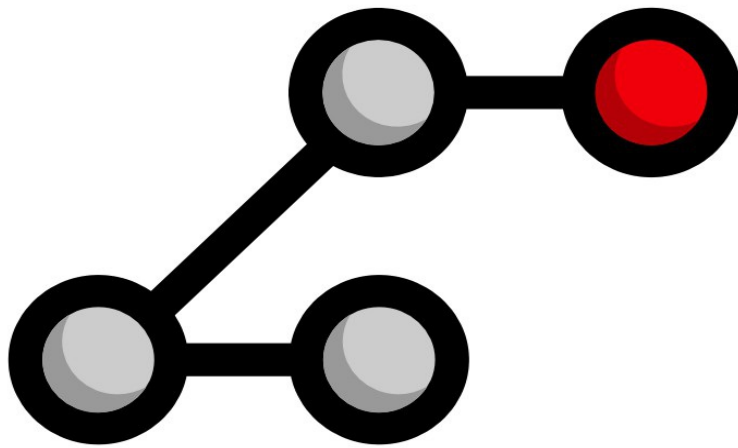


Git



Ry's Git Tutorial

By Ryan Hodson

Table of Contents

1. [Introduction](#)
 1. [A Brief History of Revision Control](#)
 2. [The Birth of Git](#)
 3. [Installation](#)
 4. [Get Ready!](#)
2. [The Basics](#)
 1. [Create the Example Site](#)
 2. [Initialize the Git Repository](#)
 3. [View the Repository Status](#)
 4. [Stage a Snapshot](#)
 5. [Commit the Snapshot](#)
 6. [View the Repository History](#)
 7. [Configure Git](#)
 8. [Create New HTML Files](#)
 9. [Stage the New Files](#)
 10. [Commit the New Files](#)
 11. [Modify the HTML Pages](#)
 12. [Stage and Commit the Snapshot](#)
 13. [Explore the Repository](#)
 14. [Conclusion](#)
 15. [Quick Reference](#)
3. [Undoing Changes](#)
 1. [Display Commit Checksums](#)
 2. [View an Old Revision](#)
 3. [View an Older Revision](#)
 4. [Return to Current Version](#)
 5. [Tag a Release](#)
 6. [Try a Crazy Experiment](#)
 7. [Stage and Commit the Snapshot](#)

8. [View the Stable Commit](#)
9. [Undo Committed Changes](#)
10. [Start a Smaller Experiment](#)
11. [Undo Uncommitted Changes](#)
12. [Conclusion](#)
13. [Quick Reference](#)

4. [Branches I](#)

1. [View Existing Branches](#)
2. [Checkout the Crazy Experiment](#)
3. [Create a New Branch](#)
4. [Make a Rainbow](#)
5. [Stage and Commit the Rainbow](#)
6. [Rename the Rainbow](#)
7. [Return to the Master Branch](#)
8. [Create a CSS Branch](#)
9. [Add a CSS Stylesheet](#)
10. [Link the Stylesheet](#)
11. [Return to the Master Branch \(Again\)](#)
12. [Merge the CSS Branch](#)
13. [Delete the CSS Branch](#)
14. [Conclusion](#)
15. [Quick Reference](#)

5. [Branches II](#)

1. [Continue the Crazy Experiment](#)
2. [Merge the CSS Updates](#)
3. [Style the Rainbow Page](#)
4. [Link to the Rainbow Page](#)
5. [Fork an Alternative Rainbow](#)
6. [Change the Rainbow](#)
7. [Emergency Update!](#)
8. [Publish the News Hotfix](#)

9. [Complete the Crazy Experiment](#)
10. [Publish the Crazy Experiment](#)
11. [Resolve the Merge Conflicts](#)
12. [Cleanup the Feature Branches](#)
13. [Conclusion](#)
14. [Quick Reference](#)

6. [Rebasing](#)

1. [Create an About Section](#)
2. [Add an About Page](#)
3. [Another Emergency Update!](#)
4. [Publish News Hotfix](#)
5. [Rebase the About Branch](#)
6. [Add a Personal Bio](#)
7. [Add Dummy Page for Mary](#)
8. [Link to the About Section](#)
9. [Clean Up the Commit History](#)
10. [Stop to Amend a Commit](#)
11. [Continue the Interactive Rebase](#)
12. [Publish the About Section](#)
13. [Conclusion](#)
14. [Quick Reference](#)

7. [Rewriting History](#)

1. [Create the Red Page](#)
2. [Create the Yellow Page](#)
3. [Link and Commit the New Pages](#)
4. [Create and Commit the Green Page](#)
5. [Begin an Interactive Rebase](#)
6. [Undo the Generic Commit](#)
7. [Split the Generic Commit](#)
8. [Remove the Last Commit](#)
9. [Open the Reflog](#)

10. [Revive the Lost Commit](#)
11. [Filter the Log History](#)
12. [Merge in the Revived Branch](#)
13. [Conclusion](#)
14. [Quick Reference](#)

8. [Remotes](#)

1. [Clone the Repository \(Mary\)](#)
2. [Configure The Repository \(Mary\)](#)
3. [Start Mary's Day \(Mary\)](#)
4. [Create Mary's Bio Page \(Mary\)](#)
5. [Publish the Bio Page \(Mary\)](#)
6. [View Remote Repositories \(Mary\)](#)
7. [Return to Your Repository \(You\)](#)
8. [Add Mary as a Remote \(You\)](#)
9. [Fetch Mary's Branches \(You\)](#)
10. [Check Out a Remote Branch](#)
11. [Find Mary's Changes](#)
12. [Merge Mary's Changes](#)
13. [Push a Dummy Branch](#)
14. [Push a New Tag](#)
15. [Conclusion](#)
16. [Quick Reference](#)

9. [Centralized Workflows](#)

1. [Create a Bare Repository \(Central\)](#)
2. [Update Remotes \(Mary and You\)](#)
3. [Push the Master Branch \(You\)](#)
4. [Add News Update \(You\)](#)
5. [Publish the News Item \(You\)](#)
6. [Update CSS Styles \(Mary\)](#)
7. [Update Another CSS Style \(Mary\)](#)
8. [Clean Up Before Publishing \(Mary\)](#)

9. Publish CSS Changes (Mary)
10. Pull in Changes (Mary)
11. Pull in Changes (You)
12. Conclusion
13. Quick Reference

10. Distributed Workflows

1. Create a Bitbucket Account
2. Create a Public Repository (You)
3. Push to the Public Repository (You)
4. Browse the Public Repository (You)
5. Clone the Repository (John)
6. Add the Pink Page (John)
7. Publish the Pink Page (John)
8. View John's Contributions (You)
9. Integrate John's Contributions (You)
10. Publish John's Contributions (You)
11. Update Mary's Repository (Mary)
12. Update John's Repository (John)
13. Conclusion

11. Patch Workflows

1. Change the Pink Page (Mary)
2. Create a Patch (Mary)
3. Add a Pink Block (Mary)
4. Create Patch of Entire Branch (Mary)
5. Mail the Patches (Mary)
6. Apply the Patches (You)
7. Integrate the Patches (You)
8. Update Mary's Repository (Mary)
9. Conclusion
10. Quick Reference

12. [Tips & Tricks](#)

1. [Archive The Repository](#)
2. [Bundle the Repository](#)
3. [Ignore a File](#)
4. [Stash Uncommitted Changes](#)
5. [Hook into Git's Internals](#)
6. [View Diffs Between Commits](#)
7. [Reset and Checkout Files](#)
8. [Aliases and Other Configurations](#)
9. [Conclusion](#)
10. [Quick Reference](#)

13. [Plumbing](#)

1. [Examine Commit Details](#)
2. [Examine a Tree](#)
3. [Examine a Blob](#)
4. [Examine a Tag](#)
5. [Inspect Git's Branch Representation](#)
6. [Explore the Object Database](#)
7. [Collect the Garbage](#)
8. [Add Files to the Index](#)
9. [Store the Index in the Database](#)
10. [Create a Commit Object](#)
11. [Update HEAD](#)
12. [Conclusion](#)
13. [Quick Reference](#)

Guide

1. [Table of Contents](#)
2. [Start of Content](#)

Introduction

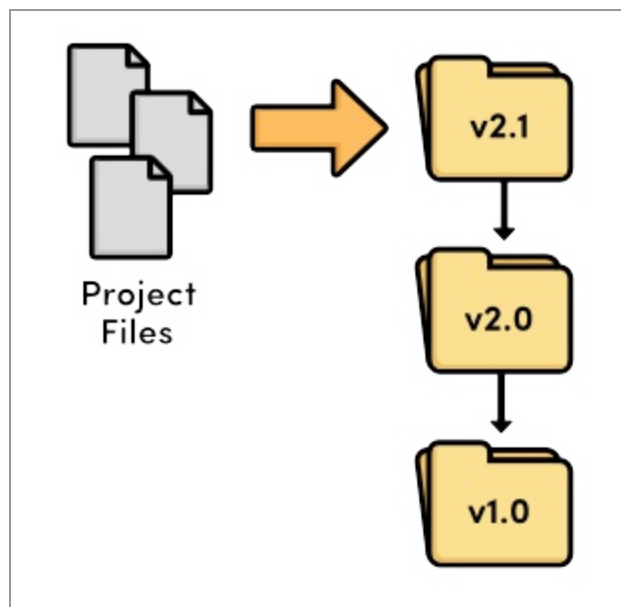
Git is a version control system (VCS) created for a single task: managing changes to your files. It lets you track every change a software project goes through, as well as where those changes came from. This makes Git an essential tool for managing large projects, but it can also open up a vast array of possibilities for your personal workflow.

A Brief History of Revision Control

We'll talk more about the core philosophy behind Git in a moment, but first, let's step through the evolution of version control systems in general.

Files and Folders

Before the advent of revision control software, there were only files and folders. The only way to track revisions of a project was to copy the entire project and give it a new name. Just think about how many times you've saved a "backup" called `my-term-paper-2.doc`. This is the simplest form of version control.

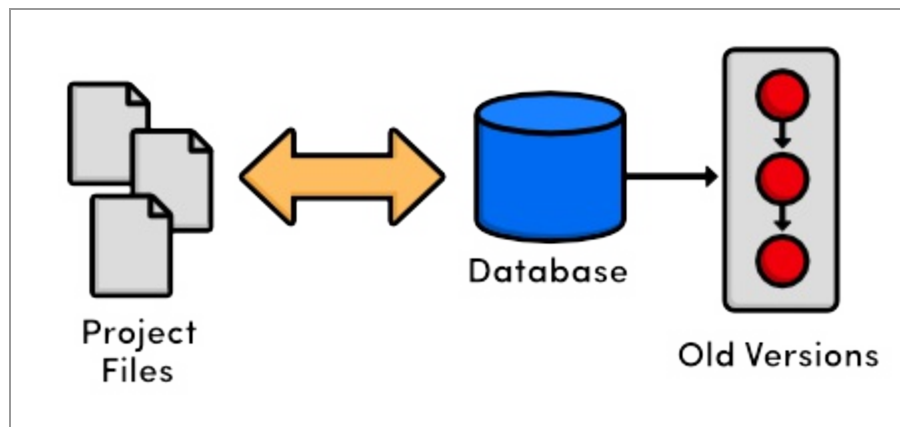


Revision control with files and folders

But, it's easy to see how copying files from folder to folder could prove disastrous for software developers. What happens if you mis-label a folder? Or if you overwrite the wrong file? How would you even know that you lost an important piece of code? It didn't take long for software developers to realize they needed something more reliable.

Local VCS

So, developers began writing utility programs dedicated to managing file revisions. Instead of keeping old versions as independent files, these new VCSs stored them in a database. When you needed to look at an old version, you used the VCS instead of accessing the file directly. That way, you would only have a single “checked out” copy of the project at any given time, eliminating the possibility of mixing up or losing revisions.



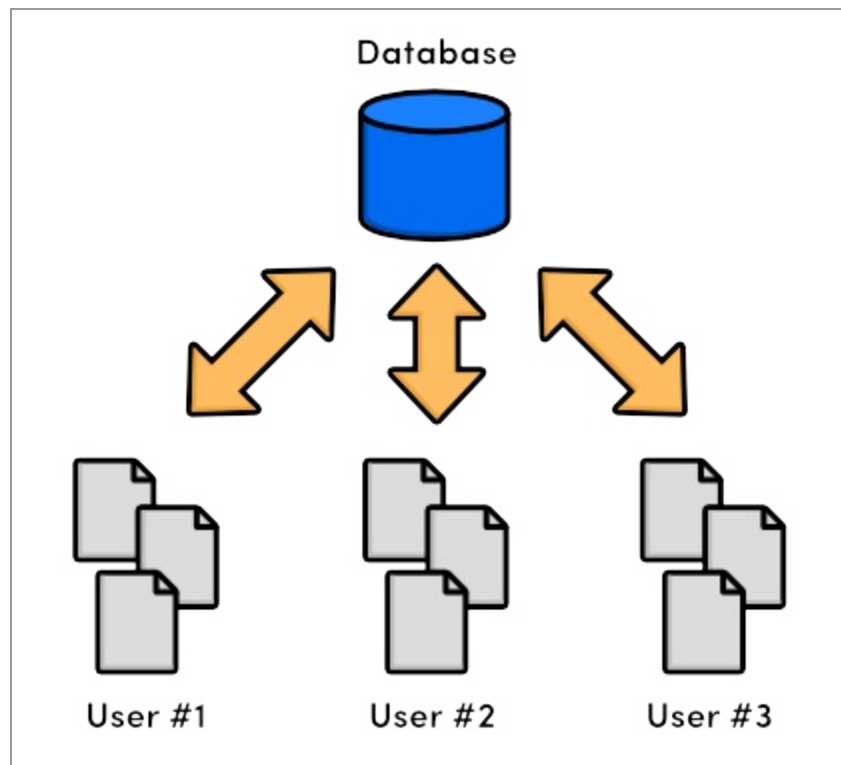
Local version control

At this point, versioning only took place on the developer's *local* computer—there was no way to efficiently share code amongst several programmers.

Centralized VCS

Enter the centralized version control system (CVCS). Instead of storing project history on the developer's hard disk, these new CVCS programs stored everything on a server. Developers checked out files and saved them back into the project over a network. This setup let several programmers

collaborate on a project by giving them a single point of entry.



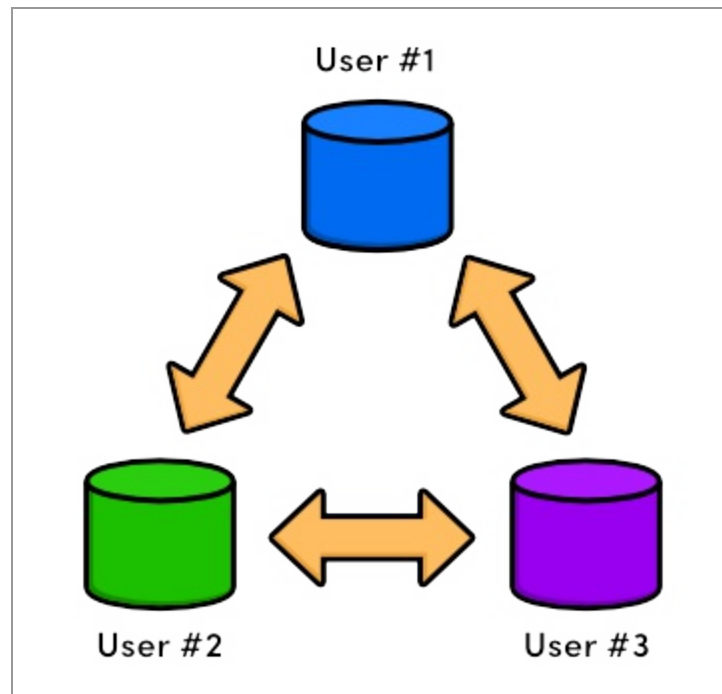
Centralized version control

While a big improvement on local VCS, centralized systems presented a new set of problems: how do multiple users work on the same files at the same time? Just imagine a scenario where two people fix the same bug and try to commit their updates to the central server. Whose changes should be accepted?

CVCSs addressed this issue by preventing users from overriding others' work. If two changes conflicted, someone had to manually go in and merge the differences. This solution worked for projects with relatively few updates (which meant relatively few conflicts), but proved cumbersome for projects with dozens of active contributors submitting several updates everyday: development couldn't continue until all merge conflicts were resolved and made available to the entire development team.

Distributed VCS

The next generation of revision control programs shifted away from the idea of a single centralized repository, opting instead to give every developer their own *local* copy of the entire project. The resulting *distributed* network of repositories let each developer work in isolation, much like a local VCS—but now the conflict resolution problem of CVCS had a much more elegant solution.



Distributed version control

Since there was no longer a central repository, everyone could develop at their own pace, store the updates locally, and put off merging conflicts until their convenience. In addition, distributed version control systems (DVCS) focused on efficient management for separate branches of development, which made it much easier to share code, merge conflicts, and experiment with new ideas.

The local nature of DVCSs also made development much faster, since you no longer had to perform actions over a network. And, since each user had a complete copy of the project, the risk of a server crash, a corrupted

repository, or any other type of data loss was much lower than that of their CVCS predecessors.

The Birth of Git

And so, we arrive at Git, a distributed version control system created to manage the Linux kernel. In 2005, the Linux community lost their free license to the BitKeeper software, a commercial DVCS that they had been using since 2002. In response, Linus Torvalds advocated the development of a new open-source DVCS as a replacement. This was the birth of Git.

As a source code manager for the entire Linux kernel, Git had several unique constraints, including:

- Reliability
- Efficient management of large projects
- Support for distributed development
- Support for non-linear development

While other DVCSs did exist at the time (e.g., GNU's Arch or David Roundy's Darcs), none of them could satisfy this combination of features. Driven by these goals, Git has been under active development for several years and now enjoys a great deal of stability, popularity, and community involvement.

Git originated as a command-line program, but a variety of visual interfaces have been released over the years. Graphical tools mask some of the complexity behind Git and often make it easier to visualize the state of a repository, but they still require a solid foundation in distributed version control. With this in mind, we'll be sticking to the command-line interface, which is still the most common way to interact with Git.

Installation

The upcoming modules will explore Git's features by applying commands to real-world scenarios. But first, you'll need a working Git installation to experiment with. Downloads for all supported platforms are available via the [official Git website](#).

For Windows users, this will install a special command shell called *Git Bash*. You should be using this shell instead of the native command prompt to run Git commands. OS X and Linux users can access Git from a normal shell. To test your installation, open a new command prompt and run `git --version`. It should output something like `git version 1.7.10.2 (Apple Git-33)`.

Get Ready!

Remember that *Ry's Git Tutorial* is designed to *demonstrate* Git's feature set, not just give you a superficial overview of the most common commands. To get the most out of this tutorial, it's important to actually execute the commands you're reading about. So, make sure you're sitting in front of a computer, and let's get to it!

The Basics

Now that you have a basic understanding of version control systems in general, we can start experimenting with Git. Using Git as a VCS is a lot like working with a normal software project. You're still writing code in files and storing those files in folders, only now you have access to a plethora of Git commands to manipulate those files.

For example, if you want to revert to a previous version of your project, all you have to do is run a simple Git command. This command will dive into Git's internal database, figure out what your project looked like at the desired state, and update all the existing files in your project folder (also known as the **working directory**). From an external standpoint, it will look like your project magically went back in time.

This module explores the fundamental Git workflow: creating a repository, staging and committing snapshots, configuring options, and viewing the state of a repository. It also introduces the HTML website that serves as the running example for this entire tutorial. A very basic knowledge of HTML and CSS will give you a deeper understanding of the purpose underlying various Git commands but is not strictly required.

Create the Example Site

Before we can execute any Git commands, we need to create the example project. Create a new folder called `my-git-repo` to store the project, then add a file called `index.html` to it. Open `index.html` in your favorite text editor and add the following HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>A Colorful Website</title>
```

```
<meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #07F">A Colorful Website</h1>
  <p>This is a website about color!</p>

  <h2 style="color: #C00">News</h2>
  <ul>
    <li>Nothing going on (yet)</li>
  </ul>
</body>
</html>
```

Save the file when you're done. This serves as the foundation of our example project. Feel free to open the `index.html` in a web browser to see what kind of website it translates to. It's not exactly pretty, but it serves our purposes.

Initialize the Git Repository

Now, we're ready to create our first Git repository. Open a command prompt (or Git Bash for Windows users) and navigate to the project directory by executing:

```
cd /path/to/my-git-repo
```

where `/path/to/my-git-repo` is a path to the folder created in the previous step. For example, if you created `my-git-repo` on your desktop, you would execute:

```
cd ~/Desktop/my-git-repo
```

Next, run the following command to turn the directory into a Git repository.

```
git init
```

This initializes the repository, which enables the rest of Git’s powerful features. Notice that there is now a `.git` directory in `my-git-repo` that stores all the tracking data for our repository (you may need to enable hidden files to view this folder). The `.git` folder is the only difference between a Git repository and an ordinary folder, so deleting it will turn your project back into an unversioned collection of files.

View the Repository Status

Before we try to start creating revisions, it would be helpful to view the status of our new repository. Execute the following in your command prompt.

```
git status
```

This should output something like:

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       index.html
nothing added to commit but untracked files present (use "git add" to track)
```

Ignoring the `On branch master` portion for the time being, this status message tells us that we’re on our initial commit and that we have nothing to commit but “untracked files.”

An **untracked file** is one that is not under version control. Git doesn't automatically track files because there are often project files that we don't want to keep under revision control. These include binaries created by a C program, compiled Python modules (.pyc files), and any other content that would unnecessarily bloat the repository. To keep a project small and efficient, you should only track *source* files and omit anything that can be *generated* from those files. This latter content is part of the build process—not revision control.

Stage a Snapshot

So, we need to explicitly tell Git to add `index.html` to the repository. The aptly named `git add` command tells Git to start tracking `index.html`:

```
git add index.html
git status
```

In place of the “Untracked files” list, you should see the following status.

```
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   index.html
```

We've just added `index.html` to the **snapshot** for the next commit. A snapshot represents the state of your project at a given point in time. In this case, we created a snapshot with one file: `index.html`. If we ever told Git to revert to this snapshot, it would replace the entire project folder with this one file, containing the exact same HTML as it does right now.

Git's term for creating a snapshot is called **staging** because we can add or remove multiple files before actually committing it to the project history.

Staging gives us the opportunity to group related changes into distinct snapshots—a practice that makes it possible to track the *meaningful* progression of a software project (instead of just arbitrary lines of code).

Commit the Snapshot

We have staged a snapshot, but we still need to **commit** it to the project history. The next command will open a text editor and prompt you to enter a message for the commit.

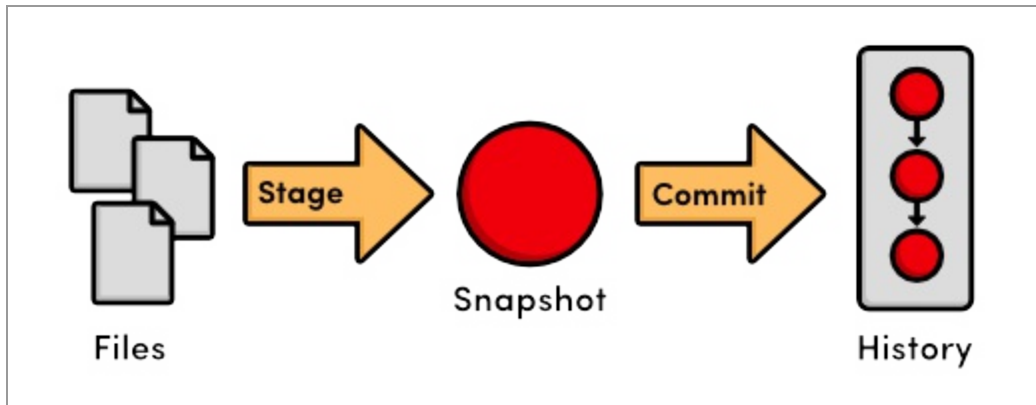
```
git commit
```

Type `Create index page` for the message, leave the remaining text, save the file, and exit the editor. You should see the message `1 files changed` among a mess of rather ambiguous output. This changed file is our `index.html`.

As we just demonstrated, saving a version of your project is a two step process:

1. **Staging.** Telling Git what files to include in the next commit.
2. **Committing.** Recording the staged snapshot with a descriptive message.

Staging files with the `git add` command doesn't actually affect the repository in any significant way—it just lets us get our files in order for the next commit. Only after executing `git commit` will our snapshot be recorded in the repository. Committed snapshots can be seen as “safe” versions of the project. Git will never change them, which means you can do almost anything you want to your project without losing those “safe” revisions. This is the principal goal of any version control system.



The stage/commit process

View the Repository History

Note that `git status` now tells us that there is nothing to commit, which means our current state matches what is stored in the repository. The `git status` command will *only* show us uncommitted changes—to view our project history, we need a new command:

```
git log
```

When you execute this command, Git will output information about our one and only commit, which should look something like this:

```
commit b650e4bd831aba05fa62d6f6d064e7ca02b5ee1b
Author: unknown <user@computer.(none)>
Date:   Wed Jan 11 00:45:10 2012 -0600
```

Create index page

Let's break this down. First, the commit is identified with a very large, very random-looking string (b650e4b...). This is an SHA-1 checksum of the commit's contents, which ensures that the commit will never be corrupted without Git knowing about it. All of your SHA-1 checksums will be different

than those displayed in this tutorial due to the different dates and authors in your commits. In the next module, we'll see how a checksum also serves as a unique ID for a commit.

Next, Git displays the author of the commit. But since we haven't told Git our name yet, it just displays unknown with a generated username. Git also outputs the date, time, and timezone (-0600) of when the commit took place. Finally, we see the commit message that was entered in the previous step.

Configure Git

Before committing any more snapshots, we should probably tell Git who we are. We can do this with the `git config` command:

```
git config --global user.name "Your Name"
git config --global user.email your.email@example.com
```

Be sure to replace `Your Name` and `your.email@example.com` with your actual name and email. The `--global` flag tells Git to use this configuration as a default for all of your repositories. Omitting it lets you specify different user information for individual repositories, which will come in handy later on.

Create New HTML Files

Let's continue developing our website a bit. Start by creating a file called `orange.html` with the following content.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Orange Page</title>
  <meta charset="utf-8" />
```

```
</head>

<body>

  <h1 style="color: #F90">The Orange Page</h1>

  <p>Orange is so great it has a

  <span style="color: #F90">fruit</span> named after it.</p>

</body>

</html>
```

Then, add a `blue.html` page:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Blue Page</title>
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #00F">The Blue Page</h1>
  <p>Blue is the color of the sky.</p>
</body>
</html>
```

Stage the New Files

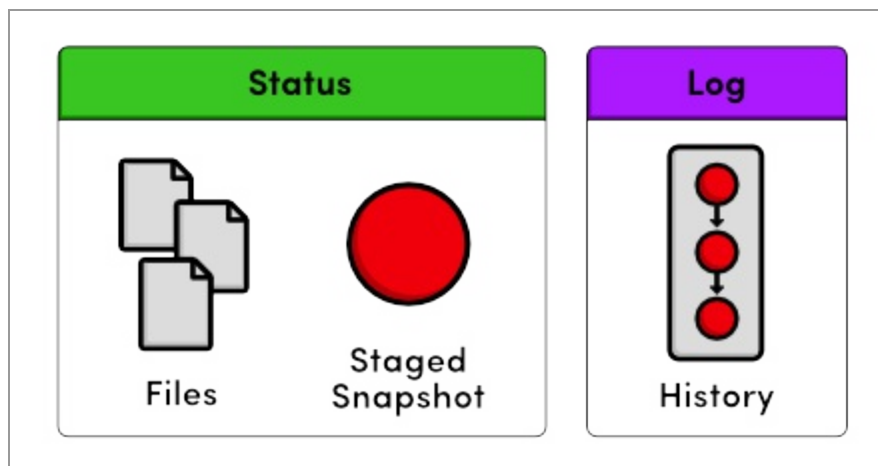
Next, we can stage the files the same way we created the first snapshot.

```
git add orange.html blue.html
git status
```

Notice that we can pass more than one file to `git add`. After adding the files, your status output should look like the following:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   blue.html
#       new file:   orange.html
```

Try running `git log`. It only outputs the first commit, which tells us that `blue.html` and `orange.html` have not yet been added to the repository's history. Remember, we can see *staged* changes with `git status`, but not with `git log`. The latter is used only for *committed* changes.



Status output vs. Log output

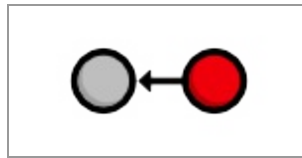
Commit the New Files

Let's commit our staged snapshot:

```
git commit
```

Use `Create blue` and `orange` pages as the commit message, then save and close the file. Now, `git log` should output two commits, the second of which

reflects our name/email configuration. This project history can be visualized as:



Current project history

Each circle represents a commit, the red circle designates the commit we're currently viewing, and the arrow points to the preceding commit. This last part may seem counterintuitive, but it reflects the underlying relationship between commits (that is, a new commit refers to its parent commit). You'll see this type of diagram many, many times throughout this tutorial.

Modify the HTML Pages

The `git add` command we've been using to stage *new* files can also be used to stage *modified* files. Add the following to the bottom of `index.html`, right before the closing `</body>` tag:

```
<h2>Navigation</h2>
<ul>
  <li style="color: #F90">
    <a href="orange.html">The Orange Page</a>
  </li>
  <li style="color: #00F">
    <a href="blue.html">The Blue Page</a>
  </li>
</ul>
```

Next, add a home page link to the bottom of `orange.html` and `blue.html` (again, before the `</body>` line):

```
<p><a href="index.html">Return to home page</a></p>
```

You can now navigate between pages when viewing them in a web browser.

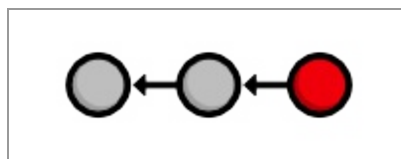
Stage and Commit the Snapshot

Once again, we'll stage the modifications, then commit the snapshot.

```
git status
git add index.html orange.html blue.html
git status
git commit -m "Add navigation links"
```

The `-m` option lets you specify a commit message on the command line instead of opening a text editor. This is just a convenient shortcut (it has the exact same effect as our previous commits).

Our history can now be represented as the following. Note that the red circle, which represents the current commit, automatically moves forward every time we commit a new snapshot.



Current project history

Explore the Repository's History

The `git log` command comes with a lot of formatting options, a few of which will be introduced throughout this tutorial. For now, we'll just use the convenient `--oneline` flag:


```
git log --oneline
```

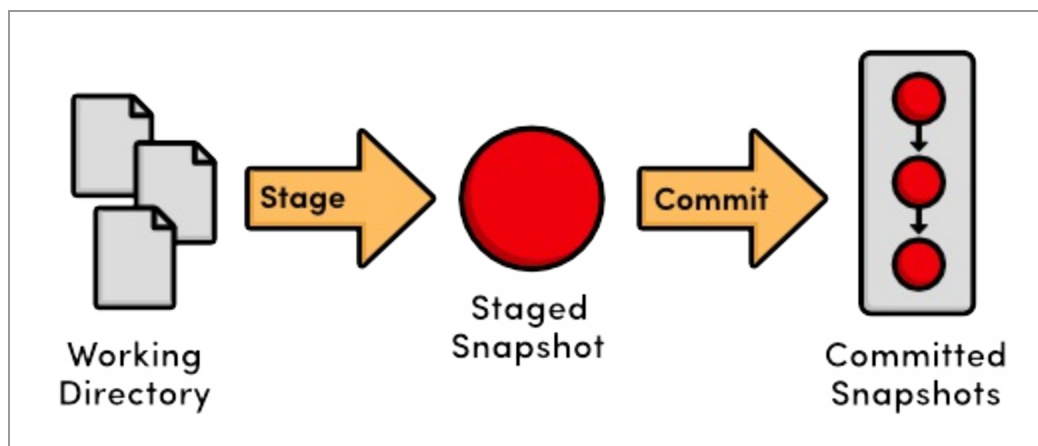
Condensing output to a single line is a great way to get a high-level overview of a repository. Another useful configuration is to pass a filename to `git log`:

```
git log --oneline blue.html
```

This displays only the `blue.html` history. Notice that the initial `Create index page` commit is missing, since `blue.html` didn't exist in that snapshot.

Conclusion

In this module, we introduced the fundamental Git workflow: edit files, stage a snapshot, and commit the snapshot. We also had some hands-on experience with the components involved in this process:



The fundamental Git workflow

The distinction between the working directory, the staged snapshot, and committed snapshots is at the very core of Git version control. Nearly all other Git commands manipulate one of these components in some way, so understanding the interplay between them is a fantastic foundation for

mastering Git.

The next module puts our existing project history to work by reverting to previous snapshots. This is all you need to start using Git as a simple versioning tool for your own projects.

Quick Reference

`git init`

Create a Git repository in the current folder.

`git status`

View the status of each file in a repository.

`git add <file>`

Stage a file for the next commit.

`git commit`

Commit the staged files with a descriptive message.

`git log`

View a repository's commit history.

`git config --global user.name "<name>"`

Define the author name to be used in all repositories.

`git config --global user.email <email>`

Define the author email to be used in all repositories.

Undoing Changes

In the last module, we learned how to record versions of a project into a Git repository. The whole point of maintaining these “safe” copies is peace of mind: should our project suddenly break, we’ll know that we have easy access to a functional version, and we’ll be able to pinpoint precisely where the problem was introduced.

To this end, storing “safe” versions isn’t much help without the ability to restore them. Our next task is to learn how to view the previous states of a project, revert back to them, and reset uncommitted changes.

[Download the repository for this module](#)

If you’ve been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repository from the above link, uncompress it, and you’re good to go.

Display Commit Checksums

As a quick review, let’s display our repository’s history. Navigate a command prompt (or Git Bash) to the `my-git-repo` folder and run the following.

