How to Use Git Effectively

This article explores the source code management and collaboration system called Git and provides guidance on how to get the most value from Git. Git can be used as a local source code management system, and can also be used for...

Part 1 of 5:

Understanding some basic Git terminology

1. **Understand the idea of a repository.** A "repository" is a collection of files that is the basic unit on which git operations are performed. Usually a git command operates on only one repository at a time (there are exceptions, for instance when dealing with submodules).
    1. On your machine, a repository includes a directory and all files in it and (recursively) in its subdirectories.
    2. It is possible to explicitly tell Git to ignore (i.e., not track) particular files or all files (recursively) inside a particular directory. The list of files and directories to exclude is in the .gitignore file. Files that are ignored will not be tracked with Git, and they may not be synced between different copies of the Git repository. It is generally considered good practice to remove large files and redundant files (such as executables that are obtained by compiling from the source code already present) from the Git repository, to keep its size manageable (since Git keeps track of many versions of a file, things can get quite unwieldy with large files).[5]
    3. All copies of the same Git repository, when synced, should have the same subdirectories and files, excluding those that are marked with .gitignore. If you are using GitHub or Bitbucket, the files you see when browsing online, or the files you get when you first clone the repository, will not include any ignored files.
    4. All git commands associated with a repository can be run from the command line (UNIX or Linux terminal or Git bash) from any directory within the repository. Being in the top (root) directory of the repository is not necessary.
2. **Understand the idea of the remote origin.** The "remote origin" for your local Git repository is the place that you first cloned the Git repository from. By default, this is referenced as origin in various git commands, though you can change the name to something else.[1] In a typical collaborative Git setup, there is a single remote origin for all users (managed by GitHub, Bitbucket, or Git running on their server) and all synchronization happens through that remote origin, i.e., each collaborator pushes to and pulls from that remote origin, rather than directly to another collaborator. Note, however, that it is possible to have a different architecture.
3. **Understand other key Git terminology.**
    1. A "commit" is a checkpoint of a complete state of the Git repository. Each commit has a unique identifier and it is possible to revert to any commit. Each commit includes a commit time, a committing user, and optionally, a commit message (that should ideally describe the most recent changes included in the commit).

2. A "branch" can be thought of as a version of the Git repository. The default branch is called "master". At any point, a new branch can be created cloning the existing state of the Git repository. The branches can subsequently diverge. One can switch between branches, and one can eventually merge branches.
3. The term HEAD is used for the most recent commit. In each repository, each branch has a HEAD.
4. The "status" of the repository provides information on the current branch, what changes may have been made that are not committed, and whether the changes are synced with the remote origin. If there are no uncommitted changes, then git considers the status to be clean (another way of saying this is that the branch is at its HEAD). A clean status reports an empty result when you run `git status --porcelain`. Note that changes to files marked to be ignored are not considered when reporting the status.

Part 2 of 5:

Understanding the three main uses of Git

1. **Understand Git's first use case:** periodic checkpointing through commits.[9]
    1. Each git commit creates a checkpoint that you can revert to at a later stage if you discover subsequent problems. In this respect, Git is a substitute for version control systems like Subversion (SVN), with the key distinction that whereas SVN operates at the level of individual files, Git operates at the level of the repository.
    2. The most relevant git commands here are `git stage` (synonymous to `git add`) and `git commit`.
    3. Other related commands are `git reset`, `git mv`, and `git rm`.
    4. You can see the full commit history using `git log`.
2. **Understand Git's second use case:** working with multiple branches.
    1. It is possible to create multiple branches of your repository where you work on different changes.
    2. The most relevant commands for creating and switching between branches are `git branch` and `git checkout`.
    3. The relevant command for reconciling changes between branches is `git merge` (more generally, `git merge` can be used to try to merge any two commits, and the commits at the HEAD for two different branches are a natural candidate for things to merge).
3. **Understanding Git's third use case:** collaborating across different devices and users, possibly including a web-based access from the remote origin.
    1. To push changes from your local repository to the remote origin, use `git push`.
    2. To get changes from the remote origin into your local repository, use `git fetch` (this simply fetches the changes) or `git pull` (this fetches and merges the changes in).
    3. The "collaboration" you get in Git is conceptually quite different from the kind of collaboration you get with tools like Google Docs or Hackpad. The latter emphasize real-time collaboration where participants are usually online and editing simultaneously, with a single global state of the document at all times. Git, on the other hand, works on a model of *offline* changes that are then *merged* (if two different users make changes separately from the same starting point).
    4. Git is more similar to Dropbox, in the sense that in both, there are local copies that can be edited offline, and there is a step of syncing the local copy to the remote origin (if a user is online and has Dropbox running then the changes will be made in real time). However, Dropbox does not handle multiple people editing the same file offline well, because it does not have a method of merging changes within a single file. Git's ability to merge changes *within a single file* is crucial to its usefulness for source code management.

