Git

Succinctly

by Ryan Hodson

Syncfusion | Technology Resource Portal

Git Succinctly

By
Ryan Hodson

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Edited by
This publication was edited by Praveen Ramesh, director of development, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the Web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The Succinctly series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly@syncfusion.com.

We sincerely hope you enjoy this book and that it helps you better understand the topic of study. Thank you for reading.

Introduction

Git is an open-source version control system known for its speed, stability, and distributed collaboration model. Originally created in 2006 to manage the entire Linux kernel, Git now boasts a comprehensive feature set, an active development team, and several free hosting communities.

Git was designed from the ground up, paying little attention to the existing standards of centralized versioning systems. So, if you're coming from an SVN or CVS background, try to forget everything you know about version control before reading this guide.

Distributed software development is fundamentally different from centralized version control systems. Instead of storing file information in a single central repository, Git gives *every* developer a full copy of the repository. To facilitate collaboration, Git lets each of these repositories share changes with any other repository.

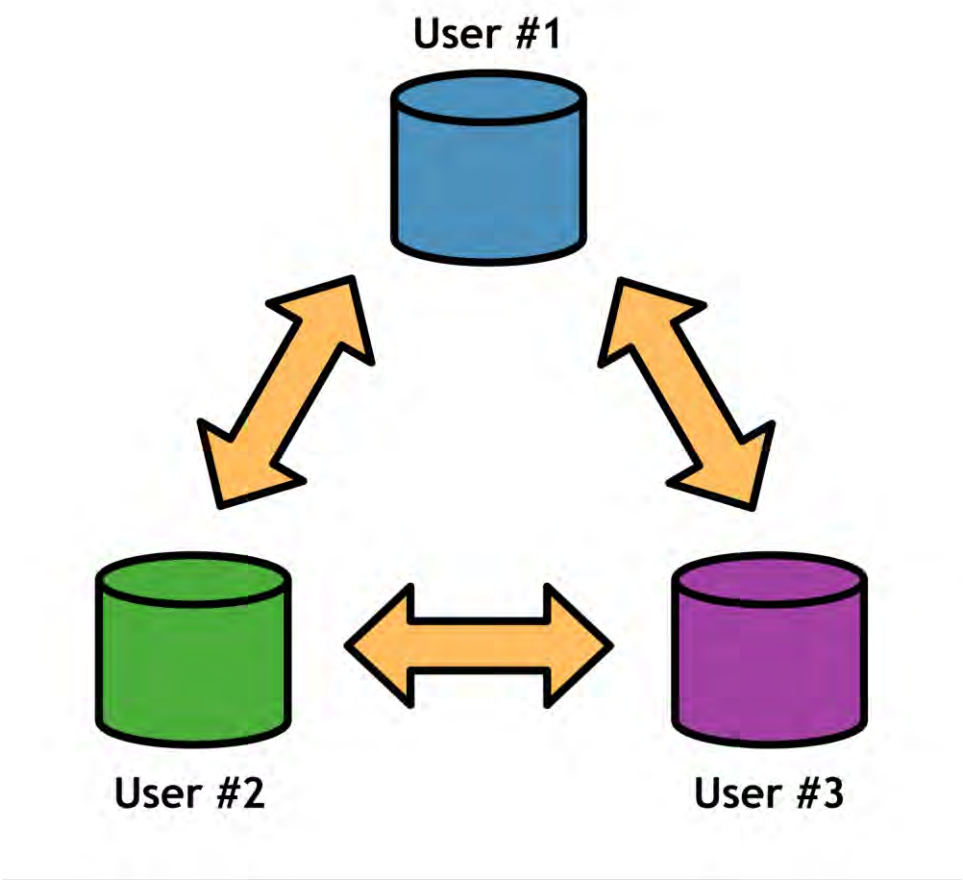

Figure 1: Distributed software development

Having a complete repository on your local machine has a far-reaching impact on the development cycle…

Faster Commands

First, a local copy of the repository means that almost all version control actions are much faster. Instead of communicating with the central server over a network connection, Git actions are performed on the local machine. This also means you can work offline without changing your workflow.

Stability

Since each collaborator essentially has a backup of the whole project, the risk of a server crash, a corrupted repository, or any other type of data loss is much lower than that of centralized systems that rely on a single point-of-access.

Isolated Environments

Every copy of a Git repository, whether local or remote, retains the full history of a project. Having a complete, isolated development environment gives each user the freedom to experiment with new additions before polishing them up into clean, publishable commits.

Efficient Merging

A complete history for each developer also means a *divergent* history for each developer. As soon as you make a single local commit, you're out of sync with everyone else on the project. To cope with this massive amount of branching, Git became very good at merging divergent lines of development.

Each Git repository contains 4 components:

- The working directory
- The staging area
- Committed history
- Development branches

Everything from recording commits to distributed collaboration revolves around these core objects.

The Working Directory

The working directory is where you actually edit files, compile code, and otherwise develop your project. For all intents and purposes, you can treat the working directory as a normal folder. Except, you now have access to all sorts of commands that can record, alter, and transfer the contents of that folder.

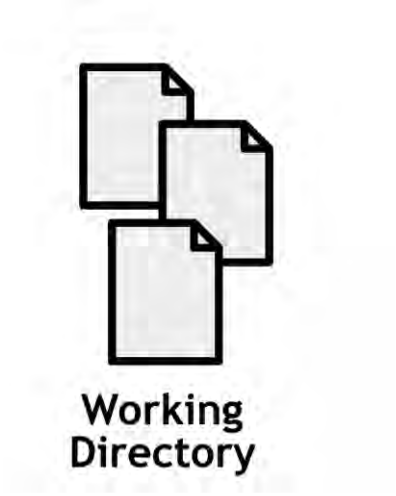

Figure 2: The working directory

The Staging Area

The staging area is an intermediary between the working directory and the project history. Instead of forcing you to commit all of your changes at once, Git lets you group them into related changesets. Staged changes are not yet part of the project history.

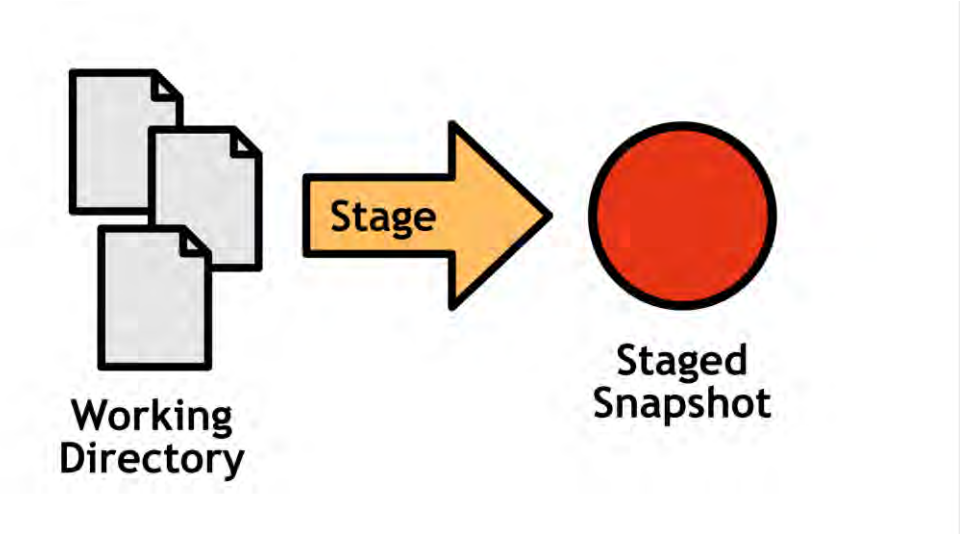

Figure 3: The working directory and the staging area

Committed History

Once you've configured your changes in the staging area, you can commit it to the project history where it will remain as a "safe" revision. Commits are "safe" in the sense that Git will never change them on its own, although it is possible for *you* to manually rewrite project history.

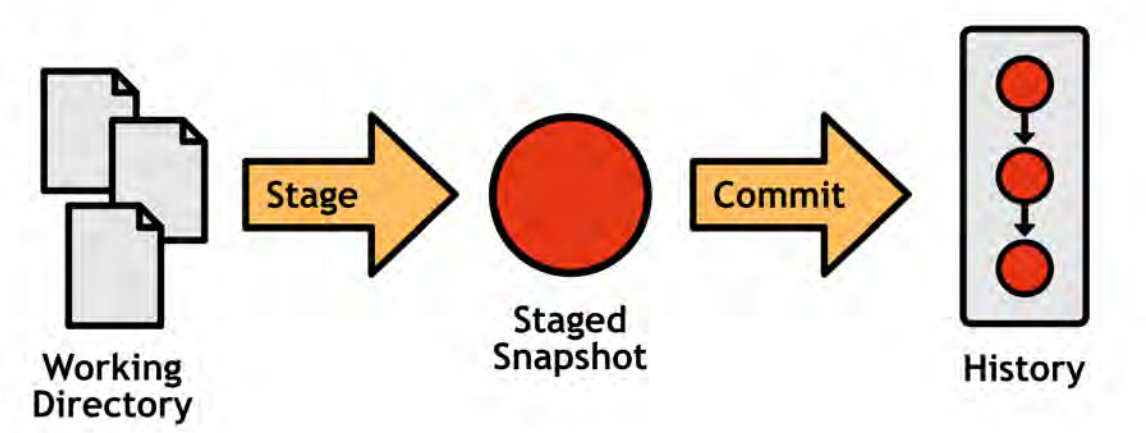

Figure 4: The working directory, staged snapshot, and committed history

Development Branches

So far, we're still only able to create a *linear* project history, adding one commit on top of another. Branches make it possible to develop multiple unrelated features in parallel by forking the project history.

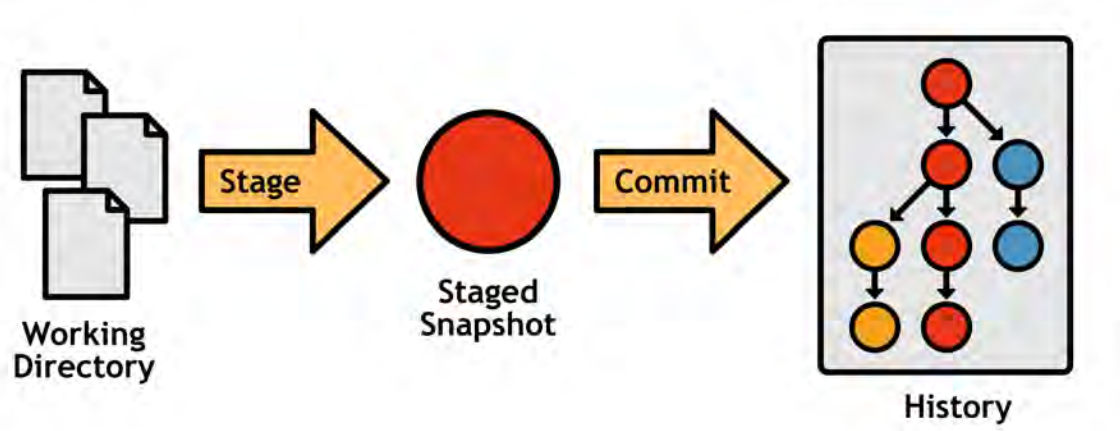

Figure 5: The complete Git workflow with a branched history

Git branches are not like the branches of centralized version control systems. They are cheap to make, simple to merge, and easy to share, so Git-based developers use branches for *everything*—from long-running features with several contributors to 5-minute fixes. Many developers *only* work in dedicated topic branches, leaving the main history branch for public releases.

Chapter 2 Getting Started

Installation

Git is available on all major platforms. The instructions below will walk you through installation on Windows, but it's always best to consult the official Git Web site for the most up-to-date information.

Git for Windows is available through the MsysGit package.

1. Download and execute the most recent version of the installer.
2. In the setup screen entitled "Adjusting your PATH environment," select the option "Use Git Bash only."
3. In the setup screen titled "Choosing the SSH executable," select "Use OpenSSH."
4. Finally, select "Checkout Windows-style, commit Unix-style line endings" and press "Next" to begin the installation.

This will install a new program called "Git Bash," which is the command prompt you should use whenever you're working with Git.

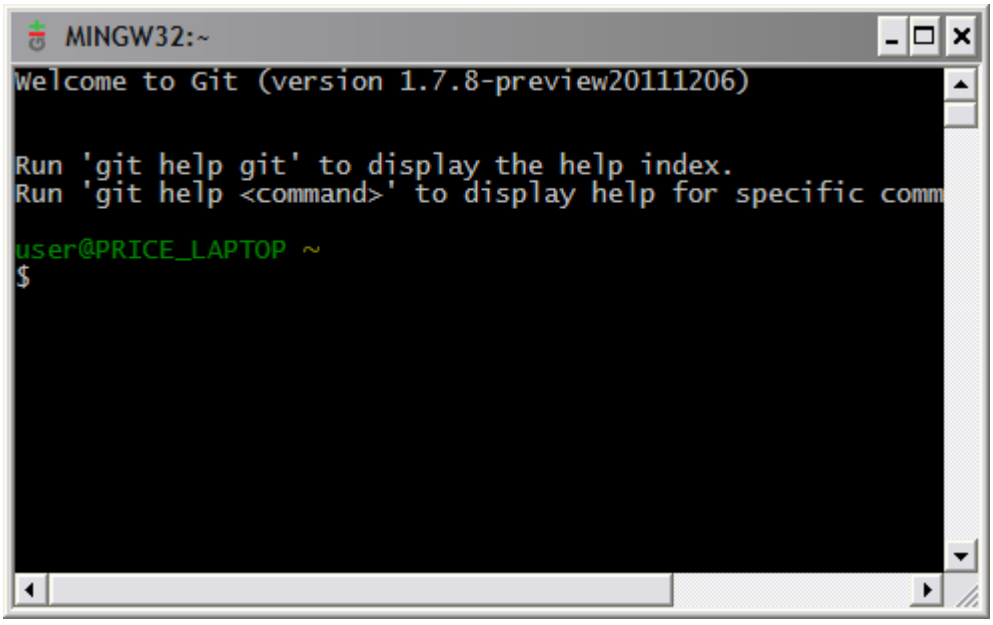

Figure 6: Screenshot of Git Bash

Configuration

Git comes with a long list of configuration options covering everything from your name to your favorite merge tool. You can set options with the `git config`

command, or by manually editing a file called `.gitconfig` in your home directory. Some of the most common options are presented below.

User Info

The first thing you'll want to do with any new Git installation is introduce yourself. Git records this information with your commits, and third-party services like GitHub use it to identify you.

```
git config --global user.name "John Smith"
git config --global user.email john@example.com
```

The `--global` flag records options in `~/.gitconfig`, making it the default for all new repositories. Omitting it lets you specify options on a per-repository basis.

Editor

Git's command-line implementation relies on a text editor for most of its input. You can force Git to use your editor-of-choice with the `core.editor` option:

```
git config --global core. editor gvim
```

Aliases

By default, Git doesn't come with any shortcuts, but you can add your own by aliasing commands. If you're coming from an SVN background, you'll appreciate the following bindings:

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

Learn more by running the `git help config` in your Git Bash prompt.

Initializing Repositories

Git is designed to be as unobtrusive as possible. The only difference between a Git repository and an ordinary project folder is an extra `.git` directory in the project root (not in *every* subfolder like SVN). To turn an ordinary project folder into a full-fledged Git repository, run the `git init` command:

```
git init <path>
```

The `<path>` argument should be a path to the repository (leaving it blank will use the current working directory). Now, you can use all of Git's wonderful version control features.

Cloning Repositories

As an alternative to `git init`, you can clone an existing Git repository using the following command:

```
git clone ssh://<user>@<host>/path/to/repo.git
```

This logs into the `<host>` machine using SSH and downloads the `repo.git` project. This is a *complete* copy, not just a link to the server's repository. You have your own history, working directory, staging area, and branch structure, and no one will see any changes you make until you push them back to a public repository.

