# Setting up a repository

[**git init**](https://www.atlassian.com/git/tutorials/setting-up-a-repository) [**git clone**](https://www.atlassian.com/git/tutorials/setting-up-a-repository) [**git config**](https://www.atlassian.com/git/tutorials/setting-up-a-repository)

This tutorial provides an overview of how to set up a repository (repo) under Git version control. This resource will walk you through initializing a Git repository for a new or existing project. Included below are workflow examples of repositories both created locally and cloned from remote repositories. This guide assumes a basic familiarity with a command-line interface.

The high level points this guide will cover are:

* Initializing a new Git repo
* Cloning an existing Git repo
* Committing a modified version of a file to the repo
* Configuring a Git repo for remote collaboration
* Common Git version control commands

By the end of this module, you should be able to create a Git repo, use common Git commands, commit a modified file, view your project’s history and configure a connection to a Git hosting service (Bitbucket).

## What is a Git repository?

A [Git repository](https://bitbucket.org/product/code-repository) is a virtual storage of your project. It allows you to save versions of your code, which you can access when needed.

## Initializing a new repository: git init

To create a new repo, you'll use the git init command. git init is a one-time command you use during the initial setup of a new repo. Executing this command will create a new .git subdirectory in your current working directory. This will also create a new main branch.

### Versioning an existing project with a new git repository

This example assumes you already have an existing project folder that you would like to create a repo within. You'll first cd to the root project folder and then execute the git init command.

cd /path/to/your/existing/code   
git init

Pointing git init to an existing project directory will execute the same initialization setup as mentioned above, but scoped to that project directory.

git init <project directory>

Visit the [git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) page for a more detailed resource on git init.

## Cloning an existing repository: git clone

If a project has already been set up in a central repository, the clone command is the most common way for users to obtain a local development clone. Like git init, cloning is generally a one-time operation. Once a developer has obtained a working copy, all [version control](https://bitbucket.org/product/version-control-software) operations are managed through their local repository.

git clone <repo url>

git clone is used to create a copy or clone of remote repositories. You pass git clone a repository URL. Git supports a few different network protocols and corresponding URL formats. In this example, we'll be using the Git SSH protocol. Git SSH URLs follow a template of: git@HOSTNAME:USERNAME/REPONAME.git

An example Git SSH URL would be: git@bitbucket.org:rhyolight/javascript-data-store.git where the template values match:

* HOSTNAME: bitbucket.org
* USERNAME: rhyolight
* REPONAME: javascript-data-store

When executed, the latest version of the remote repo files on the main branch will be pulled down and added to a new folder. The new folder will be named after the REPONAME in this case javascript-data-store. The folder will contain the full history of the remote repository and a newly created main branch.

For more documentation on git clone usage and supported Git URL formats, visit the [git clone Page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone).

## Saving changes to the repository: git add and git commit

Now that you have a repository cloned or initialized, you can commit file version changes to it. The following example assumes you have set up a project at /path/to/project. The steps being taken in this example are:

* Change directories to /path/to/project
* Create a new file CommitTest.txt with contents ~"test content for git tutorial"~
* git add CommitTest.txt to the repository staging area
* Create a new commit with a message describing what work was done in the commit

cd /path/to/project   
echo "test content for git tutorial" >> CommitTest.txt   
git add CommitTest.txt   
git commit -m "added CommitTest.txt to the repo"

After executing this example, your repo will now have CommitTest.txt added to the history and will track future updates to the file.

This example introduced two additional git commands: add and commit. This was a very limited example, but both commands are covered more in depth on the [git add](https://www.atlassian.com/git/tutorials/saving-changes) and [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit) pages. Another common use case for git add is the --all option. Executing git add --all will take any changed and untracked files in the repo and add them to the repo and update the repo's working tree.

## Repo-to-repo collaboration: git push

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 source code from an SVN repository. Unlike SVN, Git makes no distinction between the working copies and the central repository—they're all full-fledged [Git repositories](https://bitbucket.org/product/code-repository).

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 or pull commits from one repository to another.

Of course, there’s nothing stopping you from giving certain Git repos special meaning. For example, by simply designating one Git repo as the “central” repository, it’s possible to replicate a centralized workflow using Git. This is accomplished through conventions rather than being hardwired into the VCS itself.

