Versioning and Collaboration with GIT

1. What is version control

Version control systems (VCSs) are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting.

Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

Version control software is an essential part of the every-day of the modern software team's professional practices. Once accustomed to the powerful benefits of version control systems, many developers wouldn't consider working without it even for non-software projects.

One of the most popular VCS tools in use today is called Git. Git is a Distributed VCS, a category known as DVCS.

1.1. Why Use a Version Control System?

1.1.1. Collaboration

Without a VCS in place, you're probably working together in a shared folder on the same set of files. Shouting through the office that you are currently working on file "xyz" and that, meanwhile, your teammates should keep their fingers off is not an acceptable workflow. It's extremely error-prone as you're essentially doing open-heart surgery all the time: sooner or later, someone will overwrite someone else's changes.

With a VCS, everybody on the team is able to work absolutely freely - on any file at any time. The VCS will later allow you to merge all the changes into a common version. There's no question where the latest version of a file or the whole project is. It's in a common, central place: your version control system.

Other benefits of using a VCS are even independent of working in a team or on your own.

1.1.2. Storing Versions (Properly)

Saving a version of your project after making changes is an essential habit. But without a VCS, this becomes tedious and confusing very quickly:

- How much do you save? Only the changed files or the complete project? In the first case, you'll have a hard time viewing the complete project at any point in time - in the latter case, you'll have huge amounts of unnecessary data lying on your hard drive.
- How do you name these versions? If you're a very organized person, you might be able to stick to an actually comprehensible naming scheme (if you're happy with "acme-inc-redesign-2013-11-12-v23"). However, as soon as it comes to variants (say, you need to prepare one version with the header area and one without it), chances are good you'll eventually lose track.
- The most important question, however, is probably this one: How do you know what exactly is different in these versions? Very few people actually take the time to carefully document each important change and include this in a README file in the project folder.

A version control system acknowledges that there is only one project. Therefore, there's only the one version on your disk that you're currently working on. Everything else - all the past versions and variants - are neatly packed up inside the VCS. When you need it, you can request any version at any time and you'll have a snapshot of the complete project right at hand.

1.1.3. Restoring Previous Versions

Being able to restore older versions of a file (or even the whole project) effectively means one thing: you can't mess up! If the changes you've made lately prove to be garbage, you can simply undo them in a few clicks. Knowing this should make you a lot more relaxed when working on important bits of a project.

1.1.4. Understanding What Happened

Every time you save a new version of your project, your VCS requires you to provide a short description of what was changed. Additionally (if it's a code / text file), you can see what exactly was changed in the file's content. This helps you understand how your project evolved between versions.

1.1.5. Backup

A side-effect of using a distributed VCS like Git is that it can act as a backup; every team member has a full-blown version of the project on his disk - including the project's complete history. Should your beloved central server break down (and your backup drives fail), all you need for recovery is one of your teammates' local Git repository.

2. Basics

Let’s look at a basic git usage flow. Later on we’ll dig into details and advanced usage

2.1. Setting up a repository

A Git repository (*repo* for short) is a virtual storage of your project. It allows you to save versions of your code, which you can access when needed.

2.1.1. Initializing a new repository: git init

To create a new repo, you'll use the *git init* command. git init is a one-time command you use during the initial setup of a new repo. Executing this command will create a new .git subdirectory in your current working directory. This will also create a new master branch.

This example assumes you already have an existing project folder that you would like to create a repo within. You'll first cd to the root project folder and then execute the git init command.

---

ls

create a folder: `mkdir gitlesson`

cd to that folder: `cd gitlesson`

create a file (for example, `pico app.js` and enter `const port = 3000`; and save)

`ls -la`: notice that the folder is clean, it only has that file

now do:

`git init`: system answers with Initialized empty Git repository in....

`ls -la`: notice the new .git file

`git status`: sys responds with (don’t worry for now if you don’t understand this text below)

```
On branch master
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        app.js
nothing added to commit but untracked files present (use "git
add" to track)
```

notice how git tells you everything very nicely.

`git log --oneline`: system says no commits yet

Before you proceed, there are a couple of things you probably want to do:

Tell Git who you are and set your default text editor

```
git config --global user.name "<your name>"
git config --global user.email <your email>
git config credential.username <your username>
```

Select your favorite text editor

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

or replace pico with your favorite editor This will produce the ~ /.gitconfig file from the previous section. Take a more in-depth look at git config on the git config page.

2.2. Saving changes to the repository: git add and git commit

Now that you have a repository initialized, you can **commit** file version changes to it.

When working in Git, or other version control systems, the concept of "saving" is a more nuanced process than saving in a word processor or other traditional file editing applications. The traditional software expression of "saving" is synonymous with the Git term "committing". A commit is the Git equivalent of a "save". Traditional saving should be thought of as a file system operation that is used to overwrite an existing file or write a new file. Alternatively, Git committing is an operation that acts upon a collection of files and directories.

The commands: git add and git commit are all used in combination to save a snapshot of a Git project's current state. Git has an additional saving mechanism called 'the stash'. The stash is an ephemeral storage area for changes that are not ready to be committed. We will get back to this later.

The git add and git commit commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand, regardless of their team's collaboration model. They are the means to record versions of a project into the repository's history.

Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory.

When you're ready to save a copy of the current state of the project, you stage changes with *git add*.

After you're happy with the staged snapshot, you commit it to the project history with *git commit*. The *git reset* command (we will see it later) is used to undo a commit or staged snapshot.

Git commit: The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as "safe" versions of a project—Git will never change them unless you explicitly ask it to. Prior to the execution of git commit, The git add command is used to promote or 'stage' changes to the project that will be stored in a commit. These two commands git commit and git add are two of the most frequently used.

At a high-level, Git can be thought of as a timeline management utility. Commits are the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project. Commits are created with the git commit command to capture the state of a project at that point in time. **Git Snapshots are always committed to the local repository**. **Git doesn't force you to interact with the central repository until you're ready**. Just as the staging area is a buffer between the working directory and the project history, each developer's local repository is a buffer between their contributions and the central repository.

This lets developers work in an isolated environment, deferring integration until they're at a convenient point to merge with other users. While isolation and deferred integration are individually beneficial, it is in a team's best interest to integrate frequently and in small units. We will get back to this later in the document.

Git records the entire contents of each file in every commit. This makes many Git operations fast, since a particular version of a file doesn't have to be "assembled" from its diffs—the complete revision of each file is immediately available from Git's internal database.

git add: The *git add* command adds a change in the working directory to the ***staging area***. It tells Git that you want to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run git commit.

In conjunction with these commands, you'll also want to use git status to view the state of the working directory and the staging area.

The staging area is one of Git's more unique features, and it can take some time to wrap your head around it.... It helps to think of it as a buffer between the working directory and the project history. **Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history**. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it's important to create atomic commits so that it's easy to track down bugs and revert changes with minimal impact on the rest of the project.

We will now add *app.js* to the repository staging area and then create a new commit with a message describing what work was done in the commit

Do:

- Change directory to our *gitlesson* folder if you are not there already
- `git add app.js`: system does not say anything
- `git status`: app.js is now among the changes to be committed, as a new file
- `git commit -m "add port constant"`: system responds with (your output may differ slightly)

```
[master (root-commit) 7bba746] add port constant
 1 file changed, 1 insertion(+)
 create mode 100644 app.js
```

notice that git again tells us what changed since the previous commit (which was empty, in fact it tells us this is a **root commit**).

now we have a commit. this version is in our history and we can always revert back to this point.

After executing this example, your repo will now app.js added to the history **and will track future updates to the file**.

now, let's make a change and commit again.

`pico app.js` (or use whatever text editor you like) and add a line, for example insert

```
const http = require('http');
```

- `git status`: (always good to do git status, never hurts). see our changes are NOT staged for commit. if you commit now, nothing will be committed.
- `git add app.js`
- `git status`: now change is to be committed
- `git commit -m "add http library"`: system adds to commit
- `git status`
- `git log --oneline`

8bca9ed (**HEAD -> master**) add http library

7bba746 add port constant

2.2.1. Common options

git commit

Commit the staged snapshot. This will launch a text editor prompting you for a commit message. After you've entered a message, save the file and close the editor to create the actual commit.

`git commit -a`

Commit a snapshot of all changes in the working directory. This only includes modifications to tracked files (those that have been added with git add at some point in their history).

```
git commit -m "commit message"
```

A shortcut command that immediately creates a commit with a passed commit message. By default, git commit will open up the locally configured text editor, and prompt for a commit message to be entered. Passing the -m option will forgo the text editor prompt in-favor of an inline message.

```
git commit -am "commit message"
```

A power user shortcut command that combines the -a and -m options. This combination immediately creates a commit of all the staged changes and takes an inline commit message.

```
git commit --amend
```

This option adds another level of functionality to the commit command. Passing this option will modify the last commit. Instead of creating a new commit, staged changes will be added to the previous commit. This command will open up the system's configured text editor and prompt to change the previously specified commit message. we will get back to this later in this document

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what's been changed. For example:

```
Change the message displayed by hello.py

- Update the sayHello() function to output the user's name
- Change the sayGoodbye() function to a friendlier message
```

It is a common practice to use the first line of the commit message as a subject line, similar to an email. The rest of the log message is considered the body and used to communicate details of the commit change set. Note that many developers also like to use the present tense in their commit messages. This makes them read more like actions on the repository, which makes many of the history-rewriting operations more intuitive.

TRY IT YOURSELF: create a new file (e.g., a readme.md), write whatever you want in it, and add it to the next commit.

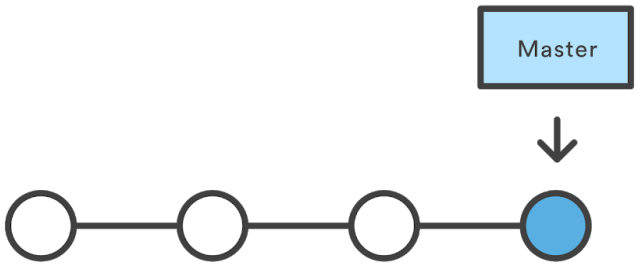
2.3. Navigating the commit history

In git, a **branch** is a pointer to a commit, and it has a readable name. When you start a git repo, git creates a default branch called **master.** Git also has a pointer called HEAD, which points to the current version of commit. Initially HEAD points to master.

When you commit, git not only creates a new commit but advances the branch pointer to point to the new commit. So, after a commit on the master branch, git advances the master pointer to the new commit, while HEAD keeps pointing to master.

you can use **git checkout** to revert back your working directory to a previous commit. Internally, the git checkout command simply updates the HEAD (a pointer to the current version of commit) of your git repository to point to either the specified **branch** or commit (a branch is nothing more than, and HEAD points to master. (and HEAD keeps pointing to master).

When HEAD points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.



- `git log --oneline`: **what is this HEAD and master thing? discuss**
- `git checkout {previous commit ID}`
- `more app.js`: see we reverted the file to the previous state
- `ls-la`: the entire folder is reverted (!)
- `git checkout master`: revert back to latests state, identified in this case by the label master

Discuss: Differences between file save, from google doc style versioning and saving, and git?

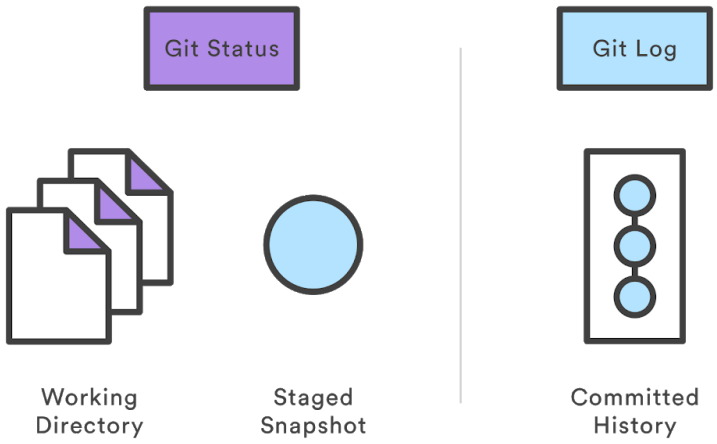
Discuss: when to commit?

TRY IT YOURSELF: go back to the INITIAL commit, see the content of app.js, then revert back to the latest commit

TRY IT YOURSELF: go back to the INITIAL commit, see the content of app.js, then revert back to the latest commit but instead of reverting back with `git checkout master`, try to checkout the latest commit. What's the difference?

2.3.1. Git status vs git log

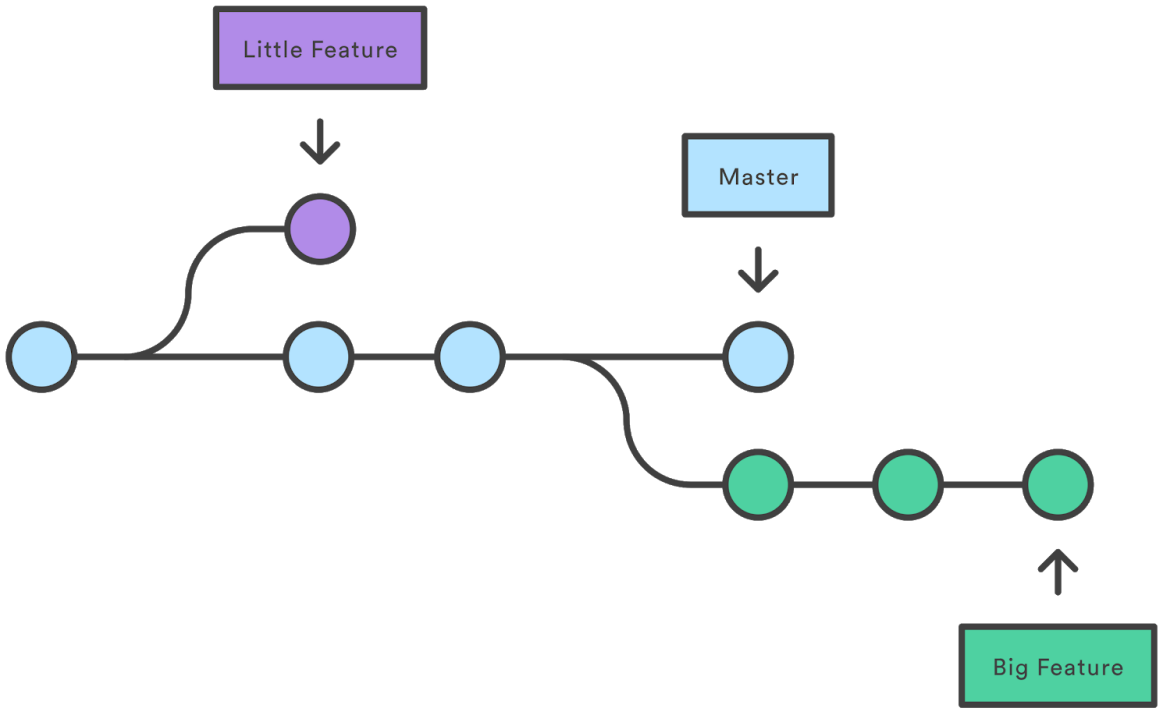
The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.



