The term "pull request" comes from git, where the git pull command is used to merge a different repository into your local one. So if someone else has a copy of your git repository, and makes changes to it that they would like you to incorporate, they can ask you to pull the changes from their repository; they're requesting a pull, hence the term "pull request".

Nowadays for most users this tends to be hidden behind a graphical interface of some sort, on Github or Bitbucket or via Gerrit for example; but the principle remains the same: someone copies your repository, makes changes and requests them to be merged into your own copy, presumably after you've reviewed them.

* "Push" is you forcing the changes being present in the target repository (git push).
* "Pull" is the target repository grabbing your changes to be present there (git pull from the other repo).

A "pull request" is you requesting the target repository to please grab your changes.

A "push request" would be the target repository requesting you to push your changes.

* When you type **Push** from Local. (git push) - You throw your codes to Github. in other words, GitHub is **Pulling** codes from you(Local).
* When you type **Pull** from Local. (git pull) - You are taking codes from GitHub. in other words, GitHub is **Pushing** codes to you(Local).

[**git remote**](https://www.atlassian.com/git/tutorials/syncing) [**git fetch**](https://www.atlassian.com/git/tutorials/syncing/git-fetch) [**git push**](https://www.atlassian.com/git/tutorials/syncing/git-push) [**git pull**](https://www.atlassian.com/git/tutorials/syncing/git-pull)

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The git pull command is actually a combination of two other commands, [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). In the first stage of operation git pull will execute a git fetch scoped to the local branch that HEAD is pointed at. Once the content is downloaded, git pull will enter a merge workflow. A new merge commit will be-created and HEAD updated to point at the new commit.

Git Cheat Sheet

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 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 current working directory, which contains all of the necessary Git metadata for the new repository. This metadata includes subdirectories for objects, refs, and template files. A HEAD file is also created which points to the currently checked out commit.

As others already mentioned, the .git directory is *hidden* from normal view, by virtue of being named .git. Files and directories with names starting with . are normally not displayed, to keep things uncluttered.

There's one other important point:

... I get this result Reinitialized **existing** Git repository in ...

git init Will create a .git folder (which is .) means a hidden folder.

Like 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.

git clone is primarily used to point to an existing repo and make a clone or copy of that repo at in a new directory, at another location. The original repository can be located on the local filesystem or on remote machine accessible supported protocols. 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.

The most basic use case for git config is to invoke it with a configuration name, which will display the set value at that name. Configuration names are dot delimited strings composed of a 'section' and a 'key' based on their hierarchy. For example: user.email

git config user.email

In this example, email is a child property of the user configuration block. This will return the configured email address, if any, that Git will associate with locally created commits.

git config user.name

Define author name to be used for all commits in current repo. Devs commonly use --global flag to set config options for current user.

git add <directory>

Stage all changes in for the next commit. Replace with a to change a specific file.

This means an SVN commit needs Internet access in order to fully 'save' project changes. Git commits can be captured and built up locally, then pushed to a remote server as needed using the git push -u origin master command.

 Git is a distributed application model whereas SVN is a centralized model.

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

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 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 add <file>

While they share the same name, git commit is nothing like svn commit. This shared term can be a point of confusion for Git newcomers who have a svn background, and it is important to emphasize the difference. To compare git commit vs svn commit is to compare a centralized application model (svn) vs a distributed application model (Git). In SVN, a commit pushes changes from the local SVN client, to a remote centralized shared SVN repository. In Git, repositories are distributed, Snapshots are committed to the local repository, and this requires absolutely no interaction with other Git repositories. Git commits can later be pushed to arbitrary remote repositories.

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.

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 point to merge with other users. While isolation and deferred integration are individually beneficial, it is in a team's best interest to integrate frequently and in small units.

Whereas SVN tracks differences of a file, Git’s version control model is based on snapshots. For example, a 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.

## **Common options**

git commit

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

git commit -a

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

git commit -m "commit message"

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

git commit -am "commit message"

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

git commit --amend

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

## **The staging area**

The primary function of the git add command, is to promote pending changes in the working directory, to the git 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. The staging area is considered one of the ["three trees" of Git](https://www.atlassian.com/git/tutorials/undoing-changes/git-reset), along with, the working directory, and the commit 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.

## **git status**

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.

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

## **Creating a tag**

To create a new tag execute the following command:

git tag <tagname>

Replace <tagname> with a semantic identifier to the state of the repo at the time the tag is being created. A common pattern is to use version numbers like git tag v1.4. Git supports two different types of tags, annotated and lightweight tags. The previous example created a lightweight tag. Lightweight tags and Annotated tags differ in the amount of accompanying meta data they store. A best practice is to consider Annotated tags as public, and Lightweight tags as private. Annotated tags store extra meta data such as: the tagger name, email, and date. This is important data for a public release. Lightweight tags are essentially 'bookmarks' to a commit, they are just a name and a pointer to a commit, useful for creating quick links to relevant commits.

## **Annotated Tags**

Annotated tags are stored as full objects in the Git database. To reiterate, They store extra meta data such as: the tagger name, email, and date. Similar to commits and commit messages Annotated tags have a tagging message. Additionally, for security, annotated tags can be signed and verified with GNU Privacy Guard (GPG). Suggested best practices for git tagging is to prefer annotated tags over lightweight so you can have all the associated meta-data.

git tag -a v1.4

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

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

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

## **Lightweight Tags**

git tag v1.4-lw

Executing this command creates a lightweight tag identified as v1.4-lw. Lightweight tags are created with the absence of the -a, -s, or -m options. Lightweight tags create a new tag checksum and store it in the .git/ directory of the project's repo.

git log

Display the entire commit history using the default format. For customization see additional options.

git diff

Show unstaged changes between your index and working directory

git commit --amend

Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit’s message.

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

git rebase

## **What is git rebase?**

## Rebasing is the process of moving or combining a sequence of commits to a new base commit.

The primary reason for rebasing is to maintain a linear project history. For example, consider a situation where the master branch has progressed since you started working on a feature branch. You want to get the latest updates to the master branch in your feature branch, but you want to keep your branch's history clean so it appears as if you've been working off the latest master branch. This gives the later benefit of a clean merge of your feature branch back into the master branch.