```
git log --oneline
```

The output for this should look similar to the following, but contain different commit checksums.

```
1c310d2 Add navigation links
54650a3 Create blue and orange pages
b650e4b Create index page
```

Git only outputs the first 7 characters of the checksum (remember that you can see the full version with the default formatting of `git log`). These first few characters effectively serve as a unique ID for each commit.

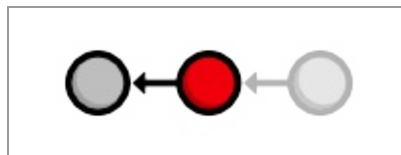
View an Old Revision

Using the new `git checkout` command, we can view the contents of a previous snapshot. Make sure to change `54650a3` in the following command to the ID of your *second* commit.

```
git checkout 54650a3
```

This will output a lot of information about a detached `HEAD` state. You can ignore it for now. All you need to know is that the above command turns your `my-git-repo` directory into an exact replica of the second snapshot we committed in [The Basics](#).

Open the HTML files in a text editor or web browser to verify that the navigation links we added in the third commit have disappeared. Running `git log` will also tell us that the third commit is no longer part of the project. After checking out the second commit, our repository history looks like the following (the red circle represents the current commit).



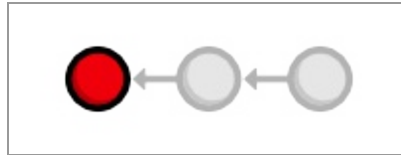
Checking out the 2nd commit

View an Older Revision

Let's go even farther back in our history. Be sure to change `b650e4b` to the ID of your *first* commit.

```
git checkout b650e4b
```

Now, the `blue.html` and `orange.html` files are gone, as is the rest of the `git` log history.



Checking out the 1st commit

In the last module, we said that Git was designed to never lose a committed snapshot. So, where did our second and third snapshots go? A simple `git status` will answer that question for us. It should return the following message:

```
# Not currently on any branch.  
nothing to commit (working directory clean)
```

Compare this with the status output from the previous module:

```
# On branch master  
nothing to commit (working directory clean)
```

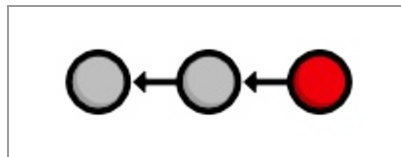
All of our actions in *The Basics* took place on the `master` branch, which is where our second and third commits still reside. To retrieve our complete history, we just have to check out this branch. This is a very brief introduction to branches, but it's all we need to know to navigate between commits. The next module will discuss branches in full detail.

Return to Current Version

We can use the same `git checkout` command to return to the master branch.

```
git checkout master
```

This makes Git update our working directory to reflect the state of the master branch's snapshot. It re-creates the `blue.html` and `orange.html` files for us, and the content of `index.html` is updated as well. We're now back to the current state of the project, and our history looks like:



Current project history

Tag a Release

Let's call this a stable version of the example website. We can make it official by **tagging** the most recent commit with a version number.

```
git tag -a v1.0 -m "Stable version of the website"
```

Tags are convenient references to official releases and other significant milestones in a software project. It lets developers easily browse and check out important revisions. For example, we can now use the `v1.0` tag to refer to the third commit instead of its random ID. To view a list of existing tags, execute `git tag` without any arguments.

In the above snippet, the `-a` flag tells Git to create an **annotated tag**, which lets us record our name, the date, and a descriptive message (specified via the `-m` flag). We'll use this tag to find the stable version after we try some crazy experiments.

Try a Crazy Experiment

We're now free to add experimental changes to the example site without affecting any committed content. Create a new file called `crazy.html` and add the following HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>A Crazy Experiment</title>
  <meta charset="utf-8" />
</head>
<body>
  <h1>A Crazy Experiment</h1>
  <p>We're trying out a <span style="color: #F0F">crazy</span>
  <span style="color: #06C">experiment</span>!

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Stage and Commit the Snapshot

Stage and commit the new file as usual.

```
git add crazy.html
git status
git commit -m "Add a crazzzy experiment"
git log
```

It's a good practice to run `git status` to see exactly what you're committing before running `git commit -m`. This will keep you from unintentionally

committing a file that doesn't belong in the current snapshot.

As expected, the new snapshot shows up in the repository's history. If your log history takes up more than one screen, you can scroll down by pressing space and return to the command line by pressing the letter q.

View the Stable Commit

Let's go back and take a look at our stable revision. Remember that the `v1.0` tag now serves as a user-friendly shortcut to the third commit's ID.

```
git checkout v1.0
```

After seeing the stable version of the site, we decide to scrap the crazy experiment. But, before we undo the changes, we need to return to the master branch. If we didn't, all of our updates would be on some non-existent branch. As we'll discover next module, you should never make changes directly to a previous revision.

```
git checkout master  
git log --oneline
```

At this point, our history should look like the following:

```
514fbe7 Add a crazzzy experiment  
1c310d2 Add navigation links  
54650a3 Create blue and orange pages  
b650e4b Create index page
```

Undo Committed Changes

We're ready to restore our stable tag by removing the most recent commit.

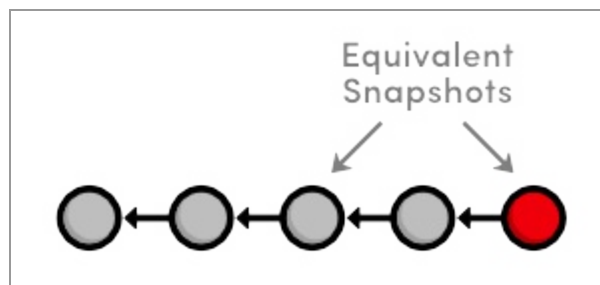
Make sure to change 514fbe7 to the ID of the *crazy experiment's* commit before running the next command.

```
git revert 514fbe7
```

This will prompt you for a commit message with a default of Revert "Add a crazzzy experiment". . . . You can leave the default message and close the file. After verifying that crazy.html is gone, take a look at your history with `git log --oneline`.

```
506bb9b Revert "Add a crazzzy experiment"  
514fbe7 Add a crazzzy experiment  
1c310d2 Add navigation links  
54650a3 Create blue and orange pages  
b650e4b Create index page
```

Notice that instead of deleting the “crazzzy experiment” commit, Git figures out how to undo the changes it contains, then tacks on another commit with the resulting content. So, our fifth commit and our third commit represent the exact same snapshot, as shown below. Again, Git is designed to *never* lose history: the fourth snapshot is still accessible, just in case we want to continue developing it.



Current project history

When using `git revert`, remember to specify the commit that you want to

undo—not the stable commit that you want to return to. It helps to think of this command as saying “undo this commit” rather than “restore this version.”

Start a Smaller Experiment

Let’s try a smaller experiment this time. Create `dummy.html` and leave it as a blank file. Then, add a link in the “Navigation” section of `index.html` so that it resembles the following.

```
<h2>Navigation</h2>
<ul>
  <li style="color: #F90">
    <a href="orange.html">The Orange Page</a>
  </li>
  <li style="color: #00F">
    <a href="blue.html">The Blue Page</a>
  </li>
  <li>
    <a href="dummy.html">The Dummy Page</a>
  </li>
</ul>
```

In the next section, we’re going to abandon this uncommitted experiment. But since the `git revert` command requires a commit ID to undo, we can’t use the method discussed above.

Undo Uncommitted Changes

Before we start undoing things, let’s take a look at the status of our repository.

```
git status
```

We have a tracked file and an untracked file that need to be changed. First, we'll take care of `index.html`:

```
git reset --hard
```

This changes all *tracked* files to match the most recent commit. Note that the `--hard` flag is what actually updates the file. Running `git reset` without any flags will simply unstage `index.html`, leaving its contents as is. In either case, `git reset` only operates on the working directory and the staging area, so our `git log` history remains unchanged.

Next, let's remove the `dummy.html` file. Of course, we could manually delete it, but using Git to reset changes eliminates human errors when working with several files in large teams. Run the following command.

```
git clean -f
```

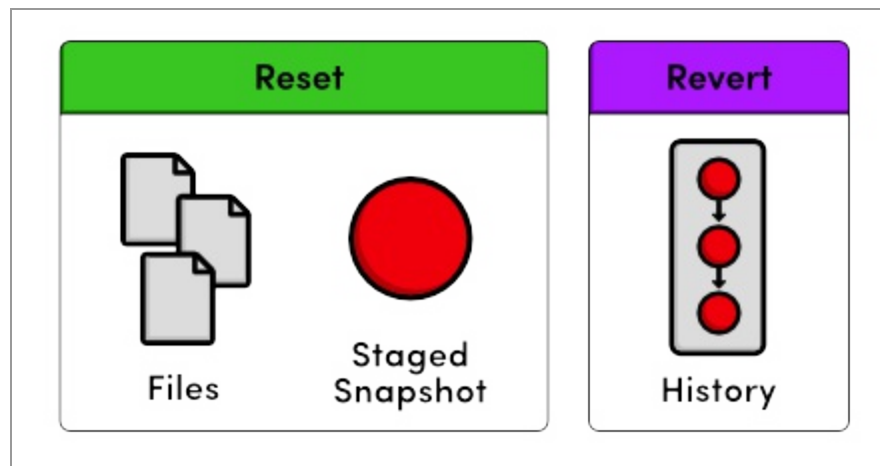
This will remove all *untracked* files. With `dummy.html` gone, `git status` should now tell us that we have a “clean” working directory, meaning our project matches the most recent commit.

Be careful with `git reset` and `git clean`. Both operate on the working directory, not on the committed snapshots. Unlike `git revert`, they ***permanently*** undo changes, so make sure you really want to trash what you're working on before you use them.

Conclusion

As noted in the previous module, most Git commands operate on one of the

three main components of a Git repository: the working directory, the staged snapshot, or the committed snapshots. The `git reset` command undoes changes to the working directory and the staged snapshot, while `git revert` undoes changes contained in committed snapshots. Not surprisingly, `git status` and `git log` directly parallel this behavior.



Resetting vs. Reverting

I mentioned that instead of completely removing a commit, `git revert` saves the commit in case you want to come back to it later. This is only one reason for preserving committed snapshots. When we start working with remote repositories, we'll see that altering the history by removing a commit has dramatic consequences for collaborating with other developers.

This module also introduces the concept of switching between various commits and branches with `git checkout`. Branches round out our discussion of the core Git components, and they offer an elegant option for optimizing your development workflow. In the next module, we'll cover the basic Git branch commands.

Quick Reference

```
git checkout <commit-id>
```

View a previous commit.

```
git tag -a <tag-name> -m "<description>"
```

Create an annotated tag pointing to the most recent commit.

```
git revert <commit-id>
```

Undo the specified commit by applying a new commit.

```
git reset --hard
```

Reset *tracked* files to match the most recent commit.

```
git clean -f
```

Remove *untracked* files.

```
git reset --hard / git clean -f
```

Permanently undo uncommitted changes.

Branches, Part I

Branches are the final component of Git version control. This gives us four core elements to work with throughout the rest of this tutorial:

- The Working Directory
- The Staged Snapshot
- Committed Snapshots
- Development Branches

In Git, a **branch** is an independent line of development. For example, if you wanted to experiment with a new idea *without* using Git, you might copy all of your project files into another directory and start making your changes. If you liked the result, you could copy the affected files back into the original project. Otherwise, you would simply delete the entire experiment and forget about it.

This is the exact functionality offered by Git branches—with some key improvements. First, branches present an error-proof method for incorporating changes from an experiment. Second, they let you store all of your experiments in a single directory, which makes it much easier to keep track of them and to share them with others. Branches also lend themselves to several standardized workflows for both individual and collaborative development, which will be explored in the latter half of the tutorial.

[Download the repository for this module](#)

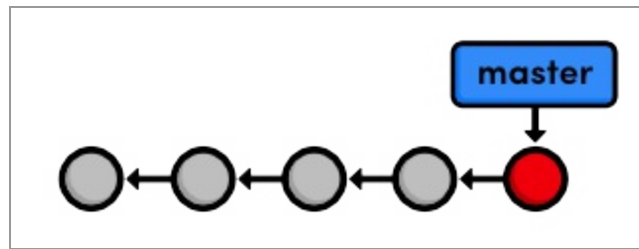
If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repository from the above link, uncompress it, and you're good to go.

View Existing Branches

Let's start our exploration by listing the existing branches for our project.

```
git branch
```

This will display our one and only branch: `* master`. The master branch is Git's default branch, and the asterisk next to it tells us that it's currently checked out. This means that the most recent snapshot in the master branch resides in the working directory:



The master branch

Notice that since there's only one working directory for each project, only one branch can be checked out at a time.

Checkout the Crazy Experiment

The previous module left out some details about how checking out previous commits actually works. We're now ready to tackle this topic in depth. First, we need the checksums of our committed snapshots.

```
git log --oneline
```

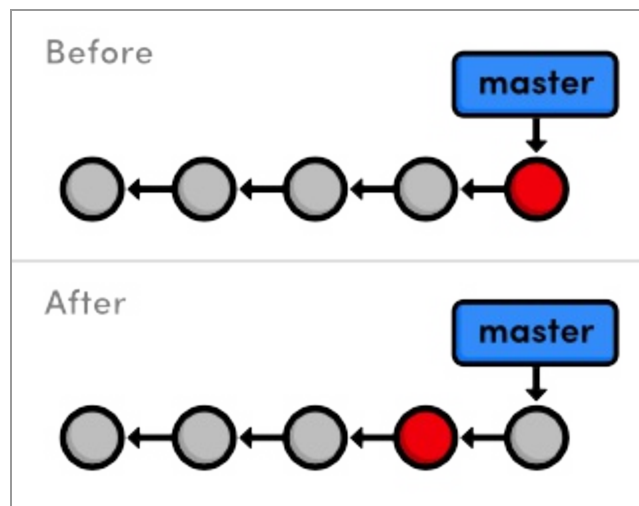
This outputs the following history.

```
506bb9b Revert "Add a crazzzy experiment"
514fbe7 Add a crazzzy experiment
1c310d2 Add navigation links
54650a3 Create blue and orange pages
b650e4b Create index page
```

Check out the crazy experiment from the last module, remembering to change 514fbe7 to the ID of your fourth commit.

```
git checkout 514fbe7
```

This command returns a message that says we're in a detached HEAD state and that the HEAD is now at 514fbe7. The **HEAD** is Git's internal way of indicating the snapshot that is currently checked out. This means the red circle in each of our history diagrams actually represents Git's HEAD. The following figure shows the state of our repository before and after we checked out an old commit.



Checking out the 4th commit

As shown in the “before” diagram, the HEAD normally resides on the tip of a development branch. But when we checked out the previous commit, the HEAD moved to the middle of the branch. We can no longer say we're on the master branch since it contains more recent snapshots than the HEAD. This is reflected in the `git branch` output, which tells us that we're currently on (no branch).

We'll continue developing our crazy experiment by changing `crazy.html` to the following.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>A Crazy Experiment</title>
  <meta charset="utf-8" />
</head>
<body>
  <h1>A Crazy Experiment</h1>
  <p>Look! A Rainbow!</p>

  <ul>
    <li style="color: red">Red</li>
    <li style="color: orange">Orange</li>
    <li style="color: yellow">Yellow</li>
    <li style="color: green">Green</li>
    <li style="color: blue">Blue</li>
    <li style="color: indigo">Indigo</li>
    <li style="color: violet">Violet</li>
  </ul>

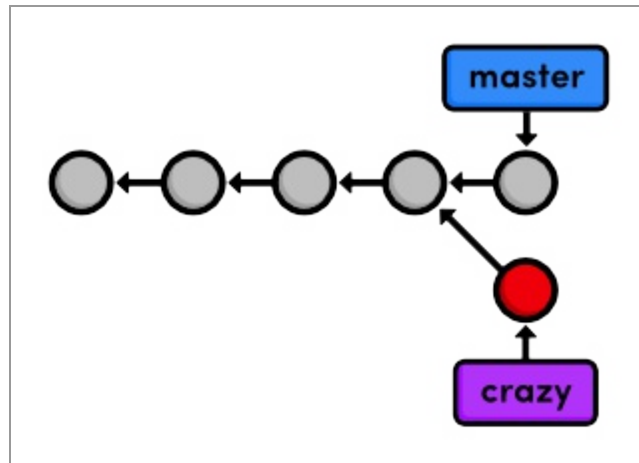
  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Stage and Commit the Rainbow

Hopefully, you're relatively familiar with staging and committing snapshots by now:

```
git add crazy.html
git status
git commit -m "Add a rainbow to crazy.html"
```

After committing on the crazy branch, we can see two independent lines of development in our project:



Forked project history

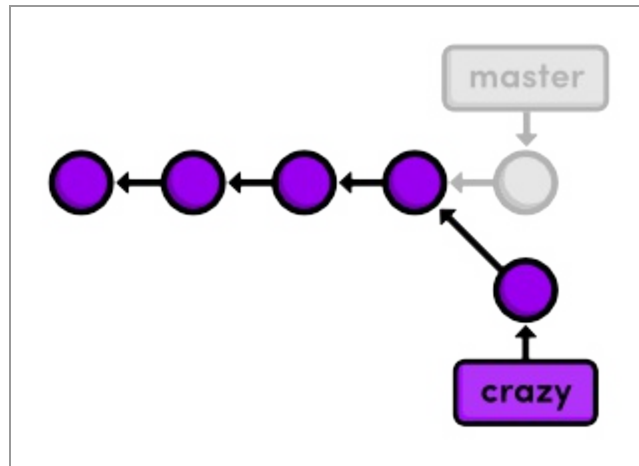
Also notice that the HEAD (designated by the red circle) automatically moved forward to the new commit, which is intuitively what we would expect when developing a project.

The above diagram represents the complete state of our repository, but `git log` only displays the history of the current branch:

```
677e0e0 Add a rainbow to crazy.html
514fbe7 Add a crazzzy experiment
*1c310d2 Add navigation links
*54650a3 Create blue and orange pages
*b650e4b Create index page
```

Note that the history *before* the fork is considered part of the new branch

(marked with asterisks above). That is to say, the crazy history spans all the way back to the first commit:



History of the crazy branch

The project as a whole now has a complex history; however, each individual branch still has a *linear* history (snapshots occur one after another). This means that we can interact with branches in the exact same way as we learned in the first two modules.

Rename the Rainbow

Let's add one more snapshot to the crazy branch. Rename `crazy.html` to `rainbow.html`, then use the following Git commands to update the repository.

```
git status
git rm crazy.html
git status
git add rainbow.html
git status
```

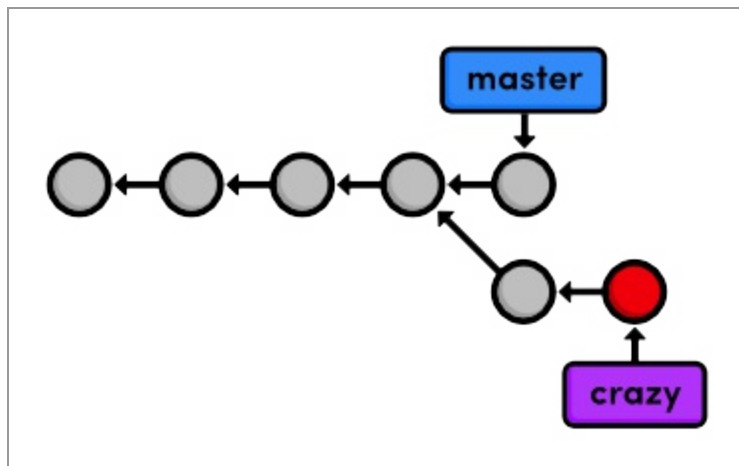
The `git rm` command tells Git to stop tracking `crazy.html` (and delete it if

necessary), and `git add` starts tracking `rainbow.html`. The renamed: `crazy.html -> rainbow.html` message in the final status output shows us that Git is smart enough to figure out when we're renaming a file.

Our snapshot is staged and ready to be committed:

```
git commit -m "Rename crazy.html to rainbow.html"
git log --oneline
```

After this addition, our complete repository history looks like the following. Remember that the `crazy` branch doesn't include any commits in `master` after the fork.



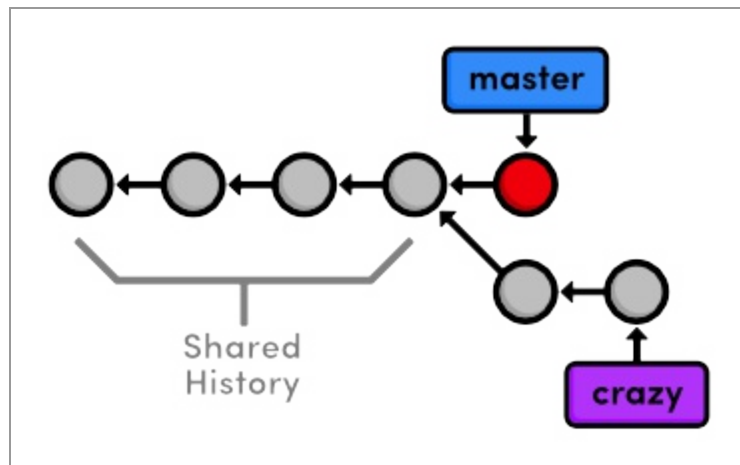
Current project history

Return to the Master Branch

Let's switch back to the `master` branch:

```
git checkout master
git branch
git log --oneline
```

After the checkout, `crazy.html` doesn't exist in the working directory, and the commits from the last few steps don't appear in the history. These two branches became *completely independent* development environments after they forked. You can think of them as separate project folders that you switch between with `git checkout`. They do, however, share their first four commits.



Shared branch history

Create a CSS Branch

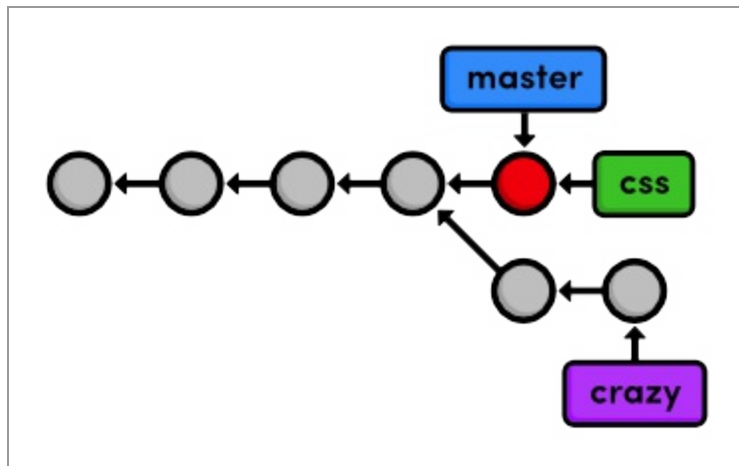
We're going to put our crazy experiment on the backburner for now and turn our attention to formatting the HTML pages with a cascading stylesheet (CSS). Again, if you're not all that comfortable with HTML and CSS, the content of the upcoming files isn't nearly as important as the Git commands used to manage them.

Let's create and check out a new branch called `css`.

```
git branch css
git checkout css
```

The new branch points to the currently checked out snapshot, which happens

to coincide with the master branch:



Creating the css branch

Add a CSS Stylesheet

Next, create a file called `style.css` with the following content. This CSS is used to apply formatting to the HTML in our other files.

```
body {  
  padding: 20px;  
  font-family: Verdana, Arial, Helvetica, sans-serif;  
  font-size: 14px;  
  color: #111;  
}  
  
p, ul {  
  margin-bottom: 10px;  
}  
  
ul {  
  margin-left: 20px;  
}
```

Commit the stylesheet in the usual fashion.

```
git add style.css
git status
git commit -m "Add CSS stylesheet"
```

Link the Stylesheet

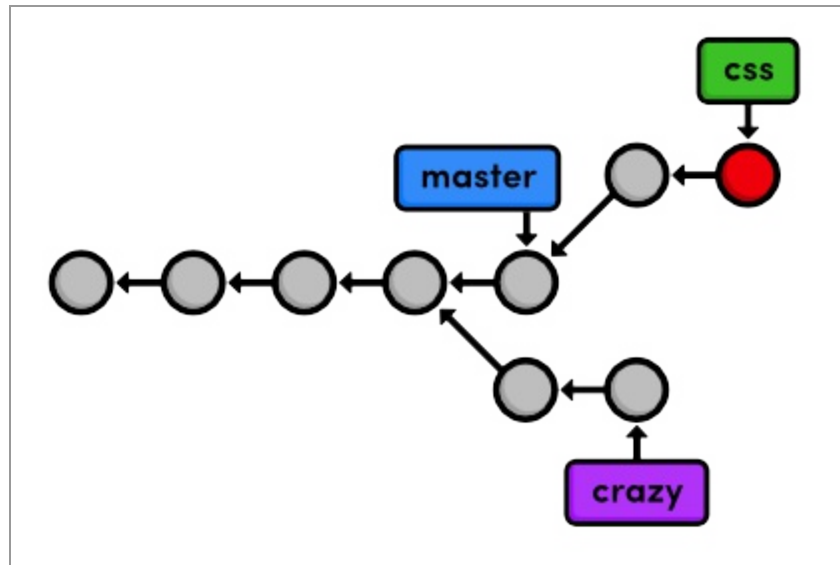
We still need to tell the HTML pages to use the formatting in `style.css`. Add the following text on a separate line after the `<title>` element in `index.html`, `blue.html`, and `orange.html` (remember that `rainbow.html` only exists in the crazy branch). You should be able to see the CSS formatting by opening `index.html` in a web browser.

```
<link rel="stylesheet" href="style.css" />
```

Commit the changes.

```
git add index.html blue.html orange.html
git status
git commit -m "Link HTML pages to stylesheet"
git log --oneline
```

This results in a repository history that looks like:

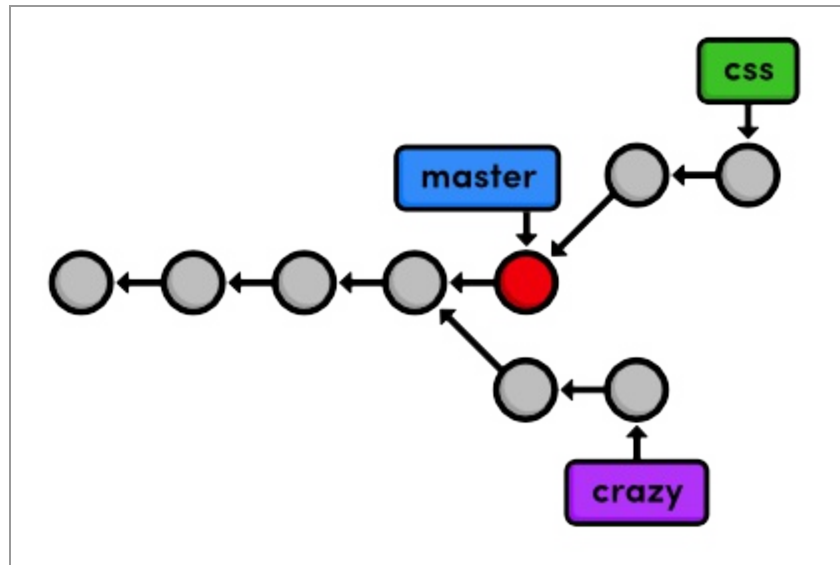


Return to the Master Branch (Again)

The `css` branch let us create and test our formatting without threatening the stability of the `master` branch. But, now we need to merge these changes into the main project. Before we attempt the merge, we need to return to the `master` branch.

```
git checkout master
```

Verify that `style.css` doesn't exist and that HTML pages aren't linked to it. Our repository history remains unchanged, but the working directory now matches the snapshot pointed to by the master branch.



Current project history

Take a look at the `git log --oneline` output as well.

```
af23ff4 Revert "Add a crazzzy experiment"
a50819f Add a crazzzy experiment
4cd95d9 Add navigation links
dcb9e07 Create blue and orange pages
f757eb3 Create index page
```

As expected, there is no mention of the CSS additions in the history of master, but we're about to change that.

Merge the CSS Branch

Use the `git merge` command to take the snapshots from the `css` branch and add them to the master branch.

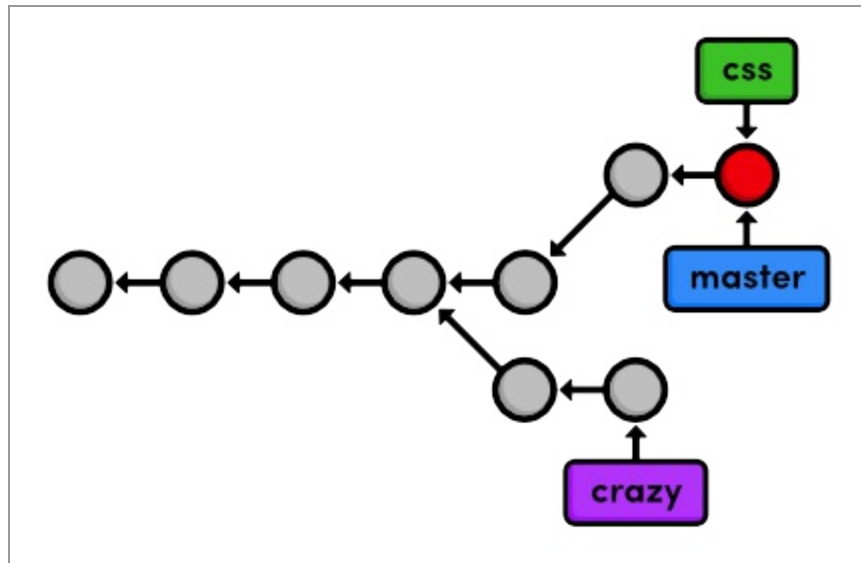
```
git merge css
```

Notice that this command always merges into the current branch: `css` remains

unchanged. Check the history to make sure that the css history has been added to master.

```
git log --oneline
```

The following diagram visualizes the merge.



Merging the css branch into master

Instead of re-creating the commits in `css` and adding them to the history of `master`, Git reuses the existing snapshots and simply moves the tip of `master` to match the tip of `css`. This kind of merge is called a **fast-forward merge**, since Git is “fast-forwarding” through the new commits in the `css` branch.

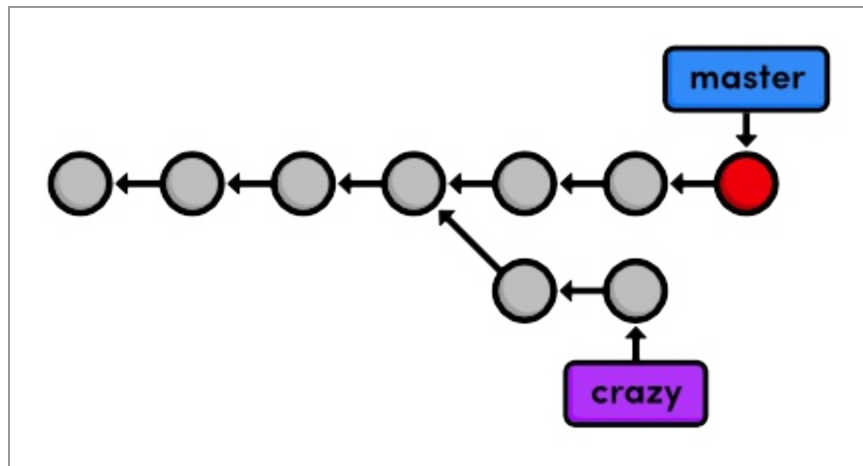
After the merge, both branches have the exact same history, which makes them redundant. Unless we wanted to keep developing on the `css` branch, we’re free to get rid of it.

Delete the CSS Branch

We can safely delete a branch by passing the `-d` flag to `git branch`.

