Portfolio 5

Version control and Git

A **version control system** (sometimes called revision control) is a tool that lets users track the history and attribution of their project files over time (stored in a repository), and which helps the developers in the team to work together. Modern version control systems help them work simultaneously, in a non-blocking way, by giving each developer his or her own sandbox, preventing their work in progress from conflicting, and all the while providing a mechanism to merge changes and synchronize work.

**Distributed version control** systems such as **Git** give each developer his or her own copy of the project's history, a clone of a repository. This is what makes Git fast:

1. Nearly all operations are performed locally, and are flexible.
2. Users can set up repositories in many ways.

Repositories meant for developing also provide a separate working area (or a worktree) with project files for each developer. The branching model used by Git enables cheap local branching and flexible branch publishing, allowing to use branches for context switching and for sandboxing different works in progress (making possible, among other things, a very flexible topic branch workflow).

The fact that the whole history is accessible allows for long-term undo, rewinding back to last working version, and so on. Tracking ownership of changes automatically makes it possible to find out who was responsible for any given area of code, and when each change was done. Users can compare different revisions, go back to the revision a user is sending a bug report against, and even automatically find out which revision introduced a regression bug. The fact that Git is tracking changes to the tips of branches with reflog allows for easy undo and recovery.

A unique feature of Git is that it enables explicit access to the staging area for creating commits (new revisions of a project). This brings additional flexibility to managing users working area and deciding on the shape of a future commit.

**Git Branch**

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed in Git Basics, the first module of this series. 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 users create, list, rename, and delete branches. It doesn’t let users 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.

**Usage**

git branch

List all of the branches in user repository.

git branch <branch>

Create a new branch called <branch>. This does not check out the new branch.

git branch -d <branch>

Delete the specified branch. This is a “safe” operation in that Git prevents user from deleting the branch if it has unmerged changes.

git branch -D <branch>

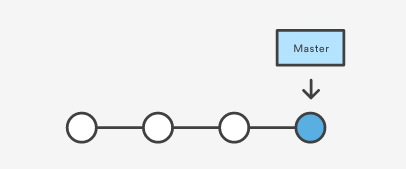
Force delete the specified branch, even if it has unmerged changes. This is the command to use if user want to permanently throw away all of the commits associated with a particular line of development.

git branch -m <branch>

Rename the current branch to <branch>.

**Creating Branches**

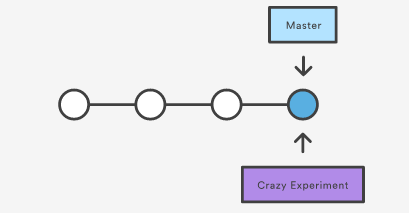
It's important to understand that branches are just pointers to commits. When user create a branch, all Git needs to do is create a new pointer and it doesn’t change the repository in any other way. So, if user start with a repository that looks like this:



Then, user create a branch using the following command:

git branch crazy-experiment

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



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

**Git Checkout**

The git checkout command lets user navigate between the branches created by git branch. Checking out a branch updates the files in the working directory to match the version stored in that branch, and it tells Git to record all new commits on that branch. Think of it as a way to select which line of development users are working on.

