Chapter 8. Source Control with Git

So far in this book, we've shown you lots of ways to add automation to your toolbox, whether via scripting languages like Python (see Chapter 4) or via templating languages like Jinja (see Chapter 6). The increased use of Python-based scripts or Jinja templates means that managing these artifacts (and by *artifacts* we mean the files that make up these scripts, templates, and other automation tools you're employing) is very important. In particular, managing the *changes* to these artifacts has significant value (we'll explain why shortly).

In this chapter, we're going to show you how to use a *source control* tool—that is, a tool that is designed to manage the artifacts you're creating and using in your network automation processes. The use of a source control tool lets you avoid messy and error-prone approaches like appending date- and timestamps to the end of filenames, and keeps you from running into accidentally deleted or overwritten files.

To start things out, let's take a closer look at the idea of source control. We're going to keep the discussion fairly generic for now; we'll delve into a specific source control tool known as Git later in the chapter. The generic qualities discussed in the next section are not specific to any particular source control tool, however.

Use Cases for Source Control

Simply put, *source control* is a way of tracking files and the changes made to those files over time (source control is also known as *version control* or *revision control*). We know that's a really generic description, so let's look at some specific use cases:

- If you're a developer writing code as part of a larger software development project, you could track the code you're writing using source control tools. This use case is probably the most well-known use case, and the one most people immediately think about when we mention the idea of source control.

- Let's say you're part of a team of administrators managing network devices. You could take the device configuration files and track them using source control tools.
- Suppose you're responsible for maintaining documentation for portions of your organization's Information Technology (IT) infrastructure. You could track the documentation using source control tools.

In each of these cases, source control is tracking files (network configurations, documentation, software source code). By *tracking* these files, we mean that the source control tool is keeping a record of the files, the changes made to the files over time, and who made each set of changes to the files. If a change to one of the files being tracked breaks something, you can revert or roll back to a previous version of the file, undoing the changes and getting back to a known good state. In some cases (depending on the tool being used), source control tools might enable you to more easily collaborate with coworkers in a distributed fashion.

Benefits of Source Control

The previous section indirectly outlined some of the benefits of using a source control tool, but let's pull out a few specific benefits that come from the use of source control.

Change Tracking

First, you're able to track the changes to the files stored in the source control tool over time. You can see the state of the files at any given point in time, and therefore you're able to relatively easily see exactly *what* changed. This is an often overlooked benefit. When you're working with lengthy network configuration files, wouldn't it be helpful to be able to see *exactly* what changed from one version to the next? Further, most source control tools also have the ability to add metadata about the change, such as why a change was made or a reference back to an issue or trouble ticket. This additional metadata can also prove quite useful in troubleshooting.

Accountability

Not only do the source control tools track changes over time, but they also track *who* made the changes. Every change is logged with who made that particular change. In a team environment where multiple team members might be working together to manage network configurations or server configuration files, this is extraordinarily useful. Never again will you have to ask, “Who made this change?” The source control tool will already have that information.

Process and Workflow

Using source control tools also helps you and your organization enforce a healthy process and workflow. We’ll get into this more in Chapter 10, but for now think about the requirement that all changes be logged in source control first *before* being pushed into production. This gives you a linear history of changes along with a log of the individual responsible for each set of changes, and enables you to enforce things like review (having someone else review your changes before they get put into production) or testing (having automated tests performed against the files in the source control system).

Benefits of Source Control for Networking

Although source control is most typically associated with software development, there are clear benefits for networking professionals. Here are just a few examples:

- Python scripts (such as the ones readers will be able to write after reading this book!) that interact with network devices can be placed in source control, so that versions of the script can be more easily managed.
- Network device configurations can be placed in source control, enabling you to see the state of a network device configuration at any point in time. A really well-known tool called RANCID uses this approach for storing network device configuration backups.
- It’s easy to highlight the changes between versions of network device configurations, allowing you and your team to easily verify that only the desired changes are in place (e.g., that you didn’t accidentally prune a VLAN from the wrong 802.1Q trunk).

- Configuration templates can be placed in source control, ensuring that you and your team can track changes to these templates *before* they are used to generate network device configurations or reports.
- You can use source control with network documentation.
- All changes to any of these types of files are captured along with the person responsible for the changes—no more "playing the blame game."

Now that you have an idea of the benefits that source control can bring to you, your organization, and your workflow, let's take a look at a specific source control tool that is widely used: Git.

Enter Git

Git is the latest in a long series of source control tools, and has emerged as the de facto source control tool for most open source projects. (It doesn't hurt that Git manages the source code for the Linux kernel.) For that reason, we'll focus our discussion of source control tools on Git, but keep in mind that other source control tools do exist. They are, unfortunately, out of the scope of this book.

Let's start with a brief history of how and why Git appeared.

Brief History of Git

As we mentioned earlier, Git is the source control tool used to manage the source code for the Linux kernel. Git was launched by Linus Torvalds, the creator of the Linux kernel, in early April 2005 in response to a disagreement between the Linux kernel developer community and the proprietary system they were using at the time (a system called BitKeeper).

Torvalds had a few key design goals when he set out to create Git:

Speed

Torvalds needed Git to be able to rapidly apply patches to the Linux source code.

Simplicity

The design for Git needed to be as simple as possible.

Strong support for nonlinear development

> The Linux kernel developers needed a system that could handle lots of parallel branches. Thus, this new system (Git) needed to support rapid branching and merging, and branches needed to be as lightweight as possible.

Support for fully distributed operation

> Every developer needed a full copy of the entire source code and its history.

Scalability

> Git needed to be scalable enough to handle large projects, like the Linux kernel.

Development of Git was fast. Within a few days of its launch, Git was self-hosted (meaning that the source code for Git was being managed by Git). The first merge of multiple branches occurred just a couple weeks later. At the end of April—just a few weeks after its launch—Git was benchmarked at applying patches to the Linux kernel tree at 6.7 patches per second. In June 2005, Git managed the 2.6.12 release of the Linux kernel, and the 1.0 release of Git occurred in late December 2005.

As of this writing, the most recent release of Git was version 2.13.1, and versions of Git were available for all major desktop operating systems (Linux, Windows, and macOS). Notable open source projects using Git include the Linux kernel (as we've already mentioned), Perl, the Gnome desktop environment, Android, the KDE Project, and the X.Org implementation of the X Window System. Additionally, some very popular online source control services are based on Git, including services like GitHub, BitBucket, and GitLab. Some of these services also offer on-premises implementations. You'll get the opportunity to look more closely at GitLab in Chapter 10, when we discuss Continuous Integration.

Git Terminology

Before we progress any further, let's be sure that we've properly defined some terminology. Some of these terms we may have used before, but we'll include them here for the sake of completeness.

Repository

In Git, a repository is the name given to the database that contains all of a project's information (files and metadata) and history. (We're using the term *project* here just to refer to an arbitrary grouping of files for a particular purpose or effort.) A repository is a complete copy of all of the files and information associated with a project throughout the lifetime of the project. It's important to note that once data is added to a repository it is immutable; that is, it can't be changed once added. This *isn't* to say that you can't make changes to files stored in a repository, just that the repository stores and tracks these files in such a way that changes to a file create a new entry in the repository (specifically, Git uses SHA hashes to create content-addressable objects in the repository).

Working directory

This is the directory where you, as the user of Git, will modify the files contained in the repository. The working directory is *not* the same as the repository. Note that the term *working directory* is also used for other purposes on Linux/UNIX/macOS systems (to refer to the current directory, as output by the `pwd` command). Git's working directory is *not* the same as the current directory, and very specifically refers to the directory where the *.git* repository is stored.

Index

The index describes the repository's directory structure and contents at a point in time. The index is a dynamic binary file maintained by Git and modified as you stage changes and commit them to the repository.

Commit

A commit is an entry in the Git repository recording metadata for each change introduced to the repository. This metadata includes the author, the date of the commit, and a commit message (a description of the change introduced to the repository). Additionally, a commit captures the state of the entire repository at the time the commit was performed. Keep in mind when we say "a change to the repository" this might mean multiple changes to multiple files; Git allows you to lump changes to multiple files together as a single commit. (We'll discuss this in a bit more detail later in this chapter.)

Overview of Git's Architecture

With the terminology from the previous section in mind, we can now provide an overview of Git's architecture. We'll limit our discussion of Git's architecture to keep it relatively high-level, but detailed enough to help with your understanding of how Git operates.

NOTE

For a more in-depth discussion of Git's architecture, we recommend *Version Control with Git*, Second Edition (O'Reilly).

As we described earlier, a Git *repository* is a database that contains all of the information about a project: the files contained in the project, the changes made to the project over time, and the metadata about those changes (who made the change, when the change was made, etc.). By default, this information is stored in a directory named *.git* in the root of your working directory (this behavior can be changed). For example, here's a file listing of a newly initialized Git repository's working directory, showing the *.git* directory where the actual repository data is found:

```
relentless:npab-examples slowe (master)$ ls -la
total 0
drwxr-xr-x   3 slowe  staff  102 May 11 15:37 .
drwxr-xr-x  16 slowe  staff  544 May 11 15:37 ..
drwxr-xr-x  10 slowe  staff  340 May 11 15:37 .git
relentless:npab-examples slowe (master)$
```

As you can tell from this prompt, this directory listing is from the directory *npab-examples*. In this example, the *working directory* is the *npab-examples* directory, and the Git *repository* is in *npab-examples/.git*. This is why we said earlier that the working directory and the repository aren't the same. It's common for new users to refer to the working directory as the repository, but keep in mind that the actual repository is in the *.git* subdirectory.

Within the *.git* directory you'll find all the various components that make up a Git repository:

- The index—which we defined earlier as representing the repository's directory structure and contents at a given point in time—is found at *.git/index*.
- The files contained within a Git repository are treated as content-addressable objects and stored in subdirectories in *.git/objects*.

- Any repository-specific configuration details are found in *.git/config*.
- Metadata about the repository, the changes stored in the repository, and the objects in the repository can be found in *.git/logs*.

All of the information stored in the *.git* directory is maintained by Git—you should never need to directly interact with the contents of this directory. Over the course of this chapter, we'll share with you the various commands that are needed to interact with the repository to add files, commit changes, revert changes, and more. In fact, that leads us directly into our next section, which will show you how to work with Git.

Working with Git

Now that you have an idea of what Git's architecture looks like, let's shift our focus to something a bit more practical: actually *working* with Git.

Throughout our discussion of working with Git, we're going to use a (hopefully) very practical example. Let's assume that you are a network engineer reponsible for rolling out some network automation tools in your environment. During this process, you're going to end up creating Python scripts, Jinja templates, and other files. You'd like to use Git to manage these files so that you can take advantage of all the benefits of source control.

The following sections walk you through each of the major steps in getting started using Git to manage the files created as part of your network automation effort.

Installing Git

The steps for installing Git are extremely well documented, so we won't go through them here. Git is often preinstalled in various distributions of Linux; if not, Git is almost always available to install via the Linux distribution's package manager (such as `dnf` for RHEL/CentOS/Fedora, or `apt` for Debian/Ubuntu). Installers are available for macOS and Windows that make it easy to install Git. Detailed instructions and options for installing Git are also available on the Git website.

Creating a Repository

Once Git is installed, the first step is to create a repository. First, you'll create a directory where the repository will be stored. Assuming you're using an Ubuntu Linux system, it might look something like this (the commands would be very similar, if not identical, on other Linux distributions or on macOS):

```
vagrant@trusty:~$ mkdir ~/net-auto
vagrant@trusty:~$
```

Then you can change into this directory and create the empty repository using the `git init` command:

```
vagrant@trusty:~$ cd net-auto
vagrant@trusty:~/net-auto$ git init
Initialized empty Git repository in /home/vagrant/net-auto/.git
vagrant@trusty:~/net-auto$
```

The `git init` command is responsible for initializing, or creating, a new Git repository. This involves creating the *.git* directory and all of the subdirectories and contents found within them.

NOTE

The various `git` commands we describe throughout this chapter should be very nearly identical across all systems on which Git runs. We'll use various Linux distributions (and this will be reflected in the shell prompts in the examples we show), but using Git on macOS should be the same as on Linux. Using Git on Windows should be similar, but there may be syntactical differences here and there due to the differences in the underlying operating systems.

If you were now to run `ls -la` in the *net-auto* directory, you'd see the *.git* directory that stores the empty Git repository created by the `git init` command. The repository is now ready for you to start adding content. You'd add content to a repository by adding files.

Adding Files to a Repository

Adding files to repository is a multistage process:

1. Add the files to the repository's working directory.
2. Stage the files to the repository's index.

3. Commit the staged files to the repository.

Let's go back to our example. You've created your new Git repository to store files created as part of your network automation project, and some of the first files you'd like to add to the repository are the current configuration files from your network devices. You already have three configuration files: *sw1.txt*, *sw2.txt*, and *sw3.txt*, that contain the current configurations for three switches.

First, copy the files into the working directory (in our example, */home/vagrant/net-auto*). More generically, remember the working directory is the parent directory of the *.git* directory (which holds the actual Git repository). On a Linux or macOS system, copying files into the working directory would involve the `cp` command; on a Windows-based machine, you'd use the `copy` command.

The files are now in the working directory but are *not* in the repository itself. This means that Git is not tracking the files or their content, and therefore you can't track changes, know who made the changes, or roll back to an earlier version.

You can verify this by running the `git status` command, which in this example would produce some output that looks like this:

```
vagrant@trusty:~/net-auto$ git status
On branch master

Initial commit

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

    sw1.txt
    sw2.txt
    sw3.txt

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

The output of the `git status` command tells you that untracked files are present in the working directory and nothing has been added to the repository. As the output indicates, you'll need to use the `git add` command to add these untracked files to the repository, like this:

```
vagrant@trusty:~/net-auto$ git add sw1.txt
vagrant@trusty:~/net-auto$ git add sw2.txt
vagrant@trusty:~/net-auto$ git add sw3.txt
vagrant@trusty:~/net-auto$
```

You could also use shell globbing to add multiple files at the same time. For example, you could use `git add sw*.txt` to add all three switch configurations with a single command.

After you've used `git add` to add the files to the staging area, you can run `git status` again to see the current status:

```
vagrant@trusty:~/net-auto$ git status
On branch master

Initial commit

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

    new file:   sw1.txt
    new file:   sw2.txt
    new file:   sw3.txt

vagrant@trusty:~/net-auto$
```

At this point, the files have been *staged* into Git's index. This means that Git's index and the working directory are in sync. Technically speaking, the files have been added as objects to Git's object store as well, but there is no "point in time" reference to these objects. In order to create that point-in-time reference, you must first *commit* the staged changes.

Committing Changes to a Repository

Before you're ready to commit changes to a repository, there are a couple of things you'll need to be sure you've done. Recall that one of the benefits of using Git as a source control tool is not only that you're able to track the changes made to the files stored in the repository, but that you're also able to know who made each set of changes. In order to have that information, you'll first need to provide that information to Git. (You could also do this right after installing Git; it's not necessary to create a repository first.)

PROVIDING USER INFORMATION TO GIT

Git has a series of configuration options; some of these configuration options are repository-specific, some are user-specific, and some are system-wide. Recall from earlier that Git stores repository-specific configuration information in *.git/config*. In this particular case—where we need to provide the user's name and email address so that Git can track who made each set of changes—it's the user-specific configuration we need to modify, not the repository-specific configuration.

So where are these values stored? These settings are found in the *.gitconfig* file in your home directory. This file is an INI-style file, and you can edit it either using your favorite text editor or using the `git config` command. In this case, we'll show you how to use the `git config` command to set this information.

To set your name and email address, use the following commands:

```
vagrant@jessie:~/net-auto$ git config --global user.name "John
Smith"
vagrant@jessie:~/net-auto$ git config --global user.email
"john.smith@networktocode.com"
vagrant@jessie:~/net-auto$
```

We used the `--global` option here to set it as a user-specific value; if you wanted to set a different user name and/or email address as a repository-specific value, just omit the `--global` flag (but be sure you are in the working directory of an active repository first; Git will report an error otherwise). With the `--global` flag, `git config` modifies the *.gitconfig* file in your home directory; without it, `git config` modifies the *.git/config* file of the current repository.

COMMITTING CHANGES

When Git has been configured with your identity, you're ready to *commit* the changes you've made to the files into the repository. Remember that before you can commit changes into the repository you must first *stage* the files using the `git add` command; this is true both for newly created files as well as modified files that were already in the repository (we'll review that scenario shortly). Since you've already staged the changes (via the `git add` command earlier) and verified it (via the `git status` command, which shows the files are staged), then you're ready to commit.

Committing changes to a repository is as simple as using the `git commit` command:

```
vagrant@jessie:~/net-auto$ git commit -m "First commit to new
repository"
[master (root-commit) 9547063] First commit to new repository
 3 files changed, 24 insertions(+)
 create mode 100644 sw1.txt
 create mode 100644 sw2.txt
 create mode 100644 sw3.txt
vagrant@jessie:~/net-auto$
```

NOTE

If you omit the `-m` parameter to `git commit`, then Git will launch the default text editor so you can provide a commit message. The text editor that Git launches is configurable (via `git config` or editing *.gitconfig* in your home directory). You could, for example, configure Git to use Sublime Text, Atom, or some other graphical text editor.

So what's happening when you commit the changes to the repository? When you added the files via `git add`, objects representing the files (and the files' content) were added to Git's object database. Specifically, Git created *blobs* (binary large objects) to represent the files' content and *tree objects* to represent the files and their directory structure. When you commit the changes via `git commit`, you're adding another type of object to the Git database (a *commit object*) that references the tree objects, which in turn reference the blobs. With a commit object, you now have a "point in time" reference to the entire state of the repository.

At this point, your repository has a single commit, and you can see that commit using the `git log` command:

```
vagrant@jessie:~/net-auto$ git log
commit 95470631aba32d6823c80fdd3c6f923824dde470
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
vagrant@jessie:~/net-auto$
```

The `git log` command shows the various commits—or checkpoints, if you will—you created over the lifetime of the repository. Every time you commit changes, you create a commit object, and that commit object references the state of the repository at the time it was created. This means you can only view the state of the repository and its contents at the time of a commit. Commits, therefore, become the “checkpoints” by which you can move backward (or forward) through the history of the repository.

RECOMMENDATIONS FOR COMMITTING CHANGES

Understanding how commits work leads to a few recommendations around committing your changes to a repository:

Commit frequently

> You can only view the state of the repository at the time when changes are committed. If you make changes, save the files, make more changes, then save and commit, you won’t be able to view the state of the repository at the first set of changes (because you didn’t commit).

Commit at logical points

> Don’t commit every time you save changes to a file in the repository. We know this sounds like a contradiction to the previous bullet, but it really only makes sense to commit changes when the changes are complete. For example, committing changes when you’re only halfway through updating a switch’s configuration doesn’t make sense; you wouldn’t want to roll back to a half-completed switch configuration. Instead, commit when you’ve finished the switch configuration.

Use helpful commit messages

> As you can see from the previous `git log` output, commit messages help you understand the changes that were contained in that commit. Try to make your commit messages helpful and straightforward—it’s likely that in six months, the commit message will be the only clue to help you decipher what you were doing at that point in time.

Before we move on to the next section, there’s one more topic we need to discuss. We’ve explained that objects in a Git repository are immutable, and that changes to an object (like a file) result in the creation of a new object (addressed by the SHA hash of the object’s content). This is true for all objects in the Git repository, including blobs (file content), tree objects, and commit objects.

What if, though, you made a commit and realized the commit contained errors? Maybe you have some typos in your network configuration, or the commit message is wrong. In this case, Git allows you to modify (or *amend*) the last commit.

AMENDING COMMITS

In a situation where the last commit is incorrect for some reason, it is possible to *amend* the commit through the use of the `--amend` flag to `git commit`. Note that you could just make another commit instead of amending the previous commit; both approaches are valid and each approach has its advantages and disadvantages, which we'll discuss shortly. First, though, let's show you how to amend a commit.

To amend a commit, you'd follow the same set of steps as with a "normal" commit:

1. Make whatever changes you need to make.
2. Stage the changes.
3. Run `git commit --amend` to commit the changes, marking it as an amendment.

Under the hood, Git is actually creating new objects—which is in line with Git's philosophy and approach of content-addressable immutable objects—but in the history of the repository, you'll see *only* the amended commit, not the original commit. This results in a "cleaner" history, although some purists may argue that simply making another commit (instead of using `--amend`) would be the best approach.

Which approach is best? That is mostly decided by you, the user, but there are a couple considerations. If you're collaborating with others via Git and a shared repository, using `--amend` to amend commits already sent to the shared repository is generally a bad idea. The one exception to this would be in an environment using Gerrit, where amended commits are used extensively. We'll talk more about Gerrit in Chapter 10, and we'll cover collaborating with Git later in this chapter in the section "Collaborating with Git".

Changing and Committing Tracked Files

You've created a repository, added some new files, and committed changes to the repository. Now, though, you need to make some changes to the files that are already in the repository. How does that work?

Fortunately, the process for committing modified versions of files into a repository looks pretty much identical to what we've shown you already:

1. Modify the file(s) in the working directory.
2. Stage the change(s) to the index using `git add`. This puts the index in sync with the working directory.
3. Commit the changes using `git commit`. This puts the repository in sync with the index, and creates a point-in-time reference to the state of the repository.

Let's review this in a bit more detail. Suppose you needed to modify one of the files, *sw1.txt*, because the switch's configuration has changed (or perhaps because you're enforcing that configurations can only be deployed *after* they've been checked into source control). After a tracked file (a file about which Git already knows and is tracking) is modified, `git status` will show that changes are present:

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

    modified:   sw1.txt

no changes added to commit (use "git add" and/or "git commit -a")
vagrant@trusty:~/net-auto$
```

Note the difference between this status message and the status message we showed you earlier. In this case, Git knows about the *sw1.txt* file (it's already been added to the repository), so the status message is different. Note how the status message changes if you were to add another switch configuration file, *sw4.txt*, to the working directory:

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

    modified:   sw1.txt

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

    sw4.txt

no changes added to commit (use "git add" and/or "git commit -a")
vagrant@trusty:~/net-auto$
```

Again, Git provides a clear distinction between tracking changes to an already known file and detecting untracked (not previously added) files to the working directory. Either way, though, the process for getting these changes (modified file and new file) into the repository is exactly the same, as you can see in the output of the `git status` command: just use the `git add` command, then the `git commit` command.

```
vagrant@trusty:~/net-auto$ git add sw1.txt
vagrant@trusty:~/net-auto$ git add sw4.txt
vagrant@trusty:~/net-auto$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   sw1.txt
    new file:   sw4.txt

vagrant@trusty:~/net-auto$ git commit -m "Update sw1, add sw4"
[master 679c41c] Update sw1, add sw4
 2 files changed, 9 insertions(+)
 create mode 100644 sw4.txt
vagrant@trusty:~/net-auto$
```

In the output of the `git status` commands, you may have noticed a reference to `git commit -a`. The `-a` option simply tells Git to add all changes from all known files. If you're only committing changes to known files *and* you are OK with committing all the changes together in a single commit, then using `git commit -a` allows you to avoid using the `git add` command first.

If, however, you want to break up changes to multiple files into separate commits, then you'll need to use `git add` followed by `git commit` instead. Why might you want to do this?

- You might want to limit the scope of changes in a single commit so that it's less impactful to revert to an earlier version.
- You may want to limit the scope of changes in a single commit so that others can review your changes more easily. (This is something we'll discuss in more detail in Chapter 10.)
- When collaborating with others, it's often considered a "best practice" to limit commits to a single logical change, which means you may include some changes in a commit but not others. We'll discuss some general guidelines for collaborating with Git later in this chapter in the section "Collaborating with Git".

You'll also notice that we've been using `git commit -m` in our examples here. The `-m` option allows the user to include a commit message on the command line. If you don't include the `-m`, then Git will open your default editor so that you can supply a commit message. Commit messages are required, and as we mentioned earlier we recommend that you make your commit messages as informative as possible. (You'll be thankful for informative commit messages when reviewing the output of `git log` in the future.) You can also combine both the `-a` and `-m` options, as in `git commit -am "Committing all changes to tracked files"`.

NOTE

For more information on the various options to any of the `git` commands, just type **`git help <command>`**, like `git help commit` or `git help add`. This will open the man page for that part of Git's documentation. If you like to use the `man` command instead, you can do that too; just put a dash into the `git` command. Thus, to see the man page for `git commit`, you'd enter **`man git-commit`**.

Now that you've committed another set of changes to the repository, let's look at the output of `git log`:

```
vagrant@trusty:~/net-auto$ git log
commit 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 20:41:19 2016 +0000

    Update sw1, add sw4

commit 95470631aba32d6823c80fdd3c6f923824dde470
```

```
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
vagrant@trusty:~/net-auto$
```

Your repository now has two commits. Before we explore how to view a repository at a particular point in time (at a particular commit), let's first review a few other commands and make some additional commits to the repository.

Unstaging Files

If you've been following along, your repository now has four switch configuration files (*sw1.txt* through *sw4.txt*) and two commits. Let's say you need to add a fifth switch configuration file (named *sw5.txt*, of course). You know already the process to follow:

1. Copy the file *sw5.txt* into the working directory.
2. Use `git add` to stage the file from the working directory into the index.

At this point, running `git status` will report that *sw5.txt* has been staged and is ready to commit to the repository:

```
vagrant@jessie:~/net-auto$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   sw5.txt

vagrant@jessie:~$
```

However, you realize after staging the file that you aren't ready to commit the file in its current state. Maybe the file isn't complete, or perhaps the file doesn't accurately reflect the actual configuration of "sw5" on the network. In such a situation, the best approach would be to *unstage* the file.

The command to unstage the file—that is, to remove it from the index so the working directory and the index are no longer synchronized—has already been given to you by Git. If you refer back to the output of `git status` shared just a couple paragraphs ago, you'll see Git telling you how to unstage the file. The command looks like this:

```
git reset HEAD file
```

In order to explain what's happening with this command, we first need to explain what *HEAD* is. HEAD is a pointer referencing the last commit you made (or the last commit you checked out into the working directory, but we haven't gotten to that point yet). Recall that when you stage a file (using `git add`), you are taking content from the working directory into the index. When you commit (using `git commit`), you are creating a point-in-time reference—a commit—to the content. Every time you commit, Git updates HEAD to point to the latest commit.

NOTE

HEAD also plays a strong role when you start working with multiple Git branches. We haven't discussed branches yet (they're covered later in this chapter in the section "Branching in Git"), but when we do get to branches we'll expand our discussion of HEAD at that time.

Here's a quick way to help illustrate this. If you've been following along with the examples we've been using in this chapter, then you can use these commands as well (just keep in mind that the SHA checksums shown here will differ from your own SHA checksums).

First, use `cat` to show the contents of *.git/HEAD*:

```
vagrant@jessie:~/net-auto$ cat .git/HEAD
ref: refs/heads/master
vagrant@jessie:~/net-auto$
```

You'll see that HEAD is a pointer to the file *refs/heads/master*. If you `cat` that file, you'll see this:

```
vagrant@jessie:~/net-auto$ cat .git/refs/heads/master
679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
vagrant@jessie:~/net-auto$
```

The contents of *.git/refs/heads/master* is a SHA checksum. Now run `git log`, and compare the SHA checksum of the latest commit against that value:

```
vagrant@jessie:~/net-auto$ git log
commit 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 20:41:19 2016 +0000
```

```
    Update sw1, add sw4

commit 95470631aba32d6823c80fdd3c6f923824dde470
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
vagrant@jessie:~/net-auto$
```

You'll note that the SHA checksum of the last commit matches the value of HEAD (which points to *refs/heads/master*), illustrating how HEAD is a pointer to the latest commit. Later in this chapter, we'll show how HEAD also incorporates branches and how it changes when you check out content to your working directory.

For now, though, let's get back to `git reset`. The `git reset` command is a powerful command, but fortunately it has some sane defaults. When used in this way—that is, without any flags and when given a filename or path—the only thing `git reset` will do is make the index look like the content referenced by HEAD (which we now know references a particular commit, by default the latest commit).

Recall that `git add` makes the index look like the working directory, which is how you stage a file. The `git reset HEAD` *`file`* command is the exact opposite, making the index look like the content referenced by HEAD. It *undoes* changes to the index made by `git add`, thus *unstaging* files.

Let's see it in action. You've already staged *sw5.txt* in preparation for committing it to the repository, so `git status` shows the file listed in the "Changes to be committed:" section. Now run `git reset`:

```
vagrant@trusty:~/net-auto$ git reset HEAD sw5.txt
vagrant@trusty:~/net-auto$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    sw5.txt

nothing added to commit but untracked files present (use "git
add" to track)
vagrant@trusty:~/net-auto$
```

You can see that *sw5.txt* is no longer listed as a change to be committed, and is instead shown as an untracked file (it's no longer in the index). Now you can continue working on the content of *sw5.txt*, committing a version of it to the repository when you're ready.

We've shown you how to create a repository, add files (both new and existing files), commit changes, and unstage files. What if you had files that need to be co-located with other files in the repository, but shouldn't be tracked by Git? This is where file exclusions come into play.

Excluding Files from a Repository

There may be occasions where there are files you need to store in the working directory—the "scratch space" for a Git repository—that you don't want included in the repository. Fortunately, Git provides a way to exclude certain files or filename patterns from inclusion in the repository.

Going back to our example, you've created a repository in which to store network automation artifacts. Let's suppose that you have a Python script that connects to your network switches in order to gather information from the switches. An example of one such Python script—in this case, one written to connect to an Arista switch and gather information—might look like this:

```
#!/usr/bin/env python



from pyeapi.client import Node as EOS

from pyeapi import connect

import yaml





def main():
    creds = yaml.load(open('credentials.yml'))

    un = creds['username']
    pwd = creds['password']

    conn = connect(host='eos-npab', username=un, password=pwd)
    device = EOS(conn)

    output = device.enable('show version')
    result = output[0]['result']

    print('Arista Switch Summary:')
    print('---------------------')
    print('OS Version:' + result['version'])
    print('Model:' + result['modelName'])
    print('System MAC:' + result['systemMacAddress'])

if __name__ == "__main__":
    main()
```

Part of how this script operates is via the use of authentication credentials stored in a separate file (in this case, a YAML file named *credentials.yml*). Now, you need these credentials to be stored with the Python script, but you wouldn't necessarily want the credentials to be tracked and managed by the repository.

TO INCLUDE OR EXCLUDE SECRETS?

Including secrets—information like passwords, SSH keys, or the like—into a Git repository will depend greatly on how the repository is being used. For a strictly private repository where per-user secrets are not needed, including secrets in the repository is probably fine. For repositories where per-user secrets should be used or for repositories that may at some point be shared publicly, you'll likely want to exclude secrets from the repository using the mechanisms outlined in this section.

Fortunately, Git provides a couple ways to exclude files from being tracked as part of a repository. Earlier in this chapter in the section "Committing Changes to a Repository", we discussed how Git configuration can be handled on a repository-specific, user-specific, or system-wide basis. Excluding files from Git repositories is similar, in that there are ways to exclude files on a per-repository basis or on a per-user basis.

EXCLUDING FILES PER-REPOSITORY

Let's start with the per-repository method. The most common way of excluding (or ignoring) files is to use a *.gitignore* file stored in the repository itself. Like any other content in the repository, the *.gitignore* file must be staged into the index and committed to the repository any time changes are made. The advantage of this approach is that the *.gitignore* file is then distributed as part of the repository, which is very useful when you are part of a team whose members are all using Git as a distributed version control system (DVCS).

The contents of the *.gitignore* file are simply a list of filenames or filename patterns, one on each line. To create your own list of files for Git to ignore, you'd simply create the file named *.gitignore* in the working directory, edit it to add the filenames or filename patterns you want ignored, and then add/commit it to the repository.

Looking at our Python script from earlier, you can see that it looks for its credentials in the file named *credentials.yml*. Let's create *.gitignore* (if you don't already have one) to ignore this file:

1. Create an empty file using `touch .gitignore`.
2. Edit *.gitignore*, using the text editor of your choice, to add *credentials.yml* on a single line in the file.

At this point, if you run `git status` you'll see that Git has noticed the addition of the *.gitignore* file, but the *credentials.yml* file is *not* listed:

```
vagrant@trusty:~/net-auto$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

nothing added to commit but untracked files present (use "git
add" to track)
vagrant@trusty:~/net-auto$
```

You can now stage and commit the *.gitignore* file into the repository using `git commit -am "Adding .gitignore file"`.

Now, if you create the *credentials.yml* file for the Python script, Git will politely ignore the file. For example, here you can see the file exists in the working directory, but `git status` reports no changes or untracked files:

```
[vagrant@centos net-auto]$ ls -la
total 40
drwxrwxr-x 3 vagrant vagrant 4096 May 31 16:32 .
drwxr-xr-x 5 vagrant vagrant 4096 May 12 17:18 ..
drwxrwxr-x 8 vagrant vagrant 4096 May 31 16:34 .git
-rw-rw-r-- 1 vagrant vagrant    8 May 31 16:27 .gitignore
-rw-rw-r-- 1 vagrant vagrant   15 May 31 16:32 credentials.yml
-rwxrwxr-x 1 vagrant vagrant    0 May 31 16:32 script.py
-rw-rw-r-- 1 vagrant vagrant   98 May 12 20:22 sw1.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw2.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw3.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 20:33 sw4.txt
-rw-rw-r-- 1 vagrant vagrant  135 May 31 14:56 sw5.txt
[vagrant@centos net-auto]$ git status
On branch master
nothing to commit, working directory clean
```

```
[vagrant@centos net-auto]$
```

Now, if you're really paying attention you might note that the fact Git reported nothing to commit isn't necessarily a guarantee the file has been ignored. Let's use a few more `git` commands to verify it. First, we'll use `git log` to show the history of commits:

```
vagrant@jessie:~/net-auto$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@jessie:~/net-auto$
```

Now, let's interrogate Git to see the contents of the repository at these various points in time. To do this, we'll use the `git ls-tree` command along with the SHA hash of the commit we'd like to inspect. You've probably noticed by now that Git often uses just the first seven characters of a SHA hash, like in the preceding output of the `git log --oneline` command (Git will automatically use more characters to keep the hashes unique as needed). In almost every case (there may be an exception out there somewhere!), that's true for commands you enter that require a SHA hash. For example, to see what was in the repository at the time of the next-to-last commit (whose SHA hash starts with `5cd13a8`), you could just do this:

```
vagrant@trusty:~/net-auto$ git ls-tree 5cd13a8
100755 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    script.py
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
100644 blob 88b23c7f60dc91f7d5bfeb094df9ed28996daeeb    sw5.txt
vagrant@trusty:~/net-auto$
```

You can see that *credentials.yml* does not exist in the repository as of this commit. What about the latest commit?

