# Git Setup Manual For Responsive And Native Applications

[Git Setup Manual For Responsive And Native Applications 1](#_Toc476131686)

[Git Command Line Tool Installation 2](#_Toc476131687)

[How To Set A Repository ? 2](#_Toc476131688)

[git init 2](#_Toc476131689)

[git clone 3](#_Toc476131690)

[How To Save Changes to the repository ? 4](#_Toc476131691)

[git add 4](#_Toc476131692)

[git commit 5](#_Toc476131693)

[git stash 7](#_Toc476131694)

[How To Inspect A Repository ? 8](#_Toc476131695)

[git status 8](#_Toc476131696)

[git log 9](#_Toc476131697)

[Collaboration @ Git Hub 11](#_Toc476131698)

[How to Sync ? 11](#_Toc476131699)

[git remote 11](#_Toc476131700)

[git push 13](#_Toc476131701)

[git fetch 14](#_Toc476131702)

[git pull 16](#_Toc476131703)

[Git Branching 17](#_Toc476131704)

[git branch - Creating Branches 17](#_Toc476131705)

[git checkout 19](#_Toc476131706)

[git merge 21](#_Toc476131707)

[How To Undo Changes To The repository? 25](#_Toc476131708)

[git checkout 25](#_Toc476131709)

[git revert 27](#_Toc476131710)

[git reset 27](#_Toc476131711)

[git clean 27](#_Toc476131712)

[How To Rewrite History? 28](#_Toc476131713)

[git commit –amend 28](#_Toc476131714)

[git rebase 30](#_Toc476131715)

[git rebase –i 32](#_Toc476131716)

[git reflog 33](#_Toc476131717)

# Git Command Line Tool Installation

1.Download git bash from <https://git-scm.com/downloads> - Git Bash

[ If already installed ,can go with the same ]

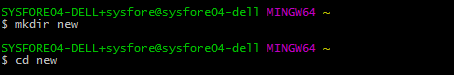
2.Click on Git Bash command line tool  ,Open Git Bash.

# How To Set A Repository ?

# git init

1. Create local repository for your project using **mkdir <repositoryname>** .

2. Change the path to newly created repository using **cd <repositoryname>**



3. Transform the current directory into a Git repository using **git init** command .

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

Also ,you can also create git repository directly using **git init <directory>** avoiding steps 1 and 2.



4. Add the files in your new local repository. This stages them for the first commit.

Here you add your Responsive , native project folders and files

Go to the repository path created in your windows or Linux system and copy the project files

to this repository.

# git clone

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.

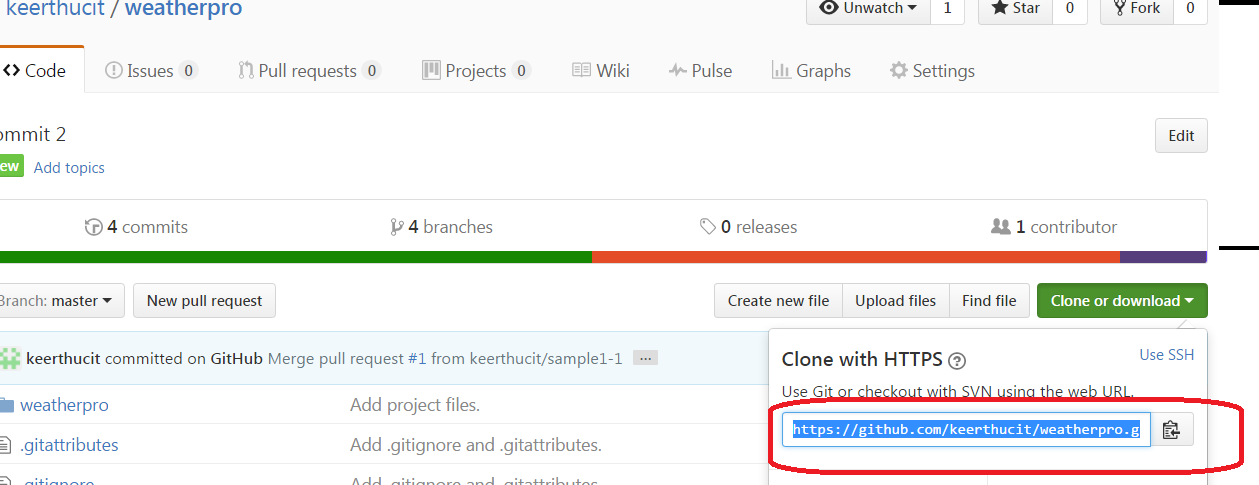
**git clone <repo>**

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>**

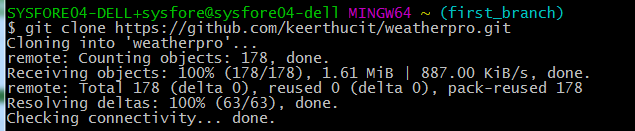
Clone the repository located at <repo> into the folder called <directory> on the local machine.

1.Copy the url of repository to be cloned.