3. Branching and merging

Branching is a feature available in most modern version control systems. In Git, branches are a part of your everyday development process. **Git branches are effectively a pointer to a snapshot of your changes**.

When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.



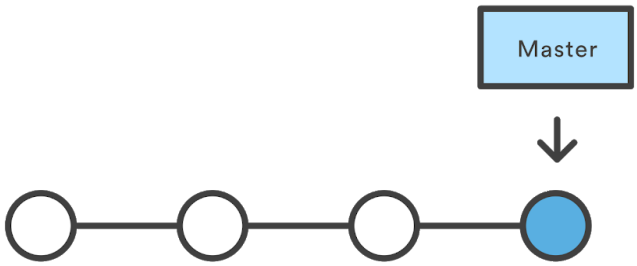
The diagram above visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. **By developing them in branches, it's not only possible to work on both of them in parallel, but it also keeps the main master branch free from questionable code**. **Git stores a branch as a reference to a commit. In this sense, a branch represents the tip of a series of commits.**

A branch represents an independent line of development. **New commits are recorded in the history for the current branch**, which results in a fork in the history of the project.

The git branch command lets you create, list, rename, and delete branches. It doesn't let you switch between branches or put a forked history back together again. For this reason, git branch is tightly integrated with the git checkout and git merge commands.

3.1. Creating Branches

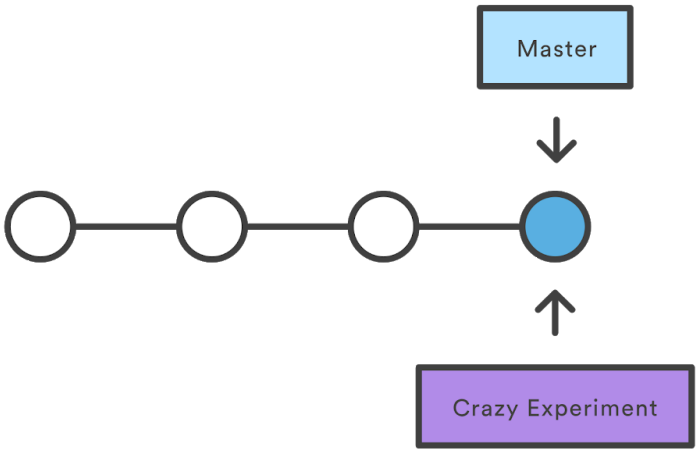
It's important to understand that branches are just pointers to commits. **When you create a branch, all Git needs to do is create a new pointer, it doesn't change the repository in any other way.** If you start with a repository that looks like this:



Then, **you create a branch** using the following command:

```
git branch crazy-experiment
```

The repository history remains unchanged. All you get is a new pointer to the current commit:



Note that this only creates the new branch. To start adding commits to it, you need to select it with git checkout, and then use the standard git add and git commit commands.

Let's try. go to one of your repos under version control, and create a new branch

- `git branch create-server`
- `git branch`: check on which branch you are (look for the asterisk)
- `git checkout create-server`
- `git branch`

- add the code below to app.js, add the file with git add and commit (git commit -m "add server code")

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

as usual do a git status after. Here is the output of my git log

```
commit c578b231f5309bdef67136aaf10e6c7b575ac38f (HEAD -> create-server)
Author: Fabio Casati <fabio.casati@gmail.com>
Date:   Wed Aug 29 15:29:04 2018 +0200

    add server code

commit 33a709f41934888fb145e6c3e006129bc06f0f71 (origin/master, master)
Author: Fab <32450818+fabiounitn@users.noreply.github.com>
Date:   Wed Aug 29 12:25:49 2018 +0200

    updated app
```

```
commit 77216ebd66bf5df16d1eceee167f5e89e3d7b4f8
Author: Fabio Casati <fabio.casati@gmail.com>
Date:   Wed Aug 29 12:15:17 2018 +0200

    a comment
```

now try this:

`git checkout master` (check the status and content of files, see that you don't see any longer the files as modified in the branch)

`git log`

`git checkout create-server` (check the status and content of files)

`git log`

Discuss: what is the meaning of (origin/master, master) and of (HEAD -> create-server) in the above log?

3.2. Deleting Branches

Once you've finished working on a branch and have merged it into the main code base, you're free to delete the branch without losing any history:

```
git branch -d create-server
```

however: if you are on that branch, you get an error (where will head point to??)

if you checkout master, you still get an error (!)

```
error: The branch 'create-server' is not fully merged.
If you are sure you want to delete it, run 'git branch -D
create-server'.
```

This protects you from losing access to that entire line of development. If you really want to delete the branch (e.g., it's a failed experiment), you can use the capital -D flag instead of -d (DON'T DO THIS FOR NOW):

This deletes the branch regardless of its status and without warnings, so use it judiciously.

3.3. Merging branches

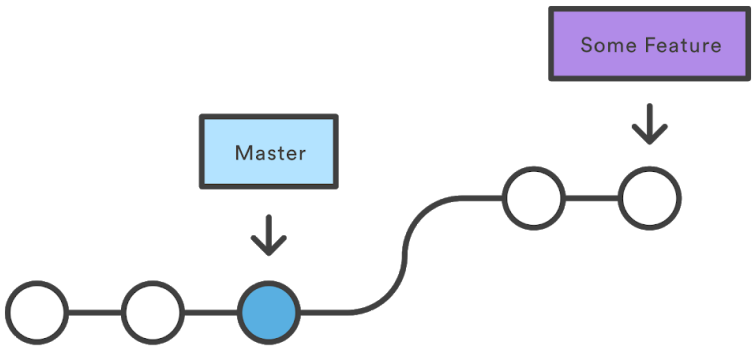
Once you've finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Merging is Git's way of putting a forked history back

together again. Depending on the structure of your repository, Git has distinct algorithms to accomplish this: a fast-forward merge or a 3-way merge.

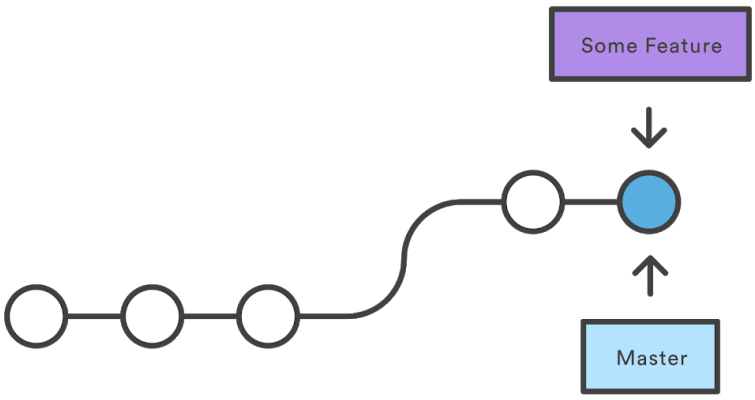
3.3.1. fast-forward merge

A fast-forward merge can occur when there is a linear path from the current branch tip to the target branch. Instead of "actually" merging the branches, all Git has to do to integrate the histories is move (i.e., "fast forward") the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the following:

Before Merging



After a Fast-Forward Merge



Example

The `git merge` command lets you take the independent lines of development created by git branch and integrate them into a single branch. For example, our main branch is called master, we have created the app server code in the create-server branch, and now we want to incorporate these changes into master. Note that all of the commands presented below **merge into the current branch**. **The current branch will be updated to reflect the merge, but the source branch will be completely unaffected**. Again, this means that git merge is often used in conjunction with git checkout for selecting the current branch and git branch -d for deleting the obsolete target branch.

to merge our branch into master, we therefore do:

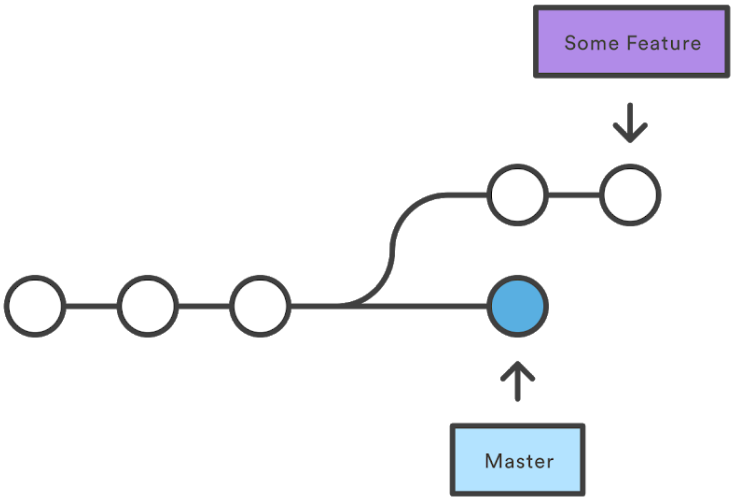
- `git checkout master`
- `git log --oneline` (not needed, but we do this to observe where head points to)
- `git merge create-server` (git will tell you that it has run a fast fwd merge)
- `git log --oneline` (see where head points to. git only moved head ahead, without adding a new commit to denote that the merge took place)
- `git log --graph --decorate --oneline`

running `git merge --no-ff <branch>` merge the specified branch into the current branch, but always generate a *merge commit* (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository. More on this later.

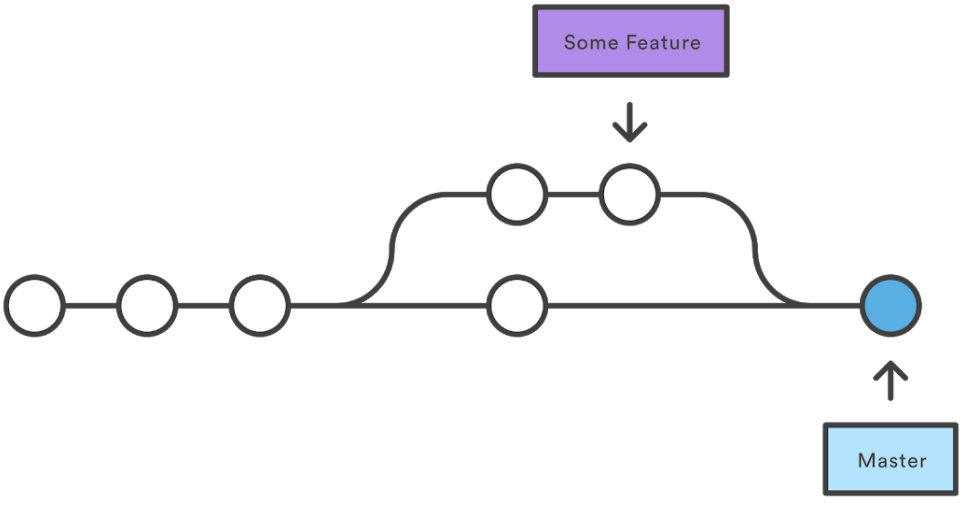
3.3.2. 3-way merge

A fast-forward merge is not possible if the branches have diverged. When there is no linear path to the target branch, Git has no choice but to combine them via a 3-way merge. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

Before Merging



After a 3-way Merge



While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through rebasing - we will see rebasing later) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

Example

we'll do the following:

- create a branch
- modify app.js on master
- modify readme on branch
- merge

let's try it:

```
git branch newport
git branch
pico readme.md (write something into this file and save)
git add .
git commit -m "add readme"
git log --oneline (notice where is head and where is master and where is branch)
git checkout newport
ls (notice readme is not there)
pico app.js - edit app to change port
git add .
git commit -m "change port to 3001"
git log --oneline
git checkout master
git merge newport - editor will open for commit message. you can just save if you want
git log --graph --decorate --oneline
```

try it yourself - create a new branch, make changes, switch to master, make changes, merge the branch

3.4. Resolving Merge Conflicts

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When you encounter a merge conflict, running the git status command

shows you which files need to be resolved. For example, if both branches modified the same section of hello.py, you would see something like the following:

```
# On branch master
# Unmerged paths:
# (use "git add/rm ..." as appropriate to mark resolution)
#
# both modified: hello.py
#
```

Then, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit. It's the exact same process as committing an ordinary snapshot, which means it's easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It's not possible to have conflicting changes in a fast-forward merge.

For most workflows, new feature would be a much larger feature that took a longer time to develop, which would be why new commits would appear on master in the meantime. If your feature branch was actually as small as the one in the above example, you would probably be better off rebasing it onto master and doing a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history. We will discuss rebasing later, don't worry if you don't know what it is for now.

let's try again, this time we will create a merge conflict

git branch 3002

git branch 8000

git branch

git checkout 3002

[edit the port number to 3002, add and commit]

git checkout 8000

[edit the port number to 8000, add and commit]

now let's merge 3002 into master -> do it yourself

git log --oneline

git checkout 8000

git log --oneline (see the different history)

git checkout master

git merge 8000

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

git status (never hurts)

let's look at app.js.... pico app.js

resolve conflicts

git add app.js

git commit

```
git log --graph --decorate --oneline
```

3.5. A note on switching branches

TRY IT YOURSELF

try to make changes to your working files, and then switch branch before committing..... what happens?

4. Collaborating and syncing with git

4.1. Cloning an existing repository: git clone

If a project has already been set up in a central repository, the **clone** command is the most common way for users to obtain a local development clone. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all version control operations are managed through their local repository.

let's do it

- create an account on github.com
- create a new repository. leave the checkbox about readme unchecked.
- get OUT of the previous repo you were working on.
- copy the repo url and on your terminal: type `git clone <repourl>`: notice a new folder has been created. if you go inside, you will find out this repo is now under version control**. HOW DO you find it out? discuss**

When executed, the latest version of the remote repo files on the master branch will be pulled down and added to a new folder. The new folder will be named after the REPONAME. The folder will contain the full history of the remote repository (in this case it is empty) and a newly created master branch.

4.2. Pushing and pulling changes

Let's make changes and update the central repository so that other people may download our changes

TRY IT YOURSELF

- create a file, add to the staging area and commit it
- `git push` (may ask you to enter github username and pwd)
- go on github.com and see the changes

TRY IT YOURSELF

- on github.com, click on the file, click on the pen to edit, enter a change, add a commit message at the bottom and commit. leave the other default options as is
- on your terminal, type `git pull`. now look at your file (`more <filename>` and see changes)

So that's it for the very basic usage of git, for versioning and collaboration. There is a LOT more of course, and what you saw here it's not enough to use it in any real project, but with this you get the basic idea.