```
git branch -d css  
git branch
```

Since `css` and `master` represent the same branch, our history looks the same, though the `css` branch has been removed. I've also put the `master` branch's commits in a straight line in the following visualization, making it easier to track during the upcoming modules.



Deleting the `css` branch

Deleting branches is a relatively “safe” operation in the sense that Git will warn you if you’re deleting an unmerged branch. This is just another example of Git’s commitment to never losing your work.

Conclusion

This module used two branches to experiment with new additions. In both cases, branches gave us an environment that was completely isolated from the “stable” version of our website (the `master` branch). One of our experiments is waiting for us in the next module, while our CSS changes have been merged into the stable project, and its branch is thus obsolete. Using branches to develop small features like these is one of the hallmarks of Git-based software management.

While this module relied heavily on branch diagrams to show the complete state of the repository, you don't need to keep this high-level overview in mind during your everyday development. Creating a new branch is really just a way to request an independent working directory, staging snapshot, and history. You can think of branches as a way to multiply the functionality presented in the first two module.

Next, we'll practice our branch management skills by examining the typical workflow of veteran Git users. We'll also discover more complicated merges than the fast-forward merge introduced above.

Quick Reference

`git branch`

List all branches.

`git branch <branch-name>`

Create a new branch using the current working directory as its base.

`git checkout <branch-name>`

Make the working directory and the HEAD match the specified branch.

`git merge <branch-name>`

Merge a branch into the checked-out branch.

`git branch -d <branch-name>`

Delete a branch.

`git rm <file>`

Remove a file from the working directory (if applicable) and stop tracking the file.

Branches, Part II

Now that we've covered the mechanics behind Git branches, we can discuss the practical impact that they have on the software development process. Instead of introducing new commands, this module covers how the typical Git user applies this workflow to real projects, as well as some of the problems that arise in a branched environment.

To Git, a branch is a branch, but it's often useful to assign special meaning to different branches. For example, we've been using `master` as the stable branch for our example project, and we've also used a temporary branch to add some CSS formatting. Temporary branches like the latter are called **topic branches** because they exist to develop a certain topic, then they are deleted. We'll work with two types of topic branches later in this module.

Amid our exploration of Git branches, we'll also discover that some merges cannot be "fast-forwarded." When the history of two branches diverges, a dedicated commit is required to combine the branches. This situation may also give rise to a merge conflict, which must be manually resolved before anything can be committed to the repository.

[Download the repository for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repository from the above link, uncompress it, and you're good to go.

Continue the Crazy Experiment

Let's start by checking out the crazy branch.

```
git branch  
git checkout crazy
```

```
git log --oneline
```

The crazy branch is a longer-running type of topic branch called a **feature branch**. This is fitting, as it was created with the intention of developing a specific *feature*. It's also a term that makes Git's contribution to the development workflow readily apparent: branches enable you to focus on developing one clearly defined feature at a time.

This brings us to my rule-of-thumb for using Git branches:

- Create a new branch for each major addition to your project.
- *Don't* create a branch if you can't give it a specific name.

Following these simple guidelines will have a dramatic impact on your programming efficiency.

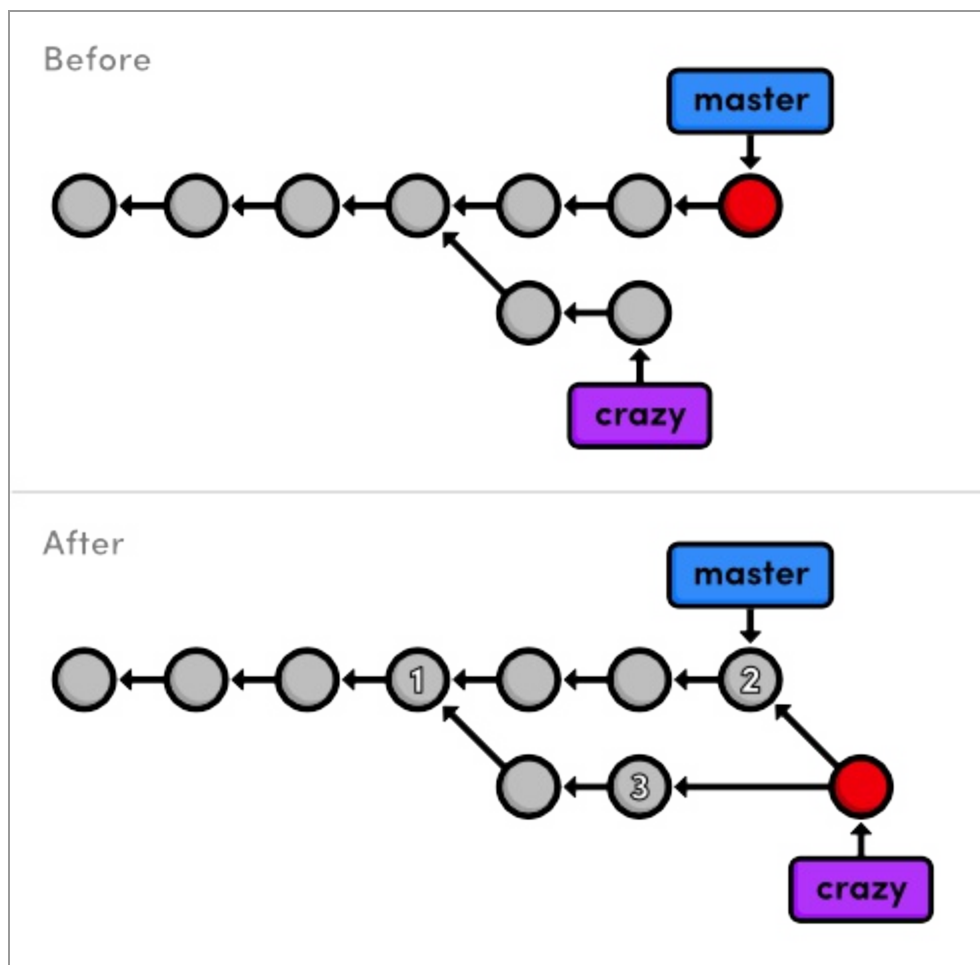
Merge the CSS Updates

Note that the CSS formatting we merged into master is nowhere to be found. This presents a bit of a problem if we want our experiment to reflect these updates. Conveniently, Git lets us merge changes into *any* branch (not just the master branch). So, we can pull the updates in with the familiar `git merge` command. Remember that merging only affects the checked-out branch.

```
git merge master  
git log --oneline
```

As of Git 1.7.10, this will open your editor and prompt you for a message explaining why the commit was necessary. You can use the default Merge branch 'master' into crazy. When you save and close the file, you'll notice an extra commit in your project history. Recall that our first merge

didn't add any new commits; it just “fast-forwarded” the tip of the master branch. This was not the case for our new merge, which is shown below.



Merging master into the crazy branch

Take a moment to examine why the current merge couldn't be a fast-forward one. How could Git have walked the crazy pointer over to the tip of the master branch? It's not possible without backtracking, which kind of defeats the idea of “fast-forwarding.” We're left with a new way to combine branches: the **3-way merge**.

A 3-way merge occurs when you try to merge two branches whose history has diverged. It creates an extra **merge commit** to function as a link between the two branches. As a result, it has *two* parent commits. The above figure

visualizes this with two arrows originating from the tip of *crazy*. It's like saying "this commit comes from both the *crazy* branch *and* from *master*." After the merge, the *crazy* branch has access to both its history and the *master* history.

The name comes from the internal method used to create the merge commit. Git looks at *three* commits (numbered in the above figure) to generate the final state of the merge.

This kind of branch interaction is a big part of what makes Git such a powerful development tool. We can not only create independent lines of development, but we can also share information between them by tying together their histories with a 3-way merge.

Style the Rainbow Page

Now that we have access to the CSS updates from *master*, we can continue developing our *crazy* experiment. Link the CSS stylesheet to `rainbow.html` by adding the following HTML on the line after the `<title>` element.

```
<link rel="stylesheet" href="style.css" />
```

Stage and commit the update, then check that it's reflected in the history.

```
git status
git commit -a -m "Add CSS stylesheet to rainbow.html"
git log --oneline
```

Notice that we skipped the staging step this time around. Instead of using `git add`, we passed the `-a` flag to `git commit`. This convenient parameter tells Git to automatically include *all* tracked files in the staged snapshot. Combined with the `-m` flag, we can stage and commit snapshots with a single command.

However, be careful not to include unintended files when using the `-a` flag.

Link to the Rainbow Page

We still need to add a navigation link to the home page. Change the “Navigation” section of `index.html` to the following.

```
<h2>Navigation</h2>
<ul>
  <li style="color: #F90">
    <a href="orange.html">The Orange Page</a>
  </li>
  <li style="color: #00F">
    <a href="blue.html">The Blue Page</a>
  </li>
  <li>
    <a href="rainbow.html">The Rainbow Page</a>
  </li>
</ul>
```

As usual, stage and commit the snapshot.

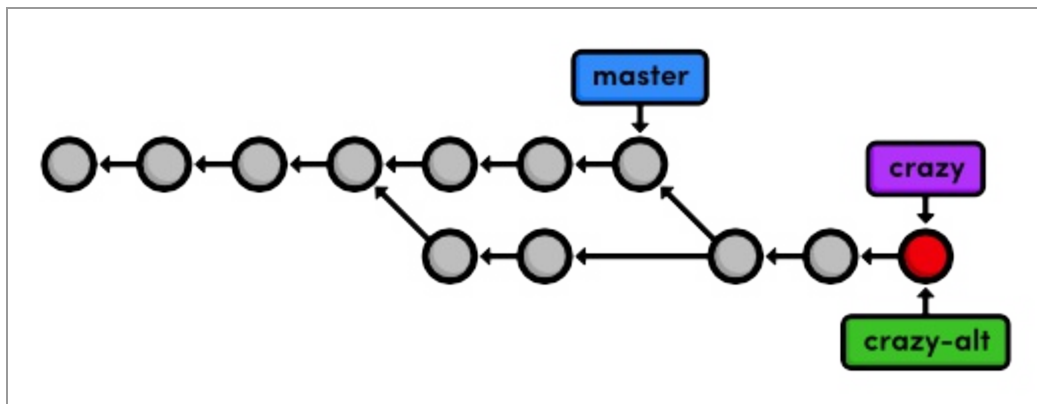
```
git commit -a -m "Link index.html to rainbow.html"
git log --oneline
```

Fork an Alternative Rainbow

Next, we’re going to brainstorm an alternative to the current `rainbow.html` page. This is a perfect time to create another topic branch:

```
git branch crazy-alt
git checkout crazy-alt
```

Remember, we can do whatever we want here without worrying about either crazy or master. When `git branch` creates a branch, it uses the current HEAD as the starting point for the new branch. This means that we begin with the same files as crazy (if we called `git branch` from master, we would have had to re-create `rainbow.html`). After creating the new branch, our repository's history looks like:



Creating the crazy-alt branch

Change the Rainbow

Change the colorful list in `rainbow.html` from:

```
<ul>
  <li style="color: red">Red</li>
  <li style="color: orange">Orange</li>
  <li style="color: yellow">Yellow</li>
  <li style="color: green">Green</li>
  <li style="color: blue">Blue</li>
  <li style="color: indigo">Indigo</li>
  <li style="color: violet">Violet</li>
</ul>
```

to the following:

```
<div style="background-color: red"></div>
<div style="background-color: orange"></div>
<div style="background-color: yellow"></div>
<div style="background-color: green"></div>
<div style="background-color: blue"></div>
<div style="background-color: indigo"></div>
<div style="background-color: violet"></div>
```

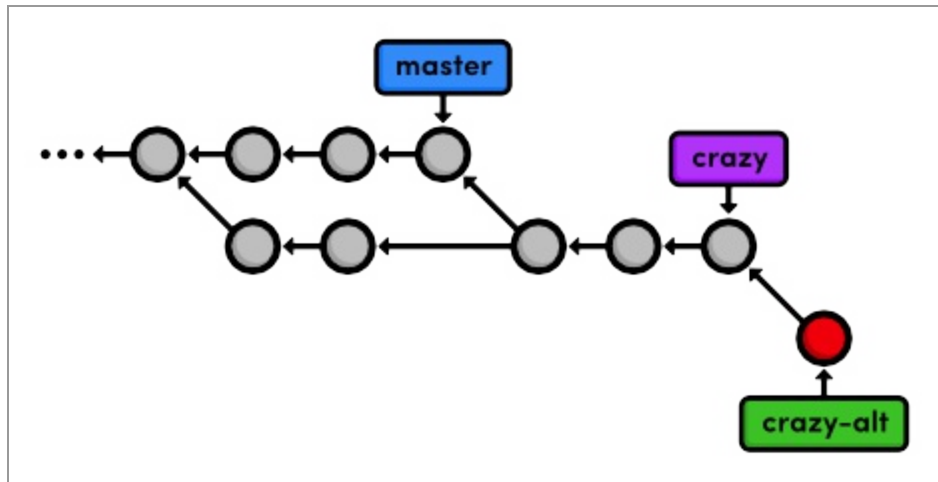
Then, add some CSS formatting to `<head>` on the line after the `<meta>` element:

```
<style>
  div {
    width: 300px;
    height: 50px;
  }
</style>
```

If you open `rainbow.html` in a browser, you should now see colorful blocks in place of the colorful text. Don't forget to commit the changes:

```
git commit -a -m "Make a REAL rainbow"
```

The resulting project history is shown below, with the first four commits omitted for the sake of presentation.



Committing on the crazy-alt branch

Emergency Update!

Our boss called in with some breaking news! He needs us to update the site immediately, but what do we do with our `rainbow.html` developments? Well, the beauty of Git branches is that we can just leave them where they are and add the breaking news to master.

We'll use what's called a **hotfix branch** to create and test the news updates. In contrast to our relatively long-running feature branch (`crazy`), hotfix branches are used to quickly patch a production release. For example, you'd use a hotfix branch to fix a time-sensitive bug in a public software project. This distinction is useful for demonstrating when it's appropriate to create a new branch, but it is purely conceptual—a branch is a branch according to Git.

```
git checkout master
git branch news-hotfix
git checkout news-hotfix
```

Change the “News” list in `index.html` to match the following.

```
<h2 style="color: #C00">News</h2>
<ul>
  <li><a href="news-1.html">Blue Is The New Hue</a></li>
</ul>
```

And, create a new HTML page called `news-1.html` with the following content.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Blue Is The New Hue</title>
  <link rel="stylesheet" href="style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #079">Blue Is The New Hue</h1>
  <p>European designers have just announced that
  <span style="color: #079">Blue</span> will be this year's
  hot color.</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

We can't use `git commit -a` to automatically stage `news-1.html` because it's an *untracked* file (as shown in `git status`). So, let's use an explicit `git add`:

```
git add index.html news-1.html
git status
git commit -m "Add 1st news item"
```

Test these additions in a browser to make sure that the links work, it's typo free, etc. If everything looks good, we can “publish” the changes by merging them into the stable master branch. Isolating this in a separate branch isn't really necessary for our trivial example, but in the real world, this would give you the opportunity to run build tests without touching your stable project.

Publish the News Hotfix

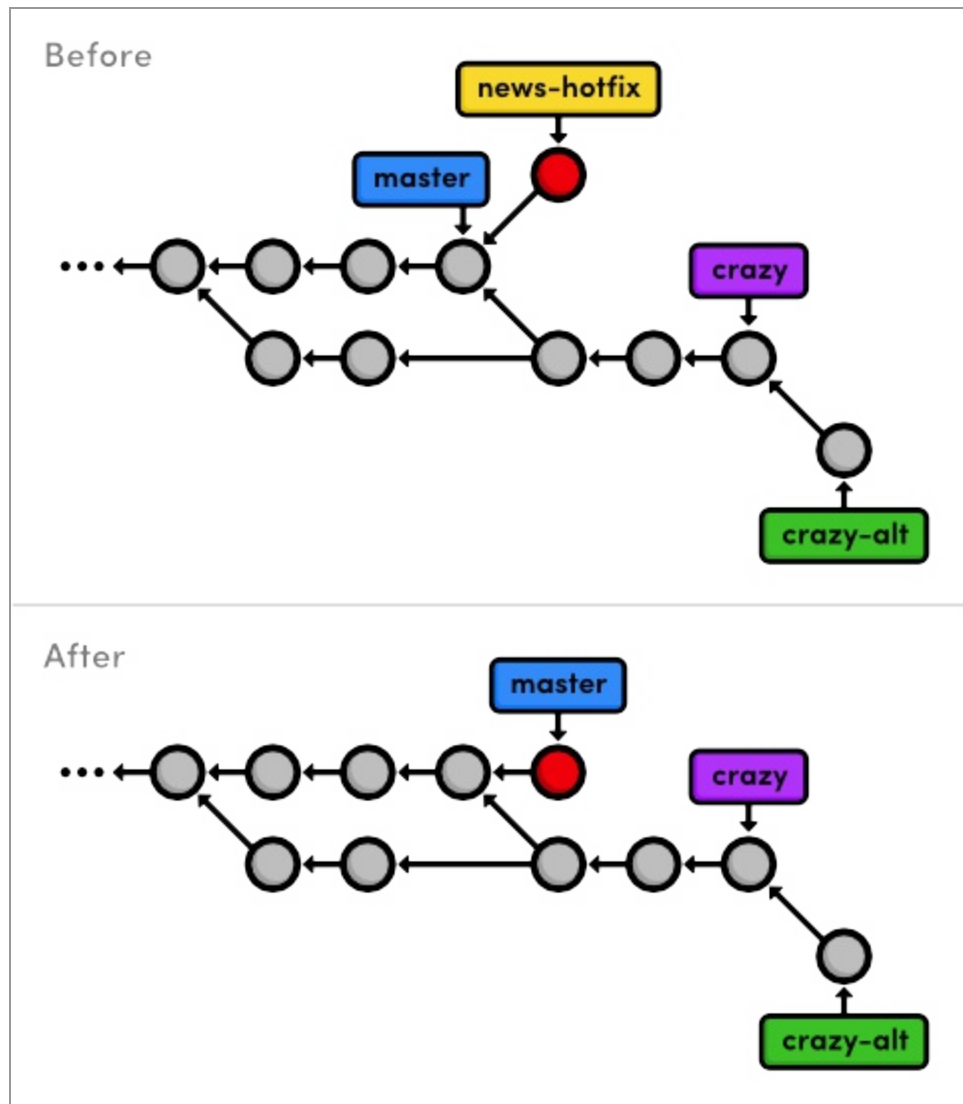
Remember that to merge into the master branch, we first need to check it out.

```
git checkout master  
git merge news-hotfix
```

Since master now contains the news update, we can delete the hotfix branch:

```
git branch -d news-hotfix  
git branch
```

The following diagram reflects our repository's history before and after the merge. Can you figure out why was this a fast-forward merge instead of a 3-way merge?



Fast-forwarding master to the news-hotfix branch

Also notice that we have another fork in our history (the commit before master branches in two directions), which means we should expect to see another merge commit in the near future.

Complete the Crazy Experiment

Ok, let's finish up our crazy experiment with one more commit.

```
git checkout crazy
```


Note that the news article is nowhere to be found, as should be expected (this branch is a completely isolated development environment).

We'll finish up our crazy experiment by adding a news item for it on the home page. Change the news list in `index.html` to the following:

```
<h2 style="color: #C00">News</h2>
<ul>
  <li><a href="rainbow.html">Our New Rainbow</a></li>
</ul>
```

Astute readers have probably observed that this directly conflicts with what we changed in the `news-hotfix` branch. We should *not* manually add in the other news item because it has no relationship with the current branch. In addition, there would be no way to make sure the link works because `news-1.html` doesn't exist in this branch. This may seem trivial, but imagine the errors that could be introduced had `news-hotfix` made dozens of different changes.

We'll simply stage and commit the snapshot as if there were no conflicts:

```
git commit -a -m "Add news item for rainbow"
git log --oneline
```

Look at all those experimental commits (marked with asterisks below)!

```
*42fa173 Add news item for rainbow
*7147cc5 Link index.html to rainbow.html
*6aa4b3b Add CSS stylesheet to rainbow.html
b9ae1bc Merge branch 'master' into crazy
ae4e756 Link HTML pages to stylesheet
98cd46d Add CSS stylesheet
```

```
*33e25c9 Rename crazy.html to rainbow.html
*677e0e0 Add a rainbow to crazy.html
506bb9b Revert "Add a crazzzy experiment"
*514fbe7 Add a crazzzy experiment
1c310d2 Add navigation links
54650a3 Create blue and orange pages
b650e4b Create index page
```

Publish the Crazy Experiment

We're finally ready to merge our crazy branch back into master.

```
git checkout master
git merge crazy
```

You should get a message that reads:

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

This is our first **merge conflict**. Conflicts occur when we try to merge branches that have edited the same content. Git doesn't know how to combine the two changes, so it stops to ask us what to do. We can see exactly what went wrong with the familiar `git status` command:

```
# On branch master
# Changes to be committed:
#
#       new file:   rainbow.html
#
# Unmerged paths:
```

```
# (use "git add/rm <file>..." as appropriate to mark resolution)
#
#      both modified:      index.html
#
```

We’re looking at the staged snapshot of a merge commit. We never saw this with the first 3-way merge because we didn’t have any conflicts to resolve. But now, Git stopped to let us modify files and resolve the conflict before committing the snapshot. The “Unmerged paths” section contains files that have a conflict.

Open up `index.html` and find the section that looks like:

```
<<<<<<< HEAD
    <li><a href="news-1.html">Blue Is The New Hue</a></li>
=====
    <li><a href="rainbow.html">Our New Rainbow</a></li>
>>>>>>> crazy
```

Git went ahead and modified the conflicted file to show us exactly which lines are afflicted. The format of the above text shows us the difference between the two versions of the file. The section labeled `<<<<<<< HEAD` shows us the version in the current branch, while the part after the `=====` shows the version in the crazy branch.

Resolve the Merge Conflicts

We can change the affected lines to whatever we want in order to resolve the conflict. Edit the news section of `index.html` to keep changes from both versions:

```
<h2 style="color: #C00">News</h2>
```

```
<ul>
  <li><a href="news-1.html">Blue Is The New Hue</a></li>
  <li><a href="rainbow.html">Our New Rainbow</a></li>
</ul>
```

The <<<<<<, =====, and >>>>>> markers are only used to show us the conflict and should be deleted. Next, we need to tell Git that we’re done resolving the conflict:

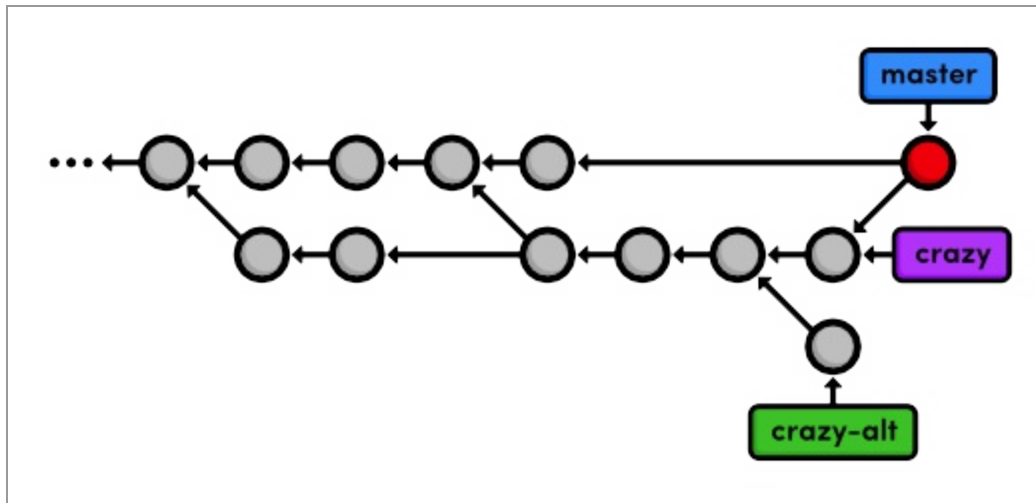
```
git add index.html
git status
```

That’s right, all you have to do is add `index.html` to the staged snapshot to mark it as resolved. Finally, complete the 3-way merge:

```
git commit
```

We didn’t use the `-m` flag to specify a message because Git already gives us a default message for merge commits. It also gives us a “Conflicts” list, which can be particularly handy when trying to figure out where something went wrong in a project. Save and close the file to create the merge commit.

The final state of our project looks like the following.



Merging the crazy branch into master

Cleanup the Feature Branches

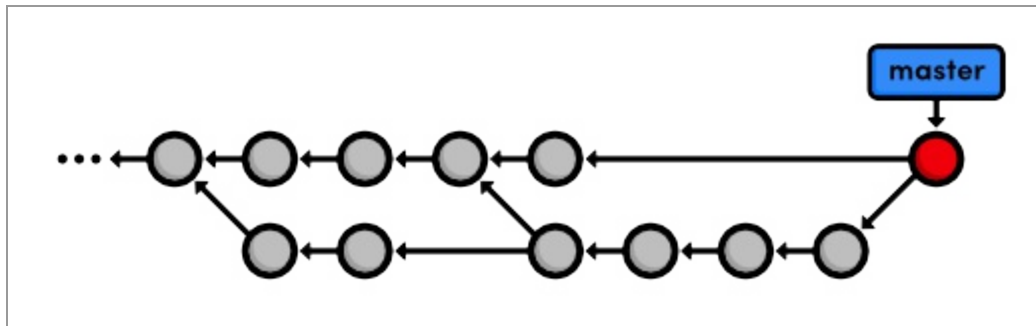
Since our crazy experiment has been successfully merged, we can get rid of our feature branches.

```
git branch -d crazy  
git branch -d crazy-alt
```

As noted in the last module, the `git branch -d` command won't let you delete a branch that contains unmerged changes. But, we really do want to scrap the alternative experiment, so we'll follow the error message's instructions for overriding this behavior:

```
git branch -D crazy-alt
```

Because we never merged `crazy-alt` into `master`, it is lost forever. However, the `crazy` branch is still accessible through its commits, which are now reachable via the `master` branch. That is to say, it is still part of the structure of the repository's history, even though we deleted our reference to it.



Deleting the feature branches

Fast-forward merges are *not* reflected in the project history. This is the tangible distinction between fast-forward merges and 3-way merges. The next module will discuss the appropriate usage of both and the potential complications of a non-linear history.

Conclusion

This module demonstrated the three most common uses of Git branches:

- To develop long-running features (crazy)
- To apply quick updates (news-hotfix)
- To record the evolution of a project (master)

In the first two cases, we needed an *isolated* environment to test some changes before integrating them with our stable code. As you get more comfortable with Git, you should find yourself doing virtually everything in an isolated topic branch, then merging it into a stable branch once you're done. Practically, this means you'll never have a broken version of your project.

We used the permanent master branch as the foundation for all of these temporary branches, effectively making it the historian for our entire project. In addition to master, many programmers often add a second permanent branch called `develop`. This lets them use master to record *really* stable

snapshots (e.g., public releases) and use `develop` as more of a preparation area for `master`.

This module also introduced the 3-way merge, which combines two branches using a dedicated commit. The 3-way merge and the fast-forward merge are actually what makes branching so powerful: they let developers share and integrate independent updates with reliable results.

Next, we'll learn how to clean up our repository's history. Using a new Git command, we'll be able to better manage merge commits and make sure our history is easy to navigate.

Quick Reference

```
git commit -a -m "<message>"
```

Stage all tracked files and commit the snapshot using the specified message.

```
git branch -D <branch-name>
```

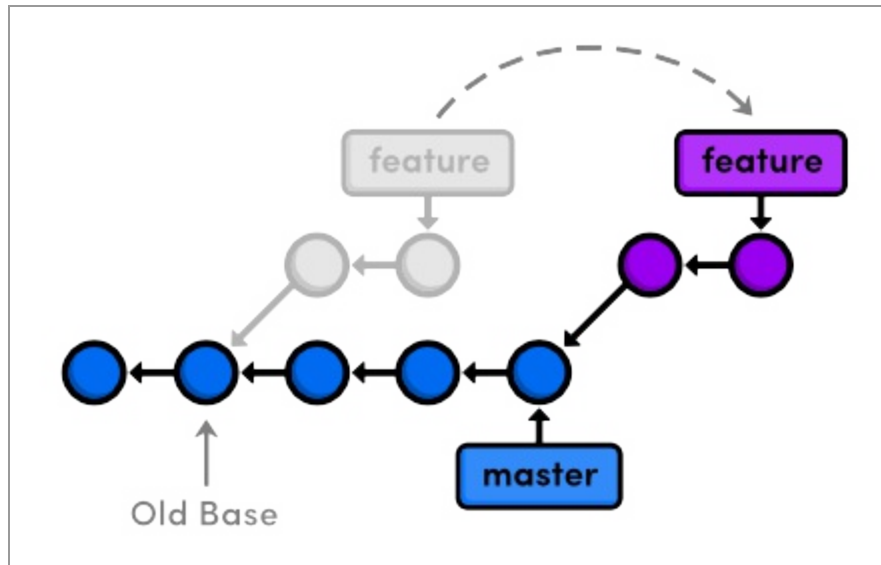
Force the removal of an unmerged branch (*be careful*: it will be lost forever).

Rebasing

Let's start this module by taking an in-depth look at our history. The six commits asterisked below are part of the same train of thought. We even developed them in their own feature branch. However, they show up interspersed with commits from other branches, along with a superfluous merge commit (b9ae1bc). In other words, our repository's history is kind of messy:

```
ec1b8cb Merge branch 'crazy'
*42fa173 Add news item for rainbow
3db88e1 Add 1st news item
*7147cc5 Link index.html to rainbow.html
*6aa4b3b Add CSS stylesheet to rainbow.html
b9ae1bc Merge branch 'master' into crazy
ae4e756 Link HTML pages to stylesheet
98cd46d Add CSS stylesheet
*33e25c9 Rename crazy.html to rainbow.html
*677e0e0 Add a rainbow to crazy.html
506bb9b Revert "Add a crazzzy experiment"
*514fbe7 Add a crazzzy experiment
1c310d2 Add navigation links
54650a3 Create blue and orange pages
b650e4b Create index page
```

Fortunately, Git includes a tool to help us clean up our commits: `git rebase`. Rebasing lets us move branches around by changing the commit that they are *based* on. Conceptually, this is what it allows us to do:



Rebasing a feature branch onto master

After rebasing, the feature branch has a new parent commit, which is the same commit pointed to by master. Instead of joining the branches with a merge commit, rebasing integrates the feature branch by building *on top of* master. The result is a perfectly linear history that reads more like a story than the hodgepodge of unrelated edits shown above.

To explore Git's rebasing capabilities, we'll need to build up our example project so that we have something to work with. Then, we'll go back and rewrite history using `git rebase`.

[Download the repository for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repository from the above link, uncompress it, and you're good to go.

Create an About Section

We'll begin by creating an about page for the website. Remember, we should be doing all of our work in isolated branches so that we don't cause any

unintended changes to the stable version of the project.

```
git branch about
git checkout about
```

The next few steps break this feature into several unnecessarily small commits so that we can see the effects of a rebase. First, make a new directory in my-git-repo called about. Then, create the empty file about/index.html. Stage and commit a snapshot.

```
git add about
git status
git commit -m "Add empty page in about section"
```

Note that `git add` can also add entire directories to the staging area.

Add an About Page

Next, we'll add some HTML to about/index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>About Us</title>
  <link rel="stylesheet" href="../style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1>About Us</h1>
  <p>We're a small, colorful website with just two employees:</p>

  <ul>
```

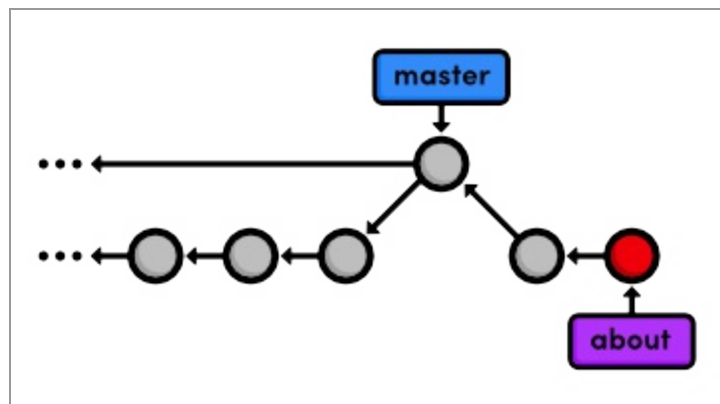
```
<li><a href="me.html">Me: The Developer</a></li>
<li><a href="mary.html">Mary: The Graphic Designer</a></li>
</ul>

<p><a href="../index.html">Return to home page</a></p>
</body>
</html>
```

Stage and commit the snapshot.

```
git status
git commit -a -m "Add contents to about page"
```

After a few commits on this branch, our history looks like the following.



Adding the about branch

Another Emergency Update!