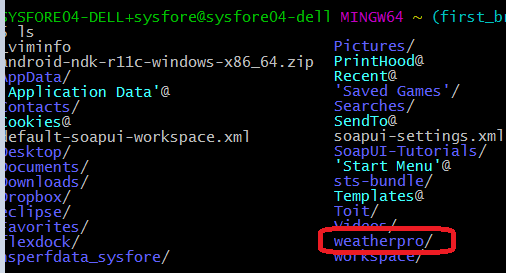


2.In the git bash command line tool paste the url along with git clone command .

**git clone <url>**



3.Check for the local repository cloned / copied in the same name as remote repository using ls command.



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/tutorials/setting-up-a-repository/git-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, [push](https://www.atlassian.com/git/tutorials/syncing/git-push) or [pull](https://www.atlassian.com/git/tutorials/syncing/git-pull) commits from one repository to another .git config

The **git config** command configures 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.

**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.

# How To Save Changes to the repository ?

# git add

The **git add** <file> command adds a change in the working directory to the staging area. It tells Git to include updates to a particular file in the next commit. However, git add doesn't really affect the repository in any significant way—changes are not actually recorded until you run [**git commit**](https://www.atlassian.com/git/tutorials/saving-changes/git-commit)**.**

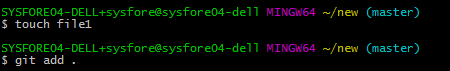
In conjunction with these commands, you'll also need [**git status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-status) to view the state of the working directory and the staging area.

Use **git add .**to add all the files in the current working directory to the staging area.

**git add <directory>** to 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 chunk of changes and prompt you for a command. Use y to stage the chunk, n to ignore the chunk, s to split it into smaller chunks, e to manually edit the chunk, and q to exit.

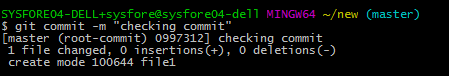


# git commit

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.

**git commit -m "<message>"**

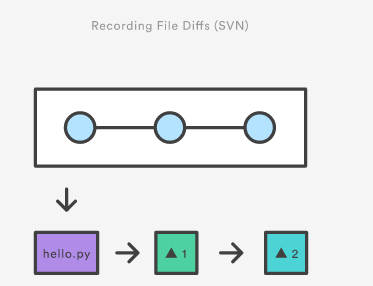


Snapshots are always committed to the local repository. This is fundamentally different from SVN, wherein the working copy is committed to the central repository. In contrast, Git doesn’t force you to interact with the central repository until you’re ready. Just as the staging area is a buffer between 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.

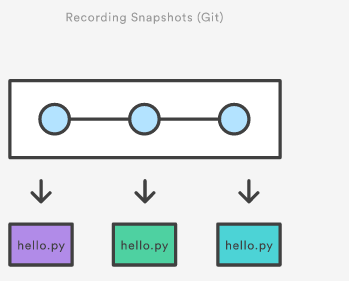
**Snapshots, Not Differences**

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.

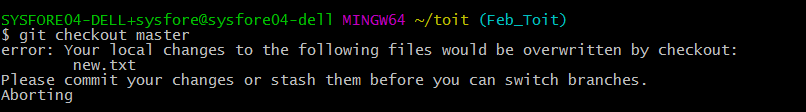


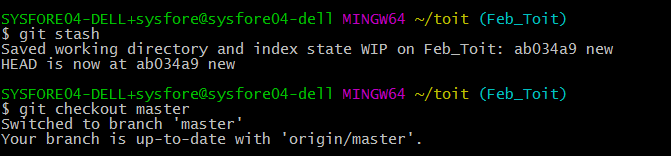
This makes many Git operations much faster than SVN, since a particular version of a file doesn’t have to be “assembled” from its diffs—the complete revision of each file is immediately available from Git's internal database.

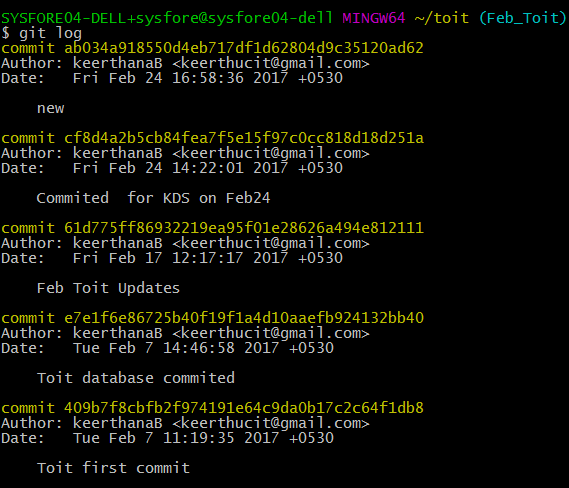
Git's snapshot model has a far-reaching impact on virtually every aspect of its version control model, affecting everything from its branching and merging tools to its collaboration workflows.