4.3. Git remote

Git is designed to give each developer an entirely isolated development environment. This means that information is not automatically passed back and forth between repositories. Instead, developers need to manually pull upstream commits into their local repository or manually push their local commits back up to the central repository. The git remote command is really just an easier way to pass URLs to these "sharing" commands.

When you clone a repository with *git clone*, it automatically creates a remote connection called *origin* pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most Git-based projects call their central repository origin.

The `git remote` command lets you create, view, and delete connections to other repositories. Remote connections are more like bookmarks rather than direct links into other repositories. Instead of providing real-time access to another repository, they serve as convenient names that can be used to reference a not-so-convenient URL.

The git remote command is essentially an interface for managing a list of remote entries that are stored in the repository's ./.git/config file. The following commands are used to view the current state of the remote list.

Try:

```
git remote
git remote -v
```

The git remote command is also a convenience or 'helper' method for modifying a repo's ./.git/config file. The commands presented below let you manage connections with other repositories. The following commands will modify the repo's /.git/config file. The result of the following commands can also be achieved by directly editing the ./.git/config file with a text editor.

```
git remote add  <name> <url>
```

Create a new connection to a remote repository. After adding a remote, you'll be able to use <name> as a convenient shortcut for <url> in other Git commands.

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

Remove the connection to the remote repository called <name>.

```
git remote rename  <old> <new>
```

Rename a remote connection from <old-name> to <new-name>.

to inspect a remote you type

```
git remote show <name>
```

Try and discuss:

git remote -v

4.3.1. Bare vs. cloned repositories

When you clone a repository with git clone, it automatically creates a remote connection called origin pointing back to the cloned repository. This is useful for developers creating a local copy of a central repository, since it provides an easy way to pull upstream changes or publish local commits. This behavior is also why most Git-based projects call their central repository origin.

If you used git init to make a fresh repo, you'll have no remote repo to push changes to. **A common pattern when initializing a new repo is to go to a hosted Git service and create a repo there. The service will provide a Git URL that you can then add to your local Git repository and git push to the hosted repo. Once you have created a remote repo with your service of choice you will need to update your local repo with a mapping.**

In addition to origin, it's often convenient to have a connection to your teammates' repositories. For example, if your co-worker, John, maintained a publicly accessible repository on dev.example.com/john.git, you could add a connection as follows:

```
git remote add john http://dev.example.com/john.git
```

Having this kind of access to individual developers' repositories makes it possible to collaborate outside of the central repository. This can be very useful for small teams working on a large project.

4.4. Creating remote branches

If you create a branch, it will exist only in your local repo. if you go on github, you don't see it. if you run git push origin <branch> then you see it also on the server. try this, and check on github:

```
git push origin create-server
```

4.5. More on Fetching and Pulling

4.5.1. Fetch

Once a remote record has been configured through the use of the git remote command, the remote name can be passed as an argument to other Git commands to communicate with the remote repo. **Both git fetch, and git pull can be used to read from a remote repository**. Both commands have different operations that are explained in further depth on their respective links.

The git fetch command downloads commits and files from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on. It **doesn't force you to actually merge the changes into your repository**. Git isolates fetched content as a from existing local content, it has absolutely no effect on your local development work.

With fetch you can download a remote branch, and the commits from that branch. Remote branches are just like local branches, except they map to commits from somebody else's repository. Remote branches are prefixed by the remote they belong to so that you don't mix them up with local branches.

Fetched content has to be explicitly checked out using the git checkout command. This makes fetching a safe way to review commits before integrating them with your local repository.

try it yourself

```
git fetch origin master
git branch -a
```

You can inspect remote branches with the usual git checkout and git log commands. **If you approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge.**

`git fetch --all` is a power move which fetches all registered remotes and their branches.

Discuss: notice the detached head warning if you checkout a remote branch, eg checkout origin/master. why? how to manage it?

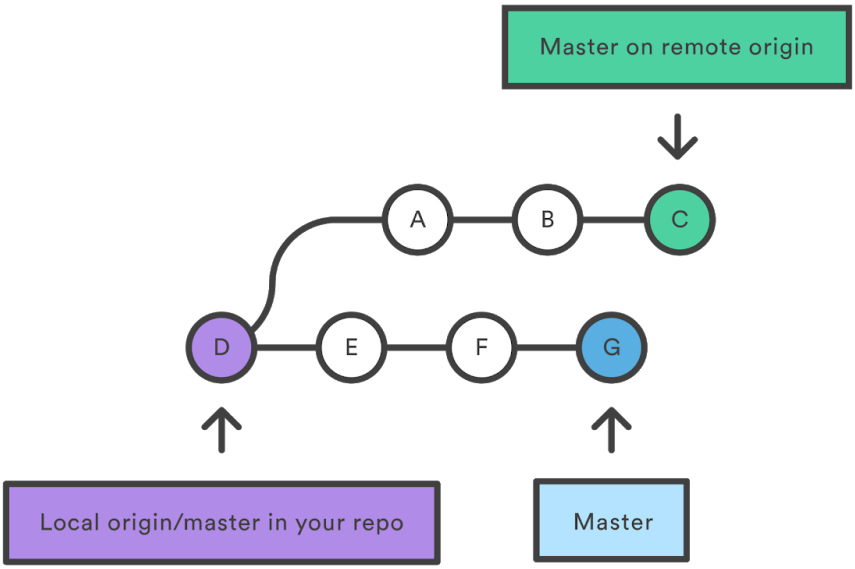
if you want to merge the changes from the remote branch into your own, you use git merge as usual, eg

```
git merge origin/master
```

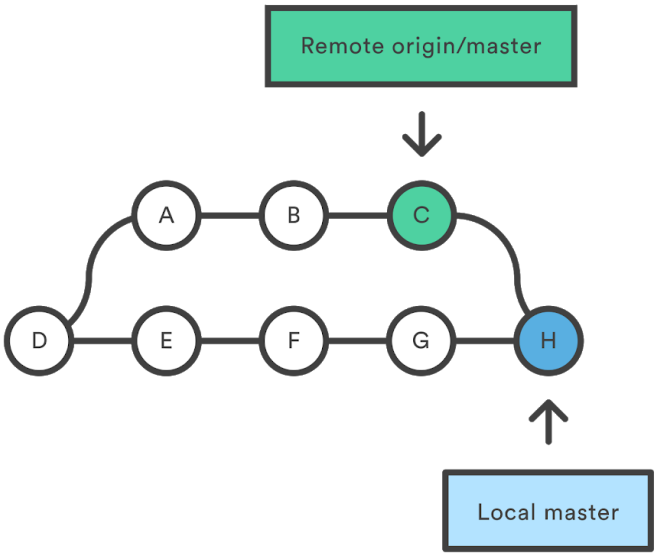
4.5.2. Pull

The git pull command first runs git fetch which downloads content from the specified remote repository. Then a git merge is executed to merge the remote content refs and heads into a new local merge commit.

To better demonstrate the pull and merging process let us consider the following example. Assume we have a repository with a master branch and a remote origin.



In this scenario, git pull will download all the changes from the point where the local and master diverged. In this example, that point is E. git pull will fetch the diverged remote commits which are A-B-C. The pull process will then create a new local merge commit containing the content of the new diverged remote commits.



In the above diagram, we can see the new commit H. This commit is a new merge commit that contains the contents of remote A-B-C commits and has a combined log message. This example is one of a few git pull merging strategies.

4.6. Pushing

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. **It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches.** Pushing has the potential to overwrite changes, caution should be taken when pushing.

to push changes from local to remote, you type

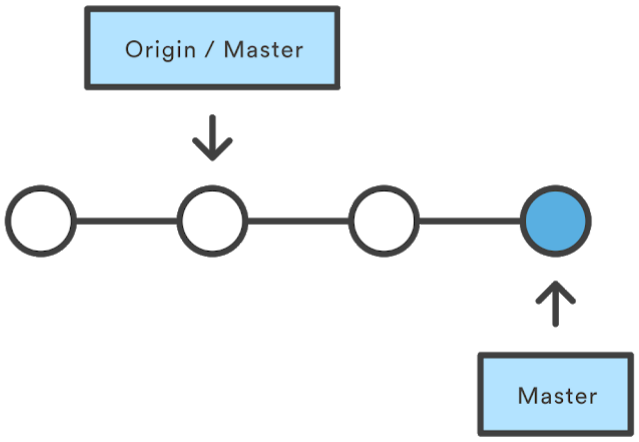
git push <remote> <branch>

Push the specified branch to <remote>, along with all of the necessary commits and internal objects. This creates a local branch in the destination repository.

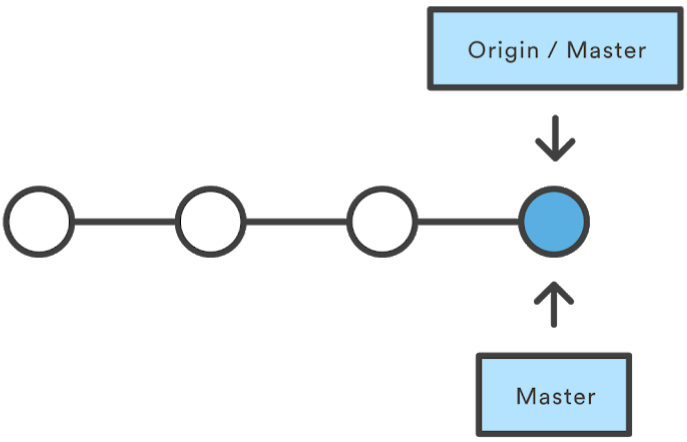
try: `git push origin master.` then go on github and see that the new commit is there.

if you now run `git remote show origin` it will show that master is up to date

Before Pushing



After Pushing



To prevent you from overwriting commits, Git won't let you push when it results in a non-fast-forward merge in the destination repository. If the remote history has diverged from

your history, you need to pull the remote branch and merge it into your local one, then try pushing again.

`git push --force` forces the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you're absolutely sure you know what you're doing. The --force flag makes the remote repository's branch match your local one, deleting any upstream changes that may have occurred since you last pulled. The only time you should ever need to force push is when you realize that the commits you just shared were not quite right and you fixed them with a git commit --amend or an interactive rebase. However, you must be absolutely certain that none of your teammates have pulled those commits before using the --force option.

git push --all

Push all of your local branches to the specified remote.

```
git push -u <remote_name> <local_branch_name>
```

The -u option does the following: For every branch that is up to date or successfully pushed, add upstream (tracking) reference, used by argument-less git-pull and other commands. So, after pushing your local branch with -u option, this local branch will be automatically linked with remote branch, and you can use `git pull` without any arguments.

To delete a remote branch execute the following:

(DON'T DO THIS FOR NOW) `git push origin --delete create-server`

try it yourself

create a new branch locally, switch to it, make a change, commit, and push the branch and the new commit on the server

4.7. Common ways to get stuck in pulling, pushing, merging, branching….

let's see examples of pushing and pulling in action.

create a repo, create two branches on it so that you have two branches on the server. Then clone this repo into a new folder.

so if you look at this repo on github, it has 2 branches. if you look at the cloned one (`git branch`), you see only master.

if you do `git branch -a` you also see the remote branches

and if you checkout explicitly create-server, then you also have that branch in sight when you do git branch.

now, let's checkout master again. and do the following: make one change on master in local, and commit, and make one change on master on github (for example via github interface) and commit

taake also a look in github at the insights/network tab, to see what it shows.

now, a `git status` **believes** that my local implem is ahead of origin master. if i do `git fetch origin master` and then `git status`, git tells me i have diverged.

so, i need to merge, or to pull (in case somebody else has pushed something in the meantime).

git merge origin/master

after the merge, i will be AHEAD of origin/ master by TWO commits (though it's not really true, one of them was kind of in parallel)

Now do a git push and then go see on the server: you'll notice a branching and merging without a new branch label as it is always about master.

notice also how annoying this is: if everybody does this, then on master we see lots of branching and merging, while ideally, a posteriori, after we finished development, we would like a nice linear history with few commits. We will get back to this later.

Discuss:

switching branches with uncommitted files

pushing and pulling from repo and branch with no common root (discuss scenario where we point at more than one repo, or we have an existing repo but we want to get a master branch without common root)

https://stackoverflow.com/questions/1125968/how-do-i-force-git-pull-to-overwrite-local-files

5. Ignoring files - GitIgnore

Git sees every file in your working copy as one of three things:

1. tracked - a file which has been previously staged or committed;
2. untracked - a file which has not been staged or committed; or
3. ignored - a file which Git has been **explicitly** told to ignore.

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Some common examples are your IDE local files or cached dependencies.

Ignored files are tracked in a special file named .gitignore that is checked in at the root of your repository. **There is no explicit git ignore command**: **instead the .gitignore file must be edited and committed by hand when you have new files that you wish to ignore.**

.gitignore files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

A an example your .gitignore file may be

```
node_modules
build
npm-debug.log
.env
.DS_Store
```

.gitignore uses globbing patterns to match against file names.In addition to these characters, you can use # to include comments in your .gitignore file:

```
# ignore all logs
*.log
```

Git ignore rules are usually defined in a .gitignore file at the root of your repository. However, you can choose to define multiple .gitignore files in different directories in your repository. Each pattern in a particular .gitignore file is tested relative to the directory containing that file. However the convention, and simplest approach, is to define a single .gitignore file in the root. As your .gitignore file is checked in, it is versioned like any other file in your repository and shared with your teammates when you push.

If you want to ignore a file that you've committed in the past, you'll need to delete the file from your repository and then add a .gitignore rule for it. Using the --cached option with git rm means that the file will be deleted from your repository, but will remain in your working directory as an ignored file.

```
echo debug.log >> .gitignore
git rm --cached debug.log
git commit -m "Start ignoring debug.log"
```

You can omit the --cached option if you want to delete the file from both the repository and your local file system.

6. Stashing

git stash temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

assume we have this output from git status

Changes **to be committed**:

(use "git reset HEAD <file>..." to unstage)

modified: readme.md

Changes **not staged for commit**:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: app.js

Untracked files:

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

test

now let's move to a different branch....

git checkout 3002

error: Your local changes to the following files would be overwritten by checkout:

app.js

Please commit your changes or stash them before you switch branches.

Aborting

now try

git stash

git status

At this point you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply your stash when you're ready.

