Git Reference

#### **Version 1.0**

#### **<29/07/2013>**

Change Log

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Revision Date | Changed By | Comments |
| 1.0 | 29-Jul-2013 | Prasen Revankar | General Introduction.  <https://www.atlassian.com/git/> |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Overview 8](#_Toc362951196)

[2. Git Basics 8](#_Toc362951197)

[2.1. The git init Command 8](#_Toc362951200)

[Usage 8](#_Toc362951201)

[Discussion 9](#_Toc362951202)

[Example 10](#_Toc362951203)

[2.2. The git clone Command 11](#_Toc362951204)

[Usage 11](#_Toc362951205)

[Discussion 11](#_Toc362951206)

[Example 12](#_Toc362951207)

[2.3. The git config Command 13](#_Toc362951208)

[Usage 13](#_Toc362951209)

[Discussion 13](#_Toc362951210)

[Example 14](#_Toc362951211)

[2.4. The git add Command 16](#_Toc362951212)

[Usage 16](#_Toc362951213)

[Discussion 16](#_Toc362951214)

[Example 17](#_Toc362951215)

[Usage 18](#_Toc362951216)

[Discussion 18](#_Toc362951217)

[Example 19](#_Toc362951218)

[2.6. The git status Command 21](#_Toc362951219)

[Usage 21](#_Toc362951220)

[Discussion 21](#_Toc362951221)

[Example 22](#_Toc362951222)

[2.7. The git log Command 24](#_Toc362951223)

[Usage 24](#_Toc362951224)

[Discussion 25](#_Toc362951225)

[Example 26](#_Toc362951226)

[3. Undoing Git Changes 27](#_Toc362951227)

[3.1. The git checkout Command 27](#_Toc362951228)

[Usage 27](#_Toc362951229)

[Discussion 28](#_Toc362951230)

[Example 29](#_Toc362951231)

[3.2. The git revert Command 31](#_Toc362951232)

[Usage 31](#_Toc362951233)

[Discussion 31](#_Toc362951234)

[Example 32](#_Toc362951235)

[3.3. The git clean Command 33](#_Toc362951236)

[Usage 33](#_Toc362951237)

[Discussion 34](#_Toc362951238)

[Examples 35](#_Toc362951239)

[3.4. The git clean Command 38](#_Toc362951240)

[Usage 38](#_Toc362951241)

[Discussion 38](#_Toc362951242)

[Example 39](#_Toc362951243)

[4. Git Branches 40](#_Toc362951244)

[4.1. The git branch Command 40](#_Toc362951245)

[Usage 40](#_Toc362951246)

[Discussion 41](#_Toc362951247)

[Example 42](#_Toc362951248)

[4.2. The git checkout Command 44](#_Toc362951249)

[Usage 44](#_Toc362951250)

[Discussion 44](#_Toc362951251)

[Example 46](#_Toc362951252)

[4.3. The git merge Command 48](#_Toc362951253)

[Usage 48](#_Toc362951254)

[Discussion 48](#_Toc362951255)

[Example 50](#_Toc362951256)

[5. Rewriting Git History 54](#_Toc362951257)

[5.1. The git commit --amend Command 54](#_Toc362951258)

[Usage 55](#_Toc362951259)

[Discussion 55](#_Toc362951260)

[Example 55](#_Toc362951261)

[5.2. The git rebase Command 56](#_Toc362951262)

[Usage 56](#_Toc362951263)

[Discussion 56](#_Toc362951264)

[Examples 58](#_Toc362951265)

[5.3. The git rebase -i Command 59](#_Toc362951266)

[Usage 59](#_Toc362951267)

[Discussion 59](#_Toc362951268)

[Examples 59](#_Toc362951269)

[5.4. The git reflog Command 62](#_Toc362951270)

[Usage 62](#_Toc362951271)

[Discussion 62](#_Toc362951272)

[Example 62](#_Toc362951273)

[6. Remote Git Repositories 64](#_Toc362951274)

[6.1. The git remote Command 64](#_Toc362951275)

[Usage 65](#_Toc362951276)

[Discussion 65](#_Toc362951277)

[Examples 66](#_Toc362951278)

[6.2. The git fetch Command 66](#_Toc362951279)

[Usage 66](#_Toc362951280)

[Discussion 66](#_Toc362951281)

[Examples 67](#_Toc362951282)

[6.3. The git pull Command 69](#_Toc362951283)

[Usage 69](#_Toc362951284)

[Discussion 69](#_Toc362951285)

[Examples 71](#_Toc362951286)

[6.4. The git push Command 71](#_Toc362951287)

[Usage 71](#_Toc362951288)

[Discussion 72](#_Toc362951289)

[Examples 73](#_Toc362951290)

[7. Git Workflows 74](#_Toc362951291)

[7.1. Centralized Workflow 74](#_Toc362951292)

[How It Works 75](#_Toc362951293)

[Managing Conflicts 75](#_Toc362951294)

[Example 76](#_Toc362951295)

[Someone initializes the central repository 76](#_Toc362951296)

[Everybody clones the central repository 77](#_Toc362951297)

[John works on his feature 78](#_Toc362951298)

[Mary works on her feature 79](#_Toc362951299)

[John publishes his feature 79](#_Toc362951300)

[Mary tries to publish her feature 80](#_Toc362951301)

[Mary rebases on top of John’s commit(s) 81](#_Toc362951302)

[Mary resolves a merge conflict 82](#_Toc362951303)

[Mary successfully publishes her feature 84](#_Toc362951304)

[7.2. Feature Branch Workflow 85](#_Toc362951305)

[How It Works 86](#_Toc362951306)

[Pull Requests 86](#_Toc362951307)

[Example 87](#_Toc362951308)

[Mary begins a new feature 87](#_Toc362951309)

[Mary goes to lunch 88](#_Toc362951310)

[Mary finishes her feature 88](#_Toc362951311)

[Bill receives the pull request 89](#_Toc362951312)

[Mary makes the changes 90](#_Toc362951313)

[Mary publishes her feature 90](#_Toc362951314)

[Meanwhile, John is doing the exact same thing 91](#_Toc362951315)

[7.3. Gitflow Workflow 91](#_Toc362951316)

[How It Works 92](#_Toc362951317)

[Historical Branches 92](#_Toc362951318)

[Feature Branches 92](#_Toc362951319)

[Release Branches 93](#_Toc362951320)

[Maintenance Branches 94](#_Toc362951321)

[Example 95](#_Toc362951322)

[Create a develop branch 95](#_Toc362951323)

[Mary and John begin new features 96](#_Toc362951324)

[Mary finishes her feature 96](#_Toc362951325)

[Mary begins to prepare a release 97](#_Toc362951326)

[Mary finishes the release 98](#_Toc362951327)

[End-user discovers a bug 99](#_Toc362951328)

[7.4. Forking Workflow 100](#_Toc362951329)

[How It Works 100](#_Toc362951330)

[The Official Repository 101](#_Toc362951331)

[Branching in the Forking Workflow 101](#_Toc362951332)

[Example 101](#_Toc362951333)

[The project maintainer initializes the official repository 101](#_Toc362951334)

[Developers fork the official repository 102](#_Toc362951335)

[Developers clone their forked repositories 102](#_Toc362951336)

[Developers work on their features 103](#_Toc362951337)

[Developers publish their features 104](#_Toc362951338)

[The project maintainer integrates their features 105](#_Toc362951339)

[Developers synchronize with the official repository 106](#_Toc362951340)

# Overview

Many teams are making the switch from Centralized version control systems like Subversion to Distributed version control systems like Git.

This is an introduction to Git using visual guides to workflows and Git commands, with examples. Keep your current workflow style while enjoying the lightweight and feature-rich benefits that Git has to offer

# Git Basics

The *Git Basics* tutorial provides a succinct overview of the most important Git commands. First, the *Setting Up a Repository* section explains all of the tools you need to start a new version-controlled project. Then, the remaining sections introduce your everyday Git commands.

By the end of this module, you should be able to create a Git repository, record snapshots of your project for safekeeping, and view your project's history.

2. 1. The git init Command

The git init command creates a new Git repository. It can be used to convert an existing, unversioned project to a Git repository or initialize a new empty repository. Most of the other Git commands are not available outside of an initialized repository, so this is usually the first command you’ll run in a new project.

Executing git init creates a .git subdirectory in the project root, which contains all of the necessary metadata for the repo. Aside from the .git directory, an existing project remains unaltered (unlike SVN, Git doesn't require a .git folder in every subdirectory).

### Usage

git init

Transform the current directory into a Git repository. This adds a .git folder to the current directory and makes it possible to start recording revisions of the project.

git init <directory>

Create an empty Git repository in the specified directory. Running this command will create a new folder called <directory> containing nothing but the .git subdirectory.

git init --bare <directory>

Initialize an empty Git repository, but omit the working directory. Shared repositories should always be created with the --bare flag (see discussion below). Conventionally, repositories initialized with the --bare flag end in .git. For example, the bare version of a repository calledmy-project should be stored in a directory called my-project.git.

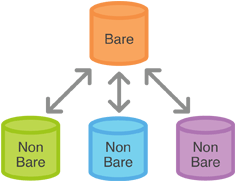
### Discussion

Compared to SVN, the git init command is an incredibly easy way to create new version-controlled projects. Git doesn’t require you to create a repository, import files, and check out a working copy. All you have to do is cd into your project folder and run git init, and you’ll have a fully functional Git repository.

However, for most projects, git init only needs to be executed once to create a central repository—developers typically don't use git init to create their local repositories. Instead, they'll usually use git clone to copy an existing repository onto their local machine.

#### Bare Repositories

The --bare flag creates a repository that doesn’t have a working directory, making it impossible to edit files and commit changes in that repository. Central repositories should always be created as bare repositories because pushing branches to a non-bare repository has the potential to overwrite changes. Think of --bare as a way to mark a repository as a storage facility, opposed to a development environment. This means that for virtually all Git workflows, the central repository is bare, and developers local repositories are non-bare.



### Example

Since git clone is a more convenient way to create local copies of a project, the most common use case for git init is to create a central repository:

ssh @  
cd path/above/repo   
git init --bare my-project.git  
First, you SSH into the server that will contain your central repository. Then, you navigate to wherever you’d like to store the project. Finally, you use the --bare flag to create a central storage repository. Developers would then [clone](https://www.atlassian.com/git/tutorial/git-basics#!clone) my-project.git to create a local copy on their development machine.

## The git clone Command

The git clone command copies an existing Git repository. This is sort of like svn checkout, except the “working copy” is a full-fledged Git repository—it has its own history, manages its own files, and is a completely isolated environment from the original repository.

As a convenience, cloning automatically creates a remote connection called origin pointing back to the original repository. This makes it very easy to interact with a central repository.