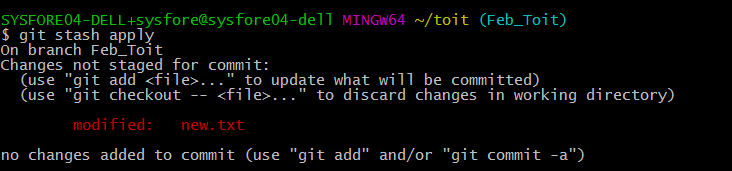


### git stash





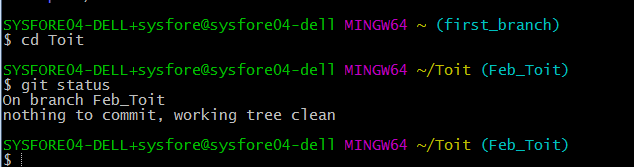




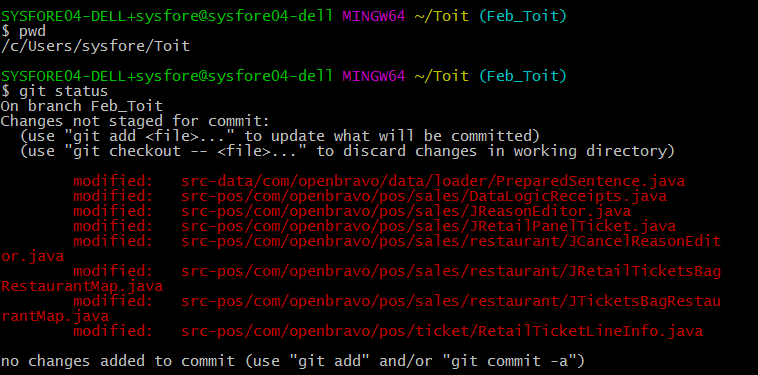
# How To Inspect A Repository ?

# git status

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



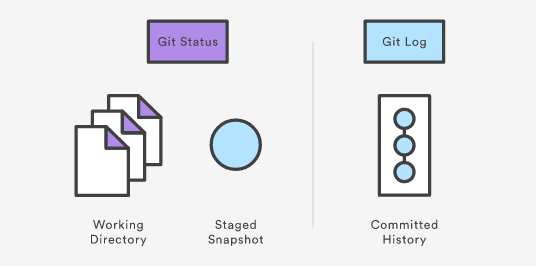
After making changes,

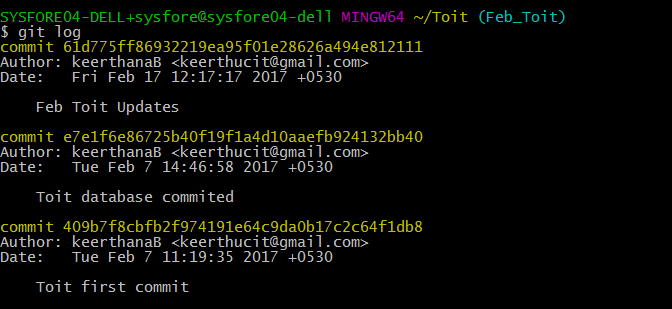


It simply shows what is going on with git add and git commit.

# git log

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





**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 <pattern> 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.

# Collaboration @ Git Hub

# How to Sync ?

# git remote

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.

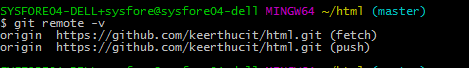
The commands presented below let you manage connections with other repositories, publish local history by "pushing" branches to other repositories, and see what others have contributed by "pulling" branches into your local repository.

**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>.

In the Command prompt, [add the URL for the remote repository](https://help.github.com/articles/adding-a-remote) where your local repository will be pushed.

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.

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.

# git push

Pushing is to transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches. This has the potential to overwrite changes, so need to be careful while using it.

**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**

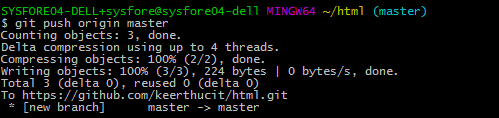
Same as the above command, but force the push even if it results in a non-fast-forward merge. Do not use the --force flag unless you’re absolutely sure you know what you’re doing.

**git push <remote> --all**

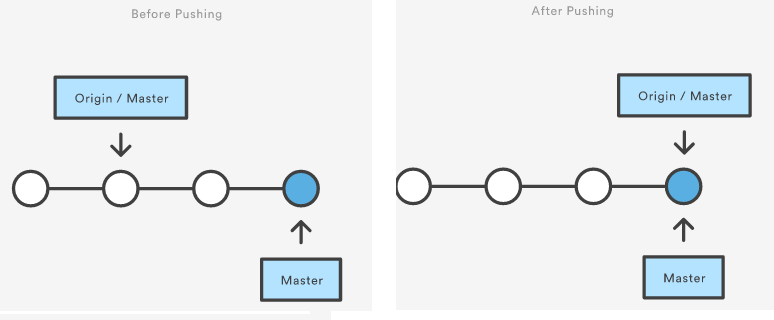
Push all of your local branches to the specified remote.

**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.



The most common use case for git push is to publish our local changes to a central repository. After we’ve accumulated several local commits and are ready to share them with the rest of the team, (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.

# git fetch

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.

**git fetch <remote>**

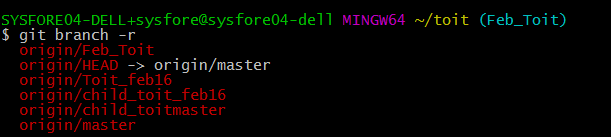
Fetch all of the branches from the repository. This also downloads all of the required commits and files from the other repository.