Chapter 3 Recording Changes

Maintaining a series of “safe” revisions of a project is the core function of any version control system. Git accomplishes this by recording **snapshots** of a project. After recording a snapshot, you can go back and view old versions, restore them, and experiment without the fear of destroying existing functionality.

SVN and CVS users should note that this is fundamentally different from their system’s implementation. Both of these programs record diffs for each file—an incremental record of the changes in a project. In contrast, Git’s snapshots are just that—*snapshots*. Each commit contains the complete version of each file it contains. This makes Git incredibly fast since the state of a file doesn’t need to be generated each time it’s requested:

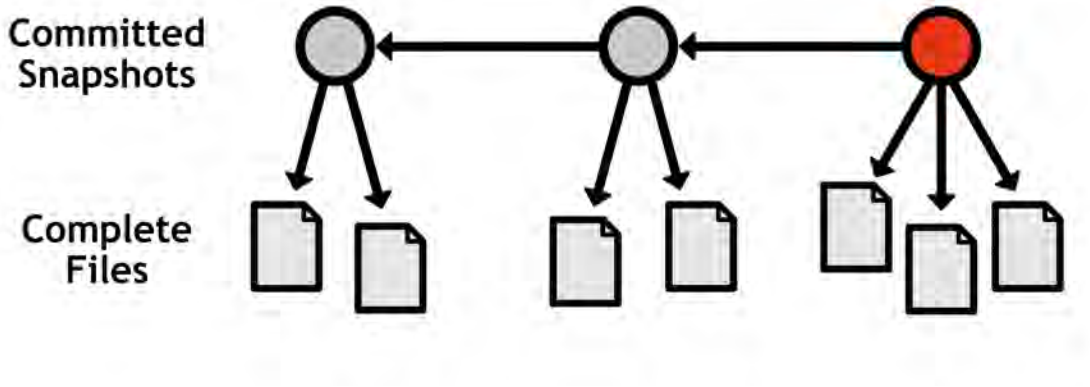

Figure 7: Recording complete snapshots, not differences between revisions

This chapter introduces the basic workflow for creating snapshots using the working directory, staging area, and committed history. These are the core components of Git-based revision control.

The Staging Area

Git’s staging area gives you a place to organize a commit before adding it to the project history. **Staging** is the process of moving changes from the working directory to the staged snapshot.

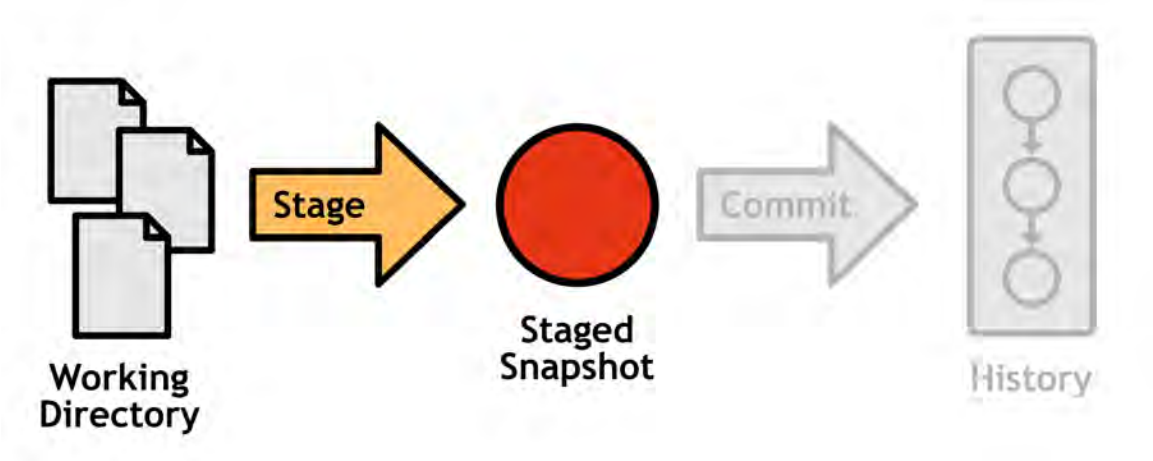

Figure 8: Components involved in staging a commit

It gives you the opportunity to pick-and-choose *related* changes from the working directory, instead of committing everything all at once. This means you can create *logical* snapshots over *chronological* ones. This is a boon to developers because it lets them separate coding activities from version control activities. When you're writing features, you can forget about stopping to commit them in isolated chunks. Then, when you're done with your coding session, you can separate changes into as many commits as you like via the stage.

To add new or modified files from the working directory to the staging area, use the following command:

```
git add <file>
```

To delete a file from a project, you need to add it to the staging area just like a new or modified file. The next command will stage the deletion and stop tracking the file, but it won't delete the file from the working directory:

```
git rm --cached <file>
```

Inspecting the Stage

Viewing the status of your repository is one of the most common actions in Git. The following command outputs the state of the working directory and staging area:

```
git status
```

This will result in a message that resembles the following (certain sections may be omitted depending on the state of your repository):

```
# On branch master
# Changes to be committed:
#
#       new file:   foobar.txt
#
# Changes not staged for commit:
#
#       modified:   foo.txt
#
# Untracked files:
#
#       bar.txt
```

The first section, “Changes to be committed” is your staged snapshot. If you were to run `git commit` right now, only these files would be added to the project history. The next section lists *tracked* files that will not be included in the next commit. Finally, “Untracked files” contains files in your working directory that haven’t been added to the repository.

Generating Diffs

If you need more detailed information about the changes in your working directory or staging area, you can generate a diff with the following command:

```
git diff
```

This outputs a diff of every *unstaged* change in your working directory. You can also generate a diff of all *staged* changes with the `--cached` flag:

```
git diff –cached
```

Note that the project history is outside the scope of `git status`. For displaying committed snapshots, you’ll need `git log`.

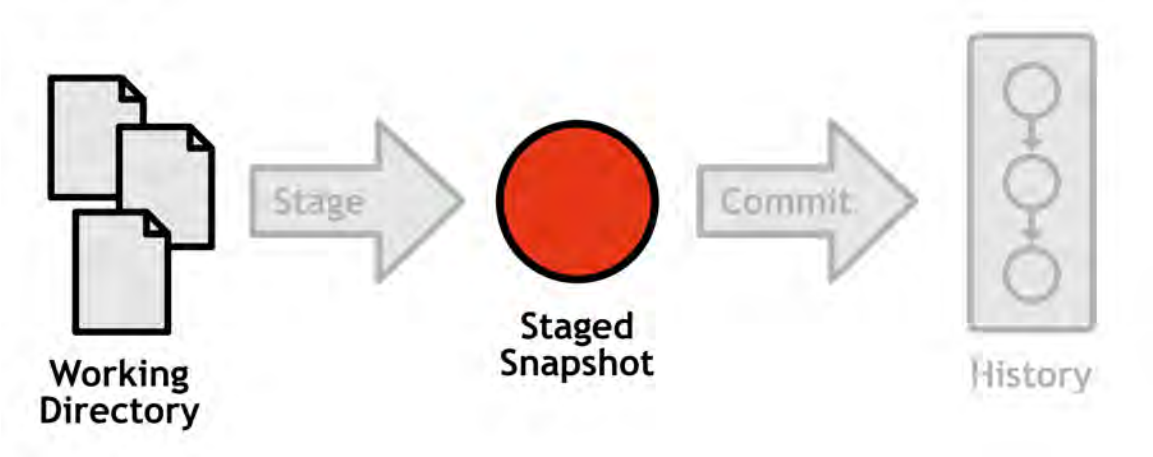

Figure 9: Components in the scope of ***`git status`***

Commits

Commits represent every saved version of a project, which makes them the atomic unit of Git-based version control. Each commit contains a snapshot of the project, your user information, the date, a commit message, and an **SHA-1 checksum** of its entire contents:

```
commit b650e3bd831aba05fa62d6f6d064e7ca02b5ee1b
Author: john <john@example.com>
Date:   Wed Jan 11 00:45:10 2012 -0600

    Some commit message
```

This checksum serves as a commit's unique ID, and it also means that a commit will *never* be corrupted or unintentionally altered without Git knowing about it.

Since the staging area already contains the desired changeset, committing doesn't require any involvement from the working directory.

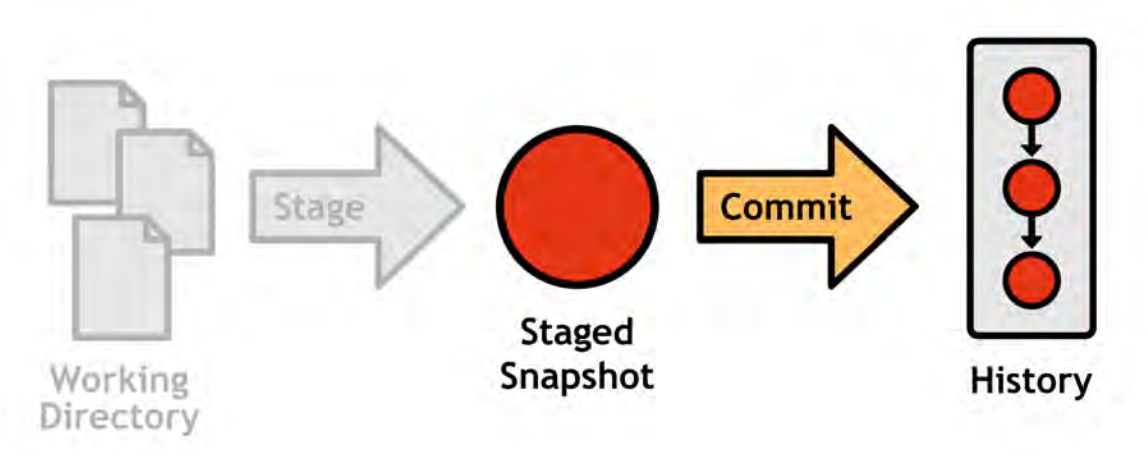

Figure 10: Components involved in committing a snapshot

To commit the staged snapshot and add it to the history of the current branch, execute the following:

```
git commit
```

You'll be presented with a text editor and prompted for a "commit message." Commit messages should take the following form:

```
<commit summary in 50 characters or less.>
<blank line>
<detailed description of changes in this commit.>
```

Git uses the first line for formatting log output, e-mailing patches, etc., so it should be brief, while still describing the entire changeset. If you can't come up with the summary line, it probably means your commit contains too many unrelated changes. You should go back and split them up into distinct commits. The summary should be followed by a blank line and a detailed description of the changes (e.g., why you made the changes, what ticket number it corresponds to).

Inspecting Commits

Like a repository's status, viewing its history is one of the most common tasks in Git version control. You can display the current branch's commits with:

```
git log
```

We now have the only two tools we need to inspect every component of a Git repository.

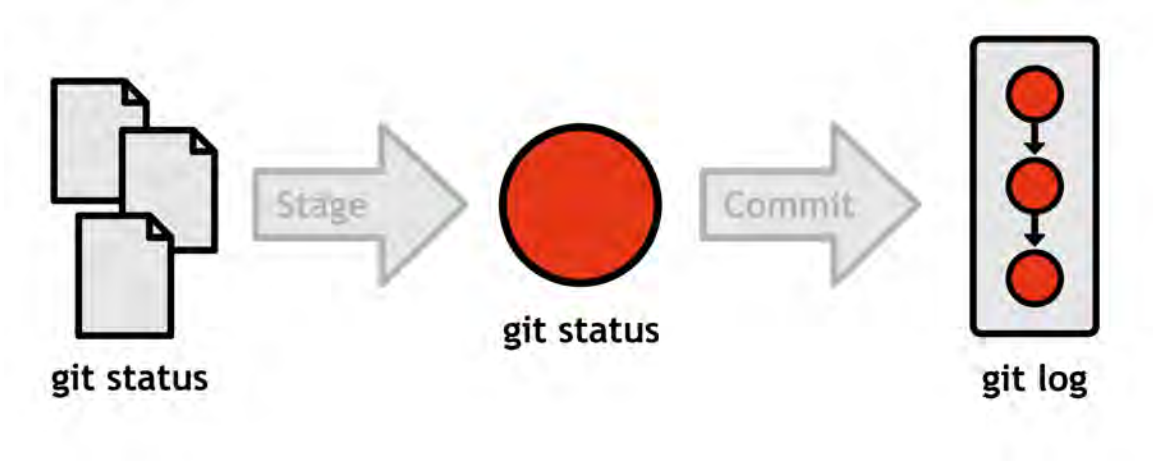

Figure 11: Output of `git status` vs. `git log`

This also gives us a natural grouping of commands:

- Stage/Working Directory: `git add`, `git rm`, `git status`
- Committed History: `git commit`, `git log`

Useful Configurations

Git provides a plethora of formatting options for `git log`, a few of which are included here. To display each commit on a single line, use:

```
git log -oneline
```

Or, to target the history of an individual file instead of the whole repository, use:

```
git log --oneline <file>
```

Filtering the log output is also very useful once your history grows beyond one screenful of commits. You can use the following to display commits contained in `<until>` but not in `<since>`. Both arguments can be a commit ID, a branch name, or a tag:

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

Finally, you can display a diffstat of the changes in each commit. This is useful to see what files were affected by a particular commit.

```
git log -stat
```

For visualizing history, you might also want to look at the `gitk` command, which is actually a separate program dedicated to graphing branches. Run `git help gitk` for details.

Tagging Commits

Tags are simple pointers to commits, and they are incredibly useful for bookmarking important revisions like public releases. The `git tag` command can be used to create a new tag:

```
git tag -a v1.0 -m "Stable release"
```

The `-a` option tells Git to create an *annotated* tag, which lets you record a message along with it (specified with `-m`).

Running the same command without arguments will list your existing tags:

```
git tag
```

Chapter 4 Undoing Changes

The whole point of maintaining “safe” copies of a software project is peace of mind: should your project suddenly break, you’ll know that you have easy access to a functional version, and you’ll be able to pinpoint precisely where the problem was introduced. To this end, recording commits is useless without the ability to undo changes. However, since Git has so many components, “undoing” can take on many different meanings. For example, you can:

- Undo changes in the working directory
- Undo changes in the staging area
- Undo an entire commit

To complicate things even further, there are multiple ways to undo a commit. You can either:

1. Simply delete the commit from the project history.
2. Leave the commit as is, using a new commit to undo the changes introduced by the first commit.

Git has a dedicated tool for each of these situations. Let’s start with the working directory.

Undoing in the Working Directory

The period of time immediately after saving a safe copy of a project is one of great innovation. Empowered by the knowledge that you’re free to do *anything* you want without damaging the code base, you can experiment to your heart’s content. However, this carefree experimentation often takes a wrong turn and leads to a working directory with a heap of off-topic code. When you reach this point, you’ll probably want to run the following commands:

```
git reset --hard HEAD
git clean -f
```

This configuration of `git reset` makes the working directory and the stage match the files in the most recent commit (also called `HEAD`), effectively obliterating all uncommitted changes in *tracked* files. To get rid of *untracked* files, you have to use the `git clean` command. Git is very careful about removing code, so you must also supply the `-f` option to force the deletion of these files.