Our boss just gave us some more breaking news! Again, we'll use a hotfix branch to update the site without affecting our about page developments. Make sure to base the updates on master, not the about branch:

```
git checkout master
```

```
git branch news-hotfix
git checkout news-hotfix
git branch
```

Change the “News” section in `index.html` to:

```
<h2 style="color: #C00">News</h2>
<ul>
  <li><a href="news-1.html">Blue Is The New Hue</a></li>
  <li><a href="rainbow.html">Our New Rainbow</a></li>
  <li><a href="news-2.html">A Red Rebellion</a></li>
</ul>
```

Commit a snapshot:

```
git status
git commit -a -m "Add 2nd news item to index page"
```

Then, create a new page called `news-2.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>A Red Rebellion</title>
  <link rel="stylesheet" href="style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #C03">A Red Rebellion</h1>

  <p>Earlier today, several American design firms
  announced that they have completely rejected the use
```

of blue in any commercial ventures. They have
opted instead for `Red.</p>`

```
<p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Stage and commit another snapshot:

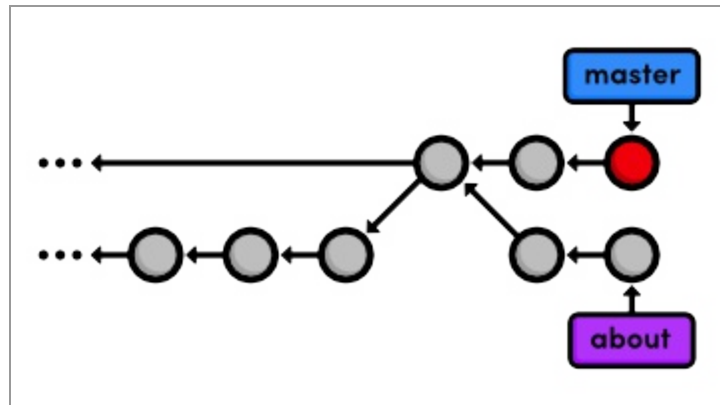
```
git add news-2.html
git status
git commit -m "Add article for 2nd news item"
```

Publish News Hotfix

We're ready to merge the news update back into master.

```
git checkout master
git merge news-hotfix
git branch -d news-hotfix
```

The master branch hasn't been altered since we created news-hotfix, so Git can perform a fast-forward merge. Our repository now looks like the following.



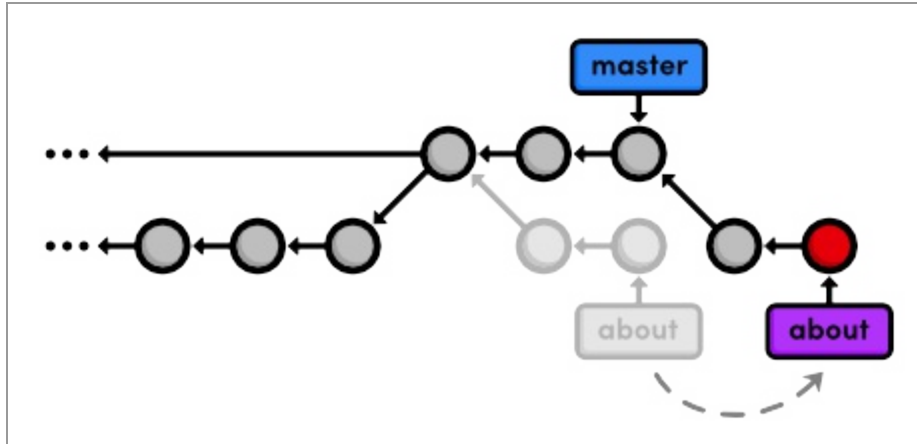
Fast-forwarding master to the news-hotfix

Rebase the About Branch

This puts us in the exact same position as we were in before our first 3-way merge. We want to pull changes from master into a feature branch, only this time we'll do it with a rebase instead of a merge.

```
git checkout about
git rebase master
git log --oneline
```

Originally, the about branch was based on the Merge branch 'crazy-experiment' commit. The rebase took the entire about branch and plopped it onto the *tip* of the master branch, which is visualized in the following diagram. Also notice that, like the `git merge` command, `git rebase` requires you to be on the branch that you want to move.



Rebasing the about branch onto master

After the rebase, about is a linear extension of the master branch, enabling us to do a fast-forward merge later on. Rebasing also allowed us to integrate the most up-to-date version of master *without a merge commit*.

Add a Personal Bio

With our news hotfix out of the way, we can now continue work on our about section. Create the file `about/me.html` with the following contents:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>About Me</title>
  <link rel="stylesheet" href="../style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1>About Me</h1>
  <p>I'm a big nerd.</p>

  <h2>Interests</h2>
```

```
<ul>
  <li>Computers</li>
  <li>Mathematics</li>
  <li>Typography</li>
</ul>

<p><a href="index.html">Return to about page</a></p>
</body>
</html>
```

Then, commit the changes to the repository.

```
git add about/me.html
git commit -m "Add HTML page for personal bio"
git log --oneline
```

Remember that thanks to the rebase, about rests on top of master. So, all of our about section commits are grouped together, which would not be the case had we merged instead of rebased. This also eliminates an unnecessary fork in our project history.

Add Dummy Page for Mary

Once again, the next two snapshots are unnecessarily trivial. However, we'll use an *interactive* rebase to combine them into a single commit later on. That's right, `git rebase` not only lets you move branches around, it enables you to manipulate individual commits as you do so.

Create a new empty file in the about section: `about/mary.html`.

```
git add about
git status
```



```
git commit -m "Add empty HTML page for Mary's bio"
```

Link to the About Section

Then, add a link to the about page in `index.html` so that its “Navigation” section looks like the following.

```
<h2>Navigation</h2>
<ul>
  <li>
    <a href="about/index.html">About Us</a>
  </li>
  <li style="color: #F90">
    <a href="orange.html">The Orange Page</a>
  </li>
  <li style="color: #00F">
    <a href="blue.html">The Blue Page</a>
  </li>
  <li>
    <a href="rainbow.html">The Rainbow Page</a>
  </li>
</ul>
```

Don’t forget to commit the change:

```
git commit -a -m "Add link to about section in home page"
```

Clean Up the Commit History

Before we merge into the master branch, we should make sure we have a clean, meaningful history in our feature branch. By rebasing interactively, we

can choose *how* each commit is transferred to the new base. Specify an interactive rebase by passing the `-i` flag to the rebase command:

```
git rebase -i master
```

This should open up a text editor populated with all of the commits introduced in the `about` branch, listed from oldest to newest. The listing defines exactly how Git will transfer the commits to the new base. Leaving it as is will do a normal `git rebase`, but if we move the lines around, we can change the order in which commits are applied.

In addition, we can replace the `pick` command before each line to edit it or combine it with other commits. All of the available commands are shown in the comment section of the rebase listing, but right now, we only need the `squash` command. This will condense our unnecessarily small commits into a single, meaningful snapshot. Change your listing to match the following:

```
pick 5cf316e Add empty page in about section
squash 964e013 Add contents to about page
pick 89db9ab Add HTML page for personal bio
squash 2bda8e5 Add empty HTML page for Mary's bio
pick 915466f Add link to about section in home page
```

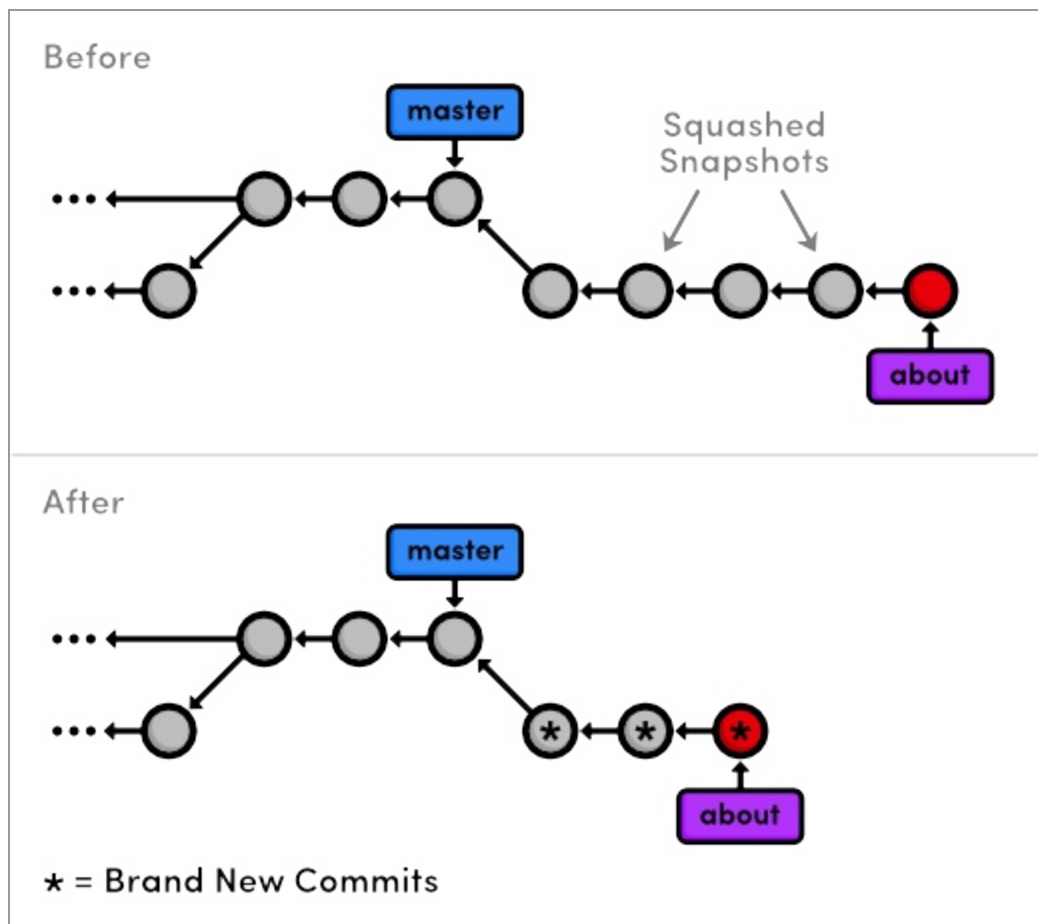
Then, begin the rebase by saving and closing the editor. The following list describes the rebasing process in-depth and tells you what you need to change along the way.

1. Git moves the `5cf316e` commit to the tip of `master`.
2. Git combines the snapshots of `964e013` and `5cf316e`.
3. Git stops to ask you what commit message to use for the combined snapshot. It automatically includes the messages of both commits, but you can delete that and simplify it to just `Create the about page`. Save

and exit the text editor to continue.

4. Git repeats this process for commits 89db9ab and 2bda8e5. Use `Begin` creating `bio` pages for the message.
5. Git adds the final commit (915466f) on top of the commits created in the previous steps.

You can see the result of all this activity with `git log --oneline`, as well as in the diagram below. The five commits originally in `about` have been condensed to three, and two of them have new messages. Also notice that they all have different commit ID's. These new ID's tell us that we didn't just *move* a couple of commits—we've literally rewritten our repository history with brand new commits.



Results of the interactive rebase

Interactive rebasing gives you complete control over your project history, but this can also be very dangerous. For example, if you were to delete a line from the rebase listing, the associated commit wouldn't be transferred to the new base, and its content would be lost forever. In a future module, we'll also see how rewriting history can get you in trouble with public Git repositories.

Stop to Amend a Commit

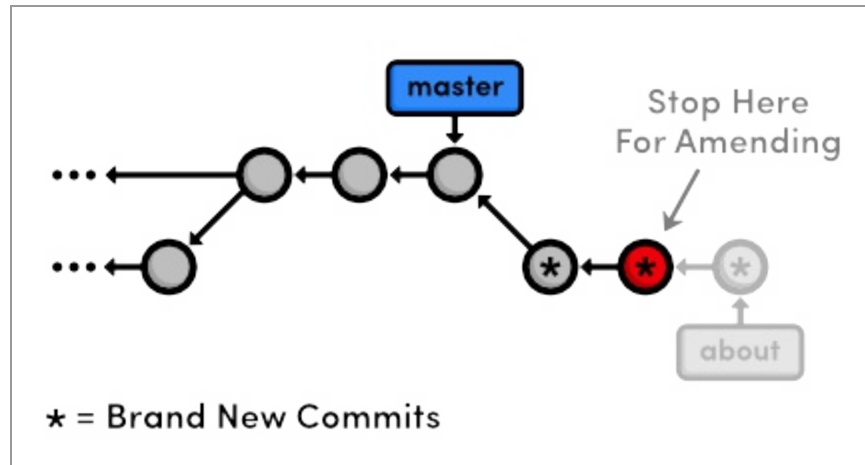
The previous rebase only stopped us to edit the *messages* of each commit. We can take this one step further and alter a *snapshot* during the rebase. Start by running another interactive rebasing session. Note that we've still been using master as the new base because it selects the desired commits from the about branch.

```
git rebase -i master
```

Specify the edit command for the second commit, as shown below.

```
pick 58dec2a Create the about page
edit 6ac8a9f Begin creating bio pages
pick 51c958c Add link to about section in home page
```

When Git starts to move the second commit to the new base, it will stop to do some “amending.” This gives you the opportunity to alter the staged snapshot before committing it.



Stopping to amend a commit

We'll leave a helpful note for Mary, whom we'll meet in the [Remotes](#) module. Open up `about/mary.html` and add the following.

```
[Mary, please update your bio!]
```

We're currently between commits in a rebase, but we can alter the staged snapshot in the exact same way as we have been throughout this entire tutorial:

```
git add about/mary.html
git status
git commit --amend
```

You can use the default message created by `git commit`. The new `--amend` flag tells Git to *replace* the existing commit with the staged snapshot instead of creating a new one. This is also very useful for fixing premature commits that often occur during normal development.

Continue the Interactive Rebase

Remember that we're in the middle of a rebase, and Git still has one more

commit that it needs to re-apply. Tell Git that we're ready to move on with the `--continue` flag:

```
git rebase --continue  
git log --oneline
```

Note that our history still appears to be the same (because we used the default commit message above), but the `Begin creating bio pages` commit contains different content than it did before the rebase, along with a new ID.

If you ever find yourself lost in the middle of a rebase and you're afraid to continue, you can use the `--abort` flag to abandon it and start over from scratch.

Publish the About Section

The point of all this interactive rebasing is to generate a *meaningful* history that we can merge back into `master`. And, since we've rebased `about` onto the tip of `master`, Git will be able to perform a fast-forward merge instead of using a merge commit to join the two branches.

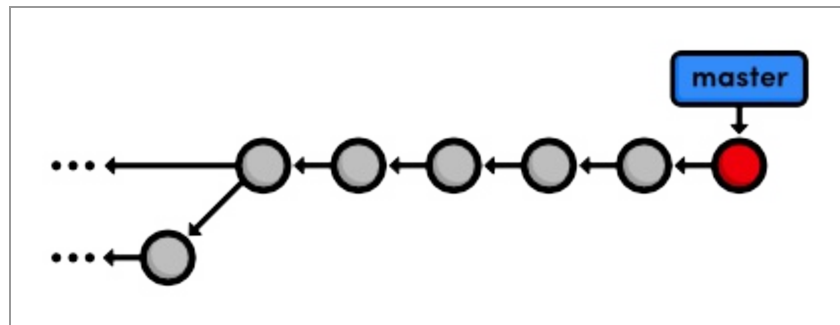
```
git checkout master  
git log --oneline  
git merge about  
git log --oneline
```

Don't forget to delete the obsolete `about` branch.

```
git branch -d about
```

Our final history is shown in the figure below. As you can see, a linear history is much easier to comprehend than the back-and-forth merging of the

previous module. But on the other hand, we don't have the slightest notion of *how* we got to our current state.



Merging and deleting the about branch

Conclusion

Rebasing enables fast-forward merges by moving a branch to the tip of another branch. It effectively eliminates the need for merge commits, resulting in a completely linear history. To an outside observer, it will seem as though you created every part of your project in a neatly planned sequence, even though you may have explored various alternatives or developed unrelated features in parallel. Rebasing gives you the power to choose exactly what gets stored in your repositories.

This can actually be a bit of a controversial topic within the Git community. Some believe that the benefits discussed in this module aren't worth the hassle of rewriting history. They take a more "pure" approach to Git by saying that your history should reflect *exactly* what you've done, ensuring that no information is ever lost. Furthermore, an advanced configuration of `git log` can display a linear history from a highly-branched repository.

But, others contend that merge commits should be *meaningful*. Instead of merging at arbitrary points just to access updates, they claim that merge commits should represent a symbolic joining of two branches. In particular, large software projects (such as the Linux kernel) typically advocate interactive rebasing to keep the repository as clean and straightforward as

possible.

The use of `git rebase` is entirely up to you. Customizing the evolution of your project can be very beneficial, but it might not be worth the trouble when you can accomplish close to the same functionality using merges exclusively. As a related note, you can use the following command to force a merge commit when Git would normally do a fast-forward merge.

```
git merge --no-ff <branch-name>
```

The next module will get a little bit more involved in our project history. We'll try fixing mistakes via complex rebases and even learn how to recover deleted commits.

Quick Reference

```
git rebase <new-base>
```

Move the current branch's commits to the tip of `<new-base>`, which can be either a branch name or a commit ID.

```
git rebase -i <new-base>
```

Perform an interactive rebase and select actions for each commit.

```
git commit --amend
```

Add staged changes to the most recent commit instead of creating a new one.

```
git rebase --continue
```

Continue a rebase after amending a commit.

```
git rebase --abort
```

Abandon the current interactive rebase and return the repository to its former state.

```
git merge --no-ff <branch-name>
```

Force a merge commit even if Git could do a fast-forward merge.

Rewriting History

The previous module on rebasing taught us how to move commits around and perform some basic edits while doing so, but now we're going to really get our hands dirty. We'll learn how to split up commits, revive lost snapshots, and completely rewrite a repository's history to our exact specifications.

Hopefully, this module will get you much more comfortable with the core Git components, as we'll be inspecting and editing the internal makeup of our project.

[Download the repository for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repository from the above link, uncompress it, and you're good to go.

Create the Red Page

First, let's create a new branch and add a few more HTML pages.

```
git checkout -b new-pages  
git branch
```

Notice that we created a new branch and checked it out in a single step by passing the `-b` flag to the `git checkout` command.

Next, create the file `red.html` and add the following content:

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
  <title>The Red Page</title>
```

```
<link rel="stylesheet" href="style.css" />
<meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #C00">The Red Page</h1>
  <p>Red is the color of <span style="color: #C00">passion</span>!</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

We'll hold off on committing this page for the moment.

Create the Yellow Page

Create a file called `yellow.html`, which should look like the following.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Yellow Page</title>
  <link rel="stylesheet" href="style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #FF0">The Yellow Page</h1>
  <p>Yellow is the color of <span style="color: #FF0">the sun</span>!</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Link and Commit the New Pages

Next, we'll link both new pages to the home page. Add the following items to the "Navigation" section in `index.html`:

```
<li style="color: #C00">
  <a href="red.html">The Red Page</a>
</li>
<li style="color: #FF0">
  <a href="yellow.html">The Yellow Page</a>
</li>
```

Then, commit all of these changes in a single snapshot.

```
git add red.html yellow.html index.html
git status
git commit -m "Add new HTML pages"
```

This is an example of a *bad* commit. It performed multiple, unrelated tasks, and it has a relatively generic commit message. Thus far, we haven't really specified when it's appropriate to commit changes, but the general rules are essentially the same as for branch creation:

- Commit a snapshot for each significant addition to your project.
- *Don't* commit a snapshot if you can't come up with a single, specific message for it.

This will ensure that your project has a meaningful commit history, which gives you the ability to see exactly when and where a feature was added or a piece of functionality was broken. However, in practice, you'll often wind up committing several changes in a single snapshot, since you won't always know what constitutes a "well-defined" addition as you're developing a project. Fortunately, Git lets us go back and fix up these problem commits

after the fact.

Create and Commit the Green Page

Let's create one more page before splitting that "bad" commit: Add the following HTML to a file called `green.html`.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Green Page</title>
  <link rel="stylesheet" href="style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #0C0">The Green Page</h1>
  <p><span style="color: #0C0">Green</span> is the color of earth.</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Add a link to `index.html` in the "Navigation" section:

```
<li style="color: #0C0">
  <a href="green.html">The Green Page</a>
</li>
```

And finally, stage the green page and commit the snapshot.

```
git add green.html index.html
git status
```

```
git commit -m "Add green page"
```

Begin an Interactive Rebase

The commits introduced in our new-pages branch are:

```
4c3027c Add green page
db96c72 Add new HTML pages
```

But, we want these commits to look more like:

```
4c3027c Add green page
9b1a64f Add yellow page
77a1cf1 Add red page
```

To achieve this, we can use the same interactive rebasing method covered in the previous module, only this time we'll actually *create* commits in the middle of the rebasing procedure.

```
git rebase -i master
```

Change the rebase listing to the following, then save the file and exit the editor to begin the rebase.

```
edit db96c72 Add new HTML pages
pick 4c3027c Add green page
```

Undo the Generic Commit

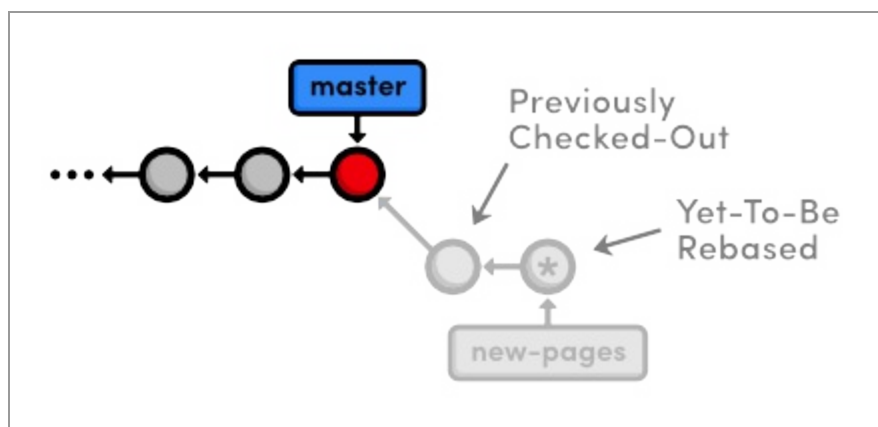
First, let's take a look at where we are with `git log --oneline`:

```
db96c72 Add new HTML pages
7070b0e Add link to about section in home page
...
```

When Git encountered the `edit` command in the rebase configuration, it stopped to let us edit the commit. As a result, the green page commit doesn't appear in our history yet. This should be familiar from the previous module. But instead of *amending* the current commit, we're going to completely remove it:

```
git reset --mixed HEAD~1
git log --oneline
git status
```

The `git reset` command moves the checked out snapshot to a new commit, and the `HEAD~1` parameter tells it to reset to the commit that occurs immediately before the current `HEAD` (likewise, `HEAD~2` would refer to second commit before `HEAD`). In this particular case, `HEAD~1` happens to coincide with `master`. The effect on our repository can be visualized as:



Resetting to HEAD~1

You may recall from [Undoing Changes](#) that we used `git reset --hard` to

undo uncommitted changes to our project. The `--hard` flag told Git to make the working directory look exactly like the most recent commit, giving us the intended effect of removing uncommitted changes.

But, this time we used the `--mixed` flag to preserve the working directory, which contains the changes we want to separate. That is to say, the `HEAD` moved, but the working directory remained unchanged. Of course, this results in a repository with uncommitted modifications. We now have the opportunity to add the `red.html` and `yellow.html` files to distinct commits.

Split the Generic Commit

Let's start with the red page. Since we only want to commit content that involves the red page, we'll have to manually go in and remove the yellow page's link from the "Navigation" section. In `index.html`, change this section to match the following:

```
<h2>Navigation</h2>
<ul>
  <li>
    <a href="about/index.html">About Us</a>
  </li>
  <li style="color: #F90">
    <a href="orange.html">The Orange Page</a>
  </li>
  <li style="color: #00F">
    <a href="blue.html">The Blue Page</a>
  </li>
  <li>
    <a href="rainbow.html">The Rainbow Page</a>
  </li>
  <li style="color: #C00">
```

```
<a href="red.html">The Red Page</a>
</li>
</ul>
```

Now, we can group the red page's updates into an independent commit.

```
git add red.html index.html
git status
git commit -m "Add red page"
```

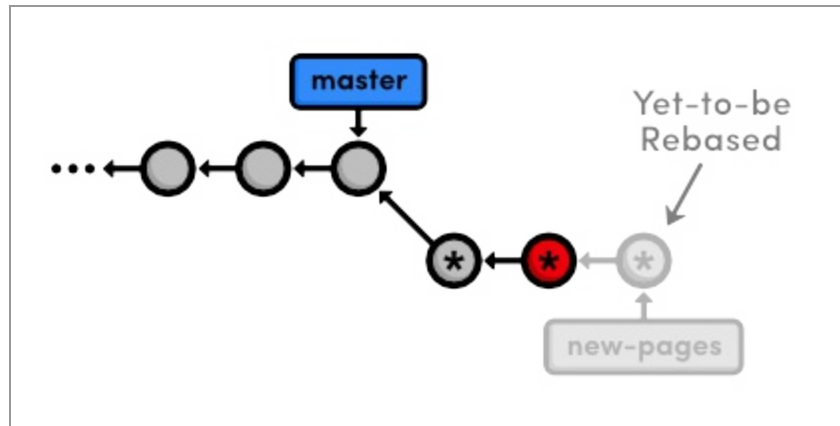
Next up is the yellow page. Go ahead and add it back to the “Navigation” section in index.html:

```
<li style="color: #FF0">
  <a href="yellow.html">The Yellow Page</a>
</li>
```

And again, stage and commit the snapshot.

```
git add yellow.html index.html
git status
git commit -m "Add yellow page"
```

We've successfully split up the contents of a single commit into two new snapshots, as shown below.



Creating two new commits while rebasing

But, don't forget that the rebase still needs to transfer the green page:

```
git rebase --continue
```

To summarize, we removed the “bad” commit from the current branch with `git reset`, keeping the contained HTML files intact with the `--mixed` flag. Then, we committed them in separate snapshots with the usual `git add` and `git commit` commands. The point to remember is that during a rebase you can add, delete, and edit commits to your heart's content, and the entire result will be moved to the new base.

Remove the Last Commit

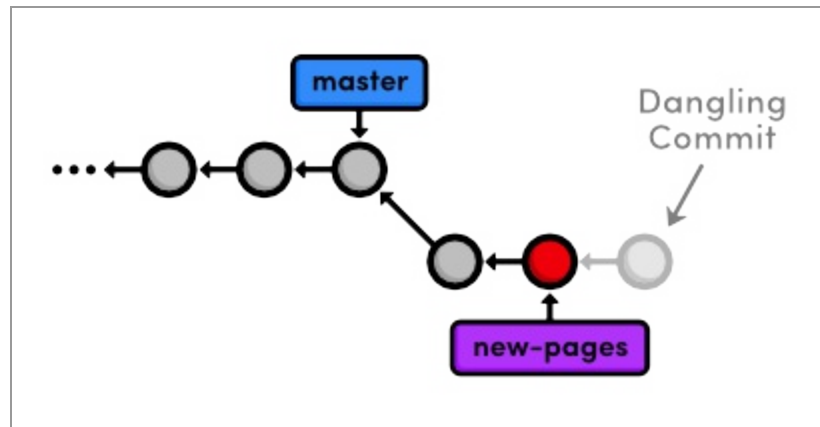
Next, we're going to “accidentally” remove the green page commit so we can learn how to retrieve it via Git's internal repository data.

```
git reset --hard HEAD~1  
git status  
git log --oneline
```

This moves the checked-out commit backward by one snapshot, along with the `new-pages` pointer. Note that `git status` tells us that we have nothing to

commit, since the `--hard` flag obliterated any changes in the working directory. And of course, the `git log` output shows that the `new-pages` branch no longer contains the green commit.

This behavior is slightly different from the reset we used in the interactive rebase: this time the *branch* moved with the new HEAD. Since we were on (no branch) during the rebase, there was no branch tip to move. However, in general, `git reset` is used to move branch tips around and optionally alter the working directory via one of its many flags (e.g., `--mixed` or `--hard`).



Removing the most recent commit

The commit that we removed from the branch is now a **dangling commit**. Dangling commits are those that cannot be reached from any branch and are thus in danger of being lost forever.

Open the Reflog

Git uses something called the **reflog** to record every change you make to your repository. Let's take a look at what it contains:

```
git reflog
```

The resulting output should look something like the following. Depending on

your version of Git, the messages might be slightly different. You can press space to scroll through the content or q to exit.

```
9b1a64f HEAD@{0}: reset: moving to HEAD~1
002185c HEAD@{1}: rebase -i (finish): returning to refs/heads/new-pages
002185c HEAD@{2}: rebase -i (pick): Add green page
9b1a64f HEAD@{3}: commit: Add yellow page
77a1cf1 HEAD@{4}: commit: Add red page
7070b0e HEAD@{5}: reset: moving to HEAD~1
...
```

The above listing reflects our last few actions. For example, the current HEAD, denoted by HEAD@{0}, resulted from resetting HEAD to HEAD~1. Four actions ago, the yellow page was applied during our rebase, as shown in HEAD@{3}.

The reflog is a *chronological* listing of our history, without regard for the repository's branch structure. This lets us find dangling commits that would otherwise be lost from the project history.

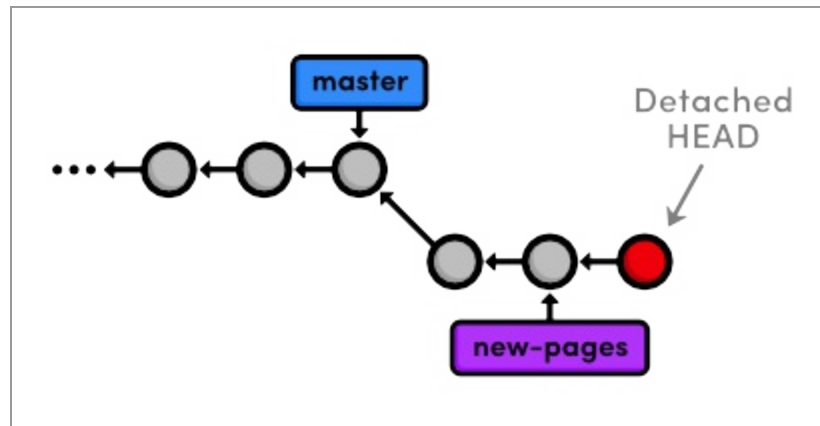
Revive the Lost Commit

At the beginning of each reflog entry, you'll find a commit ID representing the HEAD after that action. Check out the commit at HEAD@{2}, which should be where the rebase added the green page (change the ID below to the ID from *your* reflog).

```
git checkout 002185c
```

This puts us in a detached HEAD state, which means our HEAD is no longer on the tip of a branch. We're actually in the opposite situation as we were in [Undoing Changes](#) when we checked out a commit *before* the branch tip. Now, we're looking at a commit *after* the tip of the branch, but we still have

a detached HEAD:

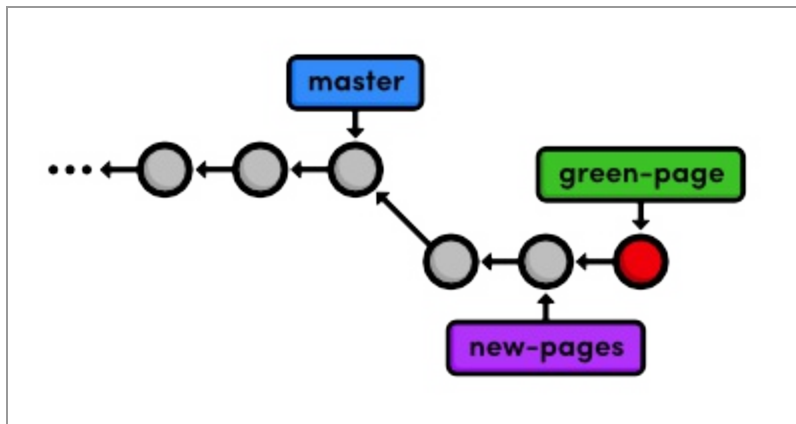


Checking out a dangling commit

To turn our dangling commit into a full-fledged branch, all we have to do is create one:

```
git checkout -b green-page
```

We now have a branch that can be merged back into the project:



Creating a branch from the dangling commit

The above diagram makes it easy to see that the green-page branch is an extension of new-pages, but how would we figure this out if we weren't drawing out the state of our repository every step of the way?

Filter the Log History

To view the differences between branches, we can use Git's log-filtering syntax.

```
git log new-pages..green-page
```

This will display all commits contained in `green-page` that aren't in the `new-pages` branch. The above command tells us that `green-page` contains one more snapshot than `new-pages`: our dangling commit (although, it's not really dangling anymore since we created a branch for it).

You can also use this syntax to limit the output of `git log`. For example, to display the last 4 commits on the current branch, you could use:

```
git log HEAD~4..HEAD
```

However, this is a bit verbose for such a common task, so Git developers added the `-n` flag as an easier way to limit output.