```
vagrant@jessie:~/net-auto$ git ls-tree ed45c95
100644 blob 2c1817fdecc27ccb3f7bce3f6bbad1896c9737fc
.gitignore
100755 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    script.py
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
```

```
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
100644 blob 88b23c7f60dc91f7d5bfeb094df9ed28996daeeb    sw5.txt
vagrant@jessie:~/net-auto$
```

(We'll leave it as an exercise for the reader to review the rest of the commits to verify that the *credentials.yml* file is *not* present in any commit.)

EXCLUDING FILES GLOBALLY

In addition to excluding files on a per-repository basis using a *.gitignore* file in the repository's working directory, you can also create a global file for excluding files for all repositories on your computer. Just create a *.gitignore_global* file in your home directory and add exclusions to that file. You may also want to run this command to ensure that Git is configured to use this new *.gitignore_global* file in your home directory:

```
git config --global core.excludesfile /path/to/.gitignore_global
```

If you placed *.gitignore_global* in your home directory, then the path to the file would typically be noted as *~/.gitignore_global*.

The use of the `git log` and `git ls-tree` commands naturally leads us into a discussion of how to view more information about a repository, its history, and its contents.

Viewing More Information About a Repository

When it comes to viewing more information about a repository, we've already shown you one command that you'll use quite a bit: the `git log` command. The `git log` command has already been used on a number of occasions, which should give you some indicator of just how useful it is.

VIEWING BASIC LOG INFORMATION

The most basic form of `git log` shows the history of commits up to HEAD, so just running `git log` will show you all the commits over the history of the repository. Here's the output of `git log` for the example repository we've been using in this chapter:

```
[vagrant@centos net-auto]$ git log
commit ed45c956da4b7e38b61b96ae050c4da77337f7ad
Author: John Smith <john.smith@networktocode.com>
```

```
Date:   Tue May 31 16:34:26 2016 +0000

    Adding .gitignore file

commit 5cd13a84de0e01f358636dee98da2df7d95e17ea
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 16:33:41 2016 +0000

    Add Python script to talk to network switches

commit 2a656c3288d5a324fba1c2cbfccbc0e29db73969
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 14:56:56 2016 +0000

    Add configuration for sw5

commit 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 20:41:19 2016 +0000

    Update sw1, add sw4

commit 95470631aba32d6823c80fdd3c6f923824dde470
Author: John Smith <john.smith@networktocode.com>
Date:   Thu May 12 17:37:22 2016 +0000

    First commit to new repository
[vagrant@centos net-auto]$
```

VIEWING BRIEF LOG INFORMATION

The `git log` command has a number of different options; there are too many for us to cover all of them here. One of the more useful options that we've already shown you, the `--oneline` option, would produce the following output for the same example repository:

```
[vagrant@centos net-auto]$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
[vagrant@centos net-auto]$
```

As you can see from the example output, the `--oneline` option abbreviates the SHA hash and lists only the commit message. For repositories with a lengthy history, it may be most helpful to start with `git log --oneline`, then drill into the details of a specific commit.

TIP

Disabling Git's default behavior to pipe output through a pager can make finding things via the use of `grep` possible. To disable Git's pager functionality, use the `--no-pager` option, as in `git --no-pager log --oneline`.

To drill into the details of a specific commit, there are a few different options. First, you can use the `git log` command and supply a range of commits to show. The syntax is **`git log`** ***`start SHA..end SHA`***. So, if we wanted to show more details on the last couple of commits in our example repository, we'd run a command that looks like this (if you're wondering where the SHA values came from, refer to the output of `git log --oneline` from earlier in this section, and recall that you only need to supply the first seven characters of the SHA hash):

```
vagrant@trusty:~/net-auto$ git log 5cd13a8..ed45c95
commit ed45c956da4b7e38b61b96ae050c4da77337f7ad
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 16:34:26 2016 +0000

    Adding .gitignore file
vagrant@trusty:~/net-auto$
```

Git also has some symbolic names that you can use in commands like `git log` (and others). We've already reviewed HEAD. If you wanted to use the commit just before HEAD, you'd reference that symbolically as `HEAD~1`. If you wanted to refer to the commit two places back from HEAD, you'd use `HEAD~2`; for three commits back, it's `HEAD~3`. (You can probably spot the pattern.) In this case, with this particular repository, this command produces the same results as the previous command we showed you:

```
vagrant@trusty:~/net-auto$ git log HEAD~1..HEAD
commit ed45c956da4b7e38b61b96ae050c4da77337f7ad
Author: John Smith <john.smith@networktocode.com>
Date:   Tue May 31 16:34:26 2016 +0000

    Adding .gitignore file
vagrant@trusty:~/net-auto$
```

When we expand our discussion of HEAD later in this chapter, you'll understand why we said "in this particular case, with this particular repository" that the two `git log` commands would produce the same output.

DRILLING INTO INFORMATION ON SPECIFIC COMMITS

Another way to drill into the details of a particular commit would be to use the `git cat-file` command. Git, like so many other UNIX/Linux tools, treats everything as a file. Thus, commits can be treated as a file, and their "contents" shown on the screen. This is what the `git cat-file` command does. So, taking the abbreviated SHA from a particular commit, you can look at more details about that commit with a command like this:

```
vagrant@jessie:~/net-auto$ git cat-file -p 2a656c3
tree 9f955969460fe47cb3b22d44e497c7a76c7a8db2
parent 679c41c13ceb5b658b988fb0dbe45a3f34f61bb3
author John Smith <john.smith@networktocode.com> 1464706616 +0000
committer John Smith <john.smith@networktocode.com> 1464706616
+0000

Add configuration for sw5
vagrant@jessie:~$
```

(The `-p` option to `git cat-file`, by the way, just does some formatting of the output based on the type of file. The man page for `git cat-file` will provide more details on this and other switches.)

You'll note this output contains a couple pieces of information that the default `git log` output doesn't show: the parent commit SHA and the tree object SHA. You can use the parent commit SHA to see this commit's "parent" commit. Every commit has a parent commit that lets you follow the chain of commits all the way back to the initial one, which is the only commit in a repository without a parent. The tree object SHA captures the files that are in the repository at the time of a given commit; we used this earlier with the `git ls-tree` command, like this:

```
vagrant@jessie:~/net-auto$ git ls-tree 9f9559
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
100644 blob 88b23c7f60dc91f7d5bfeb094df9ed28996daeeb    sw5.txt
vagrant@jessie:~/net-auto$
```

Using the SHA checksums listed here, you could then use the `git cat-file` command to view the contents of one of these files at that particular time (as of that particular commit).

Let's see how that works. In the following set of commands, we'll first use `git log --oneline` to show the history of commits to a repository. Then we'll use `git cat-file` and `git ls-tree` with the appropriate seven-character SHA hashes to display the contents of a particular file at two different points in time (as of two different commits).

```
vagrant@trusty:~/net-auto$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@trusty:~/net-auto$ git cat-file -p 9547063
tree fdad0ff90745deb944a430e2151d085aebc68d00
author John Smith <john.smith@networktocode.com> 1463074642 +0000
committer John Smith <john.smith@networktocode.com> 1463074642
+0000

First commit to new repository
vagrant@trusty:~/net-auto$ git ls-tree fdad0f
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
vagrant@trusty:~/net-auto$ git cat-file -p 02df3d
interface ethernet0

interface ethernet1

interface ethernet2

interface ethernet3

vagrant@trusty:~/net-auto$
```

This shows us the contents of *sw1.txt* as of the initial commit. Now, let's repeat the same process for the second commit:

```
vagrant@trusty:~/net-auto$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
```

```
9547063 First commit to new repository
vagrant@trusty:~/net-auto$ git cat-file -p 679c41c
tree a093d5f26677d345cb274ceb826e70bdb31ffd6f
parent 95470631aba32d6823c80fdd3c6f923824dde470
author John Smith <john.smith@networktocode.com> 1463085679 +0000
committer John Smith <john.smith@networktocode.com> 1463085679
+0000

Update sw1, add sw4
vagrant@trusty:~/net-auto$ git ls-tree a093d5
100644 blob 2567e072ca607963292d73e3acd49a5388305c53    sw1.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw2.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw3.txt
100644 blob 02df3d404d59d72c98439f44df673c6038352a27    sw4.txt
vagrant@trusty:~/net-auto$ git cat-file -p 2567e0
interface ethernet0
  duplex auto

interface ethernet1

interface ethernet2

interface ethernet3

vagrant@trusty:~/net-auto$
```

Ah, note that the contents of the *sw1.txt* file have changed! However, this is a bit laborious—wouldn't it be nice if there were an easier way to show the differences between two versions of a file within a repository? This is where the `git diff` command comes in handy.

Distilling Differences Between Versions of Files

We mentioned at the start of this chapter that one of the benefits of using version control for network automation artifacts (switch configurations, Python scripts, Jinja templates, etc.) is being able to see the differences between versions of files over time. In the previous section, we showed you a very manual method of doing so; now we're going to show you the easy way: the `git diff` command.

NOTE

Note that Git also supports integration with third-party diff tools, including graphical diff tools. In such cases, you would use `git difftool` instead of `git diff`.

EXAMINING DIFFERENCES BETWEEN COMMITS

The `git diff` command shows the differences between versions of a file (the differences between a file at two different points in time). You just need to supply the two commits and the file to be compared. Here's an example. First, we list the history using `git log`, then we use `git diff` to compare two versions of a file.

```
[vagrant@centos net-auto]$ git log --oneline
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
[vagrant@centos net-auto]$ git diff 9547063..679c41c sw1.txt
diff --git a/sw1.txt b/sw1.txt
index 02df3d4..2567e07 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,4 +1,5 @@
 interface ethernet0
+  duplex auto

 interface ethernet1

[vagrant@centos net-auto]$
```

The format in which `git diff` shows the differences between the files can be a bit confusing at first. The key in deciphering the output lies in the lines just after the `index...` line. There, `git diff` tells us that dashes will be used to represent file *a* (`--- a/sw1.txt`), and pluses will be used to represent file *b* (`+++ b/sw1.txt`). Following that is the representation of the differences in the files—lines that exist in file *a* are preceded by a dash, while lines that exist in file *b* are preceded by a plus. Lines that are the same in both files are preceded by a space.

Thus, in this example, we can see that the later commit, represented by the hash `679c41c`, the line `duplex auto` was added. Obviously, this is a very simple example, but hopefully you can begin to see just how useful this is.

OMITTING THE FILENAME IN COMMANDS

If you omit the filename with the `git diff` command (for example, if you entered `git diff` *`start SHA..end SHA`*), then Git will show a diff for *all* the files changed in that commit, rather than just a specific file referenced on the command line. Adding the filename to the `git diff` command allows you to focus on the changes a specific file.

VIEWING OTHER TYPES OF DIFFERENCES

Let's make some changes to the configuration file for *sw1.txt* so that the diff is a bit more complex, and along the way we'll also show how you can use `git diff` in other ways.

First, we'll make some changes to *sw1.txt* using the text editor of your choice. It doesn't really matter what the changes are; we'll run `git status` to provide that changes exist in the working directory. However, *before* you stage the changes, let's see if we can use `git diff` again.

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

    modified:   sw1.txt

no changes added to commit (use "git add" and/or "git commit -a")
vagrant@trusty:~/net-auto$ git diff
diff --git a/sw1.txt b/sw1.txt
index 2567e07..7005dc6 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,9 +1,11 @@
 interface ethernet0
-  duplex auto
+  switchport mode access vlan 101

 interface ethernet1
+  switchport mode trunk

 interface ethernet2
+  switchport mode access vlan 102

 interface ethernet3
-
+  switchport mode trunk
vagrant@trusty:~/net-auto$
```

Running `git diff` like this—without any parameters or options—shows you the differences between your working tree and the index. That is, it shows you the changes that have not yet been staged for the next commit.

Now, let’s stage the changes in preparation for the next commit, then see if there’s another way to use `git diff`:

```
vagrant@jessie:~/net-auto$ git add sw1.txt
vagrant@jessie:~/net-auto$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   sw1.txt

vagrant@jessie:~/net-auto$ git diff
vagrant@jessie:~/net-auto$ git diff --cached
diff --git a/sw1.txt b/sw1.txt
index 2567e07..f3b5ad5 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,9 +1,11 @@
 interface ethernet0
-  duplex auto
+  switchport mode access vlan 101

 interface ethernet1
+  switchport mode trunk

 interface ethernet2
+  switchport mode access vlan 102

 interface ethernet3
-
+  switchport mode trunk
vagrant@jessie:~/net-auto$
```

You can see that the first `git diff` command returned no results, which makes sense—there are no changes that *aren’t* staged for the next commit. However, when we add the `--cached` parameter, it tells `git diff` to show the differences between the index and HEAD. In other words, this form of `git diff` shows the differences between the index and the last commit.

Once we finally commit this last set of changes, we can circle back around to our original use of `git diff`, which allows us to see the changes between two arbitrary commits.

```
[vagrant@centos net-auto]$ git commit -m "Defined VLANs on sw1"
[master 3588c31] Defined VLANs on sw1
 1 file changed, 4 insertions(+), 2 deletions(-)
[vagrant@centos net-auto]$ git status
On branch master
nothing to commit, working directory clean
[vagrant@centos net-auto]$ git log --oneline
3588c31 Defined VLANs on sw1
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
[vagrant@centos net-auto]$ git diff 679c41c..3588c31 sw1.txt
diff --git a/sw1.txt b/sw1.txt
index 2567e07..f3b5ad5 100644
--- a/sw1.txt
+++ b/sw1.txt
@@ -1,9 +1,11 @@
 interface ethernet0
-  duplex auto
+  switchport mode access vlan 101

 interface ethernet1
+  switchport mode trunk

 interface ethernet2
+  switchport mode access vlan 102

 interface ethernet3
-
+  switchport mode trunk
[vagrant@centos net-auto]$
```

Before we move on to our next topic—we'll discuss branches in Git next—let's take a moment to review what you've done so far:

- Staged changes (using `git add`) and committed them to the repository (using `git commit`)
- Modified the configuration of Git (using `git config`)
- Unstaged changes that weren't yet ready to be committed (using `git reset`)
- Excluded files from inclusion in the repository (using `.gitignore`)
- Reviewed the history of the repository (using `git log`)
- Compared different versions of files within the repository to see the changes in each version (using `git diff`)

In the next section, we'll expand our discussion of Git to cover what is, arguably, one of Git's most powerful features: branching.

Branching in Git

If you refer back to "Brief History of Git", you'll see that one of the primary design goals for Git was strong support for nonlinear development. That's a fancy way of saying that Git needed to support multiple developers working on the same thing at the same time. So how is this accomplished? Git does it through the use of *branches*.

A *branch* in Git is a pointer to a commit. Now, that might not sound too powerful, so let's use some figures to help better explain the concept of a branch, and why non-linear development in Git can be powerful.

First, recall from the section "Overview of Git's Architecture" that Git uses a series of objects: blobs (representing the content of files in the repository), trees (representing the file and directory structure of the repository), and commits (representing a point-in-time snapshot of the repository, its structure, and its content). You can visualize this as shown in Figure 8-1.

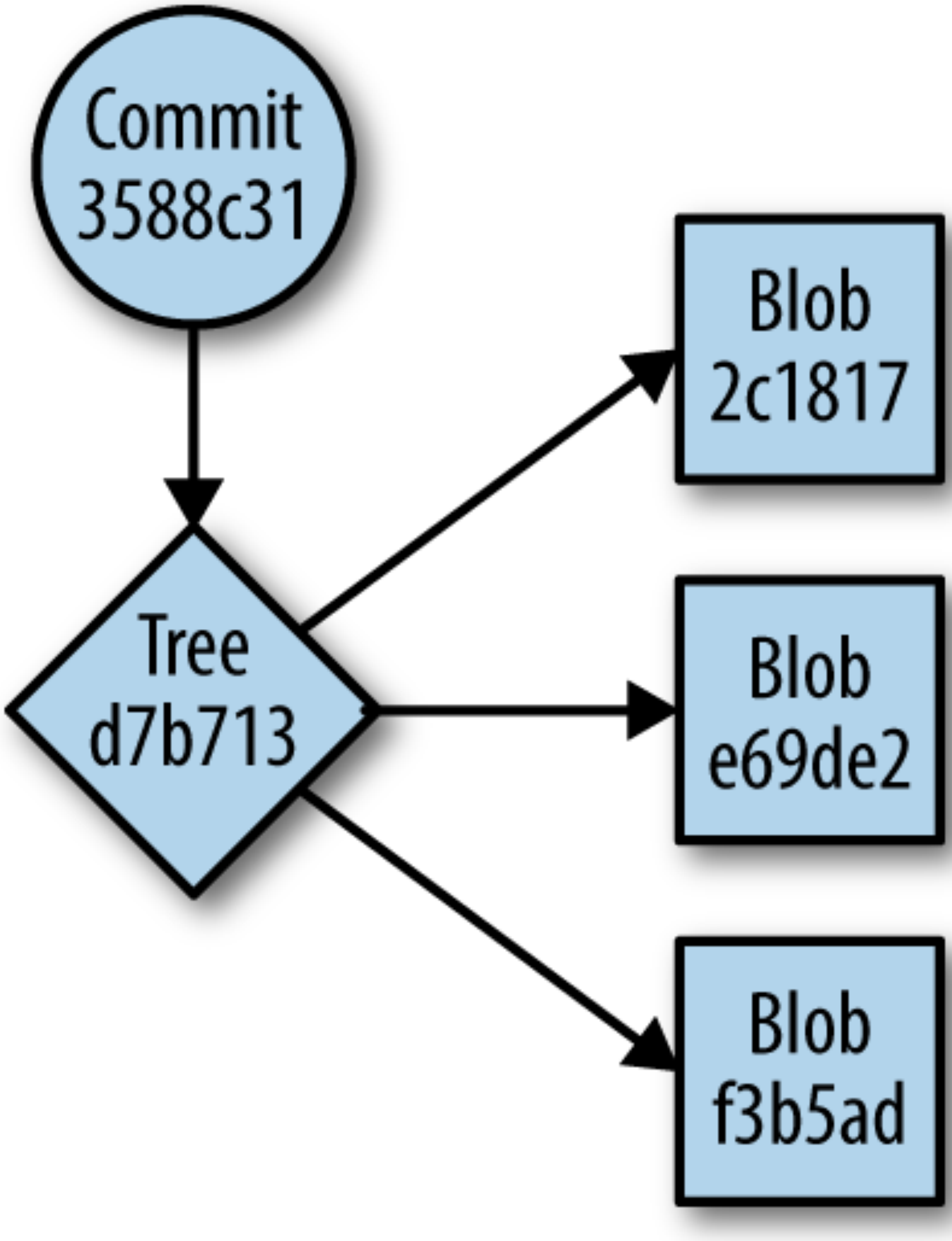

Figure 8-1. Objects in a Git repository

Each of these objects is identified by the SHA-1 hash of its contents. You've seen how commits are referenced via their SHA-1 hash, and you've seen how to use the `git ls-tree` or `git cat-file` commands to see the contents of tree and blob objects, respectively, by referencing their SHA-1 hash.

