



Intro to Software Version Control: Not So Basic Git

Annette Spyker Mechelke
Twitter: @interannette
<https://annette.mechelke.us>



What will you learn in this talk?

- Why you should use version control in general, and Git in particular.
- How to get started with Git.
- An overview of the internals of Git.
- An introduction to the way teams use Git.
- You **won't** learn any git commands



Who am I?

- Developer of Java web services.
- Work at Fetch Rewards
- Not a Git expert, but a Git user for 5 years
- Math major turned software developer



Who are you?

Who is a student? Working in IT? Working in a non-IT field?

Who has used version control before? Who has used Git before? Who is comfortable with Git?

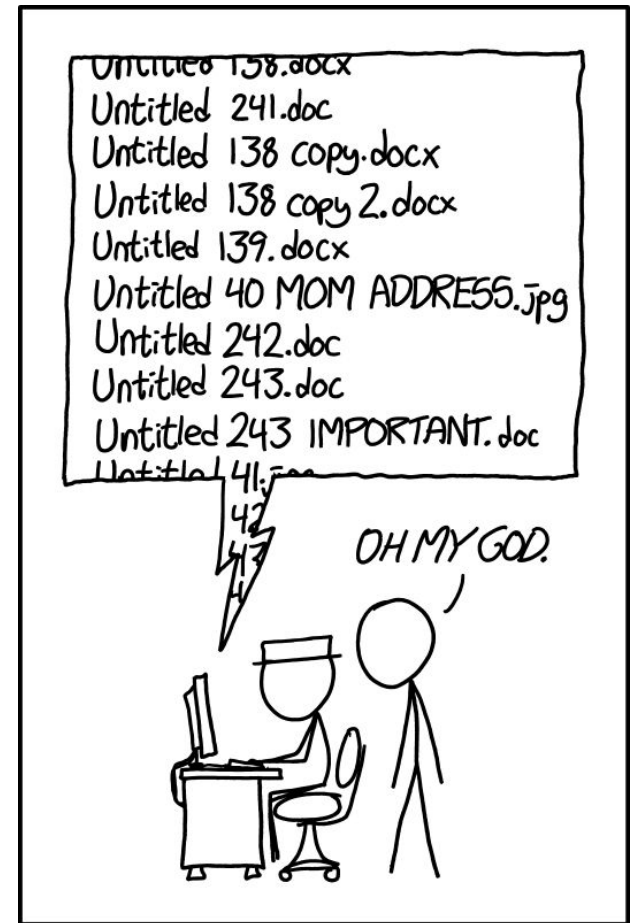
What is version control? And why do you need it?

...version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information.

From Wikipedia for [Version control](#)

Why do you need version control?

<https://xkcd.com/1459/>



PROTIP: NEVER LOOK IN SOMEONE ELSE'S DOCUMENTS FOLDER.



Why do you need it when working on a team?

- Synchronization Of Code
 - Allows everyone to work on the same version of the code, and keep up to date as changes occur
 - Example: you have a project you have completed and you want to share the project files with a team member
- Integration Of Changes
 - Allow team members to work independently, and easily integrate changes when complete.
 - Example: you and a teammate are working on independent features in a project. How do you combine your work once you have both completed your feature?
- Tracking Changes
 - Allows team members to trace each change made to the repository, and each change usually includes a description of why the change occurred and who made the change.
 - Example: The only developer that worked on a feature leaves your team. You run into a problem and want to understand why it was developed in the way that it was.



Why do you need it when working alone?

- Easily revert after making a mistake
 - Example: You make a changes, but run into unintended consequences and want to undo the changes.
- Test out a new approach without breaking project
 - Example: You want to try refactoring part of your code, but it will take a long time to do. In the meantime you need to continue to fix bugs.
- Backup
 - If you host your version control system somewhere besides your personal computer, you have a built in backup of your latest work
 - Example: your hard drive dies and all your code is lost



What type of systems are available?