```
git log -n 4
```

The `-n 4` parameter tells Git to show only the last four commits from the current `HEAD`, making it the equivalent of the `HEAD~4..HEAD` syntax shown above. Similarly, `-n 3`, `-n 2`, and `-n 1` would display three, two, and one commit, respectively. This feature becomes very useful once a repository grows beyond one screenful of history.

Merge in the Revived Branch

We've revived our lost commit, and now we're ready to merge everything back into the `master` branch. Before we do the merge, let's see exactly what

we're merging:

```
git checkout master
git log HEAD..green-page --stat
```

The `git log HEAD..green-page` command shows us only those commits in `green-page` that aren't in `master` (since `master` is the current branch, we can refer to it as `HEAD`). The new `--stat` flag includes information about which files have been changed in each commit. For example, the most recent commit tells us that 14 lines were added to the `green.html` file and 3 lines were added to `index.html`:

```
commit 002185c71e6674915eb75be2afb4ca52c2c7fd1b
Author: Ryan <ryan.example@rypress.com>
Date:   Wed Jan 11 06:49:50 2012 -0600

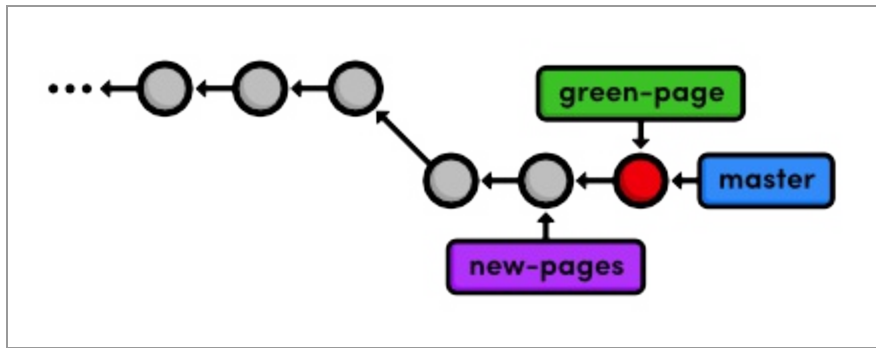
    Add green page

green.html | 14 ++++++
index.html |  3 +++
2 files changed, 17 insertions(+), 0 deletions(-)
```

If we didn't already know what was in this new commit, the log output would tell us which files we needed to look at. But, we authored all of these changes, so we can skip right to the merge.

```
git merge green-page
```

The following diagram shows the state of our repository after the merge.



Fast-forwarding master to the green-page branch

Note that the green-page branch already contains all the history of new-pages, which is why we merged the former instead of the latter. If this wasn't the case, Git would complain when we try to run the following command.

```
git branch -d new-pages
```

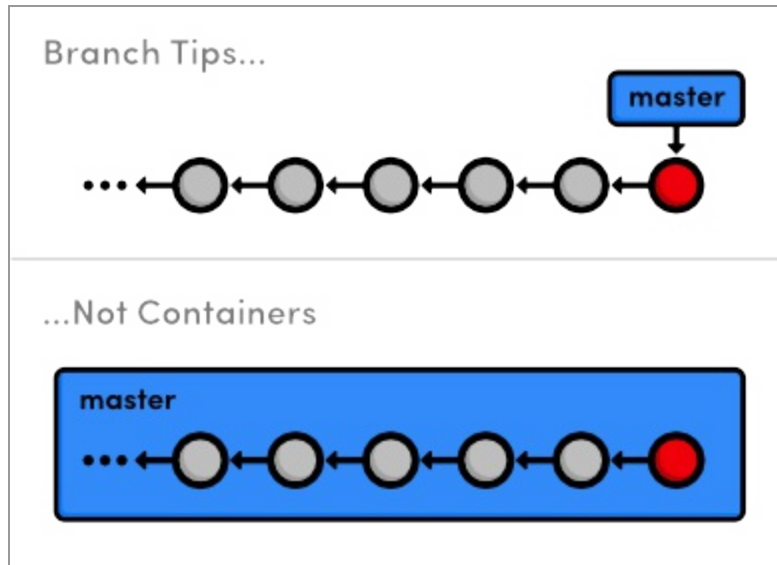
We can go ahead and delete the green page branch, too.

```
git branch -d green-page
```

Conclusion

This module took an in-depth look at rebasing, resetting, and the reflog. We learned how to split old commits into one or more new commits, and how to revive “lost” commits. Hopefully, this has given you a better understanding of the interaction between the working directory, the stage, branches, and committed snapshots. We also explored some new options for displaying our commit history, which will become an important skill as our project grows.

We did a lot of work with branch tips this module. It's important to realize that Git uses the *tip* of a branch to represent the *entire branch*. That is to say, a branch is actually a pointer to a single commit—not a container for a series of commits. This idea has been implicitly reflected in our diagrams:



Branch tips, not containers

The history is represented by the parent of each commit (designated by arrows), not the branch itself. So, to request a new branch, all Git has to do is create a reference to the current commit. And, to add a snapshot to a branch, it just has to move the branch reference to the new commit. An understanding of Git's branch representation should make it easier to wrap your head around merging, rebasing, and other kinds of branch manipulation.

We'll revisit Git's internal representation of a repository in the [final module](#) of this tutorial. But now, we're finally ready to discuss multi-user development, which happens to rely entirely on Git branches.

Quick Reference

`git reflog`

Display the local, chronological history of a repository.

`git reset --mixed HEAD~<n>`

Move the HEAD backward <n> commits, but don't change the working directory.

`git reset --hard HEAD~<n>`

Move the HEAD backward <n> commits, and change the working directory to match.

```
git log <since>..<until>
```

Display the commits reachable from <until> but not from <since>. These parameters can be either commit ID's or branch names.

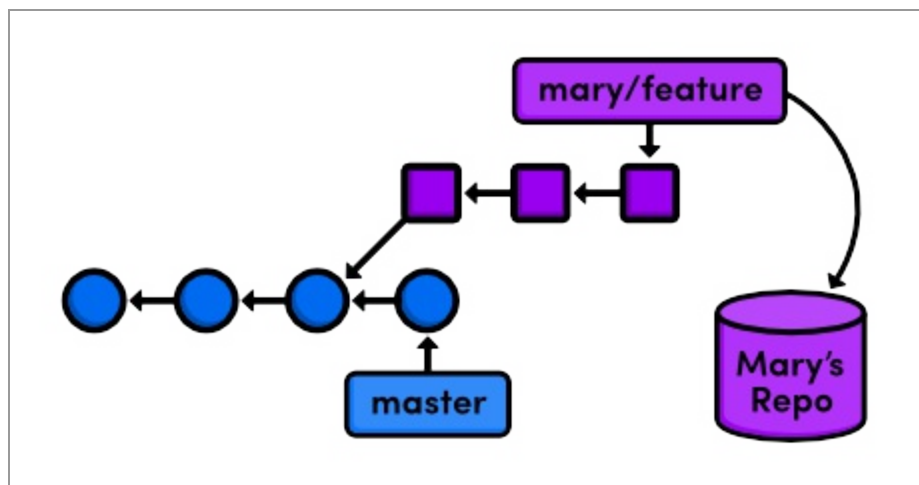
```
git log --stat
```

Include extra information about altered files in the log output.

Remotes

Simply put, a **remote repository** is one that is not your own. It can be another Git repository that's on your company's network, the internet, or even your local filesystem, but the point is that it's a repository distinct from your `my-git-repo` project.

We've already seen how branches can streamline a workflow within a single repository, but they also happen to be Git's mechanism for sharing commits between repositories. **Remote branches** act just like the local branches that we've been using, only they represent a branch in someone else's repository.



Accessing a feature branch from a remote repository

This means that we can adapt our merging and rebasing skills to make Git a fantastic collaboration tool. Over the next few modules, we'll be exploring various multi-user workflows by pretending to be different developers working on our example website.

For several parts of this module, we're going to pretend to be Mary, the graphic designer for our website. Mary's actions are clearly denoted by including her name in the heading of each step.

[Download the repository for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repository from the above link, uncompress it, and you're good to go.

Clone the Repository (Mary)

First, Mary needs her own copy of the repository to work with. The [Distributed Workflows](#) module will discuss network-based remotes, but right now we're just going to store them on the local filesystem.

```
cd /path/to/my-git-repo
cd ..
git clone my-git-repo marys-repo
cd marys-repo
```

The first two lines navigate the command shell to the directory *above* my-git-repo. Make sure to change /path/to/my-git-repo to the actual path to your repository. The git clone command copies our repository into marys-repo, which will reside in the same directory as my-git-repo. Then, we navigate to Mary's repository so we can start pretending to be Mary.

Run git log to verify that Mary's repository is in fact a copy of our original repository.

Configure The Repository (Mary)

First off, Mary needs to configure her repository so that we know who contributed what to the project.

```
git config user.name "Mary"
git config user.email mary.example@rypress.com
```

You may recall from the first module that we used a `--global` flag to set the configuration for the entire Git installation. But since Mary's repository is on the local filesystem, she needs a *local* configuration.

Use a text editor to open up the file called `config` in the `.git` folder of Mary's project (you may need to enable hidden files to see `.git`). This is where local configurations are stored, and we see Mary's information at the bottom of the file. Note that this overrides the global configuration that we set in [The Basics](#).

Start Mary's Day (Mary)

Today, Mary is going to be working on her bio page, which she should develop in a separate branch:

```
git checkout -b bio-page
```

Mary can create and check out branches just like we did in our copy of the project. Her repository is a completely isolated development environment, and she can do whatever she wants in here without worrying about what's going on in `my-git-repo`. Just as branches are an abstraction for the working directory, the staged snapshot, and a commit history, a repository is an abstraction for branches.

Create Mary's Bio Page (Mary)

Let's complete Mary's biography page. In `marys-repo`, change `about/mary.html` to:

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<title>About Mary</title>

<link rel="stylesheet" href="../style.css" />

<meta charset="utf-8" />
</head>
<body>
  <h1>About Mary</h1>
  <p>I'm a graphic designer.</p>

  <h2>Interests</h2>
  <ul>
    <li>Oil Painting</li>
    <li>Web Design</li>
  </ul>

  <p><a href="index.html">Return to about page</a></p>
</body>
</html>
```

Again, we're developing this in a branch as a best-practice step: our master branch is only for stable, tested code. Stage and commit the snapshot, then take a look at the result.

```
git commit -a -m "Add bio page for Mary"
git log -n 1
```

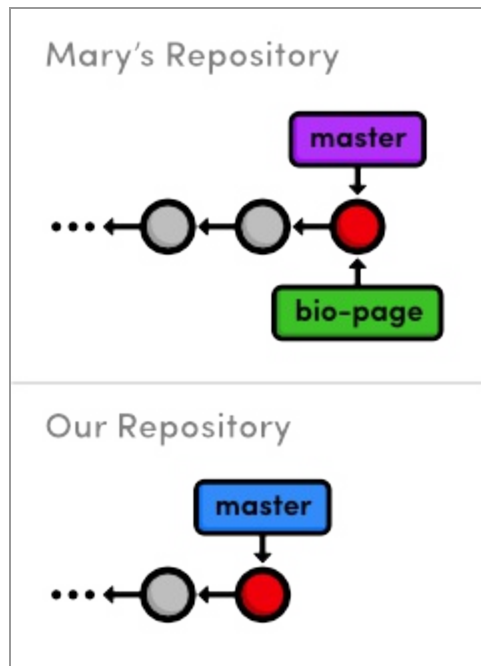
The Author field in the log output should reflect the local configurations we made for Mary's name and email. Remember that the `-n 1` flag limits history output to a single commit.

Publish the Bio Page (Mary)

Now, we can publish the bio page by merging into the master branch.

```
git checkout master
git merge bio-page
```

Of course, this results in a fast-forward merge. We'll eventually pull this update into my-git-repo once we stop pretending to be Mary. Here's what Mary's repository looks like compared to ours:



Merging Mary's bio-page branch with her master

Notice that both repositories have normal, local branches—we haven't had any interaction between the two repositories, so we don't see any remote branches yet. Before we switch back to my-git-repo, let's examine Mary's remote connections.

View Remote Repositories (Mary)

Mary can list the connections she has to other repositories using the following command.

```
git remote
```

Apparently, she has a remote called `origin`. When you clone a repository, Git automatically adds an `origin` remote pointing to the original repository, under the assumption that you'll probably want to interact with it down the road. We can request a little bit more information with the `-v` (verbose) flag:

```
git remote -v
```

This shows the full path to our original repository, verifying that `origin` is a remote connection to `my-git-repo`. The same path is designated as a “fetch” and a “push” location. We'll see what these mean in a moment.

Return to Your Repository (You)

Ok, we're done being Mary, and we can return to our own repository.

```
cd ../my-git-repo
```

Notice that Mary's bio page is still empty. It's very important to understand that this repository and Mary's repository are completely separate. While she was altering her bio page, we could have been doing all sorts of other things in `my-git-repo`. We could have even changed her bio page, which would result in a merge conflict when we try to pull her changes in.

Add Mary as a Remote (You)

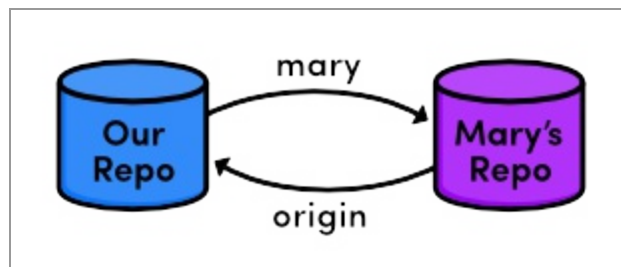
Before we can get ahold of Mary's bio page, we need access to her repository. Let's look at our current list of remotes:

```
git remote
```

We don't have any (origin was never created because we didn't clone from anywhere). So, let's add Mary as a remote repository.

```
git remote add mary ../marys-repo
git remote -v
```

We can now use `mary` to refer to Mary's repository, which is located at `../marys-repo`. The `git remote add` command is used to bookmark another Git repository for easy access, and our connections can be seen in the figure below.



Connections to remote repositories

Now that our remote *repositories* are setup, we'll spend the rest of the module discussing remote *branches*.

Fetch Mary's Branches (You)

As noted earlier, we can use remote branches to access snapshots from another repository. Let's take a look at our current remote branches with the `-r` flag:

```
git branch -r
```

Again, we don't have any. To populate our remote branch listing, we need to **fetch** the branches from Mary's repository:

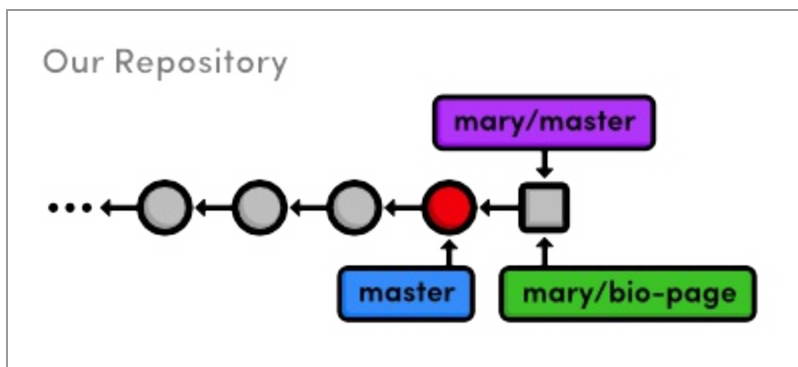

```
git fetch mary
git branch -r
```

This will go to the “fetch” location shown in `git remote -v` and download all of the branches it finds there into our repository. The resulting branches are shown below.

```
mary/bio-page
mary/master
```

Remote branches are always listed in the form `<remote-name>/<branch-name>` so that they will never be mistaken for local branches. The above listing reflects the state of Mary’s repository at the time of the fetch, but they will not be automatically updated if Mary continues developing any of her branches.

That is to say, our remote branches are not *direct* links into Mary’s repository—they are read-only copies of her branches, stored in our own repository. This means that we would have to do another fetch to access new updates.



Mary’s remote branches in our repository

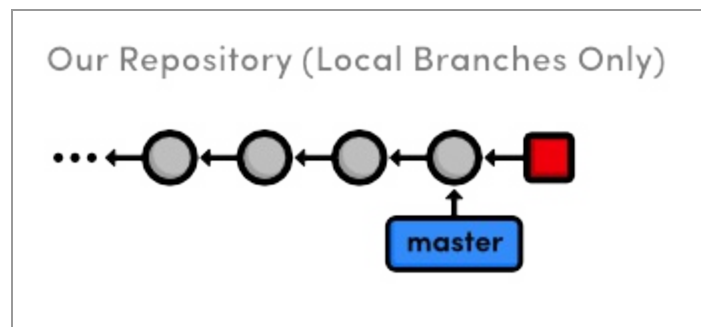
The above figure shows the state of *our* repository. We have access to Mary’s snapshots (represented as squares) and her branches, even though we don’t have a real-time connection to Mary’s repository.

Check Out a Remote Branch

Let's check out a remote branch to review Mary's changes.

```
git checkout mary/master
```

This puts us in a detached HEAD state, just like we were in when we checked out a dangling commit. This shouldn't be that surprising, considering that our remote branches are *copies* of Mary's branches. Checking out a remote branch takes our HEAD off the tip of a local branch, illustrated by the following diagram.



Checking out Mary's master branch

We can't continue developing if we're not on a local branch. To build on `mary/master` we either need to merge it into our own local master or create another branch. We did the latter in [Branches, Part I](#) to build on an old commit and in [the previous module](#) to revive a "lost" commit, but right now we're just looking at what Mary did, so the detached HEAD state doesn't really affect us.

Find Mary's Changes

We can use the same log-filtering syntax from the previous module to view Mary's changes.

```
git log master..mary/master --stat
```

This shows us what Mary has added to her master branch, but it's also a good idea to see if we've added any new changes that aren't in Mary's repository:

```
git log mary/master..master --stat
```

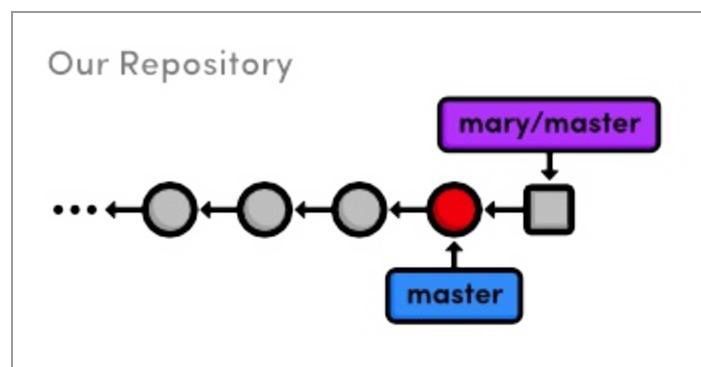
This won't output anything, since we haven't altered our database since Mary cloned it. In other words, our history hasn't *diverged*—we're just *behind* by a commit.

Merge Mary's Changes

Let's approve Mary's changes and integrate them into our own master branch.

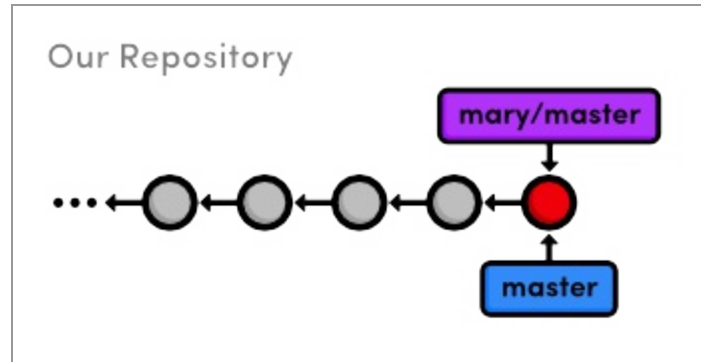
```
git checkout master  
git merge mary/master
```

Even though `mary/master` is a remote branch, this still results in a fast-forward merge because there is a linear path from our master to the tip of `mary/master`:



Before merging Mary's master branch into our own

After the merge, the snapshots from Mary's remote branch become a part of our local master branch. As a result, our master is now synchronized with Mary's:



After merging Mary's master branch into our own

Notice that we only interacted with Mary's master branch, even though we had access to her bio-page. If we hadn't been pretending to be Mary, we wouldn't have known what this feature branch was for or if it was ready to be merged. But, since we've designated master as a stable branch for the project, it was safe to integrate those updates (assuming Mary was also aware of this convention).

Push a Dummy Branch

To complement our `git fetch` command, we'll take a brief look at **pushing**. Fetching and pushing are *almost* opposites, in that fetching imports branches, while pushing exports branches to another repository. Let's take a look:

```
git branch dummy
git push mary dummy
```

This creates a new branch called `dummy` and sends it to Mary. Switch into Mary's repository to see what we did:

```
cd ../marys-repo
git branch
```

You should find a new dummy branch in her *local* branch listing. I said that `git fetch` and `git push` are *almost* opposites because pushing creates a new *local* branch, while fetching imports commits into *remote* branches.

Now, put yourself in Mary's shoes. She was developing in her own repository when, all of a sudden, a new dummy branch appeared out of nowhere. Obviously, pushing branches into other people's repositories can make for a chaotic workflow. So, as a general rule, **you should never push into another developer's repository**. But then, why does `git push` even exist?

Over the next few modules, we'll see that pushing is a necessary tool for maintaining public repositories. Until then, just remember to never, ever push into one of your friend's repositories. Let's get rid of these dummy branches and return to our repository.

```
git branch -d dummy
cd ../my-git-repo
git branch -d dummy
```

Push a New Tag

An important property of `git push` is that it does not automatically push tags associated with a particular branch. Let's examine this by creating a new tag.

```
git tag -a v2.0 -m "An even stabler version of the website"
```

We now have a `v2.0` tag in `my-git-repo`, which we can see by running the

`git tag` command. Now, let's try pushing the branch to Mary's repository.

```
git push mary master
```

Git will say her master branch is already up-to-date, and her repository will remain unchanged. Instead of pushing the branch that contains the tag, Git requires us to manually push the tag itself:

```
git push mary v2.0
```

You should now be able to see the `v2.0` tag in Mary's repository with a quick `git tag`. It's very easy to forget to push new tags, so if it seems like your project has lost a tag or two, it's most likely because you didn't push them to the remote repository.

Conclusion

In this module, we learned how remote branches can be used to access content in someone else's repository. The remotes listed in `git remote` are merely bookmarks for a full path to another repository. We used a local path, but as we'll soon see, Git can use the SSH protocol to access repositories on another computer.

The convention of master as a stable branch allowed us to pull changes without consulting Mary, but this doesn't necessarily have to be the case. When implementing your own workflow, Git offers you a lot of flexibility about when and where you should pull from your team members. As long as you clearly define your project conventions, you can designate special uses or privileges to *any* branch.

That said, it's important to note that remotes are for *people*, whereas branches are for *topics*. Do *not* create separate branches for each of your developers—

give them separate repositories and bookmark them with `git remote add`.
Branches should always be for project development, not user management.

Now that we know how Git shares information between repositories, we can add some more structure to our multi-user development environment. The next module will show you how to set up and access a shared central repository.

Quick Reference

`git clone <remote-path>`

Create a copy of a remote Git repository.

`git remote`

List remote repositories.

`git remote add <remote-name> <remote-path>`

Add a remote repository.

`git fetch <remote-name>`

Download remote branch information, but do not merge anything.

`git merge <remote-name>/<branch-name>`

Merge a remote branch into the checked-out branch.

`git branch -r`

List remote branches.

`git push <remote-name> <branch-name>`

Push a local branch to another repository.

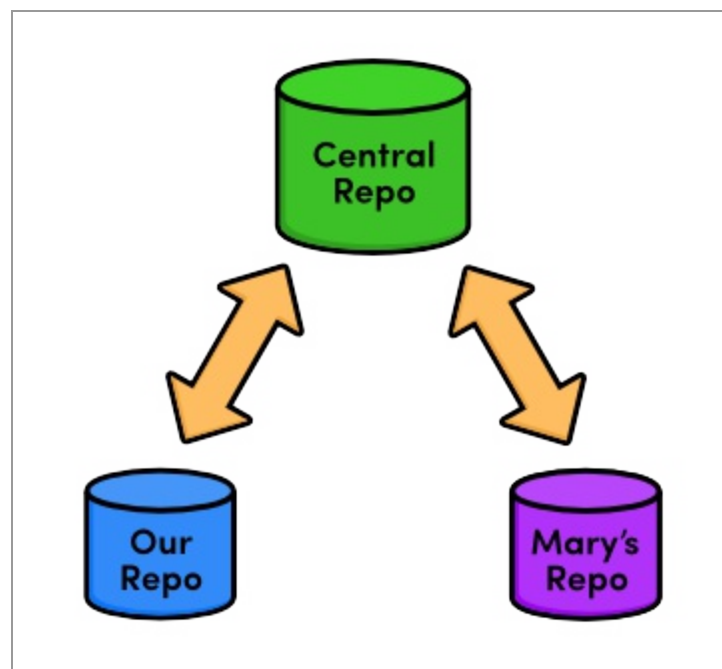
`git push <remote-name> <tag-name>`

Push a tag to another repository.

Centralized Workflows

In the previous module, we shared information directly between two developers' repositories: `my-git-repo` and `marys-repo`. This works for very small teams developing simple programs, but larger projects call for a more structured environment. This module introduces one such environment: the **centralized workflow**.

We'll use a third Git repository to act as a central communication hub between us and Mary. Instead of pulling changes into `my-git-repo` from `marys-repo` and vice versa, we'll push to and fetch from a dedicated storage repository. After this module, our workflow will look like the following.



The centralized workflow

Typically, you would store the central repository on a server to allow internet-based collaboration. Unfortunately, server configuration can vary among hosting providers, making it hard to write universal step-by-step instructions. So, we'll continue exploring remote repositories using our local filesystem, just like in the previous module.

If you have access to a server, feel free to use it to host the central repository that we're about to create. You'll have to provide SSH paths to your server-based repository in place of the paths provided below, but other than that, you can follow this module's instructions as you find them. For everyone else, our network-based Git experience will begin in the next module.

[Download the repositories for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repositories from the above link, uncompress them, and you're good to go.

Create a Bare Repository (Central)

First, let's create our central "communication hub." Again, make sure to change `/path/to/my-git-repo` to the actual path to your repository. If you've decided to host the central repository on your server, you should SSH into it and run the `git init` command wherever you'd like to store the repository.

```
cd /path/to/my-git-repo
cd ..
git init --bare central-repo.git
```

As in the very first module, `git init` creates a new repository. But this time, we used the `--bare` flag to tell Git that we don't want a working directory. This will prevent us from developing in the central repository, which eliminates the possibility of messing up another user's environment with `git push`. A central repository is only supposed to act as a *storage facility*—not a development environment.

If you examine the contents of the resulting `central-repo.git` folder, you'll

notice that it contains the exact same files as the `.git` folder in our `my-git-repo` project. Git has *literally* gotten rid of our working directory. The conventional `.git` extension in the directory name is a way to convey this property.

Update Remotes (Mary and You)

We've successfully set up a central repository that can be used to share updates between us, Mary, and any other developers. Next, we should add it as a remote to both `marys-repo` and `my-git-repo`.

```
cd marys-repo
git remote rm origin
git remote add origin ../central-repo.git
```

Now for our repository:

```
cd ../my-git-repo
git remote add origin ../central-repo.git
git remote rm mary
```

Note that we deleted the remote connections between Mary and our `my-git-repo` folder with `git remote rm`. For the rest of this module, we'll only use the central repository to share updates.

If you decided to host the central repository on a server, you'll need to change the `../central-repo.git` path to:

`ssh://user@example.com/path/to/central-repo.git`, substituting your SSH username and server location for `user@example.com` and the central repository's location for `path/to/central-repo.git`.

Push the Master Branch (You)

We didn't *clone* the central repository—we just initialized it as a bare repository. This means it doesn't have any of our project history yet. We can fix that using the `git push` command introduced in the last module.

```
git push origin master
```

Our central repository now contains our entire master branch, which we can double-check with the following.

```
cd ../central-repo.git  
git log
```

This should output the familiar history listing of the master branch.

Recall that `git push` creates *local* branches in the destination repository. We said it was dangerous to push to a friend's repository, as they probably wouldn't appreciate new branches appearing at random. However, it's safe to create local branches in `central-repo.git` because it has no working directory, which means it's impossible to disturb anyone's development.

Add News Update (You)

Let's see our new centralized collaboration workflow in action by committing a few more snapshots.

```
cd ../my-git-repo  
git checkout -b news-item
```

Create a file called `news-3.html` in `my-git-repo` and add the following HTML.

```

<!DOCTYPE html>
<html lang="en">
<head>
  <title>Middle East's Silent Beast</title>
  <link rel="stylesheet" href="style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #D90">Middle East's Silent Beast</h1>
  <p>Late yesterday evening, the Middle East's largest
  design house&mdash;until now, silent on the West's colorful
  disagreement&mdash;announced the adoption of
  <span style="color: #D90">Yellow</span> as this year's
  color of choice.</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>

```

Next, add a link to the “News” section of `index.html` so that it looks like:

```

<h2 style="color: #C00">News</h2>
<ul>
  <li><a href="news-1.html">Blue Is The New Hue</a></li>
  <li><a href="rainbow.html">Our New Rainbow</a></li>
  <li><a href="news-2.html">A Red Rebellion</a></li>
  <li><a href="news-3.html">Middle East's Silent Beast</a></li>
</ul>

```

Stage and commit a snapshot.

```
git add news-3.html index.html
```

```
git status  
git commit -m "Add 3rd news item"
```

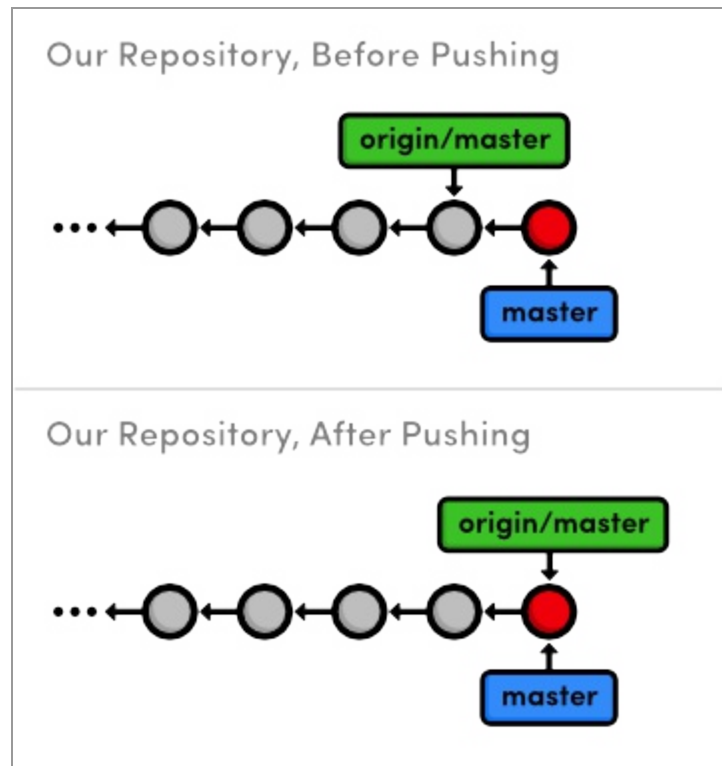
Publish the News Item (You)

Previously, “publishing” meant merging with the local master branch. But since we’re *only* interacting with the central repository, our master branch is private again. There’s no chance of Mary pulling content directly from our repository.

Instead, everyone accesses updates through the *public* master branch, so “publishing” means pushing to the central repository.

```
git checkout master  
git merge news-item  
git branch -d news-item  
git push origin master
```

After merging into master as we normally would, `git push` updates the central repository’s master branch to reflect our local master. From our perspective, the push can be visualized as the following:



Pushing master to the central repository

Note that this accomplishes the exact same thing as going into the central repository and doing a fetch/fast-forward merge, except `git push` allows us to do everything from inside `my-git-repo`. We'll see some other convenient features of this command later in the module.