### Usage

git clone <repo>

Clone the repository located at <repo> onto the local machine. The original repository can be located on the local filesystem or on a remote machine accessible via HTTP or SSH.

git clone <repo> <directory>

Clone the repository located at <repo> into the folder called <directory> on the local machine.

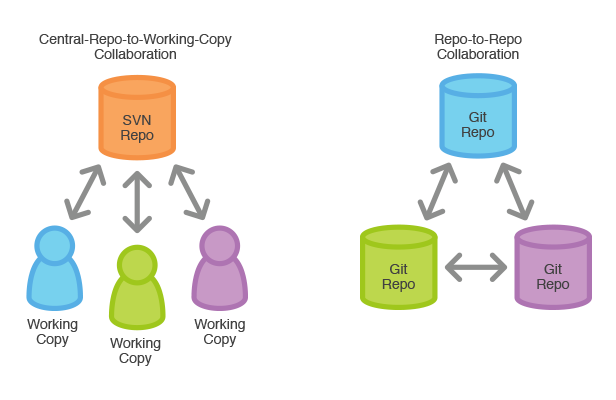
### Discussion

If a project has already been set up in a central repository, the git clone command is the most common way for users to obtain a development copy. Like [git init](https://www.atlassian.com/git/tutorial/git-basics#!init), cloning is generally a one-time operation—once a developer has obtained a working copy, all version control operations and collaborations are managed through their local repository.

#### Repo-To-Repo Collaboration

It’s important to understand that Git’s idea of a “working copy” is very different from the working copy you get by checking out code from an SVN repository. Unlike SVN, Git makes no distinction between the working copy and the central repository—they are all full-fledged Git repositories.

This makes collaborating with Git fundamentally different than with SVN. Whereas SVN depends on the relationship between the central repository and the working copy, Git’s collaboration model is based on repository-to-repository interaction. Instead of checking a working copy into SVN’s central repository, you [push](https://www.atlassian.com/git/tutorial/remote-repositories#!push) or [pull](https://www.atlassian.com/git/tutorial/remote-repositories#!pull) commits from one repository to another.



Of course, there’s nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the “central” repository, it’s possible to replicate a [Centralized workflow](https://www.atlassian.com/git/workflows#!workflow-centralized) using Git. The point is, this is accomplished through conventions rather than being hardwired into the VCS itself.

### Example

The example below demonstrates how to obtain a local copy of a central repository stored on a server accessible at example.com using the SSH username john:

git clone ssh://john@example.com/path/to/my-project.git   
cd my-project  
# Start working on the project

The first command initializes a new Git repository in the my-project folder on your local machine and populates it with the contents of the central repository. Then, you can cd into the project and start editing files, committing snapshots, and interacting with other repositories. Also note that the .git extension is omitted from the cloned repository. This reflects the non-bare status of the local copy.

## The git config Command

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.

### Usage

git config user.name <name>

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.

git config --global user.email <email>

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

git config --global alias.<alias-name> <git-command>

Create a shortcut for a Git command.

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).

git config --global --edit

Open the global configuration file in a text editor for manual editing.

### Discussion

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.

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.

### Example

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

# Select your favorite text editor  
git config --global core.editor vim

# Add some SVN-like aliases  
git config --global alias.st status  
git config --global alias.co checkout  
git config --global alias.br branch  
git config --global alias.up rebase  
git config --global alias.ci commit

This will produce the ~/.gitconfig file from the previous section.

## The git add Command

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 adddoesn't really affect the repository in any significant way—changes are not actually recorded until you run [git commit](https://www.atlassian.com/git/tutorial/git-basics#!commit).

In conjunction with these command, you'll also need [git status](https://www.atlassian.com/git/tutorial/git-basics#!status) to view the state of the working directory and the staging area.

### Usage

git add <file>

Stage all changes in <file> for the next commit.

git add <directory>

Stage all changes in <directory> for the next commit.

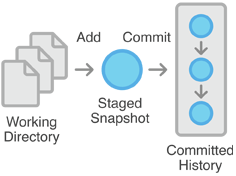
git add -p

Begin an interactive staging session that lets you choose portions of a file to add to the next commit. This will present you with a hunk of changes and prompt you for a command. Use y to stage the hunk, n to ignore the hunk, s to split it into smaller hunks, e to manually edit the hunk, and q to exit.

### Discussion

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 add command should not be confused with svn add, which adds a file to the repository. Instead, git add works on the more abstract level of *changes*. This means that git add needs to be called every time you alter a file, whereas svn add only needs to be called once for each file. It may sound redundant, but this workflow makes it much easier to keep a project organized.

#### 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 if you’re coming from an SVN (or even a Mercurial) background. 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.

### Example

When you’re starting a new project, git add serves the same function as svn import. To create an initial commit of the current directory, use the following two commands:

git add .  
git commit

Once you’ve got your project up-and-running, new files can be added by passing the path to git add:

git add hello.py  
git commit

The above commands can also be used to record changes to existing files. Again, Git doesn’t differentiate between staging changes in new files vs. changes in files that have already been added to the repository.

* 1. **The git commit Command**

The git commit command commits the staged snapshot to the project history. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicity ask it to. Along with git add, this is one of the most important Git commands.

While they share the same name, this command is nothing like svn commit. Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories.

### Usage

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 -m "<message>"

Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.

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).

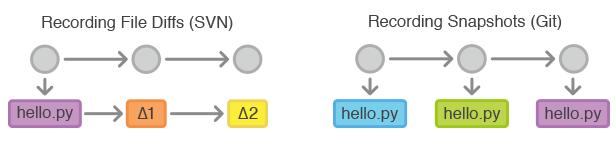
### Discussion

Snapshots are always committed to the local repository. This is fundamentally different from SVN, wherein the working copy is committed to the central repository. In contrast, 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 changes the basic development model for Git users. Instead of making a change and committing it directly to the central repo, Git developers have the opportunity to accumulate commits in their local repo. This has many advantages over SVN-style collaboration: it makes it easier to split up a feature into atomic commits, keep related commits grouped together, and clean up local history before publishing it to the central repository. It also lets developers work in an isolated environment, deferring integration until they’re at a convenient break point.

#### Snapshots, Not Differences

Aside from the practical distinctions between SVN and Git, their underlying implementation also follow entirely divergent design philosophies. Whereas SVN tracks differences of a file, Git’s version control model is based on snapshots. For example, an SVN commit consists of a diff compared to the original file added to the repository. Git, on the other hand, records the entire contents of each file in every commit.



This makes many Git operations much faster than SVN, 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's snapshot model has a far-reaching impact on virtually every aspect of its version control model, affecting everything from its branching and merging tools to its collaboration workflows.

### Example

The following example assumes you’ve edited some content in a file called hello.py and are ready to commit it to the project history. First, you need to stage the file with git add, then you can commit the staged snapshot.

git add hello.py  
git commit

This will open a text editor (customizable via git config) asking for a commit message, along with a list of what’s being committed:

# Please enter the commit message for your changes. Lines starting  
# with '#' will be ignored, and an empty message aborts the commit.  
# On branch master  
# Changes to be committed:  
# (use "git reset HEAD ..." to unstage)  
#  
# modified: hello.py  
#

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

Note that many developers also like to use 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.

## The git status Command

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven’t, and which files aren’t being tracked by Git. Status output does *not* show you any information regarding the committed project history. For this, you need to use [git log](https://www.atlassian.com/git/tutorial/git-basics#!log).

### Usage

git status

List which files are staged, unstaged, and untracked.

### Discussion

The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:

# On branch master  
# Changes to be committed:  
# (use "git reset HEAD ..." to unstage)  
#  
# modified: hello.py  
#  
# Changes not staged for commit:  
# (use "git add ..." to update what will be committed)  
# (use "git checkout -- ..." to discard changes in working directory)  
#  
# modified: main.py  
#  
# Untracked files:  
# (use "git add ..." to include in what will be committed)  
#  
# hello.pyc

#### Ignoring Files

Untracked files typically fall into two categories. They're either files that have just been added to the project and haven't been committed yet, or they're compiled binaries like .pyc, .obj, .exe, etc. While it's definitely beneficial to include the former in the git status output, the latter can make it hard to see what’s actually going on in your repository.

For this reason, Git lets you completely ignore files by placing paths in a special file called .gitignore. Any files that you'd like to ignore should be included on a separate line, and the \* symbol can be used as a wildcard. For example, adding the following to a .gitignore file in your project root will prevent compiled Python modules from appearing in git status:

\*.pyc

### Example

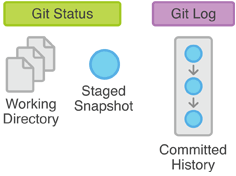
It's good practice to check the state of your repository before committing changes so that you don't accidentally commit something you don't mean to. This example displays the repository status before and after staging and committing a snapshot:

# Edit hello.py  
git status  
# hello.py is listed under "Changes not staged for commit"  
git add hello.py  
git status  
# hello.py is listed under "Changes to be committed"  
git commit  
git status  
# nothing to commit (working directory clean)

The first status output will show the file as unstaged. The git add action will be reflected in the second git status, and the final status output will tell you that there is nothing to commit—the working directory matches the most recent commit. Some Git commands (e.g., [git merge](https://www.atlassian.com/git/tutorial/git-branches#!merge)) require the working directory to be clean so that you don't accidentally overwrite changes.

## The git log Command

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.



Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format. Some of the most common configurations of git log are presented below.

### Usage

git log

Display the entire commit history using the default formatting. If the output takes up more than one screen, you can use Space to scroll and q to exit.

git log -n <limit>

Limit the number of commits by <limit>. For example, git log -n 3 will display only 3 commits.

git log --oneline

Condense each commit to a single line. This is useful for getting a high-level overview of the project history.

git log --stat

Along with the ordinary git log information, include which files were altered and the relative number of lines that were added or deleted from each of them.

git log -p

Display the patch representing each commit. This shows the full diff of each commit, which is the most detailed view you can have of your project history.

git log --author="<pattern>"

Search for commits by a particular author. The  argument can be a plain string or a regular expression.

git log --grep="<pattern>"

Search for commits with a commit message that matches <pattern>, which can be a plain string or a regular expression.

git log <since>..<until>

Show only commits that occur between <since> and <until>. Both arguments can be either a commit ID, a branch name, HEAD, or any other kind of [revision reference](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html).

git log <file>

Only display commits that include the specified file. This is an easy way to see the history of a particular file.

git log --graph --decorate ﻿--oneline

A few useful options to consider. The --graph flag that will draw a text based graph of the commits on the left hand side of the commit messages. --decorate adds the names of branches or tags of the commits that are shown. --oneline shows the commit information on a single line making it easier to browse through commits at-a-glance.