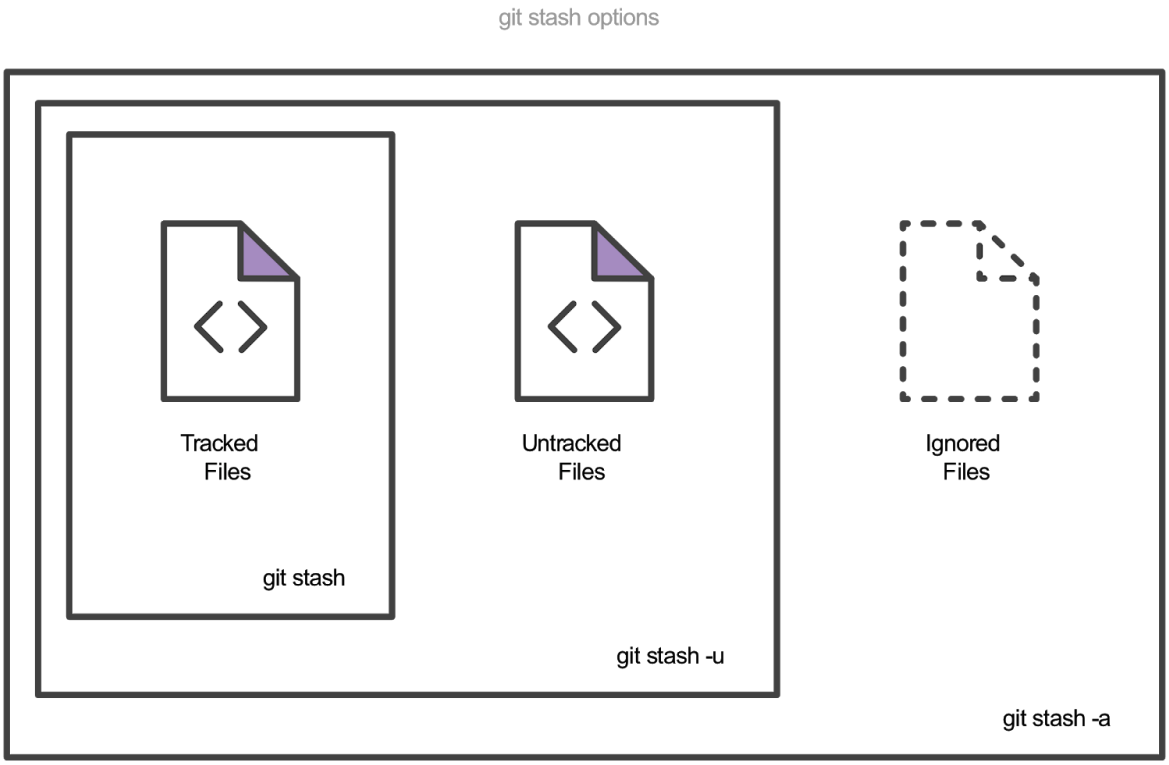
Note that the stash is local to your Git repository; stashes are not transferred to the server when you push.

You can reapply previously stashed changes with **git stash pop**

Now that you know the basics of stashing, there is one caveat with git stash you need to be aware of: by default Git won't stash changes made to untracked or ignored files.

Adding the -u option (or --include-untracked) tells git stash to also stash your untracked files.

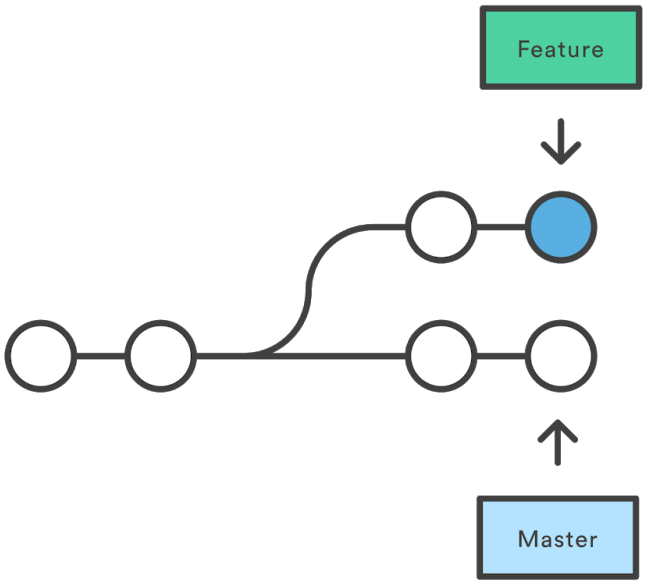
You can include changes to ignored files as well by passing the -a option (or --all) when running git stash.

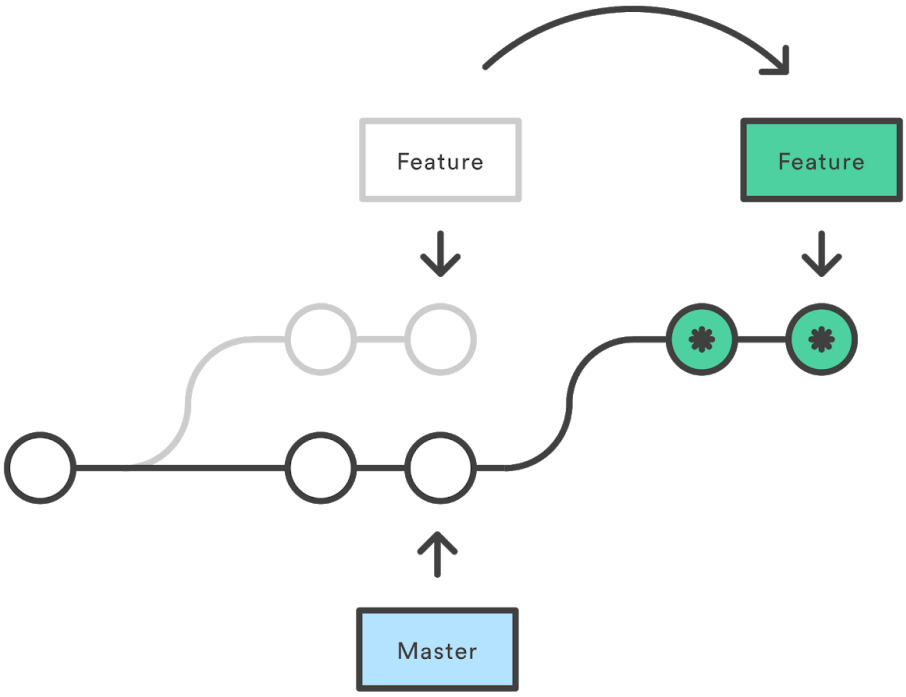


7. Rebasing

Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The general process can be visualized as the following.

You have two options for integrating your feature into the master branch: merging directly or rebasing and then merging. The former option results in a 3-way merge and a merge commit, while the latter results in a fast-forward merge and a perfectly linear history. The following diagram demonstrates how rebasing onto the master branch facilitates a fast-forward merge.





Brand New Commits

From a content perspective, rebasing is changing the base of your branch from one commit to another making it appear as if you'd created your branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. **It's very important to understand that even though the branch looks the same, it's composed of entirely new commits.**

Merge is always a forward moving change record. Alternatively, rebase has powerful history rewriting features. For a detailed look at Merge vs. Rebase, visit our Merging vs Rebasing guide. **Rebase itself has 2 main modes: "manual" and "interactive" mode.** We will cover the different Rebase modes in more detail below.

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature branch. You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch. This gives the later benefit of a clean merge of your feature branch back into the master branch. Why do we want to maintain a "clean history"? The benefits of having a clean history become tangible when performing Git operations to investigate the introduction of a regression.

Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with Git merge results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, “I want to base my changes on what everybody has already done.”

TRY IT YOURSELF

checkout master in an existing repo

create a branch (say, somefeature)

make a change and commit (on master)

checkout somefeature

make a change on somefeature

```
git rebase master


git log --graph --decorate --oneline
```

TRY IT YOURSELF

try again the above, but this time make the changes on master and branch to be conflicting.

TRY IT YOURSELF

compare these two workflows:

- you have a divergent master branch from origin/master. try the two approaches, pull/push versus fetch/rebase/push, see the difference on the server (insights/network on github)

Don't rebase public history

You should never rebase commits once they've been pushed to a public repository. The rebase would replace the old commits with new ones and it would look like that part of your project history abruptly vanished.

7.1. Git Rebase Interactive

Git rebase in standard mode will automatically take the commits in your current working branch and apply them to the head of the passed branch. This automatically rebases the current branch onto <base>, which can be any kind of commit reference (for example an ID, a branch name, a tag).

Running git rebase with the -i flag (`git rebase -i`) begins an **interactive** rebasing session. **Instead of blindly moving all of the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits.**

This opens an editor where you can enter commands (described below) for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves.

Once you've specified commands for each commit in the rebase, Git will begin playing back commits applying the rebase commands. The rebasing edit commands are as follows:

```
pick 2231360 some old commit
pick ee2adc2 Adds new feature




# Rebase 2cf755d..ee2adc2 onto 2cf755d (9 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
```

Most developers like to use an interactive rebase to polish a feature branch before merging it into the main code base. This gives them the opportunity to squash insignificant commits, delete obsolete ones, and make sure everything else is in order before committing to the “official” project history. **To everybody else, it will look like the entire feature was developed in a single series of well-planned commits.**

The real power of interactive rebasing can be seen in the history of the resulting master branch. To everybody else, it looks like you're a brilliant developer who implemented the new feature with the perfect amount of commits the first time around. This is how interactive rebasing can keep a project's history clean and meaningful.

One caveat to consider when working with Git Rebase is merge conflicts may become more frequent during a rebase workflow. This occurs if you have a long-lived branch that has strayed

from master. Eventually you will want to rebase against master and at that time it may contain many new commits that your branch changes may conflict with. **This is easily remedied by rebasing your branch frequently against master, and making more frequent commits.** The --continue and --abort command line arguments can be passed to git rebase to advance or reset the the rebase when dealing with conflicts.

A more serious rebase caveat is lost commits from interactive history rewriting. Running rebase in interactive mode and executing subcommands like squash or drop will remove commits from your branch immediate log. At first glance this can appear as though the commits are permanently gone. Using git reflog these commits can be restored and the entire rebase can be undone. For more info on using git reflog to find lost commits, visit the Git reflog documentation page (*learning reflog is optional*).

Git Rebase itself is not seriously dangerous. The real danger cases arise when executing history rewriting interactive rebases and force pushing the results to a remote branch that's shared by other users. This is a pattern that should be avoided as it has the capability to overwrite other remote users' work when they pull.

TRY IT YOURSELF

checkout master in an existing repo

create a branch (say, somefeature)

make a change and commit (on master)

checkout somefeature

make a change on somefeature (conflicting with master) and commit

make another change on somefeature and commit

make a third change on somefeature and commit

then rebase, squashing the commit into one

```
git log --graph --decorate --oneline
```

8. Tagging and blaming

8.1. Tagging

Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1.0.1). A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.

To create a new tag execute the following command:

git tag <tagname>

Replace <tagname> with a semantic identifier to the state of the repo at the time the tag is being created. A common pattern is to use version numbers like git tag v1.4. Git supports two different types of tags, annotated and lightweight tags. The previous example created a *lightweight* tag.

You can also create an Annotated tag, store extra meta data such as: the tagger name, email, and date. This is important data for a public release. Lightweight tags are essentially 'bookmarks' to a commit, they are just a name and a pointer to a commit, useful for creating quick links to relevant commits. **A best practice is to consider Annotated tags as public, and Lightweight tags as private**.

git tag -a v1.4

Executing this command will create a new annotated tag identified with v1.4. The command will then open up the configured default text editor to prompt for further meta data input.

git tag -a v1.4 -m "my version 1.4"

Executing this command is similar to the previous invocation, however, this version of the command is passed the -m option and a message. This is a convenience method similar to git commit -m that will immediately create a new tag and forgo opening the local text editor in favor of saving the message passed in with the -m option.

To list stored tags in a repo execute the following:

```
git tag
```

By default, git tag will create a tag on the commit that HEAD is referencing. Alternatively git tag can be passed as a ref to a specific commit.

git tag -a v1.2 15027957951b64cf874c3557a0f3547bd83b3ff6

Executing the above git tag invocation will create a new annotated commit identified as v1.2 for the commit we selected in the previous git log example.

You can view the state of a repo at a tag by using the git checkout command.

git checkout v1.4

The above command will checkout the v1.4 tag. This puts the repo in a detached HEAD state. This means any changes made will not update the tag. They will create a new detached commit. This new detached commit will not be part of any branch and will only be reachable directly by the commits SHA hash. Therefore it is a best practice to create a new branch anytime you're making changes in a detached HEAD state.

Deleting tags is a straightforward operation. Passing the -d option and a tag identifier to git tag will delete the identified tag.

Sharing tags is similar to pushing branches. By default, git push will not push tags. Tags have to be explicitly passed to git push.

git push origin v1.4

git push --tags

Tags are not automatically pushed. The --tags flag sends all of your local tags to the remote repository

8.2. blaming

The git blame command is a versatile troubleshooting utility that has extensive usage options. **The high-level function of git blame is the display of author metadata attached to specific committed lines in a file**. This is used to examine specific points of a file's history and get context as to who the last author was that modified the line. This is used to explore the history of specific code and answer questions about what, how, and why the code was added to a repository.

Git blame is often used with a GUI display. Online Git hosting sites like Bitbucket or github offer blame views which are UI wrappers to git blame. These views are referenced in collaborative discussions around pull requests and commits. **Find it out on github, select a file and then click on the blame button.**

9. Configuration & set up: git config (optional)

For more in-depth look at git remote, see the Git remote page.

In addition to configuring a remote repo URL, you may also need to set global Git configuration options such as username, or email. The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info, to preferences, to the behavior of a repository. Several common configuration options are listed below.

Git stores configuration options in three separate files, which lets you scope options to individual repositories (local), user (Global), or the entire system (system):

- Local: <repo>/.git/config – Repository-specific settings.
- Global: /.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
- System: $(prefix)/etc/gitconfig – System-wide settings.

Define the author name to be used for all commits in the current repository. Typically, you'll want to use the --global flag to set configuration options for the current user.

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

Define the author name to be used for all commits by the current user.

Adding the --local option or not passing a config level option at all, will set the user.name for the current local repository.

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

```
git config --system core.editor <editor>
```

Define the text editor used by commands like git commit for all users on the current machine. The <editor> argument should be the command that launches the desired editor (e.g., vi). This example introduces the --system option. The --system option will set the configuration for the entire system, meaning all users and repos on a machine. For more detailed information on configuration levels visit the git config page.

```
git config --global --edit
```

Open the global configuration file in a text editor for manual editing. An in-depth guide on how to configure a text editor for git to use can be found on the Git config page.