4. **Appreciate the relationship between the three Git use cases.**
    1. **Branching and merging depends somewhat on committing**: Without clear checkpoints recording particular states of the repository, branching would be hard to manage.
    2. **Collaboration and branching share reliance on the ability to merge**: The ability to merge changes, whether between different branches in a local repository, or within the same branch but between the local repository and remote origin, is a critical part of Git's utility.
5. **Understand how Git can be used for various combinations of these use cases.**
    1. Only (1): Git can be used purely as a checkpointing system for a linear (sequential) workflow. Changes are made in a particular sequence and the state is recorded periodically through commits. This makes it easy to, if necessary, revert to a previous commit and start afresh from there, while expecting this to be a rare occurrence. This use case is most common for a single user working on a single file.
    2. Only (2): Git can be used primarily as a way of handling cases where you want to work on two different changes in parallel, without letting them interfere with one another, and merge when they are completed. If using Git this way, you would commit *only* when you want to start branching, when you are switching between branches, and when you are finished with work on the branches and want to merge.
    3. Only (3): Git can be used primarily as a way of transferring files between different devices you may be working on, plus keeping a remote backup. In this case, you commit *only* when you wish to push to the remote, which you do either when you wish to back up the files or when you are switching devices or handing over the work to another user. Here, the primary advantage of Git is that it's easier and faster than emailing files over, particularly if you are working on the same repository repeatedly but have a large and/or regularly changing set of files.
    4. (1) and (2): Git can be used as a system for checkpointing and branching/merging for a single user. The remote origin plays no particular role (and there may not even be a remote origin!).
    5. (1) and (3): Git can be used as a system for checkpointing and for collaborating across multiple users (e.g., one user works on the code in one timezone, then hands it off to the other user who works in another timezone).
    6. (2) and (3): Git can be used primarily for branching and collaboration, with infrequent checkpointing.
    7. All of them!: This is how most sophisticated Git-using teams use it.
6. **Understand the "commit often" philosophy in Git.** A general best practice guideline for Git is that it should *always* be used for at least (1) even if that is not the primary reason one is using Git. In other words, it is good to *commit often*. This is because the benefits of having checkpoints are significant, and the costs are negligible. Whether and how to use (2) and (3) (in particular, whether it's a good idea to push frequently, and whether it's a good idea to branch out regularly instead of working from master) generally depends on workplace best practices and your particular use case.[5] The one main counterpoint is that once a change has been committed, it is very hard to permanently erase it from the Git repository. Therefore, *never* commit any changes that have sensitive information in them that you might later regret other people finding out.

Part 3 of 5:

Mastering the save-stage-commit-push workflow

1. **Understand the standard workflow for making changes to the Git repository:** save, (stage-and-commit), and push. Of these, the saving step is something you handle through your editor or IDE (i.e., the same methods that you would otherwise use). The pushing step, referenced as Step (3) in Part 2, has to do

with keeping the remote origin synced up with your local Git repository. The stage-and-commit step, referenced as Step (1) in Part 2, is what checkpoints the code.

1. The set of files and file changes that have been saved are called the "working directory" and the set of files and file change that have been staged are called the "staging area".
2. As a general rule, only changes that have been saved (i.e., recorded on disk) can be staged, only changes that have been staged can be committed, and only changes that have been committed can be pushed.
3. In particular, it means that if you have some unsaved changes, then they will not get staged even if you run `git stage` on those files. If you have some non-staged changes, they will not get committed even if you run `git commit`. If you have uncommitted changes, they will not get pushed even if you run `git push`.
4. In a simple scenario where you are working in the master branch and are not doing anything too subtle, your workflow is: save changes, then run these two commands: `git commit -am "Description of your changes"` followed by `git push origin master`.

