Setting up a repository

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

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

git init

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

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

Usage

git init

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

git init <directory>

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

git init --bare <directory>

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

Discussion

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

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

**Bare Repositories**

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

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

## 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.

### Usage

git clone <repo>

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

git clone <repo> <directory>

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

### Discussion

If a project has already been set up in a central repository, the git clone command is the most common way for users to obtain a development copy. Like [git init](https://www.atlassian.com/git/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, you [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 add

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).

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.

Usage

git add <file>

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

git add <directory>

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

git add -p

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

Discussion

The git add and git commit commands compose the fundamental Git workflow. These are the two commands that every Git user needs to understand, regardless of their team’s collaboration model. They are the means to record versions of a project into the repository’s history.

Developing a project revolves around the basic edit/stage/commit pattern. First, you edit your files in the working directory. When you’re ready to save a copy of the current state of the project, you stage changes with git add. After you’re happy with the staged snapshot, you commit it to the project history with git commit.

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

**The Staging Area**

The staging area is one of Git's more unique features, and it can take some time to wrap your head around it if you’re coming from an SVN (or even a Mercurial) background. It helps to think of it as a buffer between the working directory and the project history.

Instead of committing all of the changes you've made since the last commit, the stage lets you group related changes into highly focused snapshots before actually committing it to the project history. This means you can make all sorts of edits to unrelated files, then go back and split them up into logical commits by adding related changes to the stage and commit them piece-by-piece. As in any revision control system, it’s important to create atomic commits so that it’s easy to track down bugs and revert changes with minimal impact on the rest of the project.

Example

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

git add .

git commit

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

git add hello.py

git commit

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

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.

Usage

git commit

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

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

git commit -a

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

Discussion

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

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

**Snapshots, Not Differences**

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

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

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

Example

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

git add hello.py

git commit

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

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

#modified: hello.py

Git doesn't require commit messages to follow any specific formatting constraints, but the canonical format is to summarize the entire commit on the first line in less than 50 characters, leave a blank line, then a detailed explanation of what’s been changed. For example:

Change the message displayed by hello.py

- Update the sayHello() function to output the user's name

- Change the sayGoodbye() function to a friendlier message

Note that many developers also like to use present tense in their commit messages. This makes them read more like actions on the repository, which makes many of the history-rewriting operations more intuitive.

## [Inspecting a repository](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[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](https://www.atlassian.com/git/tutorials/inspecting-a-repository)*[not](https://www.atlassian.com/git/tutorials/inspecting-a-repository)*[show you any information regarding the committed project history. For this, you need to use](https://www.atlassian.com/git/tutorials/inspecting-a-repository)[[git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository)](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-log)[.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Usage](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[List which files are staged, unstaged, and untracked.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Discussion](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[The git status command is a relatively straightforward command. It simply shows you what's been going on with git add and git commit. Status messages also include relevant instructions for staging/unstaging files. Sample output showing the three main categories of a git status call is included below:](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# On branch master](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# Changes to be committed:](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# (use "git reset HEAD <file>..." to unstage)](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#modified: hello.py](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# Changes not staged for commit:](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# (use "git add <file>..." to update what will be committed)](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# (use "git checkout -- <file>..." to discard changes in working directory)](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#modified: main.py](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# Untracked files:](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# (use "git add <file>..." to include in what will be committed)](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[#hello.pyc](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

**[Ignoring Files](https://www.atlassian.com/git/tutorials/inspecting-a-repository)**

[Untracked files typically fall into two categories. They‘re either files that have just been added to the project and haven’t been committed yet, or they're compiled binaries like .pyc, .obj, .exe, etc. While it's definitely beneficial to include the former in the git status output, the latter can make it hard to see what’s actually going on in your repository.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

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

[\*.pyc](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Example](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[It‘s good practice to check the state of your repository before committing changes so that you don’t accidentally commit something you don't mean to. This example displays the repository status before and after staging and committing a snapshot:](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# Edit hello.py](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# hello.py is listed under "Changes not staged for commit"](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git add hello.py](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# hello.py is listed under "Changes to be committed"](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git commit](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[# nothing to commit (working directory clean)](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

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

[git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[The git log command displays committed snapshots. It lets you list the project history, filter it, and search for specific changes. While git status lets you inspect the working directory and the staging area, git log only operates on the committed history.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Log output can be customized in several ways, from simply filtering commits to displaying them in a completely user-defined format. Some of the most common configurations of git log are presented below.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Usage](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[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.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log -n <limit>](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Limit the number of commits by <limit>. For example, git log -n 3will display only 3 commits.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log --oneline](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Condense each commit to a single line. This is useful for getting a high-level overview of the project history.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log --stat](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[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.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log -p](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[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.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log --author="<pattern>"](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Search for commits by a particular author. The <pattern> argument can be a plain string or a regular expression.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log --grep="<pattern>"](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Search for commits with a commit message that matches <pattern>, which can be a plain string or a regular expression.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log <since>..<until>](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[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](https://www.atlassian.com/git/tutorials/inspecting-a-repository)[[revision reference](https://www.atlassian.com/git/tutorials/inspecting-a-repository)](http://www.kernel.org/pub/software/scm/git/docs/gitrevisions.html)[.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log <file>](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Only display commits that include the specified file. This is an easy way to see the history of a particular file.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log --graph --decorate --oneline](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[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.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Discussion](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

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

[commit 3157ee3718e180a9476bf2e5cab8e3f1e78a73b7](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Author: John Smith](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Most of this is pretty straightforward; however, the first line warrants some explanation. The 40-character string after commit is an SHA-1 checksum of the commit’s contents. This serves two purposes. First, it ensures the integrity of the commit—if it was ever corrupted, the commit would generate a different checksum. Second, it serves as a unique ID for the commit.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

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

[The ~ character is useful for making relative references to the parent of a commit. For example, 3157e~1 refers to the commit before 3157e, and HEAD~3 is the great-grandparent of the current commit.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[The idea behind all of these identification methods is to let you perform actions based on specific commits. The git log command is typically the starting point for these interactions, as it lets you find the commits you want to work with.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[Example](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[The](https://www.atlassian.com/git/tutorials/inspecting-a-repository)*[Usage](https://www.atlassian.com/git/tutorials/inspecting-a-repository)*[section provides many examples of git log, but keep in mind that several options can be combined into a single command:](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log --author="John Smith" -p hello.py](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[This will display a full diff of all the changes John Smith has made to the file hello.py.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[The .. syntax is a very useful tool for comparing branches. The next example displays a brief overview of all the commits that are in some-feature that are not in master.](https://www.atlassian.com/git/tutorials/inspecting-a-repository)

[git log --oneline master..some-feature](https://www.atlassian.com/git/tutorials/inspecting-a-repository)