**Usage**

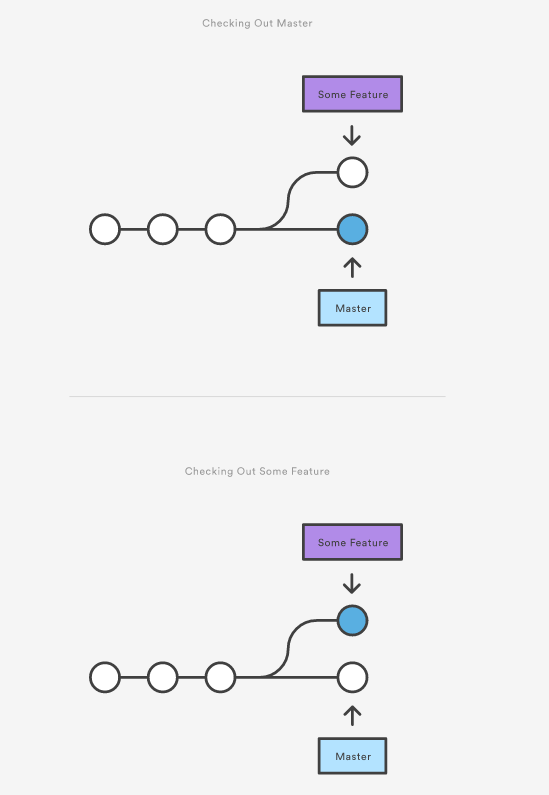
git checkout <existing-branch>

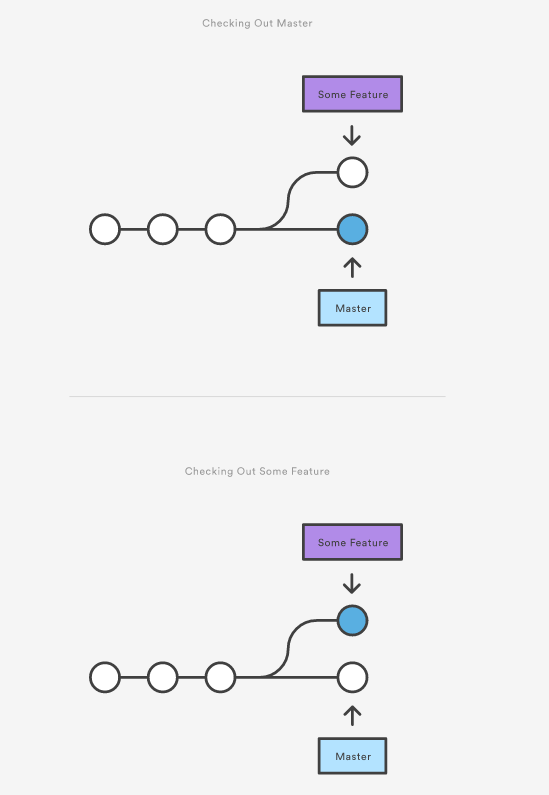
Check out the specified branch, which should have already been created with git branch. This makes <existing-branch> the current branch, and updates the working directory to match.

git checkout -b <new-branch>

Create and check out <new-branch>. The -b option is a convenience flag that tells Git to run git branch <new-branch> before running git checkout <new-branch>. git checkout -b <new-branch> <existing-branch>

Git checkout works hand-in-hand with git branch. When users want to start a new feature, users need to create a branch with git branch, then check it out with git checkout. They can work on multiple features in a single repository by switching between them with git checkout.





**Git Merge**

Merging is Git's way of putting a forked history back together again. The git merge command lets user take the independent lines of development created by git branch and integrate them into a single branch. Most often, it is necessary when a file is modified by two people on two different computers at the same time. When two branches are merged, the result is a single collection of files that contains both sets of changes.

In some cases, the merge can be performed automatically, because there is sufficient history information to reconstruct the changes, and the changes do not conflict. In other cases, a person must decide exactly what the resulting files should contain. Many revision control software tools include merge capabilities.

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 target 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.

**Usage**

git merge <branch>

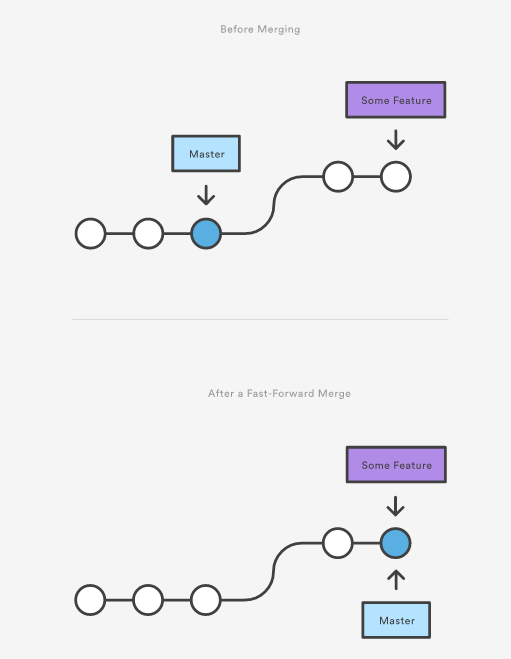
Merge the specified branch into the current branch. Git will determine the merge algorithm automatically.