### Discussion

The git log command is Git's basic tool for exploring a repository’s history. It’s what you use when you need to find a specific version of a project, figure out what changes will be introduced by merging in a feature branch, or see which developer(s) have been slacking.

commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7  
Author: John Smith

Most of this is pretty straightforward; however, the first line warrants some explanation. The 40-character string after commit is an SHA-1 checksum of the commit’s contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.

This ID can be used in commands like git log <since>..<until> to refer to specific commits. For instance, git log 3157e..5ab91 will display everything between the commits with ID's 3157e and 5ab91. Aside from checksums, branch names (discussed in the [Branch Module](https://www.atlassian.com/wac/landing/git/training/git-branches)) and the HEAD keyword are other common methods for referring to individual commits. HEAD always refers to the current commit, be it a branch or a specific commit.

The ~ character is useful for making relative references to the parent of a commit. For example, 3157e~1 refers to the commit before 3157e, and HEAD~3 is the great-grandparent of the current commit.

The idea behind all of these identification methods is to let you perform actions based on specific commits. The git log command is typically the starting point for these interactions, as it lets you find the commits you want to work with.

### Example

The *Usage* section provides many examples of git log, but keep in mind that several options can be combined into a single command:

git log --author="John Smith" -p hello.py

This will display a full diff of all the changes John Smith has made to the file hello.py.

The .. syntax is a very useful tool for comparing branches. The next example displays a brief overview of all the commits that are in some-feature that are not in master.

git log --oneline master..some-feature

# Undoing Git Changes

This tutorial provides all of the necessary skills to work with previous revisions of a software project. First, it shows you how to explore old commits, then it explains the difference between reverting public commits in the project history vs. resetting unpublished changes on your local machine.

## The git checkout Command

The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out branches. In this module, we’re only concerned with the first two configurations.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

### Usage

git checkout master

Return to the master branch. Branches are covered in depth in the next module, but for now, you can just think of this as a way to get back to the “current” state of the project.

git checkout <commit> <file>

Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

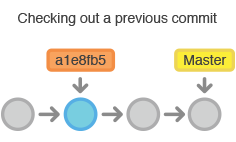
git checkout <commit>

Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit> argument. This will put you in a detached HEAD state.

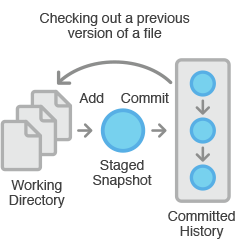
### Discussion

The whole idea behind any version control system is to store “safe” copies of a project so that you never have to worry about irreparably breaking your code base. Once you’ve built up a project history, git checkout is an easy way to “load” any of these saved snapshots onto your development machine.

Checking out an old commit is a read-only operation. It’s impossible to harm your repository while viewing an old revision. The “current” state of your project remains untouched in the masterbranch (see the [Branches Module](https://www.atlassian.com/git/tutorial/git-branches) for details). During the normal course of development, the HEAD usually points to master or some other local branch, but when you check out a previous commit, HEAD no longer points to a branch—it points directly to a commit. This is called a “detached HEAD” state, and it can be visualized as the following:



On the other hand, checking out an old file does affect the current state of your repository. You can re-commit the old version in a new snapshot as you would any other file. So, in effect, this usage of git checkout serves as a way to revert back to an old version of an individual file.



### Example

#### Viewing an Old Revision

This example assumes that you’ve started developing a crazy experiment, but you’re not sure if you want to keep it or not. To help you decide, you want to take a look at the state of the project before you started your experiment. First, you’ll need to find the ID of the revision you want to see.

git log --oneline

Let’s say your project history looks something like the following:

b7119f2 Continue doing crazy things  
872fa7e Try something crazy  
a1e8fb5 Make some important changes to hello.py  
435b61d Create hello.py  
9773e52 Initial import

You can use git checkout to view the “Make some import changes to hello.py” commit as follows:

git checkout a1e8fb5

This makes your working directory match the exact state of the a1e8fb5 commit. You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project. *Nothing* you do in here will be saved in your repository. To continue developing, you need to get back to the “current” state of your project:

git checkout master

This assumes that you're developing on the default master branch, which will be thoroughly discussed in the Branches Module.

Once you’re back in the master branch, you can use either git revert or git reset to undo any undesired changes.

#### Checking Out a File

If you’re only interested in a single file, you can also use git checkout to fetch an old version of it. For example, if you only wanted to see the hello.py file from the old commit, you could use the following command:

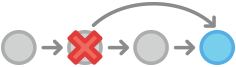
git checkout a1e8fb5 hello.py

Remember, unlike checking out a commit, this *does* affect the current state of your project. The old file revision will show up as a “Change to be committed,” giving you the opportunity to revert back to the previous version of the file. If you decide you don’t want to keep the old version, you can check out the most recent version with the following:

git checkout HEAD hello.py

## The git revert Command

The git revert command undoes a committed snapshot. But, instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a *new* commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.



### Usage

git revert <commit>

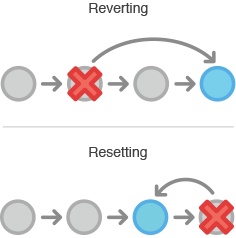
Generate a new commit that undoes all of the changes introduced in <commit>, then apply it to the current branch.

### Discussion

Reverting should be used when you want to remove an entire 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.

#### Reverting vs. Resetting

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](https://www.atlassian.com/git/tutorial/undoing-changes#!reset), not a revert.



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. For details about why altering shared history is dangerous, please see the [git reset](https://www.atlassian.com/git/tutorial/undoing-changes#!reset) page.

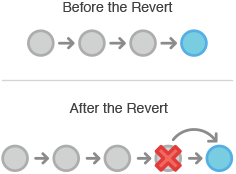
Second, git revert is able to target an individual commit at an arbitrary point in the history, whereas git reset can only work backwards from the current commit. For example, if you wanted to undo an old commit with git reset, you would have to remove all of the commits that occurred after the target commit, remove it, then re-commit all of the subsequent commits. Needless to say, this is not an elegant undo solution.

### Example

The following example is a simple demonstration of git revert. It commits a snapshot, then immediately undoes it with a revert.

# Edit some tracked files  
  
# Commit a snapshot  
git commit -m "Make some changes that will be undone"  
  
# Revert the commit we just created  
git revert HEAD

This can be visualized as the following:



Note that the 4th 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 3rd and 5th commits represent the exact same code base, and the 4th commit is still in our history just in case we want to go back to it down the road.

## The git clean Command

If git revert is a “safe” way to undo changes, you can think of git reset as the *dangerous*method. When you undo with git reset(and the commits are no longer referenced by any ref or the reflog), there is no way to retrieve the original copy—it is a *permanent* undo. Care must be taken when using this tool, as it’s one of the only Git commands that has the potential to lose your work.

Like [git checkout](https://www.atlassian.com/git/tutorial/undoing-changes#!checkout), git reset is a versatile command with many configurations. It can be used to remove committed snapshots, although it’s more often used to undo changes in the staging area and the working directory. In either case, it should only be used to undo *local*changes—you should never reset snapshots that have been shared with other developers.

### Usage

git reset <file>

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

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>

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.

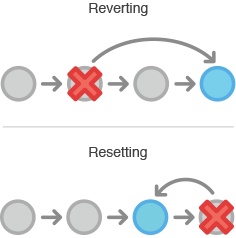
git reset --hard <commit>

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.

### Discussion

All of the above invocations are used to remove changes from a repository. Without the --hardflag, git reset is a way to clean up a repository by unstaging changes or uncommitting a series of snapshots and re-building them from scratch. The --hard flag comes in handy when an experiment has gone horribly wrong and you need a clean slate to work with.

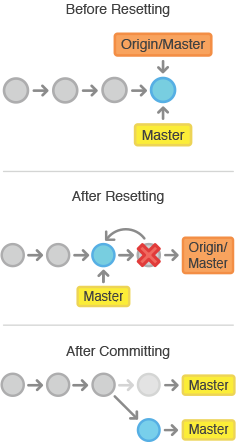
Whereas reverting is designed to safely undo a *public* commit, git reset is designed to undo*local* changes. Because of their distinct goals, the two commands are implemented differently: resetting completely removes a changeset, whereas [reverting](https://www.atlassian.com/git/tutorial/undoing-changes#!revert) maintains the original changeset and uses a new commit to apply the undo.



#### Don’t Reset Public History

You should never use git reset <commit> when any snapshots after <commit> have been pushed to a public repository. After publishing a commit, you have to assume that other developers are reliant upon it.

Removing a commit that other team members have continued developing poses serious problems for collaboration. When they try to sync up with your repository, it will look like a chunk of the project history abruptly disappeared. The sequence below demonstrates what happens when you try to reset a public commit. The origin/master branch is the central repository’s version of your local master branch.



As soon as you add new commits after the reset, Git will think that your local history has diverged from origin/master, and the merge commit required to synchronize your repositories is likely to confuse and frustrate your team.

The point is, make sure that you’re using git reset <commit> on a local experiment that went wrong—not on published changes. If you need to fix a public commit, the git revert command was designed specifically for this purpose.

### Examples

#### Unstaging a File

The git reset command is frequently encountered while preparing the staged snapshot. The next example assumes you have two files called hello.py and main.py that you’ve already added to the repository.

# Edit both hello.py and main.py  
  
# Stage everything in the current directory  
git add .  
  
# Realize that the changes in hello.py and main.py  
# should be committed in different snapshots  
  
# Unstage main.py  
git reset main.py  
  
# Commit only hello.py  
git commit -m "Make some changes to hello.py"  
  
# Commit main.py in a separate snapshot  
git add main.py  
git commit -m "Edit main.py"

As you can see, git reset helps you keep your commits highly-focused by letting you unstage changes that aren’t related to the next commit.

#### Removing Local Commits

The next example shows a more advanced use case. It demonstrates what happens when you’ve been working on a new experiment for a while, but decide to completely throw it away after committing a few snapshots.

# Create a new file called `foo.py` and add some code to it  
  
# Commit it to the project history  
git add foo.py  
git commit -m "Start developing a crazy feature"  
  
# Edit `foo.py` again and change some other tracked files, too  
  
# Commit another snapshot  
git commit -a -m "Continue my crazy feature"  
  
# Decide to scrap the feature and remove the associated commits  
git reset --hard HEAD~2

The git reset HEAD~2 command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history. Remember that this kind of reset should only be used on *unpublished* commits. Never perform the above operation if you’ve already pushed your commits to a shared repository.

## The git clean Command

The git clean command removes untracked files from your working directory. This is really more of a convenience command, since it’s trivial to see which files are untracked with git status and remove them manually. Like an ordinary rm command, git clean is *not*undoable, so make sure you really want to delete the untracked files before you run it.