- Centralized System
 - There is a central, shared server and all operations must go through that server.
 - The central server maintains the authoritative copy of the repository
 - Examples: SVN, CSV, Perforce
- Distributed System
 - Each copy of the repository contains the entire history and associated metadata for the repository. There is not necessarily a central authoritative copy of the repository
 - Most common operations can be completed without communicating with a central server
 - Examples: Git, Mercurial



Benefits Of Distributed Systems

- + Most operations are faster in distributed system than in centralized systems
- + Most operations can be completed offline
- + Easy to share changes among peers, without sending to central server
- + More flexibility in workflows than with centralized systems



Common Terminology

- **Repository** - The data structure consisting of the files being tracked, history, and metadata.
- **Clone** - Creating a copy of an existing repository
- **Commit** -
 - Verb: submit local changes made in the working copy into the repository
 - Noun: the object that results from the submitted changes
- **Push/Pull** - copy revisions from one repository to another
 - We say “push” when we are uploading our changes to a remote server
 - We say “pull” when we are retrieving changes from a remote server



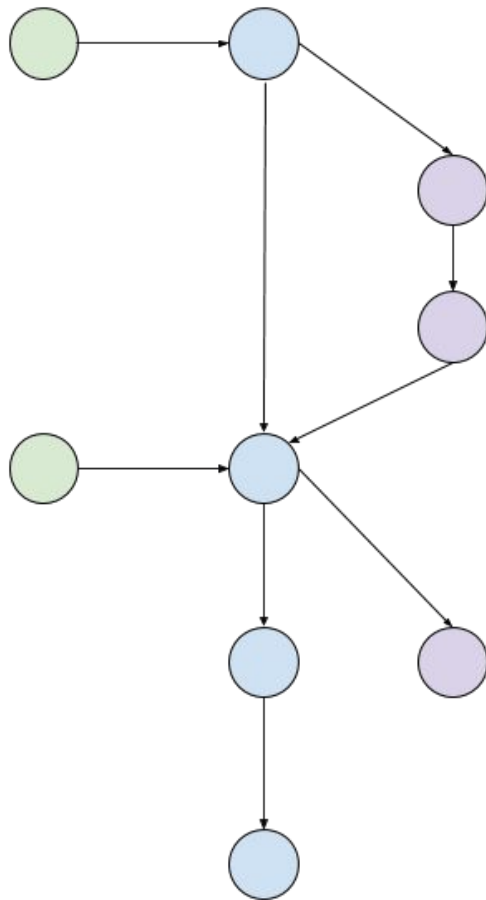
Basic Workflow

1. Ensure you have the most up to date version
 - a. **Clone** the **repository** (first time)
 - b. **Pull** from the remote **repository**
2. Make your changes.
3. **Commit** your changes.
4. **Push** your changes to make them available for others

Common Abstraction

A repository and its history can be thought of as a Directed Acyclic Graph (DAG)