2. **Understand the role of staging and committing.**
    1. Staging a change basically means that you are telling git to record this change in the next commit. You can stage file edits, addition of new files, deletion of files, and renaming of files.
    2. It is possible to stage-and-commit together, by using `git commit -a`. The "-a" option tells git to stage all changes to existing files before committing. In fact, many Git users never stage changes explicitly, and simply use `git commit -a` as the default behavior. However, newly added files are not staged by default, and need to be staged by manually running `git add` or `git stage` as discussed later.
    3. One reason we want to stage changes explicitly rather than committing them is that we may accidentally start work on multiple changes that should logically be part of separate commits (for instance, you may be making edits to the documentation, to the code, and to the configuration, that you would like to checkpoint separately). In this case, it makes sense to stage all the changes we'd like to put in a particular commit, commit that, then stage the next set of changes, commit that, and so on.
    4. Another reason is that there may be some changes we make that we don't want to commit at all, but that we need to make concurrently with other changes that we *do* want to commit. For instance, suppose you are making changes to your code, that you are testing by running your code locally. To facilitate debugging, you have set the log-level in the configuration to DEBUG, but you want the log-level in your committed code to always be INFO. You also want to keep committing your changes to the code. In this case, you can keep staging and committing the code files while not staging the file with the log-level configuration, and when you are done with testing you can revert the change to the file with the log-level configuration.
3. **Understand some more subtleties associated with staging.**
    1. To tell git to stage a change, you can do `git add` or `git stage` followed by the name of the file. This stages all changes to that file made between the previous commit and the most recently saved version of the file. This means that if you do not do any more staging, then the next time you do a git commit, these changes will get committed.
    2. If you make and save changes to a file, stage the file, make and save further changes, and then commit, only the changes made prior to the staging are committed.
    3. If you stage a file repeatedly between commits, git does not keep track of previous stagings. In other words, git does not checkpoint uncommitted stages. Rather, outside of commits, git keeps track of only two states: all staged changed, and all saved changes. It does not keep track of multiple stages.
    4. You may stage multiple files. These will not interfere with one another.

5. You can unstage changes to a file (and revert to the previous commit) by doing `git reset HEAD` followed by the file name. You can unstage all changes by doing `git reset HEAD`. Note that git reset HEAD in and of itself does not actually un*save* the changes.
6. You can unsave unstaged changes to a file by doing `git checkout --` followed by the file name.
7. You can unstage and un*save* all changes (i.e., actually change all files to the way they looked as of the previous commit) by doing `git reset --hard`.

4. **Know how to see the current git status to get a better sense of what to stage and what to commit.**
    1. The `git status` command prints out information about the list of files where changes are staged but not committed (this list is printed in green on a color terminal) and the list of files where changes are saved but not staged (this list is printed in red on a color terminal).
    2. In particular, these two lists could potentially overlap. This happens if you make and save changes, then stage them, and then make and save further changes to some of the same files.
    3. `git status` does not show the precise changes that were made to the files. You can get a list of saved-but-not-staged changes using `git diff`. To get a list of staged-but-not-committed, use `git diff --staged` (on older versions of Git, you may need to use `git diff --cached`).[10]
    4. Once you have staged all changes, `git diff` should return nothing. Once you have committed all changes, `git status` should report that your working directory is clean, and `git status --porcelain` should return nothing.
    5. If some of the files you added should not be part of the Git repository (for instance, they are compiled files, or they are configuration files local to your system) be sure to add the files, or a subdirectory containing only such files, to the .gitignore. This helps you stay clean.
5. **Add commit messages when committing!** The best way to commit is to use one of these two methods:
    1. If you are staging all changes as part of the commit itself: `git commit -am "Description of the changes in your commit"`
    2. If you have already staged the changes you want to commit: `git commit -m "Description of the changes in your commit"`
6. **Whenever you want your changes to be available in the remote origin and to other collaborators, you should push them.** A few pointers.
    1. The standard pushing command if you are in the master branch is `git push origin master`
    2. If you are in another branch, you should do `git push origin branchName`. Branches are discussed later.
    3. When you push, you send the remote origin not just your recent commit but all previous commits as well. The HEAD (most recent committed state) of the remote origin will match that of your most recent commit, but the remote origin (and therefore any other repository that clones from or fetches from it subsequently) will have access to previous commits. This is why pushing immediately after committing does not matter too much, and it is also why it is critical to make sure you never commit anything that has sensitive security information.
    4. According to some schools of thought, you should pair every commit with a push, so that every local checkpoint is recorded immediately on the remote origin. However, other schools of thought say that you should commit often, but push occasionally.[5] You may set different standards for committing and pushing. For instance, you may set a rule that every commit should at minimum compile, whereas you will push to the remote origin only after the code has been tested. The precise guidelines depend on your use case and the guidelines your team decides.
    5. Note that when working collaboratively, i.e., if others are making changes to the remote origin, your push may fail because others have made changes. In this case you need to pull (and if necessary, merge conflicts and commit), and then push again.