Update CSS Styles (Mary)

Next, let's pretend to be Mary again and add some CSS formatting (she is our graphic designer, after all).

```
cd ../marys-repo
git checkout -b css-edits
```

Add the following to the end of `style.css`:

```
h1 {  
  font-size: 32px;  
}  
  
h2 {  
  font-size: 24px;  
}  
  
a:link, a:visited {  
  color: #03C;  
}
```

And, stage and commit a snapshot.

```
git commit -a -m "Add CSS styles for headings and links"
```

Update Another CSS Style (Mary)

Oops, Mary forgot to add some formatting. Append the h3 styling to style.css:

```
h3 {  
  font-size: 18px;  
  margin-left: 20px;  
}
```

And of course, stage and commit the updates.

```
git commit -a -m "Add CSS styles for 3rd level headings"
```

Clean Up Before Publishing (Mary)

Before Mary considers pushing her updates to the central repository, she needs to make sure she has a clean history. This *must* be done by Mary, because it's near-impossible to change history after it has been made public.

```
git rebase -i master
```

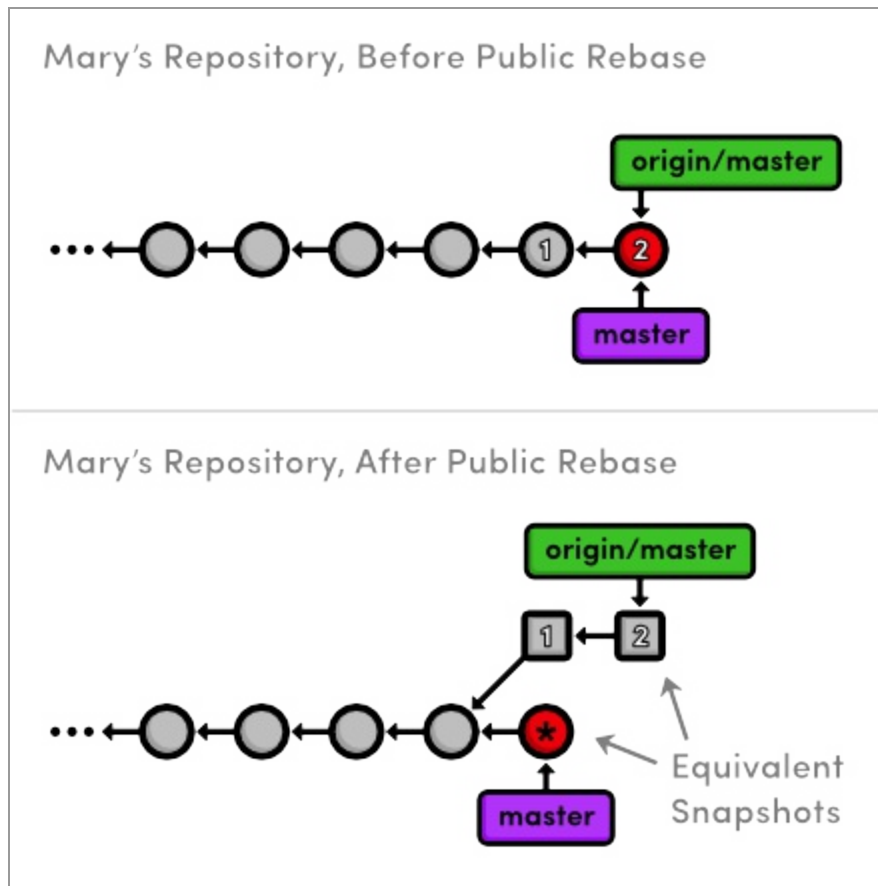
This highlights another benefit of using isolated branches to develop independent features. Mary doesn't need to go back and figure out what changes need to be rebased, since they all reside in her current branch. Change the rebase configuration to:

```
pick 681bd1c Add CSS styles for headings and links  
squash eabac68 Add CSS styles for 3rd level headings
```

When Git stops to ask for the combined commit message, just use the first commit's message:

```
Add CSS styles for headings and links
```

Consider what would have happened had Mary rebased *after* pushing to the central repository. She would be re-writing commits that other developers may have already pulled into their project. To Git, Mary's re-written commits look like entirely new commits (since they have different ID's). This situation is shown below.



Squashing a public commit

The commits labeled 1 and 2 are the public commits that Mary would be rebasing. Afterwards, the public history is still the exact same as Mary's original history, but now her local master branch has diverged from origin/master—even though they represent the same snapshot.

So, to publish her rebased master branch to the central repository, Mary would have to merge with origin/master. This cannot be a fast-forward merge, and the resulting merge commit is likely to confuse her collaborators and disrupt their workflow.

This brings us to the most important rule to remember while rebasing: **Never, ever rebase commits that have been pushed to a shared repository.**

If you need to change a public commit, use the `git revert` command that we

discussed in [Undoing Changes](#). This creates a new commit with the required modifications instead of re-writing old snapshots.

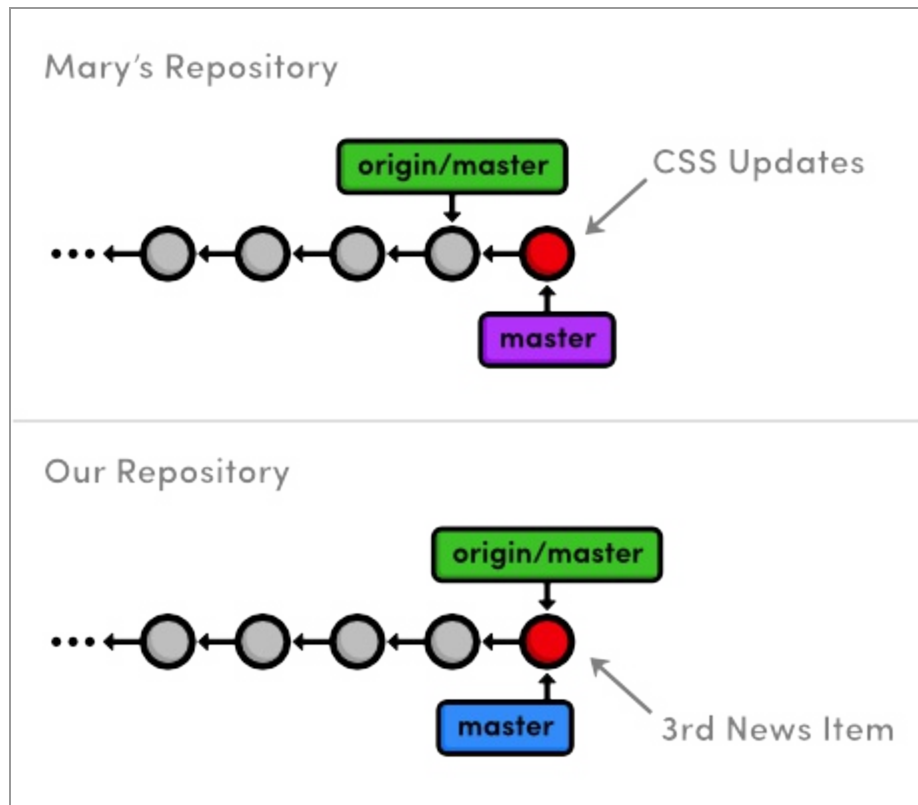
Publish CSS Changes (Mary)

Now that her history is cleaned up, Mary can publish the changes.

```
git checkout master  
git merge css-edits  
git branch -d css-edits
```

She shouldn't push the `css-edits` branch to the server, since it's no longer under development, and other collaborators wouldn't know what it contains. However, if we had all decided to develop the CSS edits together and wanted an isolated environment to do so, it would make sense to publish it as an independent branch.

Mary still needs to push the changes to the central repository. But first, let's take a look at the state of everyone's project.



Before publishing Mary's CSS changes

You might be wondering how Mary can push her local master up to the central repository, since it has progressed since Mary last fetched from it. This is a common situation when many developers are working on a project simultaneously. Let's see how Git handles it:

```
git push origin master
```

This will output a verbose rejection message. It seems that Git won't let anyone push to a remote server if it doesn't result in a fast-forward merge. This prevents us from losing the Add 3rd news item commit that would need to be overwritten for origin/master to match mary/master.

Pull in Changes (Mary)

Mary can solve this problem by pulling in the central changes before trying to

push her CSS changes. First, she needs the most up-to-date version of the origin/master branch.

```
git fetch origin
```

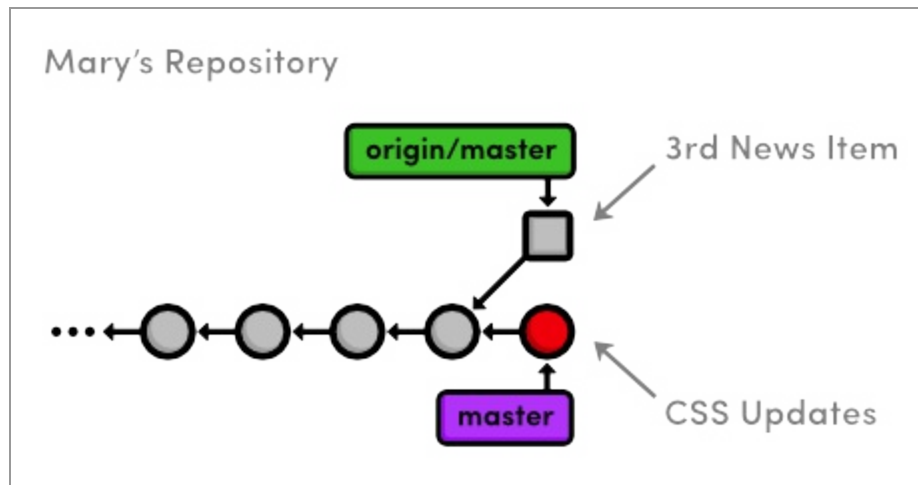
Remember that Mary can see what's in origin/master and not in the local master using the .. syntax:

```
git log master..origin/master
```

And she can also see what's in her master that's not in origin/master:

```
git log origin/master..master
```

Since both of these output a commit, we can tell that Mary's history diverged. This should also be clear from the diagram below, which shows the updated origin/master branch.



Fetching the central repository's master branch

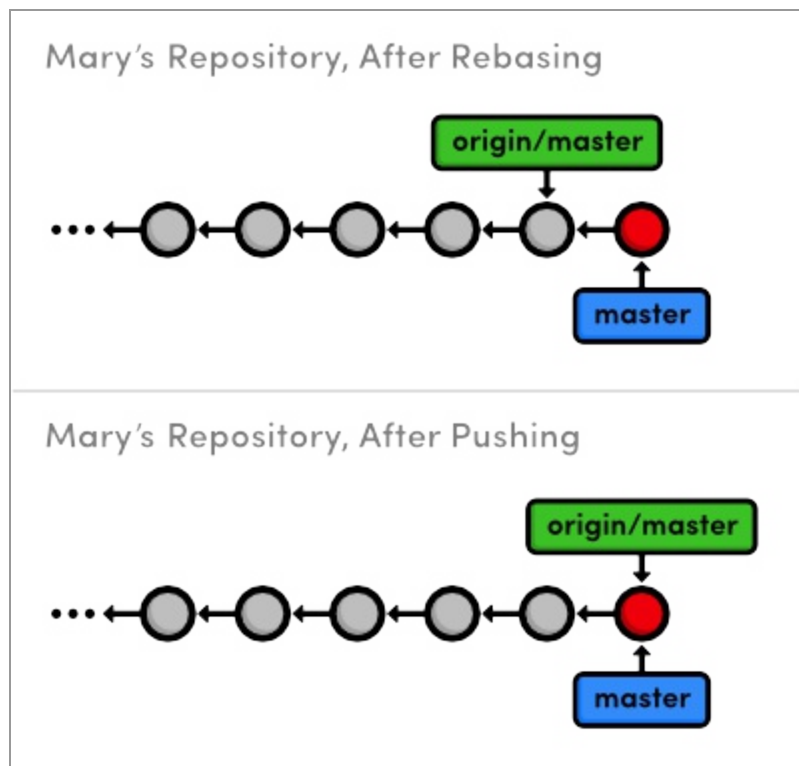
Mary is now in the familiar position of having to pull in changes from another branch. She can either merge, which *cannot* be fast-forwarded, or she

can rebase for a linear history.

Typically, you'll want to rebase your changes on top of those found in your central repository. This is the equivalent of saying, "I want to add my changes to what everyone else has already done." As previously discussed, rebasing also eliminates superfluous merge commits. For these reasons, Mary will opt for a rebase.

```
git rebase origin/master  
git push origin master
```

After the rebase, Mary's master branch contains everything from the central repository, so she can do a fast-forward push to publish her changes.



Updating the central repository's master

Pull in Changes (You)

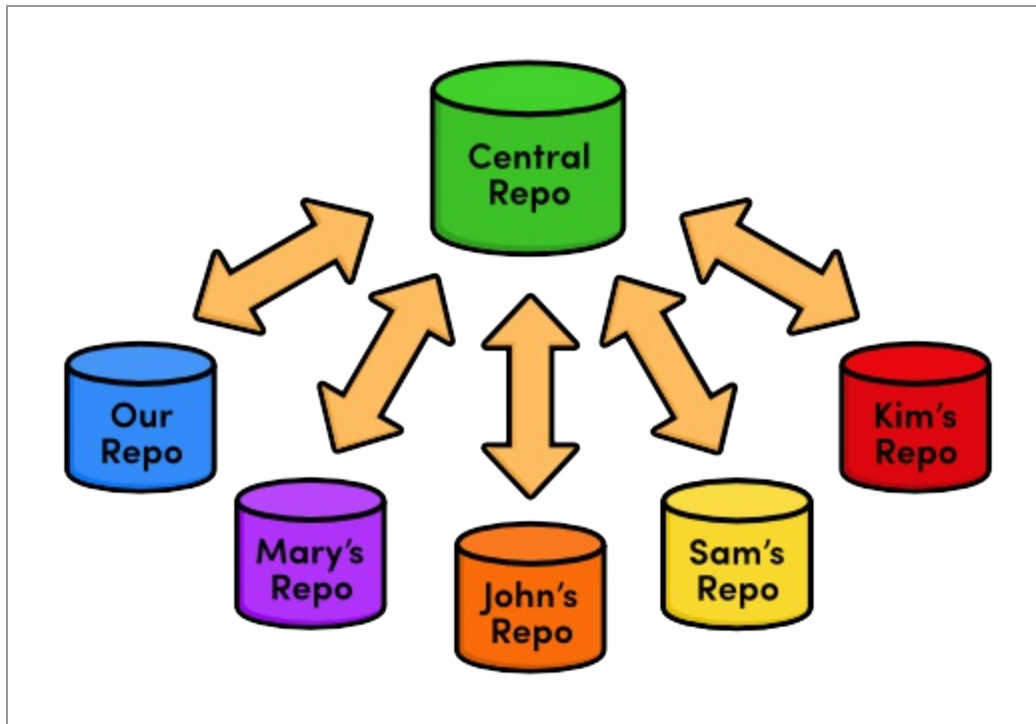
Finally, we'll switch back to our repository and pull in Mary's CSS formatting.

```
cd ../my-git-repo
git fetch origin
git log master..origin/master --stat
git log origin/master..master --stat
```

Of course, the second log command won't output anything, since we haven't added any new commits while Mary was adding her CSS edits. It's usually a good idea to check this before trying to merge in a remote branch. Otherwise, you might end up with some extra merge commits when you thought you were fast-forwarding your branch.

```
git merge origin/master
```

Our repository is now synchronized with the central repository. Note that Mary may have moved on and added some new content that we don't know about, but it doesn't matter. The only changes we need to know about are those in `central-repo.git`. While this doesn't make a huge difference when we're working with just one other developer, imagine having to keep track of a dozen different developers' repositories in real-time. This kind of chaos is precisely the problem a centralized collaboration workflow is designed to solve:



The centralized workflow with many developers

The presence of a central communication hub condenses all this development into a single repository and ensures that no one overwrites another's content, as we discovered while trying to push Mary's CSS updates.

Conclusion

In this module, we introduced another remote repository to serve as the central storage facility for our project. We also discovered bare repositories, which are just like ordinary repositories—minus the working directory. Bare repositories provide a “safe” location to push branches to, as long as you remember not to rebase the commits that it already contains.

We hosted the central repository on our local filesystem, right next to both ours and Mary's projects. However, most real-world central repositories reside on a remote server with internet access. This lets any developer fetch from or push to the repository over the internet, making Git a very powerful multi-user development platform. Having the central repository on a remote

server is also an affordable, convenient way to back up a project.

Next up, we'll configure a network-based repository using a service called GitHub. In addition to introducing network access for Git repositories, this will open the door for another collaboration standard: the integrator workflow.

Quick Reference

```
git init --bare <repository-name>
```

Create a Git repository, but omit the working directory.

```
git remote rm <remote-name>
```

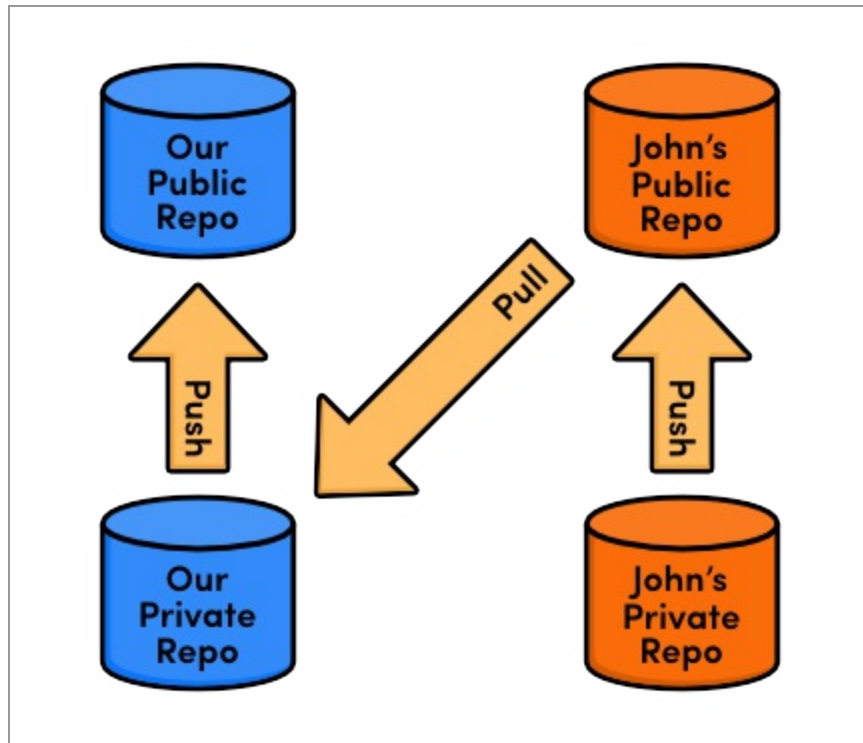
Remove the specified remote from your bookmarked connections.

Distributed Workflows

Now that we know how to share information via a centralized workflow, we can appreciate some of the drawbacks of this collaboration model. While it may be convenient, allowing everyone to push to an “official” repository raises some legitimate security concerns. It means that for anyone to contribute content, they need access to the *entire project*.

This is fine if you’re only interacting with a small team, but imagine a scenario where you’re working on an open-source software project and a stranger found a bug, fixed it, and wants to incorporate the update into the main project. You probably don’t want to give them push-access to your central repository, since they could start pushing all sorts of random snapshots, and you would effectively lose control of the project.

But, what you can do is tell the contributor to push the changes to *their own* public repository. Then, you can pull their bug fix into your private repository to ensure it doesn’t contain any undeclared code. If you approve their contributions, all you have to do is merge them into a local branch and push it to the main repository, just like we did in the previous module. You’ve become an *integrator*, in addition to an ordinary developer:



The integrator workflow

In this module, we'll experience all of this first-hand by creating a free public repository on [Bitbucket.org](https://bitbucket.org) and incorporating a contribution from an anonymous developer named John. Bitbucket is a DVCS hosting provider that makes it very easy to set up a Git repository and start collaborating with a team of developers.

[Download the repositories for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repositories from the above link, uncompress them, and you're good to go.

Create a Bitbucket Account

The first part of this module will walk you through setting up a Bitbucket account. Navigate your web browser to [Bitbucket.org](https://bitbucket.org) and sign up for a free account.



The Bitbucket logo

You can choose any username for your account, but the email address should match the one you assigned to your Git installation with `git config` in [The Basics](#). If you need to change your email, you can run another `git config -global user.email you@example.com` command.

Create a Public Repository (You)

To create our first networked Git repository, log into your Bitbucket account, and navigate to *Repositories > Create repository*. Use `my-git-repo` as the *Repository Name*, and anything you like for the *Description* field. Since this is just an example project, go ahead and uncheck the *This is a private repository* field. Select *HTML/CSS* for the *Language* field, then go ahead and click *Create repository*.

Create a new repository

Name*

Description

Access level ☐ This is a private repository

Repository type ☒ Git
☐ Mercurial

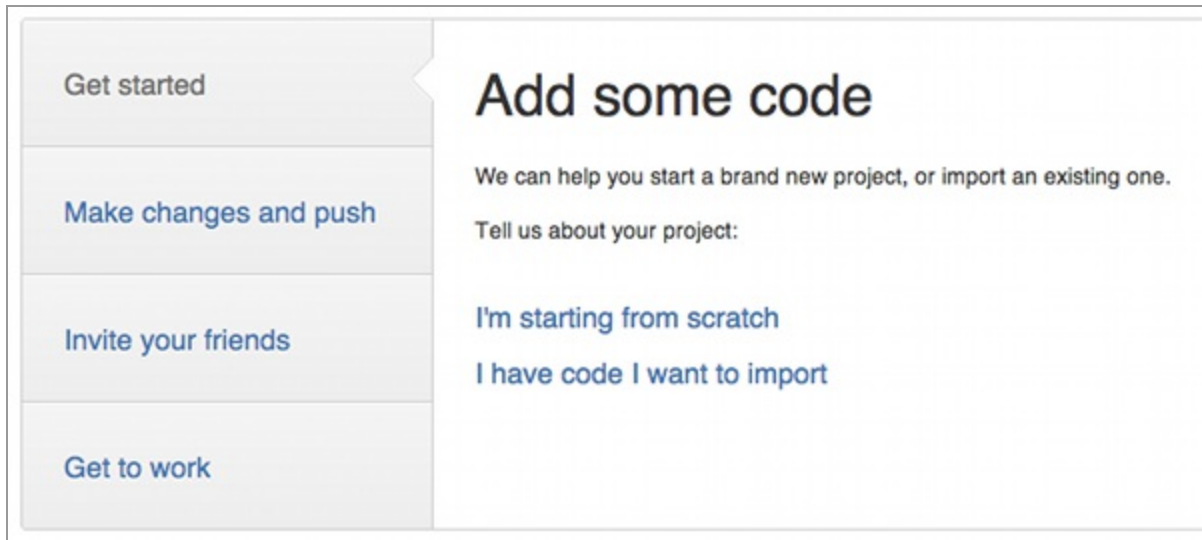
Project management ☐ Issue tracking
☐ Wiki

Language

Bitbucket's new repository form

Essentially, we just ran `git init --bare` on a Bitbucket server. We can now push to and fetch from this repository as we did with `central-repo.git` in the previous module.

After initialization, Bitbucket offers some helpful instructions, but don't follow them just yet—we'll walk through importing an existing repository in the next section.



Bitbucket's setup instructions

Push to the Public Repository (You)

Before populating this new repository with our existing `my-git-repo` project, we first need to point our origin remote to the Bitbucket repository. Be sure to change the `<username>` portion to your actual Bitbucket username.

```
cd /path/to/my-git-repo
git remote rm origin
git remote add origin https://<username>@bitbucket.org/<username>/my-git-repo.git
```

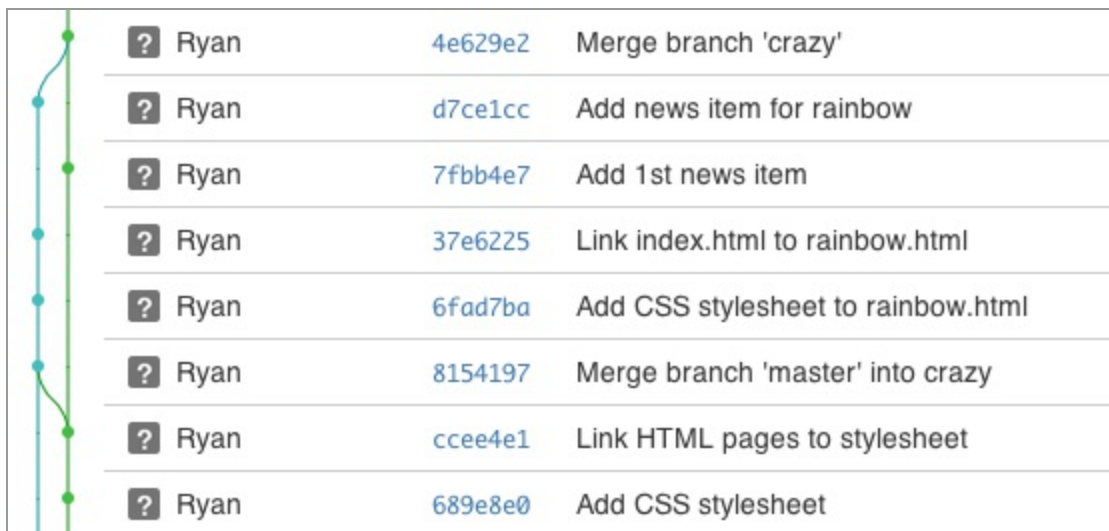
The utility of remotes should be more apparent than in previous modules, as typing the full path to this repository every time we needed to interact with it would be rather tedious.









To populate the remote repository with our existing code, we can use the same push mechanism as with a centralized workflow. When prompted for a password, use the one that you signed up with.

```
git push origin master
```

Browse the Public Repository (You)

We should now be able to see our project on the Bitbucket site. The *Source* tab displays all of the files in the project, and the *Commits* tab contains the entire commit history. Note that the branch structure of the repository is also visualized to the left of each commit.



 Ryan	4e629e2	Merge branch 'crazy'
 Ryan	d7ce1cc	Add news item for rainbow
 Ryan	7fbb4e7	Add 1st news item
 Ryan	37e6225	Link index.html to rainbow.html
 Ryan	6fad7ba	Add CSS stylesheet to rainbow.html
 Ryan	8154197	Merge branch 'master' into crazy
 Ryan	ccee4e1	Link HTML pages to stylesheet
 Ryan	689e8e0	Add CSS stylesheet

Our history in Bitbucket's Commit tab

This repository now serves as the “official” copy of our example website. We’ll tell everyone else to download from this repository, and we’ll push all the changes from our local `my-git-repo` to it. However, it’s important to note that this “official” status is merely a convention. As the master branch is just another branch, our Bitbucket repository is just another repository according to Git.

Having both a public and a private repository for each developer makes it easy to incorporate contributions from third-parties, even if you’ve never met them before.

Clone the Repository (John)

Next, we’re going to pretend to be John, a third-party contributor to our

website. John noticed that we didn't have a pink page and, being the friendly developer that he is, wants to create one for us. We'd like to let him contribute, but we don't want to give him push-access to our entire repository—this would allow him to re-write or even delete all of our hard work.

Fortunately, John knows how to exploit Bitbucket's collaboration potential. He'll start by cloning a copy of our public repository:

```
cd /path/to/my-git-repo
cd ..
git clone http://bitbucket.org/<username>/my-git-repo.git johns-repo
cd johns-repo
```

You should now have another copy of our repository called `johns-repo` in the same folder as `my-git-repo`. This is John's *private* repository—a completely isolated environment where he can safely develop the pink page. Let's quickly configure his name and email:

```
git config user.name "John"
git config user.email john.example@rypress.com
```

Add the Pink Page (John)

Of course, John should be developing his contributions in a dedicated feature branch.

```
git checkout -b pink-page
```

In addition to being a best practice, this makes it easy for the integrator to see what commits to include. When John's done, he'll tell us where to find his repository and what branch the new feature resides in. Then, we'll be able merge his content with minimal effort.

Create the file `pink.html` and add the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Pink Page</title>
  <link rel="stylesheet" href="style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #F0F">The Pink Page</h1>
  <p>Pink is <span style="color: #F0F">girly,
  flirty and fun</span>!</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Add the pink page to the “Navigation” section in `index.html`:

```
<li style="color: #F0F">
  <a href="pink.html">The Pink Page</a>
</li>
```

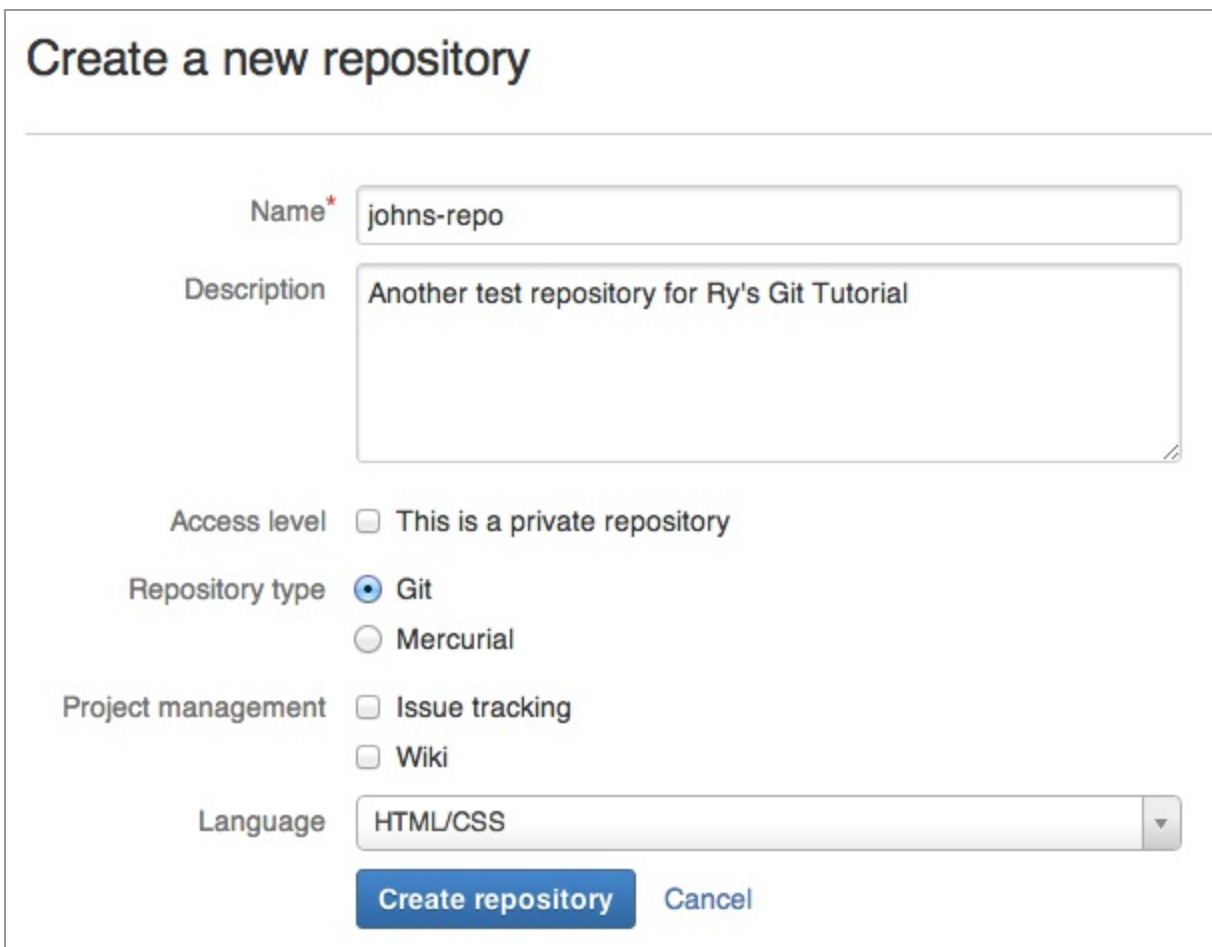
Then, stage and commit the snapshot as normal.

```
git add pink.html index.html
git status
git commit -m "Add pink page"
```

Publish the Pink Page (John)

Now, John needs to publish his contributions to a public repository. Remember that we don't want him to push to *our* public repository, which is stored in his origin remote. In fact, he *can't* push to origin for reasons we'll discuss in a moment.

Instead, he'll create his own Bitbucket repository that we can pull contributions from. In the real world, John would have his own Bitbucket account, but for convenience, we'll just store his public repository under our existing account. Once again, navigate to your Bitbucket home page and click *Repositories > Create repository* to create John's public repository. For the *Name* field, use `johns-repo`.



The screenshot shows the 'Create a new repository' form in Bitbucket. The form has a title 'Create a new repository' at the top. Below the title, there are several fields and options:

- Name***: A text input field containing 'johns-repo'.
- Description**: A text area containing 'Another test repository for Ry's Git Tutorial'.
- Access level**: A checkbox labeled 'This is a private repository' which is unchecked.
- Repository type**: Two radio buttons. 'Git' is selected (indicated by a blue dot), and 'Mercurial' is unselected.
- Project management**: Two checkboxes. 'Issue tracking' and 'Wiki' are both unchecked.
- Language**: A dropdown menu showing 'HTML/CSS'.

At the bottom of the form, there are two buttons: 'Create repository' (a blue button) and 'Cancel' (a light blue button).

John's new repository form

Back in John's private repository, we'll need to add this as a remote:

```
git remote add john-public https://<username>@bitbucket.org/<username>/johns-repo.git
```

This is where John will publish the pink page for us to access. Since he's pushing with HTTPS, he'll need to enter the password for his Bitbucket account (which is actually the password for *your* account).

```
git push john-public pink-page
```

All John needs to do now is tell us the name of the feature branch and send us a link to his repository, which will be:

```
http://bitbucket.org/<username>/johns-repo.git
```

Note that John used a different path for pushing to his public repository than the one he gave us for fetching from it. The most important distinction is the transport protocol: the former used `https://` while the latter used `http://`. Accessing a repository over HTTPS (or SSH) lets you fetch or push, but as we saw, requires a password. This prevents unknown developers from overwriting commits.

On the other hand, fetching over HTTP requires no username or password, but pushing is not possible. This lets anyone fetch from a repository without compromising its security. In the integrator workflow, other developers access your repository via HTTP, while you publish changes via HTTPS. This is also the reason why John can't push to his origin remote.

Of course, if you're working on a private project, anonymous HTTP access would be disabled for that repository.