All configuration options are stored in plaintext files, so the git config command is really just a convenient command-line interface. Typically, you'll only need to configure a Git installation the

first time you start working on a new development machine, and for virtually all cases, you'll want to use the --global flag. One important exception is to override the author email address. You may wish to set your personal email address for personal and open source repositories, and your professional email address for work-related repositories.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

- <repo>/.git/config – Repository-specific settings.
- ~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
- $(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you'll see something like the following:

```
[user] name = John Smith email = john@example.com [alias] st = status co = checkout br = branch up = rebase ci = commit [core] editor = vim
```

You can manually edit these values to the exact same effect as git config.

The first thing you'll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

Tell Git who you are: git config

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

Select your favorite text editor

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

This will produce the ~ /.gitconfig file from the previous section. Take a more in-depth look at git config on the git config page.

10. Comparing changes: git diff

***Diffing* is a function that takes two input data sets and outputs the changes between them**. git diff is a multi-use Git command that when executed runs a diff function on Git data sources. These data sources can be commits, branches, files and more. This document will discuss common invocations of git diff and diffing work flow patterns. The git diff command is often used along with git status and git log to analyze the current state of a Git repo.

Changes since last commit

By default git diff will show you any uncommitted changes since the last commit.

```
git diff
```

Comparing files between two different commits

git diff can be passed Git refs to commits to diff. Some example refs are, HEAD, tags, and branch names. Every commit in Git has a commit ID which you can get when you execute GIT LOG. You can also pass this commit ID to git diff.

Comparing branches

Branches are compared like all other ref inputs to git diff

To compare a specific file across branches, pass in the path of the file as the third argument to git diff

```
git diff master new_branch ./diff_test.txt
```

11. Undoing Commits & Changes

Git does not have a traditional 'undo' system like those found in a word processing application. It will be beneficial to refrain from mapping Git operations to any traditional 'undo' mental model. Additionally, Git has its own nomenclature for 'undo' operations that it is best to leverage in a discussion. This nomenclature includes terms like reset, revert, checkout, clean, and more.

A fun metaphor is to think of Git as timeline management utility. Commits are snapshots of a point in time or points of interest along the timeline of a project's history. Additionally, multiple timelines can be managed through the use of branches. When 'undoing' in Git, you are usually moving back in time, or to another timeline where mistakes didn't happen.

11.1. Git Reset

The git reset command is a complex and versatile tool for undoing changes. It has three primary forms of invocation. These forms correspond to command line arguments --soft, --mixed, --hard. The three arguments each correspond to Git's three internal state management mechanism, The Commit Tree (HEAD), The Staging Index, and The Working Directory.

On the commit-level, **resetting is a way to move the tip of a branch to a different commit**. This can be used to remove commits from the current branch. **For example, the following command moves the hotfix branch backwards by two commits.**

git checkout hotfix

git reset HEAD~2

The two commits that were on the end of hotfix are now dangling, or orphaned commits. This means they will be deleted the next time Git performs a garbage collection. In other words, you're saying that you want to throw away these commits.

This usage of git reset is a simple way to undo changes that haven't been shared with anyone else. It's your go-to command when you've started working on a feature and find yourself thinking, "Oh crap, what am I doing? I should just start over."

In addition to moving the current branch, you can also get git reset to alter the staged snapshot and/or the working directory by passing it one of the following flags:

- **--soft – The staged snapshot and working directory are not altered in any way.**
- **--mixed – The staged snapshot is updated to match the specified commit, but the working directory is not affected. This is the default option.**
- **--hard – The staged snapshot and the working directory are both updated to match the specified commit.**

git reset

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages all files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

git reset --hard

Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the –hard flag tells Git to overwrite all changes in the working directory, too. Put another way: **this obliterates all uncommitted changes**, so make sure you really want to throw away your local developments before using it.

git reset <commit> (or branch)

Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.

git reset --hard <commit> (or branch)

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well.

11.2. Git Revert

The git revert command can be considered an 'undo' type command, however, it is not a traditional undo operation. Instead of removing the commit from the project history, it figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

Reverting should be used when you want to apply the inverse of a commit from your project history. This can be useful, for example, if you're tracking down a bug and find that it was introduced by a single commit. Instead of manually going in, fixing it, and committing a new snapshot, you can use git revert to automatically do all of this for you.

To demonstrate this lets create an example repo using the command line examples below:

```
mkdir git_revert_test
cd git_revert_test/
git init .
Initialized empty Git repository in /git_revert_test/.git/
touch demo_file
```

```
git add demo_file
git commit -am"initial commit"
[master (root-commit) 299b15f] initial commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 demo_file
echo "initial content" >> demo_file
git commit -am"add new content to demo file"
[master 3602d88] add new content to demo file
n 1 file changed, 1 insertion(+)
echo "prepended line content" >> demo_file
git commit -am"prepend content to demo file"
[master 86bb32e] prepend content to demo file
 1 file changed, 1 insertion(+)
git log --oneline
86bb32e prepend content to demo file
3602d88 add new content to demo file
299b15f initial commit
```

Here we have initialized a repo in a newly created directory named git_revert_test. We have made 3 commits to the repo in which we have added a file demo_file and modified its content twice. At the end of the repo setup procedure, we invoke git log to display the commit history, showing a total of 3 commits. With the repo in this state, we are ready to initiate a git revert.

```
more demo_file
git revert HEAD
[master b9cd081] Revert "prepend content to demo file"
1 file changed, 1 deletion(-)
```

Git revert expects a commit ref passed in and will not execute without one. Here we have passed in the HEAD ref. This will revert the latest commit. This is same behavior as if we reverted to commit 3602d8815dbfa78cd37cd4d189552764b5e96c58. Similar to a merge, a revert will create a new commit which will open up the configured system editor prompting for a new commit message. Once a commit message has been entered and saved Git will resume operation. We can now examine the state of the repo using git log and see that there is a new commit added to the previous log:

```
git log --oneline
more demo_file
```

```
1061e79 Revert "prepend content to demo file"
86bb32e prepend content to demo file
3602d88 add new content to demo file
299b15f initial commit
```

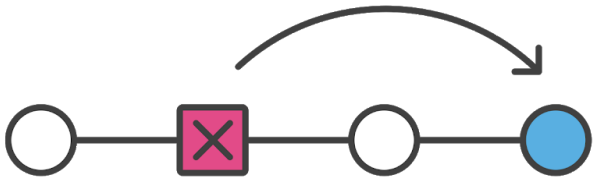
Note that the 3rd commit is still in the project history after the revert. Instead of deleting it, git revert added a new commit to undo its changes. As a result, the 2nd and 4th commits represent the exact same code base and the 3rd commit is still in our history just in case we want to go back to it down the road.

Passing the -n option will prevent git revert from creating a new commit that inverses the target commit. Instead of creating the new commit this option will add the inverse changes to the Staging Index and Working Directory. These are the other trees Git uses to manage state the state of the repository. For more info visit the git reset page.

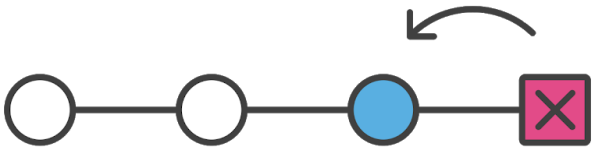
11.3. Resetting vs. reverting

It's important to understand that git revert undoes a single commit—it does not "revert" back to the previous state of a project by removing all subsequent commits. In Git, this is actually called a reset, not a revert.

Reverting



Resetting



Reverting has two important advantages over resetting. First, it doesn't change the project history, which makes it a "safe" operation for commits that have already been published to a shared repository.
Second, **git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backward from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits**. Needless to say, this is not an elegant undo solution.

11.4. Git RM

A common question when getting started with Git is "How do I tell Git not to track a file (or files) any more?" The git rm command is used to remove files from a Git repository. It can be thought of as the inverse of the git add command.

The git rm command can be used to remove individual files or a collection of files. The primary function of git rm is to remove tracked files from the Git index. **Additionally, git rm can be used to remove files from both the index and the working directory. There is no option to remove a file from only the working directory.** The files being operated on must be identical to the files in

the current HEAD. If there is a discrepancy between the HEAD version of a file and the staging index or working tree version, Git will block the removal. This block is a safety mechanism to prevent removal of in-progress changes.

The -f option is used to override the safety check that Git makes to ensure that the files in HEAD match the current content in the staging index and working directory.

The -r option is shorthand for 'recursive'. When operating in recursive mode git rm will remove a target directory and all the contents of that directory.

The --cached option specifies that the removal should happen only on the staging index. Working directory files will be left alone.

Executing git rm is not a permanent update. The command will update the staging index and the working directory. These changes will not be persisted until a new commit is created and the changes are added to the commit history. This means that the changes here can be "undone" using common Git commands.

git reset HEAD

A reset will revert the current staging index and working directory back to the HEAD commit. This will undo a git rm.

git checkout .

A checkout will have the same effect and restore the latest version of a file from HEAD.

The git rm command operates on the current branch only.

A Git repository will recognize when a regular shell rm command has been executed on a file it is tracking. It will update the working directory to reflect the removal. It will not update the staging index with the removal. An additional git add command will have to be executed on the removed file paths to add the changes to the staging index. The git rm command acts a shortcut in that it will update the working directory and the staging index with the removal.

TRY IT YOURSELF:

- a file from the working directory
- a file from the index
- a file from both

and see what happens, also see what happens as we commit. try also git log and navigate history back and forward to make sure you understand what has happened

11.5. Undoing a committed snapshot

There are technically several different strategies to 'undo' a commit. The following examples will assume we have a commit history that looks like:

```
git log --oneline
872fa7e Try something crazy
a1e8fb5 Make some important changes to hello.txt
435b61d Create hello.txt
9773e52 Initial import
```

We will focus on undoing the 872fa7e Try something crazy commit. Maybe things got a little too crazy.

11.5.1. How to undo a commit with git checkout

Using the git checkout command we can checkout the previous commit, a1e8fb5, putting the repository in a state before the crazy commit happened. Checking out a specific commit will put the repo in a "detached HEAD" state. This means you are no longer working on any branch. In a detached state, any new commits you make will be orphaned when you change branches back to an established branch. Orphaned commits are up for deletion by Git's garbage collector. The garbage collector runs on a configured interval and permanently destroys orphaned commits. To prevent orphaned commits from being garbage collected, we need to ensure we are on a branch.

From the detached HEAD state, we can execute `git checkout -b new_branch_without_crazy_commit`. This will create a new branch named new_branch_without_crazy_commitand switch to that state. The repo is now on a new history timeline in which the 872fa7e commit no longer exists. At this point, we can continue work on this new branch in which the 872fa7e commit no longer exists and consider it 'undone'. Unfortunately, if you need the previous branch, maybe it was your master branch, this undo strategy is not appropriate. Let's look at some other 'undo' strategies. For more information and examples review our in-depth git checkout discussion.

11.5.2. How to undo a public commit with git revert

Let's assume we are back to our original commit history example. The history that includes the 872fa7e commit. This time let's try a revert 'undo'. If we execute **git revert HEAD**, Git will create a new commit with the inverse of the last commit. This adds a new commit to the current branch history and now makes it look like:

```
git log --oneline
e2f9a78 Revert "Try something crazy"
872fa7e Try something crazy
a1e8fb5 Make some important changes to hello.txt
435b61d Create hello.txt
9773e52 Initial import
```

At this point, we have again technically 'undone' the 872fa7e commit. Although 872fa7e still exists in the history, the new e2f9a78 commit is an inverse of the changes in 872fa7e. Unlike our previous checkout strategy, we can continue using the same branch. This solution is a satisfactory undo. This is the ideal 'undo' method for working with public shared repositories. If

you have requirements of keeping a curated and minimal Git history this strategy may not be satisfactory.

11.5.3. How to undo a commit with git reset

For this undo strategy we will continue with our working example. git reset is an extensive command with multiple uses and functions. If we invoke **git reset --hard a1e8fb5** the commit history is reset to that specified commit. Examining the commit history with git log will now look like:

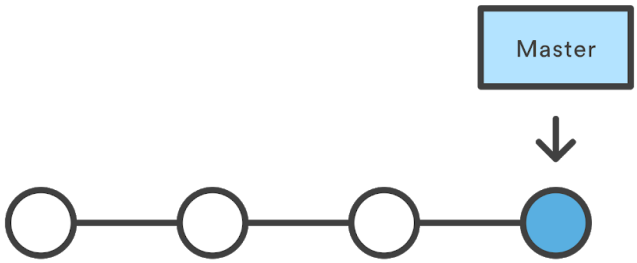
```
git log --oneline
a1e8fb5 Make some important changes to hello.txt
435b61d Create hello.txt
9773e52 Initial import
```

The log output shows the e2f9a78 and 872fa7e commits no longer exist in the commit history. At this point, we can continue working and creating new commits as if the 'crazy' commits never happened. This method of undoing changes has the cleanest effect on history. Doing a reset is great for local changes however it adds complications when working with a shared remote repository. **If we have a shared remote repository that has the 872fa7e commit pushed to it, and we try to git push a branch where we have reset the history, Git will catch this and throw an error. Git will assume that the branch being pushed is not up to date because of it's missing commits. In these scenarios, git revert should be the preferred undo method.**

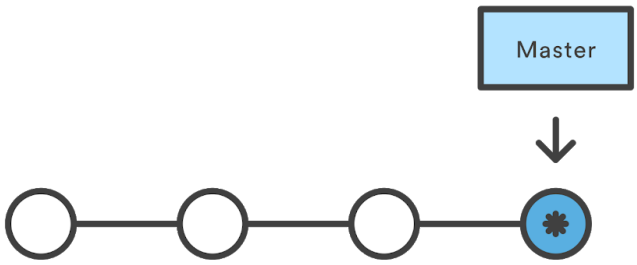
11.5.4. Changing the Last Commit: git commit --amend

The **git commit --amend** command is a convenient way to modify the most recent commit. **It lets you combine staged changes with the previous commit instead of creating an entirely new commit**. It can also be used to simply edit the previous commit message without changing its snapshot. But, **amending does not just alter the most recent commit, it replaces it entirely**, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit, which is visualized with an asterisk (*) in the diagram below.

Initial History



Amended History



Brand New Commits

There are a few common scenarios for using git commit --amend. We'll cover usage examples in the following sections.

```
git commit --amend
```

Let's say you just committed and you made a mistake in your commit log message. Running this command when there is nothing staged lets you edit the previous commit's message without altering its snapshot.

Premature commits happen all the time in the course of your everyday development. It's easy to forget to stage a file or to format your commit message the wrong way. The --amend flag is a convenient way to fix these minor mistakes.

The following example demonstrates a common scenario in Git-based development. Let's say we've edited a few files that we would like to commit in a single snapshot, but then we forget to add one of the files the first time around. Fixing the error is simply a matter of staging the other file and committing with the --amend flag:

```
# Edit hello.py and main.py
git add hello.py
git commit
# Realize you forgot to add the changes from main.py git add main.py
git commit --amend --no-edit
```

The --no-edit flag will allow you to make the amendment to your commit without changing its commit message. The resulting commit will replace the incomplete one, and it will look like we committed the changes to hello.py and main.py in a single snapshot.