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

1. A bug is identified in the master branch. A feature that was working successfully is now broken.
2. A developer examines the history of the master branch using git log because of the "clean history" the developer is quickly able to reason about the history of the project.
3. The developer can not identify when the bug was introduced using git log so the developer executes a git bisect.
4. Because the git history is clean, git bisect has a refined set of commits to compare when looking for the regression. The developer quickly finds the commit that introduced the bug and is able to act accordingly.

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

git reflog

Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs." Many Git commands accept a parameter for specifying a reference or "ref", which is a pointer to a commit.

Reflogs track when Git refs were updated in the local repository. In addition to branch tip reflogs, a special reflog is maintained for the Git stash. Reflogs are stored in directories under the local repository's .

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

## **Common Options**

git branch

List all of the branches in your repository. This is synonymous with git branch --list.

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>.

git branch -a

List all remote branches.

## **Creating Branches**

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

Then, you create a branch using the following command:

git branch crazy-experiment

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

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

## **Creating remote branches**

So far these examples have all demonstrated local branch operations. The git branch command also works on remote branches. In order to operate on remote branches, a remote repo must first be configured and added to the local repo config.

$ git remote add new-remote-repo https://bitbucket.com/user/repo.git

# Add remote repo to local repo config

$ git push <new-remote-repo> crazy-experiment~

# pushes the crazy-experiment branch to new-remote-repo

This command will push a copy of the local branch crazy-experiment to the remote repo <remote>.

## **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 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 previous commands will delete a local copy of a branch. The branch may still exist in remote repos. To delete a remote branch execute the following.

git push origin --delete crazy-experiment

Or

git push origin :crazy-experiment

This will push a delete signal to the remote origin repository that triggers a delete of the remote crazy-experiment branch.

The git checkout command may occasionally be confused with git clone. The difference between the two commands is that clone works to fetch code from a remote repository, alternatively checkout works to switch between versions of code already on the local system.

Assuming the repo you're working in contains pre-existing branches, you can switch between these branches using git checkout. To find out what branches are available and what the current branch name is, execute git branch.

$> git branch

master

another\_branch

feature\_inprogress\_branch

$> git checkout feature\_inprogress\_branch

The above example demonstrates how to view a list of available branches by executing the git branch command, and switch to a specified branch, in this case, the feature\_inprogress\_branch.

hen you want to start a new feature, you create a new branch off master using git branch new\_branch. Once created you can then use git checkout new\_branch to switch to that branch. Additionally, The git checkout command accepts a -b argument that acts as a convenience method which will create the new branch and immediately switch to it. You can work on multiple features in a single repository by switching between them with git checkout.

git checkout -b <new-branch>

When you want to start a new feature, you create a new branch off master using git branch new\_branch. Once created you can then use git checkout new\_branch to switch to that branch. Additionally, The git checkout command accepts a -b argument that acts as a convenience method which will create the new branch and immediately switch to it. You can work on multiple features in a single repository by switching between them with git checkout.

git checkout -b <new-branch>

The above example simultaneously creates and checks 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>

By default git checkout -b will base the new-branch off the current HEAD. An optional additional branch parameter can be passed to git checkout. In the above example, <existing-branch> is passed which then bases new-branch off of existing-branch instead of the current HEAD.

## **Switching Branches**

Switching branches is a straightforward operation. Executing the following will point HEAD to the tip of <branchname>.

git checkout <branchname>

Git tracks a history of checkout operations in the reflog. You can execute git reflog to view the history.

## **Git Checkout a Remote Branch**

When collaborating with a team it is common to utilize remote repositories. These repositories may be hosted and shared or they may be another colleague's local copy. Each remote repository will contain its own set of branches. In order to checkout a remote branch you have to first fetch the contents of the branch.

git fetch --all

In modern versions of Git, you can then checkout the remote branch like a local branch.

git checkout <remotebranch>

Older versions of Git require the creation of a new branch based on the remote.

git checkout <remotebranch> origin/<remotebranch>

Additionally you can checkout a new local branch and reset it to the remote branches last commit.

git checkout -b <branchname>

git reset --hard origin/<branchname>

In the most frequent use cases, git merge is used to combine two branches.

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches.

## **Fast Forward Merge**

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

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.

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 <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

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.

In the event that you require a merge commit during a fast forward merge for record keeping purposes you can execute git merge with the --no-ffoption.

git merge --no-ff <branch>

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

|  |
| --- |
| // add remote repository to local git |
|  | git remote add origin {remote repository url} |
|  |  |
|  | // pushes up the all repository |
|  | git push -u origin --all |
|  |  |
|  | // pushes up any tags |
|  | git push -u origin --tags |
|  |  |
|  | // cancel one before commit |
|  | git reset --soft HEAD~ |
|  |  |
|  | // delete remote branch(foo) |
|  | git push origin :foo |
|  |  |
|  | // create zip file from git repository |
|  | git archive [HEAD|tag|branch|commit] --output=hoge.zip |
|  |  |
|  | // cancel merge before fixing any conflict |
|  | git merge --abort |
|  |  |
|  | // cancel merge while fixing any conflict |
|  | git reset --hard HEAD |
|  |  |
|  | // cancel merge after commit |
|  | git reset --hard ORIG\_HEAD |
|  |  |
|  | // show diff on HEAD |
|  | git show |
|  |  |
|  | // show diff on commitA |
|  | git show commitA |
|  |  |
|  | // show log between tag v10 and tag v20 |
|  | git log v10..v20 |
|  |  |
|  | // remove branch which was removed from remote |
|  | git fetch origin --prune |
|  |  |
|  | // removev tag which was removed from remote(1), depends on git version |
|  | git fetch origin --prune --tags |
|  |  |
|  | // removev tag which was removed from remote(2), depends on git version |
|  | git fetch origin --prune 'refs/tags/\*:refs/tags/\*' |
|  |  |
|  | // removev tag which was removed from remote(3), on all git version |
|  | git tag -l | xargs git tag -d |
|  | git fetch origin --tags |

### Git Cheat Sheet.

Despite the GUI interfaces provided for working with Git by VS Code and IntelliJ, I tend to stick to the command line when working with Git. It is likely due to the fact that, while learning the Git basics, I followed the online Git tutorial and that stuck, regardless, these are some Git commands I use often:

git branch

git status

git fetch

git pull

git add .