View John's Contributions (You)

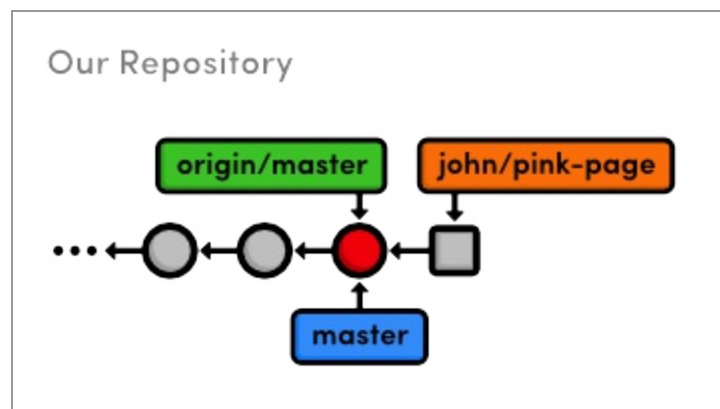
Ok, we're done being John and we're ready to integrate his code into the official project. Let's start by switching back into our repository and adding John's public repository as a remote.

```
cd ../my-git-repo
git remote add john http://bitbucket.org/<username>/johns-repo.git
```

Note that we don't care about anything in John's private repository—the only thing that matters are his published changes. Let's download his branches and take a look at what he's been working on:

```
git fetch john
git branch -r
git log master..john/pink-page --stat
```

We can visualize this history information as the following.



Before merging John's pink-page branch

Let's take a look at his actual changes:

```
git checkout john/pink-page
```

Open up the `pink.html` file to see if it's ok. Remember that John isn't a

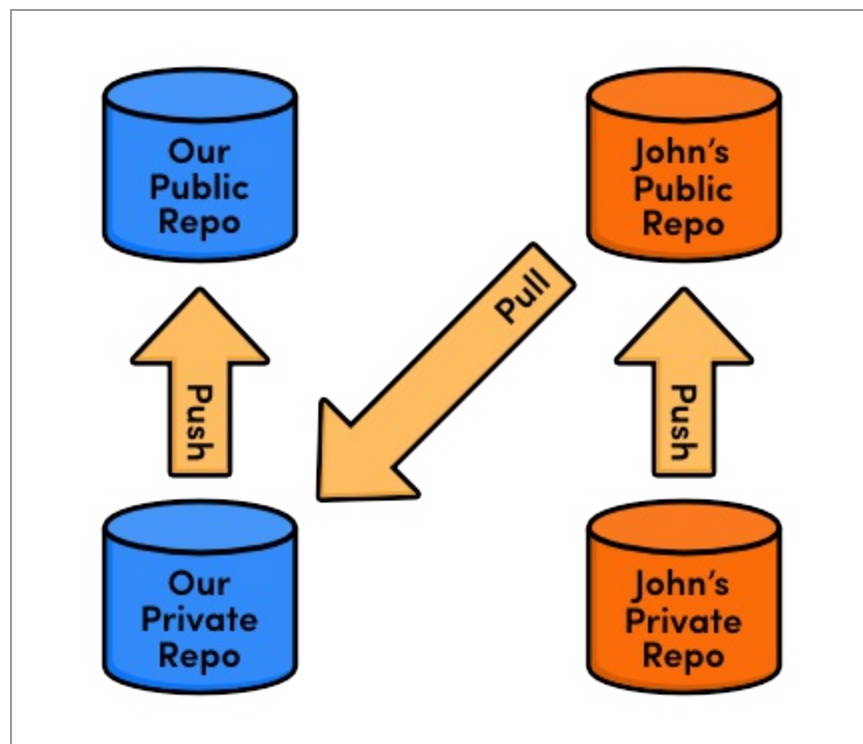
trusted collaborator, and we would normally have no idea what this file might contain. With that in mind, it's incredibly important to verify its contents. **Never blindly merge content from a third-party contributor.**

Integrate John's Contributions (You)

Assuming we approve John's updates, we're now ready to merge it into the project.

```
git checkout master  
git merge john/pink-page
```

Notice that is the exact same way we incorporated Mary's changes in the centralized workflow, except now we're pulling from and pushing to different locations:



The integrator workflow with John

Furthermore, John’s workflow is just like ours: develop in a local, private repository, then push changes to the public one. The integrator workflow is merely a standardized way of organizing the collaboration effort—nothing has changed about how we develop locally, and we’re using the same Git commands as we have been for the last few modules.

Publish John’s Contributions (You)

We’ve integrated John’s contribution into our local `my-git-repo` repository, but no one else knows what we’ve done. It’s time to publish our master branch again.

```
git push origin master
```

Since we designated our public Bitbucket repository as the “official” source for our project, everyone (i.e., Mary and John) will now be able to synchronize with it.

Update Mary’s Repository (Mary)

Mary should now be pulling changes from our Bitbucket repository instead of the central one from the previous module. This should be fairly easy for her to configure.

```
cd ../marys-repo
git remote rm origin
git remote add origin http://bitbucket.org/<username>/my-git-repo.git
```

Again, remember to change `<username>` to your Bitbucket account’s username. For the sake of brevity, we’ll do a blind merge to add John’s updates to Mary’s repository (normally, Mary should check what she’s integrating before doing so).

```
git checkout master  
git fetch origin  
git rebase origin/master
```

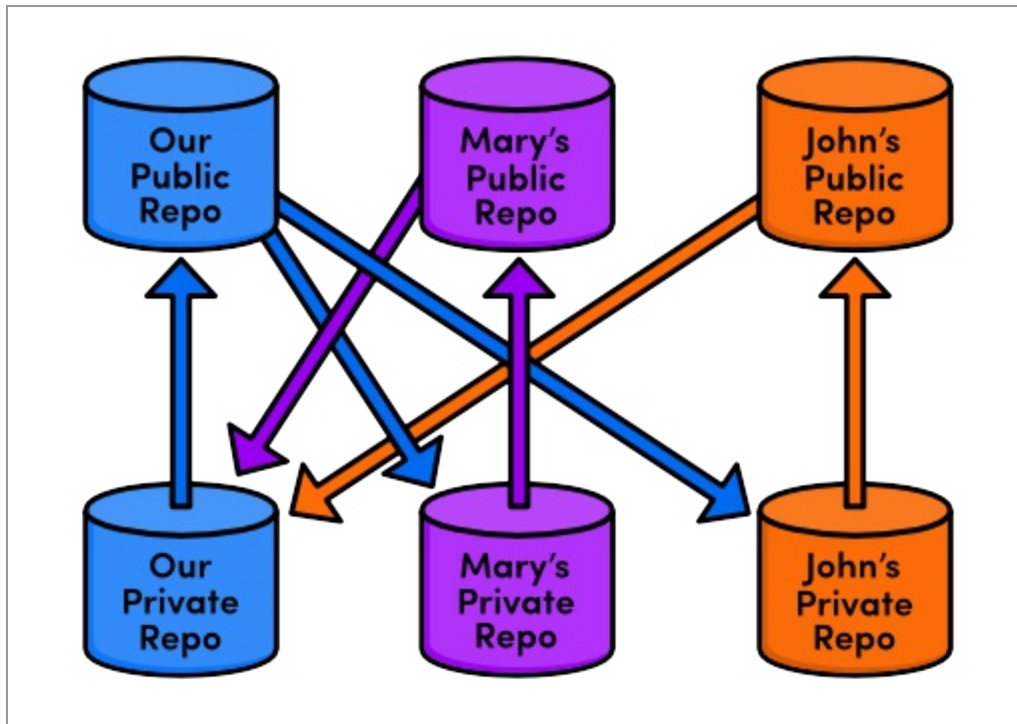
For Mary, it doesn't really matter that the updates came from John. All she has to know is that the “official” master branch moved forward, prompting her to synchronize her private repository.

Update John's Repository (John)

John still needs to incorporate the pink page into his master branch. He should *not* merge directly from his pink-page topic branch because we could have edited his contribution before publishing it or included other contributions along with it. Instead, he'll pull from the “official” master:

```
cd ../johns-repo  
git checkout master  
git fetch origin  
git rebase origin/master
```

If John had updated master directly from his local pink-page, it could have wound up out-of-sync from the main project. For this reason, the integrator workflow requires that everyone *pull* from a single, official repository, while they all *push* to their own public repositories:



The integrator workflow with many developers

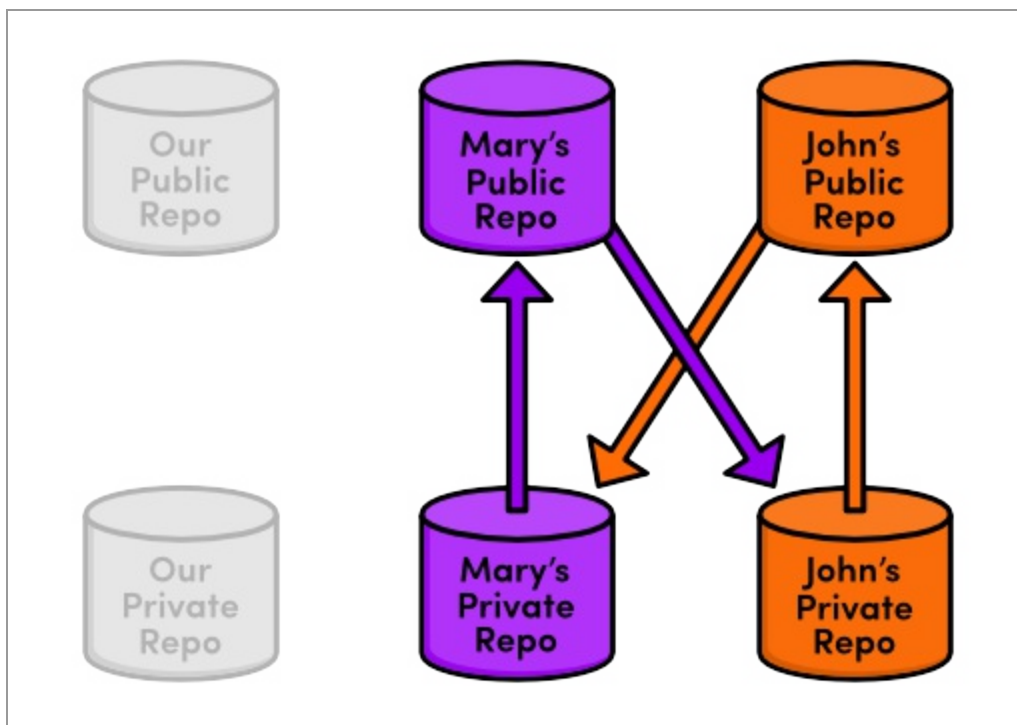
In this way, additions from one contributor can be approved, integrated, and made available to everyone without interrupting anyone's independent developments.

Conclusion

Using the integrator workflow, our private development process largely remains the same (develop a feature branch, merge it into master, and publish it). But, we've added an additional task: incorporating changes from third-party contributors. Luckily, this doesn't require any new skills—just access to a few more remote repositories.

While this setup forces us to keep track of more remotes, it also makes it much, much easier to work with a large number of developers. You'll never have to worry about security using an integrator workflow because you'll still be the only one with access to the "official" repository.

There's also an interesting side-effect to this kind of security. By giving each developer their own public repository, the integrator workflow creates a more stable development environment for open-source software projects. Should the lead developer stop maintaining the “official” repository, any of the other participants could take over by simply designating their public repository as the new “official” project. This is part of what makes Git a *distributed* version control system: there is no single central repository that Git forces everyone to rely upon.



John/Mary taking over project maintenance

In the next module, we'll take a look at an even more flexible way to share commits. This low-level approach will also give us a better understanding of how Git internally manages our content.

Patch Workflows

Thus far, all of the collaboration workflows we've seen rely heavily on branches. For example, in the last module, a contributor had to publish an entire *branch* for us to merge into our project. However, it's also possible to communicate directly on the *commit* level using a **patch**.

A patch file represents a single set of changes (i.e., a commit) that can be applied to any branch, in any order. In this sense, the patch workflow is akin to interactive rebasing, except you can easily share patches with other developers. This kind of communication lessens the importance of a project's branch structure and gives complete control to the project maintainer (at least with regards to incorporating contributions).

Integrating on the commit level will also give us a deeper understanding of how a Git repository records project history.

[Download the repositories for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repositories from the above link, uncompress them, and you're good to go. If you've set up a Bitbucket account, you should also run the following commands to configure the downloaded repositories:

```
cd /path/to/my-git-repo
git remote add origin https://<username>@bitbucket.org/<username>/my-git-repo.git
cd ../marys-repo
git remote add origin http://bitbucket.org/<username>/my-git-repo.git
```

Change the Pink Page (Mary)

We'll begin by pretending to be Mary again. Mary didn't like the pink page

that John contributed and wants to change it.

```
cd /path/to/marys-repo  
git checkout -b pink-page
```

Developing in a new branch not only gives Mary an isolated environment, it also makes it easier to create a series of patches once she's done editing the pink page. Find these lines in `pink.html`:

```
<p>Pink is <span style="color: #F0F">girly,  
flirty and fun</span>!</p>
```

and change them to the following.

```
<p>Only <span style="color: #F0F">real men</span> wear pink!</p>
```

Stage and commit the update as normal.

```
git commit -a -m "Change pink to a manly color"
```

Note that Mary's local development process doesn't change at all. Patches—like the centralized and integrator workflows—are merely a way to share changes amongst developers. It has little effect on the core Git concepts introduced in the first portion of this tutorial.

Create a Patch (Mary)

Mary can create a patch from the new commit using the `git format-patch` command.

```
git format-patch master
```

This creates a file called `0001-Change-pink-to-a-manly-color.patch` that contains enough information to re-create the commit from the last step. The `master` parameter tells Git to generate patches for every commit in the current branch that's missing from `master`.

Open up the patch file in a text editor. As shown by the addresses in the top of the file, it's actually a complete email. This makes it incredibly easy to send patches to other developer. Further down, you should see the following.

```
index 98e10a1..828dd1a 100644
--- a/pink.html
+++ b/pink.html
@@ -7,8 +7,7 @@
</head>
<body>
  <h1 style="color: #F0F">The Pink Page</h1>
- <p>Pink is <span style="color: #F0F">girly,
-  flirty and fun</span>!</p>
+ <p>Only <span style="color: #F0F">real men</span> wear pink!</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
```

This unique formatting is called a **diff**, because it shows the *difference* between two versions of a file. In our case, it tells us what happened to the `pink.html` file between the `98e10a1` and `828dd1a` commits (your patch will contain different commit ID's). The `-7,8 +7,7` portion describes the lines affected in the respective versions of the file, and the rest of the text shows us the content that has been changed. The lines beginning with `-` have been deleted in the new version, and the line starting with `+` has been added.

While you don't have to know the ins-and-outs of diffs to make use of

patches, you do need to understand that a single patch file represents a complete commit. And, since it's a normal file (and also an email), it's much easier to pass around than a Git branch.

Delete the patch file for now (we'll re-create it later).

Add a Pink Block (Mary)

Before learning how to turn patches back into commits, Mary will add one more snapshot.

In `pink.html`, add the following on the line after the `<meta>` tag.

```
<style>
  div {
    width: 300px;
    height: 50px;
  }
</style>
```

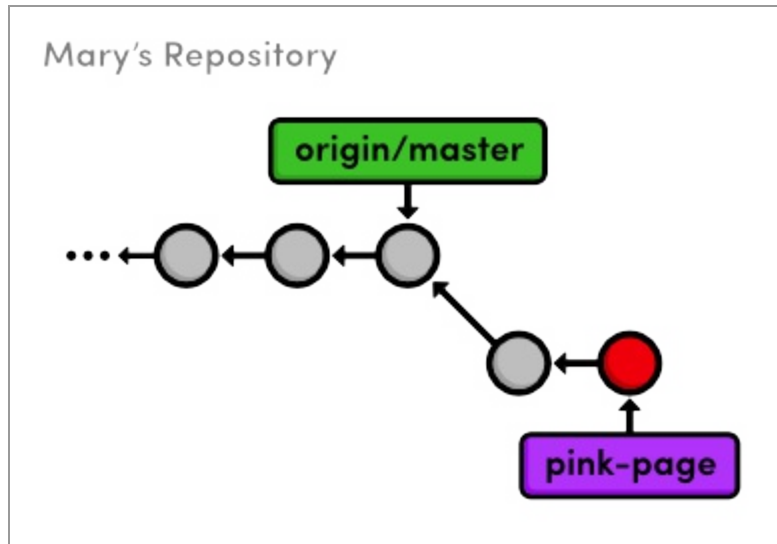
And, add the next line of HTML after `only real men wear pink!`:

```
<div style="background-color: #F0F"></div>
```

Stage and commit the snapshot.

```
git commit -a -m "Add a pink block of color"
```

Mary's repository now contains two commits after the tip of master:



Adding two commits on the pink-page branch

Create Patch of Entire Branch (Mary)

Mary can use the same command as before to generate patches for all the commits in her pink-page branch.

```
git format-patch master
```

The first patch is the exact same as we previously examined, but we also have a new one called `0002-Add-a-pink-block-of-color.patch`. Note that the first line of the commit message will always be used to make a descriptive filename for the patch. You should find the following diff in the second patch.

```
index 828dd1a..2713b10 100644
--- a/pink.html
+++ b/pink.html
@@ -4,10 +4,17 @@
<title>The Pink Page</title>
<link rel="stylesheet" href="style.css" />
```

```
<meta charset="utf-8" />
+ <style>
+   div {
+     width: 300px;
+     height: 50px;
+   }
+ </style>
</head>
<body>
  <h1 style="color: #F0F">The Pink Page</h1>
  <p>Only <span style="color: #F0F">real men</span> wear pink!</p>
+ <div style="background-color: #F0F"></div>

  <p><a href="index.html">Return to home page</a></p>
</body>
```

This is the same formatting as the first patch, except its lack of `-` lines indicate that we only added HTML during the second commit. As you can see, this patch is really just a machine-readable summary of our actions from the previous section.

Mail the Patches (Mary)

Now that Mary has prepared a series of patches, she can send them to the project maintainer (us). In the typical patch workflow, she would send them via email using one of the following methods:

- Copying and pasting the contents of the patch files into an email client. If she uses this method, Mary would have to make sure that her email application doesn't change the whitespace in the patch upon sending it.
- Sending the patch file as an attachment to a normal email.
- Using the convenient `git send-email` command and specifying a file or a directory of files to send. For example, `git send-email .` will send

all the patches in the current directory. Git also requires some special configurations for this command. Please consult the [official Git documentation](#) for details.

The point is, the .patch files need to find their way into the Git repository of whoever wants to add it to their project. For our example, all we need to do is copy the patches into the my-git-repo directory that represents our local version of the project.

Apply the Patches (You)

Copy the two patch files from marys-repo into my-git-repo. Using the new git am command, we can use these patches to add Mary's commits to our repository.

```
cd ../my-git-repo
git checkout -b patch-integration
git am < 0001-Change-pink-to-a-manly-color.patch
git log master..HEAD --stat
```

First, notice that we're doing our integrating in a new topic branch. Again, this ensures that we won't destroy our existing functionality and gives us a chance to approve changes. Second, the git am command takes a patch file and creates a new commit from it. The log output shows us that our integration branch contains Mary's update, along with her author information.

Let's repeat the process for the second commit.

```
git am < 0002-Add-a-pink-block-of-color.patch
git log master..HEAD --stat
```

The `git am` command is configured to read from something called “standard input,” and the `<` character is how we can turn a file’s contents into standard input. As it’s really more of an operating system topic, you can just think of this syntax as a quirk of the `git am` command.

After applying this patch, our integration branch now looks exactly like Mary’s `pink-page` branch. We applied Mary’s patches in the same order she did, but that didn’t necessarily have to be the case. The whole idea behind patches is that they let you isolate a commit and move it around as you please.

Integrate the Patches (You)

Once again, we’re in the familiar situation of integrating a topic branch into the stable `master` branch.

```
git checkout master
git merge patch-integration
git branch -d patch-integration
git clean -f
git push origin master
```

Mary’s updates are now completely integrated into our local repository, so we can get rid of the patch files with `git clean`. This was also an appropriate time to push changes to the public repository so other developers can access the most up-to-date version of the project.

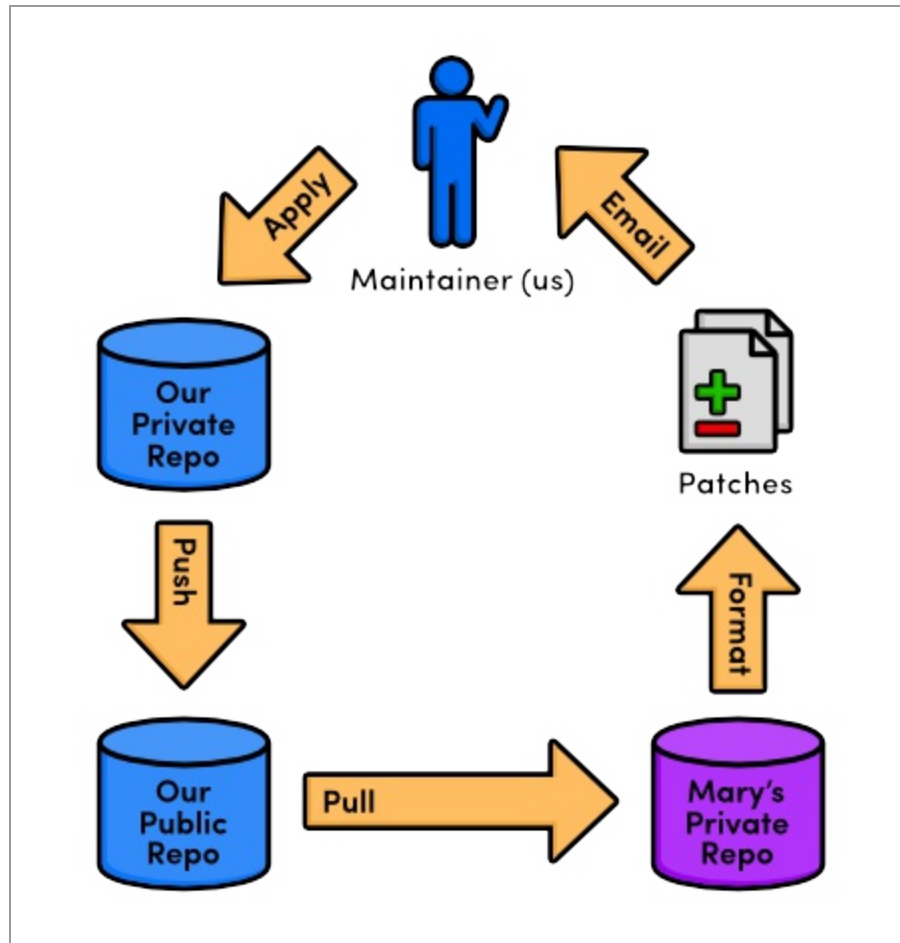
Update Mary’s Repository (Mary)

Mary might be tempted to merge her `pink-page` branch directly into her `master` branch, but this would be a mistake. Her `master` branch *must* track the “official” repository’s `master`, as discussed in the previous module.


```
cd ../marys-repo
git checkout master
git fetch origin
git rebase origin/master
git branch -D pink-page
git clean -f
```

Patches are a convenient way to share commits amongst developers, but the patch workflow still requires an “official” repository that contains *everybody’s* changes. What would have happened if Mary wasn’t the only one sending patches to us? We may very well have applied several different patches or applied Mary’s contributions in a different order. Using Mary’s pink-page to update her master branch would completely ignore all these updates.

Taking this into consideration, our final patch workflow resembles the following.



The patch workflow

Conclusion

Whereas remote repositories are a way to share entire *branches*, patches are a way to send individual *commits* to another developer. Keep in mind that patches are usually only sent to a project maintainer, who then integrates them into the “official” project for all to see. It would be impossible for everyone to communicate using only patches, as no one would be applying them in the same order. Eventually, everyone’s project history would look entirely different.

In many ways, patches are a simpler way to accept contributions than the integrator workflow from the previous module. Only the project maintainer

needs a public repository, and he'll never have to peek at anyone else's repository. From the maintainer's perspective, patches also provide the same security as the integrator workflow: he still won't have to give anyone access to his "official" repository. But, now he won't have to keep track of everybody's remote repositories, either.

As a programmer, you're most likely to use patches when you want to fix a bug in someone else's project. After fixing it, you can send them a patch of the resulting commit. For this kind of one-time-fix, it's much more convenient for you to generate a patch than to set up a public Git repository.

This module concludes our discussion of the standard Git workflows. Hopefully, you now have a good idea of how Git can better manage your personal and professional software projects using a centralized, integrator, or patch workflow. In the next module, we'll switch gears and introduce a variety of practical Git commands.

Quick Reference

```
git format-patch <branch-name>
```

Create a patch for each commit contained in the current branch but not in <branch-name>. You can also specify a commit ID instead of <branch-name>.

```
git am < <patch-file>
```

Apply a patch to the current branch.

Tips & Tricks

This module presents a broad survey of useful Git utilities. We'll take a step back from the theoretical aspects of Git and focus on common tasks like preparing a project for release and backing up a repository. While working through this module, your goal shouldn't be to master all of these miscellaneous tools, but rather to understand why they were created and when they might come in handy.

[Download the repositories for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repositories from the above link, uncompress them, and you're good to go.

Archive The Repository

First, let's export our repository into a ZIP archive. Run the following command in your local copy of `my-git-repo`.

```
git archive master --format=zip --output=../website-12-10-2012.zip
```

Or, for Unix users that would prefer a tarball:

```
git archive master --format=tar --output=../website-12-10-2012.tar
```

This takes the current `master` branch and places all of its files into a ZIP archive (or a tarball), omitting the `.git` directory. Removing the `.git` directory removes all version control information, and you're left with a single snapshot of your project.

You can send the resulting archive to a client for review, even if they don't have Git installed on their machine. This is also an easy way to create Git-

independent backups of important revisions, which is always a good idea.

Bundle the Repository

Similar to the `git archive` command, `git bundle` turns a repository into a single file. However, in this case, the file retains the versioning information of the entire project. Try running the following command.

```
git bundle create ../repo.bundle master
```

It's like we just pushed our master branch to a remote, except it's contained in a file instead of a remote repository. We can even clone it using the same `git clone` command:

```
cd ..  
git clone repo.bundle repo-copy -b master  
cd repo-copy  
git log  
cd ../my-git-repo
```

The log output should show you the entire history of our master branch, and `repo.bundle` is also the origin remote for the new repository. This is the exact behavior we encountered when cloning a “normal” Git repository.

Bundles are a great way to backup entire Git repositories (not just an isolated snapshot like `git archive`). They also let you share changes without a network connection. For example, if you didn't want to configure the SSH accounts for a private Git server, you could bundle up the repository, put it on a jump drive, and walk it over to your co-worker's computer. Of course, this could become a bit tiresome for active projects.

We won't be needing the `repo.bundle` file and `repo-copy` folder, so go

ahead and delete them now.

Ignore a File

Remember that Git doesn't automatically track files because we don't want to record generated files like C binaries or compiled Python modules. But, seeing these files under the "Untracked files" list in `git status` can get confusing for large projects, so Git lets us ignore content using a special text file called `.gitignore`. Each file or directory stored in `.gitignore` will be invisible to Git.

Let's see how this works by creating a file called `notes.txt` to store some personal (private) comments about the project. Add some text to it and save it, then run the following.

```
git status
```

As expected, this will show `notes.txt` in the "Untracked files" section. Next, create a file called `.gitignore` in the `my-git-repo` folder and add the following text to it. Windows users can create a file that starts with a period by executing the `touch .gitignore` command in Git Bash (you should also make sure hidden files are visible in your file browser).

```
notes.txt
```

Run another `git status` and you'll see that the `notes` file no longer appears under "Untracked files", but `.gitignore` does. This is a common file for Git-based projects, so let's add it to the repository.

```
git add .gitignore
git commit -m "Add .gitignore file"
git status
```

You can also specify entire directories in `.gitignore` or use the `*` wildcard to ignore files with a particular extension. For example, the following is a typical `.gitignore` file for a simple C project. It tells Git to overlook all `.o`, `.out`, and `.exe` files in the repository.

```
*.o  
*.out  
*.exe
```

Stash Uncommitted Changes

Next, we'll take a brief look at **stashing**, which conveniently “stashes” away uncommitted changes. Open up `style.css` and change the `h1` element to:

```
h1 {  
  font-size: 32px;  
  font-family: "Times New Roman", serif;  
}
```

Now let's say we had to make an emergency fix to our project. We don't want to commit an unfinished feature, and we also don't want to lose our current CSS addition. The solution is to temporarily remove these changes with the `git stash` command:

```
git status  
git stash  
git status
```

Before the stash, `style.css` was listed as “Changed by not updated.” The `git stash` command hid these changes, giving us a clean working directory.

We're now able to switch to a new hotfix branch to make our important updates—without having to commit a meaningless snapshot just to save our current state.

Let's pretend we've completed our emergency update and we're ready to continue working on our CSS changes. We can retrieve our stashed content with the following command.

```
git stash apply
```

The `style.css` file now looks the same as it did before the stash, and we can continue development as if we were never interrupted. Whereas patches represent a committed snapshot, a stash represents a set of *uncommitted* changes. It takes the uncommitted modifications, stores them internally, then does a `git reset --hard` to give us a clean working directory. This also means that stashes can be applied to *any* branch, not just the one from which it was created.

In addition to temporarily storing uncommitted changes, this makes stashing a simple way to transfer modifications between branches. So, for example, if you ever found yourself developing on the wrong branch, you could stash all your changes, checkout the correct branch, then run a `git stash apply`.

Let's undo these CSS updates before moving on.

```
git reset --hard
```

Hook into Git's Internals

Arguably, Git's most useful configuration options are its **hooks**. A hook is a script that Git executes every time a particular event occurs in a repository. In this section, we'll take a hands-on look at Git hooks by automatically

publishing our website every time someone pushes to the `central-repo.git` repository.

In the `central-repo.git` directory, open the `hooks` directory and rename the file `post-update.sample` to `post-update`. After removing the `.sample` extension, this script will be executed whenever *any* branch gets pushed to `central-repo.git`. Replace the default contents of `post-update` with the following.

```
#!/bin/sh

# Output a friendly message
echo "Publishing master branch!" >&2

# Remove the old `my-website` directory (if necessary)
rm -rf ../my-website

# Create a new `my-website` directory
mkdir ../my-website

# Archive the `master` branch
git archive master --format=tar --output=../my-website.tar

# Uncompress the archive into the `my-website` directory
tar -xf ../my-website.tar -C ../my-website

exit 0
```

While shell scripts are outside the scope of this tutorial, the majority of commands in the above code listing should be familiar to you. In short, this new `post-update` script creates an archive of the master branch, then exports it into a directory called `my-website`. This is our “live” website.

We can see the script in action by pushing a branch to the `central-repo.git` repository.

```
git push ../central-repo.git master
```

After the central repository receives the new master branch, our post-update script is executed. You should see the `Publishing master branch!` message echoed from the script, along with a new `my-website` folder in the same directory as `my-git-repo`. You can open `index.html` in a web browser to verify that it contains all the files from our master branch, and you can also see the intermediate `.tar` archive produced by the hook.

This is a simple, unoptimized example, but Git hooks are infinitely versatile. Each of the `.sample` scripts in the `hooks` directory represents a different event that you can listen for, and each of them can do anything from automatically creating and publishing releases to enforcing a commit policy, making sure a project compiles, and of course, publishing websites (that means no more clunky FTP uploads). Hooks are even configured on a per-repository basis, which means you can run different scripts in your local repository than your central repository.

For a detailed description of the available hooks, please consult the [official Git documentation](#).

View Diffs Between Commits

Up until now, we've been using `git log --stat` to view the changes introduced by new commits. However, this doesn't show us which lines have been changed in any given file. For this level of detail, we need the `git diff` command. Let's take a look at the updates from the `Add a pink block of color` commit:

```
git diff HEAD~2..HEAD~1
```

This will output the diff between the Add a pink block of color commit (HEAD~1) and the one before it (HEAD~2):

```
index 828dd1a..2713b10 100644
--- a/pink.html
+++ b/pink.html
@@ -4,10 +4,17 @@
     <title>The Pink Page</title>
     <link rel="stylesheet" href="style.css" />
     <meta charset="utf-8" />
+   <style>
+     div {
+       width: 300px;
+       height: 50px;
+     }
+   </style>
</head>
<body>
  <h1 style="color: #F0F">The Pink Page</h1>
  <p>Only <span style="color: #F0F">real men</span> wear pink!</p>
+ <div style="background-color: #F0F"></div>

  <p><a href="index.html">Return to home page</a></p>
</body>
```

This diff looks nearly identical to the patches we created in the previous module, and it shows exactly what was added to get from HEAD~2 to HEAD~1. The `git diff` command is incredibly useful for pinpointing contributions from other developers. For example, we could have used the following to view the differences between John's pink-page branch and our master

before merging it into the project in [Distributed Workflows](#).

```
git diff master..john/pink-page
```

This flexible command can also generate a detailed view of our uncommitted changes. Open up `blue.html`, make any kind of change, and save the file.

