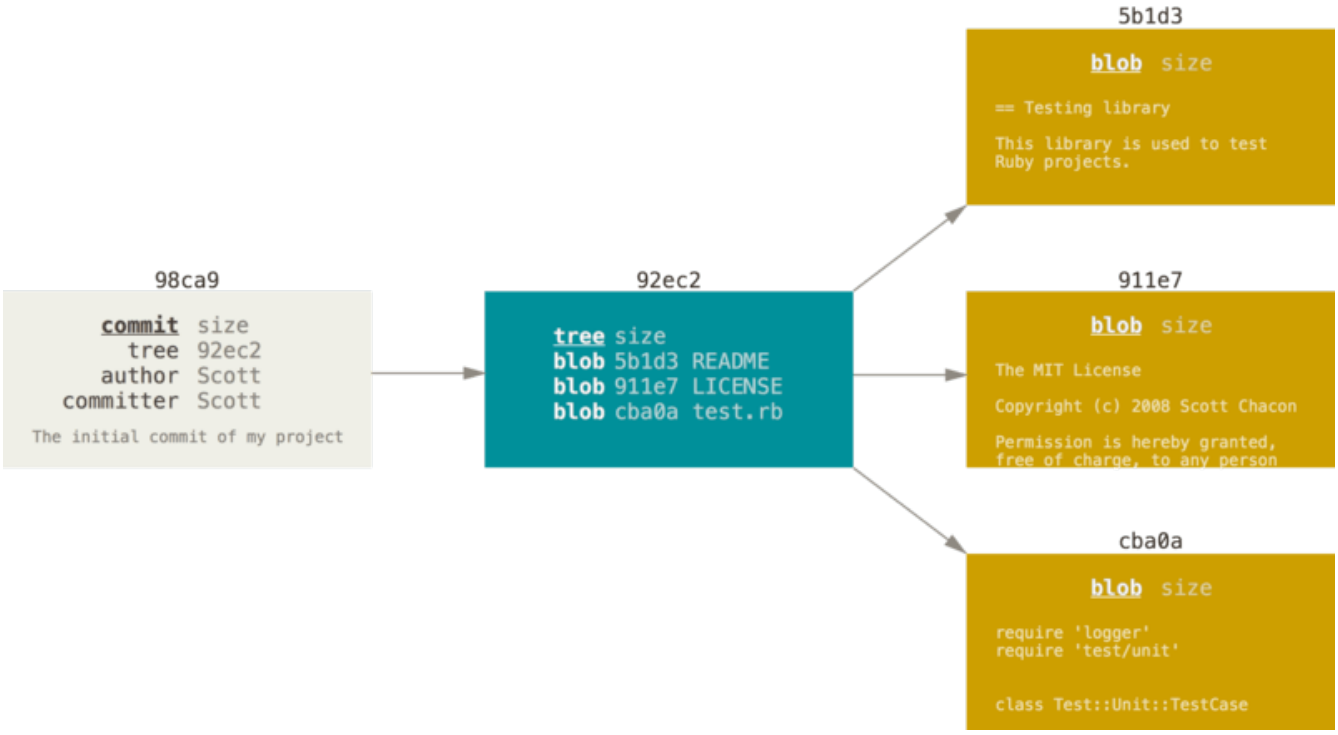

Figure 2. A commit and its tree

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.

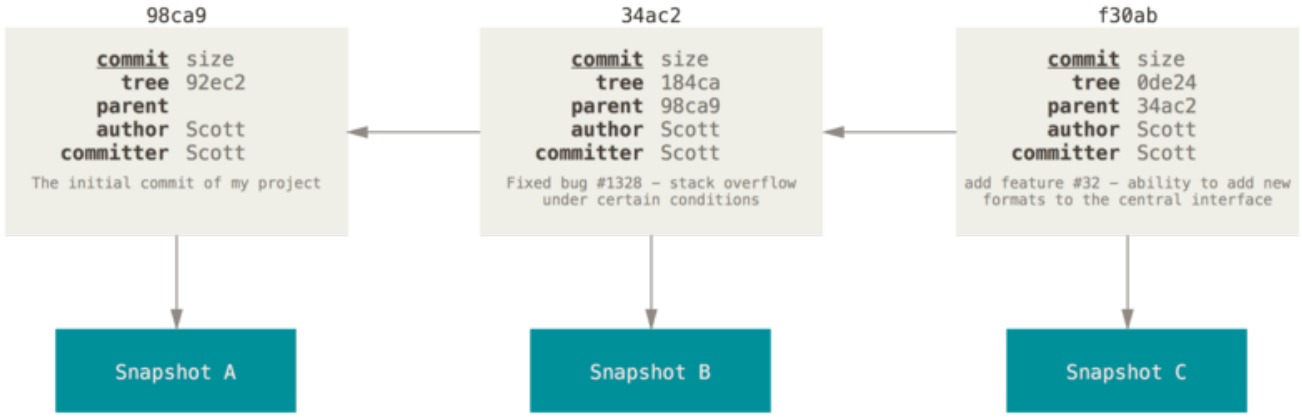

Figure 3. Commits and their parents

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is `master`. As you start making commits, you're given a `master` branch that points to the last commit you made. Every time you commit, it moves forward automatically.

NOTE The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the `git init` command creates it by default and most people don't bother to change it.

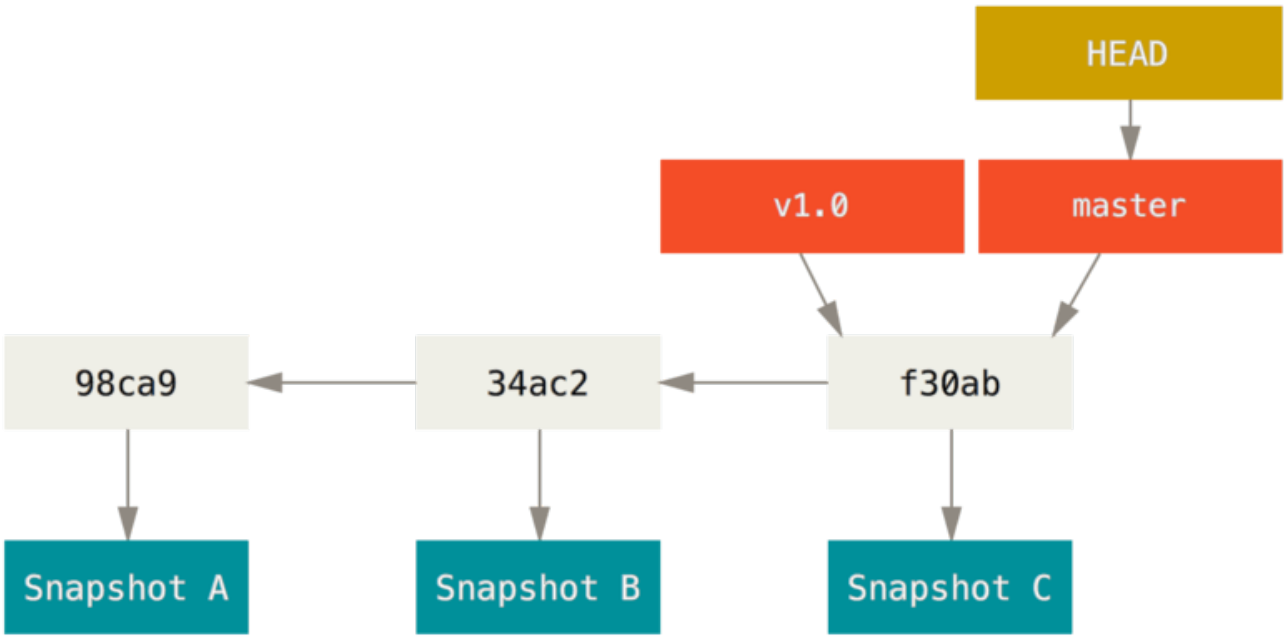

Figure 4. A branch and its commit history

Creating a New Branch

What happens if you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you create a new branch called testing. You do this with the `git branch` command:

```
$ git branch testing
```

This creates a new pointer to the same commit you're currently on.

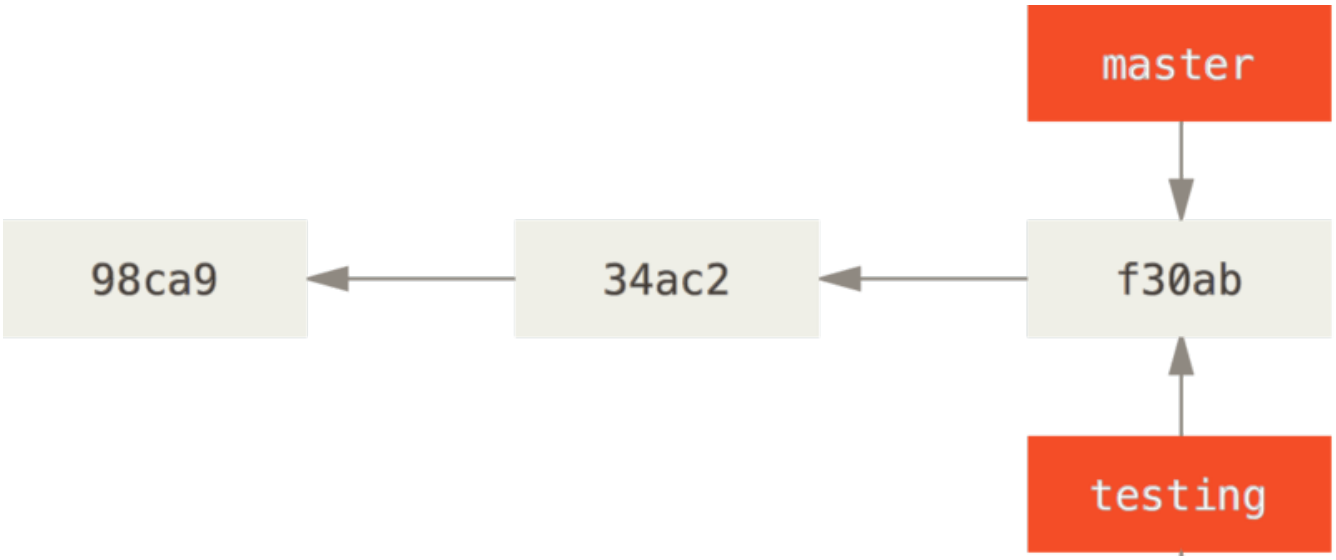

Figure 5. Two branches pointing into the same series of commits

How does Git know what branch you're currently on? It keeps a special pointer called `HEAD`. Note that this is a lot different than the concept of `HEAD` in other VCSs you may be used to, such as Subversion or CVS. In Git, this is a pointer to the local branch you're currently on. In this case, you're still on `master`. The `git branch` command only *created* a new branch – it didn't switch to that

branch.

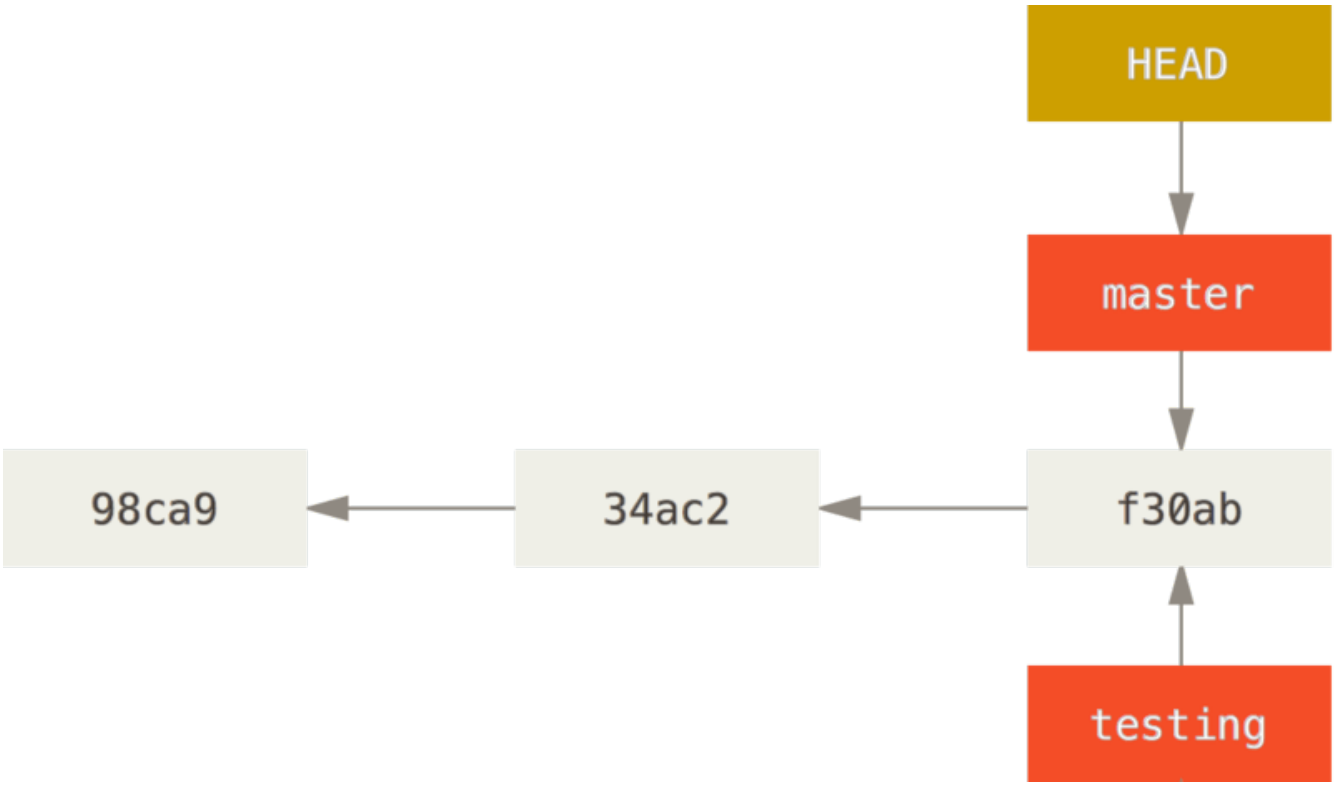

Figure 6. HEAD pointing to a branch

You can easily see this by running a simple `git log` command that shows you where the branch pointers are pointing. This option is called `--decorate`.

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

You can see the “master” and “testing” branches that are right there next to the `f30ab` commit.

Switching Branches

To switch to an existing branch, you run the `git checkout` command. Let’s switch to the new `testing` branch:

```
$ git checkout testing
```

This moves `HEAD` to point to the `testing` branch.

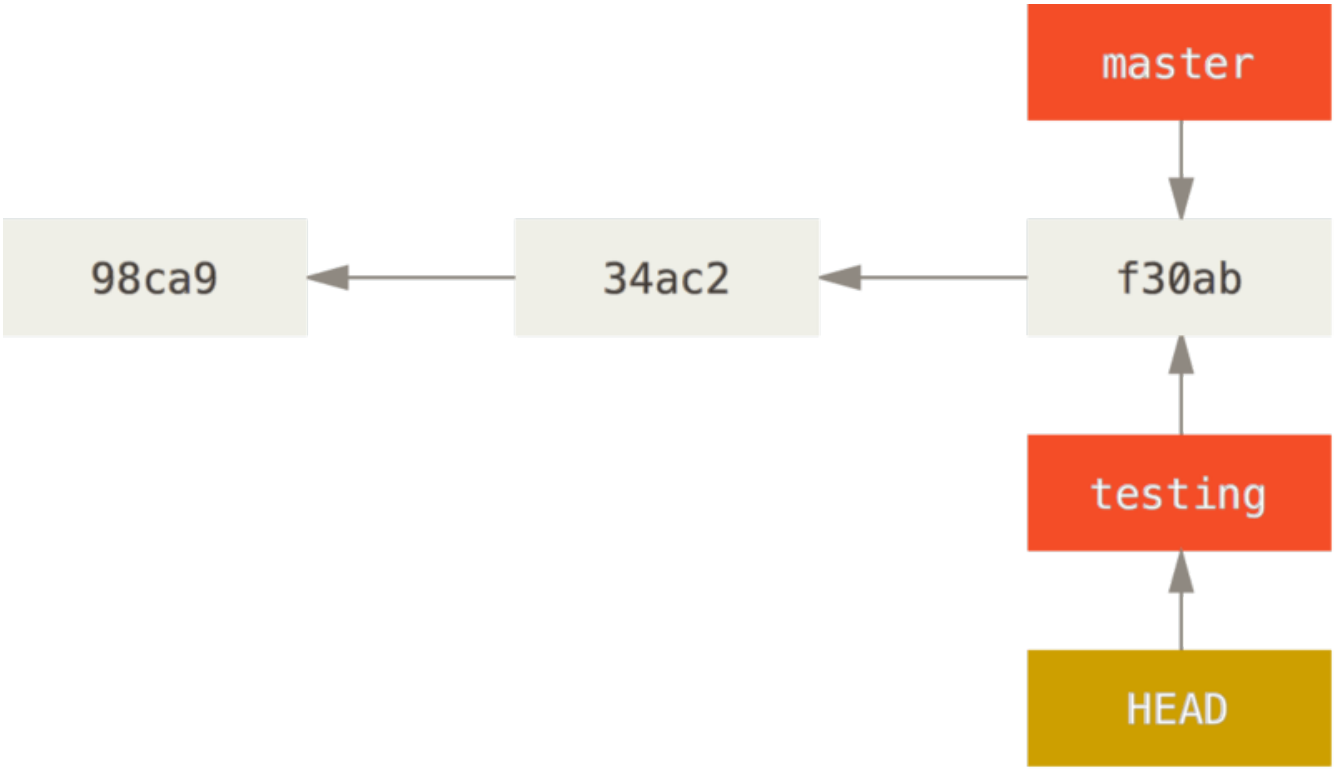

Figure 7. HEAD points to the current branch

What is the significance of that? Well, let's do another commit:

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

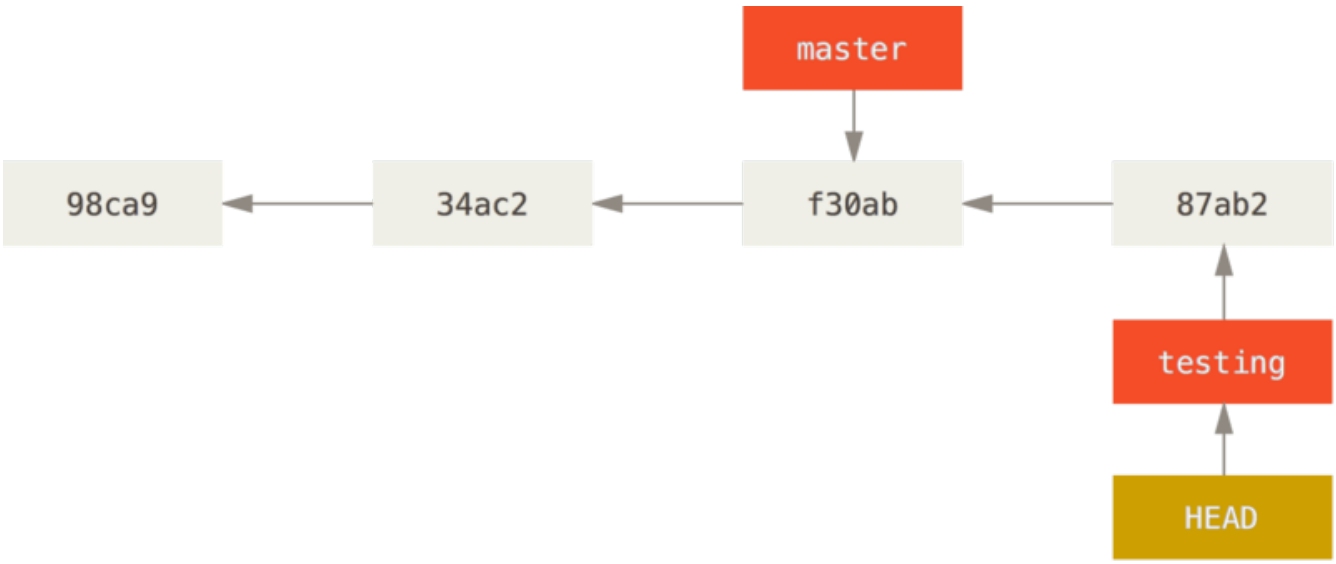

Figure 8. The HEAD branch moves forward when a commit is made

This is interesting, because now your `testing` branch has moved forward, but your `master` branch still points to the commit you were on when you ran `git checkout` to switch branches. Let's switch back to the `master` branch:

```
$ git checkout master
```

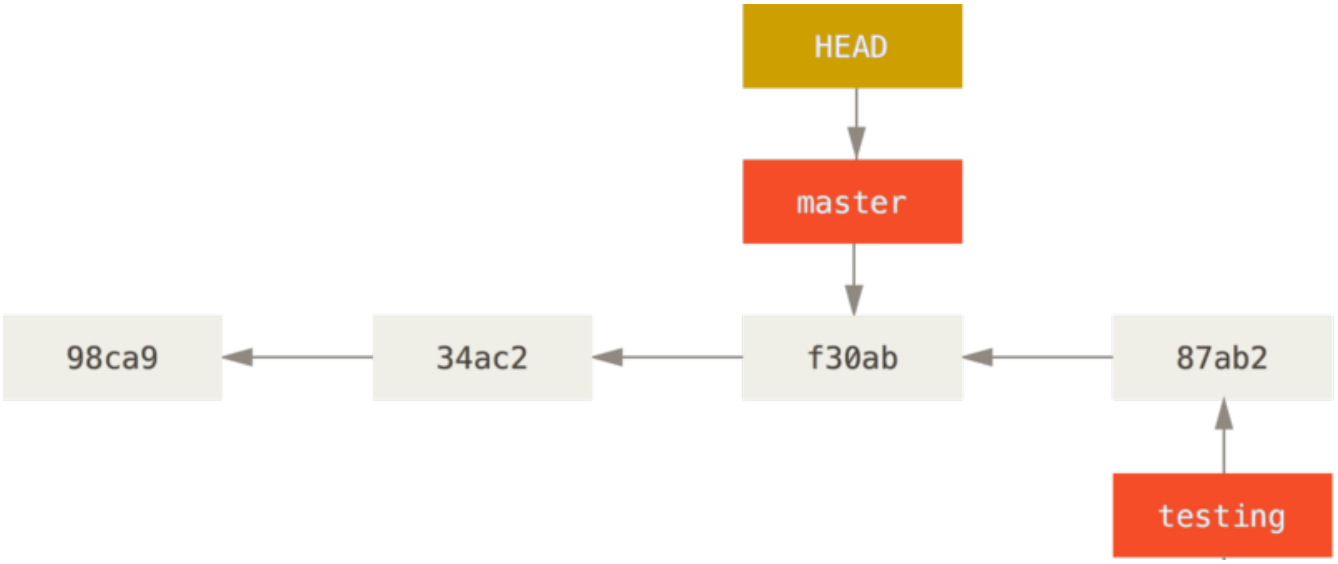

Figure 9. HEAD moves when you checkout

That command did two things. It moved the HEAD pointer back to point to the `master` branch, and it reverted the files in your working directory back to the snapshot that `master` points to. This also means the changes you make from this point forward will diverge from an older version of the project. It essentially rewinds the work you've done in your `testing` branch so you can go in a different direction.