git commit -m "Type message here"

git log -3

git remote -v

git remote add origin master

git remote remove remoteName

git set branch --set-upstream branchName remoteName/branchName

git checkout -b branchName

git checkout branchName

git clone repoUrlGoesHere

git branch -m newBranchName

git branch -m oldBranchName newBranchName

git branch -d branchName

git push remoteName branchName

git config --global --get http.proxy

git config --global --get https.proxy

git config --global --unset http.proxy

git config --global --unset https.proxy

git config --global http.proxy http://proxyUser:proxyPassword@proxyLocation.proxyTld:proxyPort

git config --global https.proxy http://proxyUser:proxyPassword@proxyLocation.proxyTld:proxyPort

Here are the Git commands which are being covered:

* **git config**
* **git init**
* **git clone**
* **git add**
* **git commit**
* **git diff**
* **git reset**
* **git status**
* **git rm**
* **git log**
* **git show**
* **git tag**
* **git branch**
* **git checkout**
* **git merge**
* **git remote**
* **git push**
* **git pull**
* **git stash**

So, let's get started!

## **Git Commands**

### git config

Usage: git config –global user.name “[name]”

Usage: git config –global user.email “[email address]”

This command sets the author name and email address respectively to be used with your commits.

Git Config Command - Git Commands - Edureka

### git init

Usage: git init [repository name]

This command is used to start a new repository.

GitInit Command - Git Commands - Edureka

### git clone

Usage: git clone [url]

This command is used to obtain a repository from an existing URL.



### git add

Usage: git add [file]

This command adds a file to the staging area.

Git Add Command - Git Commands - Edureka

Usage: git add \*

This command adds one or more to the staging area.

Git Add Command - Git Commands - Edureka

### git commit

Usage: git commit -m “[ Type in the commit message]”

This command records or snapshots the file permanently in the version history.



Usage: git commit -a

This command commits any files you’ve added with the git add command and also commits any files you’ve changed since then.

Git Commit Command - Git Commands - Edureka

### git diff

Usage: git diff

This command shows the file differences which are not yet staged.



 Usage: git diff –staged

This command shows the differences between the files in the staging area and the latest version present.



Usage: git diff [first branch] [second branch]

This command shows the differences between the two branches mentioned.



### git reset

Usage: git reset [file]

This command unstages the file, but it preserves the file contents.



Usage: git reset [commit]

This command undoes all the commits after the specified commit and preserves the changes locally.

Git Reset Command - Git Commands - Edureka

Usage: git reset –hard [commit]  This command discards all history and goes back to the specified commit.

Git Reset Command - Git Commands - Edureka

### git status

Usage: git status

This command lists all the files that have to be committed.



### git rm

Usage: git rm [file]

This command deletes the file from your working directory and stages the deletion.

Git Rm Command - Git Commands - Edureka

### git log

Usage: git log

This command is used to list the version history for the current branch.



Usage: git log –follow[file]

This command lists version history for a file, including the renaming of files also.



### git show

Usage: git show [commit]

This command shows the metadata and content changes of the specified commit.



### git tag

Usage: git tag [commitID]

This command is used to give tags to the specified commit.



### git branch

Usage: git branch

This command lists all the local branches in the current repository.

Git Branch Command - Git Commands - Edureka

Usage: git branch [branch name]

This command creates a new branch.

Git Branch Command - Git Commands - Edureka

Usage: git branch -d [branch name]

This command deletes the feature branch.

Git Branch Command - Git Commands - Edureka

### git checkout

Usage: git checkout [branch name]

This command is used to switch from one branch to another.

Git Checkout Command - Git Commands - Edureka

Usage: git checkout -b [branch name]

This command creates a new branch and also switches to it.

Git Checkout Command - Git Commands - Edureka

### git merge

Usage: git merge [branch name]

This command merges the specified branch’s history into the current branch.

Git Merge Command - Git Commands - Edureka

### git remote

Usage: git remote add [variable name] [Remote Server Link]

This command is used to connect your local repository to the remote server.

Git Remote Command - Git Commands - Edureka

### git push

Usage: git push [variable name] master

This command sends the committed changes of master branch to your remote repository.



Usage: git push [variable name] [branch]

This command sends the branch commits to your remote repository.



Usage: git push –all [variable name]

This command pushes all branches to your remote repository.



Usage: git push [variable name] :[branch name]

This command deletes a branch on your remote repository.



### git pull

Usage: git pull [Repository Link]

This command fetches and merges changes on the remote server to your working directory.



### git stash

Usage: git stash save

This command temporarily stores all the modified tracked files.

Git Stash Command - Git Commands - Edureka

Usage: git stash pop

This command restores the most recently stashed files.



Usage: git stash list

This command lists all stashed changesets.

Git Stash Command - Git Commands - Edureka

Usage: git stash drop

This command discards the most recently stashed changeset.

Git Stash Command - Git Commands - Edureka

Want to learn more about git commands? Here is a [Git Tutorial](https://www.edureka.co/blog/git-tutorial/) to get you started. Alternatively, you can take a top-down approach and start with this [DevOps Tutorial.](https://www.edureka.co/blog/devops-tutorial)