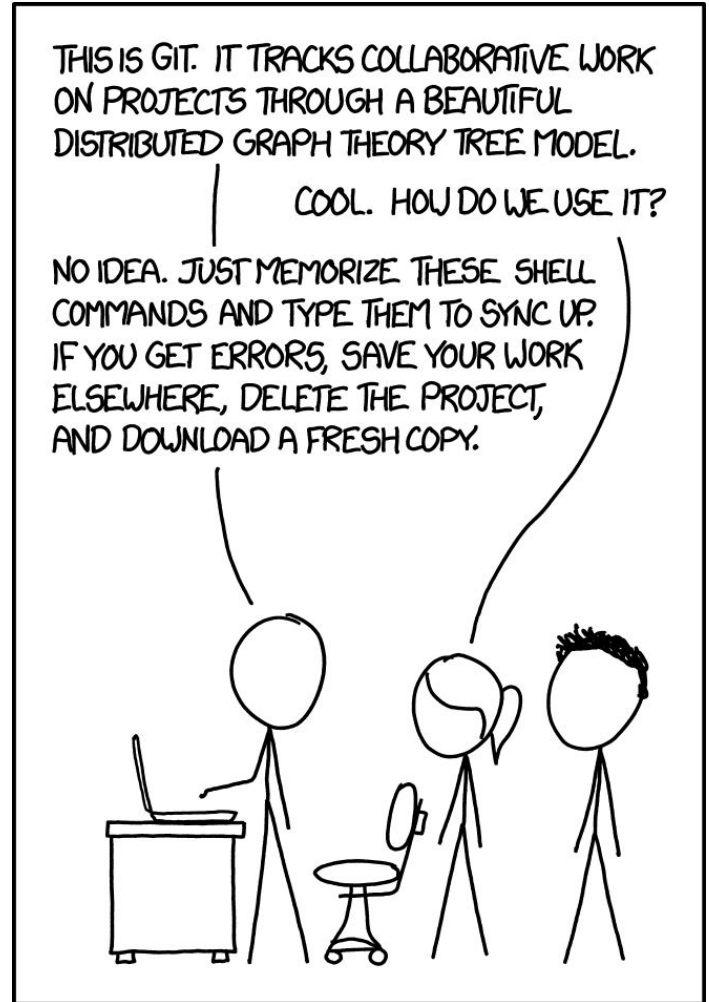
- Graph: *commits represented by nodes on the graph*
- Directed: *Nodes have one or more parents. Parent commit (i.e. the state of the repository before)*
- Acyclic: *no self reference of commit to itself*



What is Git? Why should you learn it? How do you start?

What is Git?

<https://xkcd.com/1597/>





What is Git?

- Open source distributed version control system
- Created in 2005 by Linux kernel development team (primarily Linus Torvalds) when the formerly free DVCS they were using, BitKeeper, was no longer free. Their goals for the new system were:
 - Speed
 - Strong support for non-linear development (thousands of parallel branches)
 - Fully distributed
 - Able to handle large projects like the Linux kernel efficiently (speed and data size)
- Git is currently the most commonly used source control system (according to community surveys)



Why should you learn Git?


- It's popular
 - De facto standard for version control in many circles, specifically open source projects
 - Many free online resources for learning or getting help
 - Many options for hosting repositories
- Flexible branching workflow
- Lightweight, no cost
- It's fun!



How to get started with Git?

Choose a user interface and install the required tools

- Command line
 - Available for Mac, Windows, Linux
 - Generally the same across platforms, versions
- GUI
 - Stand alone applications
 - SourceTree
 - GitHub desktop
 - and many others (list at <https://git-scm.com/downloads/guis>)
 - IDE plug ins
 - Eclipse
 - JetBrains products
 - Visual Studio



How to get started with Git?

Set up your config

- Each commit in git includes your name and email address. You should set these before beginning to use Git.
- You might also want to set the editor to your preferred text editor.
- You can set these at a global (i.e. system) level



How to get started with Git?

Try out a tutorial

- try.github.io https://try.github.io/
- [Learn Git With Bitbucket Cloud](https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud)
https://www.atlassian.com/git/tutorials/learn-git-with-bitbucket-cloud
- [git - the simple guide](http://rogerdudler.github.io/git-guide/)
http://rogerdudler.github.io/git-guide/



How to get started with Git?

Where to host?

- Hosting Service
 - Github
 - Open source
 - Github pages
 - BitBucket
 - Private repos for individual accounts
 - GitLab
 - Focused on integrated development life cycle
- Self host
 - No remote
 - Publicly accessible server

**What does the basic workflow
look in Git?**

Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.

Git thinks about its data more like a stream of snapshots.



How does Git store these snapshots?

- By default, Git stores all its internal data in the hidden “.git” folder in the root of your repository.
- All content is stored in the “objects” directory, referenced by the SHA1 digest of the contents.

- When I add the file to hello.txt to my Git repository

`my_repo/hello.txt`

Git will add the file to its internal storage

`my_repo/.git/objects/[SHA1]`



What is SHA1?

A cryptographic hash function.

What is a cryptographic hash function?

A function that takes a stream of data and outputs a fixed length alphanumeric string.