### Bare vs. cloned repositories

If you used git clone in the previous "Initializing a new Repository" section to set up your local repository, your repository is already configured for remote collaboration. git clone will automatically configure your repo with a remote pointed to the Git URL you cloned it from. This means that once you make changes to a file and commit them, you can git push those changes to the remote repository.

If you used git init to make a fresh repo, you'll have no remote repo to push changes to. A common pattern when initializing a new repo is to go to a hosted Git service like Bitbucket and create a repo there. The service will provide a Git URL that you can then add to your local Git repository and git push to the hosted repo. Once you have created a remote repo with your service of choice you will need to update your local repo with a mapping. We discuss this process in the Configuration & Set Up guide below.

If you prefer to host your own remote repo, you'll need to set up a "Bare Repository." Both git init and git clone accept a --bare argument. The most common use case for bare repo is to create a remote central Git repository

## Configuration & set up: git config

Once you have a remote repo setup, you will need to add a remote repo url to your local git config, and set an upstream branch for your local branches. The git remote command offers such utility.

git remote add <remote\_name> <remote\_repo\_url>

This command will map remote repository at  to a ref in your local repo under . Once you have mapped the remote repo you can push local branches to it.

git push -u <remote\_name> <local\_branch\_name>

This command will push the local repo branch under < local\_branch\_name > to the remote repo at < remote\_name >.

For more in-depth look at git remote, see the [Git remote page](https://www.atlassian.com/git/tutorials/syncing#git-remote).

In addition to configuring a remote repo URL, you may also need to set global Git configuration options such as username, or email. The git config command lets you configure your Git installation (or an individual repository) from the command line. This command can define everything from user info, to preferences, to the behavior of a repository. Several common configuration options are listed below.

Git stores configuration options in three separate files, which lets you scope options to individual repositories (local), user (Global), or the entire system (system):

* Local: /.git/config – Repository-specific settings.
* Global: /.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
* System: $(prefix)/etc/gitconfig – System-wide settings.

Define the author name to be used for all commits in the current repository. Typically, you’ll want to use the --global flag to set configuration options for the current user.

git config --global user.name <name>

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

Adding the --local option or not passing a config level option at all, will set the user.name for the current local repository.

git config --local user.email <email>

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

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

Create a shortcut for a Git command. This is a powerful utility to create custom shortcuts for commonly used git commands. A simplistic example would be:

git config --global alias.ci commit

This creates a ci command that you can execute as a shortcut to git commit. To learn more about git aliases visit the [git config page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config).

git config --system core.editor <editor>

Define the text editor used by commands like git commit for all users on the current machine. The < editor > argument should be the command that launches the desired editor (e.g., vi). This example introduces the --system option. The --system option will set the configuration for the entire system, meaning all users and repos on a machine. For more detailed information on configuration levels visit the [git config page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config).

git config --global --edit

Open the global configuration file in a text editor for manual editing. An in-depth guide on how to configure a text editor for git to use can be found on the [Git config page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config).

### Discussion

All configuration options are stored in plaintext files, so the git config command is really just a convenient command-line interface. Typically, you’ll only need to configure a Git installation the first time you start working on a new development machine, and for virtually all cases, you'll want to use the --global flag. One important exception is to override the author email address. You may wish to set your personal email address for personal and open source repositories, and your professional email address for work-related repositories.

Git stores configuration options in three separate files, which lets you scope options to individual repositories, users, or the entire system:

* /.git/config – Repository-specific settings.
* ~/.gitconfig – User-specific settings. This is where options set with the --global flag are stored.
* $(prefix)/etc/gitconfig – System-wide settings.

When options in these files conflict, local settings override user settings, which override system-wide. If you open any of these files, you’ll see something like the following:

[user] name = John Smith email = john@example.com [alias] st = status co = checkout br = branch up = rebase ci = commit [core] editor = vim

You can manually edit these values to the exact same effect as git config.

### Example

The first thing you’ll want to do after installing Git is tell it your name/email and customize some of the default settings. A typical initial configuration might look something like the following:

Tell Git who you are git config

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

Select your favorite text editor

git config --global core.editor vim

Add some SVN-like aliases

git config --global alias.st status   
git config --global alias.co checkout   
git config --global alias.br branch   
git config --global alias.up rebase   
git config --global alias.ci commit

This will produce the ~ /.gitconfig file from the previous section. Take a more in-depth look at git config on the [git config page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config).