|  |  |  |
| --- | --- | --- |
| Git task | Notes | Git commands |
| [**Tell Git who you are**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) | Configure the author name and email address to be used with your commits.  Note that Git [strips some characters](http://stackoverflow.com/questions/26159274/is-it-possible-to-have-a-trailing-period-in-user-name-in-git/26219423#26219423) (for example trailing periods) from user.name. | git config --global user.name "Sam Smith"  git config --global user.email sam@example.com |
| [**Create a new local repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) |  | git init |
| [**Check out a repository**](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone) | Create a working copy of a local repository: | git clone /path/to/repository |
| For a remote server, use: | git clone username@host:/path/to/repository |
| [**Add files**](https://www.atlassian.com/git/tutorials/saving-changes#git-add) | Add one or more files to staging (index): | git add <filename>  git add \* |
| [**Commit**](https://www.atlassian.com/git/tutorials/saving-changes#git-commit) | Commit changes to head (but not yet to the remote repository): | git commit -m "Commit message" |
| Commit any files you've added with git add, and also commit any files you've changed since then: | git commit -a |
| [**Push**](https://www.atlassian.com/git/tutorials/syncing#git-push) | Send changes to the master branch of your remote repository: | git push origin master |
| [**Status**](https://www.atlassian.com/git/tutorials/inspecting-a-repository#git-status) | List the files you've changed and those you still need to add or commit: | git status |
| [**Connect to a remote repository**](https://www.atlassian.com/git/tutorials/syncing#git-remote) | If you haven't connected your local repository to a remote server, add the server to be able to push to it: | git remote add origin <server> |
| List all currently configured remote repositories: | git remote -v |
| [**Branches**](https://www.atlassian.com/git/tutorials/using-branches) | Create a new branch and switch to it: | git checkout -b <branchname> |
| Switch from one branch to another: | git checkout <branchname> |
| List all the branches in your repo, and also tell you what branch you're currently in: | git branch |
| Delete the feature branch: | git branch -d <branchname> |
| Push the branch to your remote repository, so others can use it: | git push origin <branchname> |
| Push all branches to your remote repository: | git push --all origin |
| Delete a branch on your remote repository: | git push origin :<branchname> |
| [**Update from the remote repository**](https://www.atlassian.com/git/tutorials/syncing) | Fetch and merge changes on the remote server to your working directory: | git pull |
| To merge a different branch into your active branch: | git merge <branchname> |
| View all the merge conflicts:  View the conflicts against the base file:  Preview changes, before merging: | git diff  git diff --base <filename>  git diff <sourcebranch> <targetbranch> |
| After you have manually resolved any conflicts, you mark the changed file: | git add <filename> |
| **Tags** | You can use tagging to mark a significant changeset, such as a release: | git tag 1.0.0 <commitID> |
| CommitId is the leading characters of the changeset ID, up to 10, but must be unique. Get the ID using: | git log |
| Push all tags to remote repository: | git push --tags origin |
| [**Undo local changes**](https://www.atlassian.com/git/tutorials/undoing-changes) | If you mess up, you can replace the changes in your working tree with the last content in head:  Changes already added to the index, as well as new files, will be kept. | git checkout -- <filename> |
| Instead, to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it, do this: | git fetch origin  git reset --hard origin/master |
| **Search** | Search the working directory for foo(): | git grep "foo()" |

## **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 <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Develop the master branch

git checkout master

# Edit some files

git add <file>

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.

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

Getting & Creating Projects

Command Description

git init Initialize a local Git repository

git clone ssh://git@github.com/[username]/[repository-name].git Create a local copy of a remote repository

Basic Snapshotting

Command Description

git status Check status

git add [file-name.txt] Add a file to the staging area

git add -A Add all new and changed files to the staging area

git commit -m "[commit message]" Commit changes

git rm -r [file-name.txt] Remove a file (or folder)

Branching & Merging

Command Description

git branch List branches (the asterisk denotes the current branch)

git branch -a List all branches (local and remote)

git branch [branch name] Create a new branch

git branch -d [branch name] Delete a branch

git push origin --delete [branch name] Delete a remote branch

git checkout -b [branch name] Create a new branch and switch to it

git checkout -b [branch name] origin/[branch name] Clone a remote branch and switch to it

git branch -m [old branch name] [new branch name] Rename a local branch

git checkout [branch name] Switch to a branch

git checkout - Switch to the branch last checked out

git checkout -- [file-name.txt] Discard changes to a file

git merge [branch name] Merge a branch into the active branch

git merge [source branch] [target branch] Merge a branch into a target branch

git stash Stash changes in a dirty working directory

git stash clear Remove all stashed entries

Sharing & Updating Projects

Command Description

git push origin [branch name] Push a branch to your remote repository

git push -u origin [branch name] Push changes to remote repository (and remember the branch)

git push Push changes to remote repository (remembered branch)

git push origin --delete [branch name] Delete a remote branch

git pull Update local repository to the newest commit

git pull origin [branch name] Pull changes from remote repository

git remote add origin ssh://git@github.com/[username]/[repository-name].git Add a remote repository

git remote set-url origin ssh://git@github.com/[username]/[repository-name].git Set a repository's origin branch to SSH

Inspection & Comparison

Command Description

git log View changes

git log --summary View changes (detailed)

git log --oneline View changes (briefly)

git diff [source branch] [target branch] Preview changes before merging

Version control can be roughly divided into two categories: distributed version control systems (DVCS) and centralized version control systems (CVCS). SVN is centralized.

Centralized version control means that the version history is stored in a central server. When a developer wants to make changes to certain files, they pull files from that central server to their own computer. After the developer has made changes, they send the changed files back to the central server.

common complaint about SVN is its tedious branching model. Branches allow you to work on multiple versions of your code simultaneously. In SVN, branches are created as directories inside the server. Many developers dislike this directory structure. But the challenges don’t stop there

### SVN Requires You to Resolve Conflicts Manually

Merging is the other big problem that developers often complain about with SVN. If you’re working with a history where a set of changes are made and committed, then another change is made (i.e., linear) and committed, the merge will be easy.

Things get complicated when you have two or more developers working on the same code base and you need to merge. In this case, SVN fails and the developers need to resolve the conflicts manually, which wastes hours of developer time.

* **Repository:** A repository is the heart of any version control system. It is the central place where developers store all their work. Repository not only stores files but also the history. Repository is accessed over a network, acting as a server and version control tool acting as a client. Clients can connect to the repository, and then they can store/retrieve their changes to/from repository. By storing changes, a client makes these changes available to other people and by retrieving changes, a client takes other people's changes as a working copy.
* **Trunk:** The trunk is a directory where all the main development happens and is usually checked out by developers to work on the project.
* **Tags** : The tags directory is used to store named snapshots of the project. Tag operation allows to give descriptive and memorable names to specific version in the repository.

For example, LAST\_STABLE\_CODE\_BEFORE\_EMAIL\_SUPPORT is more memorable than

Repository UUID: 7ceef8cb-3799-40dd-a067-c216ec2e5247 and

Revision: 13

* **Branches:** Branch operation is used to create another line of development. It is useful when you want your development process to fork off into two different directions. For example, when you release version 5.0, you might want to create a branch so that development of 6.0 features can be kept separate from 5.0 bug-fixes.
* **Working copy:** Working copy is a snapshot of the repository. The repository is shared by all the teams, but people do not modify it directly. Instead each developer checks out the working copy. The working copy is a private workplace where developers can do their work remaining isolated from the rest of the team.
* **Commit changes:** Commit is a process of storing changes from private workplace to central server. After commit, changes are made available to all the team. Other developers can retrieve these changes by updating their working copy. Commit is an atomic operation. Either the whole commit succeeds or is rolled back. Users never see half finished commit.