The git clean command is often executed in conjunction with git reset --hard. Remember that resetting only affects tracked files, so a separate command is required for cleaning up untracked ones. Combined, these two commands let you return the working directory to the exact state of a particular commit.

### Usage

git clean -n

Perform a “dry run” of git clean. This will show you which files are going to be removed without actually doing it.

git clean -f

Remove untracked files from the current directory. The -f (force) flag is required unless the clean.requireForce configuration option is set to false (it's true by default). This will *not*remove untracked folders or files specified by .gitignore.

git clean -f <path>

Remove untracked files, but limit the operation to the specified path.

git clean -df

Remove untracked files *and* untracked directories from the current directory.

git clean -xf

Remove untracked files from the current directory as well as any files that Git usually ignores.

### Discussion

The git reset --hard and git clean -f commands are your best friends after you’ve made some embarrassing developments in your local repository and want to burn the evidence. Running both of them will make your working directory match the most recent commit, giving you a clean slate to work with.

The git clean command can also be useful for cleaning up the working directory after a build. For example, it can easily remove the .o and .exe binaries generated by a C compiler. This is occasionally a necessary step before packaging a project for release. The -x option is particularly convenient for this purpose.

Keep in mind that, along with git reset, git clean is one of the only Git commands that has the potential to permanently delete commits, so be careful with it. In fact, it’s so easy to lose important additions that the Git maintainers *require* the -f flag for even the most basic operations. This prevents you from accidentally deleting everything with a naive git clean call.

### Example

The following example obliterates all changes in the working directory, including new files that have been added. It assumes you’ve already committed a few snapshots and are experimenting with some new developments.

# Edit some existing files  
# Add some new files  
# Realize you have no idea what you're doing  
# Undo changes in tracked files  
git reset --hard  
# Remove untracked files  
git clean -df

After running this reset/clean sequence, the working directory and the staging area will look exactly like the most recent commit, and git status will report a clean working directory. You're now ready to begin again.

Note that, unlike the second example in [git reset](https://www.atlassian.com/git/tutorial/undoing-changes#!reset), the new files were *not*added to the repository. As a result, they could not be affected by git reset --hard, and git clean was required to delete them.

# Git Branches

This tutorial is a comprehensive introduction to Git branches. First, we'll take a look at creating branches, which is like requesting a new project history. Then, we'll see how git checkout can be used to select a branch. Finally, we'll learn how git merge can integrate the history of independent branches.

As you read, remember that Git branches aren't like SVN branches. Whereas SVN branches are only used to capture the occasional large-scale development effort, Git branches are an integral part of your everyday workflow.

## The git branch Command

A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process discussed in [Git Basics](https://www.atlassian.com/git/tutorial/git-basics), the first module of this series. You can think of them as a way to request a brand new working directory, staging area, and project history. 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](https://www.atlassian.com/git/tutorial/git-branches#!checkout) and [git merge](https://www.atlassian.com/git/tutorial/git-branches#!merge) commands.

### Usage

git branch

List all of the branches in your 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 you from deleting the branch if it has unmerged changes.

git branch -D <branch>

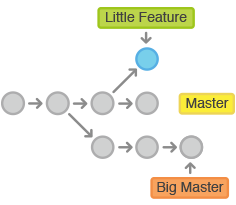
Force delete the specified branch, even if it has unmerged changes. This is the command to use if you want to permanently throw away all of the commits associated with a particular line of development.

git branch -m <branch>

Rename the current branch to <branch>.

### Discussion

In Git, branches are a part of your everyday development process. 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 sure that unstable code is never committed to the main code base, and it gives you the chance to clean up your feature’s history before merging it into the main branch.



For example, the above diagrams 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 branchfree from questionable code.

#### Branch Tips

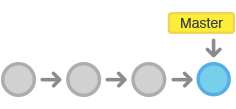
The implementation behind Git branches is much more lightweight than SVN’s model. Instead of copying files from directory to directory, Git stores a branch as a reference to a commit. In this sense, a branch represents the *tip* of a series of commits—it's not a *container* for commits. The history for a branch is extrapolated through the commit relationships.

This has a dramatic impact on Git's merging model. Whereas merges in SVN are done on a file-basis, Git lets you work on the more abstract level of commits. You can actually see merges in the project history as a joining of two independent commit histories.

### Example

#### Creating Branches

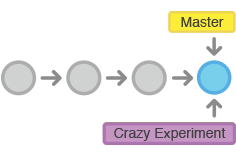
It's important to understand that branches are just *pointers* to commits. When you create a branch, all Git needs to do is create a new pointer—it doesn’t change the repository in any other way. So, if you start with a repository that looks like this:



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 withgit checkout, and then use the standard git add and git commit commands. Please see the [git checkout](https://www.atlassian.com/git/tutorial/git-branches#!checkout) section of this module for more information.

#### 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 crazy-experiment

However, if the branch hasn’t been merged, the above command will output an error message:

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

This protects you from losing your reference to those commits, which means you would effectively lose 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:

git branch -D crazy-experiment

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

## The git checkout Command

The git checkout command lets you 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 you’re working on.

In the [previous module](https://www.atlassian.com/git/tutorial/undoing-changes), we saw how git checkout can be used to view old commits. Checking out branches is similar in that the working directory is updated to match the selected branch/revision; however, new changes are saved in the project history—that is, it’s not a read-only operation.

### 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.

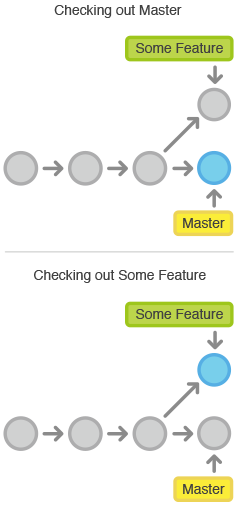
git checkout -b <new-branch>

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>

Same as the above invocation, but base the new branch off of <existing-branch> instead of the current branch.

### Discussion

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

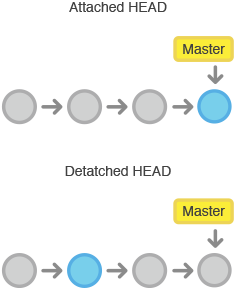


Having a dedicated branch for each new feature is a dramatic shift from the traditional SVN workflow. It makes it ridiculously easy to try new experiments without the fear of destroying existing functionality, and it makes it possible to work on many unrelated features at the same time. In addition, branches also facilitate several collaborative workflows.

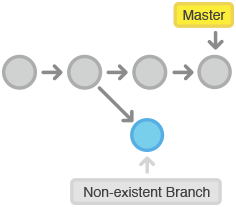
#### Detached HEADs

Now that we’ve seen the three main uses of git checkout we can talk about that “detached HEAD” we encountered in the previous module.

Remember that the HEAD is Git’s way of referring to the current snapshot. Internally, the git checkout command simply updates the HEAD to point to either the specified branch or commit. When it points to a branch, Git doesn't complain, but when you check out a commit, it switches into a “detached HEAD” state.



This is a warning telling you that everything you’re doing is “detached” from the rest of your project’s development. If you were to start developing a feature while in a detached HEAD state, there would be no branch allowing you to get back to it. When you inevitably check out another branch (e.g., to merge your feature in), there would be no way to reference your feature:



The point is, your development should always take place on a branch—never on a detached HEAD. This makes sure you always have a reference to your new commits. However, if you’re just looking at an old commit, it doesn’t really matter if you’re in a detached HEAD state or not.

### Example

The following example demonstrates the basic Git branching process. When you want to start working on a new feature, you create a dedicated branch and switch into it:

git branch new-feature  
git checkout new-feature

Then, you can commit new snapshots just like we’ve seen in previous modules:

# Edit some files  
git add <file>  
git commit -m "Started work on a new feature"  
# Repeat

All of these are recorded in new-feature, which is completely isolated from master. You can add as many commits here as necessary without worrying about what’s going on in the rest of your branches. When it’s time to get back to “official” code base, simply check out the masterbranch:

git checkout master

This shows you the state of the repository before you started your feature. From here, you have the option to merge in the completed feature, branch off a brand new, unrelated feature, or do some work with the stable version of your project.

## The git merge Command

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by [git branch](https://www.atlassian.com/git/tutorial/git-branches#!branch) and integrate them into a single branch.

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](https://www.atlassian.com/git/tutorial/git-branches#!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 (discussed below).

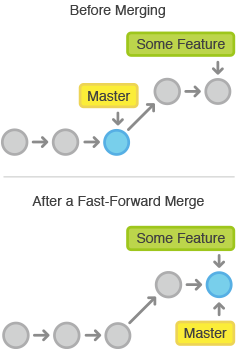
git merge --no-ff <branch>

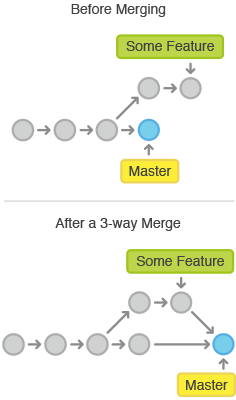
Merge the specified branch into the current branch, but *always* generate a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

### Discussion

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. Depending on the structure of your 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 intomaster 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 you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through [rebasing](https://www.atlassian.com/wac/landing/git/training/rewriting-git-history#!rebase)) 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 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](https://www.atlassian.com/wac/landing/git/training/git-basics#!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.

### Example

#### Fast-Forward Merge

Our first example demonstrates a fast-forward merge. The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.

# Start a new feature  
git checkout -b new-feature master  
  
# Edit some files  
git add   
git commit -m "Start a feature"  
  
# Edit some files  
git add   
git commit -m "Finish a feature"  
  
# Merge in the new-feature branch  
git checkout master  
git merge new-feature  
git branch -d new-feature

This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Also note that Git should not complain about the git branch -d, since new-feature is now accessible from the master branch.

#### 3-Way Merge

The next example is very similar, but requires a 3-way merge because master progresses while the feature is in-progress. This is a common scenario for large features or when several developers are working on a project simultaneously.

# Start a new feature  
git checkout -b new-feature master  
  
# Edit some files  
git add   
git commit -m "Start a feature"  
  
# Edit some files  
git add   
git commit -m "Finish a feature"  
  
# Develop the master branch  
git checkout master  
  
# Edit some files  
git add   
git commit -m "Make some super-stable changes to master"  
  
# Merge in the new-feature branch  
git merge new-feature  
git branch -d new-feature

Note that it’s impossible for Git to perform a fast-forward merge, as there is no way to move master up to new-feature without backtracking.

For most workflows, new-feature would be a much larger feature that took a long 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.