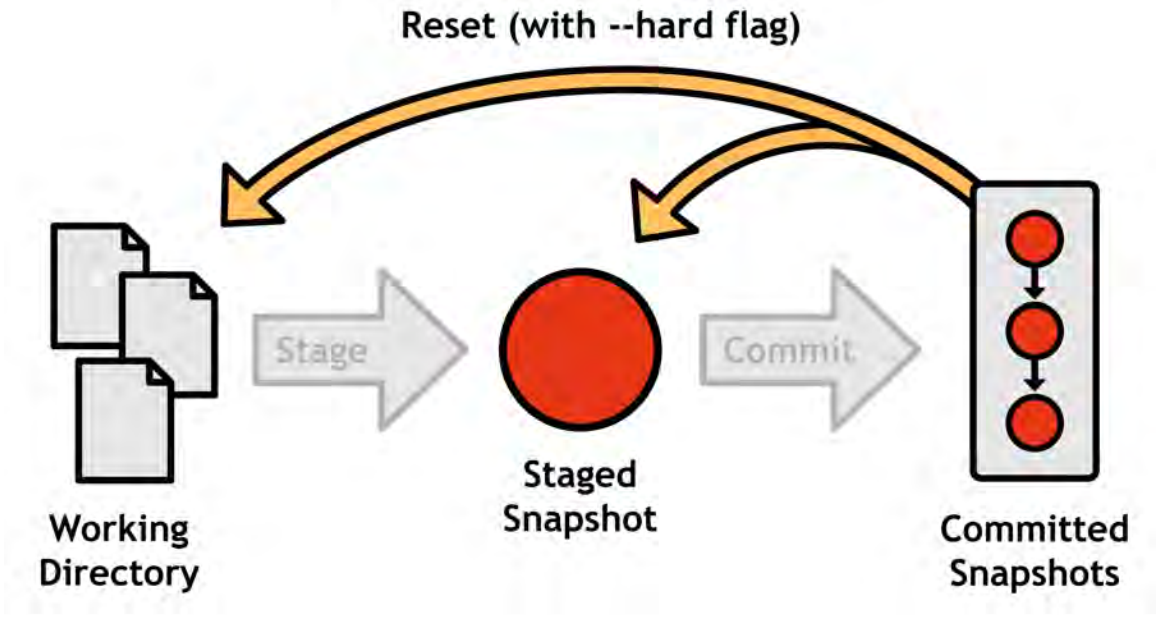

Figure 12: Resetting all uncommitted changes

Individual Files

It's also possible to target individual files. The following command will make a single file in the working directory match the version in the most recent commit.

```
git checkout HEAD <file>
```

This command doesn't change the project history at all, so you can safely replace `HEAD` with a commit ID, branch, or tag to make the file match the version in that commit. But, do *not* try this with `git reset`, as it *will* change your history (explained in *Undoing Commits*).

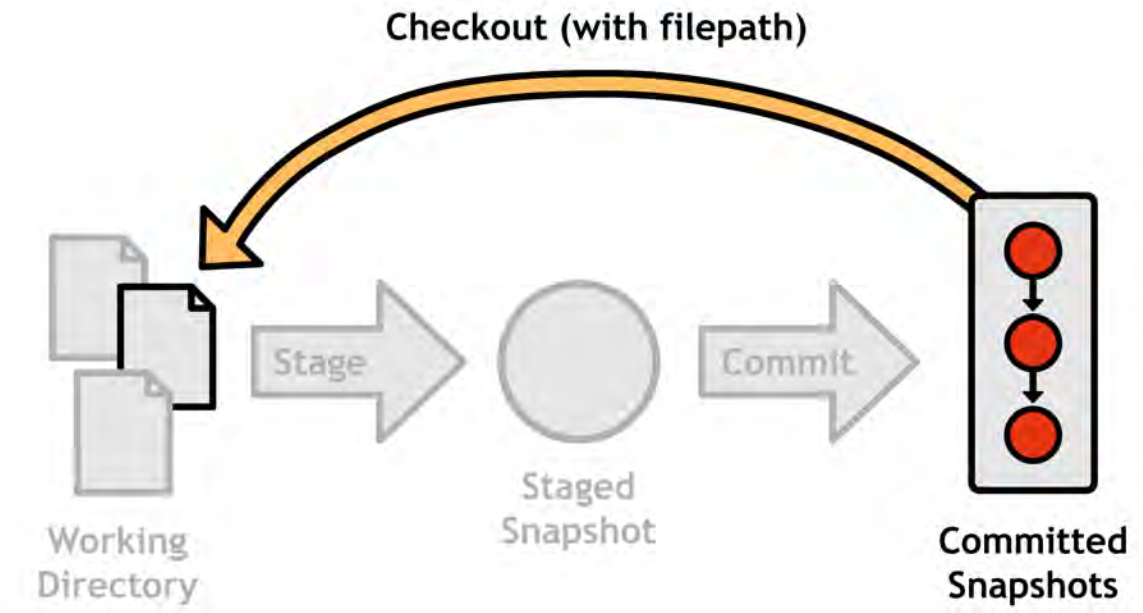

Figure 13: Reverting a file with ***`git checkout`***

Undoing in the Staging Area

In the process of configuring your next commit, you'll occasionally add an extra file to the stage. The following invocation of **`git reset`** will unstage it:

```
git reset HEAD <file>
```

Omitting the **`--hard`** flag tells Git to leave the working directory alone (opposed to **`git reset --hard HEAD`**, which resets every file in both the working directory and the stage). The staged version of the file matches **`HEAD`**, and the working directory retains the modified version. As you might expect, this results in an unstaged modification in your **`git status`** output.

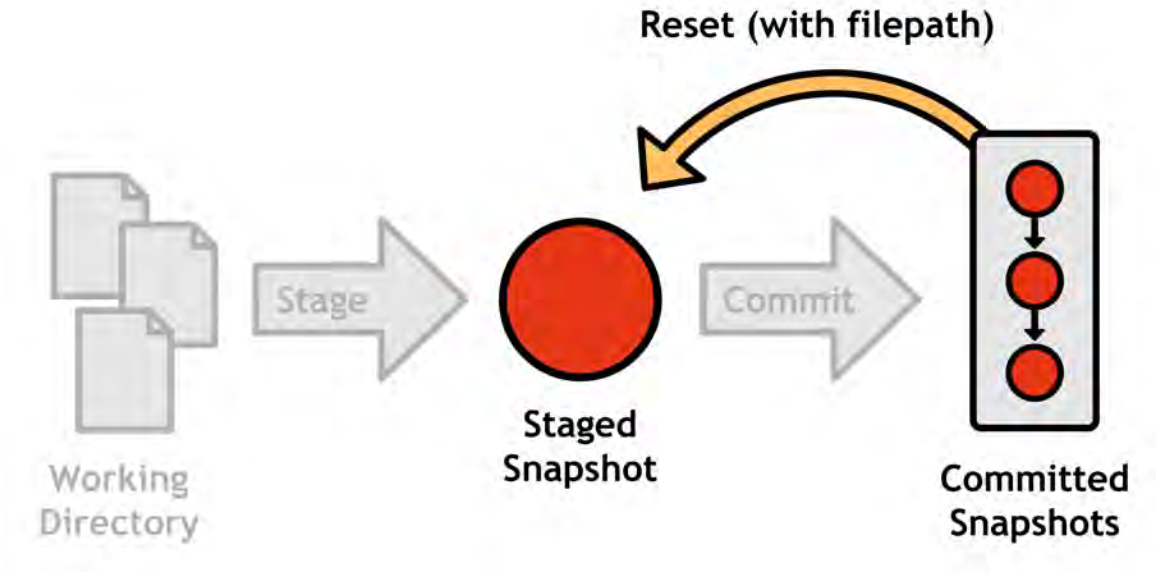

Figure 14: Unstaging a file with ***`git reset`***

Undoing Commits

There are two ways to undo a commit using Git: You can either **reset** it by simply removing it from the project history, or you can **revert** it by generating a *new* commit that gets rid of the changes introduced in the original. Undoing by introducing another commit may seem excessive, but rewriting history by completely removing commits can have dire consequences in multi-user workflows (read more in *Remote Repositories*).

Resetting

The ever-versatile `git reset` can also be used to *move* the `HEAD` reference.

```
git reset HEAD~1
```

The `HEAD~1` syntax parameter specifies the commit that occurs immediately before `HEAD` (likewise, `HEAD~2` refers to the second commit before `HEAD`). By moving the `HEAD` reference backward, you're effectively removing the most recent commit from the project's history.

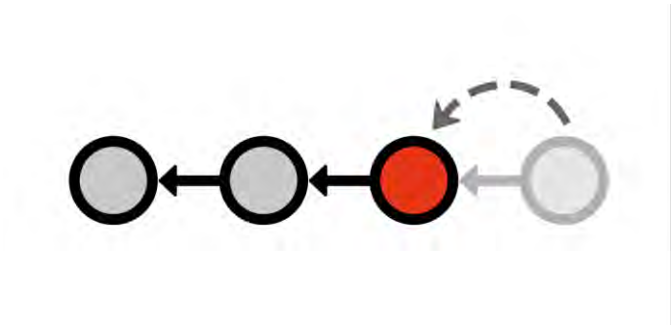

Figure 15: Moving `HEAD` to `HEAD~1` with `git reset`

This is an easy way to remove a couple of commits that veered off-topic, but it presents a serious collaboration problem. If another developer had started building on top of the commit we removed, how would he or she synchronize with our repository? The developer would have to ask us for the ID of the replacement commit, manually track it down in your repository, move all of the changes to that commit, resolve merge conflicts, and then share the "new" changes with everybody *again*. Just imagine what would happen in an open-source project with hundreds of contributors…

The point is, ***don't reset public commits***, but feel free to delete private ones that you haven't shared with anyone. We'll revisit this concept in *Remote Repositories*.

Reverting

To remedy the problems introduced by resetting public commits, Git developers devised another way to undo commits: the revert. Instead of altering existing commits, reverting adds a *new* commit that undoes the problem commit:

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

This takes the changes in the specified commit, figures out how to undo them, and creates a new commit with the resulting changeset. To Git and to other users, the revert commit looks and acts like any other commit—it just happens to undo the changes introduced by an earlier commit.

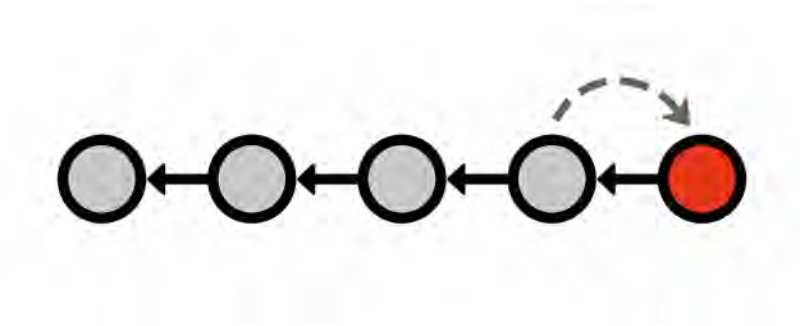

Figure 16: Undoing a commit with a revert commit

This is the ideal way of undoing changes that have already been committed to a public repository.

Amending

In addition to completely undoing commits, you can also **amend** the most recent commit by staging changes as usual, then running:

```
git commit –amend
```

This *replaces* the previous commit instead of creating a new one, which is very useful if you forgot to add a file or two. For your convenience, the commit editor is seeded with the old commit's message. Again, you must ***be careful*** when using the `--amend` flag, since it rewrites history much like `git reset`.

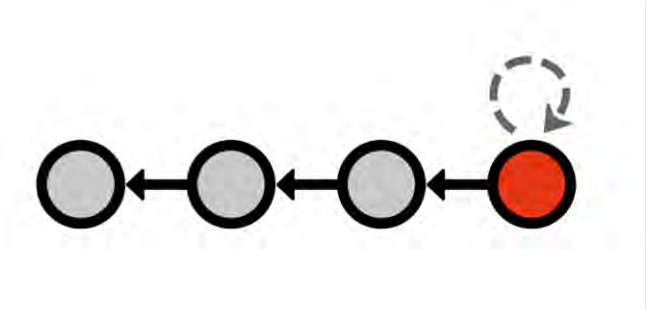

Figure 17: Amending the most recent commit

Chapter 5 Branches

Branches multiply the basic functionality offered by commits by allowing users to fork their history. Creating a new branch is akin to requesting a new development environment, complete with an isolated working directory, staging area, and project history.

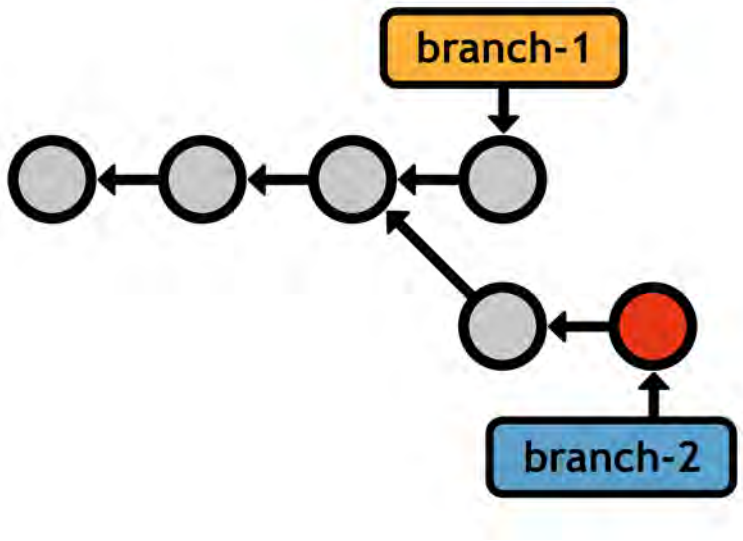

Figure 18: Basic branched development

This gives you the same peace of mind as committing a “safe” copy of your project, but you now have the additional capacity to work on multiple versions at the same time. Branches enable a **non-linear workflow**—the ability to develop unrelated features in parallel. As we’ll discover in *Remote Repositories*, a non-linear workflow is an important precursor to the distributed nature of Git’s collaboration model.

Unlike SVN or CVS, Git’s branch implementation is incredibly efficient. SVN enables branches by copying the entire project into a new folder, much like you would do without any revision control software. This makes merges clumsy, error-prone, and slow. In contrast, Git branches are simply a pointer to a commit. Since they work on the commit level instead of directly on the file level, Git branches make it much easier to merge diverging histories. This has a dramatic impact on branching workflows.

Manipulating Branches

Git separates branch functionality into a few different commands. The `git branch` command is used for listing, creating, or deleting branches.

Listing Branches

First and foremost, you'll need to be able to view your existing branches:

```
git branch
```

This will output all of your current branches, along with an asterisk next to the one that's currently "checked out" (more on that later):

```
* master
some-feature
quick-bug-fix
```

The `master` branch is Git's default branch, which is created with the first commit in any repository. Many developers use this branch as the "main" history of the project—a permanent branch that contains every major change it goes through.

Creating Branches

You can create a new branch by passing the branch name to the same `git branch` command:

```
git branch <name>
```

This creates a pointer to the current `HEAD`, but does *not* switch to the new branch (you'll need `git checkout` for that). Immediately after requesting a new branch, your repository will look something like the following.

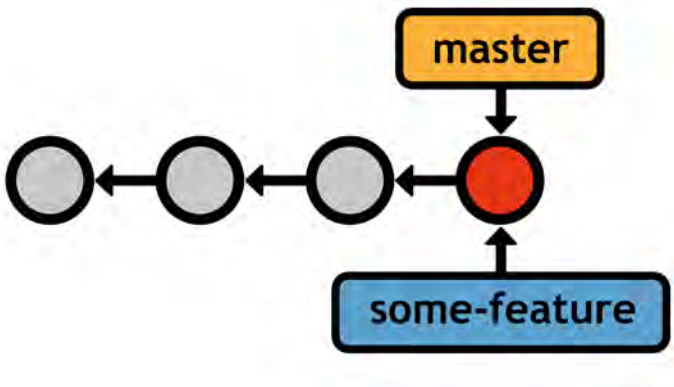

Figure 19: Creating a new branch

Your current branch (`master`) and the new branch (`some-feature`) both reference the same commit, but any new commits you record will be exclusive to the current branch. Again, this lets you work on unrelated features in parallel, while still maintaining sensible histories. For example, if your current branch was `some-feature`, your history would look like the following after committing a snapshot.