VisualSVN Server allows you to easily install and manage a fully-functional Subversion server on the Windows platform. Thanks to its robustness, unbeatable usability and unique enterprise-grade features, VisualSVN Server is useful both for small business and corporate users.

## Free for commercial use!

VisualSVN Server is freely available for commercial use under the Community license. The free Community license does not require any registration, allows an unlimited number of repositories and up to 15 users.

TortoiseSVN is a really easy to use Revision control / version control / source control software for Windows.  
Since its not an integration for a specific IDE you can use it with whatever development tools you like.  
TortoiseSVN is free to use. You dont need to get a loan or pay a full years salary to use it.

It also offers the "Project Monitor" for monitoring repositories for updates (similar to Commit Monitor or SVN Notifier)

ZenTao is an open source project management tool which focuses on software development projects and supports Scrum. Combining product management, project management, QA management, document management, bug management and todo management. It is a professional project management software, covering the core process of software development projects.

Product management: including products, stories, plans, releases, and roadmaps;  
Project management: including projects, tasks, teams, builds and burndown charts;  
Quality management:including bugs, test cases, test tasks and test results;  
Document management: including product document library, project document library, and customized document library;  
Work management: including todo management and personal work management like my task, my bug, my story and my project;  
Company management: including departments, users, groups, and privileges;  
Report: various statistical reports;  
Search feature: powerful search function helps you find the information you need.  
Extension mechanism: extensible in almost any parts of ZenTao;  
API mechanism: convenient for integration with other systems.

Team Foundation Server (commonly abbreviated TFS) is a Microsoft offering for source control, data collection, reporting, and project tracking, and is intended for collaborative software development projects. It is available either as stand-alone software, or as the server side back end platform for Visual Studio Team System. (VSTS)

AnkhSVN is a Subversion SourceControl Provider for Visual Studio. The software allows you to perform the most common version control operations directly from inside the Microsoft Visual Studio IDE. With SourceAnywhere is a source control solution designed as Visual SourceSafe (VSS) replacement. It is VSS-style source control solution with good support for Visual Studio, Eclipse, Dreamweaver and other IDEs. Using Microsoft SQL Server as the backend, it is a simple yet robust version control tool that satisfies the needs of both local and remote development team.

AnkhSVN you no longer need to leave your IDE to perform tasks like viewing the status of your source code, updating your Subversion working copy and committing changes. You can even browse your repository and you can plug-in your favorite diff tool.

Microsoft Visual SourceSafe (VSS) is a source control software package oriented towards small software development projects.

SourceAnywhere is a source control solution designed as Visual SourceSafe (VSS) replacement. It is VSS-style source control solution with good support for Visual Studio, Eclipse, Dreamweaver and other IDEs. Using Microsoft SQL Server as the backend, it is a simple yet robust version control tool that satisfies the needs of both local and remote development team.

VisualSVN Server is a package that contains everything you need to install, configure and manage Subversion server for your team on Windows platform. It includes Subversion, Apache and a management console.

You can use any Subversion client to connect to VisualSVN Server or a web browser to quickly browse though repositories.

**Apache Subversion** (often abbreviated **SVN**, after its command name *svn*) is a [software versioning](https://en.wikipedia.org/wiki/Software_versioning) and [revision control](https://en.wikipedia.org/wiki/Revision_control) system distributed as [open source](https://en.wikipedia.org/wiki/Open-source_software) under the [Apache License](https://en.wikipedia.org/wiki/Apache_License).[[2]](https://en.wikipedia.org/wiki/Apache_Subversion#cite_note-2) Software developers use Subversion to maintain current and historical versions of files such as [source code](https://en.wikipedia.org/wiki/Source_code), web pages, and documentation. Its goal is to be a mostly compatible successor to the widely used [Concurrent Versions System](https://en.wikipedia.org/wiki/Concurrent_Versions_System) (CVS).

The [open source](https://en.wikipedia.org/wiki/Open-source_software) community has used Subversion widely: for example in projects such as [Apache Software Foundation](https://en.wikipedia.org/wiki/Apache_Software_Foundation), [Free Pascal](https://en.wikipedia.org/wiki/Free_Pascal), [FreeBSD](https://en.wikipedia.org/wiki/FreeBSD), [GCC](https://en.wikipedia.org/wiki/GNU_Compiler_Collection) and [SourceForge](https://en.wikipedia.org/wiki/SourceForge" \o "SourceForge). [CodePlex](https://en.wikipedia.org/wiki/CodePlex" \o "CodePlex) was previously a common host for Subversion repositories.

A component of [software configuration management](https://en.wikipedia.org/wiki/Software_configuration_management), **version control**, also known as **revision control** or **source control**,[[1]](https://en.wikipedia.org/wiki/Version_control#cite_note-Mercurial-1) is the management of changes to documents, [computer programs](https://en.wikipedia.org/wiki/Computer_program), large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision".

VisualSVN Repository Configurator is a standalone application which allows non-administrative users to manage VisualSVN Server repositories remotely.

With SmartSVN Professional you can work with tags and branches as conveniently as if they were native Subversion features. Once you have defined the project repository locations for the trunk, the tags and branches, you won't have to deal with hard-to-remember URLs anymore: Just use tags and branches the way you're used to from other version control systems.

For example, when you need to switch to a different branch or tag, you won't have to type the branch or tag name. but Instead, you can simply select it from a Tag Browser that displays branches and tags in their hierarchical order.

# Subversion Tutorial: 10 Most Used SVN Commands with Examples

*by* SASIKALA *on* APRIL 25, 2011

SVN stands for Subversion.

Subversion is a free/open-source version control system. Subversion manages files and directories over time. A tree of files is placed into a central repository. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your code, or examine the history of how your code was changed.

This article explains some basic SVN commands with examples.

### **SVN Working Copy**

SVN is a repository that holds all our versioned data, which is also called as SVN server. SVN client program which manages local reflections of portions of that versioned data which is called as working copy. SVN client can access its repository across networks. Multiple users can access the repository at the same time.