NOTE

Switching branches changes files in your working directory

It's important to note that when you switch branches in Git, files in your working directory will change. If you switch to an older branch, your working directory will be reverted to look like it did the last time you committed on that branch. If Git cannot do it cleanly, it will not let you switch at all.

Let's make a few changes and commit again:

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

Now your project history has diverged (see Divergent history). You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple `branch`, `checkout`, and `commit` commands.

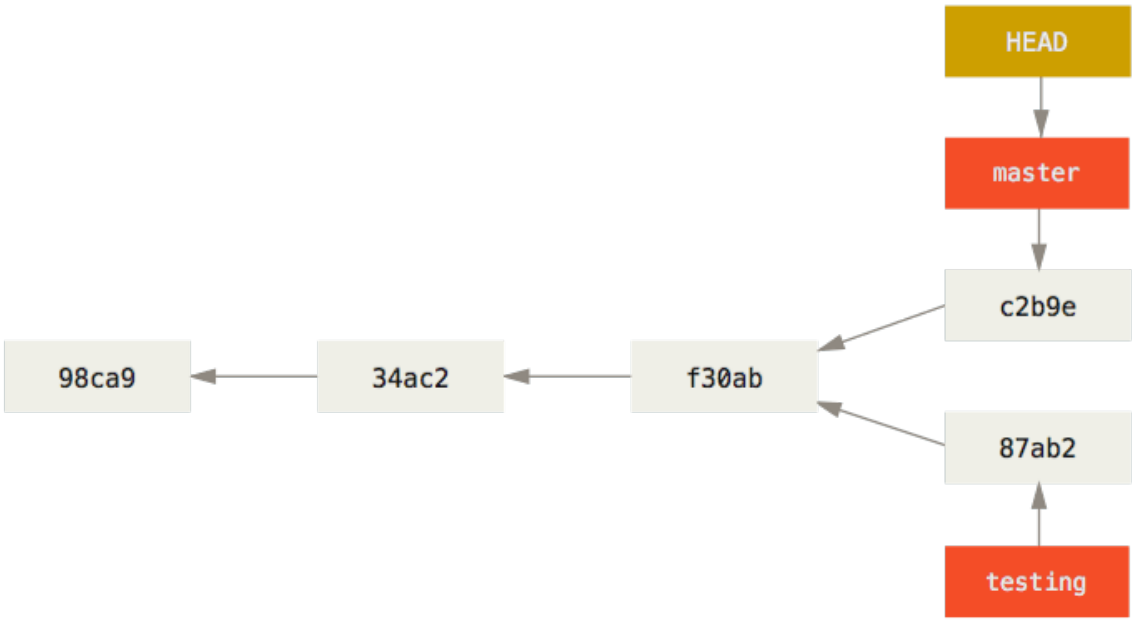

Figure 10. Divergent history

You can also see this easily with the `git log` command. If you run `git log --oneline --decorate --graph --all` it will print out the history of your commits, showing where your branch pointers are and how your history has diverged.

```
$ 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
```

Because a branch in Git is actually a simple file that contains the 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file (40 characters and a newline).

This is in sharp contrast to the way most older VCS tools branch, which involves copying all of the project's files into a second directory. This can take several seconds or even minutes, depending on the size of the project, whereas in Git the process is always instantaneous. Also, because we're recording the parents when we commit, finding a proper merge base for merging is automatically done for us and is generally very easy to do. These features help encourage developers to create and use branches often.

Let's see why you should do so.

Basic Branching and Merging

Let's go through a simple example of branching and merging with a workflow that you might use in the real world. You'll follow these steps:

1. Do work on a website.
2. Create a branch for a new story you're working on.
3. Do some work in that branch.

At this stage, you'll receive a call that another issue is critical and you need a hotfix. You'll do the following:

1. Switch to your production branch.
2. Create a branch to add the hotfix.
3. After it's tested, merge the hotfix branch, and push to production.
4. Switch back to your original story and continue working.

Basic Branching

First, let's say you're working on your project and have a couple of commits already.

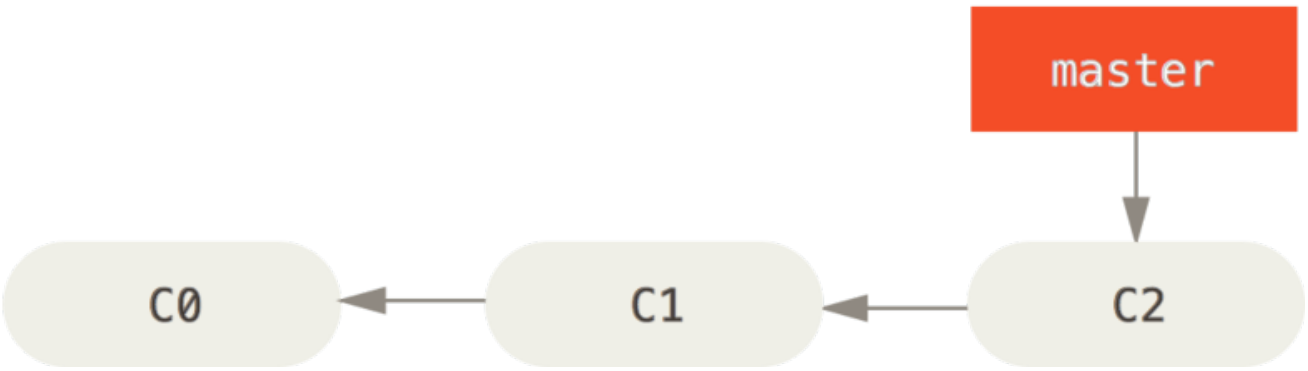

Figure 11. A simple commit history

You've decided that you're going to work on issue #53 in whatever issue-tracking system your company uses. To create a branch and switch to it at the same time, you can run the `git checkout` command with the `-b` switch:

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

This is shorthand for:

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

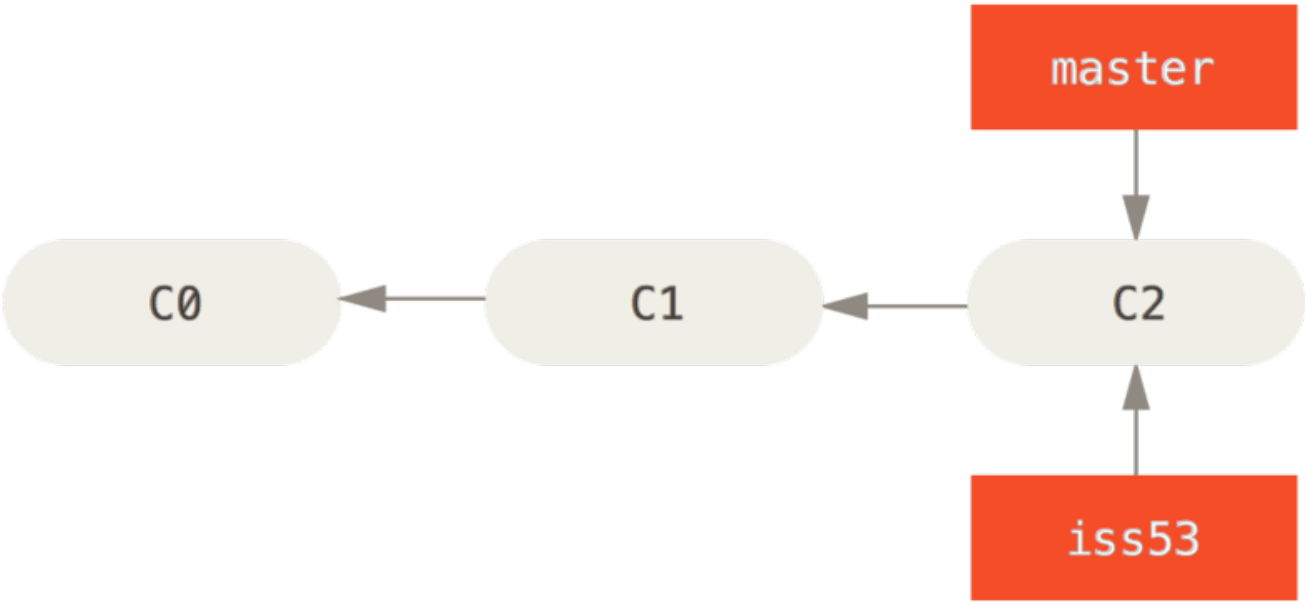

Figure 12. Creating a new branch pointer

You work on your website and do some commits. Doing so moves the `iss53` branch forward, because you have it checked out (that is, your `HEAD` is pointing to it):

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

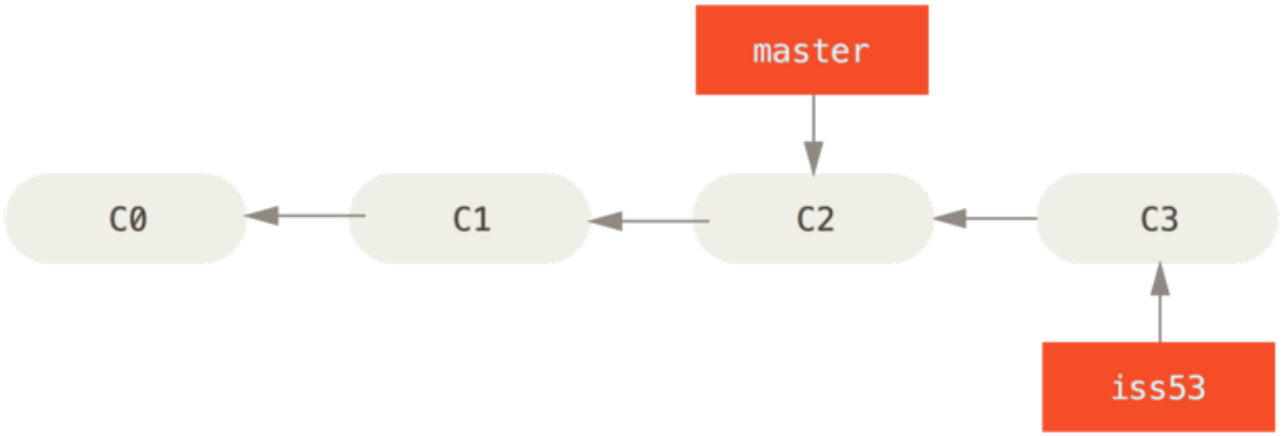

Figure 13. The `iss53` branch has moved forward with your work

Now you get the call that there is an issue with the website, and you need to fix it immediately. With Git, you don't have to deploy your fix along with the `iss53` changes you've made, and you don't have to put a lot of effort into reverting those changes before you can work on applying your fix to what is in production. All you have to do is switch back to your `master` branch.

However, before you do that, note that if your working directory or staging area has uncommitted changes that conflict with the branch you're checking out, Git won't let you switch branches. It's best to have a clean working state when you switch branches. There are ways to get around this (namely, stashing and commit amending) that we'll cover later on, in [_git_stashing]. For now, let's assume you've committed all your changes, so you can switch back to your `master` branch:

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

At this point, your project working directory is exactly the way it was before you started working on issue #53, and you can concentrate on your hotfix. This is an important point to remember: when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

Next, you have a hotfix to make. Let's create a hotfix branch on which to work until it's completed:

```
$ 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(+)
```

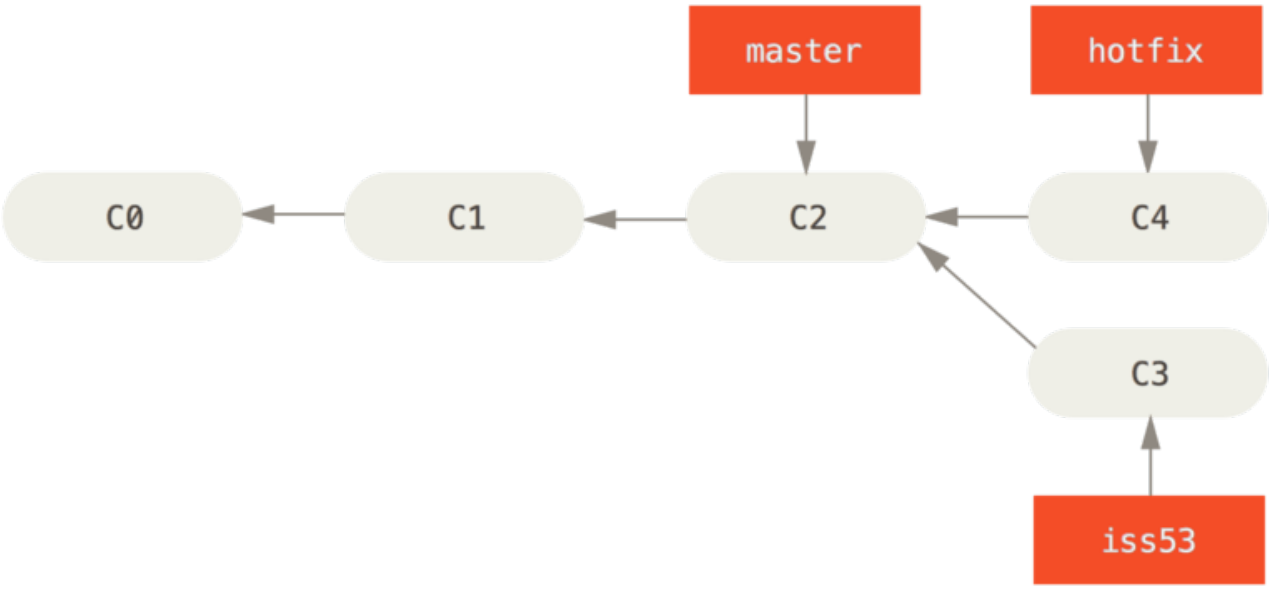

Figure 14. Hotfix branch based on `master`

You can run your tests, make sure the hotfix is what you want, and merge it back into your `master` branch to deploy to production. You do this with the `git merge` command:

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

You'll notice the phrase "fast-forward" in that merge. Because the commit `C4` pointed to by the

branch `hotfix` you merged in was directly ahead of the commit `C2` you're on, Git simply moves the pointer forward. To phrase that another way, when you try to merge one commit with a commit that can be reached by following the first commit's history, Git simplifies things by moving the pointer forward because there is no divergent work to merge together – this is called a "fast-forward."

Your change is now in the snapshot of the commit pointed to by the `master` branch, and you can deploy the fix.

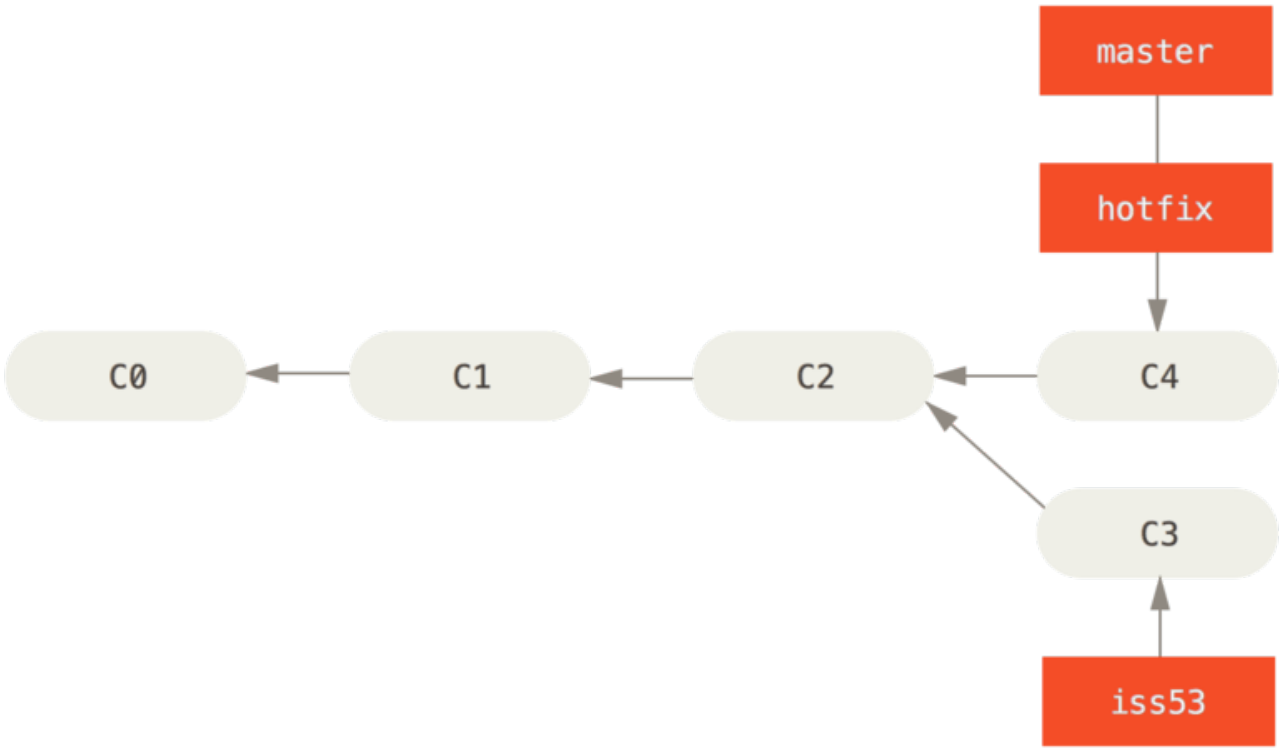

Figure 15. `master` is fast-forwarded to `hotfix`

After your super-important fix is deployed, you're ready to switch back to the work you were doing before you were interrupted. However, first you'll delete the `hotfix` branch, because you no longer need it – the `master` branch points at the same place. You can delete it with the `-d` option to `git branch`:

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

Now you can switch back to your work-in-progress branch on issue #53 and continue working on it.

```
$ 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(+)
```

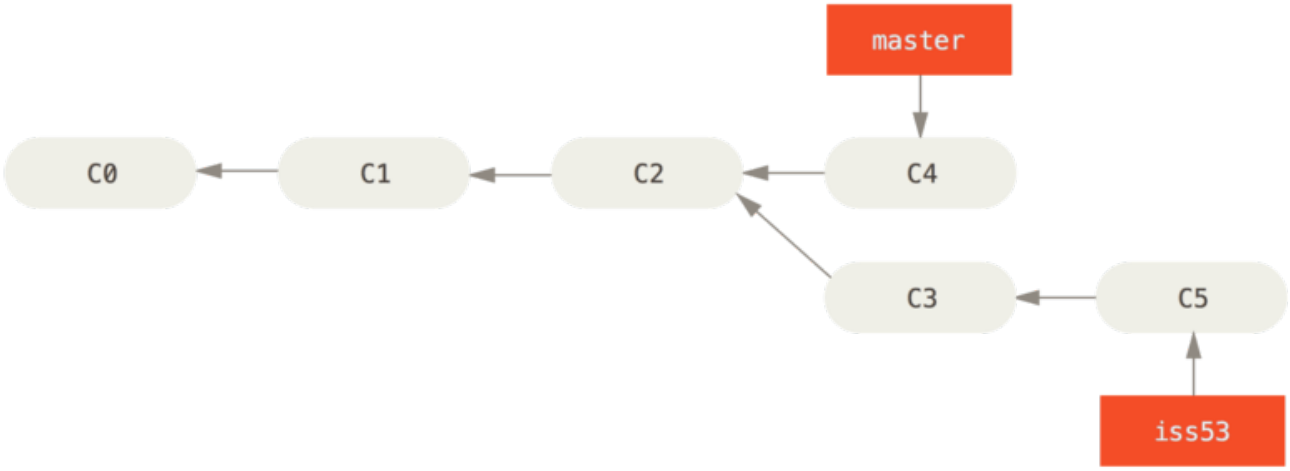

Figure 16. Work continues on `iss53`

It's worth noting here that the work you did in your `hotfix` branch is not contained in the files in your `iss53` branch. If you need to pull it in, you can merge your `master` branch into your `iss53` branch by running `git merge master`, or you can wait to integrate those changes until you decide to pull the `iss53` branch back into `master` later.