## Summary

Here we demonstarted how to create a git repository using two methods: [git init](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init) and [git clone](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone). This guide can be applied to manage software source code or other content that needs to be versioned. [Git add](https://www.atlassian.com/git/tutorials/saving-changes), [git commit](https://www.atlassian.com/git/tutorials/saving-changes/git-commit), [git push](https://www.atlassian.com/git/tutorials/syncing/git-push), and [git remote](https://www.atlassian.com/git/tutorials/syncing) were also introduced and utilized at a high level.

# git config

In this document, we'll take an in-depth look at the git config command. We briefly discussed git config usage on our [Setting up a Repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository) page. The git config command is a convenience function that is used to set Git configuration values on a global or local project level. These configuration levels correspond to .gitconfig text files. Executing git config will modify a configuration text file. We'll be covering common configuration settings like email, username, and editor. We'll discuss Git aliases, which allow you to create shortcuts for frequently used Git operations. Becoming familiar with git config and the various Git configuration settings will help you create a powerful, customized Git workflow.

## Usage

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 levels and files

Before we further discuss git config usage, let's take a moment to cover configuration levels. The git config command can accept arguments to specify which configuration level to operate on. The following configuration levels are available:

* **--local**

By default, git config will write to a local level if no configuration option is passed. Local level configuration is applied to the context repository git config gets invoked in. Local configuration values are stored in a file that can be found in the repo's .git directory: .git/config

* **--global**

Global level configuration is user-specific, meaning it is applied to an operating system user. Global configuration values are stored in a file that is located in a user's home directory. ~ /.gitconfig on unix systems and C:\Users\\.gitconfig on windows

* **--system**

System-level configuration is applied across an entire machine. This covers all users on an operating system and all repos. The system level configuration file lives in a gitconfig file off the system root path. $(prefix)/etc/gitconfig on unix systems. On windows this file can be found at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer.

Thus the order of priority for configuration levels is: local, global, system. This means when looking for a configuration value, Git will start at the local level and bubble up to the system level.

### Writing a value

Expanding on what we already know about git config, let's look at an example in which we write a value:

git config --global user.email "your\_email@example.com"

This example writes the value your\_email@example.com to the configuration name user.email. It uses the --global flag so this value is set for the current operating system user.

## git config editor - core.editor

Many Git commands will launch a text editor to prompt for further input. One of the most common use cases for git config is configuring which editor Git should use. Listed below is a table of popular editors and matching git config commands:

| **Editor** | **config command** |
| --- | --- |
| Atom | ~ git config --global core.editor "atom --wait"~ |
| emacs | ~ git config --global core.editor "emacs"~ |
| nano | ~ git config --global core.editor "nano -w"~ |
| vim | ~ git config --global core.editor "vim"~ |
| Sublime Text (Mac) | ~ git config --global core.editor "subl -n -w"~ |
| Sublime Text (Win, 32-bit install) | ~ git config --global core.editor "'c:/program files (x86)/sublime text 3/sublimetext.exe' -w"~ |
| Sublime Text (Win, 64-bit install) | ~ git config --global core.editor "'c:/program files/sublime text 3/sublimetext.exe' -w"~ |
| Textmate | ~ git config --global core.editor "mate -w"~ |

## Merge tools

In the event of a merge conflict, Git will launch a "merge tool." By default, Git uses an internal implementation of the common Unix diff program. The internal Git diff is a minimal merge conflict viewer. There are many external third party merge conflict resolutions that can be used instead. For an overview of various merge tools and configuration, see our guide on [tips and tools to resolve conflits with Git](https://developer.atlassian.com/blog/2015/12/tips-tools-to-solve-git-conflicts/).

git config --global merge.tool kdiff3

## Colored outputs

Git supports colored terminal output which helps with rapidly reading Git output. You can customize your Git output to use a personalized color theme. The git config command is used to set these color values.

### color.ui

This is the master variable for Git colors. Setting it to false will disable all Git's colored terminal output.

$ git config --global color.ui false

By default, color.ui is set to auto which will apply colors to the immediate terminal output stream. The auto setting will omit color code output if the output stream is redirected to a file or piped to another process.

You can set the color.ui value to always which will also apply color code output when redirecting the output stream to files or pipes. This can unintentionally cause problems since the receiving pipe may not be expecting color-coded input.