### **1. SVN Checkout – Create working copy**

Checkout command is used to download sources from SVN repository to working copy. If you want to access files from the SVN server, checkout is the first operation you should perform.

SVN checkout creates the working copy, from where you can do edit, delete, or add contents. You can checkout a file, directory, trunk or whole project. To checkout you should know URL of the components you want to checkout.

Syntax:

$ svn checkout/co URL PATH

* URL is the URL of the components to checkout
* If PATH is omitted, the basename of the URL will be used as the destination. If multiple URLs are given each will be checked out into a subdirectory of PATH, with the name of the subdirectory being the basename of the URL.

The following example checks out the directory to the given target directory.

$ svn co https://www.thegeekstuff.com/project/branches/release/migration/data/cfg /home/sasikala/cfg/

A /home/sasikala/cfg/ftp\_user.cfg

A /home/sasikala/cfg/inventory.cfg

A /home/sasikala/cfg/email\_user.cfg

A /home/sasikala/cfg/svn-commands

Checked out revision 811.

$ ls /home/sasikala/cfg

. .. .svn email\_user.cfg ftp\_user.cfg inventory.cfg svn-commands

When you do a checkout, it creates hidden directory named .svn, which will have the repository details.

### **2. SVN Commit – Save changes to the repository**

Whenever you do changes to the working copy, it will not reflect in SVN server. To make the changes permanent, you need to do SVN commit.

Syntax:

$ svn commit -m "log messages"

Explain why you are changing the file in the -m option.

For example, in my working copy, the file named “svn-commands” has the following content.

$ cat /home/sasikala/cfg/svn-commands

checkout

commit

add

delete

update

status

$ ls -l /home/sasikala/cfg/svn-commands

-rw-r--r-- 1 root root 41 Apr 16 11:15 svn-commands

I made a change in this file (for example, making this file empty).

$ ls -l svn-commands

-rw-r--r-- 1 root root 0 Apr 16 11:20 svn-commands

Now commit the file to make the changes permanent in the server.

$ svn commit -m "Making the file empty" svn-commands

Sending svn-commands

Transmitting file data .

Committed revision 813.

After this whenever you update your working copy or checkout, the changes will appear in the server.

### **3. SVN List – Lists directory entries**

svn list is useful when you want to view the content of the SVN repository, without downloading a working copy.

Syntax:

$ svn list

The following example lists all the files available in the given URL in the repository without downloading a working copy. When you execute svn list command with –verbose option it displays the following information.

* Revision number of the last commit
* Author of the last commit
* Size (in bytes)
* Date and time of the last commit

$ svn list --verbose https://www.thegeekstuff.com/project/branches/release/migration/data/bin

16 sasikala 28361 Apr 16 21:11 README.txt

21 sasikala 0 Apr 18 12:22 INSTALL

22 sasikala Apr 18 10:17 src/

### **4. SVN Add – Add a new file to SVN repository**

When you want to add a new file (or directory) to the repository you need to use SVN add command. The repository will have newly added file, only when you do SVN commit. Now let us add a new file called “thegeekstuff” to our repository.

* Create a file in local working copy

$ ls -l /home/sasikala/cfg/thegeekstuff

-rw-r--r-- 1 sasikala root 8 Apr 16 11:33 thegeekstuff

* Add the file into SVN repository

svn add filename will add the files into SVN repository.

$ svn add thegeekstuff

A thegeekstuff

* Commit the added the file

Until you commit, the added file will not be available in the repository.

$ svn commit -m "Adding a file thegeekstuff" thegeekstuff

Adding thegeekstuff

Transmitting file data .

Committed revision 814.

### **5. SVN Delete – Removing a file from repository**

SVN delete command deletes an item from the working copy (or repository). File will be deleted from the repository when you do a SVN commit.

Syntax:

$ svn delete URL

Now let us remove the recently created file called “thegeekstuff”.

$ svn delete thegeekstuff

D thegeekstuff

$ svn commit -m "Removing thegeekstuff file" thegeekstuff

Deleting thegeekstuff

Committed revision 814.

Now you can do svn list and check whether the file was deleted from the repository.

### **6. SVN Diff – Display the difference**

SVN diff displays the differences between your working copy and the copy in the SVN repository. You can find the difference between two revisions and two paths etc.,

Syntax:

$ svn diff filename

$ svn -r R1:R2 diff filename

The above example compares the filename@R1 and filename@R2.

Now the content of the file thegeekstuff looks like this,

$ cat /home/sasikala/cfg/thegeekstuff

testing

I edited the content of thegeekstuff file from testing to tester, which is shown below using the svn diff command.

$ svn diff thegeekstuff

Index: thegeekstuff

===================================================================

--- thegeekstuff (revision 815)

+++ thegeekstuff (working copy)

@@ -1 +1 @@

-testing

+tester

### **7. SVN Status – Status of the working copy**

Use svn status command to get the status of the file in the working copy. It displays whether the working copy is modified, or its been added/deleted, or file is not under revision control, etc.

Syntax:

$ svn status PATH

The following example shows the status of my local working copy,

$ svn status /home/sasikala/cfg

M /home/sasikala/cfg/ftp\_user.cfg

M /home/sasikala/cfg/geekstuff

‘M’ represents that the item has been modified. “svn help status” command will explain various specifiers showed in SVN status command.

### **8. SVN Log – Display log message**

As we discussed in the beginning of this article, SVN remembers every change made to your files and directories. To know all the commits made in a file or directory, use SVN log command.

Syntax:

$ svn log PATH

The following displays all the commits made on thegeekstuff file

$ svn log thegeekstuff

------------------------------------------------------------------------

r815 | sasikala | 2011-04-16 05:14:18 -0700 (Sat, 16 Apr 2011) | 1 line

Adding a file thegeekstuff

------------------------------------------------------------------------

Since we made only one commit in the file thegeekstuff, it shows only one log message with the details.

### **9. SVN Move – Rename file or directory**

This command moves a file from one directory to another or renames a file. The file will be moved on your local sandbox immediately (as well as on the repository after committing).

Syntax:

$ svn move src dest

The following command renames the file “thegeekstuff” to “tgs” in a single stroke.

$ svn move thegeekstuff tgs

A tgs

D thegeekstuff

$ ls

.# .. .svn email\_user.cfg ftp\_user.cfg inventory.cfg tgs

Now the file is renamed only in the working copy, not in the repository. To make the changes permanent, you need to commit the changes.