- The same file will always result in the same value when hashed.
- Any changes to a file will result in a new value when the updated file is hashed. In other words, two different files will have different hashes.
- Git uses the SHA1 hash of files to reference and store file contents
 - Allows git to avoid storing duplicate copies of a file
 - Allows git to compare files between repositories without sending the entire file



Parts Of A Git Repository

- The **working directory** is the actual contents the file system.
 - These are the files you edit.
 - If you do nothing to tell git about the edits it doesn't know changes occurred
- The **index**, or staging area, is a file in the .git directory that stores the list of changes that will be in the next commit
 - You can think of the index as a list of changes
- The **commit history**, is the “series of snapshots”. It is the rest of .git directory.



States of a File

- Modified
- Staged
- Committed

1. You start with a “clean” working directory.
2. You make some changes to hello.txt. hello.txt is **modified**.
3. You *add* hello.txt to the index. hello.txt is now **staged**.
4. You *commit* your changes. hello.txt is now **committed**. You have a clean working directory again.
5. Now would be a good time to *push* to your remote server.



More States of a File


- Untracked
 - When a new file is created in the working directory, git does not know about it. At this point we say it is **untracked**.
- Tracked
 - Once a new file is staged to the index git knows about it and we say it is **tracked**.
 - If you are editing an existing file, before you add it to the index, it is both **modified** and **tracked**.
- Ignored
 - You can tell git to **ignore** certain files. This prevents you from accidentally adding them and making them **tracked**.



Ignoring Files

- Often there are files you do not want Git to track. E.g. log files, build results, temporary files.
- You can tell Git to ignore files in a few different ways
 - .gitignore file
 - You can create a .gitignore file in any directory and git will honor that
 - The most common is to create a .gitignore file at the root of the working directory
 - You can include specific file names, or include patterns to specify classes of files. (i.e. *.log)
 - .gitignore files are tracked like any other files in your repository
 - You can add config values to your system wide configuration that will apply across projects.

Making A Commit



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.


<https://xkcd.com/1296/>

See also: [Commit Logs From Last Night](#)



Best Practices For Commits

- Git forces you to provide a commit message. You should make it informative to future developers (and future you).
- Match any agreed upon style in your commit message. E.g. reference issue identifier, follow formatting guidelines .
- Commit frequently. Don't save up a bunch of changes to commit at once.
- Don't commit half implemented features. But feel free to break up large features into smaller increments.
- Ensure you code compiles/builds/runs before committing.
- If available, ensure tests pass.



Not ready to commit?

Stash! Or amend!

- The command stash
 - Stores the state of any tracked files in your working directory and index.
 - Unless you specify the `--include-untracked` file argument, untracked files will not be stashed.
 - The `--patch` lets you interactively decide what changes to stash
- Commit what you have. Amend the commit later.
 - Works well when working with branches.
 - **Do not amend any commits you have already pushed.**
- In either case, your changes aren't saved to a remote server.