Basic Merging

Suppose you've decided that your issue #53 work is complete and ready to be merged into your `master` branch. In order to do that, you'll merge your `iss53` branch into `master`, much like you merged your `hotfix` branch earlier. All you have to do is check out the branch you wish to merge into and then run the `git merge` command:

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

This looks a bit different than the `hotfix` merge you did earlier. In this case, your development history has diverged from some older point. Because the commit on the branch you're on isn't a direct ancestor of the branch you're merging in, Git has to do some work. In this case, Git does a simple three-way merge, using the two snapshots pointed to by the branch tips and the common ancestor of the two.

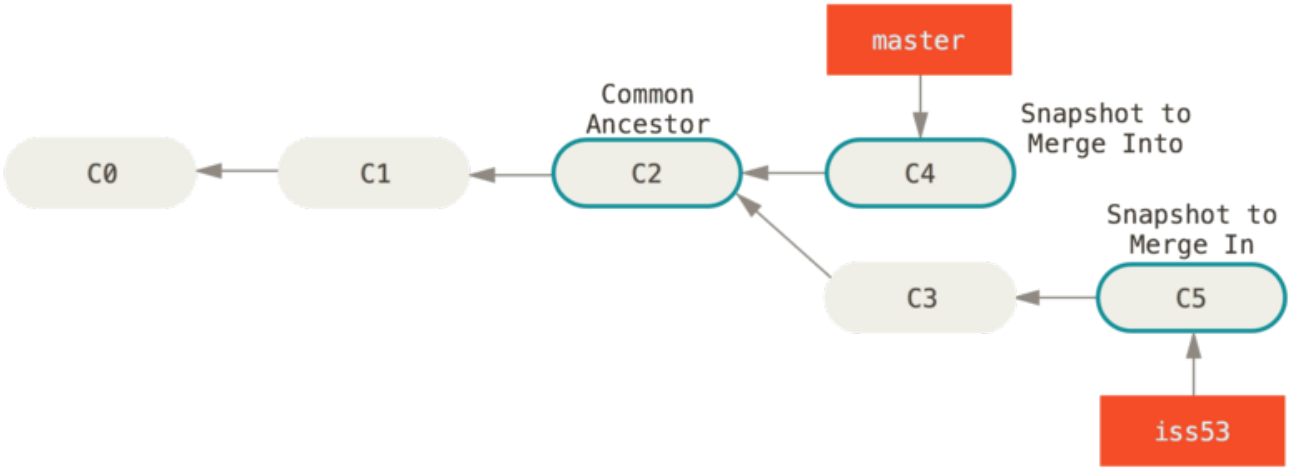

Figure 17. Three snapshots used in a typical merge

Instead of just moving the branch pointer forward, Git creates a new snapshot that results from this three-way merge and automatically creates a new commit that points to it. This is referred to as a merge commit, and is special in that it has more than one parent.

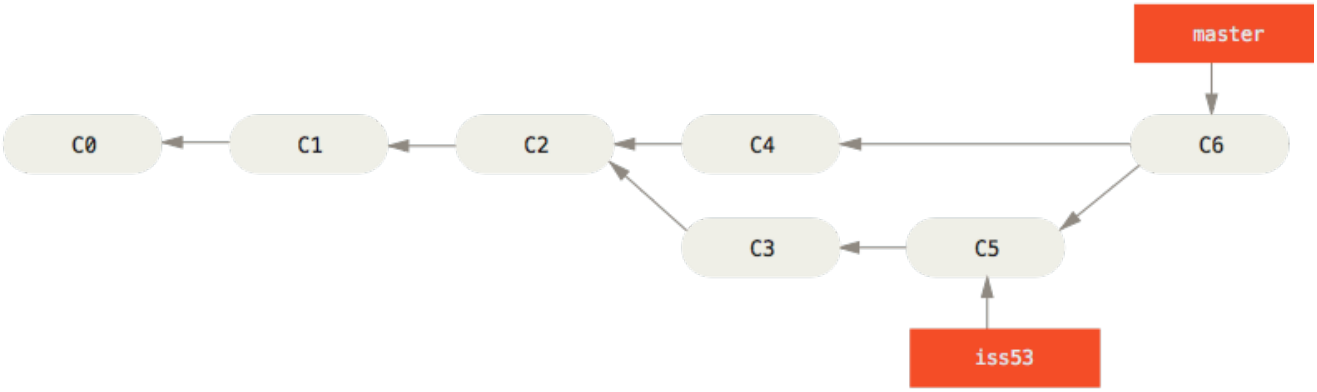

Figure 18. A merge commit

It's worth pointing out that Git determines the best common ancestor to use for its merge base; this is different than older tools like CVS or Subversion (before version 1.5), where the developer doing the merge had to figure out the best merge base for themselves. This makes merging a heck of a lot easier in Git than in these other systems.

Now that your work is merged in, you have no further need for the `iss53` branch. You can close the ticket in your ticket-tracking system, and delete the branch:

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

Basic Merge Conflicts

Occasionally, this process doesn't go smoothly. If you changed the same part of the same file differently in the two branches you're merging together, Git won't be able to merge them cleanly. If your fix for issue #53 modified the same part of a file as the `hotfix`, you'll get a merge conflict that looks something like this:

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

Git hasn't automatically created a new merge commit. It has paused the process while you resolve the conflict. If you want to see which files are unmerged at any point after a merge conflict, you can run `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")
```

Anything that has merge conflicts and hasn't been resolved is listed as unmerged. Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts. Your file contains a section that looks something like this:

```
<<<<<<< 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
```

This means the version in `HEAD` (your `master` branch, because that was what you had checked out when you ran your merge command) is the top part of that block (everything above the `=======`), while the version in your `iss53` branch looks like everything in the bottom part. In order to resolve the conflict, you have to either choose one side or the other or merge the contents yourself. For instance, you might resolve this conflict by replacing the entire block with this:

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

This resolution has a little of each section, and the `<<<<<<<`, `=======`, and `>>>>>>>` lines have been completely removed. After you've resolved each of these sections in each conflicted file, run `git add` on each file to mark it as resolved. Staging the file marks it as resolved in Git.

If you want to use a graphical tool to resolve these issues, you can run `git mergetool`, which fires up an appropriate visual merge tool and walks you through the conflicts:

```
$ 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):
```

If you want to use a merge tool other than the default (Git chose `opendiff` in this case because the command was run on a Mac), you can see all the supported tools listed at the top after "one of the following tools." Just type the name of the tool you'd rather use.

NOTE If you need more advanced tools for resolving tricky merge conflicts, we cover more on merging in [_advanced_merging].

After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you. You can run `git status` again to verify that all conflicts have been resolved:

```
$ 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
```

If you're happy with that, and you verify that everything that had conflicts has been staged, you can type `git commit` to finalize the merge commit. The commit message by default looks something like this:

```
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
#
```

If you think it would be helpful to others looking at this merge in the future, you can modify this commit message with details about how you resolved the merge and explain why you did the changes you made if these are not obvious.

Branch Management

Now that you've created, merged, and deleted some branches, let's look at some branch-management tools that will come in handy when you begin using branches all the time.

The `git branch` command does more than just create and delete branches. If you run it with no arguments, you get a simple listing of your current branches:

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

Notice the `*` character that prefixes the `master` branch: it indicates the branch that you currently have checked out (i.e., the branch that `HEAD` points to). This means that if you commit at this point, the `master` branch will be moved forward with your new work. To see the last commit on each branch, you can run `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
```

The useful `--merged` and `--no-merged` options can filter this list to branches that you have or have not yet merged into the branch you're currently on. To see which branches are already merged into the branch you're on, you can run `git branch --merged`:

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

Because you already merged in `iss53` earlier, you see it in your list. Branches on this list without the `*` in front of them are generally fine to delete with `git branch -d`; you've already incorporated their work into another branch, so you're not going to lose anything.

To see all the branches that contain work you haven't yet merged in, you can run `git branch --no-merged`:

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

This shows your other branch. Because it contains work that isn't merged in yet, trying to delete it with `git branch -d` will fail:

```
$ 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'.
```

If you really do want to delete the branch and lose that work, you can force it with `-D`, as the helpful message points out.

Branching Workflows

Now that you have the basics of branching and merging down, what can or should you do with them? In this section, we'll cover some common workflows that this lightweight branching makes possible, so you can decide if you would like to incorporate it into your own development cycle.

Long-Running Branches

Because Git uses a simple three-way merge, merging from one branch into another multiple times over a long period is generally easy to do. This means you can have several branches that are always open and that you use for different stages of your development cycle; you can merge regularly from some of them into others.

Many Git developers have a workflow that embraces this approach, such as having only code that is entirely stable in their `master` branch – possibly only code that has been or will be released. They have another parallel branch named `develop` or `next` that they work from or use to test stability – it isn't necessarily always stable, but whenever it gets to a stable state, it can be merged into `master`. It's used to pull in topic branches (short-lived branches, like your earlier `iss53` branch) when they're ready, to make sure they pass all the tests and don't introduce bugs.

In reality, we're talking about pointers moving up the line of commits you're making. The stable branches are farther down the line in your commit history, and the bleeding-edge branches are farther up the history.

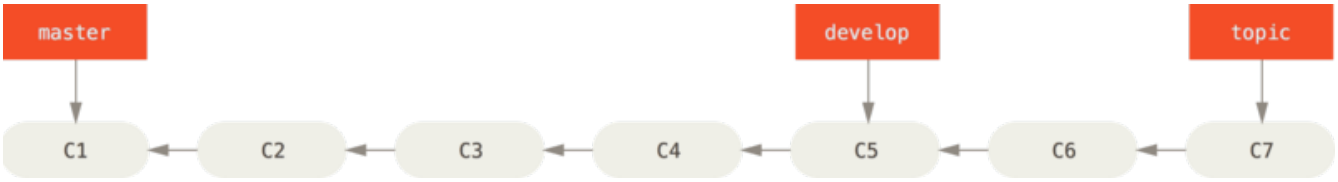

Figure 19. A linear view of progressive-stability branching

It's generally easier to think about them as work silos, where sets of commits graduate to a more stable silo when they're fully tested.

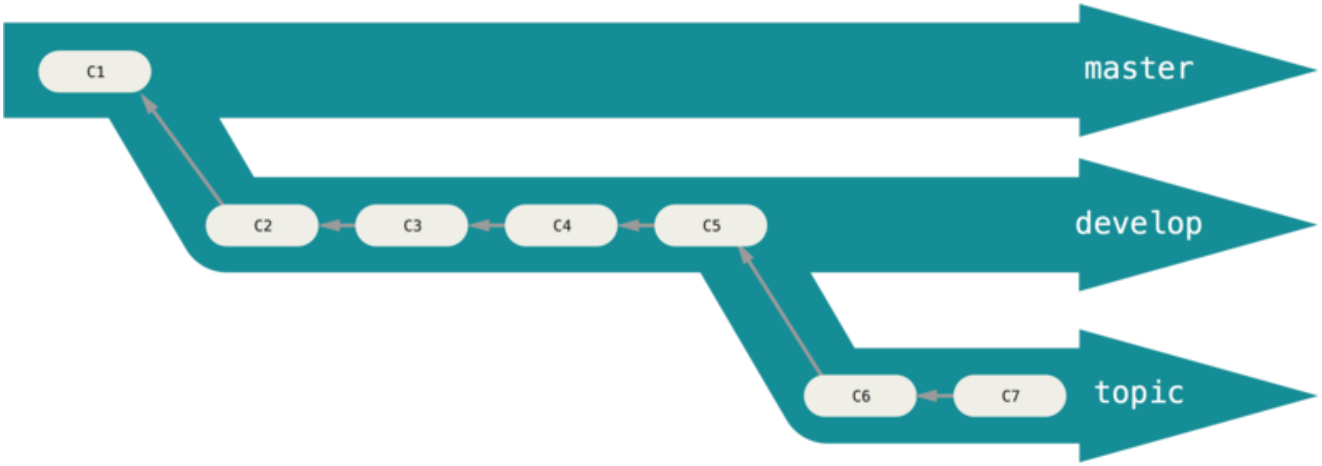

Figure 20. A "silo" view of progressive-stability branching

You can keep doing this for several levels of stability. Some larger projects also have a `proposed` or `pu` (proposed updates) branch that has integrated branches that may not be ready to go into the `next` or `master` branch. The idea is that your branches are at various levels of stability; when they reach a more stable level, they're merged into the branch above them. Again, having multiple long-running branches isn't necessary, but it's often helpful, especially when you're dealing with very large or complex projects.

Topic Branches

Topic branches, however, are useful in projects of any size. A topic branch is a short-lived branch that you create and use for a single particular feature or related work. This is something you've likely never done with a VCS before because it's generally too expensive to create and merge branches. But in Git it's common to create, work on, merge, and delete branches several times a day.

You saw this in the last section with the `iss53` and `hotfix` branches you created. You did a few

commits on them and deleted them directly after merging them into your main branch. This technique allows you to context-switch quickly and completely – because your work is separated into silos where all the changes in that branch have to do with that topic, it's easier to see what has happened during code review and such. You can keep the changes there for minutes, days, or months, and merge them in when they're ready, regardless of the order in which they were created or worked on.

Consider an example of doing some work (on `master`), branching off for an issue (`iss91`), working on it for a bit, branching off the second branch to try another way of handling the same thing (`iss91v2`), going back to your `master` branch and working there for a while, and then branching off there to do some work that you're not sure is a good idea (`dumbidea` branch). Your commit history will look something like this:

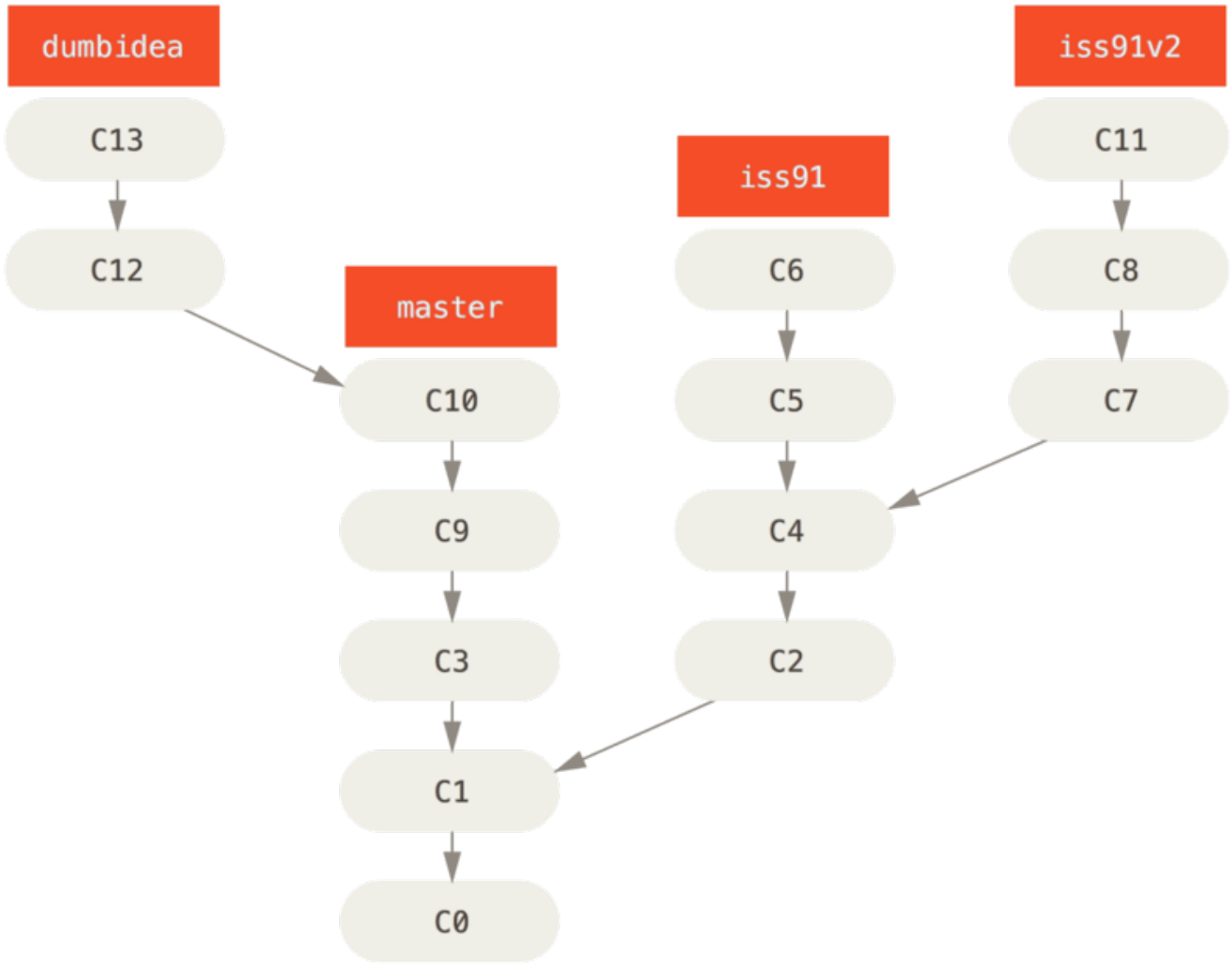

Figure 21. Multiple topic branches

Now, let's say you decide you like the second solution to your issue best (`iss91v2`); and you showed the `dumbidea` branch to your coworkers, and it turns out to be genius. You can throw away the original `iss91` branch (losing commits `C5` and `C6`) and merge in the other two. Your history then looks like this:

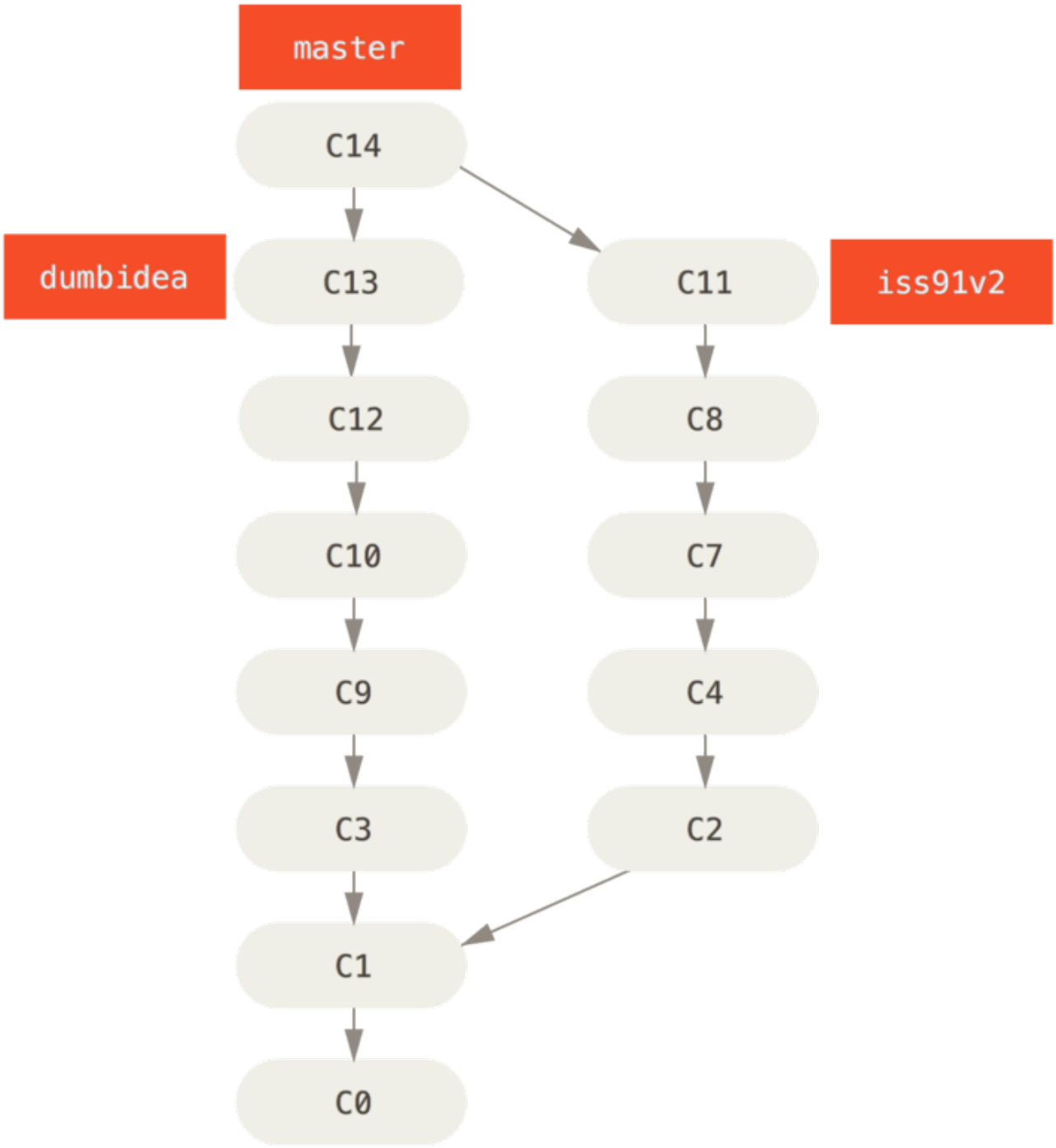

Figure 22. History after merging `dumbidea` *and* `iss91v2`

We will go into more detail about the various possible workflows for your Git project in [_distributed_git], so before you decide which branching scheme your next project will use, be sure to read that chapter.