$ svn commit -m "Renaming thegeekstuff to tgs" tgs

Adding tgs

Transmitting file data .

Committed revision 816.

### **10. SVN Update – Update the working copy.**

svn update command brings changes from the repository into your working copy. If no revision is specified, it brings your working copy up-to-date with the HEAD revision. Otherwise, it synchronizes the working copy to the revision given in the argument.

Always before you start working in your working copy, update your working copy. So that all the changes available in repository will be available in your working copy. i.e latest changes.

Syntax:

$ svn update PATH

In case some other user added/deleted file in URL, https://www.thegeekstuff.com/project/branches/release/migration/data/cfg, your working copy will not have those files by default, until you update your working copy.

$ svn update

A new/usercfg

A new/webcfg

Updated to revision 819.

In the above svn update command output, A represents that this file is “Added” to the working copy.

[svn—Subversion Command-Line Client](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.html)

[svn Options](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.html#svn.ref.svn.sw)

[svn Subcommands](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.html#svn.ref.svn.c)

[svn add](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.add.html)

[svn blame (praise, annotate, ann)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.blame.html)

[svn cat](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.cat.html)

[svn changelist (cl)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.changelist.html)

[svn checkout (co)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.checkout.html)

[svn cleanup](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.cleanup.html)

[svn commit (ci)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.commit.html)

[svn copy (cp)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.copy.html)

[svn delete (del, remove, rm)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.delete.html)

[svn diff (di)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.diff.html)

[svn export](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.export.html)

[svn help (h, ?)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.help.html)

[svn import](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.import.html)

[svn info](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.info.html)

[svn list (ls)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.list.html)

[svn lock](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.lock.html)

[svn log](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.log.html)

[svn merge](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.merge.html)

[svn mergeinfo](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.mergeinfo.html)

[svn mkdir](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.mkdir.html)

[svn move (mv)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.move.html)

[svn patch](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.patch.html)

[svn propdel (pdel, pd)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.propdel.html)

[svn propedit (pedit, pe)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.propedit.html)

[svn propget (pget, pg)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.propget.html)

[svn proplist (plist, pl)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.proplist.html)

[svn propset (pset, ps)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.propset.html)

[svn relocate](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.relocate.html)

[svn resolve](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.resolve.html)

[svn resolved](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.resolved.html)

[svn revert](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.revert.html)

[svn status (stat, st)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.status.html)

[svn switch (sw)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.switch.html)

[svn unlock](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.unlock.html)

[svn update (up)](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.update.html)

[svn upgrade](http://svnbook.red-bean.com/en/1.7/svn.ref.svn.c.upgrade.html)

[svnadmin—Subversion Repository Administration](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.html)

[svnadmin Options](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.html#svn.ref.svnadmin.sw)

[svnadmin Subcommands](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.html#svn.ref.svnadmin.c)

[svnadmin crashtest](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.crashtest.html)

[svnadmin create](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.create.html)

[svnadmin deltify](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.deltify.html)

[svnadmin dump](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.dump.html)

[svnadmin help (h, ?)](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.help.html)

[svnadmin hotcopy](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.hotcopy.html)

[svnadmin list-dblogs](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.list-dblogs.html)

[svnadmin list-unused-dblogs](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.list-unused-dblogs.html)

[svnadmin load](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.load.html)

[svnadmin lslocks](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.lslocks.html)

[svnadmin lstxns](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.lstxns.html)

[svnadmin pack](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.pack.html)

[svnadmin recover](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.recover.html)

[svnadmin rmlocks](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.rmlocks.html)

[svnadmin rmtxns](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.rmtxns.html)

[svnadmin setlog](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.setlog.html)

[svnadmin setrevprop](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.setrevprop.html)

[svnadmin setuuid](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.setuuid.html)

[svnadmin upgrade](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.upgrade.html)

[svnadmin verify](http://svnbook.red-bean.com/en/1.7/svn.ref.svnadmin.c.verify.html)

[svnlook—Subversion Repository Examination](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.html)

[svnlook Options](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.html#svn.ref.svnlook.sw)

[svnlook Subcommands](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.html#svn.ref.svnlook.c)

[svnlook author](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.author.html)

[svnlook cat](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.cat.html)

[svnlook changed](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.changed.html)

[svnlook date](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.date.html)

[svnlook diff](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.diff.html)

[svnlook dirs-changed](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.dirs-changed.html)

[svnlook filesize](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.filesize.html)

[svnlook help (h, ?)](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.help.html)

[svnlook history](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.history.html)

[svnlook info](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.info.html)

[svnlook lock](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.lock.html)

[svnlook log](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.log.html)

[svnlook propget (pget, pg)](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.propget.html)

[svnlook proplist (plist, pl)](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.proplist.html)

[svnlook tree](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.tree.html)

[svnlook uuid](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.uuid.html)

[svnlook youngest](http://svnbook.red-bean.com/en/1.7/svn.ref.svnlook.c.youngest.html)

[svnsync—Subversion Repository Mirroring](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.html)

[svnsync Options](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.html#svn.ref.svnsync.sw)

[svnsync Subcommands](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.html#svn.ref.svnsync.c)

[svnsync copy-revprops](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.c.copy-revprops.html)

[svnsync help](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.c.help.html)

[svnsync info](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.c.info.html)

[svnsync initialize (init)](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.c.init.html)

[svnsync synchronize (sync)](http://svnbook.red-bean.com/en/1.7/svn.ref.svnsync.c.sync.html)

[svnrdump—Remote Subversion Repository Data Migration](http://svnbook.red-bean.com/en/1.7/svn.ref.svnrdump.html)

[svnrdump Options](http://svnbook.red-bean.com/en/1.7/svn.ref.svnrdump.html#svn.ref.svnrdump.sw)

[svnrdump Subcommands](http://svnbook.red-bean.com/en/1.7/svn.ref.svnrdump.html#svn.ref.svnrdump.c)

[svnrdump dump](http://svnbook.red-bean.com/en/1.7/svn.ref.svnrdump.c.dump.html)

[svnrdump help](http://svnbook.red-bean.com/en/1.7/svn.ref.svnrdump.c.help.html)