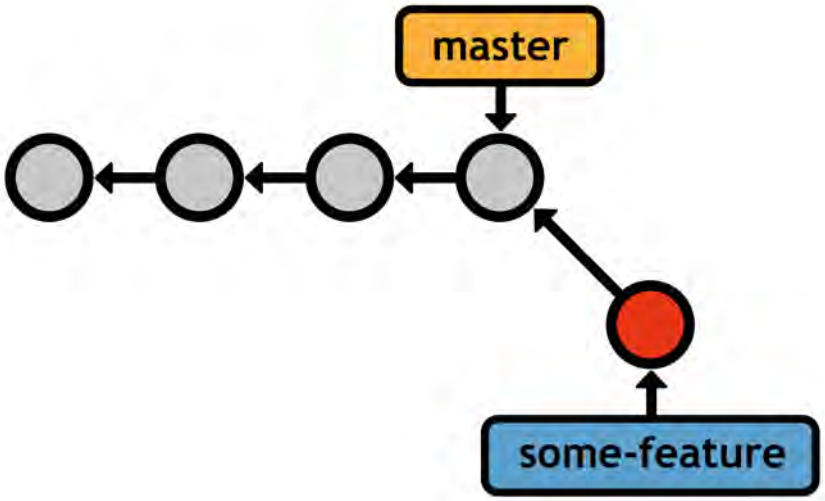

Figure 20: Committing on the `some-feature` branch

The new `HEAD` (denoted by the highlighted commit) exists only in the `some-feature` branch. It won't show up in the log output of `master`, nor will its changes appear in the working directory after you check out `master`.

You can actually see the new branch in the internal database by opening the file `.git/refs/heads/<name>`. The file contains the ID of the referenced commit, and it is the sole definition of a Git branch. This is the reason branches are so lightweight and easy to manage.

Deleting Branches

Finally, you can delete branches via the `-d` flag:

```
git branch -d <name>
```

But, Git's dedication to never losing your work prevents it from removing branches with unmerged commits. To force the deletion, use the `-D` flag instead:

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

Unmerged changes will be lost, so ***be very careful*** with this command.

Checking Out Branches

Of course, creating branches is useless without the ability to switch between them. Git calls this "checking out" a branch:

```
git checkout <branch>
```

After checking out the specified branch, your working directory is updated to match the specified branch's commit. In addition, the `HEAD` is updated to point to the new branch, and all new commits will be stored on the new branch. You can think of checking out a branch as switching to a new project folder—except it will be much easier to pull changes back into the project.

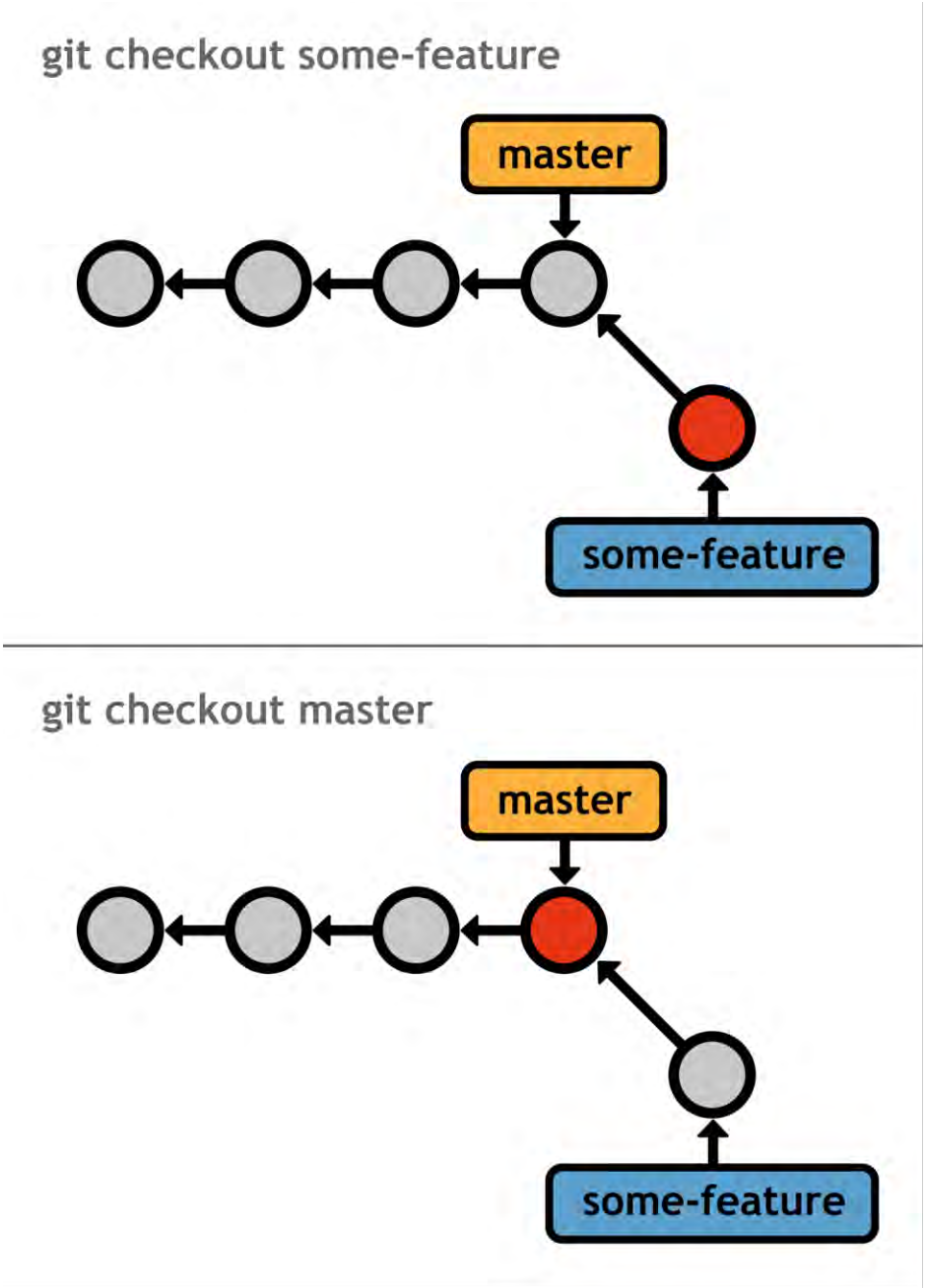

Figure 21: Checking out different branches

With this in mind, it's usually a good idea to have a **clean** working directory before checking out a branch. A clean directory exists when there are no

uncommitted changes. If this isn't the case, `git checkout` has the potential to overwrite your modifications.

As with committing a "safe" revision, you're free to experiment on a new branch without fear of destroying existing functionality. But, you now have a dedicated history to work with, so you can record the progress of an experiment using the exact same `git add` and `git commit` commands from earlier in the book.

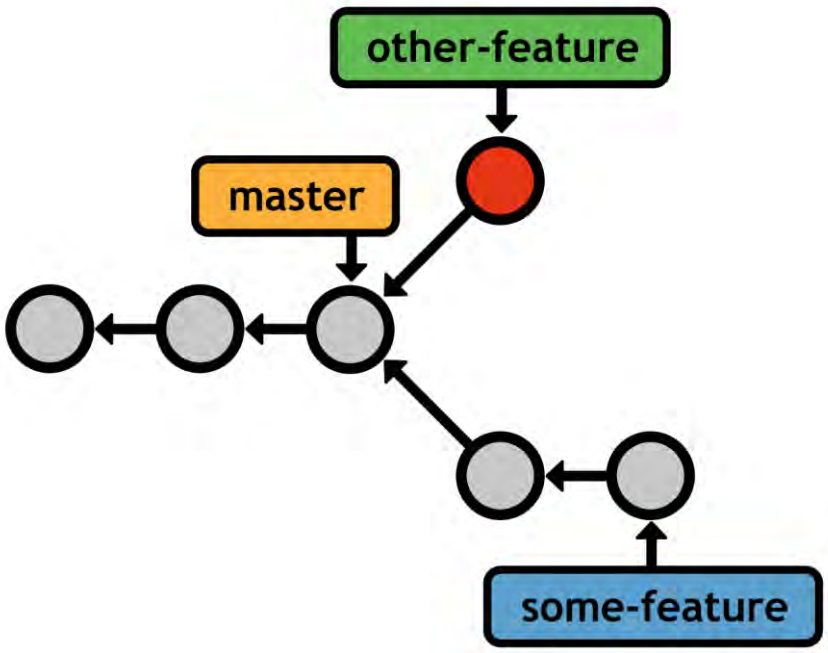

Figure 22: Developing multiple features in parallel

This functionality will become even more powerful once we learn how to merge divergent histories back into the "main" branch (e.g., `master`). We'll get to that in a moment, but first, there is an important use case of `git checkout` that must be considered…

Detached `HEAD`s

Git also lets you use `git checkout` with tags and commit IDs, but doing so puts you in a **detached HEAD state**. This means that you're not on a branch anymore—you're directly viewing a commit.

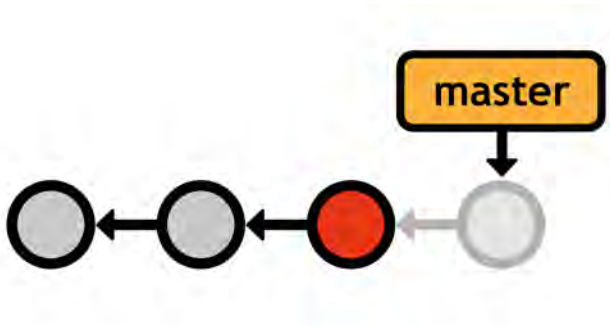

Figure 23: Checking out an old commit

You can look around and add new commits as usual, but since there is no branch pointing to the additions, you'll lose all your work as soon as you switch back to a real branch. Fortunately, creating a new branch in a detached `HEAD` state is easy enough:

```
git checkout -b <new-branch-name>
```

This is a shortcut for `git branch <new-branch-name>` followed by `git checkout <new-branch-name>`. After which, you'll have a shiny new branch reference to the formerly detached `HEAD`. This is a very handy procedure for forking experiments off of old revisions.

Merging Branches

Merging is the process of pulling commits from one branch into another. There are many ways to combine branches, but the goal is always to share information between branches. This makes merging one of the most important features of Git. The two most common merge methodologies are:

- The "fast-forward" merge
- The "3-way" merge

They both use the same command, `git merge`, but the method is automatically determined based on the structure of your history. In each case, *the branch you want to merge into must be checked out*, and the target branch will remain unchanged. The next two sections present two possible merge scenarios for the following commands:

```
git checkout master
git merge some-feature
```

Again, this merges the `some-feature` branch into the `master` branch, leaving the former untouched. You'd typically run these commands once you've completed a feature and want to integrate it into the stable project.

Fast-forward Merges

The first scenario looks like this:

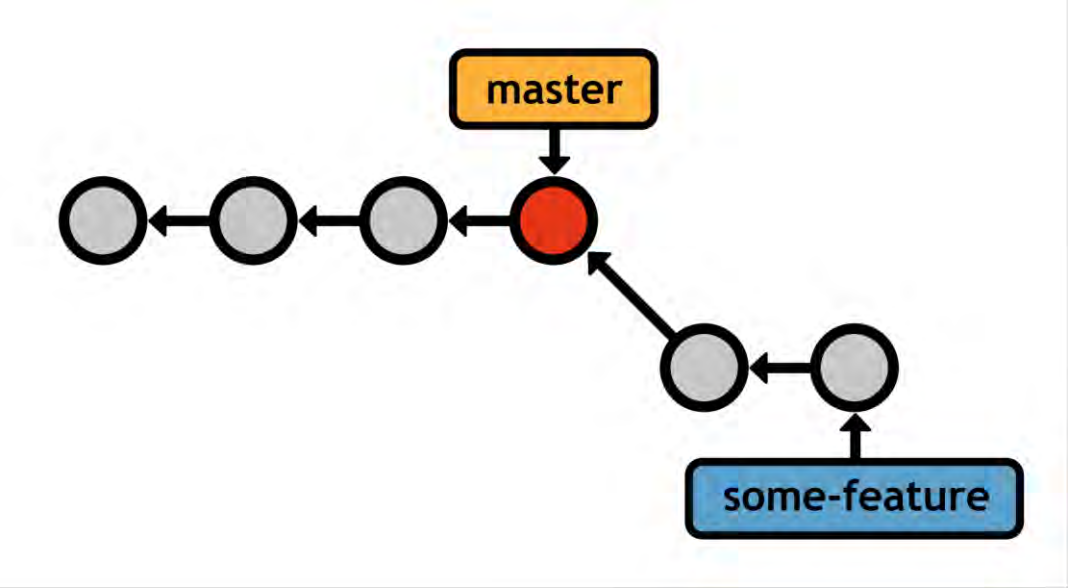

Figure 24: Before the fast-forward merge

We created a branch to develop some new feature, added two commits, and now it's ready to be integrated into the main code base. Instead of rewriting the two commits missing from `master`, Git can "fast-forward" the `master` branch's pointer to match the location of `some-feature`.

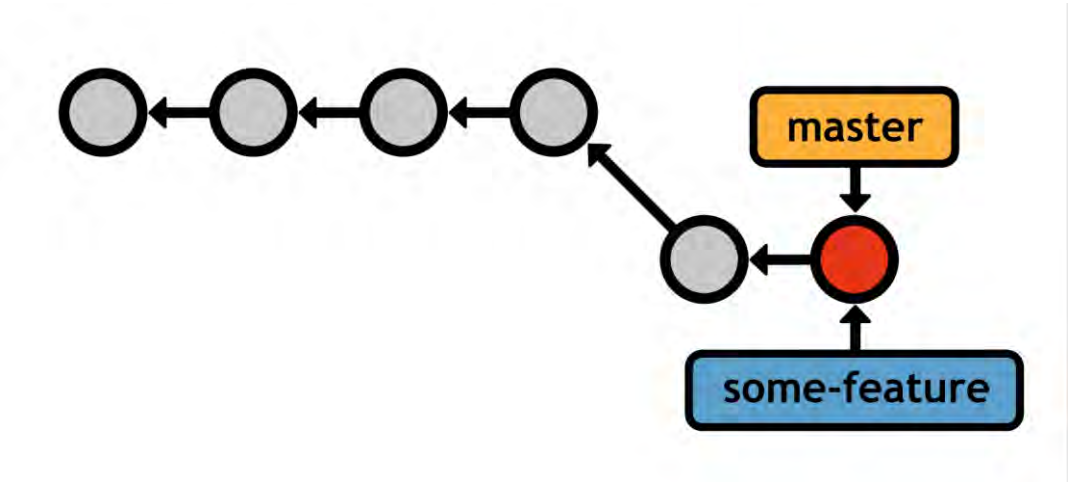

Figure 25: After the fast-forward merge

After the merge, the `master` branch contains all of the desired history, and the feature branch can be deleted (unless you want to keep developing on it). This is the simplest type of merge.

Of course, we could have made the two commits directly on the `master` branch; however, using a dedicated feature branch gave us a safe environment to experiment with new code. If it didn't turn out quite right, we could have simply deleted the branch (opposed to resetting/reverting). Or, if we added a bunch of intermediate commits containing broken code, we could clean it up before merging it into `master` (see *Rebasing*). As projects get more complicated and acquire more collaborators, this kind of branched development makes Git a fantastic organizational tool.