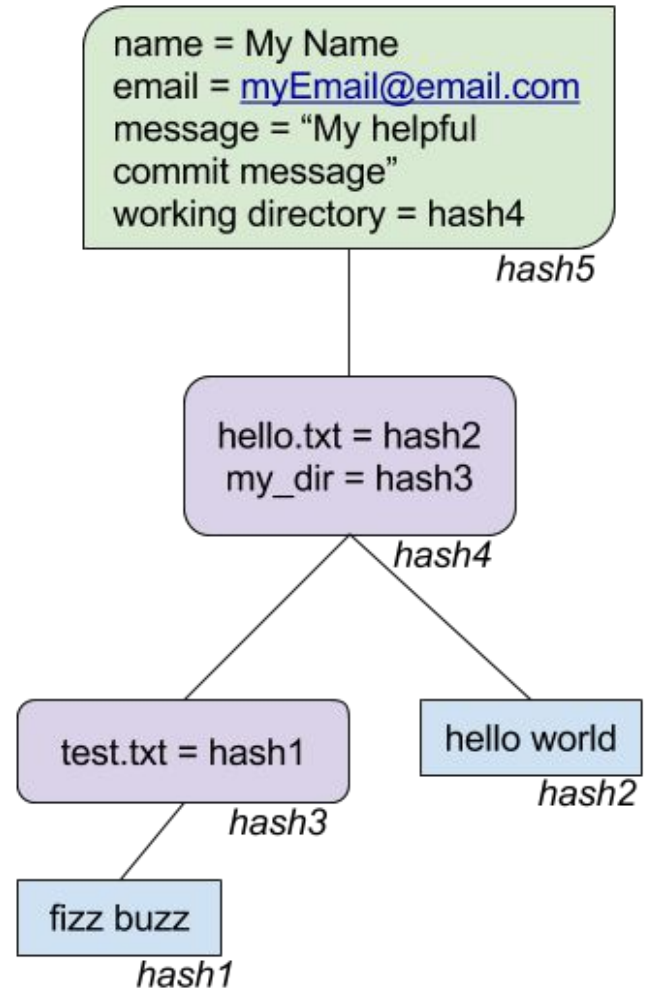


Where are commits stored?

- When you add a file to the index, Git computes the SHA1 for the file and stores the file in the **object store**.
 - Objects in the object store are stored in `.git/objects/[first 2 characters of SHA1]/[file named by the remaining 38 characters of SHA1]`
 - They are compressed before being stored. But may eventually get further compressed by being moved to a pack file.
- Git makes an object with your commit metadata (name, email, commit message, parent commit) and the index.
- It then computes the SHA1 of that content, and stores it in the object store just like the files in your directory.

What objects are in the object store?

- Blobs, e.g. files
- Trees, e.g. index
- Commits





Viewing Git History

- The command *log* lets you inspect previous commits.
 - There are many options available for the log command.
 - oneline
 - graph
 - Try setting up a git alias once you have found a display you like
- You can use gitk or other programs to visualize the Git history.



How to interact with other repositories?

- Remote repositories are the other versions of your repository, usually stored on other servers.
 - Permissions can be enforced on remotes, they can either be read/write or only read.
 - You can have references to many different remote repositories.
 - The *remote* command will list all known remote repositories
 - The remotes for a repository are stored in `.git/refs/remotes`
- The *pull* command will retrieve the contents of a remote reference and apply it to your local repository.
- The *push* command takes a remote in it's list of arguments, specifying which remote repository to push to. If you have conflicting changes, the push command will fail.

**What is a branch? How to teams
use them?**

Branching ... is the duplication of an object under revision control ... so that modifications can happen in parallel along both branches.

Branching also generally implies the ability to later merge or integrate changes back onto the parent branch. ... A branch not intended to be merged ... is usually called a fork.

From Wikipedia for [Branching \(version control\)](#)



Why use branches?

- Use separate branches for different release versions. This allows you to keep the existing version available for quick fixes, while beginning development on the next version
- Isolate research to a branch. You might want to test out a new approach. You can do that on a branch, then merge it in if you want to use it. Or delete the branch if you decide not to use it.
- Use a branch to serve as the place where multiple sets of changes are integrated and tested.



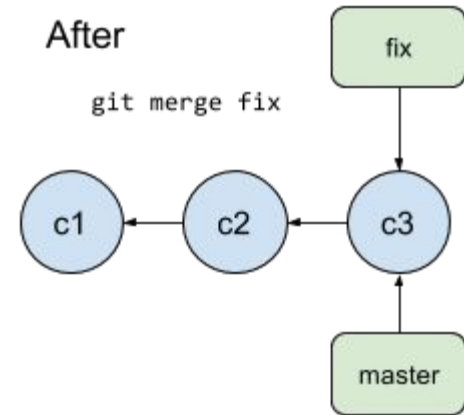
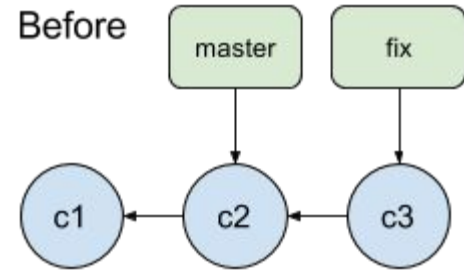
How Does Git Handle Branching?