**git fetch <remote> <branch>**

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 effect 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 branch command. 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:



Again, we can inspect these branches with the usual **git checkout and git log** commands. If we approve the changes a remote branch contains, you can merge it into a local branch with a normal git merge. So, unlike SVN, synchronizing our local repository with a remote repository is actually a two-step process: fetch, then merge. The [**git pull**](https://www.atlassian.com/git/tutorials/syncing/git-pull)command is a convenient shortcut for this process.

This example walks through the typical workflow for synchronizing our 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 using origin/master as a filter

git log --oneline master..origin/master

To approve the changes and merge them into your local master branch with the following commands:

**git checkout master**

**git log origin/master**

Then we can use git merge origin/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.

# git pull

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/tutorials/syncing/git-fetch) followed by [**git merge**](https://www.atlassian.com/git/tutorials/using-branches/git-merge), but **git pull** rolls this into a single command.

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.**

We 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.**

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.

# Git Branching

# git branch - Creating Branches

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. S

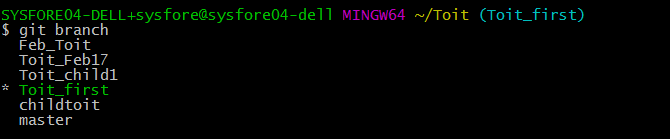
Start with a repository that looks like the figure in left and then create a branch using the following command , **git branch crazy-experiment**

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



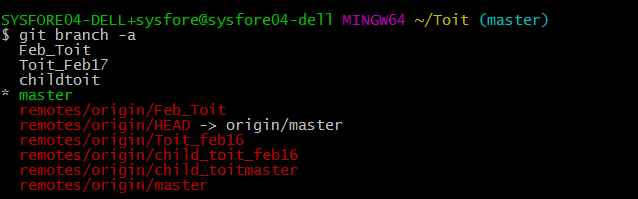
**git branch**

List all of the branches in local repository.



**git branch -a**

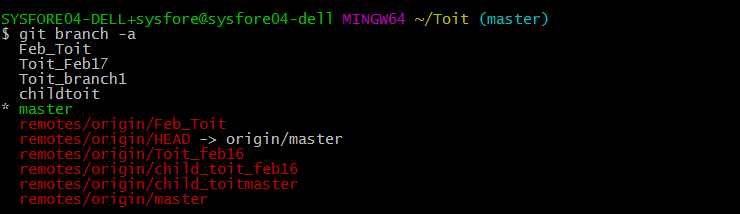
List all of the branches in local as well as remote repository.



**git branch <branch>**



Look for the branch name created by listing out the branchs using **git branch –a**



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

**git branch -d <branch>**

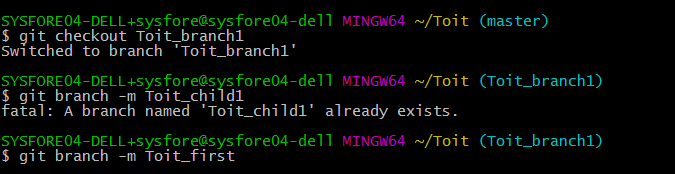
Delete the specified branch. This is a “safe” operation in that Git prevents you from deleting the branch if it has unmerged changes.

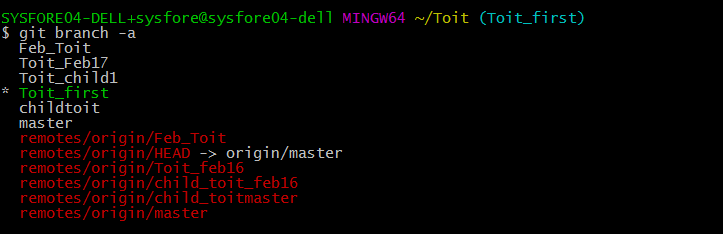
**git branch -D <branch>**

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> after checkout



List out the branches using **git branch –a** and look for the renamed branch

# git checkout

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/tutorials/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.

**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.

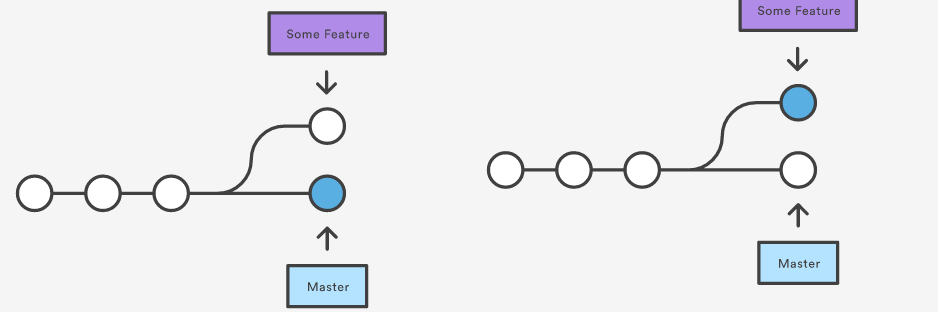
**Git checkout –b <ne-branch>**

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.

**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**

Refer *git branch* for examples.



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.