3-way Merges

But, not all situations are simple enough for a fast-forward commit. Remember, the main advantage of branches is the ability to explore many independent lines of development simultaneously. As a result, you'll often encounter a scenario that looks like the following:

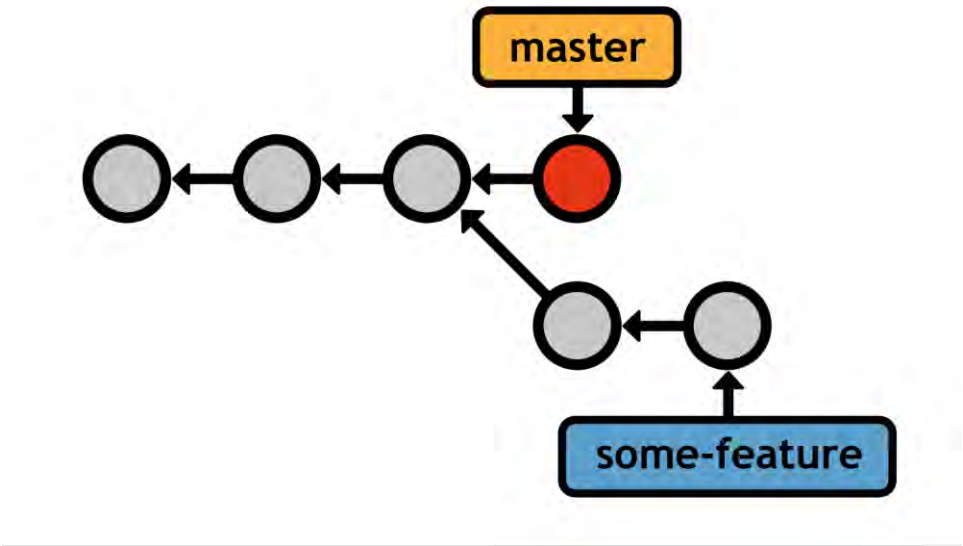

Figure 26: Before the 3-way merge

This started out like a fast-forward merge, but we added a commit to the `master` branch while we were still developing `some-feature`. For example, we could have stopped working on the feature to fix a time-sensitive bug. Of course, the bug-fix should be added to the main repository as soon as possible, so we wind up in the scenario shown above.

Merging the feature branch into `master` in this context results in a "3-way" merge. This is accomplished using the exact same commands as the fast-forward merge from the previous section.

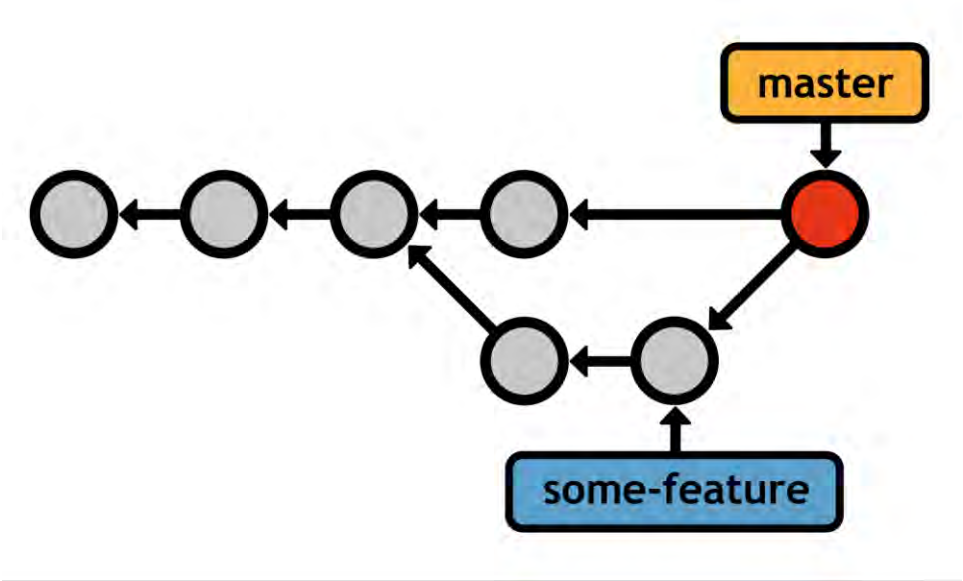

Figure 27: After the 3-way merge

Git can't fast-forward the `master` pointer to `some-feature` without backtracking. Instead, it generates a new **merge commit** that represents the combined snapshot of both branches. Note that this new commit has *two* parent commits, giving it access to both histories (indeed, running `git log` after the 3-way merge shows commits from both branches).

The name of this merge algorithm originates from the internal method used to create the merge commit. Git looks at *three* commits to generate the final state of the merge.

Merge Conflicts

If you try to combine two branches that make different changes to the same portion of code, Git won't know which version to use. This is called a **merge conflict**. Obviously, this can never happen during a fast-forward merge. When Git encounters a merge conflict, you'll see the following message:

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

Instead of automatically adding the merge commit, Git stops and asks you what to do. Running `git status` in this situation will return something like the following:

```
# On branch master
# Unmerged paths:
#
#       both modified:      <file>
```

Every file with a conflict is stored under the “Unmerged paths” section. Git annotates these files to show you the content from both versions:

```
<<<<<<< HEAD
  This content is from the current branch.
=======
    This is a conflicting change from another branch.
>>>>>>> some-feature
```

The part before the `=======` is from the `master` branch, and the rest is from the branch you’re trying to integrate.

To resolve the conflict, get rid of the `<<<<<<<`, `=======`, and `>>>>>>>` notation, and change the code to whatever you want to keep. Then, tell Git you’re done resolving the conflict with the `git add` command:

```
git add <file>
```

That’s right; all you have to do is stage the conflicted file to mark it as resolved. Finally, complete the 3-way merge by generating the merge commit:

```
git commit
```

The log message is seeded with a merge notice, along with a “conflicts” list, which can be particularly useful when trying to figure out where something went wrong in a project.

And that’s all there is to merging in Git. Now that we have an understanding of the mechanics behind Git branches, we can take an in-depth look at how veteran Git users leverage branches in their everyday workflow.

Branching Workflows

The workflows presented in this section are the hallmark of Git-based revision control. The lightweight, easy-to-merge nature of Git’s branch implementation makes them one of the most productive tools in your software development arsenal.

All branching workflows revolve around the `git branch`, `git checkout`, and `git merge` commands presented earlier this chapter.

Types of Branches

It’s often useful to assign special meaning to different branches for the sake of organizing a project. This section introduces the most common types of branches, but keep in mind these distinctions are purely superficial—to Git, a branch is a branch.

All branches can be categorized as either **permanent branches** or **topic branches**. The former contain the main history of a project (e.g., `master`), while the latter are temporary branches used to implement a specific *topic*, then discarded (e.g., `some-feature`).

Permanent Branches

Permanent branches are the lifeblood of any repository. They contain every major waypoint of a software project. Most developers use `master` exclusively for stable code. In these workflows, you *never* commit directly on `master`—it is only an integration branch for completed features that were built in dedicated topic branches.

In addition, many users add a second layer of abstraction in another integration branch (conventionally called `develop`, though any name will suffice). This frees up the `master` branch for *really* stable code (e.g., public commits), and uses `develop` as an internal integration branch to prepare for a public release. For example, the following diagram shows several features being integrated in `develop`, then a single, final merge into `master`, which symbolizes a public release.

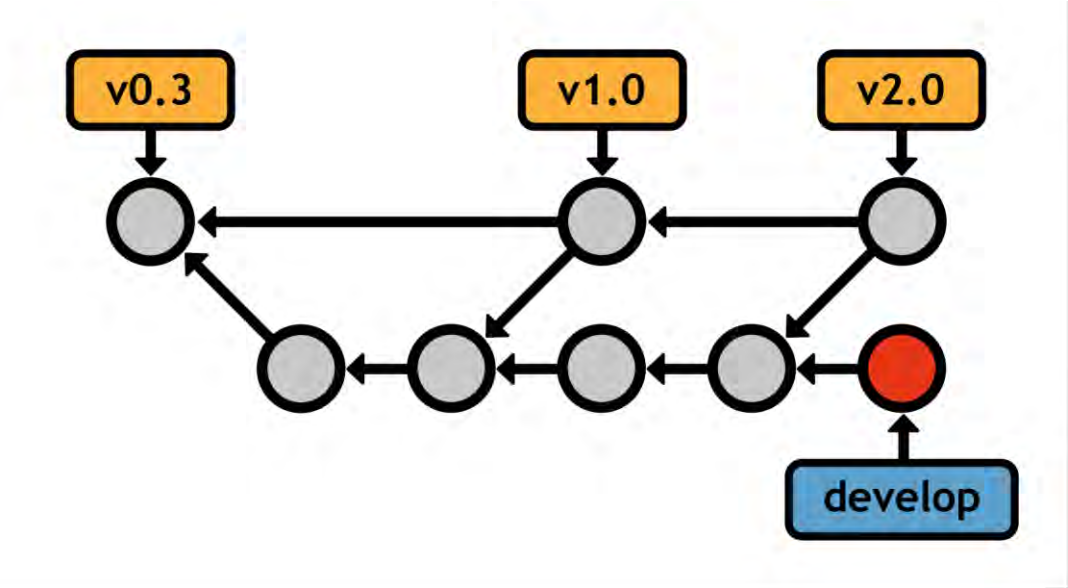

Figure 28: Using the `master` branch exclusively for public releases

Topic Branches

Topic branches generally fall into two categories: **feature branches** and **hotfix branches**. Feature branches are temporary branches that encapsulate a new feature or refactor, protecting the main project from untested code. They typically stem from another feature branch or an integration branch, but not the “super stable” branch.

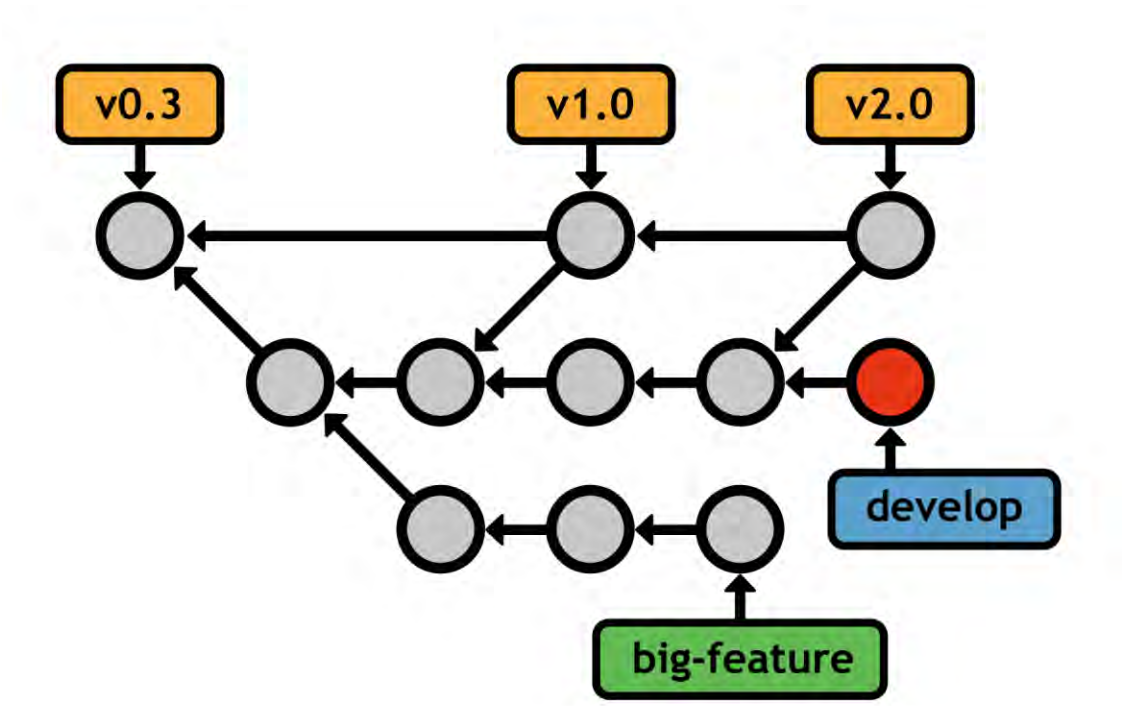

Figure 29: Developing a feature in an isolated branch

Hotfix branches are similar in nature, but they stem from the public release branch (e.g., `master`). Instead of developing new features, they are for quickly patching the main line of development. Typically, this means bug fixes and other important updates that can't wait until the next major release.

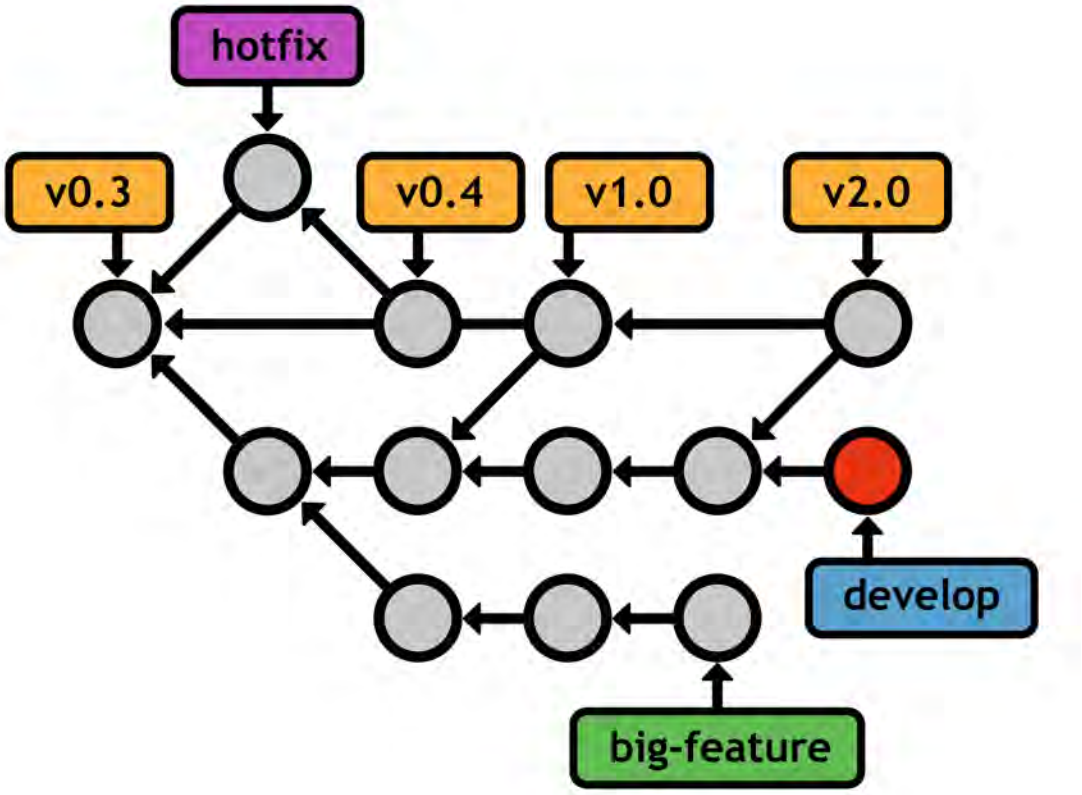

Figure 30: Patching `master` with a hotfix branch

Again, the meanings assigned to each of these branches are purely conventional—Git sees no difference between `master`, `develop`, features, and hotfixes. With that in mind, don't be afraid to adapt them to your own ends. The beauty of Git is its flexibility. When you understand the mechanics behind Git branches, it's easy to design novel workflows that fit your project and personality.