- Git branches are lightweight and fast, in contrast to many other VCSs.
- In Git a **branch** is simply a pointer to a commit object.
 - Creating a new branch is simply a matter of making a new pointer.
 - When you commit while on a branch, the pointer for that branch is updated to point to the new commit.
 - These pointers are stored in `.git/branches`
- Git keeps a special pointer called **HEAD** that points to the local branch you are currently on.
- Changing branches results:
 - in the HEAD pointer being moved to point to the new branch and
 - updating the working directory to the state indicated in the commit HEAD is now pointing to.

Merging Branches

Fast Forward Merge

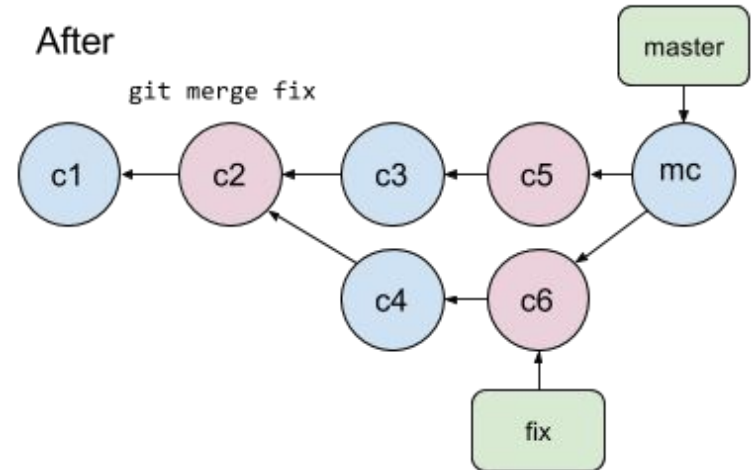
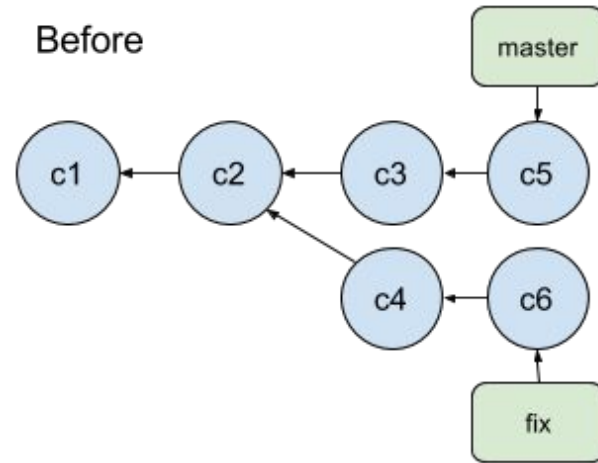
- A merge where there is no divergent work.
- Git can simply update the pointer to point to the latest commit.