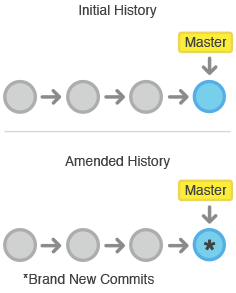
# Rewriting Git History

Git's main job is to make sure you never lose a committed change. But, it's also designed to give you total control over your development workflow. This includes letting you define exactly what your project history looks like; however, it also creates the potential to lose commits. Git provides its history-rewriting commands under the disclaimer that using them *may* result in lost content.

This tutorial discusses some of the most common reasons for overwriting committed snapshots and shows you how to avoid the pitfalls of doing so.

### The git commit --amend Command

The git commit --amend command is a convenient way to fix up the most recent commit. It lets you combine staged changes with the previous commit instead of committing it as an entirely new snapshot. It can also be used to simply edit the previous commit message without changing its snapshot.



But, amending doesn’t just alter the most recent commit—it replaces it entirely. To Git, it will look like a brand new commit, which is visualized with an asterisk (\*) in the diagram above. It’s important to keep this in mind when working with public repositories.

### Usage

git commit --amend

Combine the staged changes with the previous commit and replace the previous commit with the resulting snapshot. Running this when there is nothing staged lets you edit the previous commit’s message without altering its snapshot.

### Discussion

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 little mistakes.

#### Don’t Amend Public Commits

On the [git reset](https://www.atlassian.com/git/tutorial/undoing-changes#!reset) page, we talked about how you should never reset commits that have been shared with other developers. The same goes for amending: never amend commits that have been pushed to a public repository.

Amended commits are actually entirely new commits, and the previous commit is removed from the project history. This has the same consequences as resetting a public snapshot. If you amend a commit that other developers have based their work on, it will look like the basis of their work vanished from the project history. This is a confusing situation for developers to be in and it’s complicated to recover from.

### Example

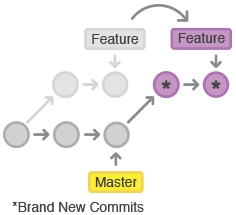
The following example demonstrates a common scenario in Git-based development. We edit a few files that we would like to commit in a single snapshot, but then we forget to add one of the files the first time around. Fixing the error is simply a matter of staging the other file and committing with the --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 editor will be populated with the message from the previous commit and including the --no-edit flag will allow you to make the ammendment to your commit without changing its commit message. You can change it if necessary, otherwise just save and close the file as usual. 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.

### The git rebase Command

Rebasing is the process of moving a branch to a new base commit. The general process can be visualized as the following:



From a content perspective, rebasing really is just moving a branch from one commit to another. But internally, Git accomplishes this by creating new commits and applying them to the specified base—it’s literally rewriting your project history. It’s very important to understand that, even though the branch looks the same, it’s composed of entirely new commits.

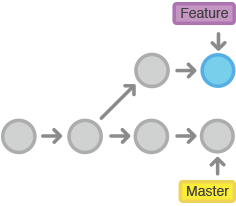
### Usage

git rebase <base>

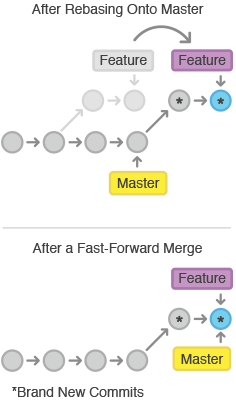
Rebase the current branch onto <base>, which can be any kind of commit reference (an ID, a branch name, a tag, or a relative reference to HEAD).

### Discussion

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:



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 master facilitates a fast-forward merge.



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.”

#### Don’t Rebase Public History

As we’ve discussed with git commit --amend and git reset, you should never rebase commits that have 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.

### Examples

The example below combines git rebase with git merge to maintain a linear project history. This is a quick and easy way to ensure that your merges will be fast-forwarded.

# Start a new feature  
git checkout -b new-feature master  
# Edit files  
git commit -a -m "Start developing a feature"

In the middle of our feature, we realize there’s a security hole in our project

# Create a hotfix branch based off of master  
git checkout -b hotfix master  
# Edit files  
git commit -a -m "Fix security hole"  
# Merge back into master  
git checkout master  
git merge hotfix  
git branch -d hotfix

After merging the hotfix into master, we have a forked project history. Instead of a plain git merge, we’ll integrate the feature branch with a rebase to maintain a linear history:

git checkout new-feature  
git rebase master

This moves new-feature to the tip of master, which lets us do a standard fast-forward merge from master:

git checkout master  
git merge new-feature

### The git rebase -i Command

Running git rebase with the -i flag 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. It’s like git commit --amend on steroids.

### Usage

git rebase -i <base>

Rebase the current branch onto <base>, but use an interactive rebasing session. 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.

### Discussion

Interactive rebasing gives you complete control over what your project history looks like. This affords a lot of freedom to developers, as it lets them commit a “messy” history while they’re focused on writing code, then go back and clean it up after the fact.

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.

### Examples

The example found below is an interactive adaptation of the one from the non-interactive git rebase page.

# Start a new feature  
git checkout -b new-feature master  
# Edit files  
git commit -a -m "Start developing a feature"  
# Edit more files  
git commit -a -m "Fix something from the previous commit"  
  
# Add a commit directly to master  
git checkout master  
# Edit files  
git commit -a -m "Fix security hole"  
  
# Begin an interactive rebasing session  
git checkout new-feature  
git rebase -i master

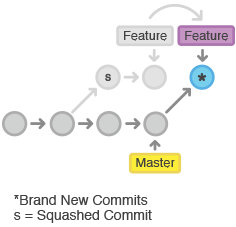
The last command will open an editor populated with the two commits from new-feature, along with some instructions:

pick 32618c4 Start developing a feature  
pick 62eed47 Fix something from the previous commit

You can change the pick commands before each commit to determine how it gets moved during the rebase. In our case, let’s just combine the two commits with a squash command:

pick 32618c4 Start developing a feature  
squash 62eed47 Fix something from the previous commit

Save and close the editor to begin the rebase. This will open another editor asking for the commit message for the combined snapshot. After defining the commit message, the rebase is complete and you should be able to see the squashed commit in your git log output. This entire process can be visualized as follows:



Note that the squashed commit has a different ID than either of the original commits, which tells us that it is indeed a brand new commit.

Finally, you can do a fast-forward merge to integrate the polished feature branch into the main code base:

git checkout master  
git merge new-feature

The real power of interactive rebasing can be seen in the history of the resulting master branch—the extra 62eed47 commit is nowhere to be found. 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.

### The git reflog Command

Git keeps track of updates to the tip of branches using a mechanism called reflog. This allows you to go back to changesets even though they are not referenced by any branch or tag. After rewriting history, the reflog contains information about the old state of branches and allows you to go back to that state if necessary.

### Usage

git reflog

Show the reflog for the local repository.

git reflog --relative-date

Show the reflog with relative date information (e.g. 2 weeks ago).

### Discussion

Every time the current HEAD gets updated (by switching branches, pulling in new  
changes, rewriting history or simply by adding new commits) a new entry will  
be added to the reflog.

### Example

To understand git reflog, let's run through an example.

0a2e358 HEAD@{0}: reset: moving to HEAD~2  
0254ea7 HEAD@{1}: checkout: moving from 2.2 to master  
c10f740 HEAD@{2}: checkout: moving from master to 2.2

The reflog above shows a checkout from master to the 2.2 branch and back. From there, there's a hard reset to an older commit. The latest activity is represented at the top labeled HEAD@{0}.

If it turns out that you accidentially moved back, the reflog will contain the commit master pointed to (0254ea7) before you accidentially dropped 2 commits.

git reset --hard 0254ea7

Using [git reset](https://www.atlassian.com/git/tutorial/undoing-changes#!reset) it is then possible to change master back to the commit it was before. This provides a safety net in case history was accidentially changed.

It's important to note that the reflog only provides a safety net if changes have been commited to your local repository and that it only tracks movements.

# Remote Git Repositories

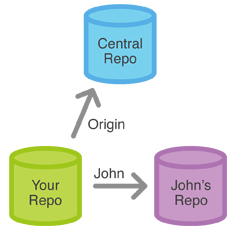
SVN uses a single central repository to serve as the communication hub for developers, and collaboration takes place by passing changesets between the developers’ working copies and the central repository. This is different from Git’s collaboration model, which gives every developer their own copy of the repository, complete with its own local history and branch structure. Users typically need to share a series of commits rather than a single changeset. Instead of committing a changeset from a working copy to the central repository, Git lets you share entire branches between repositories.

The commands presented below let you manage connections with other repositories, publish local history by "pushing" branches to other repositories, and see what others have contributed by "pulling" branches into your local repository.

### The git remote Command

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.

For example, the following diagram shows two remote connections from your repo into the central repo and another developer’s repo. Instead of referencing them by their full URLs, you can pass the origin and john shortcuts to other Git commands.



### Usage

git remote

List the remote connections you have to other repositories.

git remote -v

Same as the above command, but include the URL of each connection.

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-name> <new-name>

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

### Discussion

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.

#### The origin Remote

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.

#### Repository URLs

Git supports many ways to reference a remote repository. Two of the easiest ways to access a remote repo are via the HTTP and the SSH protocols. HTTP is an easy way to allow anonymous, read-only access to a repository. For example:

http://host/path/to/repo.git

But, it’s generally not possible to push commits to an HTTP address (you wouldn’t want to allow anonymous pushes anyways). For read-write access, you should use SSH instead:

ssh://user@host/path/to/repo.git

You’ll need a valid SSH account on the host machine, but other than that, Git supports authenticated access via SSH out of the box.

### Examples

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.

### The git fetch Command

The git fetch command imports commits from a remote repository into your local repo. The resulting commits are stored as remote branches instead of the normal local branches that we’ve been working with. This gives you a chance to review changes before integrating them into your copy of the project.

### Usage

git fetch <remote>

Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

git fetch <remote> <branch>

Same as the above command, but only fetch the specified branch.

### Discussion

Fetching is what you do when you want to see what everybody else has been working on. Since fetched content is represented as a remote branch, it has absolutely no affect on your local development work. This makes fetching a safe way to review commits before integrating them with your local repository. It’s similar to svn update in that it lets you see how the central history has progressed, but it doesn’t force you to actually merge the changes into your repository.

#### Remote Branches

Remote branches are just like local branches, except they represent commits from somebody else’s repository. You can check out a remote branch just like a local one, but this puts you in a detached HEAD state (just like checking out an old commit). You can think of them as read-only branches. To view your remote branches, simply pass the -r flag to the git branchcommand. Remote branches are prefixed by the remote they belong to so that you don’t mix them up with local branches. For example, the next code snippet shows the branches you might see after fetching from the origin remote:

git branch -r  
# origin/master  
# origin/develop  
# origin/some-feature

Again, you can inspect these 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. So, unlike SVN, synchronizing your local repository with a remote repository is actually a two-step process: fetch, then merge. The [git pull](https://www.atlassian.com/git/tutorial/remote-repositories#!pull) command is a convenient shortcut for this process.