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 master branch:

**git checkout master**

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

# git merge

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/tutorials/using-branches/git-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/tutorials/using-branches/git-checkout)for selecting the current branch and **git branch -d** for deleting the obsolete target branch.

**git merge <branch>**

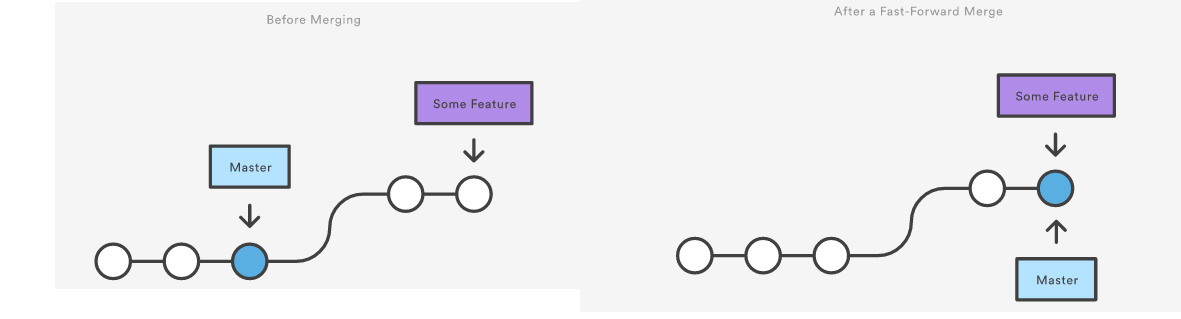
Merge the specified branch into the current branch. Git will determine the merge algorithm automatically (discussed below).

Git merge –no--ff <branch>

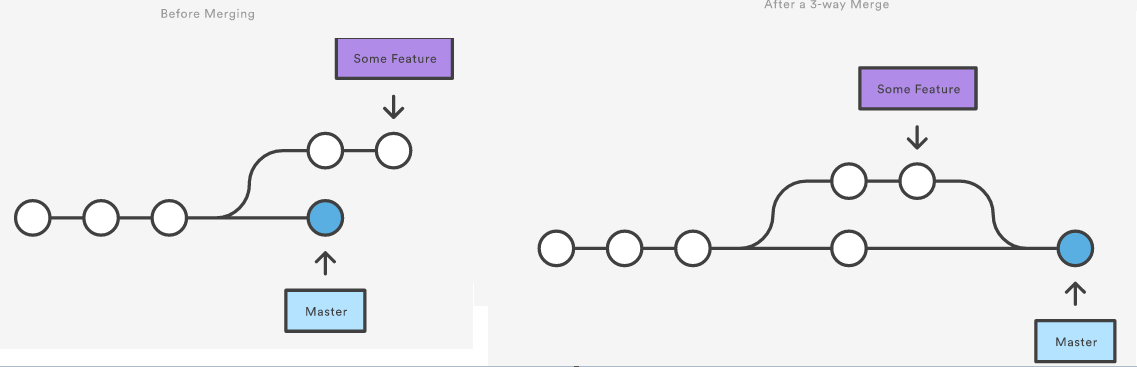
Merge the specified branch into the current branch, but always generate a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

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 into master would look something like the following:



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

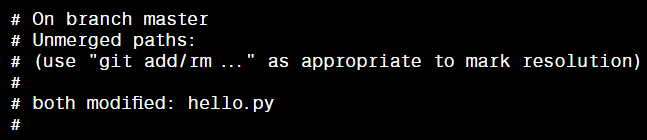


While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through [**rebasing**](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running 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/git/tutorials/inspecting-a-repository/git-status) command shows you which files need to be resolved. For example, if both branches modified the same section of hello.py, you would see something like the following:



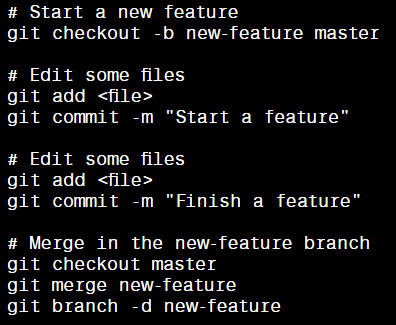
Then, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run **git add** on the conflicted file(s) to tell Git they're resolved. Then, you run a normal **git commit** to generate the merge commit. It’s the exact same process as committing an 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.

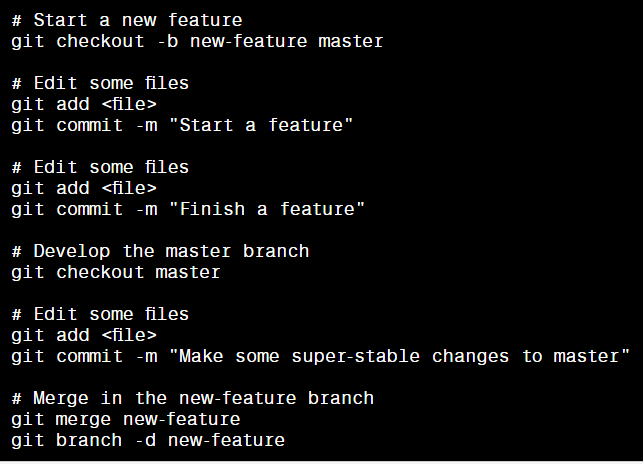


This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

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.