Merging Branches

Three Way Merge

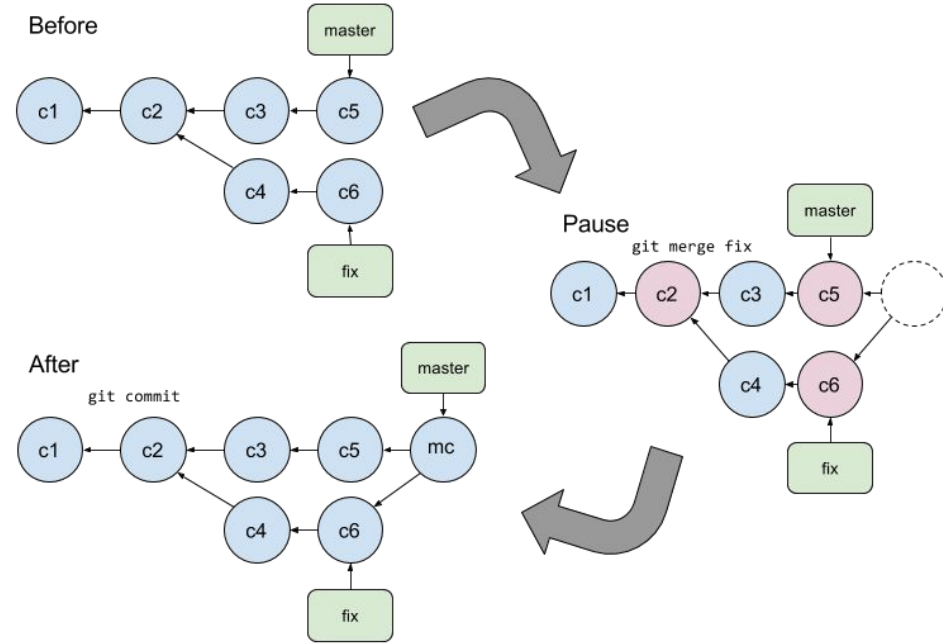
- If the branches have diverged, Git attempts to merge the two branches automatically.
- If successful, Git commits the merged content as a **merge commit**. A merge commit is just a regular commit except that it has two parents.
- It is called a **three way merge** because Git uses the two branches and their common ancestor to compute the merge.



Merging Branches

Merge With Conflicts

- If Git cannot automatically merge the contents, it pauses the process and prompts you to resolve the conflicts manually.
- Once you have resolved the conflicts, you manually create the merge commit.





What does a conflicted merge look like?

- Any files that do not have conflicts are merged and added to the index.
- Any files that have merge conflicts are not added to the index.
- Files with conflicts are updated in your working directory using conflict markers.
- You **resolve conflicts** by updating the files, and adding the files to the index.

- Conflict markers look something like this:

```
<<<<<< HEAD:hello.txt
Hello world
=====
Hello world!
>>>>>> fix:hello.txt
```



Common Branch Terms

- The branch **master** is usually used for the current production version of the code.
 - Most repositories have a master branch because Git creates it when you initialize a repository.
 - To Git, there is nothing special about master. It is just like any other branch.
- Many teams utilize **long running branches** to integrate different features. Test builds are often based off of these branches.
- **Feature branches** are used to contain all the work for a given feature until it is complete
- **Remote tracking branches** are references in your local repository to branches on a remote server.



Working With Remote Tracking Branches

- Remote tracking branches are referenced by a name of the form `<remote>/<branch>`, e.g. `origin/master`
- In most cases you have a local and remote tracking pair. (Git will handle this automatically if the branches have the same name.)
- You can manually set the remote tracking branch if Git does not automatically pair the local and remote branch.
- You can use the *fetch* command to retrieve remote changes without updating your local copy. Git *pull* fetches any changes in the remote branch and applies them to your local repository.



Workflows In Git

- Because Git is very flexible, there are many different approaches to using Git in the software development life cycle.
- These workflows should be thought of as a starting place. You are free to modify them so they work with your team's process.
- The most important part of a Git workflow is that your team agrees on it and everyone follows it. (Or a compatible variation.)
- Some things to consider when selecting, or honing, a workflow:
 - How does your team work? Git should be a tool that enables your team, not a hoop you have to jump through.
 - What is the “standard” process? Everyone should know how it works.
 - What do you do if you make a mistake? Mistakes happen, you need to have a plan for how to recover.



Centralized Workflow

Developers make changes on local tracking branches of master.

Features are pushed to remote master when complete.

Before pushing, the developer must ensure they are up to date with remote master.

- In this workflow there are no branches besides master.
- Centralized workflows use Git to mimic the behavior of a centralized VCS.
- Useful when transitioning a team from a centralized system to a Git
- Useful for individuals or very small teams.
- Requires that features be able to apply to remote master without conflicts.



Feature Branch Workflow

Central repository master serves as the official current version of code.

All development is done on a feature branch and merged into master when complete.

- This branching strategy is used as the basis for more complex Git workflows.
- Popular with teams that utilize continuous integration or continuous delivery
- Feature branches should be pushed to the central server.
- Pull requests are often utilized when merging a feature branch into master.