As you make changes and commit them to the repository, you create more commit objects (more snapshots), each of which points back at the previous commit (referred to as its "parent" commit; you saw this in the section "Viewing More Information About a Repository"). After a few commits, you can visualize it like Figure 8-2 (we've omitted the blobs to simplify the diagram).

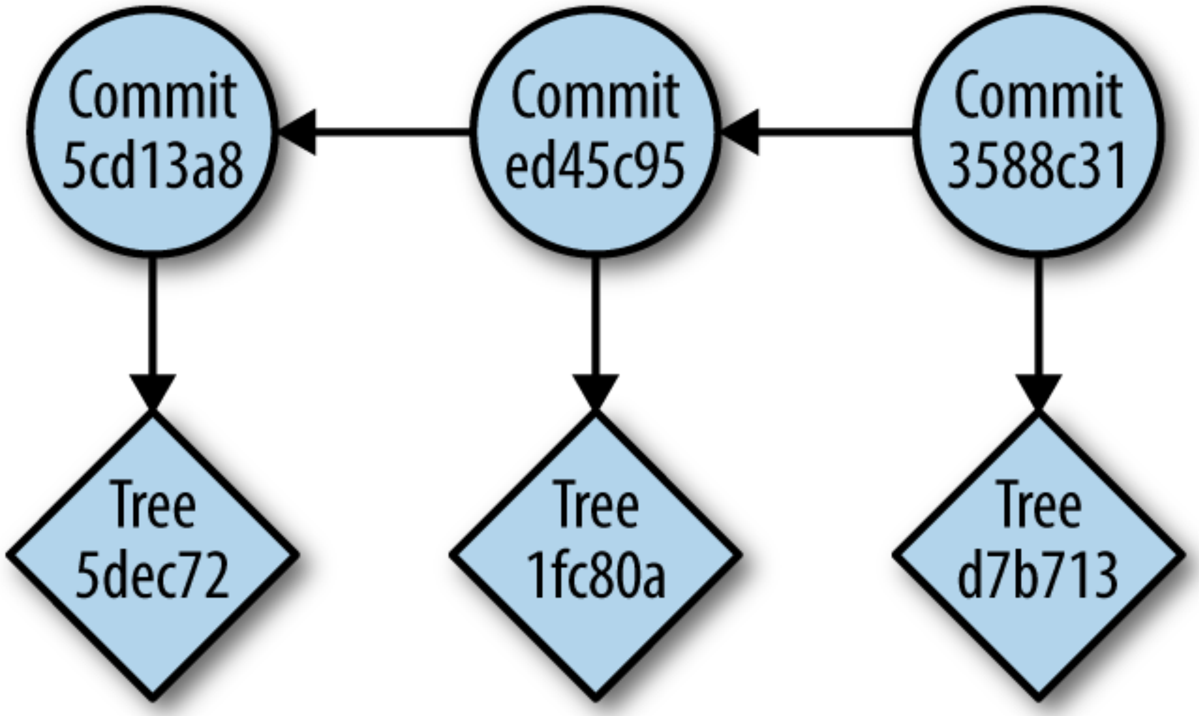

Figure 8-2. A chain of commits in a Git repository

Each commit points to a tree object, and each tree object points to blobs that represent the contents of the repository at the time of the commit. Using the reference to the tree object and the associated blobs, you can re-create the state of the repository at any given commit—hence why we refer to commits as point-in-time snapshots of the repository.

This is all well and good—and helps to explain Git's architecture a bit more fully—but what does it have to do with branching in Git? To answer that question (and we *will* answer it, we promise!), we need to revisit the concept of HEAD. In the section "Unstaging Files", we defined HEAD as a pointer that points to the latest commit, or to the commit we've checked out into the working directory. You visualize HEAD as something like Figure 8-3.

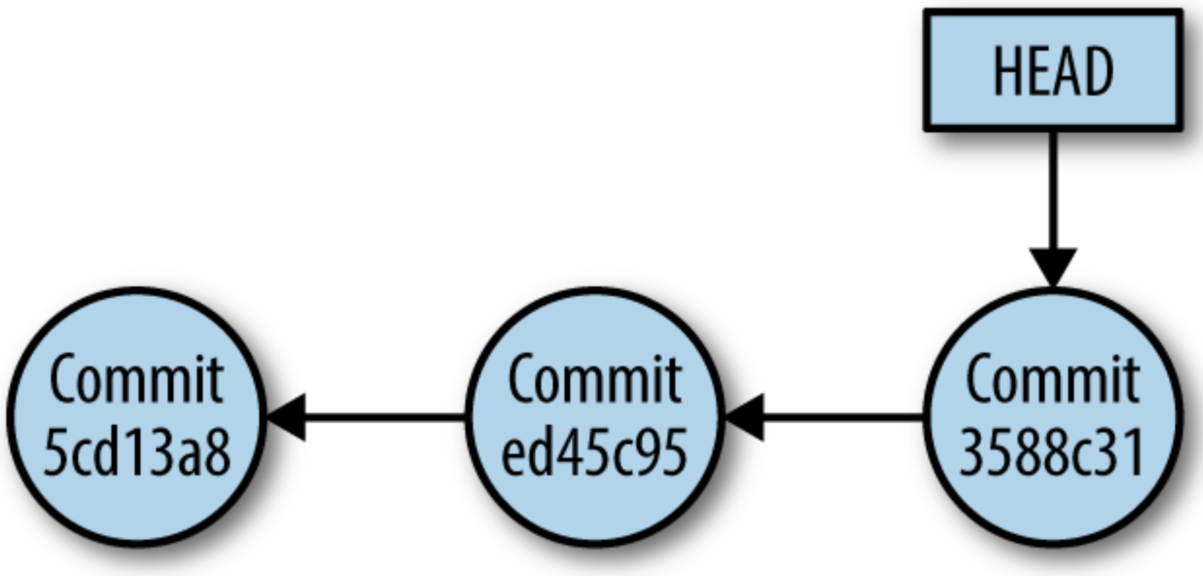

Figure 8-3. HEAD pointing to the latest commit

You can verify this using a procedure we outlined earlier in this chapter (this assumes you haven't checked out a different branch or different commit, something we'll discuss shortly):

1. From the repository's working directory, run **`cat .git/refs/heads/master`**. Note the value displayed.
2. Compare the value of the previous command to the value of the last commit from the output of `git log --oneline`. You should see the same value in both places, indicating that HEAD points to the latest commit.

By default, *every* Git repository starts out with a single branch, named *master*. As a branch is just a reference to a commit, this is illustrated graphically in Figure 8-4.

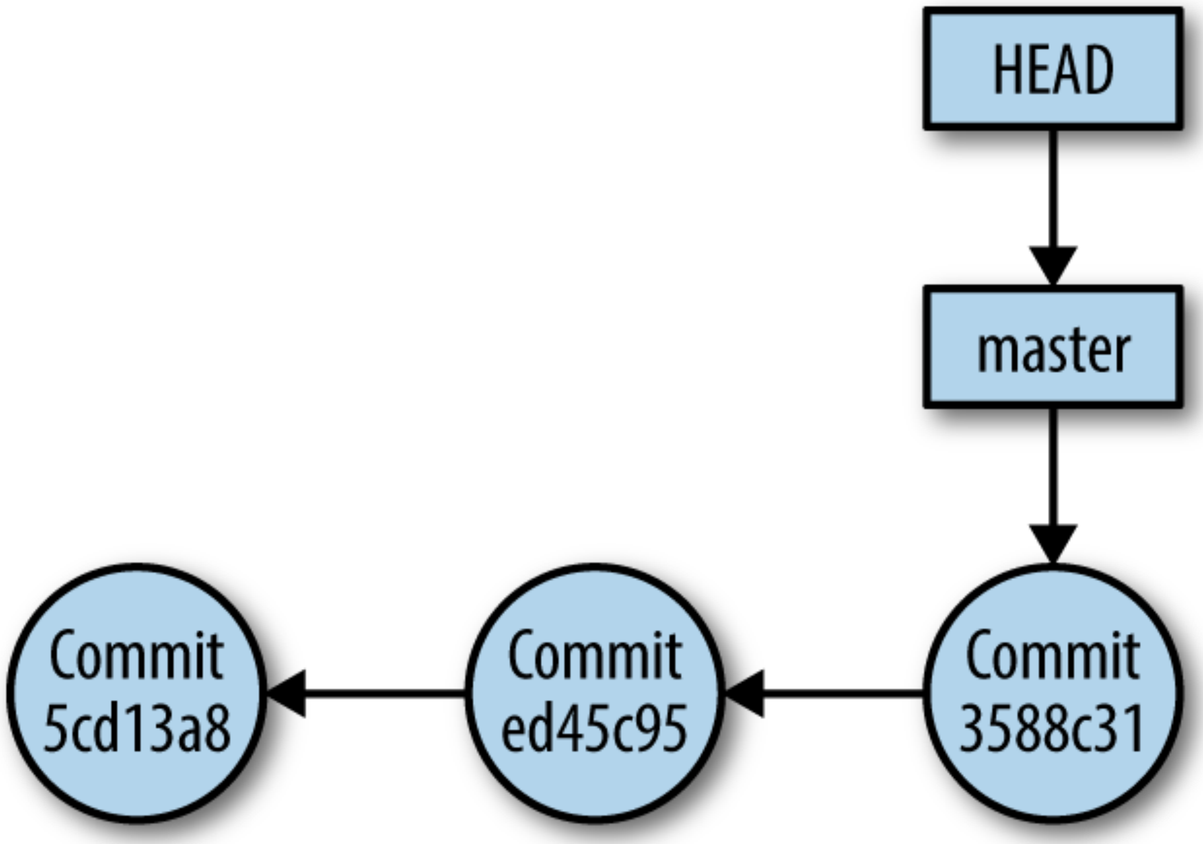

Figure 8-4. HEAD pointing to the latest commit in the master branch

You can see that the branch reference points to a commit, and that HEAD points to the branch reference.

However, you're not limited to only a single branch in a Git repository. In fact, because branches are so lightweight (a reference to a commit), you're strongly encouraged to use multiple branches. So, when you create a new branch—let's call this new branch *testing*, though the name doesn't really matter—the organization of the Git objects now looks something like Figure 8-5.

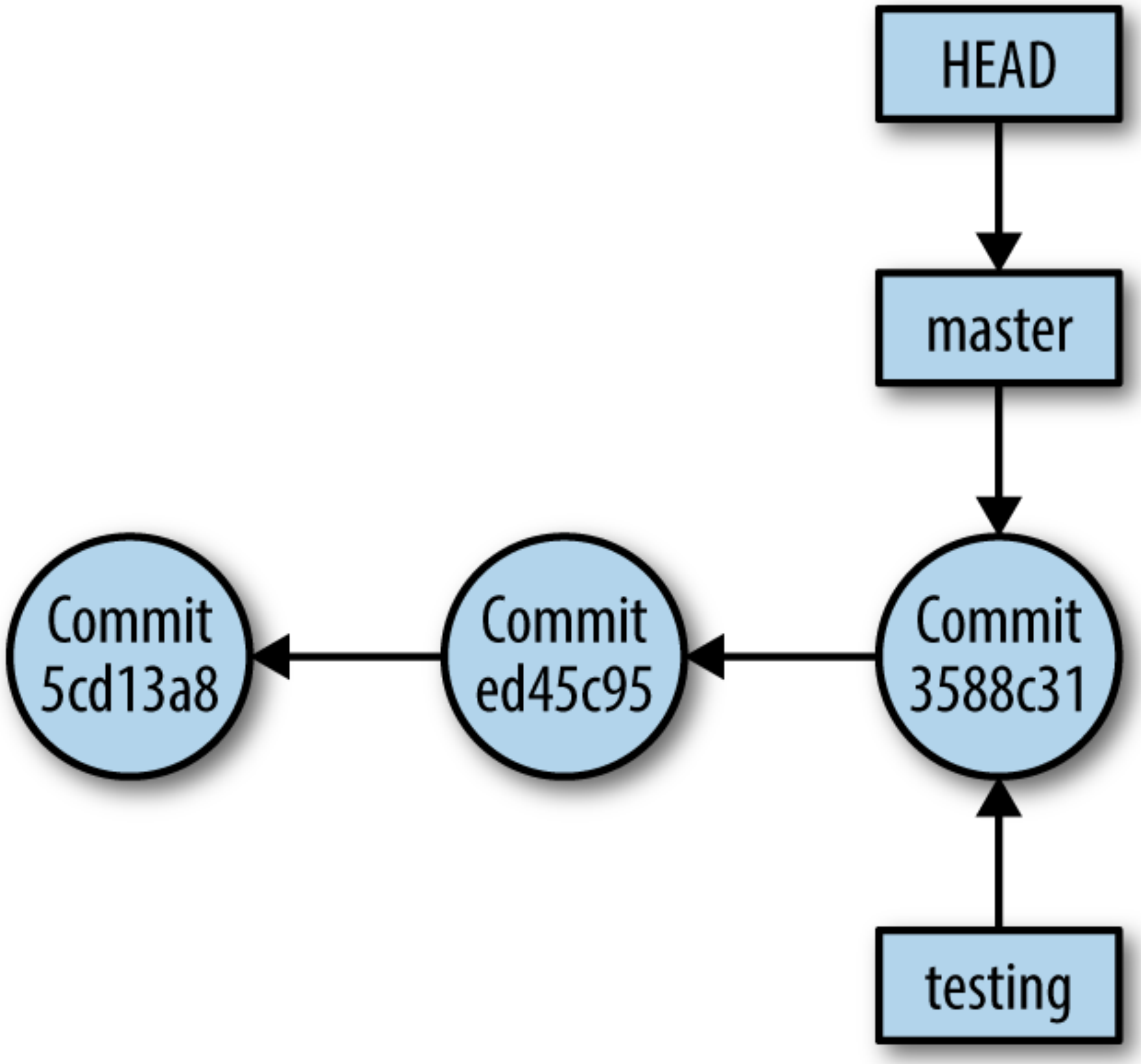

Figure 8-5. New branch created in a Git repository

So, you've created the new branch—we'll show you the commands to do that shortly—and the new branch now references a particular commit. However, HEAD hasn't moved. HEAD gets moved when you check out content into the repository, so in order to move HEAD to the new branch, you first have to *check out* the new branch. Similarly, if you wanted to work with the repository at an earlier point in time (at an earlier commit) you'd need to check out that particular commit. Once you check out a branch, HEAD now points to the new branch, as in Figure 8-6.