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.

# How To Undo Changes To The repository?

# git checkout

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.

**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.

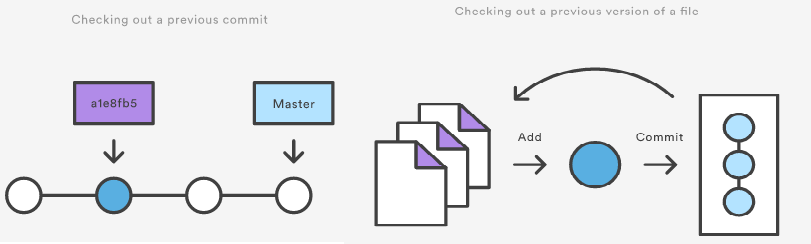
**git checkout <commit>**

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.

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 master branch (Refer [Branches Module](https://www.atlassian.com/git/tutorials/using-branches) ). 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 in the figure.

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.

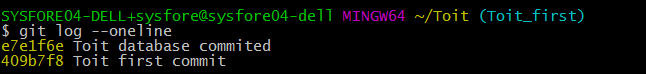


**Viewing an Old Revision**

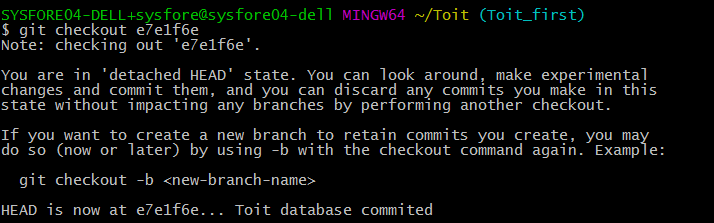
This example assumes that you’ve started developing a crazy experiment, but you’re not sure if you want to keep it or not. To help you decide, you want to take a look at the state of the project before you started your experiment. First, you’ll need to find the ID of the revision you want to see.

**git log --oneline**

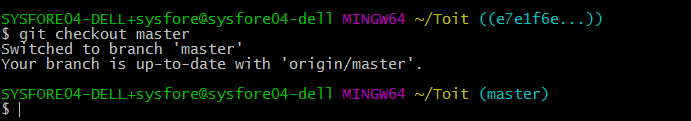
Let’s say your project history looks something like the following:



You can use git checkout to view the “Make some import changes to Toit Database commited” commit as follows:



This makes your working directory match the exact state of the **e7e1f6e** 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**we 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**

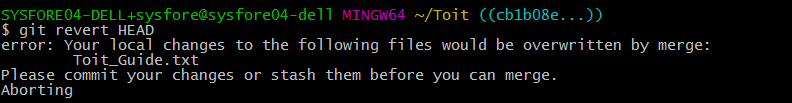
If you’re only interested in a single file, you can also use git checkout to fetch an old version of it. For example, if you only wanted to see the hello.py file from the old commit, you could use the following command:

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:

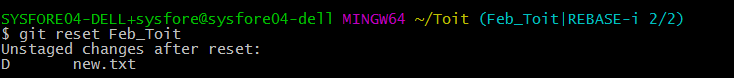
git checkout HEAD hello.py

### git revert



### git reset

git reset will get back to the status of things in the remote



### git clean

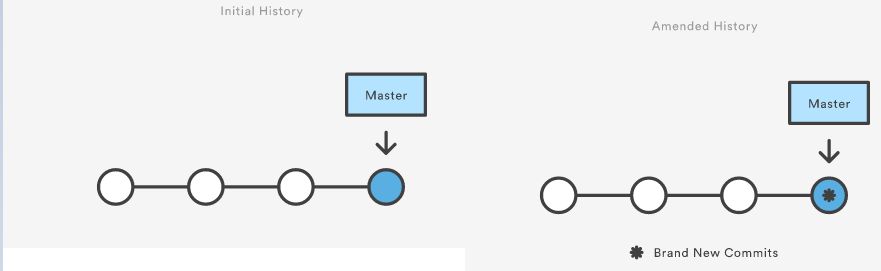
The git clean command removes untracked files from your working directory. This is really more of a convenience command, since it’s trivial to see which files are untracked with git status and remove them manually. Like an ordinary rm command, git clean is not undoable, so make sure you really want to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

# How To Rewrite History?

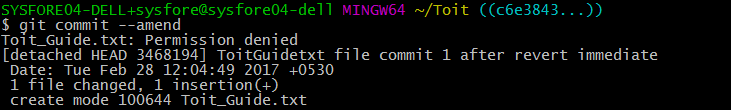
# git commit –amend

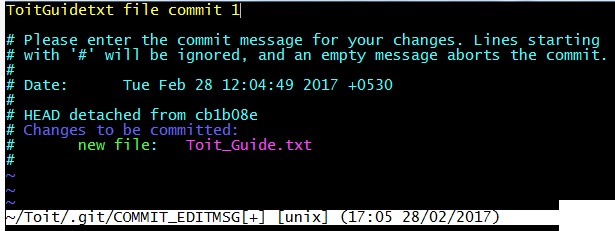
The git commit --amend command is a convenient way to fix up the most recent commit. It lets you combine staged changes with the previous commit instead of committing it as an entirely new snapshot. It can also be used to simply edit the previous commit message without changing its snapshot.