It's important to remember when you're doing all this that these branches are completely local. When you're branching and merging, everything is being done only in your Git repository – no server communication is happening.

Remote Branches

Remote references are references (pointers) in your remote repositories, including branches, tags,

and so on. You can get a full list of remote references explicitly with `git ls-remote [remote]`, or `git remote show [remote]` for remote branches as well as more information. Nevertheless, a more common way is to take advantage of remote-tracking branches.

Remote-tracking branches are references to the state of remote branches. They're local references that you can't move; they're moved automatically for you whenever you do any network communication. Remote-tracking branches act as bookmarks to remind you where the branches in your remote repositories were the last time you connected to them.

They take the form `(remote)/(branch)`. For instance, if you wanted to see what the `master` branch on your `origin` remote looked like as of the last time you communicated with it, you would check the `origin/master` branch. If you were working on an issue with a partner and they pushed up an `iss53` branch, you might have your own local `iss53` branch; but the branch on the server would point to the commit at `origin/iss53`.

This may be a bit confusing, so let's look at an example. Let's say you have a Git server on your network at `git.ourcompany.com`. If you clone from this, Git's `clone` command automatically names it `origin` for you, pulls down all its data, creates a pointer to where its `master` branch is, and names it `origin/master` locally. Git also gives you your own local `master` branch starting at the same place as origin's `master` branch, so you have something to work from.

NOTE

"origin" is not special

Just like the branch name "master" does not have any special meaning in Git, neither does "origin". While "master" is the default name for a starting branch when you run `git init` which is the only reason it's widely used, "origin" is the default name for a remote when you run `git clone`. If you run `git clone -o booyah` instead, then you will have `booyah/master` as your default remote branch.

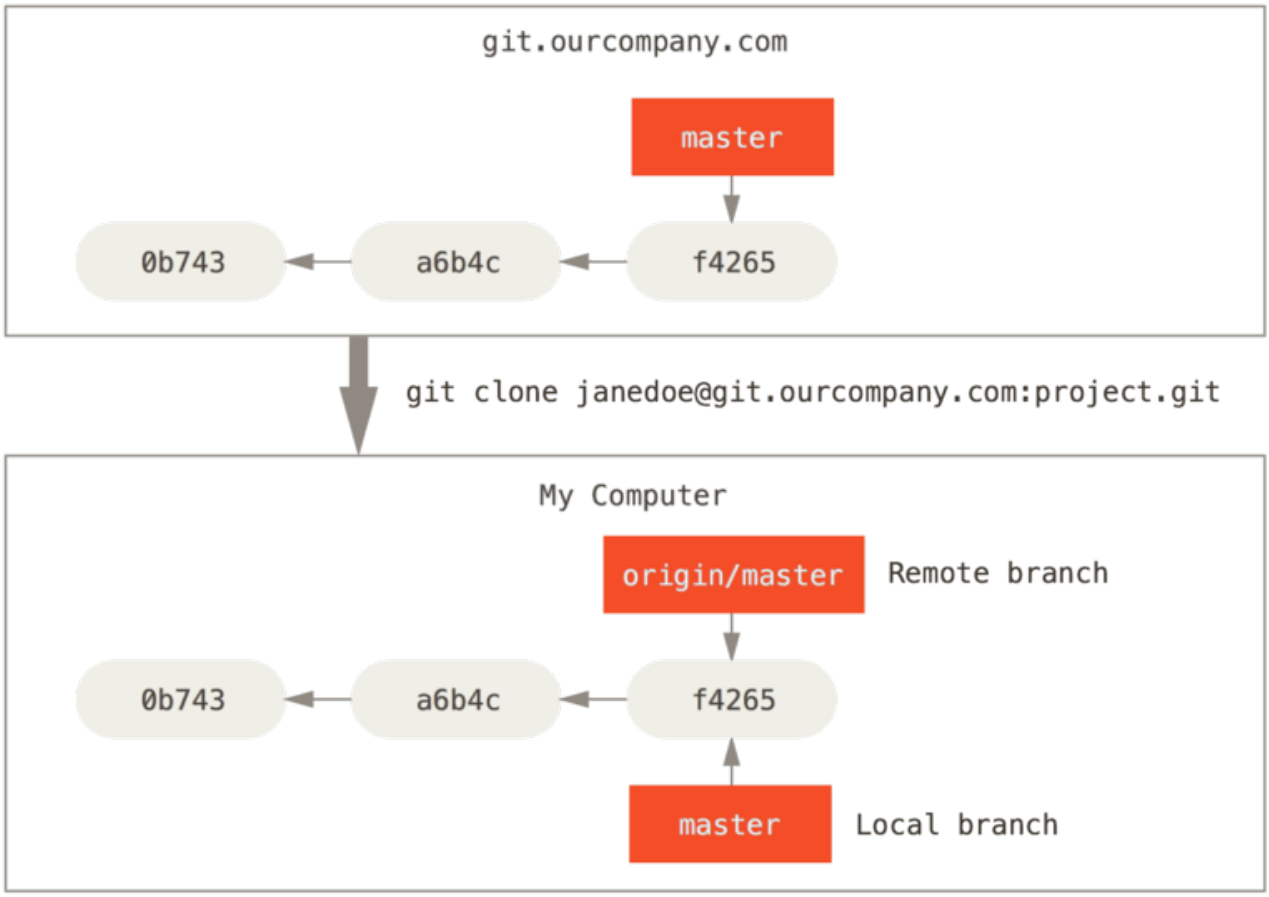

Figure 23. Server and local repositories after cloning

If you do some work on your local `master` branch, and, in the meantime, someone else pushes to `git.ourcompany.com` and updates its `master` branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your `origin/master` pointer doesn't move.

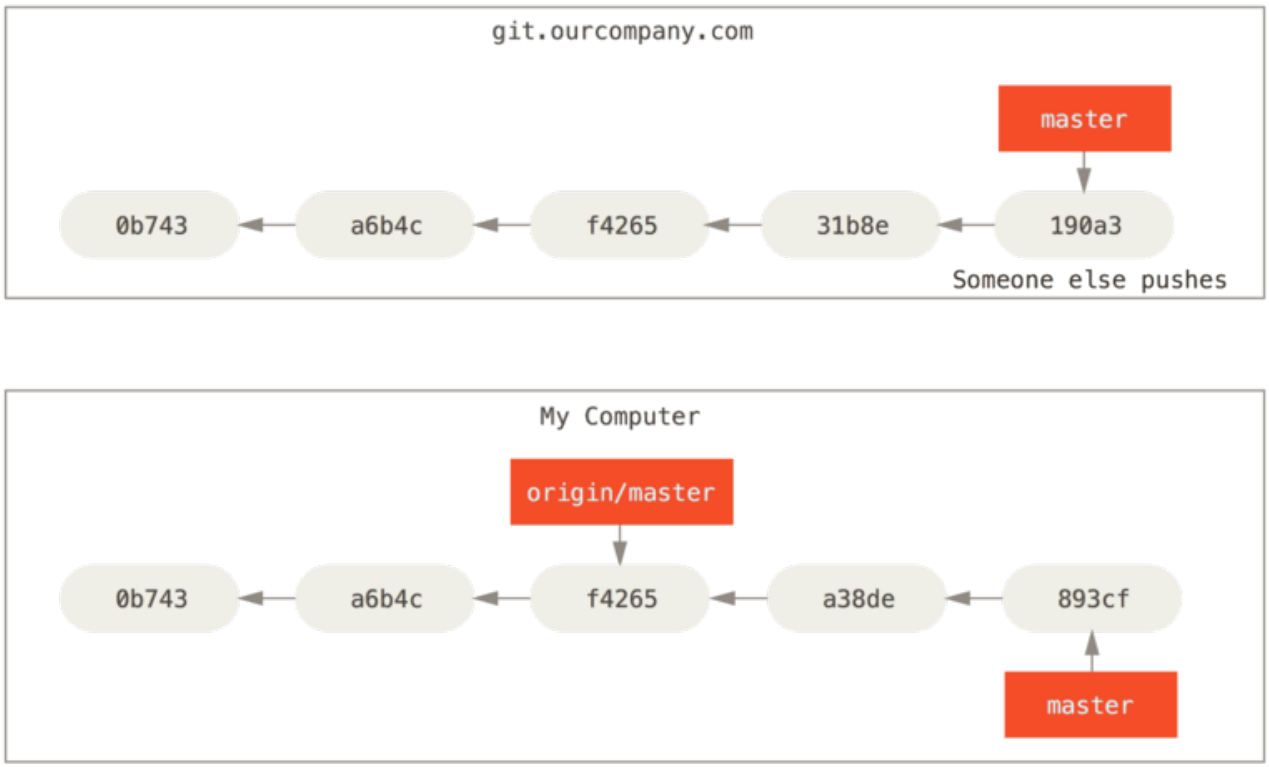

Figure 24. Local and remote work can diverge

To synchronize your work, you run a `git fetch origin` command. This command looks up which server "origin" is (in this case, it's `git.ourcompany.com`), fetches any data from it that you don't yet have, and updates your local database, moving your `origin/master` pointer to its new, more up-to-date position.

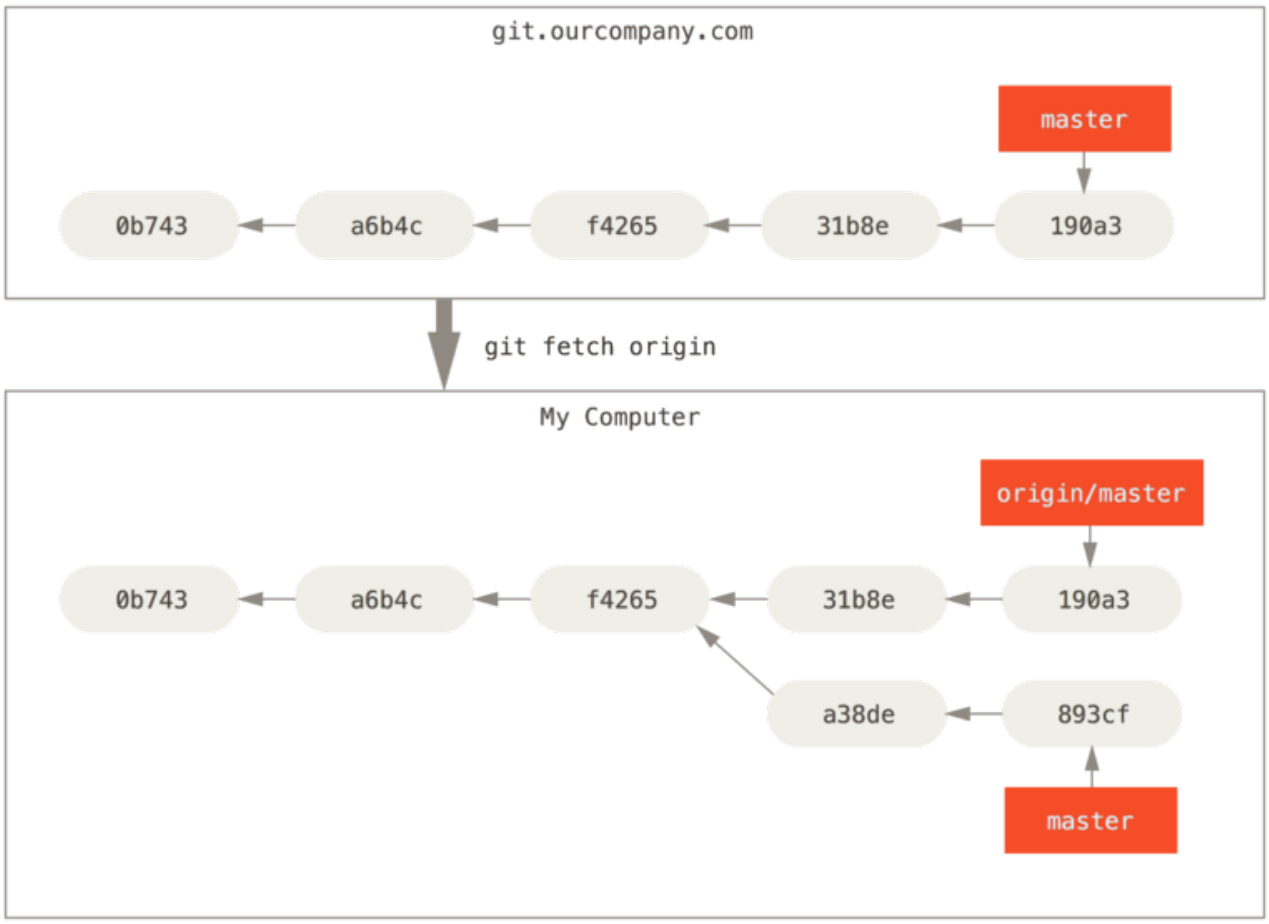

Figure 25. `git fetch` updates your remote references

To demonstrate having multiple remote servers and what remote branches for those remote projects look like, let's assume you have another internal Git server that is used only for development by one of your sprint teams. This server is at `git.team1.ourcompany.com`. You can add it as a new remote reference to the project you're currently working on by running the `git remote add` command as we covered in Git Basics. Name this remote `teamone`, which will be your shortname for that whole URL.

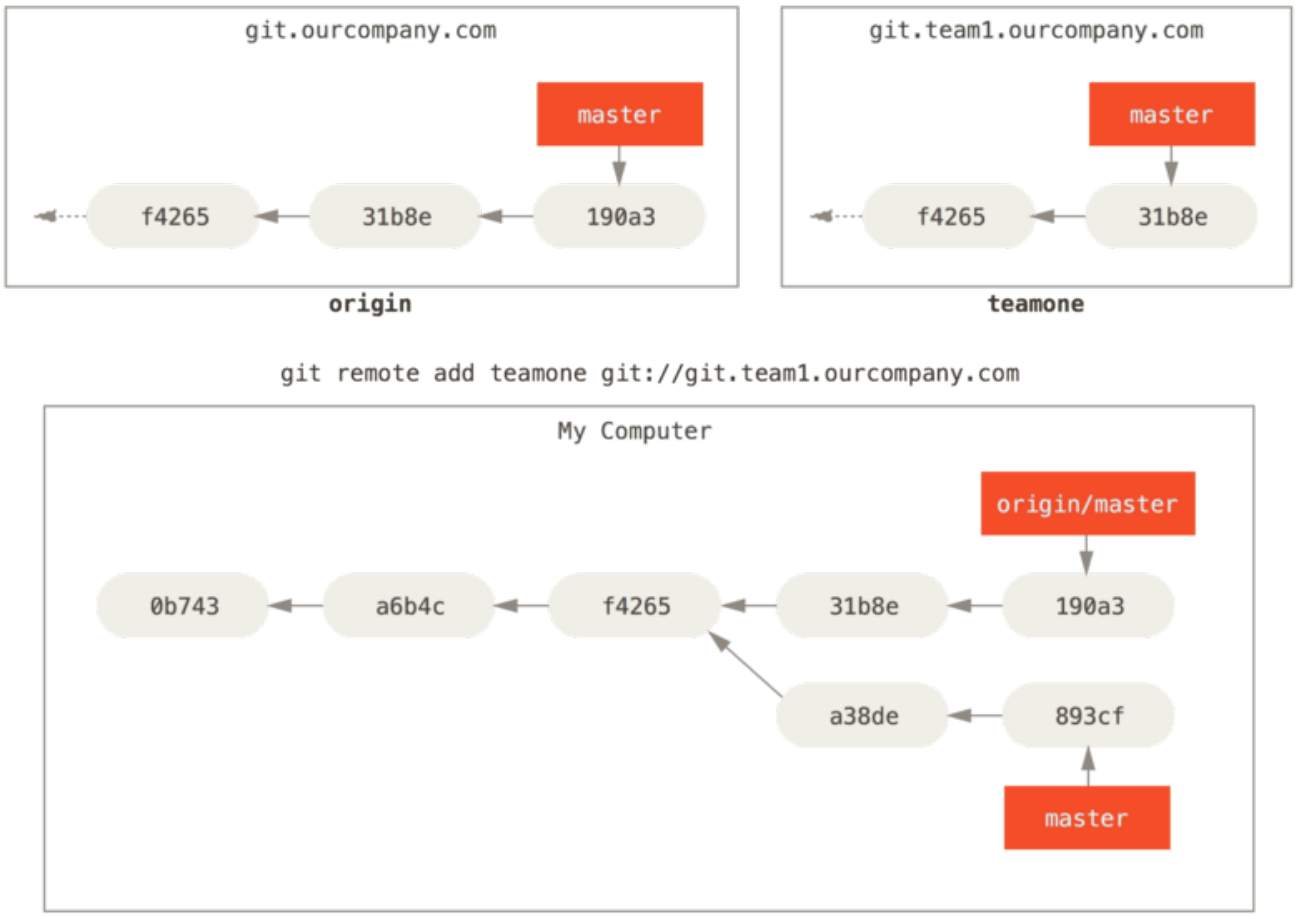

Figure 26. Adding another server as a remote

Now, you can run `git fetch teamone` to fetch everything the remote `teamone` server has that you don't have yet. Because that server has a subset of the data your `origin` server has right now, Git fetches no data but sets a remote-tracking branch called `teamone/master` to point to the commit that `teamone` has as its `master` branch.

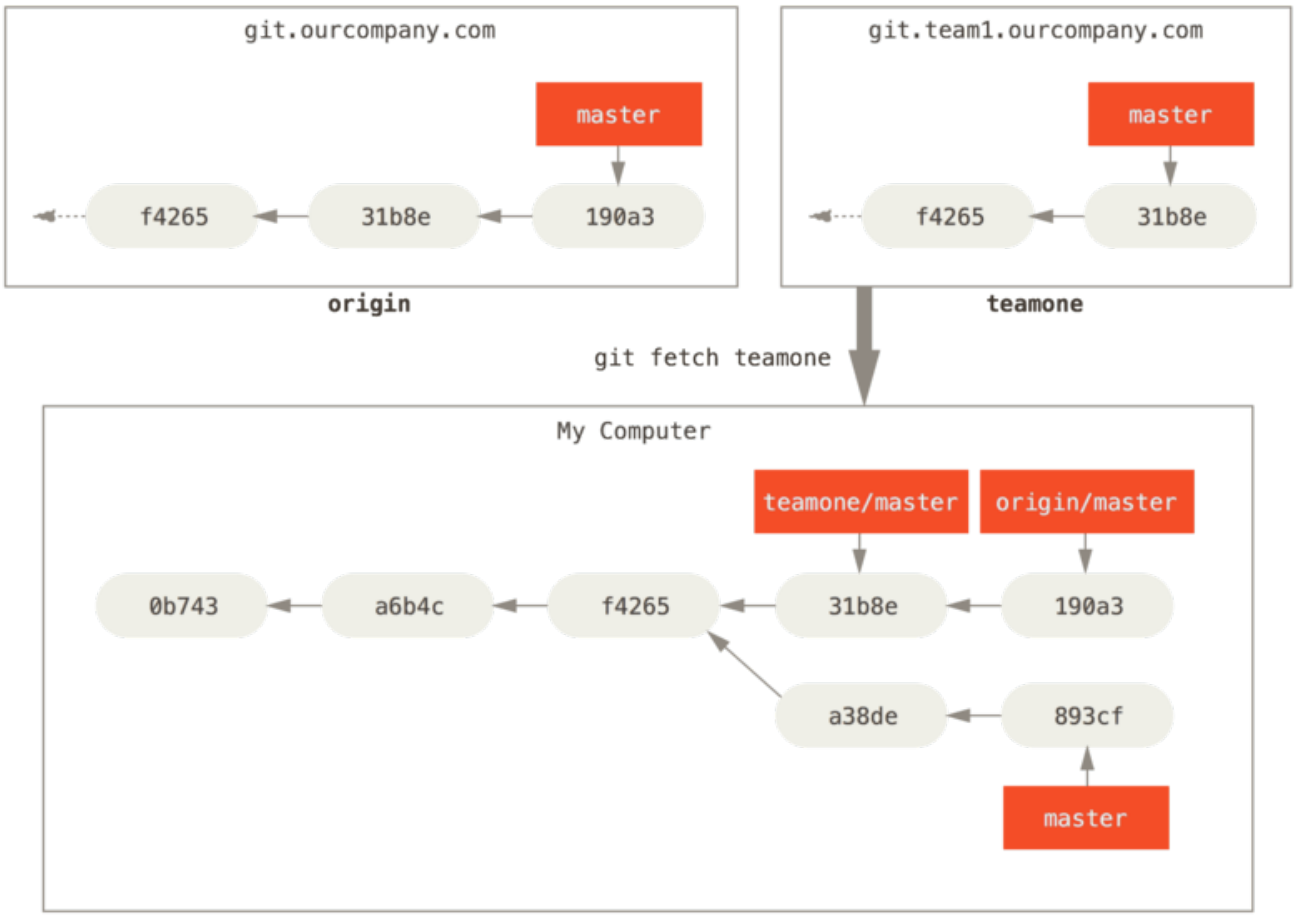

Figure 27. Remote tracking branch for `teamone/master`

Pushing

When you want to share a branch with the world, you need to push it up to a remote that you have write access to. Your local branches aren't automatically synchronized to the remotes you write to – you have to explicitly push the branches you want to share. That way, you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.