### Examples

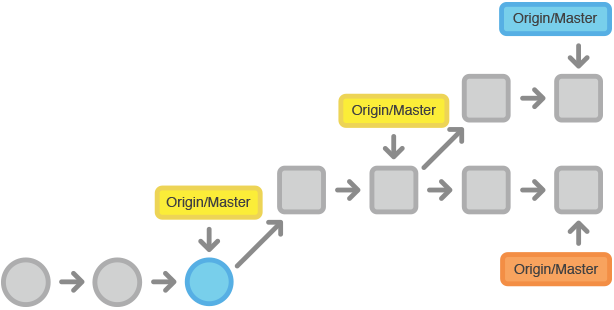
This example walks through the typical workflow for synchronizing your local repository with the central repository's master branch.

git fetch origin

This will display the branches that were downloaded:

a1e8fb5..45e66a4 master -> origin/master  
a1e8fb5..9e8ab1c develop -> origin/develop  
\* [new branch] some-feature -> origin/some-feature

The commits from these new remote branches are shown as squares instead of circles in the diagram below. As you can see, git fetch gives you access to the entire branch structure of another repository.



To see what commits have been added to the upstream master, you can run a git log usingmaster branch with the following commands:

git checkout master  
git merge origin/master

The origin/master and master branches now point to the same commit, and you are synchronized with the upstream developments.

### The git pull Command

Merging upstream changes into your local repository is a common task in Git-based collaboration workflows. We already know how to do this with [git fetch](https://www.atlassian.com/git/tutorial/remote-repositories#!fetch) followed by [git merge](https://www.atlassian.com/git/tutorial/git-branches#!merge), but git pull rolls this into a single command.

### Usage

git pull <remote>

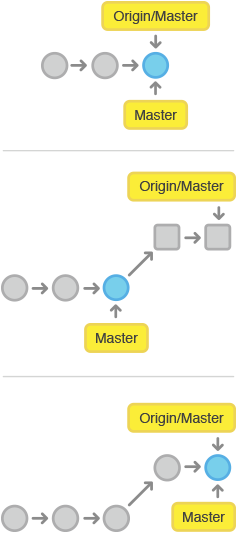
Fetch the specified remote’s copy of the current branch and immediately merge it into the local copy. This is the same as git fetch <remote> followed by git merge origin/<current-branch>.

git pull --rebase <remote>

Same as the above command, but instead of using git merge to integrate the remote branch with the local one, use git rebase.

### Discussion

You can think of git pull as Git's version of svn update. It’s an easy way to synchronize your local repository with upstream changes. The following diagram explains each step of the pulling process.



You start out thinking your repository is synchronized, but then git fetch reveals that origin's version of master has progressed since you last checked it. Then git merge immediately integrates the remote master into the local one:

#### Pulling via Rebase

The --rebase option can be used to ensure a linear history by preventing unnecessary merge commits. Many developers prefer rebasing over merging, since it’s like saying, "I want to put my changes on top of what everybody else has done." In this sense, using git pull with the --rebase flag is even more like svn update than a plain git pull.

In fact, pulling with --rebase is such a common workflow that there is a dedicated configuration option for it:

git config --global branch.autosetuprebase always

After running that command, all git pull commands will integrate via git rebase instead of git merge.

### Examples

The following example demonstrates how to synchronize with the central repository's master branch:

git checkout master  
git pull --rebase origin

This simply moves your local changes onto the top of what everybody else has already contributed.

### The git push Command

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 remote branches, pushing exports commits to local branches. This has the potential to overwrite changes, so you need to be careful how you use it. These issues are discussed below.

### Usage

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. 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.

git push <remote> --force

Same as the above command, but force 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.

git push <remote> --all

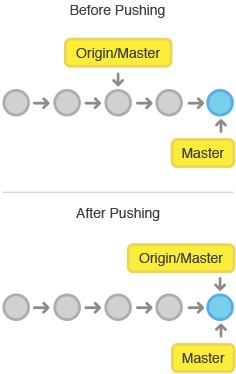
Push all of your local branches to the specified remote.

git push <remote> --tags

Tags are not automatically pushed when you push a branch or use the --all option. The --tags flag sends all of your local tags to the remote repository.

### Discussion

The most common use case for git push is to publish your local changes to a central repository. After you’ve accumulated several local commits and are ready to share them with the rest of the team, you (optionally) clean them up with an interactive rebase, then push them to the central repository.



The above diagram shows what happens when your local master has progressed past the central repository’s master and you publish changes by running git push origin master. Notice how git push is essentially the same as running git merge master from inside the remote repository.

#### Force Pushing

Git prevents you from overwriting the central repository’s history by refusing push requests when they result in a non-fast-forward merge. So, 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. This is similar to how SVN makes you synchronize with the central repository via svn update before committing a changeset.

The --force flag overrides this behavior and 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.

#### Only Push to Bare Repositories

In addition, you should only push to repositories that have been created with the --bare flag. Since pushing messes with the remote branch structure, it’s important to never push to another developer’s repository. But because bare repos don’t have a working directly, it’s impossible to interrupt anybody’s developments.

### Examples

The following example describes one of the standard methods for publishing local contributions to the central repository. First, it makes sure your local master is up-to-date by fetching the central repository’s copy and rebasing your changes on top of them. The interactive rebase is also a good opportunity to clean up your commits before sharing them. Then, the git push command sends all of the commits on your local master to the central repository.

git checkout master  
git fetch origin master  
git rebase -i origin/master  
# Squash commits, fix up commit messages etc.  
git push origin master

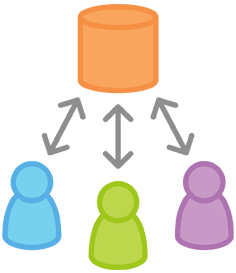
Since we already made sure the local master was up-to-date, this should result in a fast-forward merge, and git push should not complain about any of the non-fast-forward issues discussed above.

# Git Workflows

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 enterprise 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.

## Centralized Workflow



Transitioning to a distributed version control system may seem like a daunting task, but you don’t have to change your existing workflow to take advantage of Git. Your team can develop projects in the exact same way as they do with Subversion.

However, using Git to power your development workflow presents a few advantages over SVN. First, it gives every developer their own local copy of the entire project. This isolated environment lets each developer work independently of all other changes to a project—they can add commits to their local repository and completely forget about upstream developments until it's convenient for them.

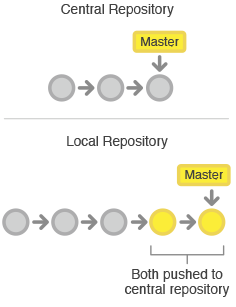
Second, it gives you access to Git’s robust branching and merging model. Unlike SVN, Git branches are designed to be a fail-safe mechanism for integrating code and sharing changes between repositories.

## How It Works

Like Subversion, the Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of trunk, 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.

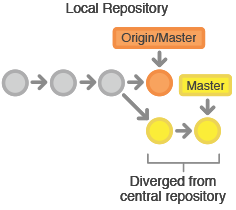
Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes as they would with SVN; however, these new commits are storedlocally—they’re completely isolated from the central repository. This lets developers defer synchronizing upstream until they’re at a convenient break point.

To publish changes to the official project, developers “push” their local master branch to the central repository. This is the equivalent of svn commit, except that it adds all of the local commits that aren’t already in the central master branch.



### Managing Conflicts

The central repository represents the official project, so its commit history should be treated as sacred and immutable. If a developer’s local commits diverge from the central repository, Git will refuse to push his changes because this would overwrite official commits.



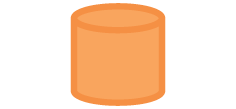
Before the developer can publish his feature, he needs to fetch the updated central commits and rebase his changes on top of them. This is like saying, “I want to add my changes to what everyone else has already done.” The result is a perfectly linear history, just like in traditional SVN workflows.

If local changes directly conflict with upstream commits, Git will pause the rebasing process and give you a chance to manually resolve the conflicts. The nice thing about Git is that it uses the same [git status](https://www.atlassian.com/git/tutorial/git-basics#!status) and [git add](https://www.atlassian.com/git/tutorial/git-basics#!add) commands for both generating commits and resolving merge conflicts. This makes it easy for new developers to manage their own merges. Plus, if they get themselves into trouble, Git makes it very easy to abort the entire rebase and try again (or go find help).

## Example

Let’s take a step-by-step look 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.

### Someone initializes the central repository



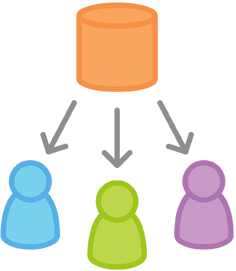
First, someone needs to create the central repository on a server. If it’s a new project, you can initialize an empty repository. Otherwise, you’ll need to import an existing Git or SVN repository.

Central repositories should always be bare repositories (they shouldn’t have a working directory), which can be created as follows:

ssh user@host  
git init --bare /path/to/repo.git

Be sure to use a valid SSH username for user, the domain or IP address of your server for host, and the location where you'd like to store your repo for /path/to/repo.git. Note that the .git extension is conventionally appended to the repository name to indicate that it’s a bare repository.

### Everybody clones the central repository

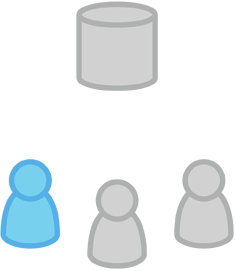


Next, each developer creates a local copy of the entire project. This is accomplished via the [git clone](https://www.atlassian.com/git/tutorial/git-basics#!clone) command:

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

When you clone a repository, Git automatically adds a shortcut called origin that points back to the “parent” repository, under the assumption that you'll want to interact with it further on down the road.

### John works on his feature

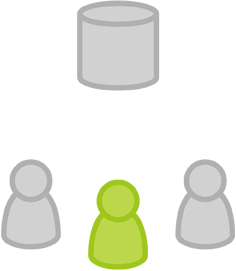


In his local repository, John can develop features using the standard Git commit process: edit, stage, and commit. If you’re not familiar with the staging area, it’s a way to prepare a commit without having to include every change in the working directory. This lets you create highly focused commits, even if you’ve made a lot of local changes.

git status # View the state of the repo  
git add # Stage a file  
git commit # Commit a file

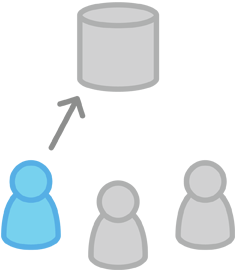
Remember that since these commands create local commits, John can repeat this process as many times as he wants without worrying about what’s going on in the central repository. This can be very useful for large features that need to be broken down into simpler, more atomic chunks.