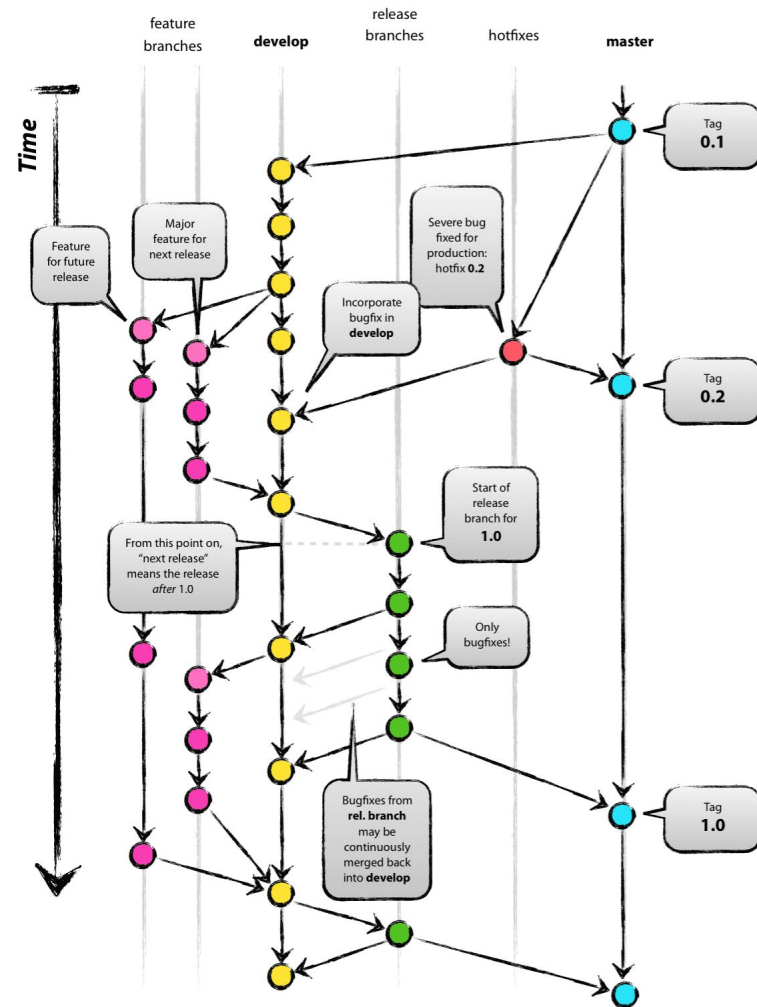
Gitflow Workflow

Similar to the feature branch workflow, but with a few long running branches.

Each branch has a very specific purpose and rules about how it should be used.

- First outlined in 2010 in a blog post [A successful Git branching model](#) by [Vincent Driessen](#)
- Based on feature branches. Teams can still utilize pull requests.
- Useful for large teams with strict release cycles or process requirements.
- Gitflow plugin is available to facilitate the operations specified in the workflow

Gitflow Workflow Illustrated





Fork Workflow

Official central repository that only the maintainer can push to.

Contributors each create their own “fork” of the official repository.

Contributors complete their work in their own repository, then submit a pull request to the maintainer of the official repository.

- Useful for public, open source projects.
- A “fork” is just a clone of the original repository.
 - We use the term fork for the server side copy of the repository
 - Each users will also have a local copy of their forked repository.
- Github and Bitbucket have these operations (fork and pull request) built into their interfaces.

But wait, there's more!



Other Features To Be Aware Of

- Some things you should know about as you dive into Git. In no particular order:
 - Git Hooks
 - Git blame
 - Tags
 - Rebase
 - Cherry pick



Resources

- Pro Git by Scott Chacon and Ben Straub (free ebook version)
- Version Control With Git by Jon Loeliger, Matthew McCullough
- Atlassian Tutorials - <https://www.atlassian.com/git/tutorials>
- Wikipedia
 - Version Control
 - Branching (Version Control)



Questions?

Thank you!