Part 4 of 5:

Using branch, checkout, and merge

1. **Understand how to use git branches to better manage your code changes.**
    1. **Branches intended for merging back into master**: One practice often used is to start a git branch whenever working on a new, reasonably nontrivial feature, then keep committing and pushing within that branch, and once the feature is fully tested, merge that branch back into master. The merging step itself might be controlled: not everybody may have permission to merge into master, so the merge is submitted as a pull request, and the administrator approves the merge. Often, the branches are tied to git issues (tickets for feature changes).
    2. **Branches intended as slight modifications of master**: You might create a branch that modifies some configuration settings from master so as to facilitate a use case such as local testing. In this case, your new branch will not have many commits of its own, and will not be merged into master. However, it will regularly be merging in *from* master. For instance, a branch that you create that has slightly different configurations for local testing might be used by you sporadically to test recent changes to master; every time you use it this way, you pull in from master.
    3. **Branches that diverge for good**: In this case, the branch leaves the master for good. It may or may not continue to be developed further. In some cases, it may be used simply as an easier checkpoint of an earlier state of the project. In other cases, it may be developed further or even into a separate fork of the repository.
2. **To create a new branch, with a name that is not currently in use, you can run one of these:**

    ```
    git branch branchName
    ```

    or
    ```
    git checkout -b branchName
    ```

    1. The first of these commands creates a new branch with that name, but does not switch to it. The second command creates the branch and switches to it.
    2. With both commands, the entire current state is transferred to the new branch. This means that the commit history till this point, the staging area (i.e., the set of staged-but-not-committed changes), and the rest of the working directory (i.e., the set of saved-but-not-staged changes), all get inherited by the new branch.
    3. To switch to an already existing branch in your local repository, do `git checkout branchName`
    4. The save-stage-commit-push workflow within a branch is the same as for master. The key difference is that when pushing, you need to use `git push origin branchName` instead of `git push origin master`
    5. It is also possible for multiple people to collaborate on the same branch. The first person to create the branch simply does `git push origin branchName`. People who subsequently want to collaborate on that branch must do `git fetch origin branchName` so that their local repository gets that branch, and subsequently they can work on it in the same way as the person who created the branch.
3. **Understand what happens to stages and commits when switching between branches using `git checkout`**
    1. Committed changes do not get transferred.
    2. However, git attempts to transfer all the staged-but-not-committed changes as well as all the saved-but-not-staged changes. It prints out these files to the terminal. Existing files are printed with a "M"

prefix whereas new files are printed with an "A" prefix.
3. If any of the files to which either set of changes applies is different in the two branches, git will throw an error, telling you to commit, stash, or discard your changes to avoid them being overwritten by checkout. If the changes involve adding new files that you have not yet staged, git will print an error about untracked working tree files.