Don’t amend public commits

Amended commits are actually entirely new commits and the previous commit will no longer be on your current branch. This has the same consequences as resetting a public snapshot. Avoid amending a commit that other developers have based their work on. This is a confusing situation for developers to be in and it’s complicated to recover from.

11.5.5. Changing older or multiple commits

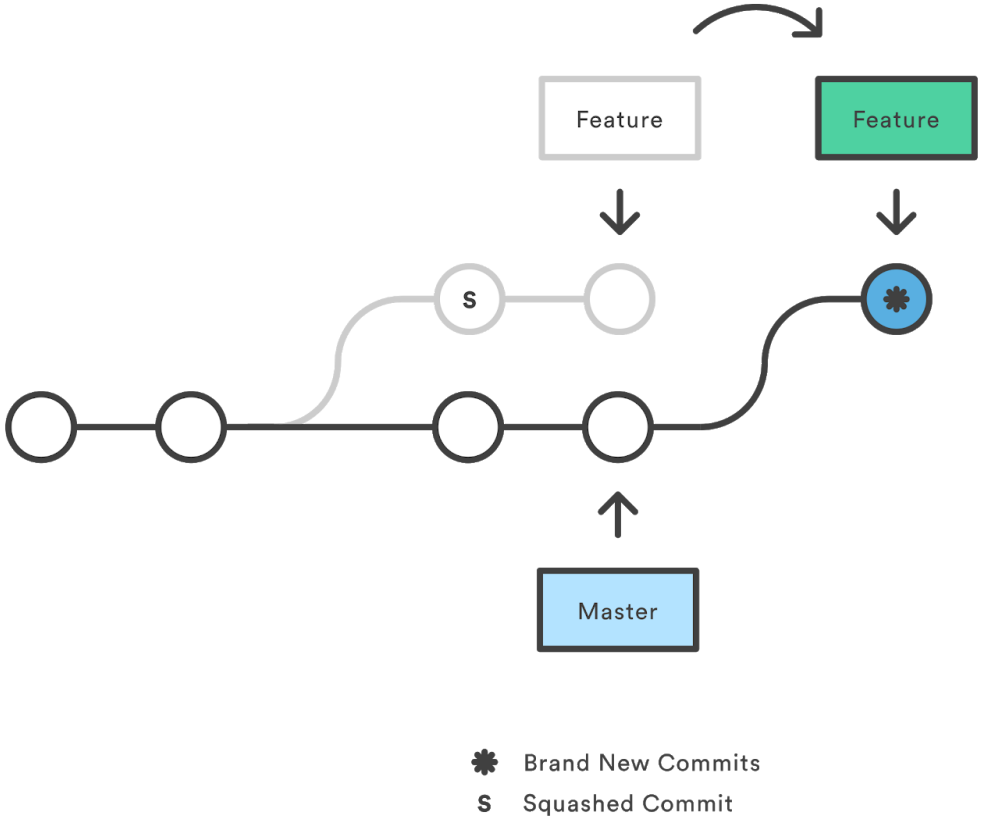
To modify older or multiple commits, you can use **git rebase** to combine a sequence of commits into a new base commit. In standard mode, git rebase allows you to literally rewrite history — automatically applying commits in your current working branch to the passed branch head. Since your new commits will be replacing the old, it's important to not use git rebase on commits that have been pushed public, or it will appear that your project history disappeared.

In these or similar instances where it's important to preserve a clean project history, adding the -i option to git rebase allows you to run rebase interactive. This gives you the opportunity to alter individual commits in the process, rather than moving all commits. You can learn more about interactive rebasing and additional rebase commands on the git rebase page.

Each regular Git commit will have a log message explaining what happened in the commit. These messages provide valuable insight into the project history. During a rebase, you can run a few commands on commits to modify commit messages.

- Reword or 'r' will stop rebase playback and let you rewrite the individual commit message during.
- Squash or 's' during rebase playback, any commits marked s will be paused on and you will be prompted to edit the separate commit messages into a combined message. More on this in the squash commits section below.
- Fixup or 'f' has the same combining effect as squash. Unlike squash, fixup commits will not interrupt rebase playback to open an editor to combine commit messages. The commits marked 'f' will have their messages discarded in-favor of the previous commit's message.

The s "squash" command is where we see the true utility of rebase. Squash allows you to specify which commits you want to merge into the previous commits. This is what enables a "clean history." During rebase playback, Git will execute the specified rebase command for each commit. In the case of squash commits, Git will open your configured text editor and prompt to combine the specified commit messages. This entire process can be visualized as follows:



12. GIT WORKFLOWS

A Git Workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Git workflows encourage users to leverage Git effectively and consistently. Git offers a lot of flexibility in how users manage changes. Given Git's focus on flexibility, there is no standardized process on how to interact with Git. **When working with a team on a Git managed project, it's important to make sure the team is all in agreement on how the flow of changes will be applied.** To ensure the team is on the same page, an agreed upon Git workflow should be developed or selected. There are several publicized Git workflows that may be a good fit for your team. Here, we'll be discussing some of these workflow options.

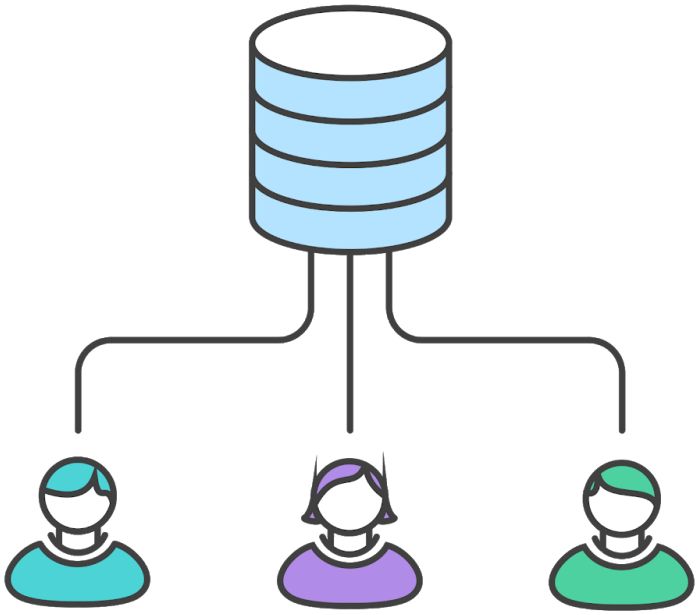
The array of possible workflows can make it hard to know where to begin when implementing Git in the workplace. This page provides a starting point by surveying the most common Git workflows for software teams.

As you read through, remember that these workflows are designed to be guidelines rather than concrete rules. We want to show you what's possible, so you can mix and match aspects from different workflows to suit your individual needs.

When evaluating a workflow for your team, it's most important that you consider your team's culture. You want the workflow to enhance the effectiveness of your team and not be a burden that limits productivity. Some things to consider when evaluating a Git workflow are:

- Does this workflow scale with team size?
- Is it easy to undo mistakes and errors with this workflow?
- Does this workflow impose any new unnecessary cognitive overhead to the team?

12.1. Centralized Workflow



in this flow, the default development branch is called master and all changes are committed into this branch. **This workflow doesn't require any other branches besides master**. A Centralized Workflow is generally better suited for very small teams. The central repository represents the official project, so its commit history should be treated as sacred and immutable.

Example

Let's take a general example at how a typical small team would collaborate using this workflow. We'll see how two developers, John and Mary, can work on separate features and share their contributions via a centralized repository.

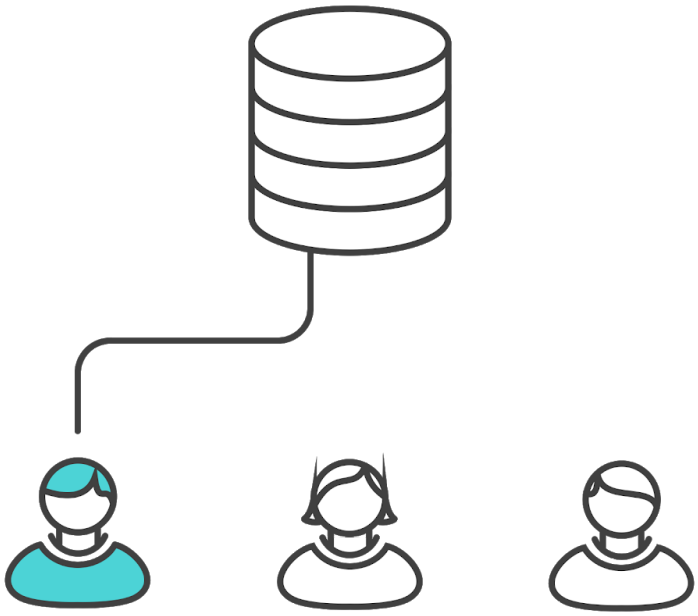
John works on his feature

In his local repository, John can develop features using the standard Git commit process: edit, stage, and commit. Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what's going on in the central repository.

Mary works on her feature

Meanwhile, Mary is working on her own feature in her own local repository using the same edit/stage/commit process. Like John, she doesn't care what's going on in the central repository, and she really doesn't care what John is doing in his local repository, since all local repositories are private.

John publishes his feature

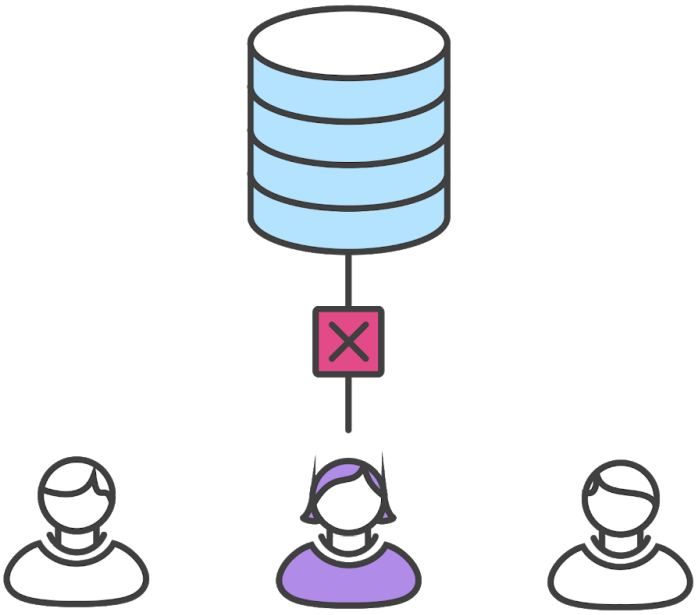


Once John finishes his feature, he should publish his local commits to the central repository so other team members can access it. He can do this with the git push command, like so:

```
git push origin master
```

Remember that origin is the remote connection to the central repository that Git created when John cloned it. The master argument tells Git to try to make the origin's master branch look like his local master branch. Since the central repository hasn't been updated since John cloned it, this won't result in any conflicts and the push will work as expected.

Mary tries to publish her feature



Let’s see what happens if Mary tries to push her feature after John has successfully published his changes to the central repository. She can use the exact same push command:

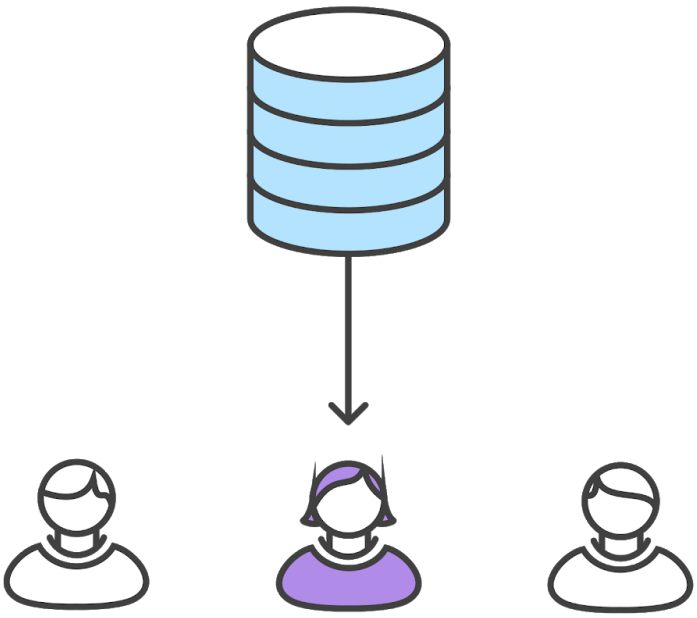
git push origin master

But, since her local history has diverged from the central repository, Git will refuse the request with a rather verbose error message:

error: failed to push some refs to '/path/to/repo.git'

This prevents Mary from overwriting official commits. She needs to pull John’s updates into her repository, integrate them with her local changes, and then try again.