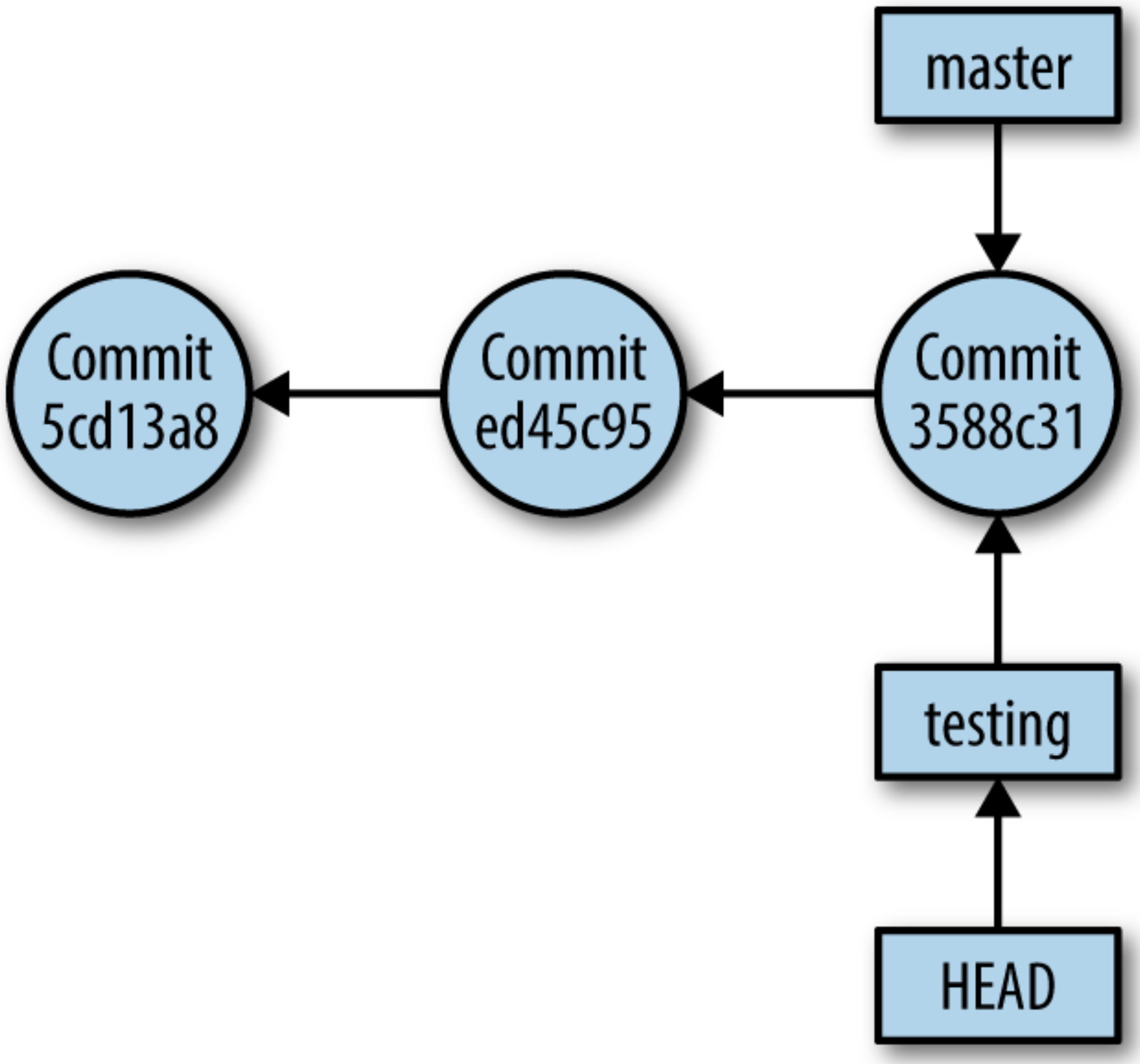

Figure 8-6. HEAD pointing to a checked-out branch

At this point, you can now start making changes and committing them to the repository. This is where branches start to really show their power: they *isolate new changes from the master branch*. Let's assume you've made some changes to the testing branch and have committed those changes to the repository. The graphical view of the objects and relationships inside the repository now looks like Figure 8-7.

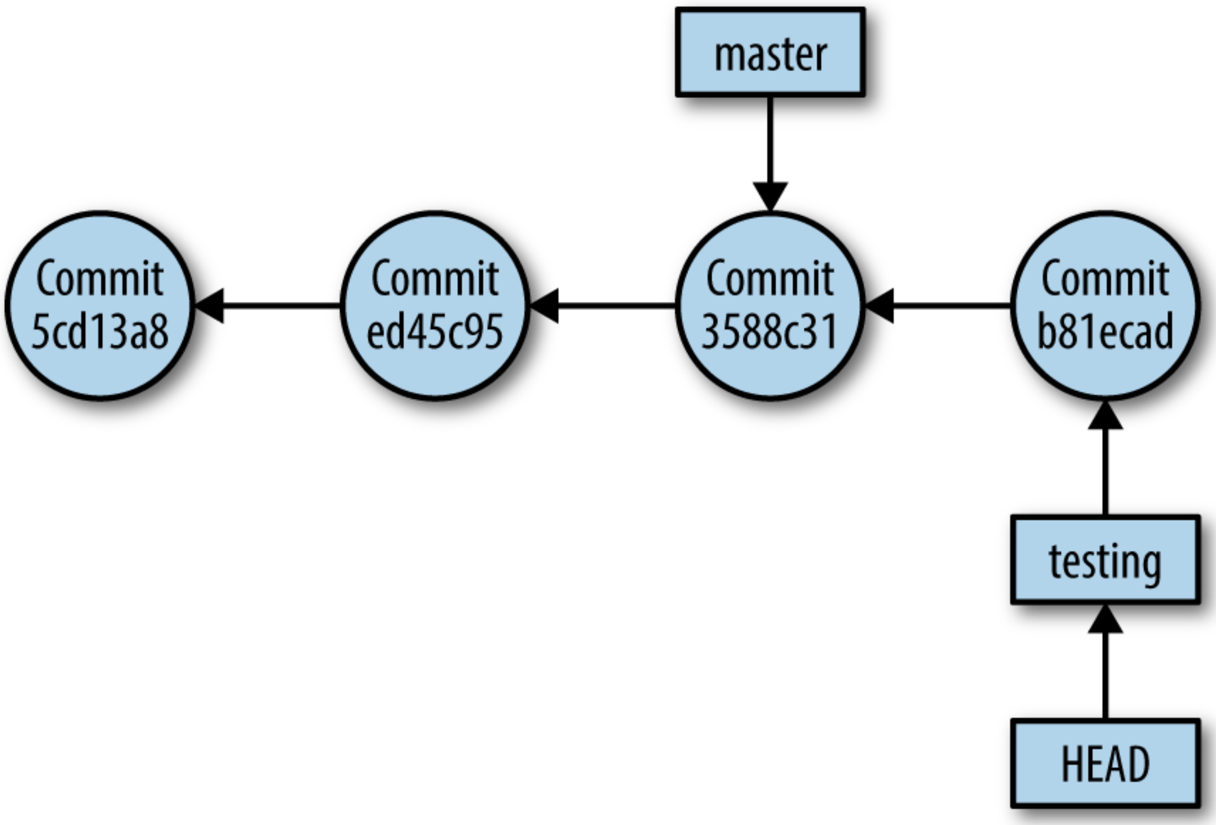

Figure 8-7. Adding a commit to a branch

You'll note that the testing branch—and HEAD—move forward to represent the latest commit, but the master branch remains *untouched*. At any point, you can check out the master branch and be right back where you were before you created the new branch and made the changes. This diagram shows an example of how multiple branches can evolve over time and allow for the development of hotfixes, new features, and new releases without affecting the master branch.

Figure 8-8 is a complicated example of branches, but it gives an idea of how branches *might* be used in a typical software development environment.

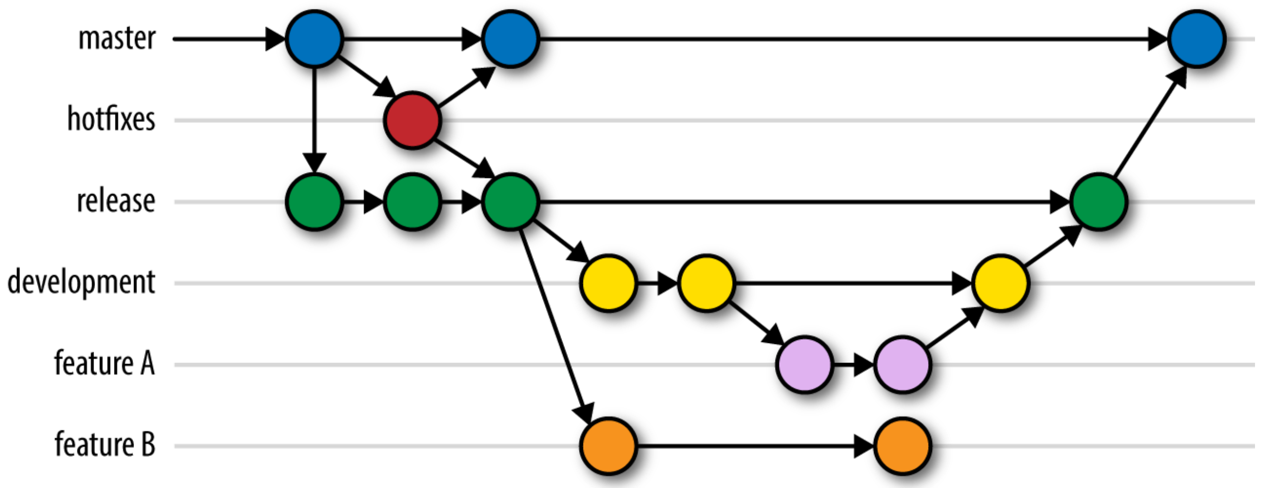

Figure 8-8. Multiple branches in a development cycle

Hopefully, the wheels in your head are turning and you're starting to think about the possibilities that branches create:

- You can create a new branch when you want to try something new or different, without affecting what's in the master branch. If it doesn't work out, no big deal—the content in the master branch remains untouched.
- Branches form the basis by which you can collaborate with other authors on the same repository. If you're working in branch A and your coworker is working in branch B, then you're assured that you won't affect each other's changes. (Now, you might have some issues when it comes time to bring the changes from the two branches together when you *merge*, but that's a different story—one we'll tackle later in this chapter in the section "Merging and Deleting Branches".)

Let's turn our attention now to the practical side of working with Git branches, where we can see the theory we've been describing in practice.

Creating a Branch

To create a Git branch—which, again, is just a reference to a commit—you'll use the `git branch` command. So, to create the testing branch we discussed in the previous section, you'd simply run `git branch testing`. The command doesn't produce any output, but there *is* a way you can verify that it actually did something.

First, look in the *.git/refs/heads* directory, and you'll see a new entry there named after your newly created branch. If you run `cat` on that new file, you'll see that it points to the commit referenced by HEAD when you created the branch. Let's see that in action:

```
vagrant@trusty:~/net-auto$ git branch testing
vagrant@trusty:~/net-auto$ ls -la .git/refs/heads
total 16
drwxrwxr-x 2 vagrant vagrant 4096 Jun  4 19:16 .
drwxrwxr-x 4 vagrant vagrant 4096 May 12 14:52 ..
-rw-rw-r-- 1 vagrant vagrant   41 Jun  4 19:16 master
-rw-rw-r-- 1 vagrant vagrant   41 Jun  4 19:13 testing
vagrant@trusty:~/net-auto$ cat .git/refs/heads/testing
3588c31cbf958fe1a28d5e1f19ace669de99bb8c
vagrant@trusty:~/net-auto$ git log --oneline
3588c31 Defined VLANs on sw1
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@trusty:~/net-auto$
```

The presence of the *testing* file in *.git/refs/heads*, along with the content of the file referencing the latest commit as of the time of creation, shows that the branch has been created. You can also verify this by simply running `git branch`, which will output the list of branches. The active branch—the branch that is *checked out* for use in the working directory—will have an asterisk before it (and, if colors are enabled in your terminal, may be listed in a different color). This shows you that your testing branch has been created, but not checked out—the master branch is still active.

To switch the active branch, you must first check out the branch.

Checking Out a Branch

To *check out* a branch means to make it the active branch, and the branch that will be available in the working directory for you to edit/modify. To check out a branch, use the `git checkout` command, supplying the name of the branch you'd like to check out:

```
vagrant@jessie:~/net-auto$ git branch
* master
  testing
```

```
vagrant@jessie:~/net-auto$ git checkout testing
Switched to branch 'testing'
vagrant@jessie:~/net-auto$
```

TIP

You can create a branch and check it out at the same time using `git checkout -b <branch name>`.

Let's make a simple change to the repository—say, let's add a file—then switch back to the "master" branch to see how Git handles this. First, we'll stage *sw6.txt* to the repository and commit it to the testing branch:

```
[vagrant@centos net-auto]$ git add sw6.txt
[vagrant@centos net-auto]$ git commit -m "Add sw6 configuration"
[testing b45a2b1] Add sw6 configuration
 1 file changed, 12 insertions(+)
 create mode 100644 sw6.txt
[vagrant@centos net-auto]$
```

Note that the response from Git when you commit the change includes the branch name and the SHA hash of the commit (`[testing b45a2b1]`). A quick `git log --oneline` will verify that the latest commit has the same hash as reported by the `git commit` command. Likewise, a quick `cat .git/HEAD` will show that you're on the testing branch (because it points to *.git/refs/heads/testing*), and `cat .git/refs/heads/testing` will also show the latest commit SHA hash. This shows that HEAD points to the latest commit in the checked-out branch.

VIEWING GIT BRANCH INFORMATION IN THE PROMPT

When you're working with multiple branches in a Git repository, it can sometimes be challenging to know *which* branch is currently active (checked out). To help address this, most distributions of Git since version 1.8 have included support to allow bash—the shell most Linux distributions use by default—to display the currently active Git branch in the bash prompt. On Ubuntu 14.04 and Debian 8.x, this file is named *git-sh-prompt* and is found in the */usr/lib/git-core* directory. On CentOS 7 (and presumably on RHEL 7), the file is named *git-prompt.sh* and is found in the */usr/share/git-core/contrib/completion* directory. On macOS, the file is installed as part of the XCode command-line tools, is named *git-prompt.sh*, and is found in the */Library/Developer/CommandLineTools/usr/share/git-core* directory. The instructions for using this functionality are found at the top of the appropriate file for your OS.

Now, let's switch back to the master branch and see what the working directory looks like:

```
vagrant@trusty:~/net-auto$ git checkout master
Switched to branch 'master'
vagrant@trusty:~/net-auto$ ls -la
total 40
drwxrwxr-x 3 vagrant vagrant 4096 Jun  7 16:49 .
drwxr-xr-x 5 vagrant vagrant 4096 May 12 17:18 ..
drwxrwxr-x 8 vagrant vagrant 4096 Jun  7 16:49 .git
-rw-rw-r-- 1 vagrant vagrant    8 May 31 16:27 .gitignore
-rw-rw-r-- 1 vagrant vagrant   15 May 31 16:32 pw.txt
-rwxrwxr-x 1 vagrant vagrant    0 Jun  7 16:34 script.py
-rw-rw-r-- 1 vagrant vagrant  200 Jun  3 20:47 sw1.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw2.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw3.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 20:33 sw4.txt
-rw-rw-r-- 1 vagrant vagrant  135 May 31 14:56 sw5.txt
vagrant@trusty:~/net-auto$
```

Wait—the *sw6.txt* file is *gone!* What happened? Not to worry, you haven't lost anything. Recall that checking out a branch makes it the active branch, and therefore the branch that will be present in the working directory for you to modify. The *sw6.txt* file isn't in the master branch, it's in the testing branch, so when you switched to master using `git checkout master`, that file was removed from the working directory. Recall also that the working directory *isn't* the same as the repository—even though the file has been removed from the working directory, it's *still* in the repository, as you can easily verify:

```
vagrant@jessie:~/net-auto$ git checkout testing
Switched to branch 'testing'
vagrant@jessie:~/net-auto$ ls -la
total 44
drwxrwxr-x 3 vagrant vagrant 4096 Jun  7 16:53 .
drwxr-xr-x 5 vagrant vagrant 4096 May 12 17:18 ..
drwxrwxr-x 8 vagrant vagrant 4096 Jun  7 16:53 .git
-rw-rw-r-- 1 vagrant vagrant    8 May 31 16:27 .gitignore
-rw-rw-r-- 1 vagrant vagrant   15 May 31 16:32 pw.txt
-rwxrwxr-x 1 vagrant vagrant    0 Jun  7 16:34 script.py
-rw-rw-r-- 1 vagrant vagrant  200 Jun  3 20:47 sw1.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw2.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 17:17 sw3.txt
-rw-rw-r-- 1 vagrant vagrant   84 May 12 20:33 sw4.txt
-rw-rw-r-- 1 vagrant vagrant  135 May 31 14:56 sw5.txt
-rw-rw-r-- 1 vagrant vagrant  221 Jun  7 16:53 sw6.txt
vagrant@jessie:~/net-auto$
```

This illustrates how branches help isolate changes from the master branch, a key benefit of using branches. Using a branch, you can make some changes, test those changes, and then discard them if necessary—all while knowing that your master branch remains safe and untouched.

However, what if the changes you made in a branch were changes you wanted to keep? Perhaps you tried out a new Jinja template, it worked perfectly in your testing, and now you'd like to make that a permanent part of your repository? This is where *merging* branches comes into play.

Merging and Deleting Branches

Before we get into merging branches, let's revisit the contents of a commit object in Git. In our example repository, we'll examine the contents of the latest commit object for the testing branch:

```
vagrant@jessie:~/net-auto$ git checkout testing
Switched to branch 'testing'
vagrant@jessie:~/net-auto$ git cat-file -p b45a2b1
tree f6b5dfbfdbf6bc29f04300c9a82c6936397d9b27
parent 3588c31cbf958fe1a28d5e1f19ace669de99bb8c
author John Smith <john.smith@networktocode.com> 1465317796 +0000
committer John Smith <john.smith@networktocode.com> 1465317796
+0000

Add sw6 configuration
vagrant@jessie:~/net-auto$
```

What does this tell us?

1. This particular commit references the tree object with the hash `f6b5df`.
2. The author and committer of this commit is John Smith.
3. The commit message indicates that this commit captures the addition of the configuration for sw6.
4. Finally, note that this commit has a parent commit with a hash of `3588c31`.

We've mentioned before that every commit (except the very first commit) has a pointer to a parent commit. This is illustrated in Figures 8-2 through 8-7, where the commit objects point "backward in time" to the previous commit.