If you have a branch named `serverfix` that you want to work on with others, you can push it up the same way you pushed your first branch. Run `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
```

This is a bit of a shortcut. Git automatically expands the `serverfix` branchname out to `refs/heads/serverfix:refs/heads/serverfix`, which means, "Take my serverfix local branch and push it to update the remote's serverfix branch." We'll go over the `refs/heads/` part in detail in [_git_internals], but you can generally leave it off. You can also do `git push origin`

`serverfix:serverfix`, which does the same thing – it says, "Take my serverfix and make it the remote's serverfix." You can use this format to push a local branch into a remote branch that is named differently. If you didn't want it to be called `serverfix` on the remote, you could instead run `git push origin serverfix:awesomebranch` to push your local `serverfix` branch to the `awesomebranch` branch on the remote project.

NOTE

Don't type your password every time

If you're using an HTTPS URL to push over, the Git server will ask you for your username and password for authentication. By default it will prompt you on the terminal for this information so the server can tell if you're allowed to push.

If you don't want to type it every single time you push, you can set up a "credential cache". The simplest is just to keep it in memory for a few minutes, which you can easily set up by running `git config --global credential.helper cache`.

For more information on the various credential caching options available, see [_credential_caching].

The next time one of your collaborators fetches from the server, they will get a reference to where the server's version of `serverfix` is under the remote branch `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
```

It's important to note that when you do a fetch that brings down new remote-tracking branches, you don't automatically have local, editable copies of them. In other words, in this case, you don't have a new `serverfix` branch – you only have an `origin/serverfix` pointer that you can't modify.

To merge this work into your current working branch, you can run `git merge origin/serverfix`. If you want your own `serverfix` branch that you can work on, you can base it off your remote-tracking branch:

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

This gives you a local branch that you can work on that starts where `origin/serverfix` is.

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a "tracking branch" (and the branch it tracks is called an "upstream branch"). Tracking branches are

local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type `git pull`, Git automatically knows which server to fetch from and branch to merge into.

When you clone a repository, it generally automatically creates a `master` branch that tracks `origin/master`. However, you can set up other tracking branches if you wish – ones that track branches on other remotes, or don't track the `master` branch. The simple case is the example you just saw, running `git checkout -b [branch] [remotename]/[branch]`. This is a common enough operation that Git provides the `--track` shorthand:

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

In fact, this is so common that there's even a shortcut for that shortcut. If the branch name you're trying to checkout (a) doesn't exist and (b) exactly matches a name on only one remote, Git will create a tracking branch for you:

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

To set up a local branch with a different name than the remote branch, you can easily use the first version with a different local branch name:

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

Now, your local branch `sf` will automatically pull from `origin/serverfix`.

If you already have a local branch and want to set it to a remote branch you just pulled down, or want to change the upstream branch you're tracking, you can use the `-u` or `--set-upstream-to` option to `git branch` to explicitly set it at any time.

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

NOTE

Upstream shorthand

When you have a tracking branch set up, you can reference its upstream branch with the `@{upstream}` or `@{u}` shorthand. So if you're on the `master` branch and it's tracking `origin/master`, you can say something like `git merge @{u}` instead of `git merge origin/master` if you wish.

If you want to see what tracking branches you have set up, you can use the `-vv` option to `git branch`. This will list out your local branches with more information including what each branch is tracking

and if your local branch is ahead, behind or both.

```
$ 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
```

So here we can see that our `iss53` branch is tracking `origin/iss53` and is "ahead" by two, meaning that we have two commits locally that are not pushed to the server. We can also see that our `master` branch is tracking `origin/master` and is up to date. Next we can see that our `serverfix` branch is tracking the `server-fix-good` branch on our `teamone` server and is ahead by three and behind by one, meaning that there is one commit on the server we haven't merged in yet and three commits locally that we haven't pushed. Finally we can see that our `testing` branch is not tracking any remote branch.

It's important to note that these numbers are only since the last time you fetched from each server. This command does not reach out to the servers, it's telling you about what it has cached from these servers locally. If you want totally up to date ahead and behind numbers, you'll need to fetch from all your remotes right before running this. You could do that like this:

```
$ git fetch --all; git branch -vv
```

Pulling

While the `git fetch` command will fetch down all the changes on the server that you don't have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself. However, there is a command called `git pull` which is essentially a `git fetch` immediately followed by a `git merge` in most cases. If you have a tracking branch set up as demonstrated in the last section, either by explicitly setting it or by having it created for you by the `clone` or `checkout` commands, `git pull` will look up what server and branch your current branch is tracking, fetch from that server and then try to merge in that remote branch.

Generally it's better to simply use the `fetch` and `merge` commands explicitly as the magic of `git pull` can often be confusing.

Deleting Remote Branches

Suppose you're done with a remote branch – say you and your collaborators are finished with a feature and have merged it into your remote's `master` branch (or whatever branch your stable codeline is in). You can delete a remote branch using the `--delete` option to `git push`. If you want to delete your `serverfix` branch from the server, you run the following:

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

Basically all this does is remove the pointer from the server. The Git server will generally keep the data there for a while until a garbage collection runs, so if it was accidentally deleted, it's often easy to recover.

Rebasing

In Git, there are two main ways to integrate changes from one branch into another: the `merge` and the `rebase`. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it.

The Basic Rebase

If you go back to an earlier example from Basic Merging, you can see that you diverged your work and made commits on two different branches.

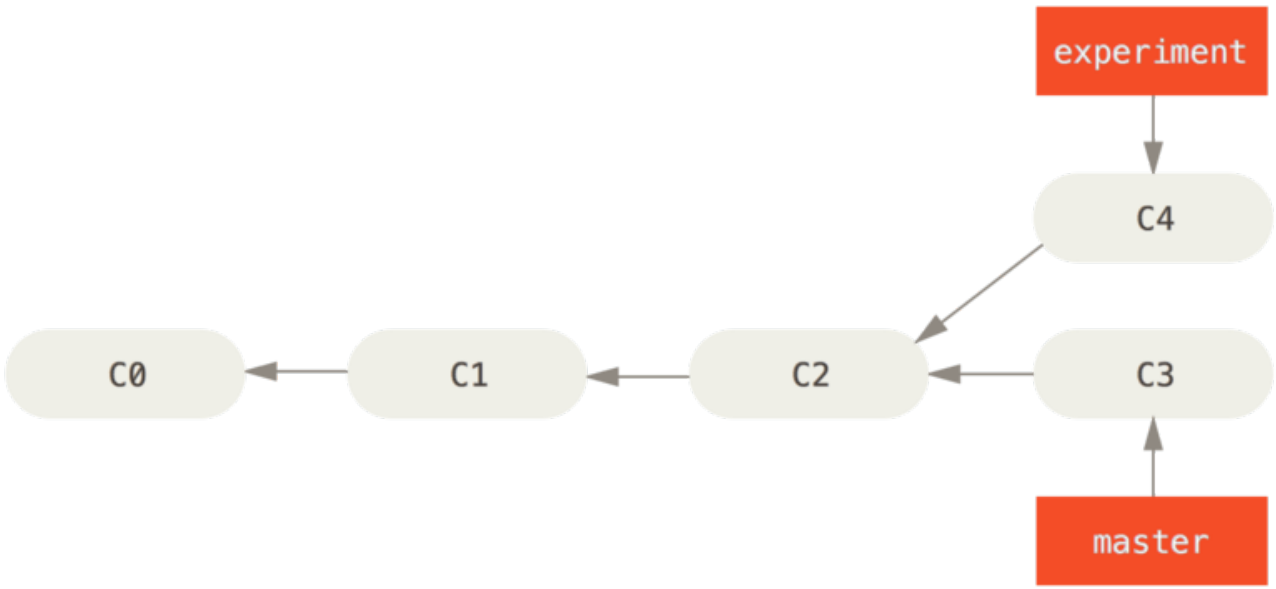

Figure 28. Simple divergent history

The easiest way to integrate the branches, as we've already covered, is the `merge` command. It performs a three-way merge between the two latest branch snapshots (`C3` and `C4`) and the most recent common ancestor of the two (`C2`), creating a new snapshot (and commit).

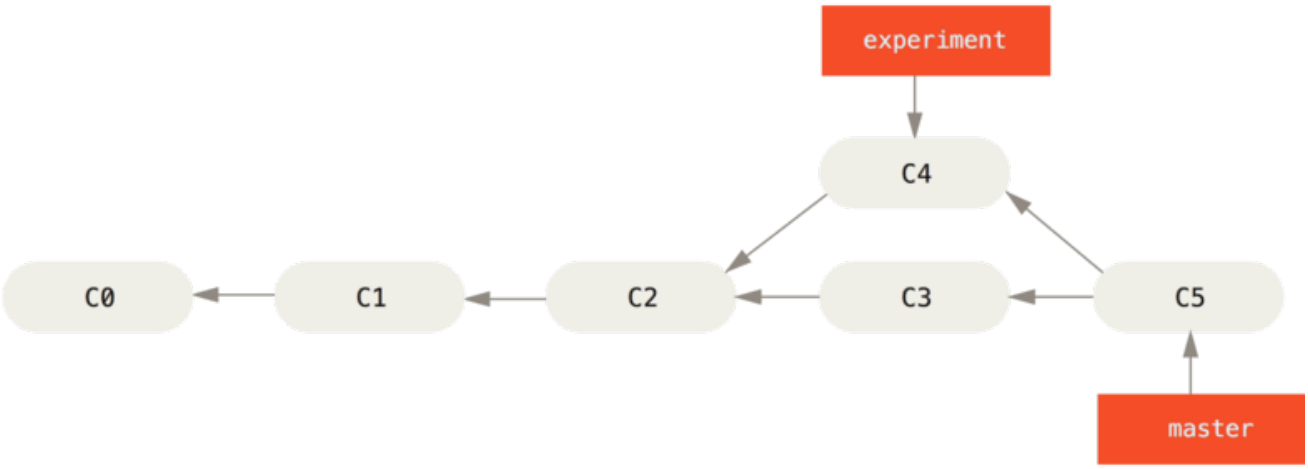

Figure 29. Merging to integrate diverged work history

However, there is another way: you can take the patch of the change that was introduced in `C4` and reapply it on top of `C3`. In Git, this is called *rebasing*. With the `rebase` command, you can take all the changes that were committed on one branch and replay them on another one.

In this example, you'd run the following:

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

It works by going to the common ancestor of the two branches (the one you're on and the one you're rebasing onto), getting the diff introduced by each commit of the branch you're on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

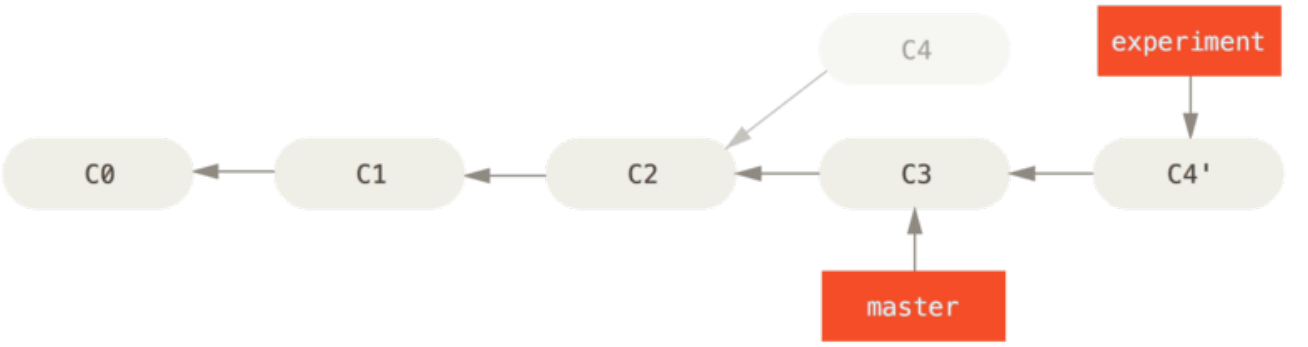

Figure 30. Rebasing the change introduced in `C4` onto `C3`

At this point, you can go back to the `master` branch and do a fast-forward merge.

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

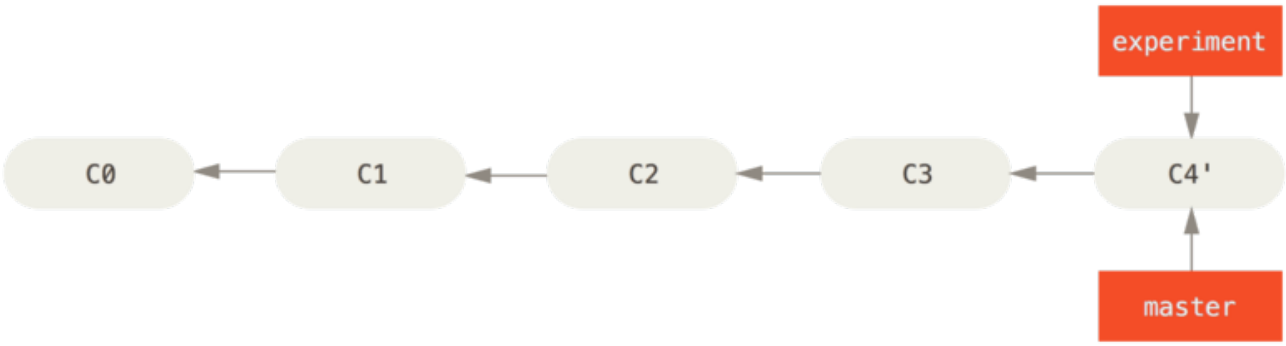

Figure 31. Fast-forwarding the master branch

Now, the snapshot pointed to by `C4'` is exactly the same as the one that was pointed to by `C5` in the merge example. There is no difference in the end product of the integration, but rebasing makes for a cleaner history. If you examine the log of a rebased branch, it looks like a linear history: it appears that all the work happened in series, even when it originally happened in parallel.

Often, you'll do this to make sure your commits apply cleanly on a remote branch – perhaps in a project to which you're trying to contribute but that you don't maintain. In this case, you'd do your work in a branch and then rebase your work onto `origin/master` when you were ready to submit your patches to the main project. That way, the maintainer doesn't have to do any integration work – just a fast-forward or a clean apply.

Note that the snapshot pointed to by the final commit you end up with, whether it's the last of the rebased commits for a rebase or the final merge commit after a merge, is the same snapshot – it's only the history that is different. Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

More Interesting Rebases

You can also have your rebase replay on something other than the rebase target branch. Take a history like A history with a topic branch off another topic branch, for example. You branched a topic branch (`server`) to add some server-side functionality to your project, and made a commit. Then, you branched off that to make the client-side changes (`client`) and committed a few times. Finally, you went back to your server branch and did a few more commits.

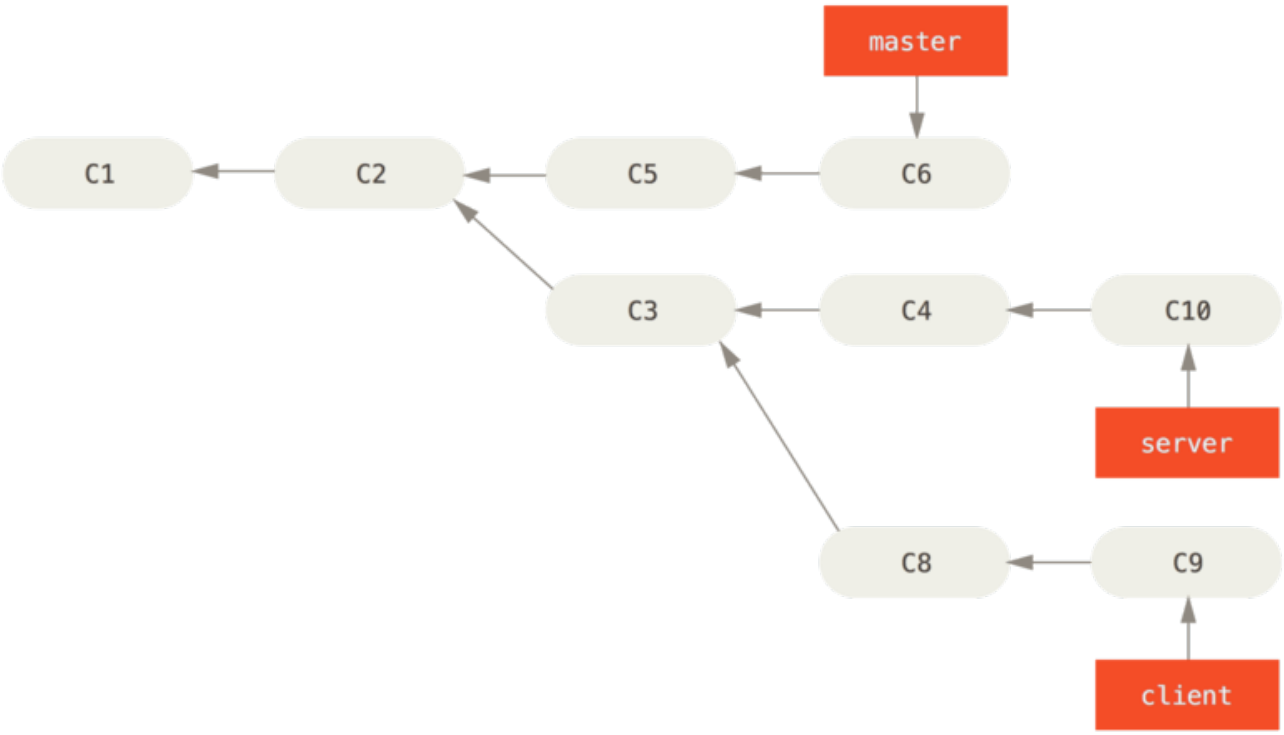

Figure 32. A history with a topic branch off another topic branch

Suppose you decide that you want to merge your client-side changes into your mainline for a release, but you want to hold off on the server-side changes until it's tested further. You can take the changes on client that aren't on server (`C8` and `C9`) and replay them on your `master` branch by using the `--onto` option of `git rebase`:

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

This basically says, "Take the `client` branch, figure out the patches since it diverged from the `server` branch, and replay these patches in the `client` branch as if it was based directly off the `master` branch instead." It's a bit complex, but the result is pretty cool.

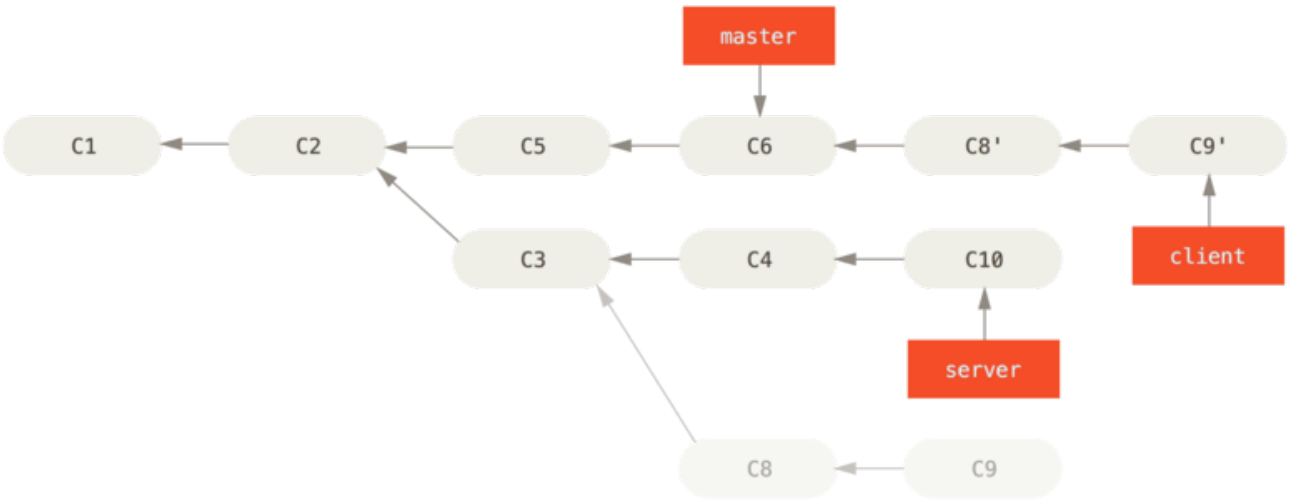

Figure 33. Rebasing a topic branch off another topic branch

Now you can fast-forward your `master` branch (see Fast-forwarding your master branch to include

the client branch changes):

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

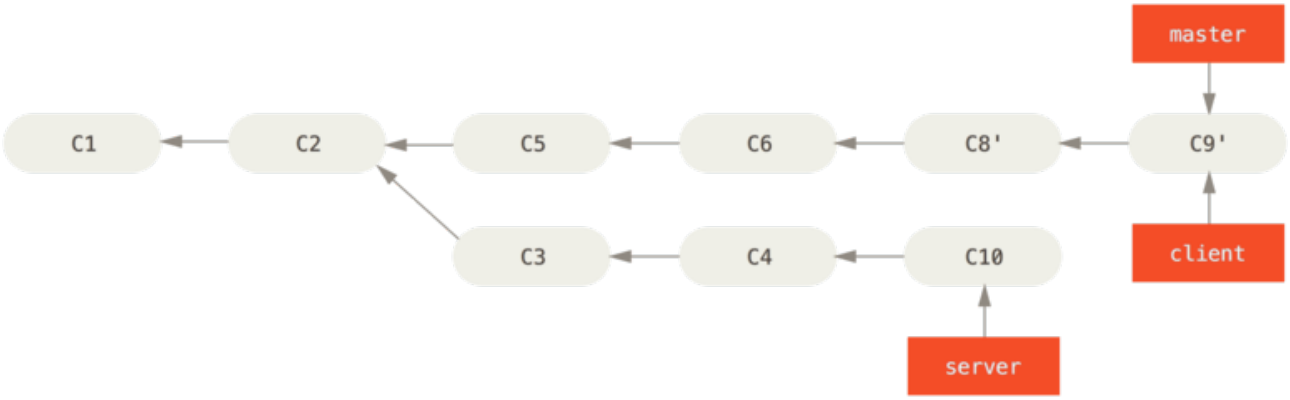

Figure 34. Fast-forwarding your master branch to include the client branch changes

Let's say you decide to pull in your server branch as well. You can rebase the server branch onto the `master` branch without having to check it out first by running `git rebase [basebranch] [topicbranch]` – which checks out the topic branch (in this case, `server`) for you and replays it onto the base branch (`master`):

```
$ git rebase master server
```

This replays your `server` work on top of your `master` work, as shown in Rebasing your server branch on top of your master branch.