Mary rebases on top of John’s commit(s)

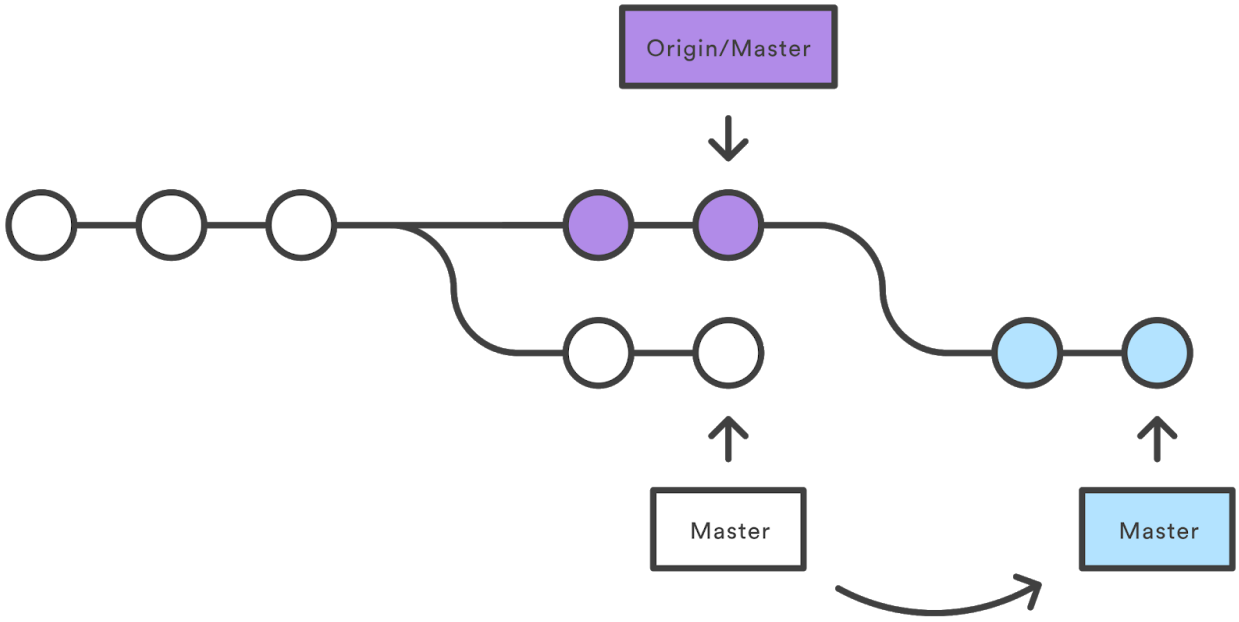


Mary can use git pull to incorporate upstream changes into her repository.

git pull --rebase origin master

The --rebase option tells Git to move all of Mary's commits to the tip of the master branch after synchronising it with the changes from the central repository, as shown below:

Mary's Repository



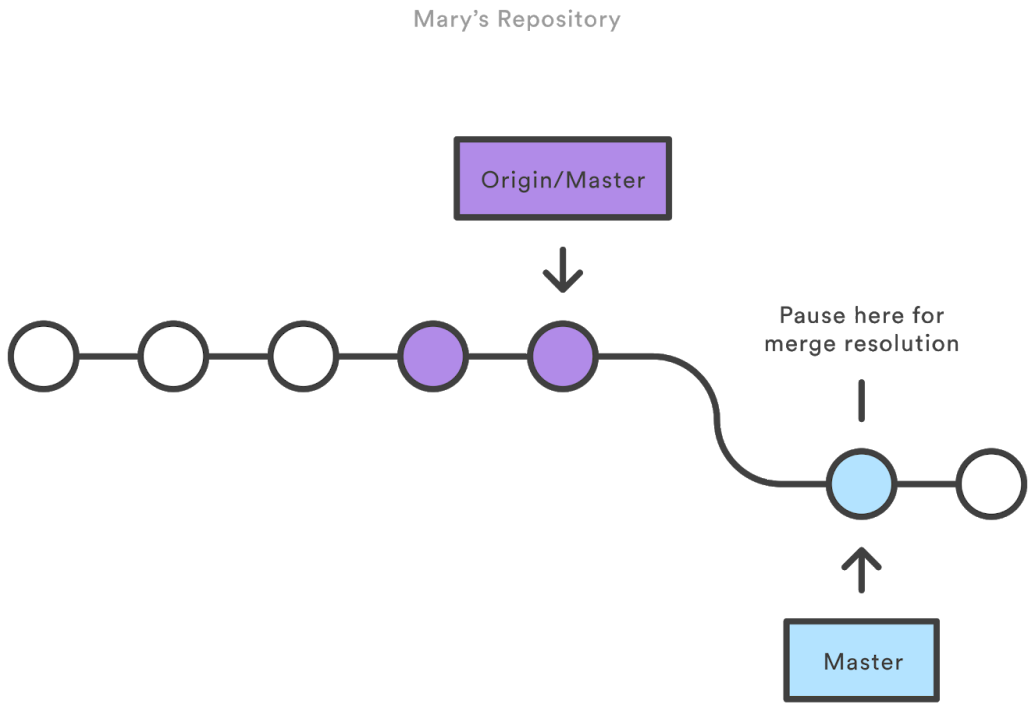
The pull would still work if you forgot this option, but you would wind up with a superfluous "merge commit" every time someone needed to synchronize with the central repository. For this workflow, it's always better to rebase instead of generating a merge commit.

Mary resolves a merge conflict

Rebasing works by transferring each local commit to the updated master branch one at a time. This means that you catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit. This keeps your commits as focused as possible and makes for a clean project history. In turn, this makes it much easier to figure out where bugs were introduced and, if necessary, to roll back changes with minimal impact on the project.

If Mary and John are working on unrelated features, it's unlikely that the rebasing process will generate conflicts. But if it does, Git will pause the rebase at the current commit and output the following message, along with some relevant instructions:

CONFLICT (content): Merge conflict in <some-file>



The great thing about Git is that anyone can resolve their own merge conflicts. In our example, Mary would simply run a git status to see where the problem is. Conflicted files will appear in the Unmerged paths section:

Unmerged paths:

(use "git reset HEAD <some-file>..." to unstage)

(use "git add/rm <some-file>..." as appropriate to mark resolution)

#

both modified: <some-file>

Then, she'll edit the file(s) to her liking. Once she's happy with the result, she can stage the file(s) in the usual fashion and let git rebase do the rest:

git add <some-file>

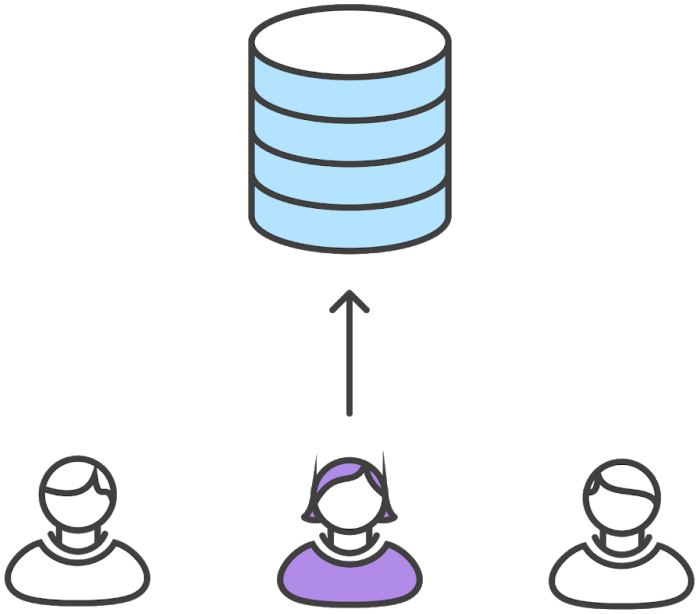
git rebase --continue

And that's all there is to it. Git will move on to the next commit and repeat the process for any other commits that generate conflicts.

If you get to this point and realize and you have no idea what's going on, don't panic. Just execute the following command and you'll be right back to where you started:

git rebase --abort

1.1.1 Mary successfully publishes her feature



After she's done synchronizing with the central repository, Mary will be able to publish her changes successfully:

git push origin master

12.2. Git Feature Branch Workflow

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the master branch will never contain broken code, which is a huge advantage for continuous integration environments.

Encapsulating feature development also makes it possible to leverage pull requests, which are a way to initiate discussions around a branch. They give other developers the opportunity to sign off on a feature before it gets integrated into the official project. Or, if you get stuck in the middle of a feature, you can open a pull request asking for suggestions from your colleagues. The point is, **pull requests make it incredibly easy for your team to comment on each other's work.**

The Feature Branch Workflow assumes a central repository, and master represents the official project history. Instead of committing directly on their local master branch, developers create

a new branch every time they start work on a new feature. Feature branches should have descriptive names, like animated-menu-items or issue-#1061. The idea is to give a clear, highly-focused purpose to each branch. Git makes no technical distinction between the master branch and feature branches, so developers can edit, stage, and commit changes to a feature branch.

In addition, **feature branches can (and should) be pushed to the central repository**. This makes it possible to share a feature with other developers without touching any official code. Since master is the only "special" branch, storing several feature branches on the central repository doesn't pose any problems. Of course, this is also a convenient way to back up everybody's local commits. The following is a walk-through of the life-cycle of a feature branch.

12.2.1. Start with the master branch

All feature branches are created off the latest code state of a project. This guide assumes this is maintained and updated in the master branch.

```
git checkout master
git fetch origin
git reset --hard origin/master
```

[**make sure you understand what the code above does and why it is like that - if not, read back your class notes or come ask me in class**]

This switches the repo to the master branch, pulls the latest commits and resets the repo's local copy of master to match the latest version.

12.2.2. Create a new-branch

Use a separate branch for each feature or issue you work on. After creating a branch, check it out locally so that any changes you make will be on that branch.

git checkout -b new-feature

This checks out a branch called new-feature based on master, and the -b flag tells Git to create the branch if it doesn't already exist.

12.2.3. Update, add, commit, and push changes

On this branch, edit, stage, and commit changes in the usual fashion, building up the feature with as many commits as necessary. Work on the feature and make commits like you would any time you use Git. When ready, push your commits, updating the feature branch on Bitbucket.

```
git status
git add <some-file>
git commit
```

12.2.4. Push feature branch to remote

It’s a good idea to push the feature branch up to the central repository. This serves as a convenient backup, when collaborating with other developers, this would give them access to view commits to the new branch.

```
git push origin new-feature
```

This command pushes new-feature to the central repository (origin).

12.2.5. Pull requests

Aside from isolating feature development, branches make it possible to discuss changes via pull requests. **Once someone completes a feature, they don’t immediately merge it into master. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into master**. This gives other developers an opportunity to review the changes before they become a part of the main codebase.

TRY IT YOURSELF

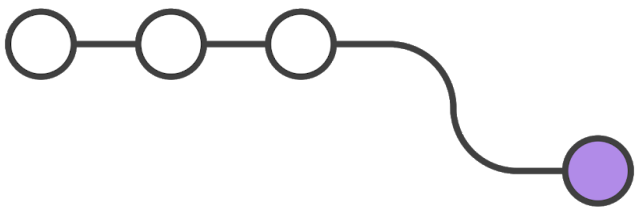
push a branch on github, and click pull request. see the options to assign it, and to review the code line by line

Code review is a major benefit of pull requests, but they’re actually designed to be a generic way to talk about code. You can think of pull requests as a discussion dedicated to a particular branch. This means that they can also be used much earlier in the development process. For example, if a developer needs help with a particular feature, all they have to do is file a pull request. Interested parties will be notified automatically, and they’ll be able to see the question right next to the relevant commits.

Once a pull request is accepted, the actual act of publishing a feature is much the same as in the Centralized Workflow. First, you need to make sure your local master is synchronized with the upstream master. Then, you merge the feature branch into master and push the updated master back to the central repository.

The following is an **example** of the type of scenario in which a feature branching workflow is used. The scenario is that of a team doing code review around on a new feature pull request. This is one example of the many purposes this model can be used for.

Mary begins a new feature



Before she starts developing a feature, Mary needs an isolated branch to work on. She can request a new branch with the following command:

git checkout -b marys-feature master

This checks out a branch called marys-feature based on master, and the -b flag tells Git to create the branch if it doesn't already exist. On this branch, Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

git status

git add <some-file>

git commit

Mary goes to lunch

Mary adds a few commits to her feature over the course of the morning. Before she leaves for lunch, it's a good idea to push her feature branch up to the central repository. This serves as a convenient backup, but if Mary was collaborating with other developers, this would also give them access to her initial commits.

git push -u origin marys-feature

This command pushes marys-feature to the central repository (origin), and the -u flag adds it as a remote tracking branch. After setting up the tracking branch, Mary can call `git push` without any parameters to push her feature.

Mary finishes her feature

When Mary gets back from lunch, she completes her feature. Before merging it into master, she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

git push

Then, she files the pull request in her Git GUI asking to merge marys-feature into master, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions about specific changesets.

Bill receives the pull request

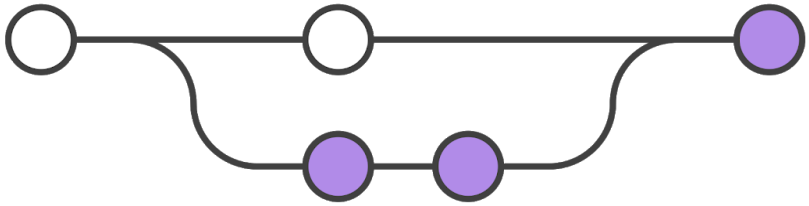
Bill gets the pull request and takes a look at marys-feature. He decides he wants to make a few changes before integrating it into the official project, and he and Mary have some back-and-forth via the pull request.

Mary makes the changes

To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. She edits, stages, commits, and pushes updates to the central repository. All her activity shows up in the pull request, and Bill can still make comments along the way.

If he wanted, Bill could pull marys-feature into his local repository and work on it on his own. Any commits he added would also show up in the pull request.

Mary publishes her feature



Once Bill is ready to accept the pull request, someone needs to merge the feature into the stable project (this can be done by either Bill or Mary):

```
git checkout master

git pull

git pull origin marys-feature

git push
```

This process often results in a merge commit. Some developers like this because it's like a symbolic joining of the feature with the rest of the code base. But, if you're partial to a linear history, it's possible to rebase the feature onto the tip of master before executing the merge, resulting in a fast-forward merge.

Some GUI's will automate the pull request acceptance process by running all of these commands just by clicking an "Accept" button. If yours doesn't, it should at least be able to automatically close the pull request when the feature branch gets merged into master.

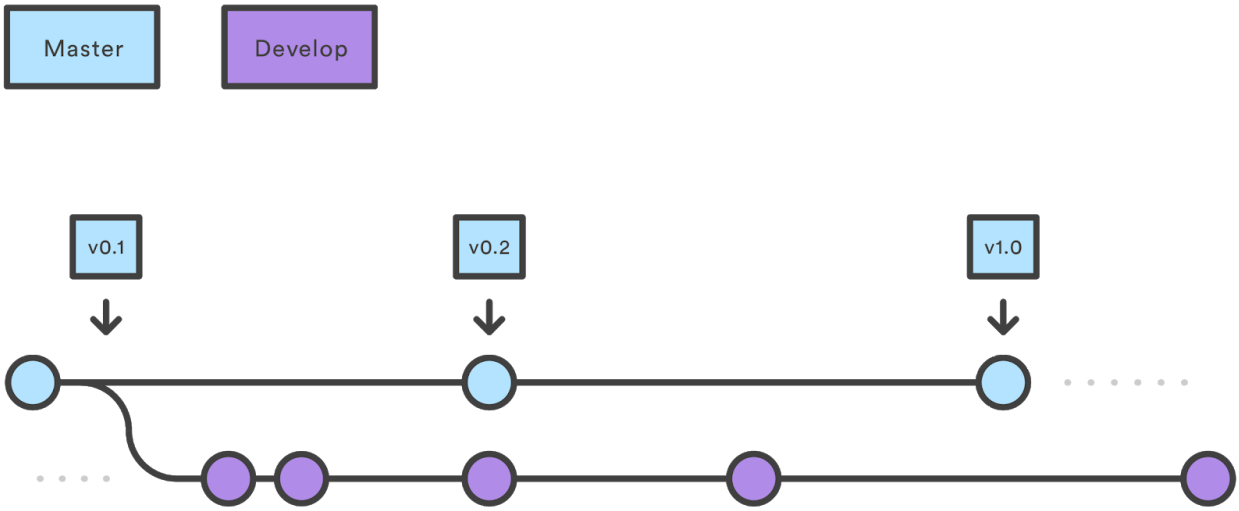
Meanwhile, John is doing the exact same thing

While Mary and Bill are working on mary's-feature and discussing it in her pull request, John is doing the exact same thing with his own feature branch. By isolating features into separate branches, everybody can work independently, yet it's still trivial to share changes with other developers when necessary.