### 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

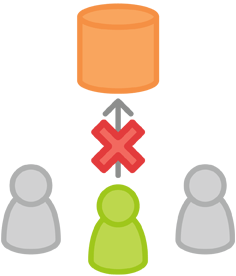


Once John finishes his feature, he should publish his local commits to the central repository so other team members can access it. He can do this with the [git push](https://www.atlassian.com/git/tutorial/remote-repositories#!push) command, like so:

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:

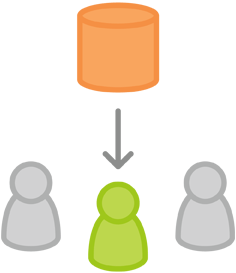
git push origin master

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'  
hint: Updates were rejected because the tip of your current branch is behind  
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')  
hint: before pushing again.  
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

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.

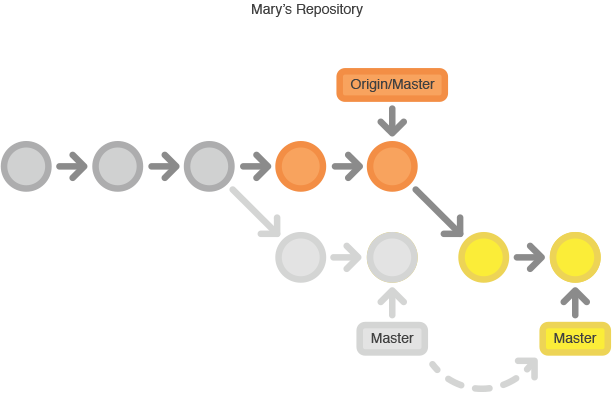
### Mary rebases on top of John’s commit(s)



Mary can use [git pull](https://www.atlassian.com/git/tutorial/remote-repositories#!pull) to incorporate upstream changes into her repository. This command is sort of like svn update—it pulls the entire upstream commit history into Mary’s local repository and tries to integrate it with her local commits:

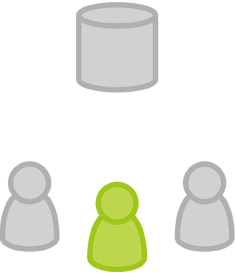
git pull --rebase origin master

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:



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.

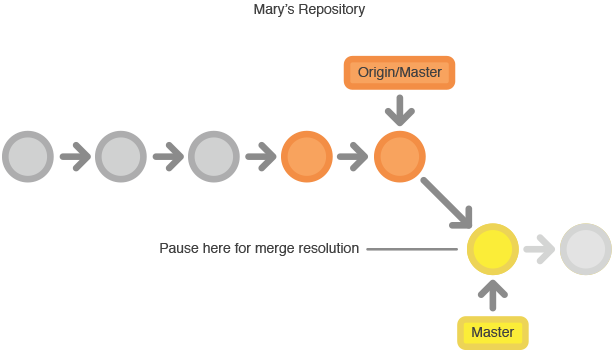
### Mary resolves a merge conflict



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](https://www.atlassian.com/git/tutorial/git-basics#!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](https://www.atlassian.com/git/tutorial/rewriting-git-history#!rebase) do the rest:

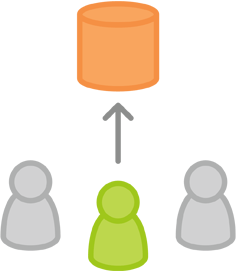
git add <some-file>   
git rebase --continue </some-file>

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 before you ran [git pull --rebase](https://www.atlassian.com/git/tutorial/remote-repositories#!pull):

git rebase --abort

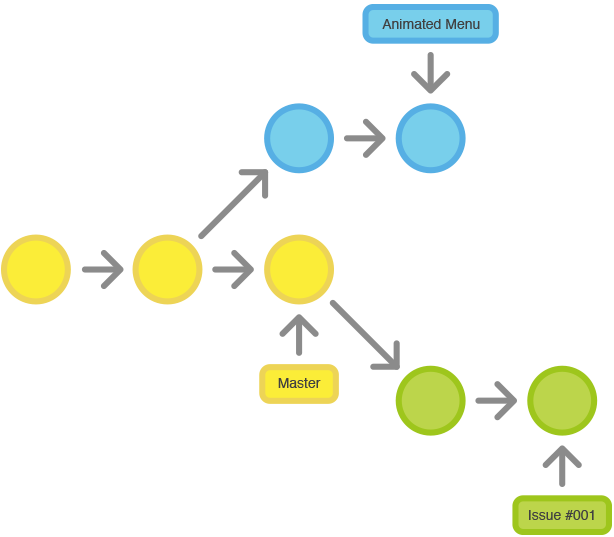
### Mary successfully publishes her feature



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

git push origin master

## Feature Branch Workflow



Once you've got the hang of the [Centralized Workflow](https://www.atlassian.com/git/workflows#!workflow-centralized), adding feature branches to your development process is an easy way to encourage collaboration and streamline communication between developers.

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.

## How It Works

The Feature Branch Workflow still uses a central repository, and master still represents the official project history. But, 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 just as they did in the Centralized Workflow.

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.

### 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.

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.

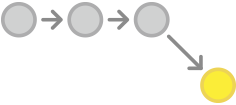
Once a pull request is accepted, the actual act of publishing a feature is much the same as in the Centralized Workflow. First, you need to make sure your local master is synchronized with the upstream master. Then, you merge the feature branch into master and push the updated master back to the central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

Pull requests can be facilitated by product respitory management solutions like [Bitbucket](http://bitbucket.org/) or[Stash](http://www.atlassian.com/stash). View the Stash [pull requests documentation](https://confluence.atlassian.com/display/STASH/Using+pull+requests+in+Stash) for an example.

## Example

The example included below demonstrates a pull request as a form of code review, but remember that they can serve many other purposes.

### Mary begins a new feature



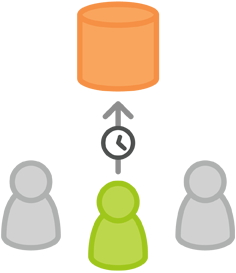
Before she starts developing a feature, Mary needs an isolated branch to work on. She can[request a new branch](https://www.atlassian.com/git/tutorial/git-branches#!checkout) with the following command:

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

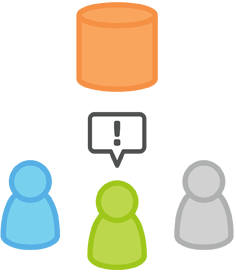


Mary [adds a few commits to her feature](https://www.atlassian.com/git/tutorial/git-basics#!commit) over the course of the morning. Before she leaves for lunch, it’s a good idea to [push her feature branch up to the central repository](https://www.atlassian.com/git/tutorial/remote-repositories#!push). 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

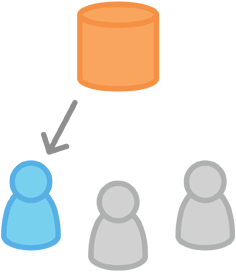


When Mary gets back from lunch, she completes her feature. [Before merging it into master](https://www.atlassian.com/git/tutorial/git-branches#!merge), she needs to file a pull request letting the rest of the team know she's done. But first, she should make sure the central repository has her most recent commits:

git push

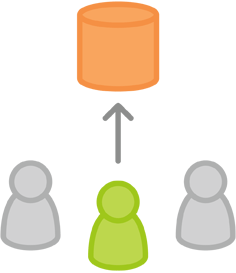
Then, she files the pull request in her Git GUI asking to merge marys-feature into master, and team members will be notified automatically. The great thing about pull requests is that they show comments right next to their related commits, so it's easy to ask questions about specific changesets.

### 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.

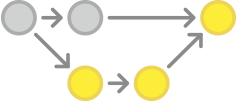
### Mary makes the changes



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

First, whoever’s performing the merge needs to check out their master branch and make sure it’s up to date. Then, git pull origin marys-feature merges the central repository’s copy of marys-feature. You could also use a simple git merge marys-feature, but the command shown above makes sure you’re always pulling the most up-to-date version of the feature branch. Finally, the updated master needs to get pushed back to origin.

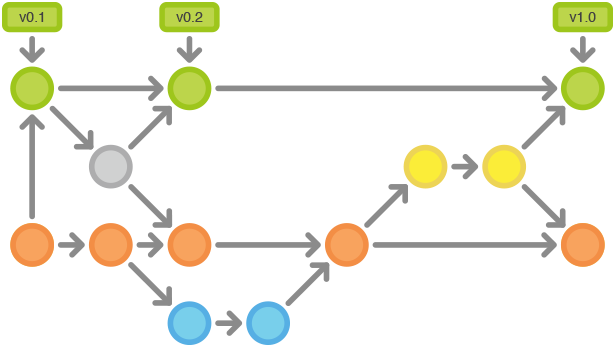
This process often results in a merge commit. Some developers like this because it’s like a symbolic joining of the feature with the rest of the code base. But, if you’re partial to a linear history, it’s possible to rebase the feature onto the tip of master before executing the merge, resulting in a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

## Gitflow Workflow



The [Gitflow Workflow](http://nvie.com/posts/a-successful-git-branching-model/) section below is derived from Vincent Driessen at [nvie](http://nvie.com/).

The Gitflow Workflow defines a strict branching model designed around the project release. While somewhat more complicated than the [Feature Branch Workflow](https://www.atlassian.com/git/workflows#!workflow-release-cycle), this provides a robust framework for managing larger projects.

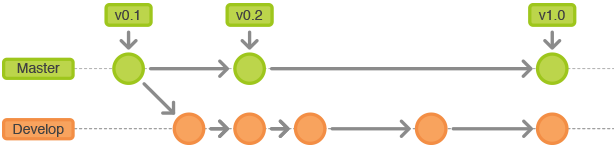
This workflow doesn’t add any new concepts or commands beyond what’s required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact. In addition to feature branches, it uses individual branches for preparing, maintaining, 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.

## How It Works

The Gitflow Workflow still uses a central repository as the communication hub for all developers. And, as in the [other workflows](https://www.atlassian.com/git/workflows), developers work locally and push branches to the central repo. The only difference is the branch structure of the project.

### Historical 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 masterbranch with a version number.



The rest of this workflow revolves around the distinction between these two branches.

### Feature Branches

Each new feature should reside in its own branch, which can be [pushed to the central repository for backup/collaboration](https://www.atlassian.com/git/tutorial/remote-repositories#!push). 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](https://www.atlassian.com/git/tutorial/git-branches#!merge). 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.

### 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 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).

Common conventions:

* branch off: develop
* merge into: master
* naming convention: release-\* or release/\*

### Maintenance Branches



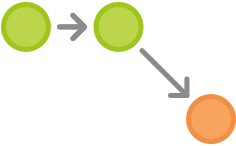
Maintenance or “hotfix” branches are used to quickly patch production releases. This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master should be tagged with an updated 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.