[svnrdump load](http://svnbook.red-bean.com/en/1.7/svn.ref.svnrdump.c.load.html)

[svnserve—Custom Subversion Server](http://svnbook.red-bean.com/en/1.7/svn.ref.svnserve.html)

[svnserve Options](http://svnbook.red-bean.com/en/1.7/svn.ref.svnserve.html#svn.ref.svnserve.sw)

[svndumpfilter—Subversion History Filtering](http://svnbook.red-bean.com/en/1.7/svn.ref.svndumpfilter.html)

[svndumpfilter Options](http://svnbook.red-bean.com/en/1.7/svn.ref.svndumpfilter.html#svn.ref.svndumpfilter.options)

[svndumpfilter Subcommands](http://svnbook.red-bean.com/en/1.7/svn.ref.svndumpfilter.html#svn.ref.svndumpfilter.c)

[svndumpfilter exclude](http://svnbook.red-bean.com/en/1.7/svn.ref.svndumpfilter.commands.c.exclude.html)

[svndumpfilter include](http://svnbook.red-bean.com/en/1.7/svn.ref.svndumpfilter.commands.c.include.html)

[svndumpfilter help](http://svnbook.red-bean.com/en/1.7/svn.ref.svndumpfilter.commands.c.help.html)

[svnversion—Subversion Working Copy Version Info](http://svnbook.red-bean.com/en/1.7/svn.ref.svnversion.html)

[svnversion](http://svnbook.red-bean.com/en/1.7/svn.ref.svnversion.re.html)

[mod\_dav\_svn—Subversion Apache HTTP Server Module](http://svnbook.red-bean.com/en/1.7/svn.ref.mod_dav_svn.html)

[mod\_dav\_svn Configuration Directives](http://svnbook.red-bean.com/en/1.7/svn.ref.mod_dav_svn.conf.html)

[mod\_authz\_svn—Subversion Apache HTTP Authorization Module](http://svnbook.red-bean.com/en/1.7/svn.ref.mod_authz_svn.html)

[mod\_authz\_svn Configuration Directives](http://svnbook.red-bean.com/en/1.7/svn.ref.mod_authz_svn.conf.html)

[Subversion Properties](http://svnbook.red-bean.com/en/1.7/svn.ref.properties.html)

[Versioned Properties](http://svnbook.red-bean.com/en/1.7/svn.ref.properties.html#svn.ref.properties.versioned-props)

[Unversioned Properties](http://svnbook.red-bean.com/en/1.7/svn.ref.properties.html#svn.ref.properties.unversioned-props)

[Repository Hooks](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.html)

[start-commit](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.start-commit.html)

[pre-commit](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.pre-commit.html)

[post-commit](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.post-commit.html)

[pre-revprop-change](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.pre-revprop-change.html)

[post-revprop-change](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.post-revprop-change.html)

[pre-lock](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.pre-lock.html)

[post-lock](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.post-lock.html)

[pre-unlock](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.pre-unlock.html)

[post-unlock](http://svnbook.red-bean.com/en/1.7/svn.ref.reposhooks.post-unlock.html)

**Basic SVN Commands**

Here are the basic SVN commands that every developer and admin should know.

**svn admincreate**

The svn admincreate command creates a new, empty repository.

**svn import**

The svn import command commits an unversioned tree of files into a repository (and creates intermediate directories, if needed).

**svn checkout**

The svn checkout command checks out a working copy from the repository. This command is sometimes shortened to svn co.

**svn commit**

The svn commit command sends your changes back to the [SVN server](https://www.perforce.com/blog/vcs/what-svn).

**svn add**

The svn add command will add a new file to the repository — but only after you've done a svn commit.

**svn delete**

The svn delete command will delete a file from your working copy of the repository.

**svn list**

The svn list command allows you to see a list of files in a repository without creating a working copy.

**svn diff**

The svn diff command reveals the differences between your working copy and the copy in the master SVN repository.

**svn status**

The svn status command prints the status of working copy files and directories.

**svn info**

The svn info command displays information about a local or remote item.

**svn log**

The svn log command shows log messages from the repository.

**svn move**

The svn move command moves a file from one directory to another (or renames it).

**svn merge**

The svn merge command combines two different versions into your working copy.

**svn revert**

The svn revert command reverts changes in your working copy, as well as property changes.

**svn update**

The svn update command updates your working copy with changes from the repository.

**svn shelve**

The svn shelve command stores your changes without submitting them.

**svn help**

The svn help command provides a summary of available commands.

**SVN Commands Cheat Sheet**

Need a handy SVN commands cheat sheet to reference?

Here, we've compiled a quick reference guide to every SVN command you'll need to know. *We've also included the Helix Core P4 command equivalent, if you're considering*[*migrating to Helix Core*](https://www.perforce.com/resources/vcs/why-svn-users-are-migrating-helix-core)*.*

|  |  |  |
| --- | --- | --- |
| **Task** | **SVN Commands** | **Helix Core P4 Commands** |
| Create a new depot/repo. | svnadmin create | p4 depot |
| Add files to the depot/repo. | svn import | p4 reconcile, then p4 submit |
| Discard changes made to open files and revert back to latest synced version. | svn checkout -r <revision> url://path/to/repo | p4 revert |
| Copy files into the client workspace. | svn checkout <URL> <target\_name> | p4 sync, then p4 edit |
| Send changes to the depot. | svn commit | p4 submit |
| Open files in a client workspace to add them to the depot. | svn add <file> , then svn commit | p4 add <file>, then p4 submit |
| Remove a file. | svn delete <file> | p4 delete <file> |
| Obtain list of files. | svn list | p4 files |
| Compare files on the client workspace with revisions in the depot. | svn diff <file> | p4 diff <file> |
| Display information about the workspace files. | svn status | p4 fstat, p4 opened, p4 have, p4 files |
| Display information about the current client and server. | svn info | p4 info |
| Provide information on changelists and changelists’ files. | svn log | p4 describe |
| Move a file. | svn move | p4 move |
| Combine two different revisions. | svn merge | p4 merge |
| Discard changes made to an open file. | svn revert <file or directory> | p4 revert <file or directory> |
| Obtain and update changes from the depot/repo to the client workspace. | svn update | p4 sync |
| Store files without submitting. | svn shelve | p4 shelve |
| Request help. | svn help | p4 help |