Rebasing

Rebasing is the process of moving a branch to a new *base*. Git's rebasing capabilities make branches even more flexible by allowing users to manually organize their branches. Like merging, `git rebase` requires the branch to be checked out and takes the new base as an argument:

```
git checkout some-feature
git rebase master
```

This moves the entire `some-feature` branch onto the tip of `master`:

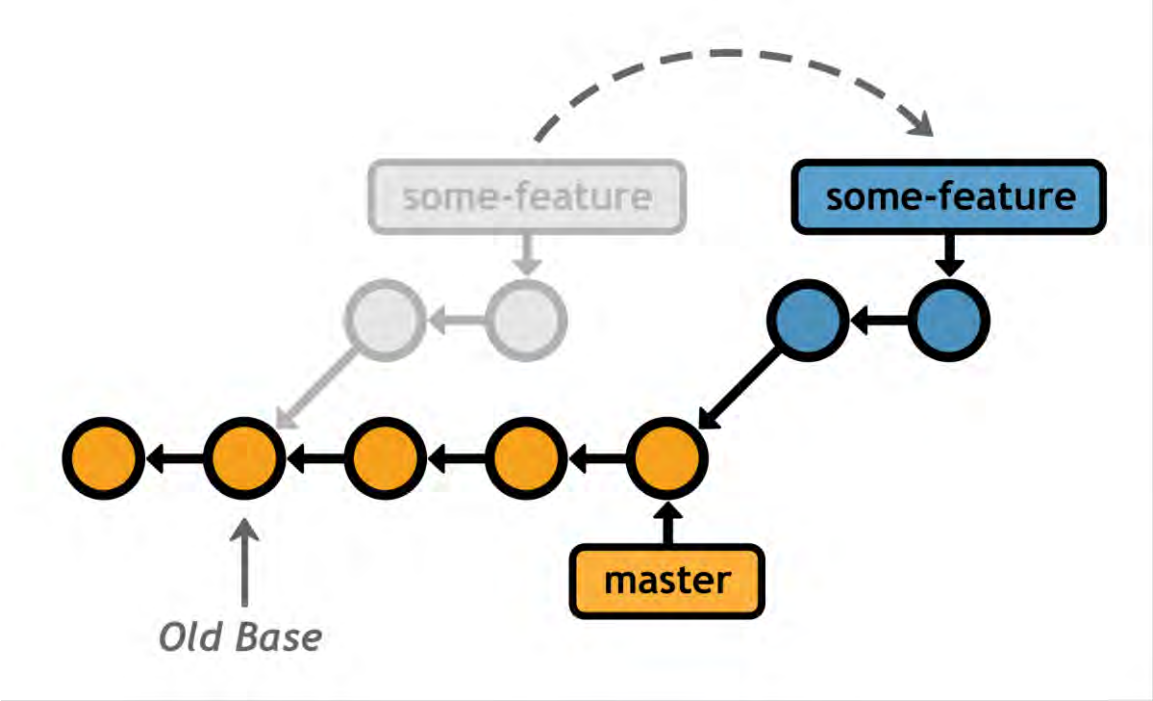

Figure 31: Rebasing `some-feature` onto the `master` branch

After the rebase, the feature branch is a *linear extension* of `master`, which is a much cleaner way to integrate changes from one branch to another. Compare this linear history with a merge of `master` into `some-feature`, which results in the exact same code base in the final snapshot:

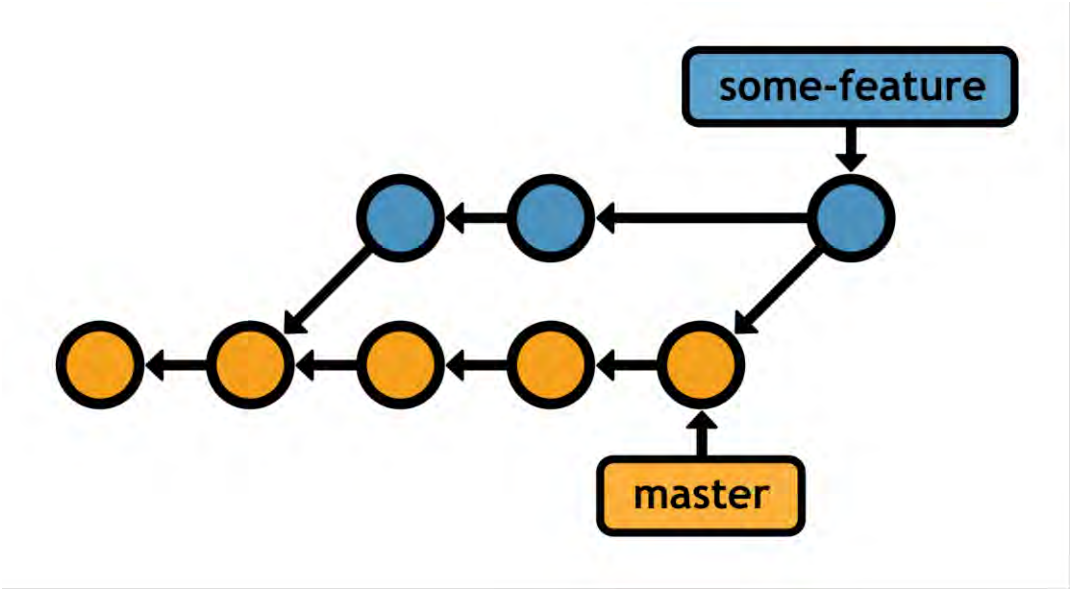

Figure 32: Integrating `master` into `some-feature` with a 3-way merge

Since the history has diverged, Git has to use an extra merge commit to combine the branches. Doing this many times over the course of developing a long-running feature can result in a very messy history.

These extra merge commits are superfluous—they exist only to pull changes from `master` into `some-feature`. Typically, you'll want your merge commits to *mean* something, like the completion of a new feature. This is why many developers choose to pull in changes with `git rebase`, since it results in a completely linear history in the feature branch.

Interactive Rebasing

Interactive rebasing goes one step further and allows you to *change* commits as you're moving them to the new base. You can specify an interactive rebase with the `-i` flag:

```
git rebase -i master
```

This populates a text editor with a summary of each commit in the feature branch, along with a command that determines *how* it should be transferred to the new base. For example, if you have two commits on a feature branch, you might specify an interactive rebase like the following:

```
pick 58dec2a First commit for new feature
squash 6ac8a9f Second commit for new feature
```

The default `pick` command moves the first commit to the new base just like the normal `git rebase`, but then the `squash` command tells Git to combine the second commit with the previous one, so you wind up with one commit containing all of your changes:

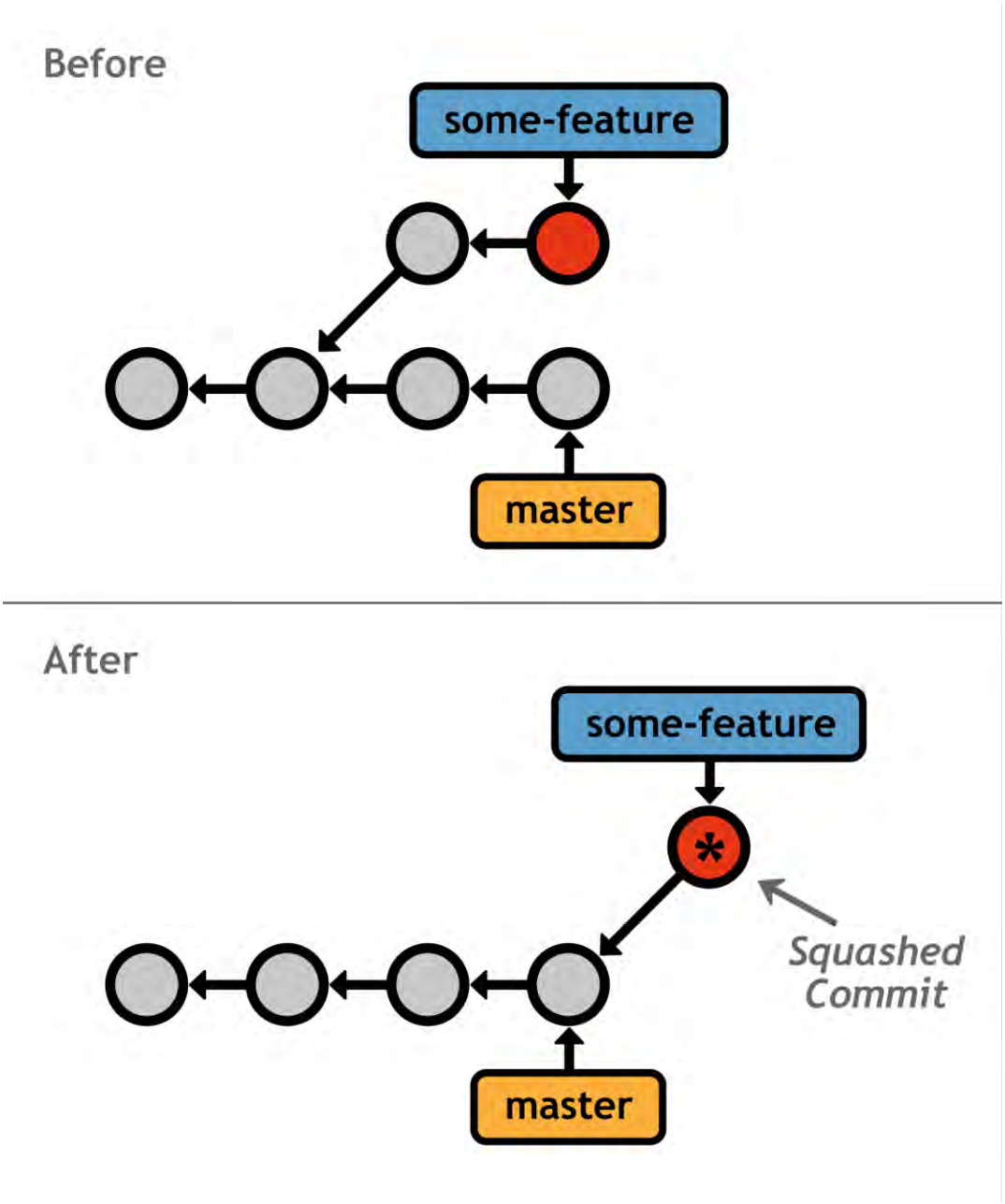

Figure 33: Interactively rebasing the `some-feature` branch

Git provides several interactive rebasing commands, each of which are summarized in the comment section of the configuration listing. The point is interactive rebasing lets you *completely* rewrite a branch's history to your exact specifications. This means you can add as many intermediate commits to a feature branch as you need, then go back and fix them up into meaningful progression after the fact.

Other developers will think you are a brilliant programmer, and knew precisely how to implement the entire feature in one fell swoop. This kind of organization is very important for ensuring large projects have a navigable history.

Rewriting History

Rebasing is a powerful tool, but you must be judicious in your rewriting of history. Both kinds of rebasing don't actually *move* existing commits—they *create* brand new ones (denoted by an asterisk in the above diagram). If you inspect commits that were subjected to a rebase, you'll notice that they have different IDs, even though they represent the same content. This means rebasing *destroys* existing commits in the process of "moving" them.

As you might imagine, this has dramatic consequences for collaborative workflows. Destroying a public commit (e.g., anything on the `master` branch) is like ripping out the basis of everyone else's work. Git won't have any idea how to combine everyone's changes, and you'll have a whole lot of apologizing to do. We'll take a more in-depth look at this scenario after we learn how to communicate with remote repositories.

For now, just abide by the golden rule of rebasing: ***never rebase a branch that has been pushed to a public repository***.

Chapter 6 Remote Repositories

Simply put, a remote repository is one that is not your own. It could be on a central server, another developer's personal computer, or even your file system. As long as you can access it from some kind of network protocol, Git makes it incredibly easy to share contributions with other repositories.

The primary role of remote repositories is to represent other developers within your own repository. Branches, on the other hand, should only deal with project development. That is to say, don't try to give individual developers their own branch to work on—give them a complete repository and reserve branches for developing features.

This chapter begins by covering the mechanics of remotes, and then presents the two most common workflows of Git-based collaboration: the centralized workflow and the integrator workflow.

Manipulating Remotes

Similar to `git branch`, the `git remote` command is used to manage connections to other repositories. Remotes are nothing more than bookmarks to other repositories—instead of typing the full path, they let you reference it with a user-friendly name. We'll learn how we can use these bookmarks within Git in *Remote Workflows*.

Listing Remotes

You can view your existing remotes by calling the `git remote` command with no arguments:

```
git remote
```

If you have no remotes, this command won't output any information. If you used `git clone` to get your repository, you'll see an `origin` remote. Git automatically adds this connection, under the assumption that you'll probably want to interact with it down the road.

You can request a little bit more information about your remotes with the `-v` flag:

```
git remote -v
```

This displays the complete path to the repository. Specifying remote paths is discussed in the next section.

Creating Remotes

The `git remote add` command creates a new connection to a remote repository.

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

After running this command, you can reach the Git repository at `<path-to-repo>` using `<name>`. Again, this is simply a convenient bookmark for a long path name—it does *not* create a direct link into someone else's repository.

Git accepts many network protocols for specifying the location of a remote repository, including `file://`, `ssh://`, `http://`, and its custom `git://` protocol. For example:

```
git remote add some-user ssh://git@github.com/some-user/some-repo.git
```

After running this command, you can access the repository at `github.com/some-user/some-repo.git` using only `some-user`. Since we used `ssh://` as the protocol, you'll probably be prompted for an SSH password before you're allowed to do anything with the account. This makes SSH a good choice for granting write access to developers, whereas HTTP paths are generally used to give read-only access. As we'll soon discover, this is designed as a security feature for distributed environments.

Deleting Remotes

Finally, you can delete a remote connection with the following command:

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

Remote Branches

Commits may be the atomic unit of Git-based version control, but *branches* are the medium in which remote repositories communicate. **Remote branches** act just like the local branches we've covered thus far, except they represent a branch in someone else's repository.

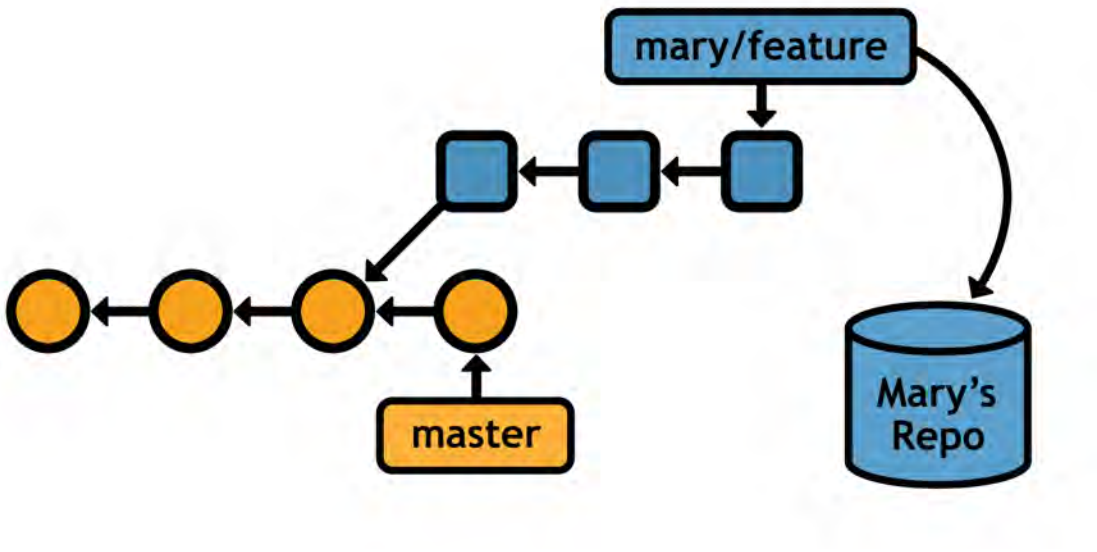

Figure 34: Accessing a feature branch from a remote repository

Once you've downloaded a remote branch, you can inspect, merge, and extend it like any other branch. This makes for a very short learning curve if you understand how to use branches locally.

Fetching Remote Branches

The act of downloading branches from another repository is called **fetching**. To fetch a remote branch, you can specify the repository and the branch you're looking for:

```
git fetch <remote> <branch>
```

Or, if you want to download *every* branch in `<remote>`, simply omit the branch name. After fetching, you can see the downloaded branches by passing the `-r` option to `git branch`:

```
git branch -r
```

This gives you a branch listing that looks something like:

```
origin/master
origin/some-feature
origin/another-feature
```

Remote branches are always prefixed with the remote name (`origin/`) to distinguish them from local branches.

Remember, Git uses remote repositories as *bookmarks*—not real-time connections with other repositories. Remote branches are *copies* of the local branches of another repository. Outside of the actual fetch, repositories are

completely isolated development environments. This also means Git will never automatically fetch branches to access updated information—you must do this manually.

But, this is a good thing, since it means you don't have to constantly worry about what everyone else is contributing while doing your work. This is only possible due to the non-linear workflow enabled by Git branches.