## Example

The example below demonstrates how this workflow can be used to manage a single release cycle. We’ll assume you have already created a central repository.

### Create a develop branch



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](https://www.atlassian.com/git/tutorial/git-branches#!branch) and push it to the server:

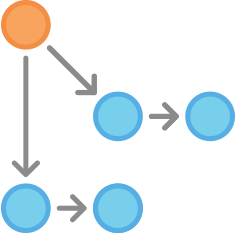
git branch develop  
git push -u origin develop

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](https://www.atlassian.com/git/tutorial/git-basics#!clone) and create a tracking branch for develop:

git clone ssh://user@host/path/to/repo.git  
git checkout -b develop origin/develop

Everybody now has a local copy of the historical branches set up.

### Mary and John begin new features



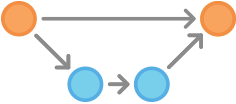
Our example starts with John and Mary working on separate features. They both need to create separate branches for their respective features. Instead of basing it on master, they should both [base their feature branches on develop](https://www.atlassian.com/git/tutorial/git-branches#!checkout):

git checkout -b some-feature develop

Both of them add commits to the feature branch in the usual fashion: edit, stage, commit:

git status  
git add <some-file>  
git commit

### Mary finishes her feature

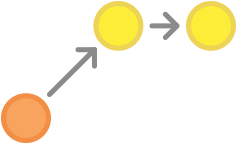


After adding a few commits, Mary decides her feature is ready. If her team is using pull requests, this would be an appropriate time to open one asking to merge her feature into develop. Otherwise, she can merge it into her local develop and push it to the central repository, like so:

git pull develop  
git checkout develop  
git merge some-feature  
git push  
git branch -d some-feature

The first command makes sure the develop branch is up to date before trying to merge in the feature. Note that features should never be merged directly into master. Conflicts can be resolved in the same way as in the [Centralized Workflow](https://www.atlassian.com/git/workflows#!workflow-svn).

### Mary begins to prepare a release



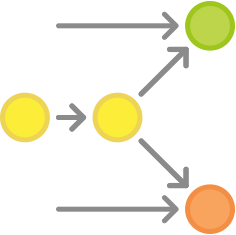
While John is still working on his feature, Mary starts to prepare the first official release of the project. Like feature development, she uses a new branch to encapsulate the release preparations. This step is also where the release’s version number is established:

git checkout -b release-0.1 develop

This branch is a place to clean up the release, test everything, update the documentation, and do any other kind of preparation for the upcoming release. It’s like a feature branch dedicated to polishing the release.

As soon as Mary creates this branch and pushes it to the central repository, the release is feature-frozen. Any functionality that isn’t already in develop is postponed until the next release cycle.

### Mary finishes the release



Once the release is ready to ship, Mary merges it into master and develop, then deletes the release branch. 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. Again, if Mary’s organization stresses code review, this would be an ideal place for a pull request.

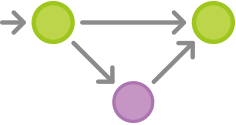
git checkout master  
git merge release-0.1  
git push  
git checkout develop  
git merge release-0.1  
git push  
git branch -d release-0.1

Release branches act as a buffer between feature development (develop) and public releases (master). Whenever you merge something into master, you should tag the commit for easy reference:

git tag -a 0.1 -m "Initial public release"  
git push --tags

Git comes with several hooks, which are scripts that execute whenever a particular event occurs within a repository. It’s possible to configure a hook to automatically build a public release whenever you push the master branch to the central repository or push a tag.

### End-user discovers a bug



After shipping the release, Mary goes back to developing features for the next release with John. That is, until an end-user opens a ticket complaining about a bug in the current release. To address the bug, Mary (or John) creates a maintenance branch off of master, fixes the issue with as many commits as necessary, then merges it directly back into master.

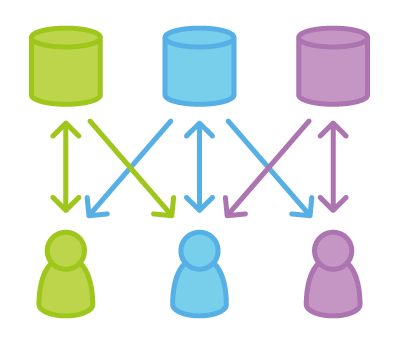
git checkout -b issue-#001 master  
# Fix the bug  
git checkout master  
git merge issue-#001  
git push

Like release branches, maintenance branches contain important updates that need to be included in develop, so Mary needs to perform that merge as well. Then, she’s free to [delete the branch](https://www.atlassian.com/git/tutorial/git-branches#!branch):

git checkout develop  
git merge issue-#001  
git push  
git branch -d issue-#001

## Forking Workflow

The Forking Workflow is fundamentally different than the other workflows discussed in this tutorial. Instead of using a single server-side repository to act as the “central” codebase, it gives everydeveloper a 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 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 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.

## How It Works

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 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](https://www.atlassian.com/git/tutorial/git-basics#!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 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.

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 his localmaster branch](https://www.atlassian.com/git/tutorial/git-branches#!merge), then [pushes](https://www.atlassian.com/git/tutorial/remote-repositories#!push) 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.

### The Official Repository

It’s important to understand that the notion of an “official” repository in the Forking Workflow is merely a convention. From a technical standpoint, Git doesn’t see any difference between each developer’s public repository and the official one. In fact, the only thing that makes the official repository so official is that it’s the public repository of the project maintainer.

### Branching in the Forking Workflow

All of these personal public repositories are really just a convenient way to share branches with other developers. Everybody should still be using branches to isolate individual features, just like in the [Feature Branch Workflow](https://www.atlassian.com/git/workflows#!workflow-feature-branch) and the [Gitflow Workflow](https://www.atlassian.com/git/workflows" \l "!workflow-gitflow). The only difference is how those branches get shared. In the Forking Workflow, they are pulled into another developer’s local repository, while in the Feature Branch and Gitflow Workflows they are pushed to the official repository.

## Example

### The project maintainer initializes the official repository

Forking Workflow: Shared Repository

As with any Git-based project, the first step is to create an official repository on a server accessible to all of the team members. Typically, this repository will also serve as the public repository of the project maintainer.

[Public repositories should always be bare](https://www.atlassian.com/git/tutorial/git-basics#!init), regardless of whether they represent the official codebase or not. So, the project maintainer should run something like the following to set up the official repository:

ssh user@host  
git init --bare /path/to/repo.git

Bitbucket and Stash also provide a convenient GUI alternative to the above commands. This is the exact same process as setting up a central repository for the other workflows in this tutorial. The maintainer should also push the existing codebase to this repository, if necessary.

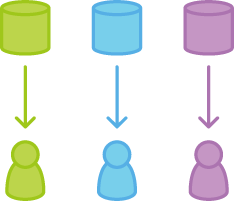
### Developers fork the official repository

Forking Workflow: Forking the official repository.

Next, all of the other developers need to fork this official repository. It’s possible to do this by[SSH](https://confluence.atlassian.com/display/BITBUCKET/Set+up+SSH+for+Git)’ing into the server and running git clone to copy it to another location on the server—yes, forking is basically just a server-side clone. But again, Bitbucket and Stash let developers fork a repository with the click of a button.

After this step, every developer should have their own server-side repository. Like the official repository, all of these should be bare repositories.

### Developers clone their forked repositories



Next each developer needs to clone their own public repository. They can do with the familiar git clone command.

Our example assumes the use of Bitbucket to host these repositories. Remember, in this situation, each developer should have their own Bitbucket account and they should clone their server-side repository using:

git clone https://user@bitbucket.org/user/repo.git

Whereas the other workflows in this tutorial 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.

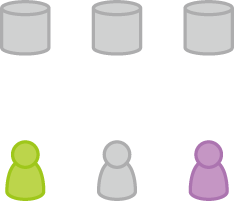
git remote add upstream https://bitbucket.org/maintainer/repo

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, like so:

git remote add upstream https://user@bitbucket.org/maintainer/repo.git

This requires users to supply a valid password before cloning or pulling from the official codebase.

### Developers work on their features



In the local repositories that they just cloned, developers can edit code, [commit changes](https://www.atlassian.com/git/tutorial/git-basics#!commit), and[create branches](https://www.atlassian.com/git/tutorial/git-branches#!branch) just like they did in the other workflows:

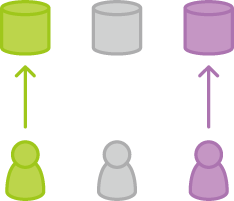
git checkout -b some-feature  
# Edit some code  
git commit -a -m "Add first draft of some feature"

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](https://www.atlassian.com/git/tutorial/remote-repositories#!pull):

git pull upstream master

Since developers should be working in a dedicated feature branch, this should generally [result in a fast-forward merge](https://www.atlassian.com/git/tutorial/git-branches#!merge).

### Developers publish their features



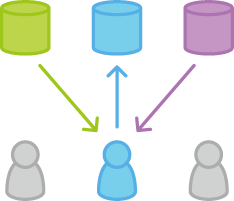
Once a developer is ready to share their new feature, they need to do two things. First, they have to make their contribution accessible to other developers by pushing it to their public repository. Their origin remote should already be set up, so all they should have to do is the following:

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 Stash provide a “[Pull request](https://confluence.atlassian.com/display/STASH/Using+pull+requests+in+Stash)” 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.

### The project maintainer integrates their features



When the project maintainer receives the pull request, their job is to decide whether or not to integrate it into the official codebase. They can do this in one of two ways:

1) Inspect the code directly in the pull request

2) Pull the code into their local repository and manually merge it

The first option is simpler, as it lets the maintainer view a diff of the changes, comment on it, and perform the merge via a graphical user interface. However, the second option is necessary if the pull request results in a merge conflict. In this case, the maintainer needs to [fetch](https://www.atlassian.com/git/tutorial/remote-repositories#!fetch) the feature branch from the developer’s server-side repository, merge it into their local master branch, and resolve any conflicts:

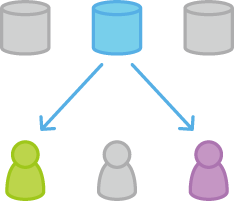
git fetch https://bitbucket.org/user/repo feature-branch  
# Inspect the changes  
git checkout master  
git merge FETCH\_HEAD

Once the changes are integrated into their local master, the maintainer needs to push it to the official repository on the server so that other developers can access it:

git push origin master

Remember that the maintainer's origin points to their public repository, which also serves as the official codebase for the project. The developer's contribution is now fully integrated into the project.

### Developers synchronize with the official repository



Since the main codebase has moved forward, other developers should synchronize with the official repository:

git pull upstream master