When we merge branches, Git is going to create a new commit object—called a *merge commit* object—that will actually have *two* parents. Each of these parents represents the two branches that were brought together as part of the merge process. In so doing, Git maintains the link back to previous commits so that you can always "roll back" to previous versions.

At a high level, the merge process looks like this:

1. Switch to the branch into which the other branch should be merged. If you're merging back into master, check out (switch to) master.
2. Run the `git merge` command, specifying the name of the branch to be merged into master.
3. Supply a message (a commit message for the merge commit) describing the changes being merged.

REVIEWING FAST-FORWARD MERGES

Let's see this in action. Let's take the testing branch, which has a new switch configuration (*sw6.txt*) that isn't present in the master branch, and merge it back into master.

First, let's ensure you are on the master branch:

```
[vagrant@centos net-auto]$ git branch
  master
* testing
[vagrant@centos net-auto]$ git checkout master
Switched to branch 'master'
[vagrant@centos net-auto]$
```

Next, let's actually merge the testing branch into the master branch:

```
[vagrant@centos net-auto]$ git merge testing
Updating 3588c31..b45a2b1
Fast-forward
 sw6.txt | 12 ++++++++++++
 1 file changed, 12 insertions(+)
 create mode 100644 sw6.txt
[vagrant@centos net-auto]$
```

Note the "Fast-forward" in the response from Git; this indicates that it was possible to merge the branches by simply replaying the same set of changes to the master branch as was performed on the branch being merged. In situations like this—a simple merge—you won't see an additional merge commit:

```
vagrant@trusty:~/net-auto$ git log --oneline
b45a2b1 Add sw6 configuration
3588c31 Defined VLANs on sw1
ed45c95 Adding .gitignore file
5cd13a8 Add Python script to talk to network switches
2a656c3 Add configuration for sw5
679c41c Update sw1, add sw4
9547063 First commit to new repository
vagrant@trusty:~/net-auto$ ls -la sw6.txt
-rw-rw-r-- 1 vagrant vagrant 221 Jun  7 20:53 sw6.txt
vagrant@trusty:~/net-auto$
```

DELETING A BRANCH

Once a branch (and its changes) have been merged, you can delete the branch using **`git branch -d`** ***`branch-name`***. Note that you generally don't want to delete a branch before it's been merged; otherwise, you'll lose the changes stored in that branch (Git will prompt you if you try to delete a branch that hasn't been merged). Once a branch has been merged, though, its changes are safely stored in another branch (typically the master branch, but not always), and it's therefore now safe to delete.

REVIEWING MERGES WITH A MERGE COMMIT

Now, let's look at a more complex example. First, let's create a new branch to store some changes we'll make relative to the configuration for sw4. To do that, we'll simply run `git checkout -b sw4`. This creates the new branch *and* checks it out so it's the active branch. Once you've made some changes to *sw4.txt*, use `git add` and `git commit` to stage and commit the changes.

Next, let's switch back to master (using `git checkout master`) and make some changes to a *different* switch configuration. Stage and commit the changes to the master branch. Now what happens when we try to merge the sw4 branch into master?

Before we answer that question, let's explore the commit objects a bit. Here are the contents of the last commit object in the sw4 branch:

```
vagrant@jessie:~/net-auto$ git checkout sw4
```

```
Switched to branch 'sw4'
vagrant@jessie:~/net-auto$ git log --oneline HEAD~2..HEAD
40e88b8 Add port descriptions for sw4
b45a2b1 Add sw6 configuration
vagrant@jessie:~/net-auto$ git cat-file -p 40e8b8
tree 845b53f6715d73c36d90b3fe3224bfb494853ba5
parent b45a2b162334376f2100974687742de1a23c2594
author John Smith <john.smith@networktocode.com> 1465333657 +0000
committer John Smith <john.smith@networktocode.com> 1465333657
+0000

Add port descriptions for sw4
vagrant@jessie:~/net-auto$
```

The `git log --oneline HEAD~2..HEAD` command just shows the last two commits leading up to HEAD (which points to the last commit on the active branch). As you can see, this commit object points to a parent commit of `b45a2b1`.

Here's the last commit on the master branch:

```
vagrant@jessie:~/net-auto$ git checkout master
Switched to branch 'master'
vagrant@jessie:~/net-auto$ git log --oneline HEAD~2..HEAD
183b8fe Fix sw3 configuration for hypervisor
b45a2b1 Add sw6 configuration
vagrant@jessie:~/net-auto$ git cat-file -p 183b8fe
tree 42a59b4058f927ef5af049e581480cd5530bd3b1
parent b45a2b162334376f2100974687742de1a23c2594
author John Smith <john.smith@networktocode.com> 1465333836 +0000
committer John Smith <john.smith@networktocode.com> 1465333836
+0000

Fix sw3 configuration for hypervisor
vagrant@jessie:~/net-auto$
```

This commit, the latest in the master branch, *also* points to the same parent commit—showing that the two branches diverge.

Now let's run the merge:

```
vagrant@jessie:~/net-auto$ git branch
* master
  sw4
vagrant@jessie:~/net-auto$ git merge sw4
(default Git editor opens to allow user to provide commit
message)
Merge made by the 'recursive' strategy.
```

```
 sw4.txt | 4 ++++
 1 file changed, 4 insertions(+)
vagrant@jessie:~/net-auto$ git log --oneline HEAD~3..HEAD
81f5963 Merge branch 'sw4'
183b8fe Fix sw3 configuration for hypervisor
40e88b8 Add port descriptions for sw4
b45a2b1 Add sw6 configuration
vagrant@jessie:~/net-auto$
```

In this instance, changes on *both* branches needed to be reconciled when merging the branches. It wasn't possible to just "replay" the changes from the sw4 branch to master, because master had some changes of its own. Thus, Git creates a *merge commit*. Let's look at that file real quick:

```
[vagrant@centos net-auto]$ git cat-file -p 81f5963
tree e71cfa26549241b609fa39e69dc51fdafd1d7cb4
parent 183b8fe2d02bbd6b2b7a19ecc39dc9e792fe2e75
parent 40e88b88271b535cceb311bb904d6afac20c15c3
author John Smith <john.smith@networktocode.com> 1465334492 +0000
committer John Smith <john.smith@networktocode.com> 1465334492
+0000

Merge branch 'sw4'
[vagrant@centos net-auto]$
```

Note the presence of *two* parent commits, which—if you look—represent the commits we made to each branch before merging the sw4 branch into master. This is how Git knows that the branches have converged, and how Git maintains the relationship between commits over time.

Now that the commits in the sw4 branch have been merged into the master branch, you can just delete the branch using `git branch -d sw4`:

```
[vagrant@centos net-auto]$ git branch -d sw4
Deleted branch sw4 (was 40e88b8).
[vagrant@centos net-auto]$ git branch
* master
[vagrant@centos net-auto]$
```

DELETING AN UNMERGED BRANCH

Note that it's possible to delete an unmerged branch using `git branch -D` *`branch`*. However, in such situations, you will *lose* the changes in that branch, so tread carefully.

So, to recap:

- To create a branch, use `git branch` *`new branch name`*.
- To check out a branch, use `git checkout` *`branch`*.
- To create a new branch and check it out in one step, use `git checkout -b` *`new branch name`*.
- To merge a branch into master, run `git merge` *`branch`* while the master branch is checked out.
- To delete a branch after its changes have been merged, use `git branch -d` *`branch name`*.

Let's now turn our attention to using Git's distributed nature to collaborate with others via Git.

Collaborating with Git

As we discussed in the section "Brief History of Git", one of the key design goals for Git was that it was a fully distributed system. Thus, every developer needed to be able to work from a full copy of the source code stored in the repository as well as the repository's full history. When you combine this fully distributed nature with Git's other key design goals—speed, simplicity, scalability, and strong support for non-linear development via lightweight branches—you can see why Git has become a leading option for users needing a collaborative version control system.

On its own, Git can act as a "server" and provides mechanisms for communications between systems running Git. Git supports a variety of transport protocols, including Secure Shell (SSH), HTTPS, and Git's own protocol (using TCP port 9418). If you're simply using Git on a couple of different systems and need to keep repositories in sync, you can do this with no additional software.

Further, Git's distributed nature has enabled a number of online services based on Git to appear. Many Git users take advantage of online Git-based services such as GitHub or BitBucket. There's also a wide variety of open source projects to facilitate collaboration via Git, such as GitLab, Gitblit, or Djacket (which, somewhat ironically, is hosted on GitHub). As you can see, there's no shortage of ways by which you can collaborate with others using Git and Git-based tools.

In this section, we'll explore how to collaborate using Git. That collaboration might be as simple as keeping repositories in sync on multiple systems, but we'll also cover using public Git-based services (focusing on GitHub). Along the way, you'll learn about cloning repositories; Git remotes; pushing, fetching, and pulling changes from other repositories; and using branches when collaborating.

Let's start with exploring a simple scenario involving multiple systems running Git, where you need to share/sync one or more repositories between these systems.

Collaborating Between Multiple Systems Running Git

So far in this chapter, you've been building your collection of network configurations, scripts, and templates in a Git repository on a single system. What happens when you need or want to be able to access this repository from a separate system? Maybe you have a desktop system at work and a laptop that you use for travel and at home. How do you use your network automation repository from both systems? Fortunately, because of Git's fully distributed design, this is pretty straightforward.

Can it be as simple as copying files? Let's see what happens when we simply copy a repository and its working directory to a new location on the same system. First, we'll run `git log --oneline HEAD~2..HEAD` in the existing repository:

```
vagrant@trusty:~/net-auto$ git log --oneline HEAD~2..HEAD
81f5963 Merge branch 'sw4'
183b8fe Fix sw3 configuration for hypervisor
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/net-auto$
```

Now, let's copy the repository and working directory to a new location on the same system, run the same `git log` command, and see what we get:

```
vagrant@trusty:~$ cp -ar net-auto netauto2
vagrant@trusty:~$ cd netauto2
vagrant@trusty:~/netauto2$ git log --oneline HEAD~2..HEAD
81f5963 Merge branch 'sw4'
183b8fe Fix sw3 configuration for hypervisor
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/netauto2$
```

Looks like the contents are identical! If you were to continue to explore the contents of the repository at *~/netauto2* using `git ls-tree`, `git cat-file`, or other commands, you'd find that the two repositories are, in fact, identical. Why is this? Recall that Git uses SHA1 hashes to identify all content: blobs, tree objects, and commit objects. A key property of the SHA1 hashes is that *identical content produces identical hashes.* Recall also that we stated that the contents of the Git repository are immutable (once created, they can't be modified). The combination of these attributes and Git's architecture means that it's possible to copy a repository using simple tools like `cp` and end up with an intact version of the repository. It's this ability to copy repositories—with all data and metadata intact—that is a key factor in Git's fully distributed nature.

Note there's no link between the copies, so changes made in one copy *won't* be automatically reflected in the other copy, or vice versa. (You can verify this, if you'd like, by making a commit in either copy, then using `git log` in both repositories.) In order to create a link between the copies, we have to explore something known as a *remote*.

LINKING REPOSITORIES WITH REMOTES

A Git *remote* is really nothing more than a reference to another repository. Git uses lightweight references pretty extensively—you've seen this already in the use of branches and HEAD—and in this case remotes are very similar. A remote is a lightweight reference to another repository, specified by a location.

Let's add a remote to the *netauto2* repository that refers back to the original repository in *net-auto*. To do this, we'll use the `git remote` command:

```
vagrant@jessie:~/netauto2$ git remote
vagrant@jessie:~/netauto2$ git remote add first ~/net-auto
vagrant@jessie:~/netauto2$ git remote
first
vagrant@jessie:~/netauto2$
```

When you use `git remote` with no parameters, it simply lists any existing remotes. In this case, there are none (yet). So you next run the `git remote add` command, which takes two parameters:

- The name to use for the remote repository. This name is purely symbolic—it can be whatever makes sense to you. In this case, we used `first` as the name for the remote.
- The location of the remote repository. In this case, the remote repository is on the same system (for now), so the location is simply a filesystem path.

Finally, running `git remote` again shows that the new remote has been added.

With the remote in place, you now have an asymmetric link between the two remotes. That is, *netauto2* has a reference to *net-auto*, but the reverse is *not* true. Via this asymmetric link, we can exchange information between Git repositories. Let's see how this works.

First, let's list the branches available in our *netauto2* repository. We'll add the `-a` parameter here, which we'll explain in more detail shortly:

```
vagrant@trusty:~/netauto2$ git branch -a
* master
vagrant@trusty:~/netauto2$
```

Now, let's fetch—and we're using the term *fetch* here intentionally, for reasons that will be evident later in this section—information from the remote repository, which you configured earlier. We'll update the information using the `git remote update` command, then run `git branch -a` again:

```
vagrant@trusty:~/netauto2$ git remote update first
Fetching first
From /home/vagrant/net-auto
 * [new branch]      master     -> first/master
vagrant@trusty:~/netauto2$ git branch -a
* master
  remotes/first/master
vagrant@trusty:~/netauto2$
```

Note there's now a new branch listed here. This is a special kind of branch known as a *remote tracking branch*. You won't make changes or commits to this branch, as it is only a reference to the branch that exists in the remote repository. You'll notice the `first` in the name of the branch; this refers back to the symbolic name you gave the Git remote when you added it. We had to use the `-a` parameter to `git branch` in order to show remote tracking branches, which aren't listed by default.

TIP

Instead of the two-step `git remote add` followed by `git remote update`, you can fetch information from a remote repository when you add the remote using the syntax **`git remote add -f`** ***`name location`***.

So what does this new remote tracking branch allow us to do? It allows us to *transfer* information between repositories in order to keep two repositories up-to-date. We'll show you how this works in the next section.

FETCHING AND MERGING INFORMATION FROM REMOTE REPOSITORIES

Once a remote has been configured for a repository, information has been retrieved from the configured remote, and remote tracking branches have been created, it's possible to start to transfer information between remotes using various `git` commands. You could use these commands to keep branches of repositories or entire repositories in sync.

To see this in action, we'll want to change one of the two repositories on your system (the *net-auto* repository) and see how to get that information into the *netauto2* repository.

First, in the *net-auto* repository, let's make a change to the *sw2.txt* configuration file and commit that change to the repository. (We won't go through all the steps here, as we've covered that previously. Need a hint? Edit the file, use `git add`, then `git commit`.)

Verify that you can see the new commit in the *net-auto* directory using `git log --oneline HEAD~1..HEAD`, then switch to the *netauto2* repository and run the same `git log` command. The commit(s) listed will be different.

To get the updated information from *net-auto* over to *netauto2*, you have a few options:

- You can run **`git remote update`** ***`name`***, which updates *only* the specified remote.
- You can run **`git remote update`** (without a remote's name), which updates *all* remotes for this repository.

- You can run **`git fetch`** ***`name`***, which will update (or *fetch*) information from the specified remote repository. In this respect, `git fetch` is a lot like `git remote update`, although the syntax is slightly different (again we refer you to the man pages or the help screens for specific details). Note that `git fetch` is considered the "conventional" way of retrieving information from a remote, as opposed to using `git remote update` as we did earlier.

So, let's run `git fetch first`, which will pull information from the repository named *first*. You'll see output that looks something like this (the SHA hashes will differ, of course):

```
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/vagrant/net-auto
   81f5963..7c2a3e6  master     -> first/master
```

OK, so we've retrieved information from *first/master* (the master branch of the remote named *first*). Why, then, does `git log` in `netauto2` not show this? This is because you've only *fetched* (updated) the information from the remote repository; you haven't actually made it part of the current repository.

CAUTION

We caution you against using the word "pull" when referring to simply retrieving information from a remote repository. In Git, the idea of "pulling" from a remote repository has a very specific meaning and its own command (both of which we'll discuss shortly). Try to train yourself to use "fetching" or "retrieving" when referring to the act of getting information from a remote repository.

So if you've only fetched the changes across but not made them a part of the current repository, how do you do that? That is, how do you make them part of the current repository? The changes in the remote repository are stored in a branch, which means they are kept separate from the default master branch of the current repository. How do we get changes from one branch to another branch? That's right—you *merge* the changes:

```
vagrant@jessie:~/netauto2$ git checkout master
Already on 'master'
```

```
vagrant@jessie:~/netauto2$ git merge first/master
Updating 81f5963..7c2a3e6
Fast-forward
 sw2.txt | 7 +++++++
 1 file changed, 7 insertions(+)
vagrant@jessie:~/netauto2$
```

As you can see from Git's output, it has taken the changes applied to *first/master* (the master branch of the remote repository named *first*) and merged them—via a fast-forward—into the master branch of the current repository. Because this is a fast-forward, there will not be a merge commit, and now both repositories are in sync.

NOTE

If you noted that the use of `git fetch` and `git merge` on the master branch of two repositories doesn't necessarily keep the repositories in sync, then you are *really* paying attention! In fact, only the master branches of the two repositories are in sync. To keep the entire repositories in sync, you'd need to perform this operation on all branches.

PULLING INFORMATION FROM REMOTE REPOSITORIES

Why the two-step process of first `git fetch` and then `git merge`? The primary reason is you might want to be able to review the changes from the remote repository *before* you merge them, in the event that you aren't ready for those changes to be applied to the current repository.

As with so many things in Git, though, there is a shortcut. If you'd like to fetch changes and merge them in a single operation, you can use **`git pull`** ***`name`***, where *`name`* is the name of the remote from which you'd like to get and merge changes into the current branch. The `git pull` command is simply combining the `git fetch` and `git merge` operations.