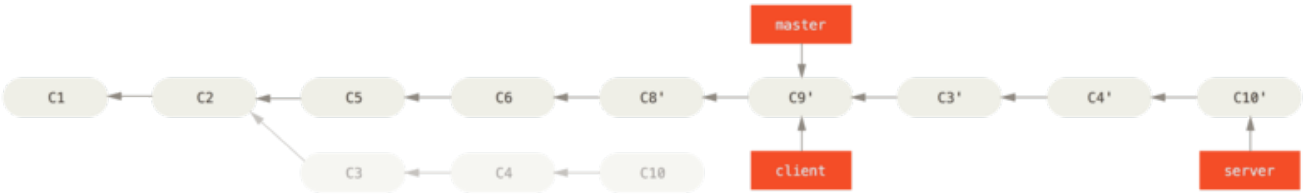

Figure 35. Rebasing your server branch on top of your master branch

Then, you can fast-forward the base branch (`master`):

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

You can remove the `client` and `server` branches because all the work is integrated and you don't need them anymore, leaving your history for this entire process looking like Final commit history:

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

Figure 36. Final commit history

The Perils of Rebasing

Ahh, but the bliss of rebasing isn't without its drawbacks, which can be summed up in a single line:

Do not rebase commits that exist outside your repository.

If you follow that guideline, you'll be fine. If you don't, people will hate you, and you'll be scorned by friends and family.

When you rebase stuff, you're abandoning existing commits and creating new ones that are similar but different. If you push commits somewhere and others pull them down and base work on them, and then you rewrite those commits with `git rebase` and push them up again, your collaborators will have to re-merge their work and things will get messy when you try to pull their work back into yours.

Let's look at an example of how rebasing work that you've made public can cause problems. Suppose you clone from a central server and then do some work off that. Your commit history looks like this:

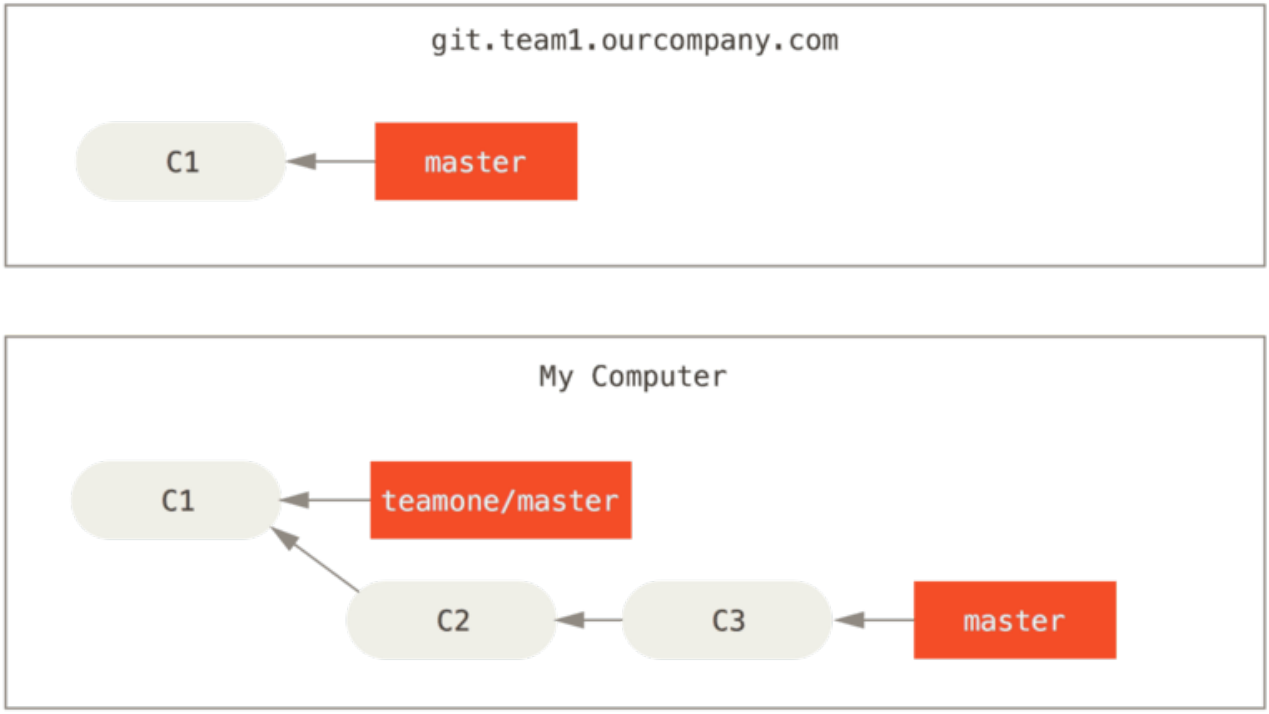

Figure 37. Clone a repository, and base some work on it

Now, someone else does more work that includes a merge, and pushes that work to the central server. You fetch it and merge the new remote branch into your work, making your history look something like this:

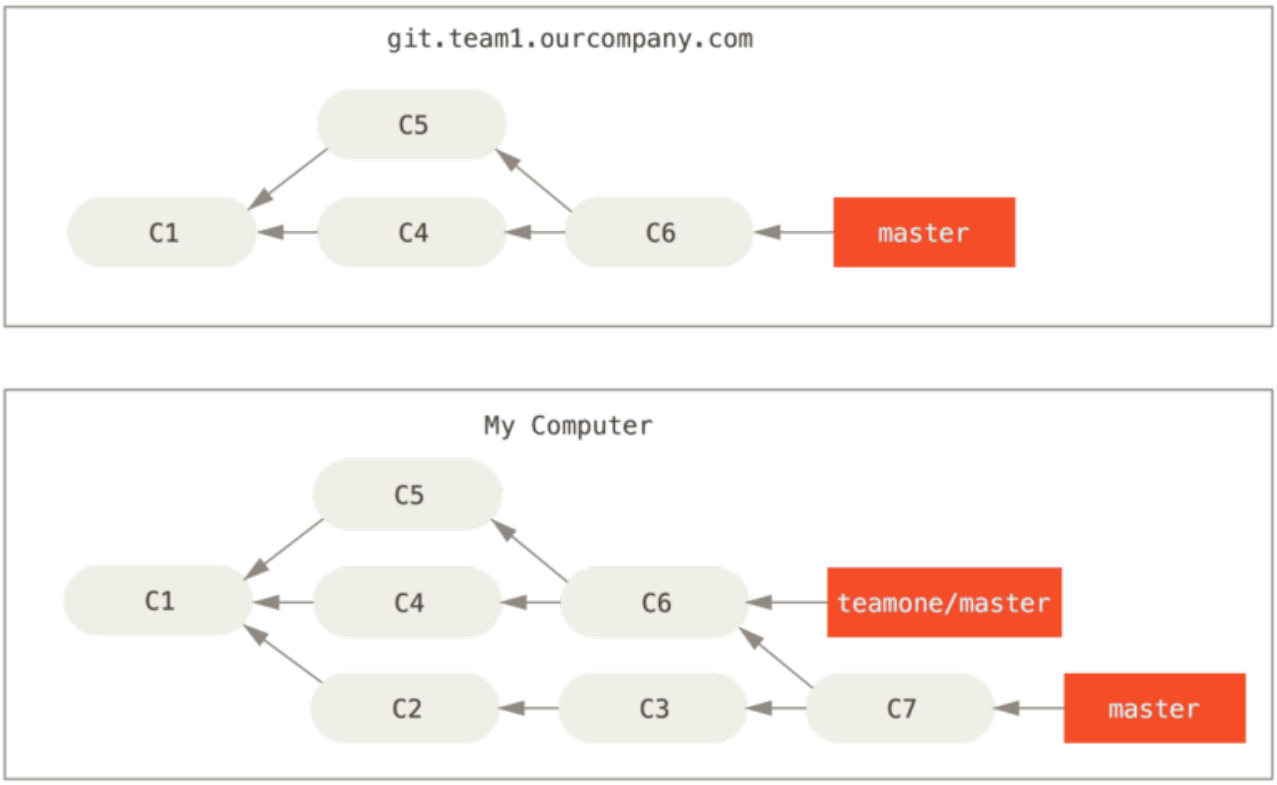

Figure 38. Fetch more commits, and merge them into your work

Next, the person who pushed the merged work decides to go back and rebase their work instead; they do a `git push --force` to overwrite the history on the server. You then fetch from that server, bringing down the new commits.

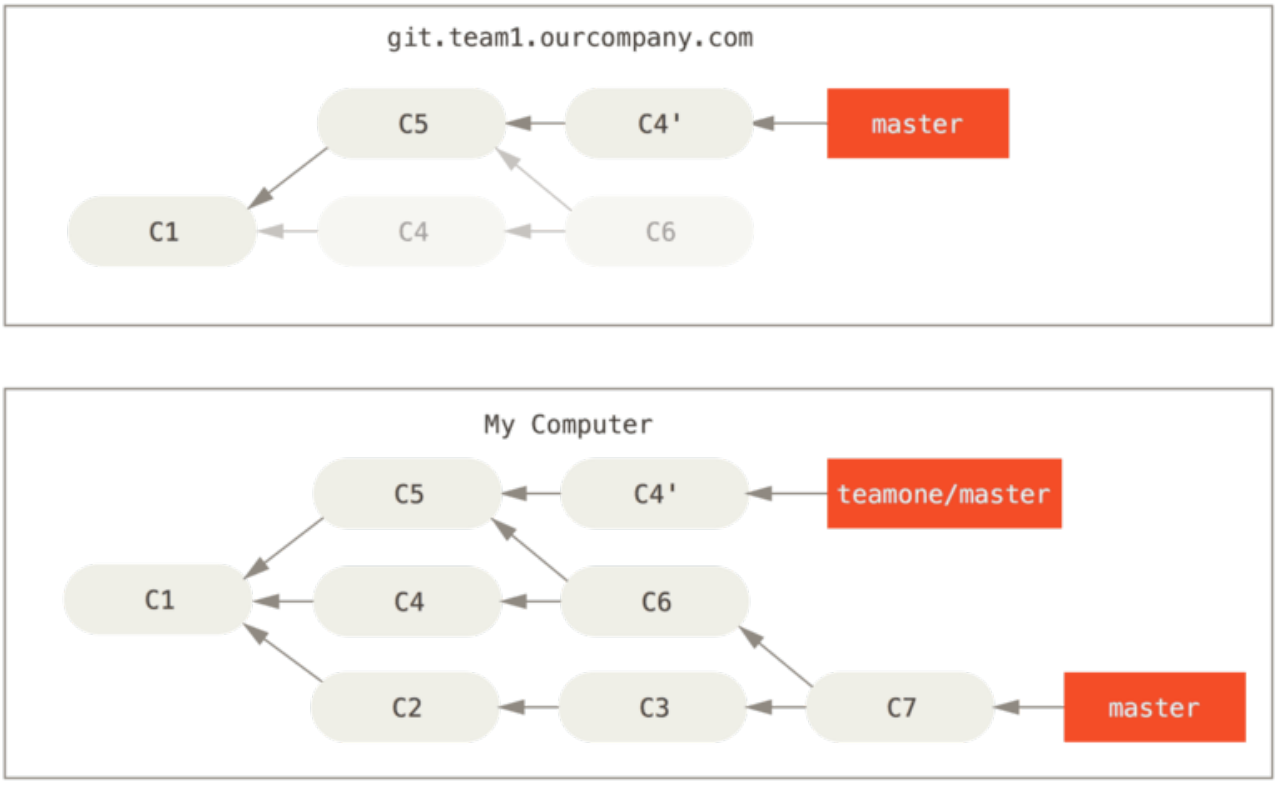

Figure 39. Someone pushes rebased commits, abandoning commits you've based your work on

Now you're both in a pickle. If you do a `git pull`, you'll create a merge commit which includes both lines of history, and your repository will look like this:

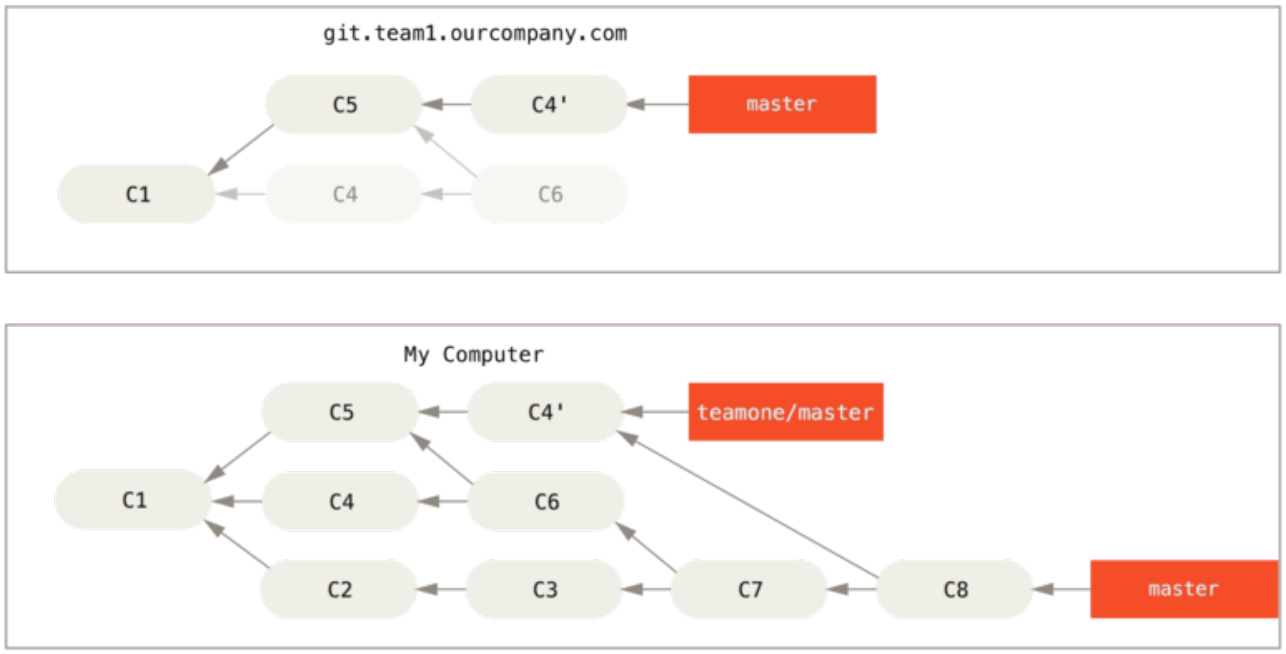

Figure 40. You merge in the same work again into a new merge commit

If you run a `git log` when your history looks like this, you'll see two commits that have the same author, date, and message, which will be confusing. Furthermore, if you push this history back up to the server, you'll reintroduce all those rebased commits to the central server, which can further confuse people. It's pretty safe to assume that the other developer doesn't want `C4` and `C6` to be in the history; that's why they rebased in the first place.

Rebase When You Rebase

If you **do** find yourself in a situation like this, Git has some further magic that might help you out. If someone on your team force pushes changes that overwrite work that you've based work on, your challenge is to figure out what is yours and what they've rewritten.

It turns out that in addition to the commit SHA-1 checksum, Git also calculates a checksum that is based just on the patch introduced with the commit. This is called a "patch-id".

If you pull down work that was rewritten and rebase it on top of the new commits from your partner, Git can often successfully figure out what is uniquely yours and apply them back on top of the new branch.

For instance, in the previous scenario, if instead of doing a merge when we're at Someone pushes rebased commits, abandoning commits you've based your work on we run `git rebase teamone/master`, Git will:

- Determine what work is unique to our branch (C2, C3, C4, C6, C7)
- Determine which are not merge commits (C2, C3, C4)
- Determine which have not been rewritten into the target branch (just C2 and C3, since C4 is the same patch as C4')

- Apply those commits to the top of `teamone/master`

So instead of the result we see in You merge in the same work again into a new merge commit, we would end up with something more like Rebase on top of force-pushed rebase work..

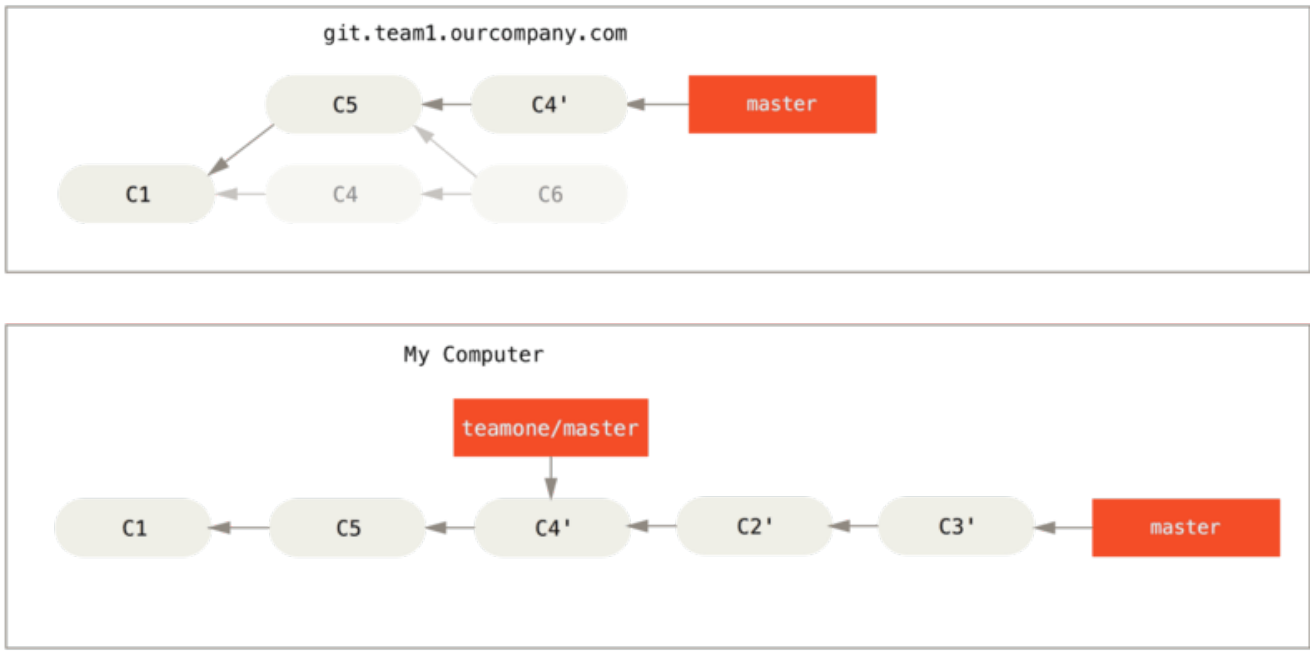

Figure 41. Rebase on top of force-pushed rebase work.

This only works if C4 and C4' that your partner made are almost exactly the same patch. Otherwise the rebase won't be able to tell that it's a duplicate and will add another C4-like patch (which will probably fail to apply cleanly, since the changes would already be at least somewhat there).

You can also simplify this by running a `git pull --rebase` instead of a normal `git pull`. Or you could do it manually with a `git fetch` followed by a `git rebase teamone/master` in this case.

If you are using `git pull` and want to make `--rebase` the default, you can set the `pull.rebase` config value with something like `git config --global pull.rebase true`.

If you treat rebasing as a way to clean up and work with commits before you push them, and if you only rebase commits that have never been available publicly, then you'll be fine. If you rebase commits that have already been pushed publicly, and people may have based work on those commits, then you may be in for some frustrating trouble, and the scorn of your teammates.