4. **Appreciate that once you are working in a separate branch, this reduces the barrier to pushing to the remote origin.**
    1. If you are making all your changes in master, then pushing incomplete changes to the remote origin can be disruptive to collaborators and confuse any person or automated process that deploys directly to production from the remote origin. If, however, you are in your own branch, then you can push this branch to the remote origin. It will not affect the copy of the master in the remote origin, and therefore not disrupt anything.
    2. A problem can arise if different people are using the same name for different local branches. For instance, if everybody uses a branch name called "testing" for their own local testing branches, these can confuse git when they attempt to push the branch to the remote origin. It is therefore good practice to choose a distinctive name, and to push to the remote origin early on to "reserve" the name so that other collaborators don't use the same name accidentally.
5. **Merge with master (or whatever branch you want to merge into) with caution.**
    1. It is generally good practice to first update your master from the remote, e.g., using `git checkout master` and then `git pull origin master`
    2. It is preferable to first merge in from master into your branch rather than the other way around, so that your local master branch is not corrupted by a bad merge. If you wish, you can create a third branch just for merging, though this is usually overkill.
    3. To merge in from master into your branch, do `git checkout branchName` followed by `git merge master`. If you are okay with the default merge edit message, use `git merge --no-edit master`. If you want to merge from master into a branch without switching to the branch you are merging into, do `git merge master branchName`.
    4. The merge may execute without conflicts. In this case, Git automatically creates a merge commit with the default commit message (or your own edit message if you chose to enter one).
    5. Examine the changes and do any further testing to make sure that any changes that occurred to master while you were working on your branch do not interfere with the functionality of what you did. Note that just because Git merges without conflicts does *not* imply that the merge is non-problematic: there could be subtle semantic problems introduced by the merge that will become apparent only after you compile, test, or run your code.
    6. Once you are confident that your changes are good, do `git checkout master` and `git merge --no-edit branchName`. There should be no conflicts at this time. Now, do `git push origin master`.
    7. You may want to delete your branch by doing `git branch -D branchName`, or you may want to preserve it. Note that even if you delete your branch, all its associated commits are preserved if you have merged it into master.
    8. When merging, there is an option to "fast-forward" changes that is active by default on the command line. This will effectively avoid creating a separate commit for the merge if it is not necessary, i.e., if the branch being merged into has had no changes to it since the other branch diverged from it. The "merge pull request" option on GitHub by default does *not* fast-forward changes, so as to always provide an identifiable commit responsible for the merge. You can deactivate this option to by doing `--no-ff`. As a general rule, fast-forwarding makes sense for short-lived branches but non-fast-forward merges make the commit history easier to understand and more faithful to reality.[11]

9. In some cases, the merge may generate conflicts. This usually happens if both branches have made edits to the same file in a way that Git can't automatically reconcile. Git's default behavior is to replace the file with a file where each of the conflicting sections is marked. You have to manually examine and clear the conflicts. While this is sometimes easy, it is often easier to just discard the failed merge and manually attempt to understand the sets of changes and figure out how to merge them. In the worst case, the badly merged file can just be overridden by a new, rewritten file. Such manual merges need to be explicitly committed after the merge is completed. Until this commit is done, git is in an "unresolved index" state and does not allow you to switch branches.
10. Another way some people prefer to merge is by first rebasing. Rebasing rearranges the commit history to have the commits in the branch being merged from happen first, and *then* execute the commits in the branch being merged to. In other words, it rewrites the history of the order in which the commits happened: rather than keep the full branched history of the commits, it pretends that the commits in the branch being merged to happened later in a linear flow. Since rebasing involves a rewrite of the branch history, it is considered a dangerous option for beginners but it can be quite useful at times.[12][13]
11. Note that in the absence of merge conflicts, rebasing will *always* avoid creating additional merge commits. Fast-forwarding avoids additional merge commits only if there are no additional changes in the branch being merged *into*. We can think of fast-forwarding as a special case of rebasing in the case where no changes have been made in the branch being merged into.

Part 5 of 5:

Understanding pulling and pushing

1. **Understand how git fetch and git pull work.**
    1. `git fetch origin branchName` fetches the version of the `branchName` branch from the remote origin and stores it under the name `origin/branchName` and can be manipulated in a way similar to local branches. Note that this requires one to be able to connect to the remote server.
    2. `git pull origin branchName` is shorthand for the following pair of commands: `git fetch origin branchName`, followed by `git merge origin/branchName`. Note that only the first of these requires connectivity to the remote server. If `git merge origin/branchName` is run in isolation, it does not connect to the remote server, instead simply running the merge from whenever the fetch previously occurred.
    3. As with all merge efforts, if there are no changes to merge (i.e., if all changes in the remote are already present locally) then it will print "Already up-to-date."
    4. There may, however, be changes to merge and conflicts to resolve, and you need to deal with them the same way as you deal with merges between branches, as described in Step (5) of the previous part.
    5. It is generally not good practice to run `git pull origin branchName` if your working directory is unclean, because the pulled-in changes may conflict with uncommitted changes in your working directory.
2. **Understand how git push works.**
    1. If you run `git status`, then git will inform you if it notices that your local version of your current branch is ahead of the most recently fetched remote version. Note that git will *not* try to fetch a remote version unless explicitly told to do so, so this is based on potentially outdated information. Note also that unpushed changes do not compromise git's evaluation of whether the worrying directory is clean.

2. If you run `git push origin branchName`, git attempts to push your changes to the remote origin. At this time, if git discovers that there were changes on the remote origin that are not in your local version, the attempt will be rejected. You would then need to do `git pull` (as described in Step (1)).