So you've seen how to get changes from *net-auto* to *netauto2*, but what about the reverse? We mentioned that adding a remote to *netauto2* is an asymmetric relationship in that *netauto2* now knows about *net-auto*, but the reverse is not true. In a situation such as we've described here—where you, as a single user, want to keep repositories in sync on separate systems—the best approach would be to add a remote from *net-auto* to *netauto2*, and then use `git` `fetch` and `git merge` to move changes in either direction. Graphically, this would look something like Figure 8-9.

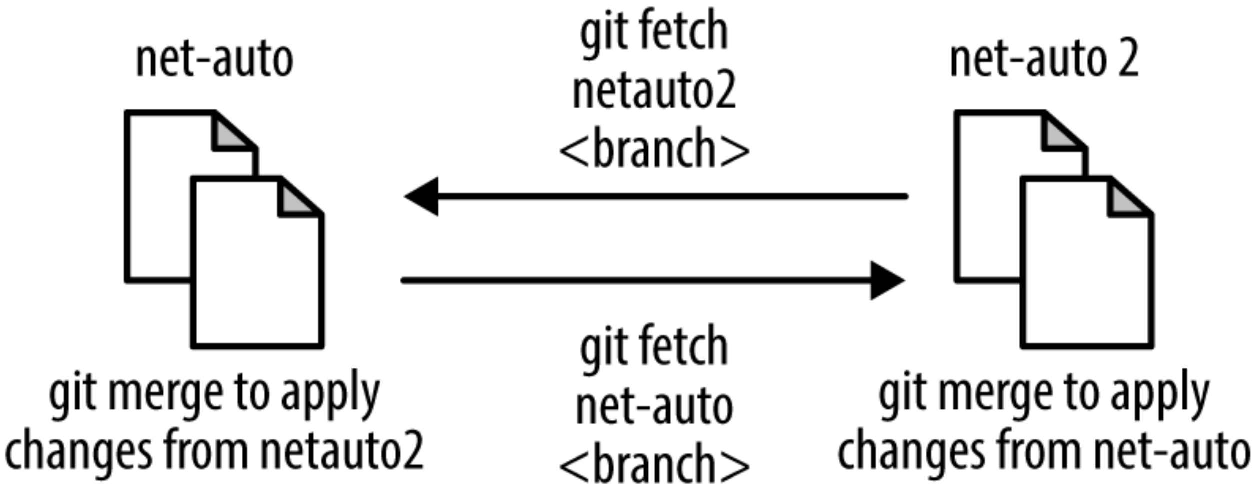
net-auto
git fetch netauto2 <branch>
net-auto 2
git merge to apply changes from netauto2
git fetch net-auto <branch>
git merge to apply changes from net-auto

Figure 8-9. Using git fetch and git merge between repositories

NOTE

Git almost always offers multiple ways to do something, which can be both useful (in that it is very flexible) and frustrating (in that there's no "one way" to do something). Two-way transfer of information between Git repositories as we've described in this section is one of these areas where there's more than one way to get the job done. For new users of Git, this is probably the easiest way to handle it.

We started this section asking the question, "How do I use my network automation repository from multiple systems?" We've shown you how to make a copy of a repository, how to use Git remotes to link repositories, and how to use various Git commands to transfer data between repositories. In the next two sections, we're going to show you a simpler, easier way of copying and linking repositories, and we'll extend our working model across multiple systems, respectively.

CLONING REPOSITORIES

In the previous sections, we showed you that you could simply copy a repository from one location to another and then use `git remote` to create a remote that would allow you to transfer information between repositories.

This process isn't difficult, but what if there were an even easier way? There is, and it's called *cloning* a repository via the `git clone` command. Let's see how this works.

The general syntax of this command is **`git clone`** ***`repository directory`***.

In this command, *`repository`* is the location of the repository you're cloning, and *`directory`* is the (optional) directory where you'd like to place the cloned repository. If you omit *`directory`*, then Git will place the cloned repository into a directory with the same name as the repository. Adding the *`directory`* parameter to the `git clone` command gives you some flexibility in where you'd like to place the cloned repository.

To illustrate how `git clone` works, let's kill the *netauto2* repository. It shouldn't have any changes in it, but if it does then you should know how to get those changes back into the original *net-auto* repository. (Need a hint? Add a remote, fetch changes, and merge the changes.)

```
vagrant@trusty:~$ rm -rf netauto2
vagrant@trusty:~$ git clone ~/net-auto na-clone
Cloning into 'na-clone'...
done.
vagrant@trusty:~$ cd na-clone
vagrant@trusty:~/na-clone$ git log --oneline HEAD~2..HEAD
7c2a3e6 Update sw2 configuration
81f5963 Merge branch 'sw4'
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/na-clone$
```

As you can see, `git clone` makes a copy of the repository, just like we did manually in the previous section. There's more, though—now run `git remote` in this new cloned repository:

```
vagrant@trusty:~/na-clone$ git remote -v
origin
vagrant@trusty:~/na-clone$
```

Here's the advantage of using `git clone` over the manual steps we showed you earlier—it *automatically* creates a remote pointing back to the original repository from which this repository was cloned. Further, it *automatically* creates remote tracking branches for you (you can verify this using `git branch -a` or `git branch -r`). Because it handles these extra steps for you, `git clone` should be your preferred mechanism for cloning a repository.

Before we move on, let's talk a bit about the "origin" branch that was automatically created by `git clone`. While the name of a remote is strictly symbolic, origin does have some special significance for Git. You can think of it as a default remote name. In cases where you have multiple remotes (yes, this is definitely possible!) and you run a `git fetch` without specifying a remote, Git will default to origin. Aside from this behavior, though, there are no special attributes given to the remote named origin.

As an example of using multiple remotes, this book itself was written using Git and multiple Git remotes. One Git remote was GitHub; the other was O'Reilly's repository. Here's the output of `git remote -v` from one author's repository:

```
oreilly     git@git.atlas.oreilly.com:oreillymedia/network-
automation.git (fetch)
oreilly     git@git.atlas.oreilly.com:oreillymedia/network-
automation.git (push)
origin      https://github.com/jedelman8/network-automation-
book.git (fetch)
origin      https://github.com/jedelman8/network-automation-
book.git (fetch)
```

Now we're finally ready to tackle the last step, which is taking everything we've learned so far and applying it to extend our Git working model with repositories across multiple systems.

EXTENDING OUR WORKING MODEL ACROSS MULTIPLE SYSTEMS

When we discussed the idea of creating a Git remote (see the section "Linking repositories with remotes"), we said that a remote had two attributes: the name (symbolic in nature) and the location. So far you've only seen remotes on the same system, but Git natively supports remotes on *different* systems across a variety of protocols.

For example, a remote on the same system uses a location like this:

```
/path/to/git/repository

file:///path/to/git/repository
```

However, a remote could also use various network protocols to reach a repository on a separate system:

```
git://host.domain.com/path/to/git/repository

ssh://[user@]host.domain.com/path/to/git/repository
http://host.domain.com/path/to/git/repository
https://host.domain.com/path/to/git/repository
```

The `git://` syntax references Git's native protocol, which is unauthenticated and therefore used for anonymous access (generally read-only access). The `ssh://` syntax refers to Secure Shell; this is actually Git's protocol tunneled over SSH for authenticated access. Finally, you have HTTP and HTTPS variants as well.

This means that you could take the working model we've described throughout this section and *easily* extend it to multiple systems using whatever network protocol best suits your needs. In this section, we're going to focus on the use of SSH, and later in this chapter we'll show you some examples of using HTTPS with public Git hosting services.

Going back to our example, let's say you need to be able to work on your network automation repository from both your desktop system and a laptop that you take with you. Let's assume that both systems support SSH (i.e., they are running Linux, macOS, or some other UNIX variant). The first step would be to configure passwordless authentication for SSH, generally using SSH keys. This will allow the various `git` commands to work without prompting for a password. Configuring SSH falls outside the scope of this book, but it's very well documented online.

The next step is then to create the necessary Git remotes. The repository already exists on your desktop (for the purposes of this example, we'll use our Ubuntu "Trusty Tahr" system to represent your desktop), but it doesn't exist on your laptop (which we'll represent with our Debian "Jessie" system). So, the first thing to do would be to clone the repository over to the laptop:

```
vagrant@jessie:~$ git clone ssh://trusty.domain.com/~/net-auto
net-auto
Cloning into 'net-auto'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (30/30), done.
remote: Total 32 (delta 12), reused 0 (delta 0)
Receiving objects: 100% (32/32), 2.99 KiB | 0 bytes/s, done.
Resolving deltas: 100% (12/12), done.
Checking connectivity... done.
```

```
vagrant@jessie:~$
```

This copies the repository from your desktop (*trusty.domain.com*) to your laptop (placing it in the directory specified; in this case, *net-auto*), creates a Git remote named origin pointing back to the original, and creates remote tracking branches. You can verify all this by using `git remote` to see the remote, and `git branch -r` to see remote tracking branches.

If you'd prefer to have a remote name that more clearly identifies where the remote is found, you can rename the remote from the default name, origin. Let's rename the remote to reflect that it's coming from our Ubuntu-based desktop:

```
vagrant@jessie:~/net-auto$ git remote
origin
vagrant@jessie:~/net-auto$ git remote rename origin trusty
vagrant@jessie:~/net-auto$ git remote
trusty
vagrant@jessie:~/net-auto$
```

Now, back on the Ubuntu system, we need to create a remote to the repository on the Debian laptop. The repository already exists here, so we can't use `git clone`; instead, we'll need to add the remote manually, and then fetch information from the remote to create the remote tracking branches:

```
vagrant@trusty:~/net-auto$ git remote add jessie
ssh://jessie.domain.com/~/net-auto
vagrant@trusty:~/net-auto$ git remote
jessie
vagrant@trusty:~$ git fetch jessie
From ssh://jessie.domain.com/~/net-auto
 * [new branch]      master     -> jessie/master
vagrant@trusty:~/net-auto$ git branch -r
  jessie/master
vagrant@trusty:~/net-auto$
```

Great—now you have the repository on both systems, a remote on each system pointing back to the other, and remote tracking branches created on each side. From here, the workflow is exactly like we described in earlier sections:

1. Make changes on either system (not both at the same time!) and commit those changes to the repository. Ideally, you should work *exclusively* in branches other than the master branch.

2. When you get back to the other system, run a `git fetch` and `git merge` to fetch and merge changes from remote branches into local branches. (If you don't care to review the changes before merging, you can use `git pull`.) Be sure to do this *before* you get started working!
3. Repeat as needed to keep branches on both systems up-to-date with each other.

This approach works fairly well for a single developer on two systems, but what about more than one developer? While it's possible to build a "full mesh" of Git remotes and remote tracking branches, this can quickly become very unwieldy. Using a shared repository in cases like this greatly simplifies how things work, as you'll see in the next section.

USING A SHARED REPOSITORY

If you've been poking around Git remotes with the `git remote` command, you may have discovered the `-v` switch, which enables more verbose output. For example, running `git remote -v` from one of the two systems configured in the previous section shows this:

```
trusty  ssh://trusty.domain.com/~/net-auto (fetch)
trusty  ssh://trusty.domain.com/~/net-auto (push)
```

This is useful, as it shows the full location of the remote repository. We've discussed the use of `git fetch` to retrieve information from the remote repository, but what's this idea of push?

So far we've only discussed the idea of retrieving information from a remote repository to your local repository through the use of commands like `git remote update`, `git fetch`, and `git pull`. It is possible, though, to send (Git uses the term *push*) changes to a remote repository from your local repository. However, in such cases, it is strongly recommended that the remote repository should be a *bare repository.*

What is a *bare repository?* Put simply, a bare repository is a Git repository without a working directory. (Recall that the term *working directory* has a very specific definition in Git, and shouldn't be taken to mean the same as the current directory.) All the discussions of Git repositories so far have assumed the presence of a working directory, because someone—a user like you—was going to be working on the repository. You, as the user of the repository, had to have some way of interacting with the content in the repository, and the working directory was how Git provided that method of interaction.

The reason you are strongly recommended against pushing to a non-bare repository (a repository with a working directory) is that a push doesn't reset the working directory. Let's go back to our previous example—two systems configured with remotes and remote tracking branches pointing to the other system—and see how this might cause problems:

1. Let's say you're being a really good Git citizen and you're working from a branch. We'll call this branch *new-feature*. You've got new-feature checked out on your first system, so it's the contents of the new-feature branch that are in the working directory. As the day ends, you still have a few unfinished changes left in the working directory, but you commit a few other changes.
2. From your second system, you fetch the changes, review them, merge them into the local new-feature branch, and continue working. You know that you can't see the uncommitted changes in the working directory on your first system, but that's no problem. All is well so far.
3. It's the end of the evening now, and you've just completed some work. You decide to push your changes to your work system's new-feature branch.
4. The next day, you come into work and decide to get started. Your uncommitted changes are still in the working directory, but you don't see the changes you pushed last night. What's going on here?

This is the issue with pushing to a non-bare repository: the changes were pushed to the remote repository, but the working directory was not updated. That's why you can't see the changes. In order to be able to see the changes, you'll have to run `git reset --hard HEAD`, which will *throw away* the changes in the working directory in order to show the pushed changes. Not a good situation, right?

Using a bare repository eliminates these problems, but it also eliminates the possibility of being able to interactively work with the repository. This is probably perfectly fine for a shared repository being used by multiple developers, though.

To create a new, bare repository, simply add the `--bare` option to `git init`:

```
vagrant@trusty:~$ git init --bare shared-repo.git
Initialized empty Git repository in /home/vagrant/shared-
repo.git/
vagrant@trusty:~$ git init non-bare-repo
Initialized empty Git repository in /home/vagrant/non-bare-
repo/.git/
```

Note the difference in the output of Git when `--bare` is used and when it is not used. In a non-bare repository, the actual Git repository is in the *.git* subdirectory, and Git's response indicates this. In a bare repository, though, there's no working directory, so the Git repository sits *directly* at the root of the directory specified.

NOTE

Although not required, it is accepted convention to end the name of a bare repository in *.git*.

In our case, though, we have an *existing* repository, and we need to somehow transition that into a bare repository that we can now share among multiple users. Git is prepared for such a scenario: we can use `git clone` to clone an existing repository into a new bare repository.

```
vagrant@trusty:~$ git clone --bare net-auto na-shared.git
Cloning into bare repository 'na-shared.git'...
done.
vagrant@trusty:~$
```

When you use `git clone --bare`, Git does not add any remotes or remote tracking branches. This makes sense, if you think about it; generally, remotes and remote tracking branches are useful only when you are interacting with the repository directly. With a bare repository, you aren't interacting with the repository directly; you'll use a clone on another system, which will have remotes and remote tracking branches.

Let's take our two-system setup (with the repository on Trusty and Jessie and remotes pointing back to each other) and transition it into a shared, bare repository on a third system. We'll introduce our third system, a CentOS system, to serve as the shared repository.

First, we need to get the repository onto the CentOS system. Here's where `git clone --bare` will come into play:

```
[vagrant@centos ~]$ git clone --bare
ssh://trusty.domain.com/~/net-auto
na-shared.git
Cloning into bare repository 'na-shared.git'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (30/30), done.
Receiving objects: 100% (32/32), done.
remote: Total 32 (delta 12), reused 0 (delta 0)
Resolving deltas: 100% (12/12), done.
[vagrant@centos ~]$
```

Now we can clone this bare repository onto our two work systems. First, the Ubuntu system:

```
vagrant@trusty:~$ git clone ssh://centos.domain.com/~vagrant/na-
shared.git
na-shared
Cloning into 'na-shared'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 32 (delta 12), reused 32 (delta 12)
Receiving objects: 100% (32/32), done.
Resolving deltas: 100% (12/12), done.
Checking connectivity... done.
vagrant@trusty:~$ cd na-shared
vagrant@trusty:~/na-shared$ git remote -v
origin  ssh://centos.domain.com/~vagrant/na-shared.git (fetch)
origin  ssh://centos.domain.com/~vagrant/na-shared.git (push)
vagrant@trusty:~/na-shared$ git branch -r
  origin/HEAD -> origin/master
  origin/master
vagrant@trusty:~/na-shared$ git log --oneline HEAD~2..HEAD
7c2a3e6 Update sw2 configuration
81f5963 Merge branch 'sw4'
40e88b8 Add port descriptions for sw4
vagrant@trusty:~/na-shared$
```

You can see that the `git clone` into the bare repository and subsequently back down to your Ubuntu system preserved all the data and metadata in the repository, and automatically created Git remotes and remote tracking branches. (You can verify the Git history, if you'd like, by running `git log` in the new `na-shared` repository as well as in the old `net-auto` repository still on your system.)

Next, we perform the same steps on the Debian system:

```
vagrant@jessie:~$ git clone ssh://centos.domain.com/~vagrant/na-
shared.git
na-shared
Cloning into 'na-shared'...
remote: Counting objects: 32, done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 32 (delta 12), reused 32 (delta 12)
Receiving objects: 100% (32/32), done.
Resolving deltas: 100% (12/12), done.
Checking connectivity... done.
vagrant@jessie:~$ cd na-shared
vagrant@jessie:~/na-shared$ git remote -v
origin  ssh://centos.domain.com/~vagrant/na-shared.git (fetch)
origin  ssh://centos.domain.com/~vagrant/na-shared.git (push)
vagrant@jessie:~/na-shared$ git branch -r
  origin/HEAD -> origin/master
  origin/master
vagrant@jessie:~/na-shared$
```

Now that you have the new *na-shared* repository on all your systems, you can simply remove the old *net-auto* repository with `rm -rf net-auto`.