12.3. Gitflow Workflow

Gitflow Workflow is a Git workflow design that was first published and made popular by Vincent Driessen at nvie. **The Gitflow Workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects.**

Gitflow is ideally suited for projects that have a scheduled release cycle. **This workflow doesn't add any new concepts or commands beyond what's required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, and recording releases.** Of course, you also get to leverage all the benefits of the Feature Branch Workflow: pull requests, isolated experiments, and more efficient collaboration.



12.3.1. Develop and Master Branches

Instead of a single master branch, this workflow uses two branches to record the history of the project. The *master* branch stores the official release history, and the *develop* branch serves as an integration branch for features. It's also convenient to tag all commits in the master branch with a version number.

The first step is to complement the default master with a develop branch. A simple way to do this is for one developer to create an empty develop branch locally and push it to the server:

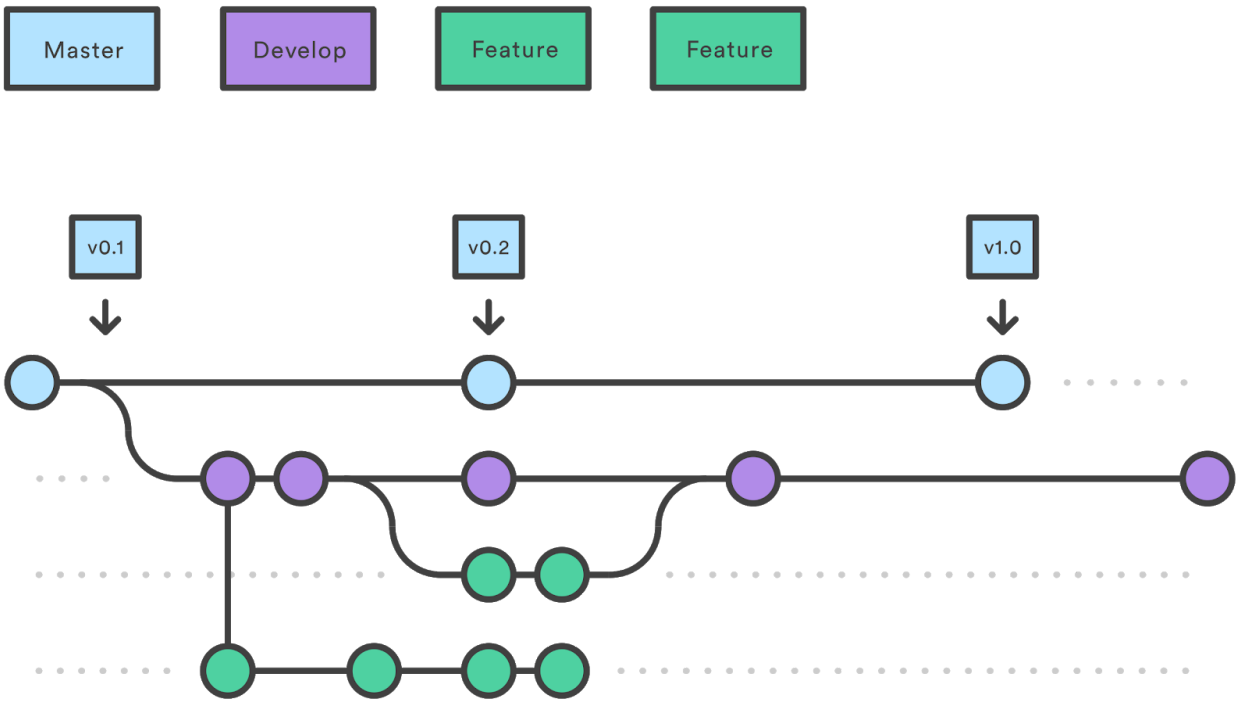
```
git branch develop
```

```
git push -u origin develop
```

This branch will contain the complete history of the project, whereas master will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for develop.

12.3.2. Feature Branches

Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. But, instead of branching off of master, feature branches use *develop* as their parent branch. When a feature is complete, it gets merged back into develop. **Features should never interact directly with master.**



Note that feature branches combined with the develop branch is, for all intents and purposes, the Feature Branch Workflow. But, the Gitflow Workflow doesn't stop there.

Feature branches are generally created off to the latest develop branch.

12.3.3. Creating a feature branch

```
git checkout develop
git checkout -b feature_branch
```

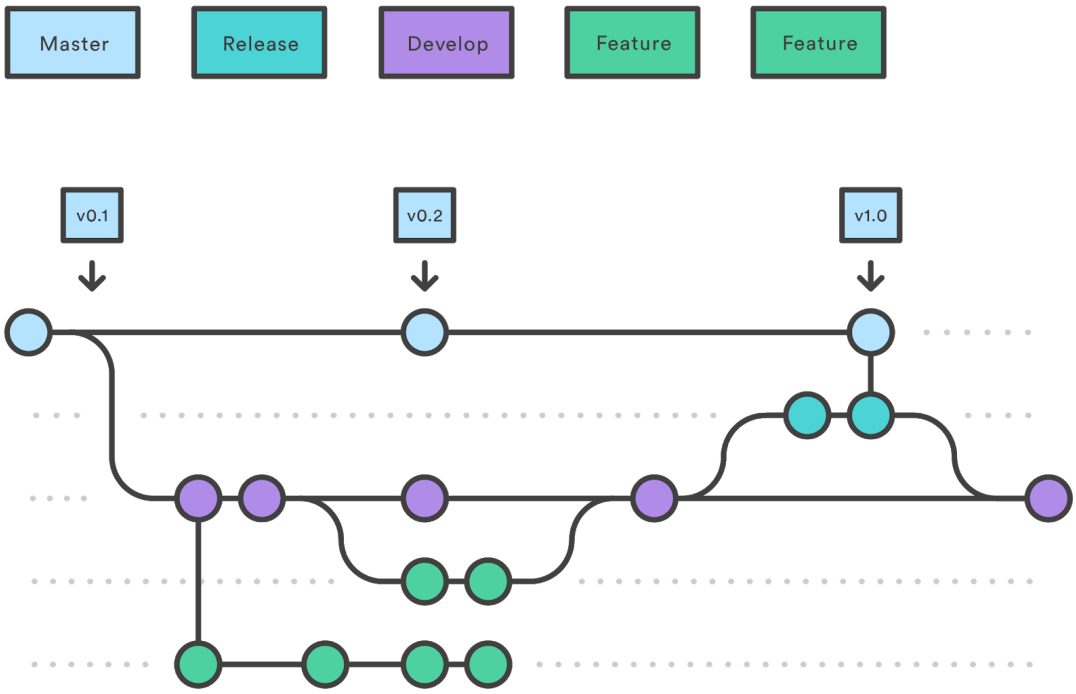
12.3.4. Finishing a feature branch

When you're done with the development work on the feature, the next step is to merge the feature_branch into develop.

git checkout develop

git merge feature_branch

12.3.5. Release Branches



Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it's easy to say, “This week we're preparing for version 4.0,” and to actually see it in the structure of the repository).

Making release branches is another straightforward branching operation.
Like feature branches, release branches are based on the develop branch. A new release branch can be created using the following methods.

```
git checkout develop
```

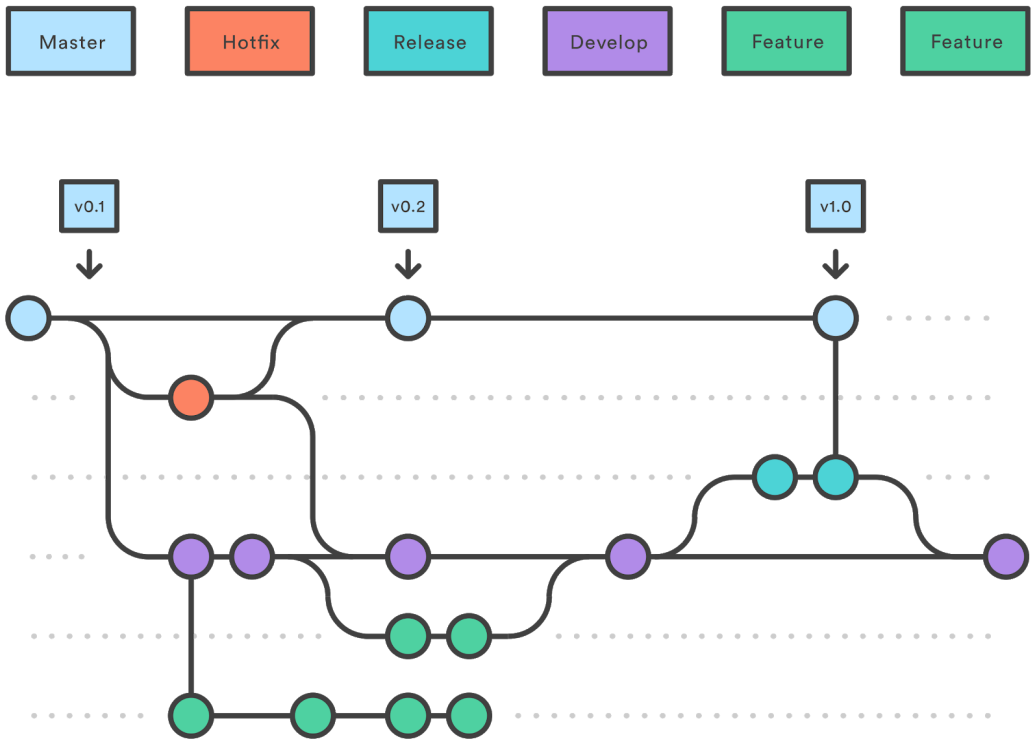
```
git checkout -b release/0.1.0
```

Once the release is ready to ship, it will get merged it into master and develop, then the release branch will be deleted. It’s important to merge back into develop because critical updates may have been added to the release branch and they need to be accessible to new features. If your organization stresses code review, this would be an ideal place for a pull request.

To finish a release branch, use the following methods:

```
git checkout develop
git merge release/0.1.0
```

12.3.6. Hotfix Branches



Maintenance or "hotfix" branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on master instead of develop. **This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master should be tagged with an updated version number.**

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with master. A hotfix branch can be created using the following methods:

```
git checkout master
git checkout -b hotfix_branch
```

12.4. Forking Workflow

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the "central" codebase, it gives every developer

their own server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one. The Forking Workflow is most often seen in public open source projects.

The main advantage of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository. This allows the maintainer to accept commits from any developer without giving them write access to the official codebase.

The Forking Workflow typically follows a branching model based on the Gitflow Workflow. This means that complete feature branches will be purposed for merge into the original project maintainer's repository. The result is a distributed workflow that provides a flexible way for large, organic teams (including untrusted third-parties) to collaborate securely. This also makes it an ideal workflow for open source projects.

As in the other Git workflows, the Forking Workflow begins with an official public repository stored on a server. But when a new developer wants to start working on the project, they do not directly clone the official repository. Instead, they *fork* the official repository to create a copy of it on the server. This new copy serves as their personal public repository—no other developers are allowed to push to it, but they can pull changes from it (we'll see why this is important in a moment). **After they have created their server-side copy, the developer performs a git clone to get a copy of it onto their local machine**. This serves as their private development environment, just like in the other workflows.

When they're ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated. The pull request also serves as a convenient discussion thread if there are issues with the contributed code. The following is a step-by-step example of this workflow.

1. A developer 'forks' an 'official' server-side repository. This creates their own server-side copy.
2. The new server-side copy is cloned to their local system.
3. A Git remote path for the 'official' repository is added to the local clone.
4. A new local feature branch is created.
5. The developer makes changes on the new branch.
6. New commits are created for the changes.
7. The branch gets pushed to the developer's own server-side copy.
8. The developer opens a pull request from the new branch to the 'official' repository.
9. The pull request gets approved for merge and is merged into the original server-side repository

To integrate the feature into the official codebase, the maintainer pulls the contributor's changes into their local repository, checks to make sure it doesn't break the project, merges it into their local master branch, then pushes the master branch to the official repository on the server. **The contribution is now part of the project, and other developers should pull from the official repository to synchronize their local repositories.**

It's important to understand that the notion of an "official" repository in the Forking Workflow is merely a convention. In fact, the only thing that makes the official repository so official is that it's the public repository of the project maintainer.

12.4.1. Forking vs cloning

It's important to note that "forked" repositories and "forking" are not special git operations. Forked repositories are created using the standard git clone command. Forked repositories are generally "server-side clones" and usually managed and hosted by a 3rd party Git service. There is no unique Git command to create forked repositories. A clone operation is essentially a copy of a repository and its history.

All new developers to a Forking Workflow project need to fork the official repository. Next each developer needs to clone their own public forked repository. They can do this with the familiar git clone command.

Whereas other Git workflows use a single origin remote that points to the central repository, the Forking Workflow requires two remotes—one for the official repository, and one for the developer's personal server-side repository. **While you can call these remotes anything you want, a common convention is to use origin as the remote for your forked repository (this will be created automatically when you run git clone) and *upstream* for the official repository**.

```
git remote add upstream <repo>
```

You'll need to create the upstream remote yourself using the above command. This will let you easily keep your local repository up-to-date as the official project progresses. Note that if your upstream repository has authentication enabled (i.e., it's not open source), you'll need to supply a username

In the developer's local copy of the forked repository they can edit code, commit changes, and create branches just like in other Git workflows:

git checkout -b some-feature

Edit some code

git commit -a -m "Add first draft of some feature"

All of their changes will be entirely private until they push it to their public repository. And, if the official project has moved forward, they can access new commits with git pull:

git pull upstream master

Since developers should be working in a dedicated feature branch, this should generally result in a fast-forward merge.

Once a developer is ready to share their new feature, they need to do two things. First, they have to make their contribution accessible to other developers by pushing it to their public repository. Their origin remote should already be set up, so all they should have to do is the following:

git push origin feature-branch

This diverges from the other workflows in that the origin remote points to the developer's personal server-side repository, not the main codebase.

Second, they need to notify the project maintainer that they want to merge their feature into the official codebase. Bitbucket and github provide a "pull request" button that leads to a form asking you to specify which branch you want to merge into the official repository. Typically, you'll want to integrate your feature branch into the upstream remote's master branch.