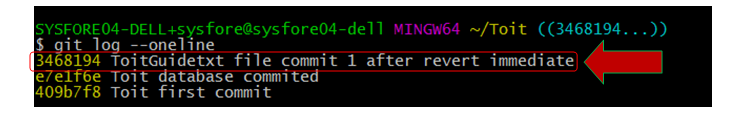
But, amending doesn’t just alter the most recent commit—it replaces it entirely. To Git, it will look like a brand new commit, which is visualized with an asterisk (\*) in the diagram above. It’s important to keep this in mind when working with public repositories.

**git commit – amend** will open the editor like below to 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.





Check with **git log - -oneline**

****

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**

From the [**git rese**t](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset) page, we 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

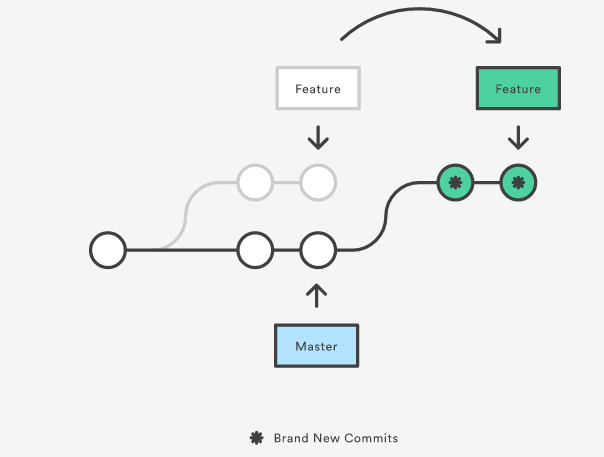
git add main.py

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 amendment 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.

# git rebase

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

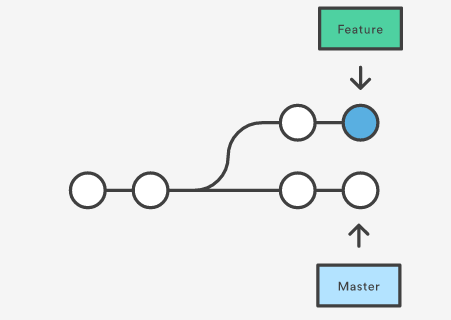


. From a content perspective, rebasing really is just moving a branch from one commit to another. But internally, Git accomplishes this by creating new commits and applying them to the specified base—it’s literally rewriting your project history. It’s very important to understand that, even though the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

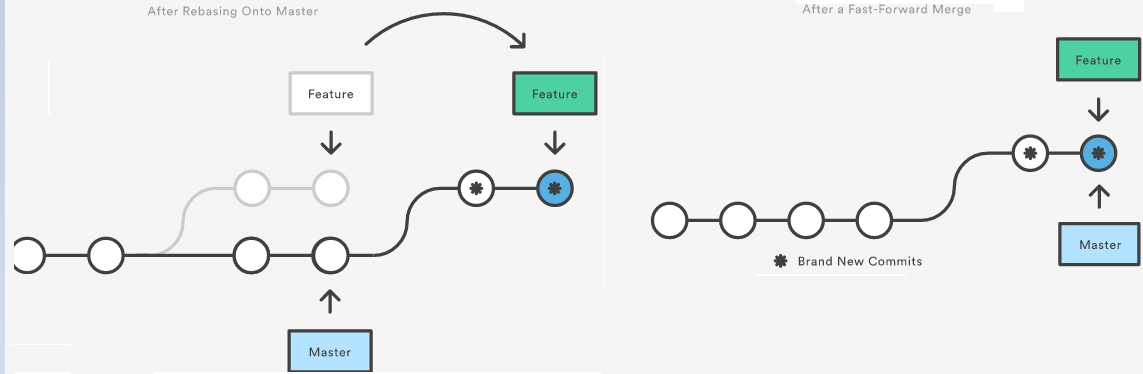
**git rebase <base>**

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).

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.



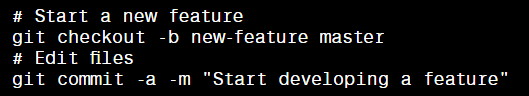
Rebasing is a common way to integrate upstream changes into your local repository. Pulling in upstream changes with **git merge** results in a superfluous merge commit every time you want to see how the project has progressed. On the other hand, rebasing is like saying, “I want to base my changes on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

**Don’t Rebase Public History**

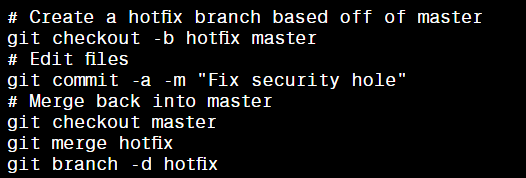
As we’ve discussed with **git commit --amend** and **git reset,** you should never rebase commits that have been pushed to a public repository. The rebase would replace the old commits with new ones, and it would look like that part of your project history abruptly vanished.