If you or a partner does find it necessary at some point, make sure everyone knows to run `git pull --rebase` to try to make the pain after it happens a little bit simpler.

Rebase vs. Merge

Now that you've seen rebasing and merging in action, you may be wondering which one is better. Before we can answer this, let's step back a bit and talk about what history means.

One point of view on this is that your repository's commit history is a **record of what actually happened.** It's a historical document, valuable in its own right, and shouldn't be tampered with. From this angle, changing the commit history is almost blasphemous; you're *lying* about what

actually transpired. So what if there was a messy series of merge commits? That's how it happened, and the repository should preserve that for posterity.

The opposing point of view is that the commit history is the **story of how your project was made.** You wouldn't publish the first draft of a book, and the manual for how to maintain your software deserves careful editing. This is the camp that uses tools like rebase and filter-branch to tell the story in the way that's best for future readers.

Now, to the question of whether merging or rebasing is better: hopefully you'll see that it's not that simple. Git is a powerful tool, and allows you to do many things to and with your history, but every team and every project is different. Now that you know how both of these things work, it's up to you to decide which one is best for your particular situation.

In general the way to get the best of both worlds is to rebase local changes you've made but haven't shared yet before you push them in order to clean up your story, but never rebase anything you've pushed somewhere.

Summary

We've covered basic branching and merging in Git. You should feel comfortable creating and switching to new branches, switching between branches and merging local branches together. You should also be able to share your branches by pushing them to a shared server, working with others on shared branches and rebasing your branches before they are shared. Next, we'll cover what you'll need to run your own Git repository-hosting server.

Git on the Server

At this point, you should be able to do most of the day-to-day tasks for which you'll be using Git. However, in order to do any collaboration in Git, you'll need to have a remote Git repository. Although you can technically push changes to and pull changes from individuals' repositories, doing so is discouraged because you can fairly easily confuse what they're working on if you're not careful. Furthermore, you want your collaborators to be able to access the repository even if your computer is offline – having a more reliable common repository is often useful. Therefore, the preferred method for collaborating with someone is to set up an intermediate repository that you both have access to, and push to and pull from that.

Running a Git server is fairly straightforward. First, you choose which protocols you want your server to communicate with. The first section of this chapter will cover the available protocols and the pros and cons of each. The next sections will explain some typical setups using those protocols and how to get your server running with them. Last, we'll go over a few hosted options, if you don't mind hosting your code on someone else's server and don't want to go through the hassle of setting up and maintaining your own server.

If you have no interest in running your own server, you can skip to the last section of the chapter to see some options for setting up a hosted account and then move on to the next chapter, where we discuss the various ins and outs of working in a distributed source control environment.

A remote repository is generally a *bare repository* – a Git repository that has no working directory. Because the repository is only used as a collaboration point, there is no reason to have a snapshot checked out on disk; it's just the Git data. In the simplest terms, a bare repository is the contents of your project's `.git` directory and nothing else.

The Protocols

Git can use four major protocols to transfer data: Local, HTTP, Secure Shell (SSH) and Git. Here we'll discuss what they are and in what basic circumstances you would want (or not want) to use them.

Local Protocol

The most basic is the *Local protocol,* in which the remote repository is in another directory on disk. This is often used if everyone on your team has access to a shared filesystem such as an NFS mount, or in the less likely case that everyone logs in to the same computer. The latter wouldn't be ideal, because all your code repository instances would reside on the same computer, making a catastrophic loss much more likely.

If you have a shared mounted filesystem, then you can clone, push to, and pull from a local file-based repository. To clone a repository like this or to add one as a remote to an existing project, use the path to the repository as the URL. For example, to clone a local repository, you can run something like this:

```
$ git clone /srv/git/project.git
```

Or you can do this:

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

Git operates slightly differently if you explicitly specify `file://` at the beginning of the URL. If you just specify the path, Git tries to use hardlinks or directly copy the files it needs. If you specify `file://`, Git fires up the processes that it normally uses to transfer data over a network which is generally a lot less efficient method of transferring the data. The main reason to specify the `file://` prefix is if you want a clean copy of the repository with extraneous references or objects left out – generally after an import from another version-control system or something similar (see [_git_internals] for maintenance tasks). We'll use the normal path here because doing so is almost always faster.

To add a local repository to an existing Git project, you can run something like this:

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

Then, you can push to and pull from that remote as though you were doing so over a network.

The Pros

The pros of file-based repositories are that they're simple and they use existing file permissions and network access. If you already have a shared filesystem to which your whole team has access, setting up a repository is very easy. You stick the bare repository copy somewhere everyone has shared access to and set the read/write permissions as you would for any other shared directory. We'll discuss how to export a bare repository copy for this purpose in Git on the Server.

This is also a nice option for quickly grabbing work from someone else's working repository. If you and a co-worker are working on the same project and they want you to check something out, running a command like `git pull /home/john/project` is often easier than them pushing to a remote server and you pulling down.

The Cons

The cons of this method are that shared access is generally more difficult to set up and reach from multiple locations than basic network access. If you want to push from your laptop when you're at home, you have to mount the remote disk, which can be difficult and slow compared to network-based access.

It's important to mention that this isn't necessarily the fastest option if you're using a shared mount of some kind. A local repository is fast only if you have fast access to the data. A repository on NFS is often slower than the repository over SSH on the same server, allowing Git to run off local disks on each system.

Finally, this protocol does not protect the repository against accidental damage. Every user has full shell access to the "remote" directory, and there is nothing preventing them from changing or removing internal Git files and corrupting the repository.

The HTTP Protocols

Git can communicate over HTTP in two different modes. Prior to Git 1.6.6 there was only one way it could do this which was very simple and generally read-only. In version 1.6.6 a new, smarter protocol was introduced that involved Git being able to intelligently negotiate data transfer in a manner similar to how it does over SSH. In the last few years, this new HTTP protocol has become very popular since it's simpler for the user and smarter about how it communicates. The newer version is often referred to as the "Smart" HTTP protocol and the older way as "Dumb" HTTP. We'll cover the newer "smart" HTTP protocol first.

Smart HTTP

The "smart" HTTP protocol operates very similarly to the SSH or Git protocols but runs over standard HTTP/S ports and can use various HTTP authentication mechanisms, meaning it's often easier on the user than something like SSH, since you can use things like username/password authentication rather than having to set up SSH keys.

It has probably become the most popular way to use Git now, since it can be set up to both serve anonymously like the `git://` protocol, and can also be pushed over with authentication and encryption like the SSH protocol. Instead of having to set up different URLs for these things, you can now use a single URL for both. If you try to push and the repository requires authentication (which it normally should), the server can prompt for a username and password. The same goes for read access.

In fact, for services like GitHub, the URL you use to view the repository online (for example, "https://github.com/schacon/simplegit[]") is the same URL you can use to clone and, if you have access, push over.

Dumb HTTP

If the server does not respond with a Git HTTP smart service, the Git client will try to fall back to the simpler "dumb" HTTP protocol. The Dumb protocol expects the bare Git repository to be served like normal files from the web server. The beauty of the Dumb HTTP protocol is the simplicity of setting it up. Basically, all you have to do is put a bare Git repository under your HTTP document root and set up a specific `post-update` hook, and you're done (See [_git_hooks]). At that point, anyone who can access the web server under which you put the repository can also clone your repository. To allow read access to your repository over HTTP, do something like this:

```
$ 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
```

That's all. The `post-update` hook that comes with Git by default runs the appropriate command (`git update-server-info`) to make HTTP fetching and cloning work properly. This command is run when you push to this repository (over SSH perhaps); then, other people can clone via something like

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

In this particular case, we're using the `/var/www/htdocs` path that is common for Apache setups, but you can use any static web server – just put the bare repository in its path. The Git data is served as basic static files (see [_git_internals] for details about exactly how it's served).

Generally you would either choose to run a read/write Smart HTTP server or simply have the files accessible as read-only in the Dumb manner. It's rare to run a mix of the two services.

The Pros

We'll concentrate on the pros of the Smart version of the HTTP protocol.

The simplicity of having a single URL for all types of access and having the server prompt only when authentication is needed makes things very easy for the end user. Being able to authenticate with a username and password is also a big advantage over SSH, since users don't have to generate SSH keys locally and upload their public key to the server before being able to interact with it. For less sophisticated users, or users on systems where SSH is less common, this is a major advantage in usability. It is also a very fast and efficient protocol, similar to the SSH one.

You can also serve your repositories read-only over HTTPS, which means you can encrypt the content transfer; or you can go so far as to make the clients use specific signed SSL certificates.

Another nice thing is that HTTP/S are such commonly used protocols that corporate firewalls are often set up to allow traffic through these ports.

The Cons

Git over HTTP/S can be a little more tricky to set up compared to SSH on some servers. Other than that, there is very little advantage that other protocols have over the "Smart" HTTP protocol for serving Git.

If you're using HTTP for authenticated pushing, providing your credentials is sometimes more complicated than using keys over SSH. There are however several credential caching tools you can use, including Keychain access on OSX and Credential Manager on Windows, to make this pretty painless. Read [_credential_caching] to see how to set up secure HTTP password caching on your system.

The SSH Protocol

A common transport protocol for Git when self-hosting is over SSH. This is because SSH access to servers is already set up in most places – and if it isn't, it's easy to do. SSH is also an authenticated network protocol; and because it's ubiquitous, it's generally easy to set up and use.

To clone a Git repository over SSH, you can specify ssh:// URL like this:

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

Or you can use the shorter scp-like syntax for the SSH protocol:

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

You can also not specify a user, and Git assumes the user you're currently logged in as.

The Pros

The pros of using SSH are many. First, SSH is relatively easy to set up – SSH daemons are commonplace, many network admins have experience with them, and many OS distributions are set up with them or have tools to manage them. Next, access over SSH is secure – all data transfer is encrypted and authenticated. Last, like the HTTP/S, Git and Local protocols, SSH is efficient, making the data as compact as possible before transferring it.

The Cons

The negative aspect of SSH is that you can't serve anonymous access of your repository over it. People must have access to your machine over SSH to access it, even in a read-only capacity, which doesn't make SSH access conducive to open source projects. If you're using it only within your corporate network, SSH may be the only protocol you need to deal with. If you want to allow anonymous read-only access to your projects and also want to use SSH, you'll have to set up SSH for you to push over but something else for others to fetch over.

The Git Protocol

Next is the Git protocol. This is a special daemon that comes packaged with Git; it listens on a dedicated port (9418) that provides a service similar to the SSH protocol, but with absolutely no authentication. In order for a repository to be served over the Git protocol, you must create the `git-daemon-export-ok` file – the daemon won't serve a repository without that file in it – but other than that there is no security. Either the Git repository is available for everyone to clone or it isn't. This means that there is generally no pushing over this protocol. You can enable push access; but given the lack of authentication, if you turn on push access, anyone on the internet who finds your project's URL could push to your project. Suffice it to say that this is rare.

The Pros

The Git protocol is often the fastest network transfer protocol available. If you're serving a lot of traffic for a public project or serving a very large project that doesn't require user authentication for read access, it's likely that you'll want to set up a Git daemon to serve your project. It uses the same data-transfer mechanism as the SSH protocol but without the encryption and authentication overhead.

The Cons

The downside of the Git protocol is the lack of authentication. It's generally undesirable for the Git protocol to be the only access to your project. Generally, you'll pair it with SSH or HTTPS access for the few developers who have push (write) access and have everyone else use `git://` for read-only access. It's also probably the most difficult protocol to set up. It must run its own daemon, which

requires `xinetd` configuration or the like, which isn't always a walk in the park. It also requires firewall access to port 9418, which isn't a standard port that corporate firewalls always allow. Behind big corporate firewalls, this obscure port is commonly blocked.

Getting Git on a Server

Now we'll cover setting up a Git service running these protocols on your own server.

NOTE	Here we'll be demonstrating the commands and steps needed to do basic, simplified installations on a Linux based server, though it's also possible to run these services on Mac or Windows servers. Actually setting up a production server within your infrastructure will certainly entail differences in security measures or operating system tools, but hopefully this will give you the general idea of what's involved.

In order to initially set up any Git server, you have to export an existing repository into a new bare repository – a repository that doesn't contain a working directory. This is generally straightforward to do. In order to clone your repository to create a new bare repository, you run the clone command with the `--bare` option. By convention, bare repository directories end in `.git`, like so:

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

You should now have a copy of the Git directory data in your `my_project.git` directory.

This is roughly equivalent to something like

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

There are a couple of minor differences in the configuration file; but for your purpose, this is close to the same thing. It takes the Git repository by itself, without a working directory, and creates a directory specifically for it alone.

Putting the Bare Repository on a Server

Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols. Let's say you've set up a server called `git.example.com` that you have SSH access to, and you want to store all your Git repositories under the `/srv/git` directory. Assuming that `/srv/git` exists on that server, you can set up your new repository by copying your bare repository over:

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

At this point, other users who have SSH access to the same server which has read-access to the `/srv/git` directory can clone your repository by running

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

If a user SSHs into a server and has write access to the `/srv/git/my_project.git` directory, they will also automatically have push access.

Git will automatically add group write permissions to a repository properly if you run the `git init` command with the `--shared` option.

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

You see how easy it is to take a Git repository, create a bare version, and place it on a server to which you and your collaborators have SSH access. Now you're ready to collaborate on the same project.

It's important to note that this is literally all you need to do to run a useful Git server to which several people have access – just add SSH-able accounts on a server, and stick a bare repository somewhere that all those users have read and write access to. You're ready to go – nothing else needed.

In the next few sections, you'll see how to expand to more sophisticated setups. This discussion will include not having to create user accounts for each user, adding public read access to repositories, setting up web UIs and more. However, keep in mind that to collaborate with a couple of people on a private project, all you *need* is an SSH server and a bare repository.

Small Setups

If you're a small outfit or are just trying out Git in your organization and have only a few developers, things can be simple for you. One of the most complicated aspects of setting up a Git server is user management. If you want some repositories to be read-only to certain users and read/write to others, access and permissions can be a bit more difficult to arrange.

SSH Access

If you have a server to which all your developers already have SSH access, it's generally easiest to set up your first repository there, because you have to do almost no work (as we covered in the last section). If you want more complex access control type permissions on your repositories, you can handle them with the normal filesystem permissions of the operating system your server runs.

If you want to place your repositories on a server that doesn't have accounts for everyone on your team whom you want to have write access, then you must set up SSH access for them. We assume that if you have a server with which to do this, you already have an SSH server installed, and that's how you're accessing the server.

There are a few ways you can give access to everyone on your team. The first is to set up accounts for everybody, which is straightforward but can be cumbersome. You may not want to run `adduser`

and set temporary passwords for every user.

A second method is to create a single *git* user on the machine, ask every user who is to have write access to send you an SSH public key, and add that key to the `~/.ssh/authorized_keys` file of your new *git* user. At that point, everyone will be able to access that machine via the *git* user. This doesn't affect the commit data in any way – the SSH user you connect as doesn't affect the commits you've recorded.

Another way to do it is to have your SSH server authenticate from an LDAP server or some other centralized authentication source that you may already have set up. As long as each user can get shell access on the machine, any SSH authentication mechanism you can think of should work.

Generating Your SSH Public Key

That being said, many Git servers authenticate using SSH public keys. In order to provide a public key, each user in your system must generate one if they don't already have one. This process is similar across all operating systems. First, you should check to make sure you don't already have a key. By default, a user's SSH keys are stored in that user's `~/.ssh` directory. You can easily check to see if you have a key already by going to that directory and listing the contents:

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

You're looking for a pair of files named something like `id_dsa` or `id_rsa` and a matching file with a `.pub` extension. The `.pub` file is your public key, and the other file is your private key. If you don't have these files (or you don't even have a `.ssh` directory), you can create them by running a program called `ssh-keygen`, which is provided with the SSH package on Linux/Mac systems and comes with Git for 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
```

First it confirms where you want to save the key (`.ssh/id_rsa`), and then it asks twice for a passphrase, which you can leave empty if you don't want to type a password when you use the key.

Now, each user that does this has to send their public key to you or whoever is administrating the Git server (assuming you're using an SSH server setup that requires public keys). All they have to

do is copy the contents of the `.pub` file and email it. The public keys look something like this:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

For a more in-depth tutorial on creating an SSH key on multiple operating systems, see the GitHub guide on SSH keys at https://help.github.com/articles/generating-ssh-keys.

Setting Up the Server

Let's walk through setting up SSH access on the server side. In this example, you'll use the `authorized_keys` method for authenticating your users. We also assume you're running a standard Linux distribution like Ubuntu. First, you create a `git` user and a `.ssh` directory for that user.

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

Next, you need to add some developer SSH public keys to the `authorized_keys` file for the `git` user. Let's assume you have some trusted public keys and have saved them to temporary files. Again, the public keys look something like this:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

You just append them to the `git` user's `authorized_keys` file in its `.ssh` directory:

```
$ 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
```

Now, you can set up an empty repository for them by running `git init` with the `--bare` option, which initializes the repository without a working directory:

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

Then, John, Josie, or Jessica can push the first version of their project into that repository by adding it as a remote and pushing up a branch. Note that someone must shell onto the machine and create a bare repository every time you want to add a project. Let's use `gitserver` as the hostname of the server on which you've set up your `git` user and repository. If you're running it internally, and you set up DNS for `gitserver` to point to that server, then you can use the commands pretty much as is (assuming that `myproject` is an existing project with files in it):

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

At this point, the others can clone it down and push changes back up just as easily:

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

With this method, you can quickly get a read/write Git server up and running for a handful of developers.

You should note that currently all these users can also log into the server and get a shell as the `git` user. If you want to restrict that, you will have to change the shell to something else in the `passwd` file.

You can easily restrict the `git` user to only doing Git activities with a limited shell tool called `git-shell` that comes with Git. If you set this as your `git` user's login shell, then the `git` user can't have normal shell access to your server. To use this, specify `git-shell` instead of bash or csh for your user's login shell. To do so, you must first add `git-shell` to `/etc/shells` if it's not already there:

```
$ 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
```

Now you can edit the shell for a user using `chsh <username> -s <shell>`:

```
$ sudo chsh git -s $(which git-shell)
```

Now, the `git` user can only use the SSH connection to push and pull Git repositories and can't shell onto the machine. If you try, you'll see a login rejection like this:

```
$ 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.
```

Now Git network commands will still work just fine but the users won't be able to get a shell. As the output states, you can also set up a directory in the `git` user's home directory that customizes the `git-shell` command a bit. For instance, you can restrict the Git commands that the server will accept or you can customize the message that users see if they try to SSH in like that. Run `git help shell` for more information on customizing the shell.

Git Daemon

Next we'll set up a daemon serving repositories over the "Git" protocol. This is common choice for fast, unauthenticated access to your Git data. Remember that since it's not an authenticated service, anything you serve over this protocol is public within its network.

If you're running this on a server outside your firewall, it should only be used for projects that are publicly visible to the world. If the server you're running it on is inside your firewall, you might use it for projects that a large number of people or computers (continuous integration or build servers) have read-only access to, when you don't want to have to add an SSH key for each.

In any case, the Git protocol is relatively easy to set up. Basically, you need to run this command in a daemonized manner:

```
$ git daemon --reuseaddr --base-path=/srv/git/ /srv/git/
```

`--reuseaddr` allows the server to restart without waiting for old connections to time out, the `--base-path` option allows people to clone projects without specifying the entire path, and the path at the end tells the Git daemon where to look for repositories to export. If you're running a firewall, you'll also need to punch a hole in it at port 9418 on the box you're setting this up on.

You can daemonize this process a number of ways, depending on the operating system you're running. On an Ubuntu machine, you can use an Upstart script. So, in the following file

```
/etc/init/local-git-daemon.conf
```

you put this script:

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

For security reasons, it is strongly encouraged to have this daemon run as a user with read-only permissions to the repositories – you can easily do this by creating a new user *git-ro* and running the daemon as them. For the sake of simplicity we'll simply run it as the same *git* user that `git-shell` is running as.

When you restart your machine, your Git daemon will start automatically and respawn if it goes down. To get it running without having to reboot, you can run this:

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

On other systems, you may want to use `xinetd`, a script in your `sysvinit` system, or something else – as long as you get that command daemonized and watched somehow.

Next, you have to tell Git which repositories to allow unauthenticated Git server-based access to. You can do this in each repository by creating a file named `git-daemon-export-ok`.

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

The presence of that file tells Git that it's OK to serve this project without authentication.

Smart HTTP

We now have authenticated access through SSH and unauthenticated access through `git://`, but there is also a protocol that can do both at the same time. Setting up Smart HTTP is basically just enabling a CGI script that is provided with Git called `git-http-backend` on the server. This CGI will read the path and headers sent by a `git fetch` or `git push` to an HTTP URL and determine if the client can communicate over HTTP (which is true for any client since version 1.6.6). If the CGI sees that the client is smart, it will communicate smartly with it, otherwise it will fall back to the dumb behavior (so it is backward compatible for reads with older clients).

Let's walk through a very basic setup. We'll set this up with Apache as the CGI server. If you don't have Apache setup, you can do so on a Linux box with something like this:

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

This also enables the `mod_cgi`, `mod_alias`, and `mod_env` modules, which are all needed for this to work properly.

You'll also need to set the Unix user group of the `/srv/git` directories to `www-data` so your web server can read- and write-access the repositories, because the Apache instance running the CGI script will (by default) be running as that user:

```
$ chgrp -R www-data /srv/git
```

Next we need to add some things to the Apache configuration to run the `git-http-backend` as the handler for anything coming into the `/git` path of your web server.

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

If you leave out `GIT_HTTP_EXPORT_ALL` environment variable, then Git will only serve to unauthenticated clients the repositories with the `git-daemon-export-ok` file in them, just like the Git daemon did.

Finally you'll want to tell Apache to allow requests to `git-http-backend` and make writes be authenticated somehow, possibly with an Auth block like this:

```
<Files "git-http-backend">
    AuthType Basic
    AuthName "Git Access"
    AuthUserFile /srv/git/.htpasswd
    Require expr !(%{QUERY_STRING} -strmatch '*service=git-receive-pack*' ||
%{REQUEST_URI} =~ m#/git-receive-pack$#)
    Require valid-user
</Files>
```

That will require you to create a `.htpasswd` file containing the passwords of all the valid users. Here is an example of adding a "schacon" user to the file:

```
$ htpasswd -c /srv/git/.htpasswd schacon
```

There are tons of ways to have Apache authenticate users, you'll have to choose and implement one of them. This is just the simplest example we could come up with. You'll also almost certainly want to set this up over SSL so all this data is encrypted.

We don't want to go too far down the rabbit hole of Apache configuration specifics, since you could well be using a different server or have different authentication needs. The idea is that Git comes with a CGI called `git-http-backend` that when invoked will do all the negotiation to send and receive data over HTTP. It does not implement any authentication itself, but that can easily be controlled at

the layer of the web server that invokes it. You can do this with nearly any CGI-capable web server, so go with the one that you know best.

NOTE	For more information on configuring authentication in Apache, check out the Apache docs here: http://httpd.apache.org/docs/current/howto/auth.html

GitWeb

Now that you have basic read/write and read-only access to your project, you may want to set up a simple web-based visualizer. Git comes with a CGI script called GitWeb that is sometimes used for this.

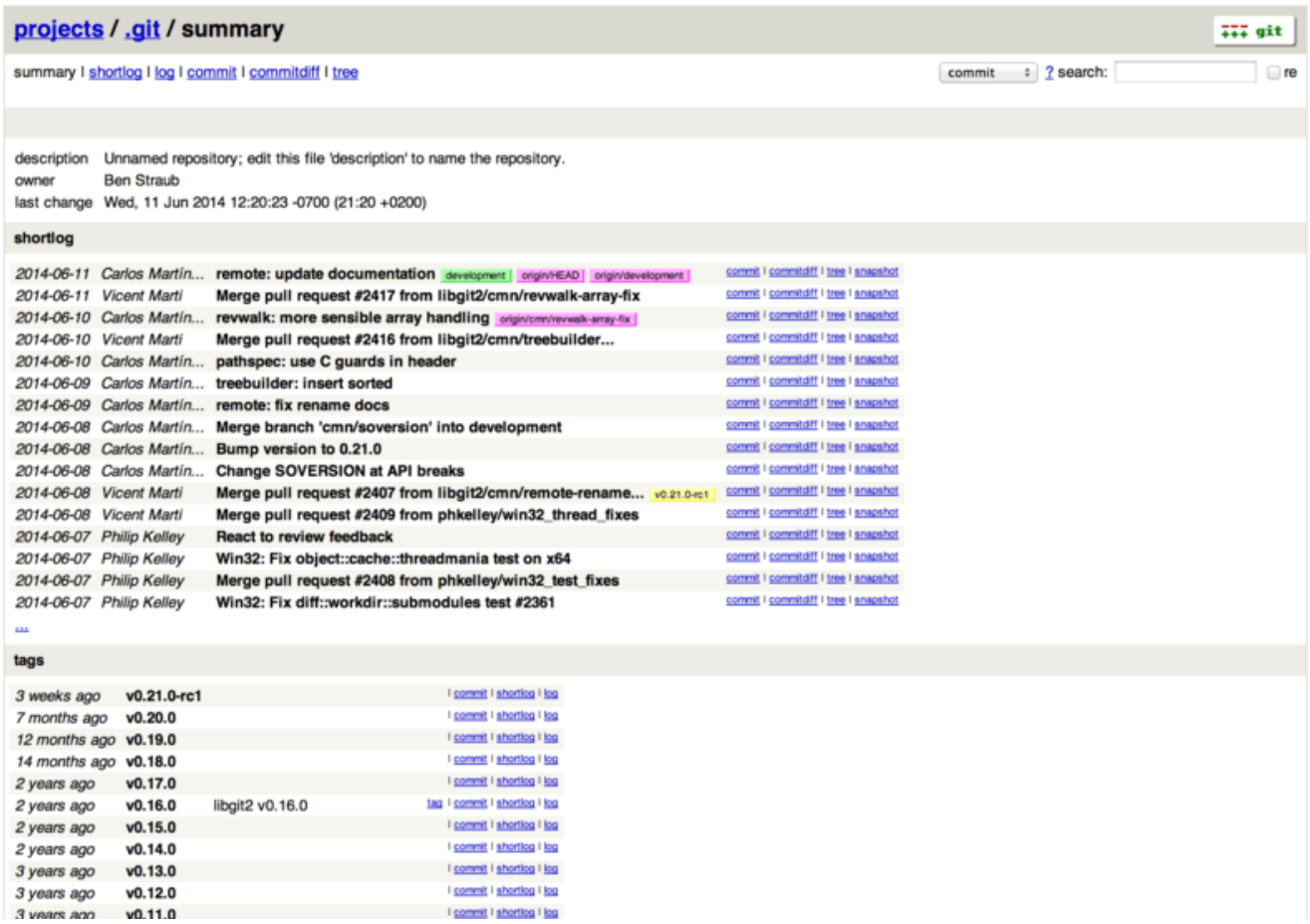

Figure 42. The GitWeb web-based user interface.

If you want to check out what GitWeb would look like for your project, Git comes with a command to fire up a temporary instance if you have a lightweight server on your system like `lighttpd` or `webrick`. On Linux machines, `lighttpd` is often installed, so you may be able to get it to run by typing `git instaweb` in your project directory. If you're running a Mac, Leopard comes preinstalled with Ruby, so `webrick` may be your best bet. To start `instaweb` with a non-lighttpd handler, you can run it with the `--httpd` option.

```
$ 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]
```

That starts up an HTTPD server on port 1234 and then automatically starts a web browser that opens on that page. It's pretty easy on your part. When you're done and want to shut down the server, you can run the same command with the `--stop` option:

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

If you want to run the web interface on a server all the time for your team or for an open source project you're hosting, you'll need to set up the CGI script to be served by your normal web server. Some Linux distributions have a `gitweb` package that you may be able to install via `apt` or `yum`, so you may want to try that first. We'll walk through installing GitWeb manually very quickly. First, you need to get the Git source code, which GitWeb comes with, and generate the custom CGI script:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/srv/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/
```

Notice that you have to tell the command where to find your Git repositories with the `GITWEB_PROJECTROOT` variable. Now, you need to make Apache use CGI for that script, for which you can add a 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>
```

Again, GitWeb can be served with any CGI or Perl capable web server; if you prefer to use something else, it shouldn't be difficult to set up. At this point, you should be able to visit `http://gitserver/` to view your repositories online.

GitLab

GitWeb is pretty simplistic though. If you're looking for a more modern, fully featured Git server,

there are some several open source solutions out there that you can install instead. As GitLab is one of the more popular ones, we'll cover installing and using it as an example. This is a bit more complex than the GitWeb option and likely requires more maintenance, but it is a much more fully featured option.

Installation

GitLab is a database-backed web application, so its installation is a bit more involved than some other Git servers. Fortunately, this process is very well-documented and supported.

There are a few methods you can pursue to install GitLab. To get something up and running quickly, you can download a virtual machine image or a one-click installer from https://bitnami.com/stack/gitlab, and tweak the configuration to match your particular environment. One nice touch Bitnami has included is the login screen (accessed by typing alt-→); it tells you the IP address and default username and password for the installed GitLab.

Figure 43. The Bitnami GitLab virtual machine login screen.

For anything else, follow the guidance in the GitLab Community Edition readme, which can be found at https://gitlab.com/gitlab-org/gitlab-ce/tree/master. There you'll find assistance for installing GitLab using Chef recipes, a virtual machine on Digital Ocean, and RPM and DEB packages (which, as of this writing, are in beta). There's also "unofficial" guidance on getting GitLab running with non-standard operating systems and databases, a fully-manual installation script, and many other topics.

Administration

GitLab's administration interface is accessed over the web. Simply point your browser to the hostname or IP address where GitLab is installed, and log in as an admin user. The default username is `admin@local.host`, and the default password is `5iveL!fe` (which you will be prompted to

change as soon as you enter it). Once logged in, click the “Admin area” icon in the menu at the top right.

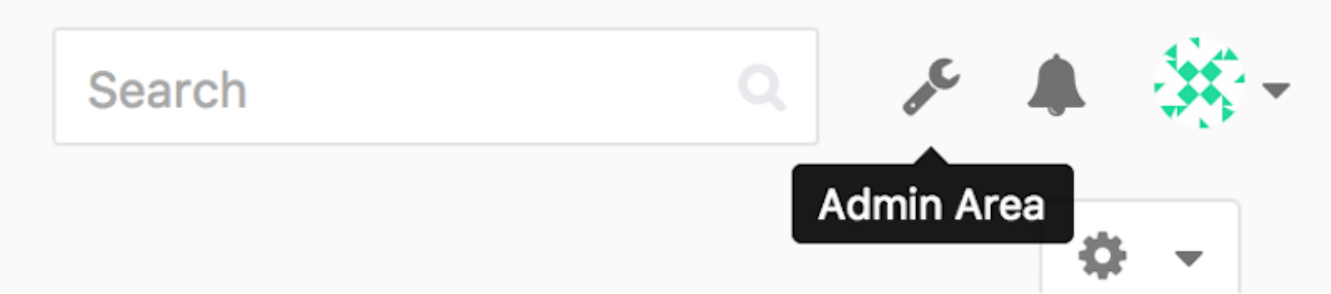

Figure 44. The “Admin area” item in the GitLab menu.

Users

Users in GitLab are accounts that correspond to people. User accounts don’t have a lot of complexity; mainly it’s a collection of personal information attached to login data. Each user account comes with a **namespace**, which is a logical grouping of projects that belong to that user. If the user `jane` had a project named `project`, that project’s url would be http://server/jane/project.

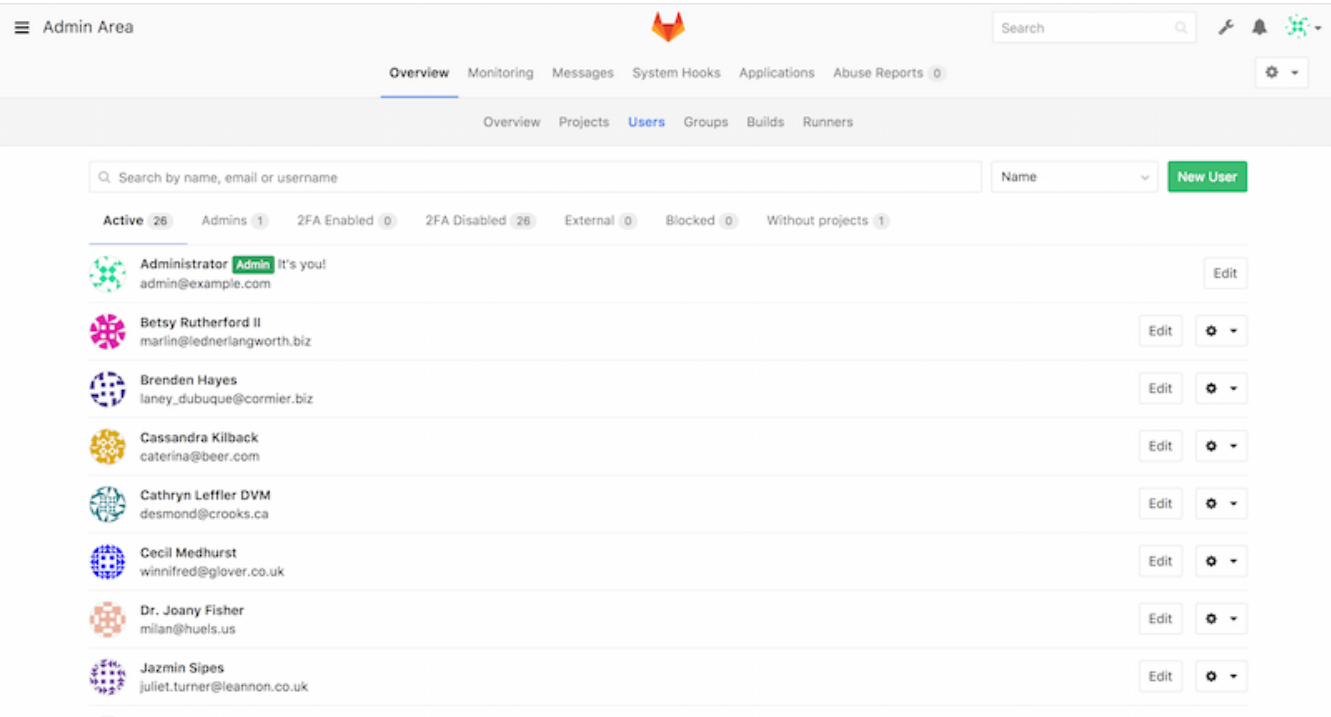

Figure 45. The GitLab user administration screen.

Removing a user can be done in two ways. “Blocking” a user prevents them from logging into the GitLab instance, but all of the data under that user’s namespace will be preserved, and commits signed with that user’s email address will still link back to their profile.

“Destroying” a user, on the other hand, completely removes them from the database and filesystem. All projects and data in their namespace is removed, and any groups they own will also be removed. This is obviously a much more permanent and destructive action, and its uses are rare.

Groups

A GitLab group is an assemblage of projects, along with data about how users can access those

projects. Each group has a project namespace (the same way that users do), so if the group `training` has a project `materials`, its url would be http://server/training/materials.

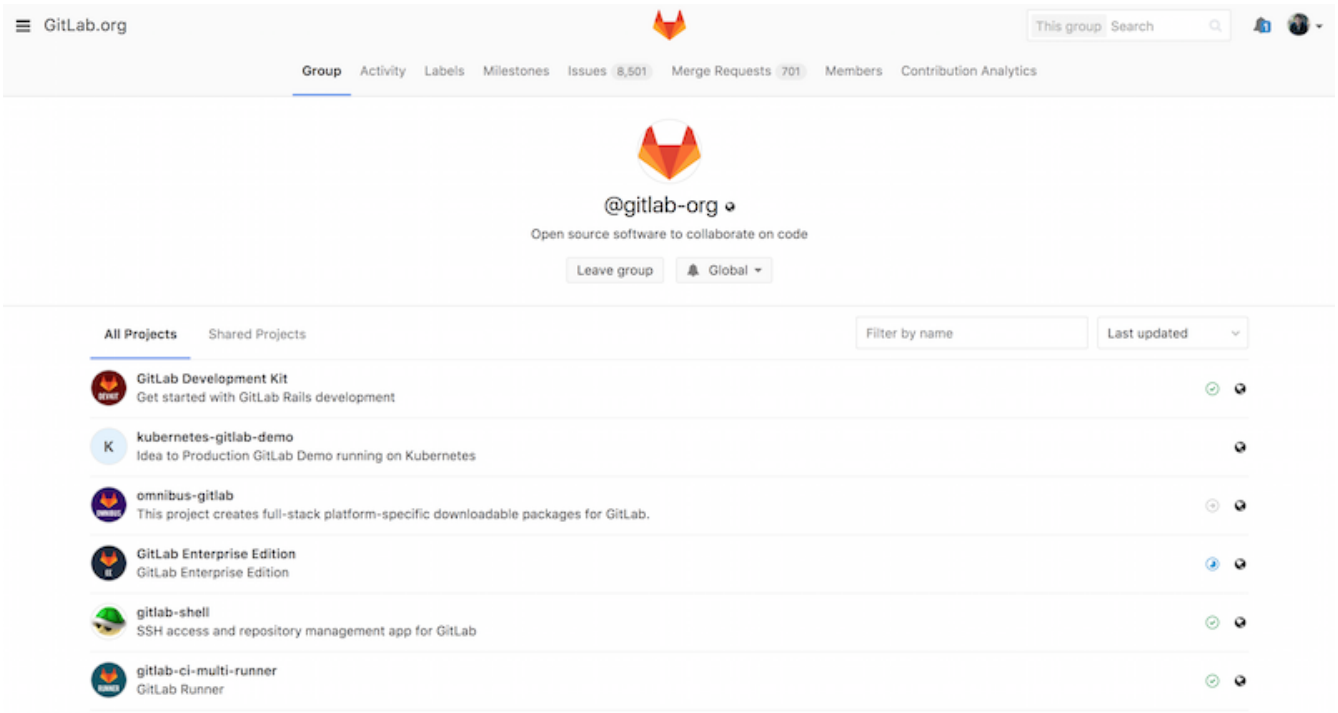

Figure 46. The GitLab group administration screen.

Each group is associated with a number of users, each of which has a level of permissions for the group's projects and the group itself. These range from "Guest" (issues and chat only) to "Owner" (full control of the group, its members, and its projects). The types of permissions are too numerous to list here, but GitLab has a helpful link on the administration screen.

Projects

A GitLab project roughly corresponds to a single Git repository. Every project belongs to a single namespace, either a user or a group. If the project belongs to a user, the owner of the project has direct control over who has access to the project; if the project belongs to a group, the group's user-level permissions will also take effect.

Every project also has a visibility level, which controls who has read access to that project's pages and repository. If a project is *Private*, the project's owner must explicitly grant access to specific users. An *Internal* project is visible to any logged-in user, and a *Public* project is visible to anyone. Note that this controls both `git fetch` access as well as access to the web UI for that project.

Hooks

GitLab includes support for hooks, both at a project or system level. For either of these, the GitLab server will perform an HTTP POST with some descriptive JSON whenever relevant events occur. This is a great way to connect your Git repositories and GitLab instance to the rest of your development automation, such as CI servers, chat rooms, or deployment tools.

Basic Usage

The first thing you'll want to do with GitLab is create a new project. This is accomplished by

clicking the "+" icon on the toolbar. You'll be asked for the project's name, which namespace it should belong to, and what its visibility level should be. Most of what you specify here isn't permanent, and can be re-adjusted later through the settings interface. Click "Create Project", and you're done.

Once the project exists, you'll probably want to connect it with a local Git repository. Each project is accessible over HTTPS or SSH, either of which can be used to configure a Git remote. The URLs are visible at the top of the project's home page. For an existing local repository, this command will create a remote named `gitlab` to the hosted location:

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

If you don't have a local copy of the repository, you can simply do this:

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

The web UI provides access to several useful views of the repository itself. Each project's home page shows recent activity, and links along the top will lead you to views of the project's files and commit log.

Working Together

The simplest way of working together on a GitLab project is by giving another user direct push access to the Git repository. You can add a user to a project by going to the "Members" section of that project's settings, and associating the new user with an access level (the different access levels are discussed a bit in Groups). By giving a user an access level of "Developer" or above, that user can push commits and branches directly to the repository with impunity.

Another, more decoupled way of collaboration is by using merge requests. This feature enables any user that can see a project to contribute to it in a controlled way. Users with direct access can simply create a branch, push commits to it, and open a merge request from their branch back into `master` or any other branch. Users who don't have push permissions for a repository can "fork" it (create their own copy), push commits to *that* copy, and open a merge request from their fork back to the main project. This model allows the owner to be in full control of what goes into the repository and when, while allowing contributions from untrusted users.

Merge requests and issues are the main units of long-lived discussion in GitLab. Each merge request allows a line-by-line discussion of the proposed change (which supports a lightweight kind of code review), as well as a general overall discussion thread. Both can be assigned to users, or organized into milestones.

This section is focused mainly on the Git-related features of GitLab, but as a mature project, it provides many other features to help your team work together, such as project wikis and system maintenance tools. One benefit to GitLab is that, once the server is set up and running, you'll rarely need to tweak a configuration file or access the server via SSH; most administration and general usage can be accomplished through the in-browser interface.

Third Party Hosted Options

If you don't want to go through all of the work involved in setting up your own Git server, you have several options for hosting your Git projects on an external dedicated hosting site. Doing so offers a number of advantages: a hosting site is generally quick to set up and easy to start projects on, and no server maintenance or monitoring is involved. Even if you set up and run your own server internally, you may still want to use a public hosting site for your open source code – it's generally easier for the open source community to find and help you with.

These days, you have a huge number of hosting options to choose from, each with different advantages and disadvantages. To see an up-to-date list, check out the GitHosting page on the main Git wiki at https://git.wiki.kernel.org/index.php/GitHosting

We'll cover using GitHub in detail in [_github], as it is the largest Git host out there and you may need to interact with projects hosted on it in any case, but there are dozens more to choose from should you not want to set up your own Git server.

Summary

You have several options to get a remote Git repository up and running so that you can collaborate with others or share your work.

Running your own server gives you a lot of control and allows you to run the server within your own firewall, but such a server generally requires a fair amount of your time to set up and maintain. If you place your data on a hosted server, it's easy to set up and maintain; however, you have to be able to keep your code on someone else's servers, and some organizations don't allow that.

It should be fairly straightforward to determine which solution or combination of solutions is appropriate for you and your organization.