Inspecting Remote Branches

For all intents and purposes, remote branches behave like read-only branches. You can safely inspect their history and view their commits via `git checkout`, but you cannot continue developing them before integrating them into your local repository. This makes sense when you consider the fact that remote branches are *copies* of other users' commits.

The `..` syntax is very useful for filtering log history. For example, the following command displays any new updates from `origin/master` that are not in your local `master` branch. It's generally a good idea to run this before merging changes so you know exactly what you're integrating:

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

If this outputs any commits, it means you are behind the official project and you should probably update your repository. This is described in the next section.

It is possible to checkout remote branches, but it will put you in a detached `HEAD` state. This is safe for viewing other user's changes before integrating them, but any changes you add will be lost unless you create a new local branch tip to reference them.

Merging/Rebasing

Of course, the whole point of fetching is to integrate the resulting remote branches into your local project. Let's say you're a contributor to an open-source project, and you've been working on a feature called `some-feature`. As the "official" project (typically pointed to by `origin`) moves forward, you may want to incorporate its new commits into your repository. This would ensure that your feature still works with the bleeding-edge developments.

Fortunately, you can use the exact same `git merge` command to incorporate changes from `origin/master` into your feature branch:

```
git checkout some-feature
git fetch origin
git merge origin/master
```

Since your history has diverged, this results in a 3-way merge, after which your `some-feature` branch has access to the most up-to-date version of the official project.

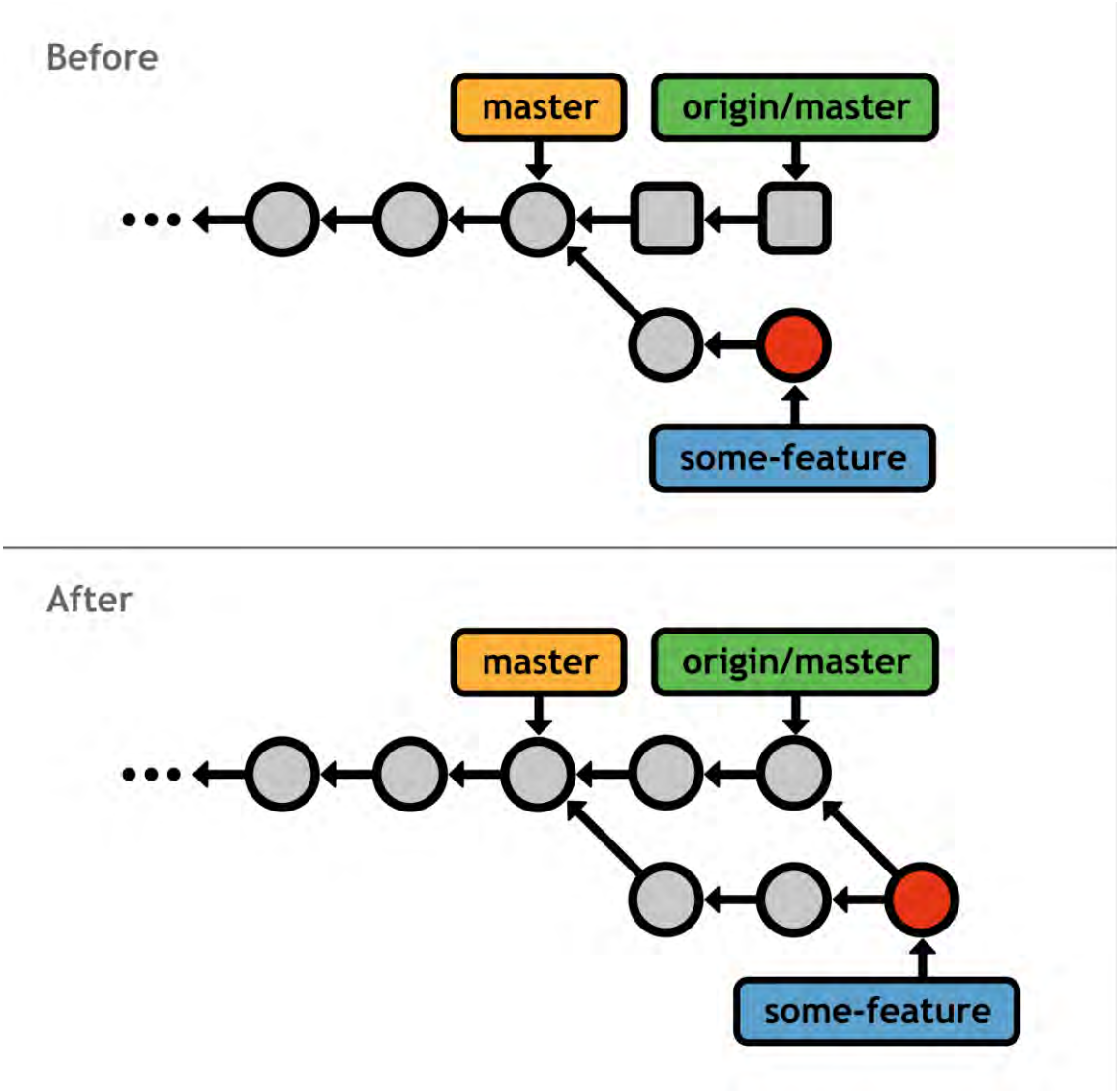

Figure 35: Merging the official `master` into a feature branch

However, frequently merging with `origin/master` just to pull in updates eventually results in a history littered with meaningless merge commits. Depending on how closely your feature needs to track the rest of the code base, rebasing might be a better way to integrate changes:

```
git checkout some-feature
git fetch origin
git rebase origin/master
```

As with local rebasing, this creates a perfectly linear history free of superfluous merge commits:

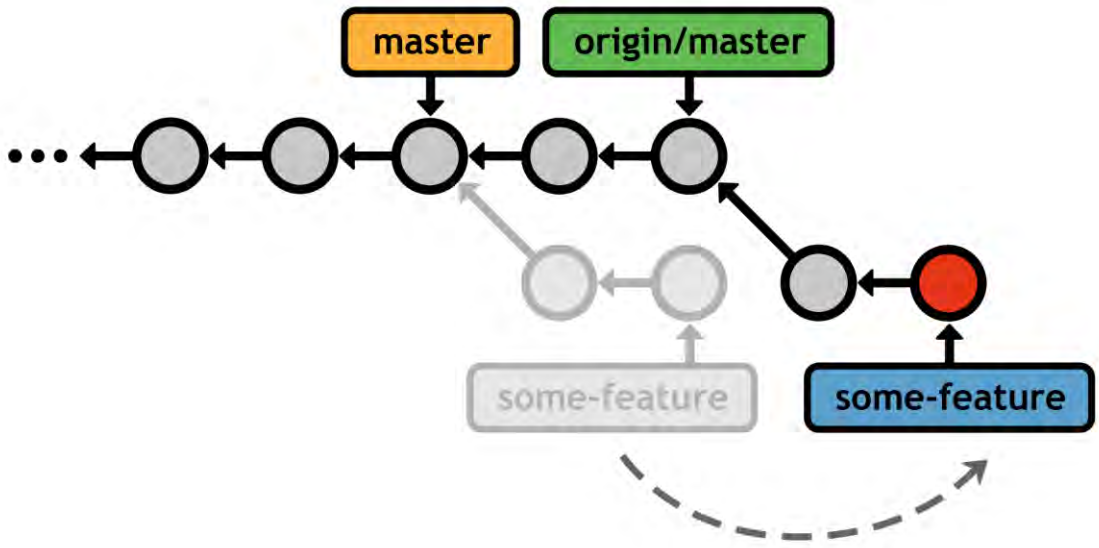

Figure 36: Rebasing the feature branch onto the official `master`

Rebasing/merging remote branches has the exact same trade-offs as discussed in the chapter on local branches.

Pulling

Since the fetch/merge sequence is such a common occurrence in distributed development, Git provides a `pull` command as a convenient shortcut:

```
git pull origin/master
```

This fetches the origin's master branch, and then merges it into the current branch in one step. You can also pass the `--rebase` option to use `git rebase` instead of `git merge`.

Pushing

To complement the `git fetch` command, Git also provides a `push` command. Pushing is *almost* the opposite of fetching, in that fetching imports branches, while pushing exports branches to another repository.

```
git push <remote> <branch>
```

The above command sends the local `<branch>` to the specified remote repository. Except, instead of a *remote* branch, `git push` creates a *local* branch. For example, executing `git push mary my-feature` in your local

repository will look like the following from Mary's perspective (your repository will be unaffected by the push).

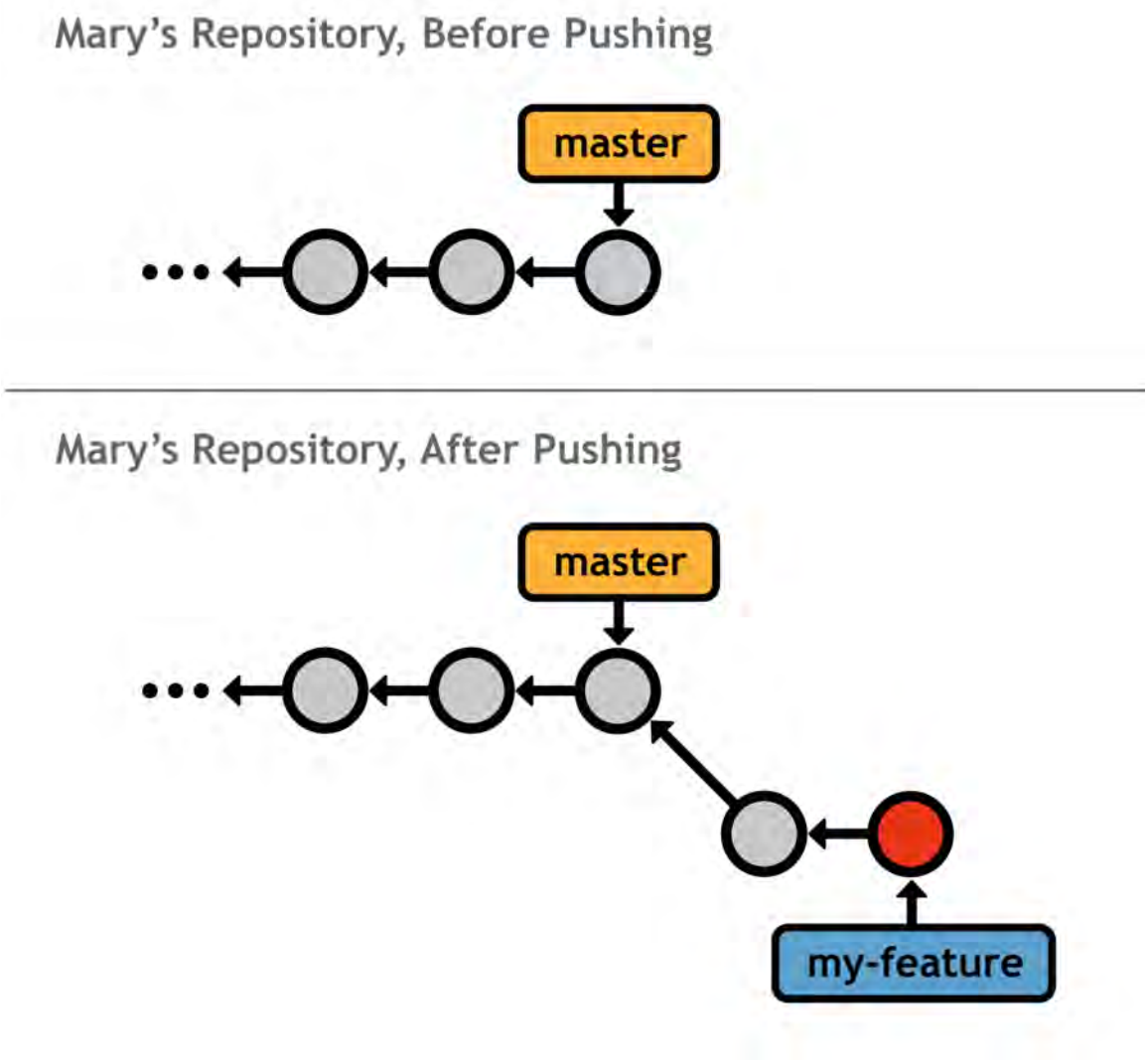

Figure 37: Pushing a feature branch from your repository into Mary's

Notice that `my-feature` is a *local* branch in Mary's repository, whereas it would be a *remote* branch had she fetched it herself.

This makes pushing a dangerous operation. Imagine you're developing in your own local repository, when, all of a sudden, a new local branch shows up out of nowhere. But, repositories are supposed to serve as completely isolated development environments, so why should `git push` even exist? As we'll discover shortly, pushing is a necessary tool for maintaining public Git repositories.

Remote Workflows

Now that we have a basic idea of how Git interacts with other repositories, we can discuss the real-world workflows that are supported by these commands. The two most common collaboration models are: the centralized workflow and the integrator workflow. SVN and CVS users should be quite comfortable with Git's flavor of centralized development, but using Git means you'll also get to leverage its highly-efficient merge capabilities. The integrator workflow is a typical distributed collaboration model and is not possible in purely centralized systems.

As you read through these workflows, keep in mind that Git treats all repositories as equals. There is no "master" repository according to Git as there is with SVN or CVS. The "official" code base is merely a project convention—the only reason it's the official repository is because that's where everyone's `origin` remote points.

Public (Bare) Repositories

Every collaboration model involves at least one *public* repository that serves as a point-of-entry for multiple developers. Public repositories have the unique constraint of being **bare**—they must not have a working directory. This prevents developers from accidentally overwriting each others' work with `git push`. You can create a bare repository by passing the `--bare` option to `git init`:

```
git init --bare <path>
```

Public repositories should only function as *storage facilities*—not development environments. This is conveyed by adding a `.git` extension to the repository's file path, since the internal repository database resides in the project root instead of the `.git` subdirectory. So, a complete example might look like:

```
git init --bare some-repo.git
```

Aside from a lack of a working directory, there is nothing special about a bare repository. You can add remote connections, push to it, and pull from it in the usual fashion.

The Centralized Workflow

The centralized workflow is best suited to small teams where each developer has write access to the repository. It allows collaboration by using a single central repository, much like the SVN or CVS workflow. In this model, *all* changes must be shared through the central repository, which is usually stored on a server to enable Internet-based collaboration.

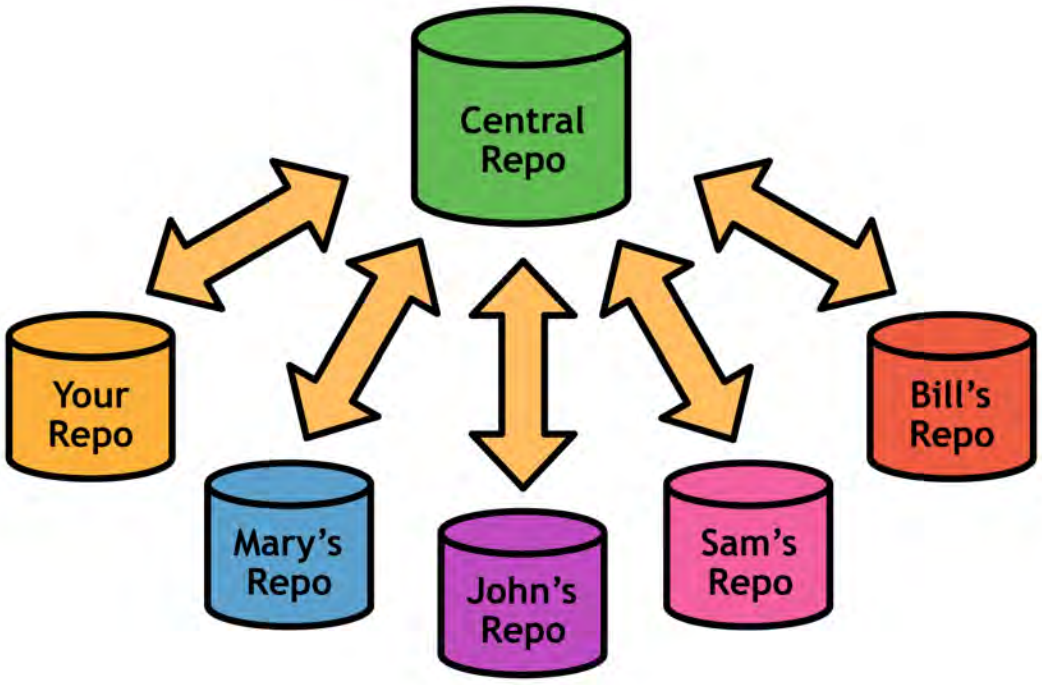

Figure 38: The centralized workflow with many developers

Individual developers work in their own local repository, which is completely isolated from everyone else. Once they've completed a feature and are ready to share their code, they clean it up, integrate it into their local `master`, and push it to the central repository (e.g., `origin`). This also means all developers need SSH access to the central repository.