Then, run `git diff` without any arguments:

```
git diff
```

And, just in case we forgot what was added to the staging area, we can use the `--cached` flag to generate a diff between the staged snapshot and the most recent commit:

```
git add blue.html  
git diff --cached
```

A plain old `git diff` won't output anything after `blue.html` is added to the staging area, but the changes are now visible through the `--cached` flag. These are the three main configurations of the `git diff` command.

Reset and Checkout Files

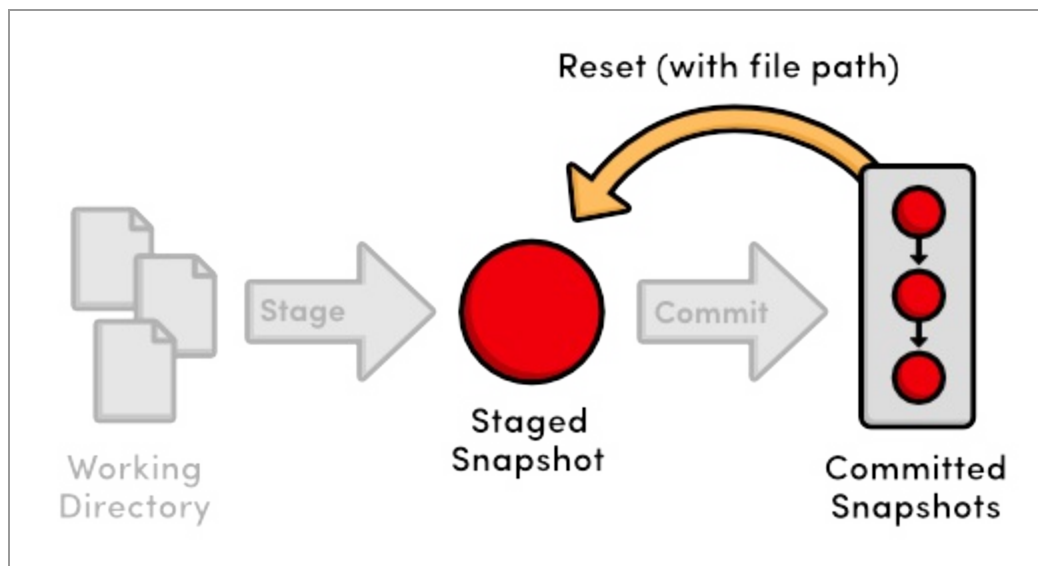
We've used `git reset` and `git checkout` many times throughout this tutorial; however, we've only seen them work with branches/commits. You can also reset and check out individual files, which slightly alters the behavior of both commands.

The `git reset` we're accustomed to moves the current branch to a new commit and optionally updates the working directory to match. But when we pass a file path, `git reset` updates the *staging area* to match the given

commit instead of the working directory, and it doesn't move the current branch pointer. This means we can remove `blue.html` from the staged snapshot with the following command.

```
git reset HEAD blue.html
git status
```

This makes the `blue.html` in the staging area match the version stored in HEAD, but it leaves the working directory and current branch alone. The result is a staging area that matches the most recent commit and a working directory that contains the modified `blue.html` file. In other words, this type of `git reset` can be used to unstage a file:

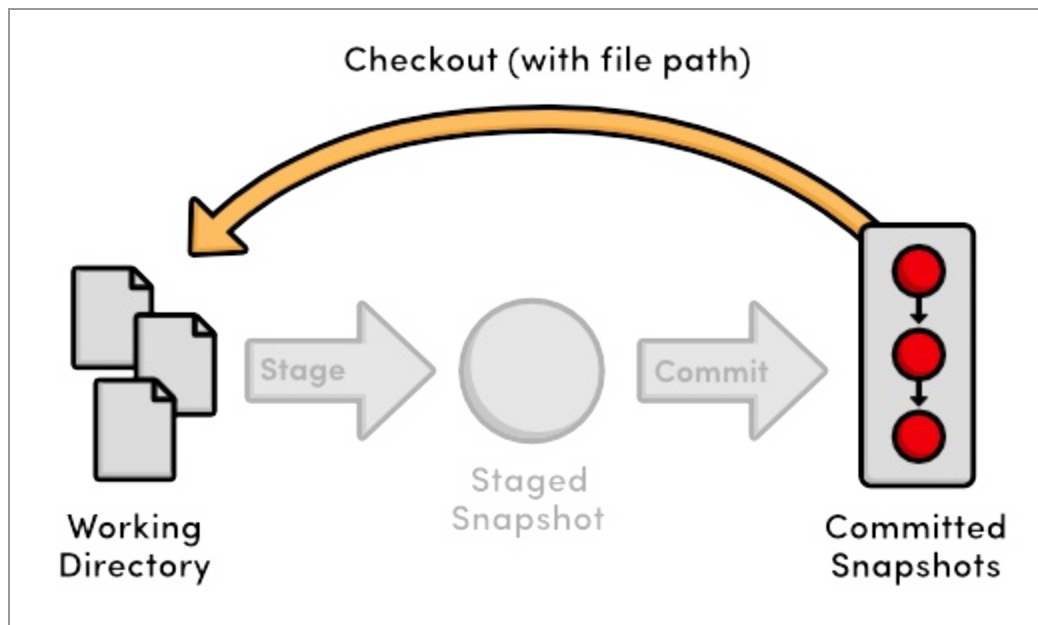


Using `git reset` with a file path

Let's take this one step further with `git checkout`. The `git checkout` we've been using updates the working directory *and* switches branches. If we add a file path to `git checkout`, it narrows its focus to only the specified file and does *not* update the branch pointer. This means that we can “check out” the most recent version of `blue.html` with the following command.

```
git checkout HEAD blue.html  
git status
```

Our `blue.html` file now looks exactly like the version stored in `HEAD`, and we thus have a clean working directory. Passing a file path to `git checkout` reverts that file to the specified commit.



Using `git checkout` with a file path

To summarize the file-path behavior of `git reset` and `git checkout`, both take a committed snapshot as an reference point and make a file in the staging area or the working directory match that reference, respectively.

Aliases and Other Configurations

Typing `git checkout` every time you wanted to see a new branch over the last ten modules has been a bit verbose. Fortunately, Git lets you create **aliases**, which are shortcuts to other commands. Let's create a few aliases for our most common commands:

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

Now, we can use `git co` instead of `git checkout`, `git ci` for committing, and `git br` for listing branches. We can even use `git br <branch-name>` for creating a new branch.

Git stores these aliases in a global config file, similar to the local config file we looked at in Mary's repository (`marys-repo/.git/config`). By default, global configurations reside in `~/.gitconfig`, where the `~` character represents your home directory. This file should resemble the following.

```
[user]
  name = Ryan
  email = ryan.example@rypress.com

[alias]
  co = checkout
  ci = commit
  br = branch
```

Of course, your settings should reflect the name and email you entered in [The Basics](#). As you can see, all of our new aliases are also stored in `.gitconfig`. Let's add a few more useful configurations by modifying this file directly. Append the following to `.gitconfig`.

```
[color]
  status = always

[core]
  editor = gvim
```

This makes sure Git colorizes the output of `git status` and that it uses the gVim text editor for creating commit messages. To use a different editor, simply change `gvim` to the command that opens your editor. For example, Emacs users would use `emacs`, and Notepad users would use `notepad.exe`.

Git includes a long list of configuration options, all of which can be found in the [official manual](#). Note that storing your global configurations in a plaintext file makes it incredibly easy to transfer your settings to a new Git installation: just copy `~/.gitconfig` onto your new machine.

Conclusion

In this module, we learned how to export snapshots, backup repositories, ignore files, stash temporary changes, hook into Git's internals, generate diffs, reset individual files, and create shorter aliases for common commands. While it's impossible to cover all of Git's supporting features in a hands-on guide such as this, I hope that you now have a clearer picture of Git's numerous capabilities.

With all of these convenient features, it's easy to get so caught up in designing the perfect workflow that you lose sight of Git's underlying purpose. As you add new commands to your repertoire, remember that Git should always make it *easier* to develop a software project—never harder. If you ever find that Git is causing more harm than good, don't be scared to drop some of the advanced features and go back to the basics.

The final module will go a long way towards helping you realize the full potential of Git's version control model. We'll explore Git's internal database by manually inspecting and creating snapshots. Equipped with this low-level knowledge, you'll be more than ready to venture out into the reality of Git-based project management.

Quick Reference

```
git archive <branch-name> --format=zip --output=<file>
```

Export a single snapshot to a ZIP archive called <file>.

```
git bundle create <file> <branch-name>
```

Export an entire branch, complete with history, to the specified file.

```
git clone repo.bundle <repo-dir> -b <branch-name>
```

Re-create a project from a bundled repository and checkout <branch-name>.

```
git stash
```

Temporarily stash changes to create a clean working directory.

```
git stash apply
```

Re-apply stashed changes to the working directory.

```
git diff <commit-id>..<commit-id>
```

View the difference between two commits.

```
git diff
```

View the difference between the working directory and the staging area.

```
git diff --cached
```

View the difference between the staging area and the most recent commit.

```
git reset HEAD <file>
```

Unstage a file, but don't alter the working directory or move the current branch.

```
git checkout <commit-id> <file>
```

Revert an individual file to match the specified commit without switching branches.

```
git config --global alias.<alias-name> <git-command>
```

Create a shortcut for a command and store it in the global configuration file.

Plumbing

In [Rewriting History](#), I talked about the internal representation of a Git repository. I may have mislead you a bit. While the reflog, interactive rebasing, and resetting may be more complex features of Git, they are still considered part of the **porcelain**, as is every other command we've covered. In this module, we'll take a look at Git's **plumbing**—the low-level commands that give us access to Git's *true* internal representation of a project.

Unless you start hacking on Git's source code, you'll probably never need to use the plumbing commands presented below. But, manually manipulating a repository will fill in the conceptual details of how Git actually stores your data, and you should walk away with a much better understanding of the techniques that we've been using throughout this tutorial. In turn, this knowledge will make the familiar porcelain commands even more powerful.

We'll start by inspecting Git's object database, then we'll manually create and commit a snapshot using only Git's low-level interface.

[Download the repository for this module](#)

If you've been following along from the previous module, you already have everything you need. Otherwise, download the zipped Git repository from the above link, uncompress it, and you're good to go.

Examine Commit Details

First, let's take a closer look at our latest commit with the `git cat-file plumbing` command.

```
git cat-file commit HEAD
```

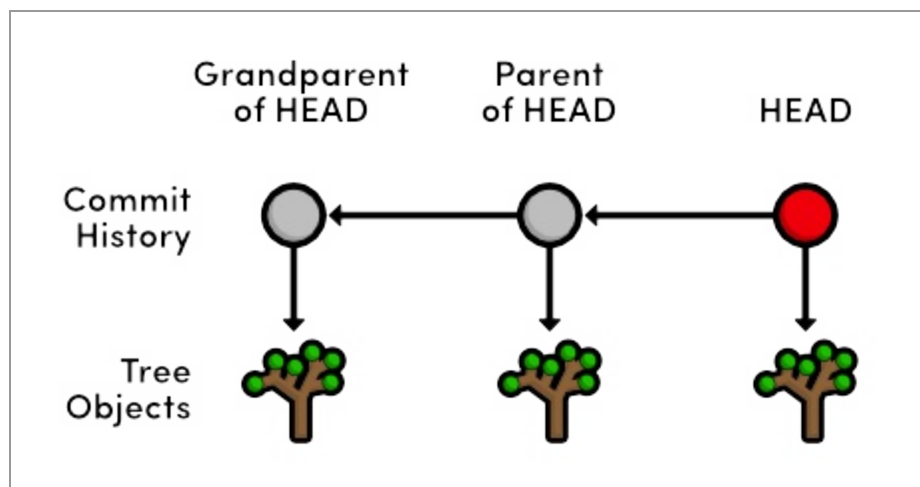
The `commit` parameter tells Git that we want to see a commit object, and as we already know, `HEAD` refers to the most recent commit. This will output the following, although your IDs and user information will be different.

```
tree 552acd444696ccb1c3afe68a55ae8b20ece2b0e6
parent 6a1d380780a83ef5f49523777c5e8d801b7b9ba2
author Ryan <ryan.example@rypress.com> 1326496982 -0600
committer Ryan <ryan.example@rypress.com> 1326496982 -0600

Add .gitignore file
```

This is the complete representation of a commit: a tree, a parent, user data, and a commit message. The user information and commit message are relatively straightforward, but we’ve never seen the *tree* or *parent* values before.

A **tree object** is Git’s representation of the “snapshots” we’ve been talking about since the beginning of this tutorial. They record the state of a directory at a given point, without any notion of time or author. To tie trees together into a coherent project history, Git wraps each one in a **commit object** and specifies a **parent**, which is just another commit. By following the parent of each commit, you can walk through the entire history of a project.



Commit and tree objects

Notice that each commit refers to one and only one tree object. From the `git cat-file` output, we can also infer that trees use SHA-1 checksums for their ID's. This will be the case for all of Git's internal objects.

Examine a Tree

Next, let's try to inspect a tree using the same `git cat-file` command.

Make sure to change 552acd4 to the ID of your tree from the previous step.

```
git cat-file tree 552acd4
```

Unfortunately, trees contain binary data, which is quite ugly when displayed in its raw form. So, Git offers another useful plumbing command:

```
git ls-tree 552acd4
```

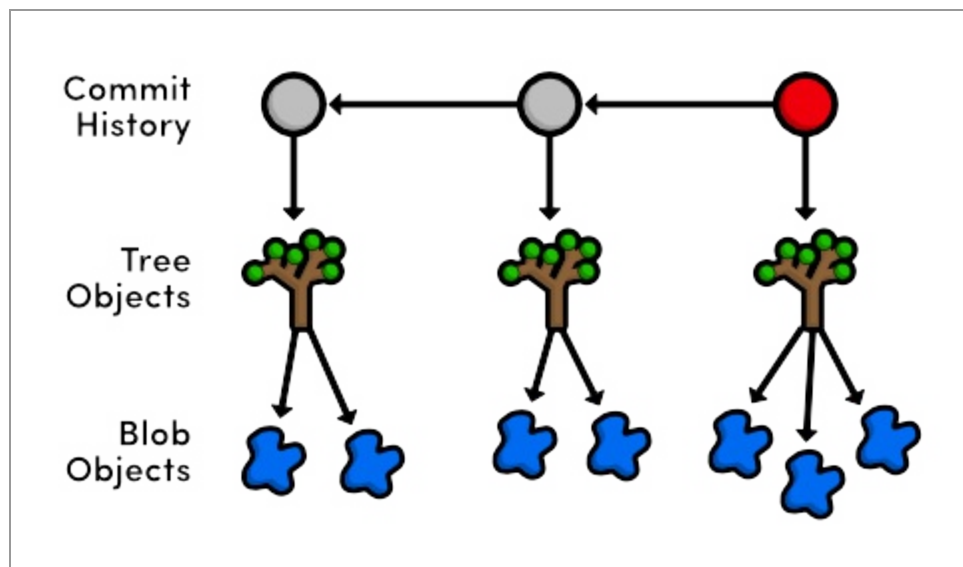
This will output the contents of the tree, which looks an awful lot like a directory listing:

```
100644 blob 99ed0d431c5a19f147da3c4cb8421b5566600449    .gitignore
040000 tree ab4947cb27ef8731f7a54660655afaedaf45444d    about
100644 blob cefb5a651557e135666af4c07c7f2ab4b8124bd7    blue.html
100644 blob cb01ae23932fd9704fdc5e077bc3c1184e1af6b9    green.html
100644 blob e993e5fa85a436b2bb05b6a8018e81f8e8864a24    index.html
100644 blob 2a6deedee35cc59a83b1d978b0b8b7963e8298e9    news-1.html
100644 blob 0171687fc1b23aa56c24c54168cdebaefecf7d71    news-2.html
...
```

By examining the above output, we can presume that “blobs” represent files in our repository, whereas trees represent folders. Go ahead and examine the

about tree with another `git ls-tree` to see if this really is the case. You should see the contents of our about folder.

So, **blob objects** are how Git stores our file data, tree objects combine blobs and other trees into a directory listing, then commit objects tie trees into a project history. These are the only types of objects that Git needs to implement nearly all of the porcelain commands we've been using, and their relationship is summed up as follows:



Commit, tree, and blob objects

Examine a Blob

Let's take a look at the blob associated with `blue.html` (be sure to change the following to the ID next to `blue.html` in *your* tree output).

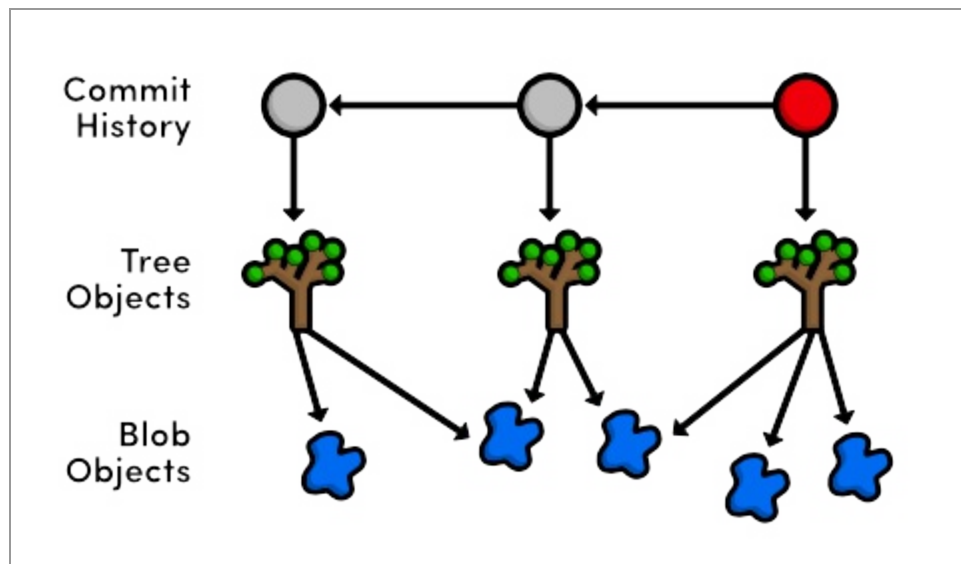
```
git cat-file blob cefb5a6
```

This should display the entire contents of `blue.html`, confirming that blobs really are plain data files. Note that blobs are pure content: there is no mention of a filename in the above output. That is to say, the name

`blue.html` is stored in the *tree that contains the blob*, not the blob itself.

You may recall from [The Basics](#) that an SHA-1 checksum ensures an object's contents is never corrupted without Git knowing about it. Checksums work by using the object's contents to generate a unique character sequence. This not only functions as an identifier, it also guarantees that an object won't be silently corrupted (the altered content would generate a different ID).

When it comes to blob objects, this has an additional benefit. Since two blobs with the same data will have the same ID, Git *must* share blobs across multiple trees. For example, our `blue.html` file hasn't been changed since it was created, so our repository will only have a single associated blob, and all subsequent trees will refer to it. By not creating duplicate blobs for each tree object, Git vastly reduces the size of a repository. With this in mind, we can revise our Git object diagram to the following.



Commit, tree, and shared blob objects

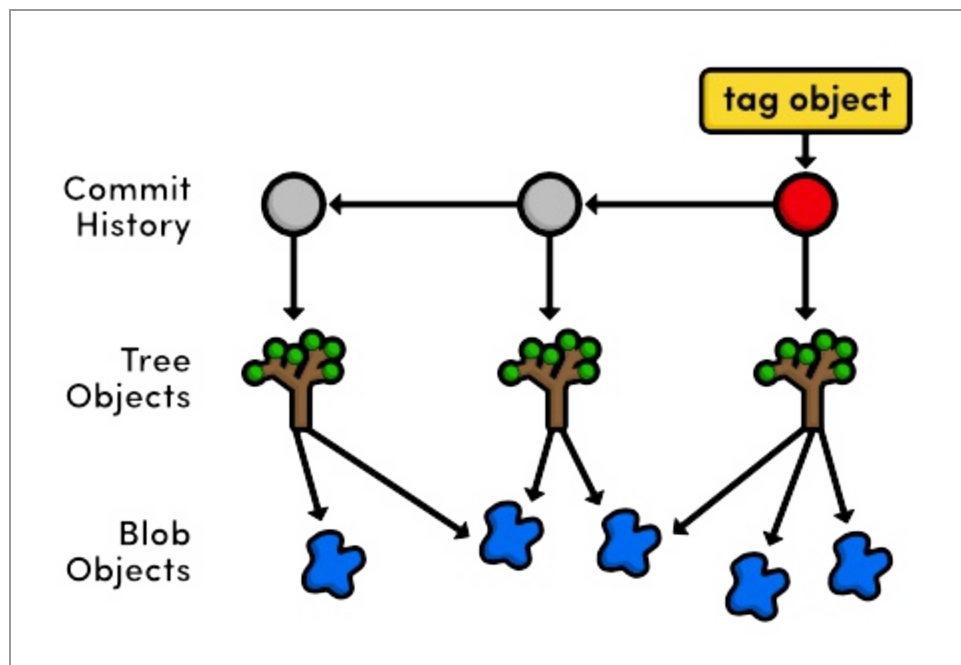
However, as soon as you change a single line in a file, Git must create a new blob object because its contents will have changed, resulting in a new SHA-1 checksum.

Examine a Tag

The fourth and final type of Git object is the **tag object**. We can use the same `git cat-file` command to show the details of a tag.

```
git cat-file tag v2.0
```

This will output the commit ID associated with `v2.0`, along with the tag's name, author, creation date, and message. The straightforward relationship between tags and commits gives us our finalized Git object diagram:



Commit, tree, blob, and tag objects

Inspect Git's Branch Representation

We now have the tools to fully explore Git's branch representation. Using the `-t` flag, we can determine what kind of object Git uses for branches.

```
git cat-file -t master
```

That's right, a branch is just a reference to a commit object, which means we can view it with a normal `git cat-file`.

```
git cat-file commit master
```

This will output the exact same information as our original `git cat-file commit HEAD`. It seems that both the master branch and HEAD are simply references to a commit object.

Using a text editor, open up the `.git/refs/heads/master` file. You should find the commit checksum of the most recent commit, which you can view with `git log -n 1`. This single file is all Git needs to maintain the master branch—all other information is extrapolated through the commit object relationships discussed above.

The HEAD reference, on the other hand, is recorded in `.git/HEAD`. Unlike the branch tips, HEAD is not a direct link to a commit. Instead, it refers to a branch, which Git uses to figure out which commit is currently checked out. Remember that a detached HEAD state occurred when HEAD did not coincide with the tip of any branch. Internally, all this means to Git is that `.git/HEAD` doesn't contain a local branch. Try checking out an old commit:

```
git checkout HEAD~1
```

Now, `.git/HEAD` should contain a commit ID instead of a branch. This tells Git that we're in a detached HEAD state. Regardless of what state you're in, the `git checkout` command will always record the checked-out reference in `.git/HEAD`.

Let's get back to our master branch before moving on:


```
git checkout master
```

Explore the Object Database

While we have a basic understanding of Git's object interaction, we have yet to explore where Git keeps all of these objects. In your `my-git-repo` repository, open the folder `.git/objects`. This is Git's object database.

Each object, regardless of type, is stored as a file, using its SHA-1 checksum as the filename (sort of). But, instead of storing all objects in a single folder, they are split up using the first two characters of their ID as a directory name, resulting in an object database that looks something like the following.

```
00  10  28  33  3e  51  5c  6e  77  85  95  f7
01  11  29  34  3f  52  5e  6f  79  86  96  f8
02  16  2a  35  41  53  63  70  7a  87  98  f9
03  1c  2b  36  42  54  64  71  7c  88  99  fa
0c  26  30  3c  4e  5a  6a  75  83  91  a0  info
0e  27  31  3d  50  5b  6b  76  84  93  a2  pack
```

For example, an object with the following ID:

```
7a52bb857229f89bffa74134ee3de48e5e146105
```

is stored in a folder called `7a`, using the remaining characters (`52bb8...`) as a filename. This gives us an object ID, but before we can inspect items in the object database, we need to know what type of object it is. Again, we can use the `-t` flag:

```
git cat-file -t 7a52bb8
```

Of course, change the object ID to an object from *your* database (don't forget to combine the folder name with the filename to get the full ID). This will output the type of commit, which we can then pass to a normal call to `git cat-file`.

```
git cat-file blob 7a52bb8
```

My object was a blob, but yours may be different. If it's a tree, remember to use `git ls-tree` to turn that ugly binary data into a pretty directory listing.

Collect the Garbage

As your repository grows, Git may automatically transfer your object files into a more compact form known as a “pack” file. You can force this compression with the garbage collection command, but beware: this command is undo-able. If you want to continue exploring the contents of the `.git/objects` folder, you should do so before running the following command. Normal Git functionality will not be affected.

```
git gc
```

This compresses individual object files into a faster, smaller pack file and removes dangling commits (e.g., from a deleted, unmerged branch).

Of course, all of the same object ID's will still work with `git cat-file`, and all of the porcelain commands will remain unaffected. The `git gc` command only changes Git's storage mechanism—not the contents of a repository. Running `git gc` every now and then is usually a good idea, as it keeps your repository optimized.

Add Files to the Index

Thus far, we’ve been discussing Git’s low-level representation of committed snapshots. The rest of this module will shift gears and use more “plumbing” commands to manually prepare and commit a new snapshot. This will give us an idea of how Git manages the working directory and the staging area.

Create a new file called `news-4.html` in `my-git-repo` and add the following HTML.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Indigo Invasion</title>
  <link rel="stylesheet" href="style.css" />
  <meta charset="utf-8" />
</head>
<body>
  <h1 style="color: #A0C">Indigo Invasion</h1>
  <p>Last week, a coalition of Asian designers, artists,
  and advertisers announced the official color of Asia:
  <span style="color: #A0C">Indigo</span>.</p>

  <p><a href="index.html">Return to home page</a></p>
</body>
</html>
```

Then, update the `index.html` “News” section to match the following.

```
<h2 style="color: #C00">News</h2>
<ul>
  <li><a href="news-1.html">Blue Is The New Hue</a></li>
  <li><a href="rainbow.html">Our New Rainbow</a></li>
  <li><a href="news-2.html">A Red Rebellion</a></li>
```

```
<li><a href="news-3.html">Middle East's Silent Beast</a></li>
<li><a href="news-4.html">Indigo Invasion</a></li>
</ul>
```

Instead of `git add`, we'll use the low-level `git update-index` command to add files to the staging area. The **index** is Git's term for the staged snapshot.

```
git status
git update-index index.html
git update-index news-4.html
```

The last command will throw an error—Git won't let you add a new file to the index without explicitly stating that it's a new file:

```
git update-index --add news-4.html
git status
```

We've just moved the working directory into the index, which means we have a snapshot prepared for committal. However, the process won't be quite as simple as a mere `git commit`.

Store the Index in the Database

Remember that all commits refer to a tree object, which represents the snapshot for that commit. So, before creating a commit object, we need to add our index (the staged tree) to Git's object database. We can do this with the following command.

```
git write-tree
```

This command creates a tree object from the index and stores it in `.git/objects`. It will output the ID of the resulting tree (yours may be

different):

```
5f44809ed995e5b861acf309022ab814ceaaafd6
```

You can examine your new snapshot with `git ls-tree`. Keep in mind that the only new blobs created for this commit were `index.html` and `news-4.html`. The rest of the tree contains references to existing blobs.

```
git ls-tree 5f44809
```

So, we have our tree object, but we have yet to add it to the project history.

Create a Commit Object

To commit the new tree object, we need to manually figure out the ID of the parent commit.

```
git log --oneline -n 1
```

This will output the following line, though your commit ID will be different. We'll use this ID to specify the parent of our new commit object.

```
3329762 Add .gitignore file
```

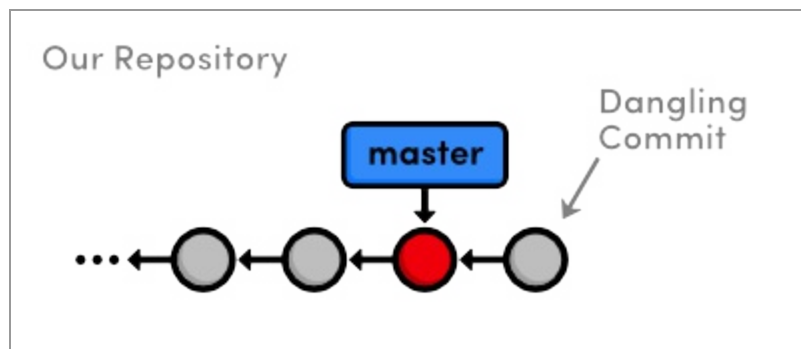
The `git commit-tree` command creates a commit object from a tree and a parent ID, while the author information is taken from an environment variable set by Git. Make sure to change `5f44809` to your tree ID, and `3329762` to your most recent commit ID.

```
git commit-tree 5f44809 -p 3329762
```

This command will wait for more input: the commit message. Type `Add 4th news item` and press `Enter` to create the commit message, then `Ctrl-Z` and `Enter` for Windows or `Ctrl-D` for Unix to specify an “End-of-file” character to end the input. Like the `git write-tree` command, this will output the ID of the resulting commit object.

```
c51dc1b3515f9f8e80536aa7acb3d17d0400b0b5
```

You’ll now be able to find this commit in `.git/objects`, but neither `HEAD` nor the branches have been updated to include this commit. It’s a *dangling commit* at this point. Fortunately for us, we know where Git stores its branch information.



Creating a dangling commit

Update HEAD

Since we’re not in a detached `HEAD` state, `HEAD` is a reference to a branch. So, all we need to do to update `HEAD` is move the `master` branch forward to our new commit object. Using a text editor, replace the contents of `.git/refs/heads/master` with the output from `git commit-tree` in the previous step.

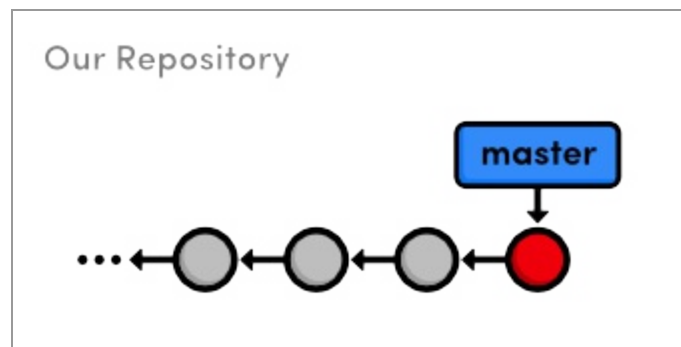
If this file seems to have disappeared, don’t fret! This just means that the `git gc` command packed up all of our branch references into single file. Instead

of `.git/refs/heads/master`, open up `.git/packed-refs`, find the line with `refs/heads/master`, and change the ID to the left of it.

Now that our master branch points to the new commit, we should be able to see the `news-4.html` file in the project history.

```
git log -n 2
```

The last four sections explain everything that happens behind the scenes when we execute `git commit -a -m "Some Message"`. Aren't you glad you won't have to use Git's plumbing ever again?



Manually updating the master branch

Conclusion

After this module, you hopefully have a solid grasp of the object database that underlies almost every Git command. We examined commits, trees, blobs, tags, and branches, and we even created a commit object from scratch. All of this was meant to give you a deeper understanding of Git's porcelain commands, and you should now feel ready to adapt Git to virtually any task you could possibly demand from a version control system.

As you migrate these skills to real-world projects, remember that Git is merely a tool for tracking your files, not a cure-all for managing software projects. No amount of intimate Git knowledge can make up for a haphazard

set of conventions within a development team.

Thus concludes our journey through Git-based revision control. This tutorial was meant to prepare you for the realities of distributed software development—not to transform you into a Git expert overnight. You should be able to manage your own projects, collaborate with other Git users, and, perhaps most importantly, understand exactly what any other piece of Git documentation is trying to convey.

Your job now is to take these skills and apply them to new projects, sift through complex histories that you’ve never seen before, talk to other developers about their Git workflows, and take the time to actually try all of those “I wonder what would have happened if...” scenarios. Good luck!

For questions, comments, or suggestions, please [contact us](#).

Quick Reference

```
git cat-file <type> <object-id>
```

Display the specified object, where <type> is one of commit, tree, blob, or tag.

```
git cat-file -t <object-id>
```

Output the type of the specified object.

```
git ls-tree <tree-id>
```

Display a pretty version of the specified tree object.

```
git gc
```

Perform a garbage collection on the object database.

```
git update-index [--add] <file>
```

Stage the specified file, using the optional --add flag to denote a new untracked file.


```
git write-tree
```

Generate a tree from the index and store it in the object database. Returns the ID of the new tree object.

```
git commit-tree <tree-id> -p <parent-id>
```

Create a new commit object from the given tree object and parent commit. Returns the ID of the new commit object.