**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.



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



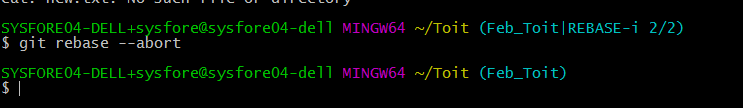
After merging the hotfix into master, we have a forked project history. Instead of a plain git merge, we’ll integrate the feature branch with a rebase to maintain a linear history:



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



git rebase --abort



### git rebase –i

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.

git rebase –i <base>

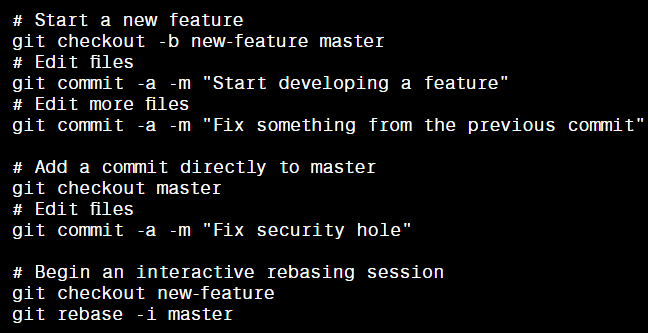
Rebase the current branch onto <base>, but use an interactive rebasing session. This opens an editor where you can enter commands (described below) for each commit to be rebased. These commands determine how individual commits will be transferred to the new base. You can also reorder the commit listing to change the order of the commits themselves.

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.

****

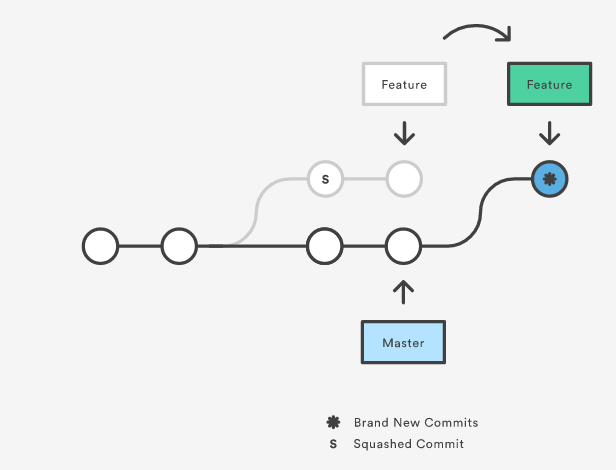
The last command will open an editor populated with the two commits from new-feature, along with some instructions:



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:



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:



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.

# git reflog

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.

**git reflog**

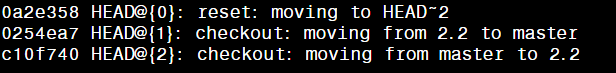
Show the reflog for the local repository.

**git reflog - -relative-date**

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

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.

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

****

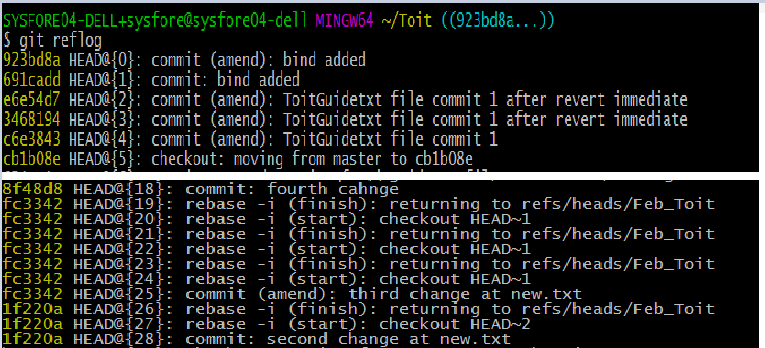
The reflog above shows a checkout from master to the 2.2 branch and back. From there, there's a hard reset to an older commit. The latest activity is represented at the top labeled **HEAD@{0}.**

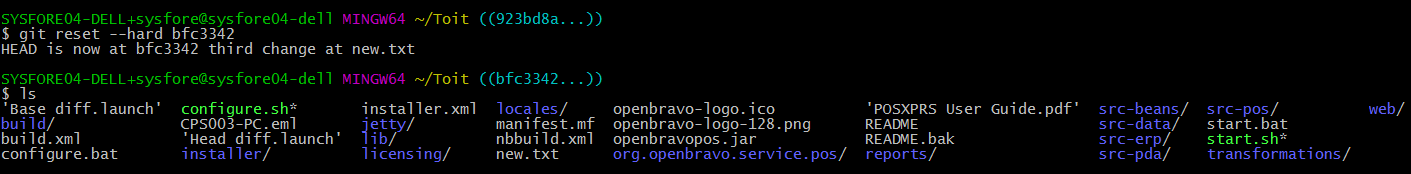
If it turns out that you accidentially moved back, the reflog will contain the commit master pointed to (0254ea7) before you accidentially dropped 2 commits.

Using [**git reset**](https://www.atlassian.com/git/tutorials/undoing-changes/git-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.

**git reset --hard bfc3342**

The below scenario shows reflog and reset to commit **bfc3342 **

****

-----------------------------------------------END-------------------------------------------------------------