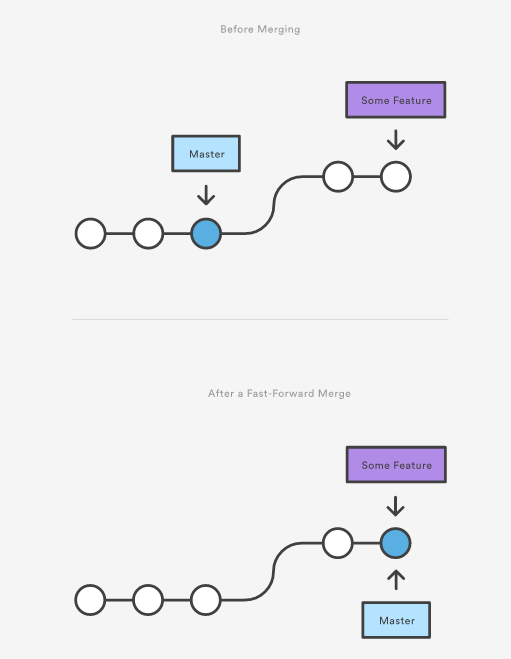
git merge --no-ff <branch>

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 user’s repository.

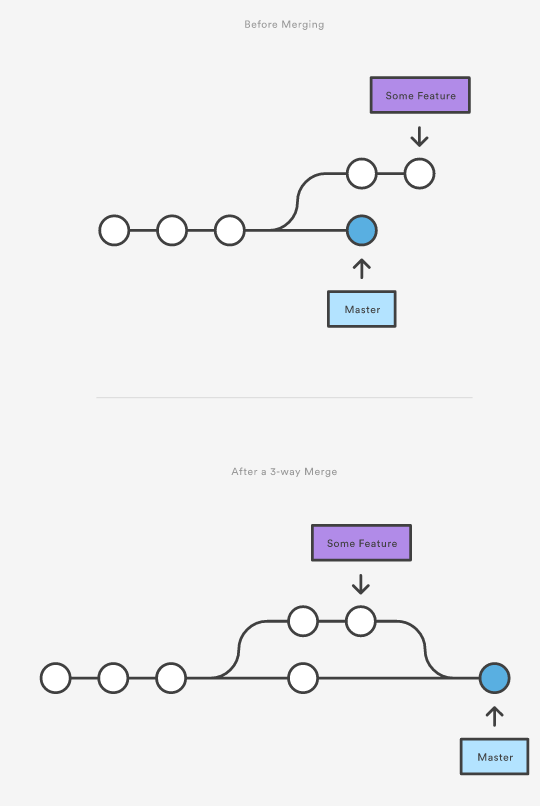
Once users have finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of user’s repository, Git has several distinct algorithms to accomplish this: a fast-forward merge or a 3-way merge.

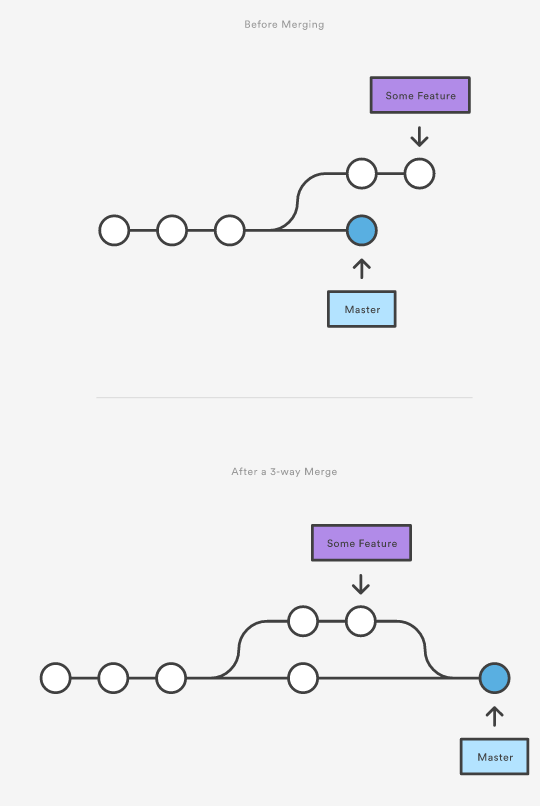
A **fast-forward merge** can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively 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:





However, a fast-forward merge is not possible if the branches have diverged. When there is not a 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.



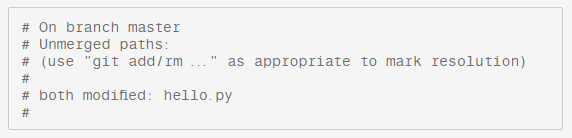


While user can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through rebasing) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a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

**Resolving Conflicts**

If the two branches users are 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 user 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 user encounter a merge conflict, running the git status command shows user which files need to be resolved. For example, if both branches modified the same section of hello.py, user would see something like the following:



Then, users can go in and fix up the merge to they liking. When users are ready to finish the merge, all they have to do is run git add on the conflicted file(s) to tell Git they're resolved. Then, users 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.