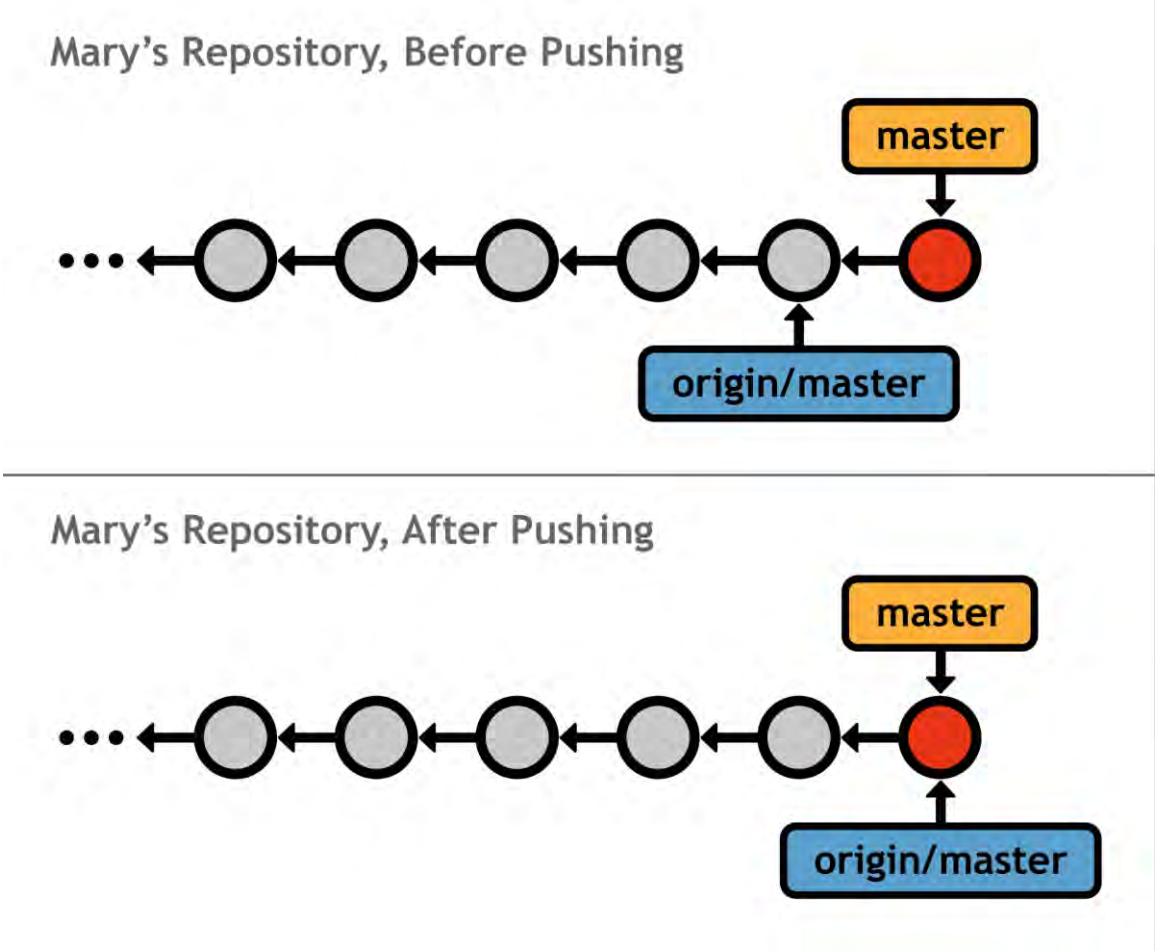

Figure 39: Mary pushing her updates to the central repository

Then, everyone else can fetch the new commits and incorporate them into their local projects. Again, this can be done with either a merge or a rebase, depending on your team's conventions.

This is the core process behind centralized workflows, but it hits a bump when multiple users try to simultaneously update the central repository. Imagine a scenario where two developers finished a feature, merged it into their local `master`, and tried to publish it at the same time (or close to it).

Whoever gets to the server first can push his or her commits as normal, but then the second developer gets stuck with a divergent history, and Git cannot perform a fast-forward merge. For example, if a developer named John were to push his changes right before Mary, we'd see a conflict in Mary's repository:

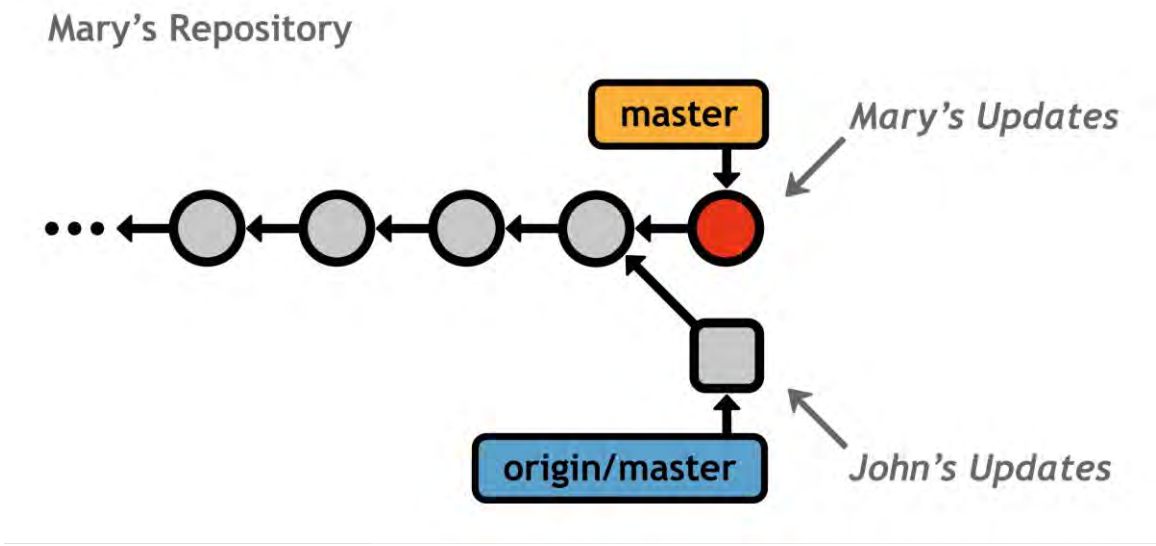

Figure 40: Conflicting updates during a push

The only way to make the `origin`'s master (updated by John) match Mary's `master` is to *overwrite* John's commit. Obviously, this would be very bad, so Git aborts the push and outputs an error message:

```
! [rejected] master -> master (non-fast-forward)
error: failed to push some refs to 'some-repo.git'
```

To remedy this situation, Mary needs to synchronize with the central repository. Then, she'll be able to push her changes in the usual fashion.

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

Other than that, the centralized workflow is relatively straightforward. Individual developers stay in their own local repository, periodically pulling/pushing to the central repository to keep everything up-to-date. It's a convenient workflow to set up, as only one server is required, and it leverages existing SSH functionality.

The Integrator Workflow

The integrator workflow is a distributed development model where individual users maintain a *public* repository, in addition to a private one. It exists as a solution to the security and scalability problems inherent in the centralized workflow.

The main drawback of the centralized workflow is that *every* developer needs push access to the entire project. This is fine if you're working with a small team of trusted developers, 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 him push access to the central repository, since he could start pushing all sorts of random commits, and you would effectively lose control of the project.

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

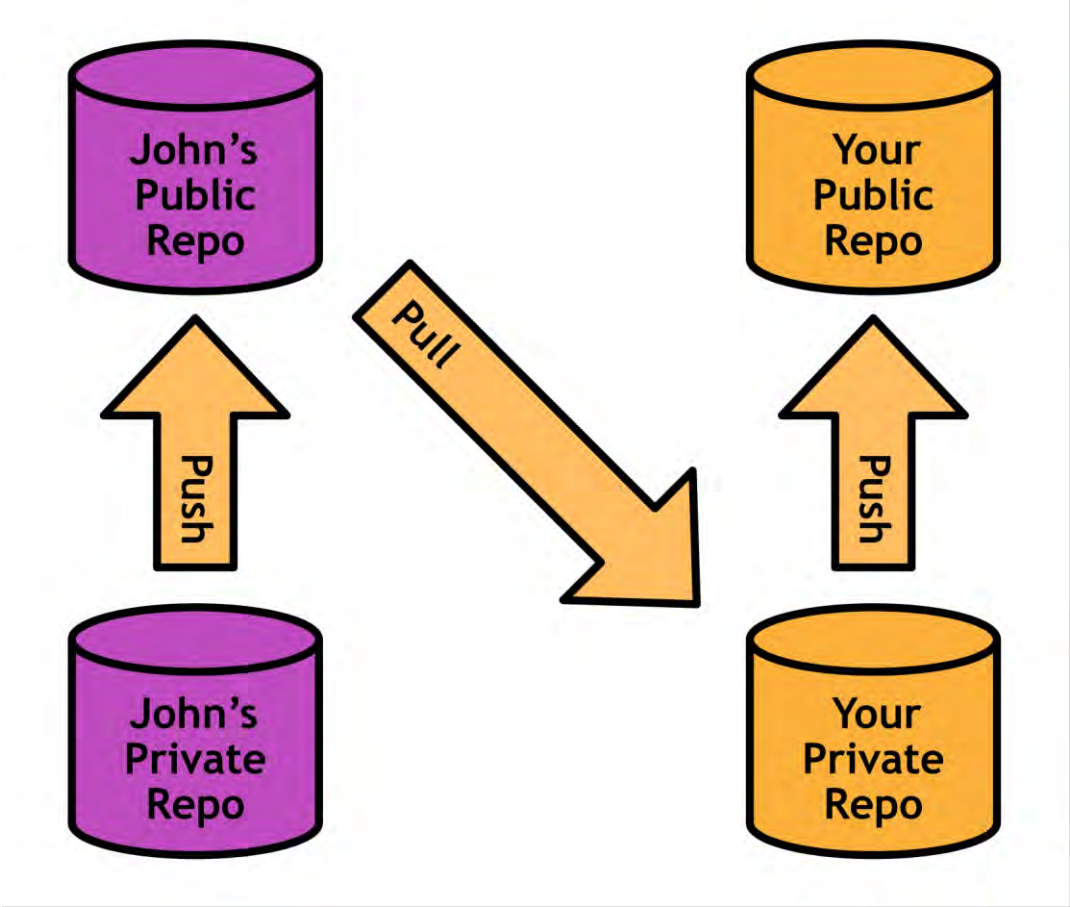

Figure 41: Integrating changes from John's public repository

In this workflow, individual developers only need push access to *their own* public repositories. Contributors use SSH to push to their public repositories, but the integrator can fetch the changes over HTTP (a read-only protocol). This makes for a more secure environment for everyone, even when you add more collaborators:

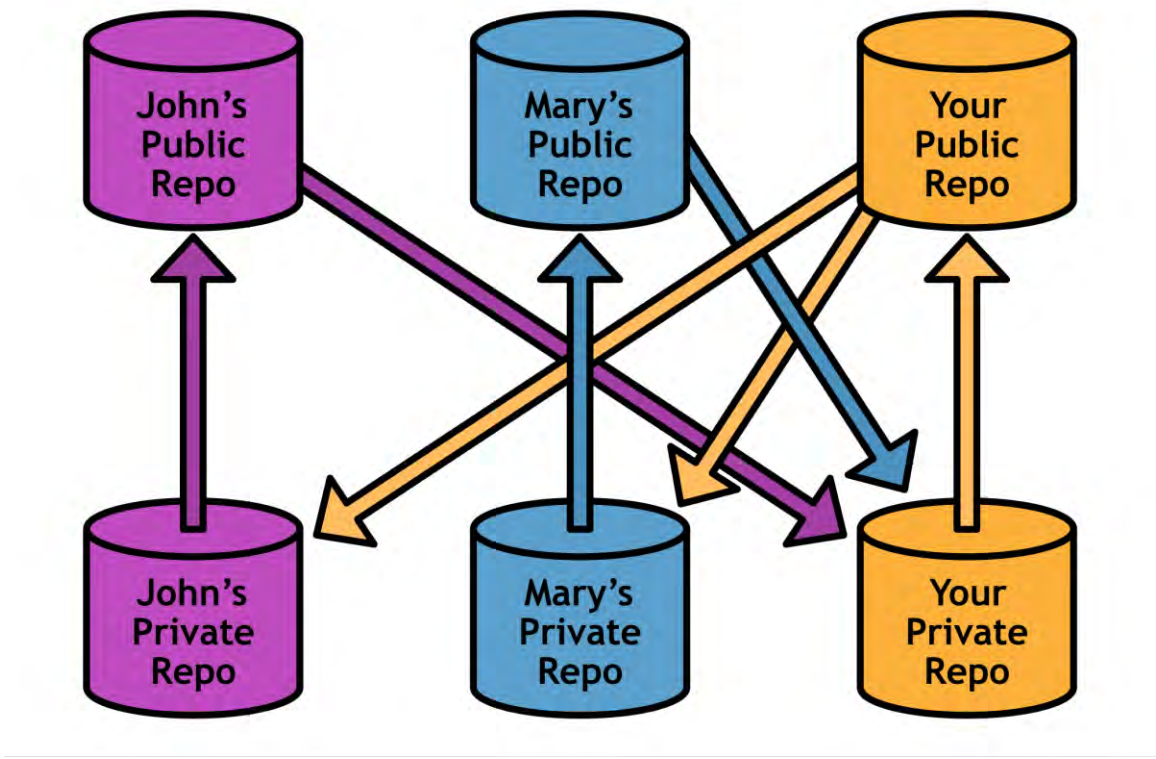

Figure 42: The integrator workflow with many developers

Note that the team must still agree on a single "official" repository to pull from—otherwise changes would be applied out-of-order and everyone would wind up out-of-sync very quickly. In the above diagram, "Your Public Repo" is the official project.

As an integrator, you have to keep track of more remotes than you would in the centralized workflow, but this gives you the freedom and security to incorporate changes from any developer without threatening the stability of the project.

In addition, the integrator workflow has no single point-of-access to serve as a choke point for collaboration. In centralized workflows, everyone must be completely up-to-date before publishing changes, but that is not the case in distributed workflows. Again, this is a direct result of the nonlinear development style enabled by Git's branch implementation.

These are huge advantages for large open-source projects. Organizing hundreds of developers to work on a single project would not be possible without the security and scalability of distributed collaboration.

Conclusion

Supporting these centralized and distributed collaboration models was all Git was ever meant to do. The working directory, the stage, commits, branches, and remotes were all specifically designed to enable these workflows, and virtually everything in Git revolves around these components.

True to the UNIX philosophy, Git was designed as a suite of interoperable tools, not a single monolithic program. As you continue to explore Git's numerous capabilities, you'll find that it's very easy to adapt individual commands to entirely novel workflows.

I now leave it to you to apply these concepts to real-world projects, but as you begin to incorporate Git into your daily workflow, remember that it's not a silver bullet for project management. It is merely a tool for tracking your files, and no amount of intimate Git knowledge can make up for a haphazard set of conventions within a development team.