What does the workflow look like now?

1. You'll still want to work almost exclusively in branches other than the master branch. This becomes particularly important when working with other users in the same shared repository.
2. Before starting work on the local clone on any system, run `git fetch` to retrieve any changes present on the shared repository but not in your local clone. Merge the changes into local branches as needed with `git merge`.
3. Make changes in the local repository and commit them to your local clone.
4. Push the changes up to the shared repository using `git push`.

We haven't talked about the `git push` command before, so it probably deserves a bit of explanation.

PUSHING CHANGES TO A SHARED REPOSITORY

Now that you have a bare repository, you can push changes to the remote using `git push`. The general syntax is **`git push`** ***`remote branch`***, where *`remote`* is the name of the Git remote and *`branch`* is the name of the branch to which these changes should be pushed.

To illustrate this in action, let's make some changes to the network automation repository on our Debian system. We'll add a Jinja template, *hv-tor-config.j2*, that represents the base configuration for a top-of-rack switch to which hypervisors are connected.

First, because we don't want to work off the master branch, we create a new branch to hold our changes:

```
vagrant@jessie:~/na-shared$ git checkout -b add-sw-tmpl
Switched to a new branch 'add-sw-tmpl'
vagrant@jessie:~/na-shared$
```

After we add the file to the working directory (by creating it from scratch or by copying it in from elsewhere), we'll stage and commit the changes:

```
vagrant@jessie:~/na-shared$ git add hv-tor-config.j2
vagrant@jessie:~/na-shared$ git commit -m "Add Jinja template for
TOR config"
[add-sw-tmpl 8cbbe6f] Add Jinja template for TOR config
 1 file changed, 15 insertions(+)
 create mode 100644 hv-tor-config.j2
vagrant@jessie:~/na-shared$
```

Now, we'll push the changes to the origin remote, which points to our shared (bare) repository:

```
vagrant@jessie:~/na-shared$ git push origin add-sw-tmpl
Counting objects: 3, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 423 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To ssh://centos.domain.com/~vagrant/na-shared.git
 * [new branch]      add-sw-tmpl -> add-sw-tmpl
vagrant@jessie:~/na-shared$
```

This allows coworkers and others with whom we are collaborating to then fetch the changes on their systems. They would just use `git fetch` to retrieve the changes, make a local branch corresponding to the remote tracking branch, then review the changes using whatever methods they wanted. Here, we'll show `git diff`, which isn't terribly useful considering the only change is adding a single new file.

```
vagrant@trusty:~/na-shared$ git fetch origin
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://centos.domain.com/~vagrant/na-shared
 * [new branch]      add-sw-tmpl -> origin/add-sw-tmpl
vagrant@trusty:~/na-shared$ git checkout --track -b add-sw-tmpl
origin/add-sw-tmpl
Branch add-sw-tmpl set up to track remote branch add-sw-tmpl from
origin.
Switched to a new branch 'add-sw-tmpl'
vagrant@trusty:~/na-shared$ git diff master..HEAD
diff --git a/hv-tor-config.j2 b/hv-tor-config.j2
new file mode 100644
index 0000000..989d723
--- /dev/null
+++ b/hv-tor-config.j2
@@ -0,0 +1,15 @@
+interface ethernet0
+  switchport mode access vlan {{ mgmt_vlan_id }}
+
+interface ethernet1
+  description Connected to {{ server_uplink_1 }}
+  switchport mode trunk
+
+interface ethernet2
+  description Connected to {{ server_uplink_2 }}
+  switchport mode trunk
+
+interface ethernet3
+  description Connected to {{ server_uplink_3 }}
+  switchport mode trunk
+
vagrant@trusty:~/na-shared$
```

Once everyone agrees that the changes are OK, then you can merge the changes into the master branch. First, perform the merge locally:

```
vagrant@jessie:~/na-shared$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'
vagrant@jessie:~$ git merge add-sw-tmpl
Updating 7c2a3e6..8cbbe6f
Fast-forward
```

```
 hv-tor-config.j2 | 15 +++++++++++++++
 1 file changed, 15 insertions(+)
 create mode 100644 hv-tor-config.j2
vagrant@jessie:~/na-shared$
```

This a fast-forward, so there’s no commit merge. Now push the changes to the shared repository:

```
vagrant@jessie:~/na-shared$ git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To ssh://centos.domain.com/~vagrant/na-shared.git
   7c2a3e6..8cbbe6f  master -> master
vagrant@jessie:~/na-shared$
```

Finally, delete your branch (also frequently referred to as a *feature branch* or a *topic branch*) and push that change to the shared repository:

```
vagrant@jessie:~/na-shared$ git branch -d add-sw-tmpl
Deleted branch add-sw-tmpl (was 8cbbe6f).
vagrant@jessie:~/na-shared$ git push origin --delete add-sw-tmpl
To ssh://centos.domain.com/~vagrant/na-shared.git
 - [deleted]         add-sw-tmpl
vagrant@jessie:~/na-shared$
```

Your collaborators can then get the changes that were merged into the master branch, delete the local branch they created, then delete the remote tracking branch that is no longer needed using the `git fetch --prune` command:

```
vagrant@trusty:~/na-shared$ git pull origin master
From ssh://centos.domain.com/~vagrant/na-shared
 * branch            master     -> FETCH_HEAD
   7c2a3e6..8cbbe6f  master     -> origin/master
Updating 7c2a3e6..8cbbe6f
Fast-forward
 hv-tor-config.j2 | 15 +++++++++++++++
 1 file changed, 15 insertions(+)
 create mode 100644 hv-tor-config.j2
vagrant@trusty:~/na-shared$ git fetch --prune origin
From ssh://192.168.100.12/~vagrant/na-shared
 x [deleted]         (none)     -> origin/add-sw-tmpl
vagrant@trusty:~/na-shared$ git branch -d add-sw-tmpl
Deleted branch add-sw-tmpl (was 8cbbe6f).
vagrant@trusty:~/na-shared$
```

The `git fetch --prune` command is new; it’s used to delete a remote tracking branch when the branch no longer exists on the remote. In this particular case, we're removing the remote tracking branch for *origin/add-sw-tmpl*, as noted in the output of the command.

BE PATIENT WHEN LEARNING GIT

We know that all this may sound really complicated if you're new to Git. It's OK—everyone was new to Git at some point (except maybe Linus). Take it slow, and be patient with yourself. After a little while of using Git, the commands will start to feel more natural. Until then, you might find it handy to have a Git "cheat sheet" nearby to remind you of some of the commands and their syntax.

Before we move on to our final topic—collaborating using Git-based online services—let's recap what we've discussed in this section:

- Git uses the concept of *remotes* to create links between repositories. You'll use the `git remote` command to manipulate remotes. A remote can point to a filesystem location as well as a location across the network, such as another system via SSH.
- To retrieve changes from a remote repository into your local repository, use `git fetch` *`remote`*.
- Git relies heavily on branches when working with remote repositories. Special branches known as remote tracking branches are automatically created when you use `git fetch` to retrieve changes.
- Changes retrieved from a remote repository can be merged into your local repository just like any other branch merge using `git merge`.
- If you don't want to follow the two-step `git fetch` followed by `git merge`, you can use `git pull`.
- You'll use `git push` to push changes to a remote repository, but this remote repository should be a bare repository.
- Using a bare repository as a central, shared repository can enable multiple users to collaborate on a single repository. Changes are exchanged via branches and through the use of `git push`, `git fetch`, `git merge`, and `git pull`.

Our last section in this chapter builds on everything we've shown you so far, and focuses on using Git-based online services to collaborate with other users.

Collaborating Using Git-Based Online Services

Fundamentally, collaborating with other users using a Git-based online services will look and feel very much like what we described in the previous section. All the same concepts apply—using clones to make copies of repositories, using remotes and remote tracking branches, and working in branches to exchange changes with other users in the same repository. You'll even continue to use the same commands: `git fetch`, `git push`, `git merge`, and `git pull`.

All that being said, there are a few differences that we'd like to cover. For the sake of brevity, we'll focus here on the use of GitHub as the Git-based online service by which you are collaborating. The topics we'll cover in this section are:

- Forking repositories
- Pull requests

Ready? Let's start with forking repositories.

FORKING REPOSITORIES

Forking a GitHub project is essentially the same as *cloning* a Git repository. (We'll use the terms *project* and *repository* somewhat interchangeably in this section.) When you fork a GitHub repository, you are issuing a command to GitHub's servers to clone the repository into your user account. At the time of creation, your fork will be a full and complete copy of the original repository, including all the content and the commit history. Once the repository has been forked into your account, it's just as if you'd issued a `git clone` from the command line—links are maintained back to the original project, much like a Git remote. (These remotes are not exposed to the user, though.) The key difference here is that forking a repository on GitHub does *not* create a local copy of the repository; you'll still need to use `git clone` to clone the forked copy down to your local system, as we'll show you shortly.

So why fork a repository? In the case of a large online service such as GitHub, there are hundreds of thousands of repositories hosted there. Each of these repositories is associated with a GitHub user ID, and that user ID is allowed to control who may or may not contribute to the repository. What if you found a repository to which you wanted to contribute? The owner of that repository may not know you (a very likely situation), and may not trust your ability to contribute to his or her repository. However, if you had your own copy of the repository, you could make the contributions you wanted to make and then let the owner of the original decide if such contributions were worthwhile. So, instead of trying to get approval to contribute directly to a repository, you instead *fork* (clone) the repository to your own account, where you can work with it. At some point later, you can (optionally) see if the original repository wants to include your changes moving forward (we'll discuss this in the section "Pull requests").

To fork a GitHub repository, just follow these steps:

1. Log into GitHub using your security credentials.
2. Locate the repository you'd like to fork into your own account, and click the Fork button in the upper-right corner of the screen.
3. If you are a member of any GitHub organizations, you may be prompted for the user account or organization where you'd like this repository to be forked. Choose your own user account unless you know you need a different option.

That's it! GitHub will fork (clone) the repository into your user account.

Because GitHub repositories are bare repositories, you'll generally need to then clone this bare repository down to your local system to work with it. (Note that GitHub provides some web-based tools to create files, edit files, make commits, and similar.) To clone a GitHub repository out of your account, you'd just use the `git clone` command, followed by the URL of the GitHub repository. For example, here's the URL of one of the authors' GitHub repositories: ***https://github.com/lowescott/learning-tools.git***

Let's say your GitHub username is npabook (this user did not exist at the time of this writing). If you were to fork the preceding repository, it would make a full and complete copy of the repository into your user account, just as if you'd used `git clone`. At this point, the URL for your forked repository would be: ***https://github.com/npabook/learning-tools.git***

If you ran `git clone https://github.com/npabook/learning-tools.git` from your local system, Git would clone the repository down to your local system, create a remote named origin that points back to your forked repository on GitHub, and create remote tracking branches—just as `git clone` worked in our earlier examples.

Once you have a clone of the repository on your local system, working with your forked GitHub repository is *exactly* like we described in the previous section:

1. Create new feature/topic branches locally to isolate changes away from the master branch.
2. Use `git push` to push those changes to the remote GitHub repository.
3. Merge the changes into the master branch using `git merge` whenever you're ready.
4. Use `git fetch` followed by `git merge` to pull down the changes to the master branch, or combine those steps using `git pull`.
5. Delete the local feature/topic branch and the remote branch on the GitHub repository.

So far, this should all seem pretty straightforward to you—we haven't really seen anything different from what we described earlier. There is one situation, though, that requires some discussion: how do you keep your fork in sync with the original?

KEEPING FORKED REPOSITORIES IN SYNC

Although GitHub maintains links back to the original repository when you fork it into your user account, GitHub does not provide a way to keep the two repositories synchronized. Why would it be important to keep your fork synchronized with the original? Suppose you wanted to contribute to an ongoing project. Over time, your forked copy would fall hopelessly behind the original as development continued, branches were merged, and changes committed to the original. In order for your fork to be useful for you to use to contribute changes, it needs to be up-to-date with the original.

To keep your forked repository up-to-date, you're going to need to use multiple remotes. (We did say earlier that multiple remotes is definitely something you might need to use with Git.) Let's walk through how this would work. We'll assume you've already forked the repository in GitHub.

First, clone the forked repository down to your system using `git clone`. The command would look something like this:

```
git clone https://username@github.com/username/repository-name.git
```

This will clone the repository down to your local system, create a remote named origin that points back to the URL specified before, and creates remote tracking branches. At this point, if you ran `git remote` in this repository, you'd see a single remote named origin (remember that `git remote -v` will also show the location of the remote—in this case, the HTTPS URL).

Next, add a *second* remote that points to the original repository. The command would look something like this:

```
git remote upstream add https://github.com/original-user/repository-
name.git
```

The name "upstream" here is strictly symbolic, but we like to use it, as this reminds us that the remote points to the upstream (or original) project. (We've also found that "upstream" is commonly used, so it may make sense to use the same remote name that others use for consistency.) Your local repository now has two remotes: origin, which points to your forked repository, and upstream, which points to the original repository.

Now, to keep your repository up-to-date with the original (all these steps are taken from within the cloned Git repository on your local system):

1. Check out the master branch using `git checkout master`.
2. Get the changes from the original repository. You can use a combination of `git fetch upstream master` followed by `git merge upstream/master`, or you can use `git pull upstream master` (which combines the steps). Your local, cloned repository is now in sync with the original repository.
3. Push the changes from your local repository to the forked repository using `git push origin master`. Now your forked repository is up-to-date with the original repository.

This process doesn't keep any feature/topic branches up-to-date, but that's generally not a problem—most of the time you'll only want to keep master synchronized between the original and the forked repository.

The next section talks about how to let the owner of a repository know that you have changes you'd like her to consider including in her repository.

PULL REQUESTS

Let's quickly recap the recommended process for working with a shared repository such as that offered by GitHub:

1. Create a local branch—called a feature or topic branch—in which to store the changes you're going to make.
2. Stage and commit the changes to the new local branch.
3. Push the local branch to the remote repository using `git push remote branch`.

That gets the changes into your forked repository, but how does the owner of the original repository know that you've pushed some changes up to your forked copy? In short: he or she *doesn't.* Why? Well, for one, it's entirely possible that you are truly forking—creating a divergent codebase—and don't want or need the original author(s) to know about your changes. Second, what if the changes you committed don't contain all the changes you want the original authors to consider? How would Git or GitHub know when you are ready? In short: it *can't*. Only you can know when your code is ready for the original authors to review for inclusion, and that's the purpose of a pull request.

A *pull request* is a notification to the authors of the original repository that you have changes you'd like them to consider including in their repository. Creating a pull request comes after step 3 in the preceding list. Once you've pushed your changes into a branch in your forked repository, you can create a pull request against the original repository. (Note that other Git-based platforms, such as GitLab, may use terms like *merge request* instead of pull request. The basic idea and workflow are much the same.)

To create a pull request after pushing a branch up to GitHub, go to the original repository. Just under the line listing the commits, branches, releases, and contributors, a new line will appear with a button labeled "Compare & pull request." This is illustrated in Figure 8-10.

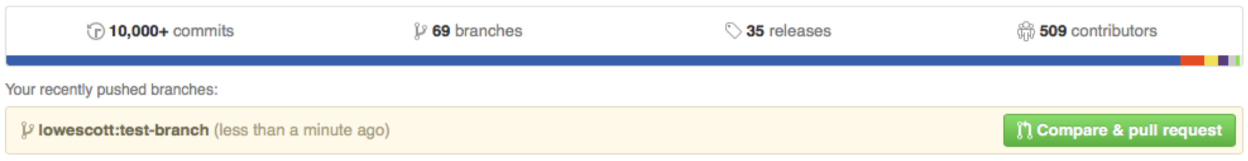
10,000+ commits
69 branches
35 releases
509 contributors
Your recently pushed branches:
lowescott:test-branch (less than a minute ago)
Compare & pull request

Figure 8-10. Creating a new pull request in GitHub

Click that button, and GitHub will open a screen to create the pull request. The base fork, base branch, head fork, and comparison branch will all be automatically filled in for you, and the notes in the pull request will be taken from the last commit message. Make any changes as needed, then click the green “Create pull request” button.

The owners of the original repository then have to decide if the changes found in your branch can and should be merged into their repository. If they agree—or if you are the one receiving the pull request—then you can merge the changes in GitHub’s web interface.

Once the changes have been merged into the original repository, you can update your fork’s master branch from the original (using `git fetch` and `git merge`, or the one-step `git pull`). Since the changes from your feature/topic branch are now found in the master branch, you can then delete the branch (as well as any remote tracking branches for your forked repository), as it is no longer needed.

As you can see, aside from a few minor differences, the general workflow for collaborating via GitHub is very similar to the workflow for collaborating using only a shared (bare) repository. By and large, the same terms, concepts, and commands are used in both cases, which makes it easier for you to collaborate with others using Git.

Summary

In this chapter, we’ve provided an introduction to Git, a very widely used version control system. Git is a fully distributed version control system that provides strong support for nonlinear development with branches. Like other version control systems, Git offers accountability (who made what changes) and change tracking (knowing the changes that were made). These attributes are just as applicable in networking-centric use cases as they are in developer-centric use cases. Branches are a key part of collaborating with Git. To help with Git collaboration, a number of online services (such as GitHub or BitBucket) have appeared, allowing users across organizations to collaborate on repositories with relative ease.