### Git color values

In addition to color.ui, there are many other granular color settings. Like color.ui, these color settings can all be set to false, auto, or always. These color settings can also have a specific color value set. Some examples of supported color values are:

* normal
* black
* red
* green
* yellow
* blue
* magenta
* cyan
* white

Colors may also be specified as hexadecimal color codes like #ff0000, or ANSI 256 color values if your terminal supports it.

### Git color configuration settings

1. color.branch

* Configures the output color of the Git branch command

2. color.branch.<slot>

* This value is also applicable to Git branch output. <slot> is one of the following:
  + 1. current: the current branch
  + 2. local: a local branch
  + 3. remote: a remote branch ref in refs/remotes
  + 4. upstream: an upstream tracking branch
  + 5. plain: any other ref

3. color.diff

* Applies colors to git diff, git log, and git show output

4. color.diff.<slot>

* Configuring a <slot> value under color.diff tells git which part of the patch to use a specific color on.
  + 1. context: The context text of the diff. Git context is the lines of text content shown in a diff or patch that highlights changes.
  + 2. plain: a synonym for context
  + 3. meta: applies color to the meta information of the diff
  + 4. frag: applies color to the "hunk header" or "function in hunk header"
  + 5. old: applies a color to the removed lines in the diff
  + 6. new: colors the added lines of the diff
  + 7. commit: colors commit headers within the diff
  + 8. whitespace: sets a color for any whitespace errors in a diff

5. color.decorate.<slot>

* Customize the color for git log --decorate output. The supported <slot> values are: branch, remoteBranch, tag, stash, or HEAD. They are respectively applicable to local branches, remote-tracking branches, tags, stashed changes and HEAD.

6. color.grep

* Applies color to the output of git grep.

7. color.grep. <slot>

* Also applicable to git grep. The <slot> variable specifies which part of the grep output to apply color.
  + 1. context: non-matching text in context lines
  + 2. filename: filename prefix
  + 3. function: function name lines
  + 4. linenumber: line number prefix
  + 5. match: matching text
  + 6. matchContext: matching text in context lines
  + 7. matchSelected: matching text in selected lines
  + 8. selected: non-matching text in selected lines
  + 9. separator: separators between fields on a line (:, -, and =) and between hunks (--)

8. color.interactive

* This variable applies color for interactive prompts and displays. Examples are git add --interactive and git clean --interactive

9. color.interactive.<slot>

* The <slot> variable can be specified to target more specific "interactive output". The available <slot> values are: prompt, header, help, error; and each act on the corresponding interactive output.

10. color.pager

* Enables or disables colored output when the pager is in use

11. color.showBranch

* Enables or disables color output for the git show branch command

12. color.status

* A boolean value that enables or disables color output for Git status

13. color.status.<slot>

Used to specify custom color for specified git status elements. <slot> supports the following values:

* 1. header
  + Targets the header text of the status area
* 2. added or updated
  + Both target files which are added but not committed
* 3. changed
  + Targets files that are modified but not added to the git index
* 4. untracked
  + Targets files which are not tracked by Git
* 5. branch
  + Applies color to the current branch
* 6. nobranch
  + The color the "no branch" warning is shown in
* 7. unmerged
  + Colors files which have unmerged changes

## Aliases

You may be familiar with the concept of aliases from your operating system command-line; if not, they're custom shortcuts that define which command will expand to longer or combined commands. Aliases save you the time and energy cost of typing frequently used commands. Git provides its own alias system. A common use case for Git aliases is shortening the commit command. Git aliases are stored in Git configuration files. This means you can use the git config command to configure aliases.

git config --global alias.ci commit

This example creates a ci alias for the git commit command. You can then invoke git commit by executing git ci. Aliases can also reference other aliases to create powerful combos.

git config --global alias.amend ci --amend

This example creates an alias amend which composes the ci alias into a new alias that uses --amend flag.

## Formatting & whitespace

Git has several "whitespace" features that can be configured to highlight whitespace issues when using git diff. The whitespace issues will be highlighted using the configured color color.diff.whitespace

The following features are enabled by default:

* blank-at-eol highlights orphan whitespaces at the line endings
* space-before-tab highlights a space character that appears before a tab character when indenting a line
* blank-at-eof highlights blank lines inserted at the end of a file

The following features are disabled by default

* indent-with-non-tab highlights a line that is indented with spaces instead of tabs
* tab-in-indent highlights an initial tab indent as an error
* trailing-space is shorthand for both blank-at-eol and blank-at-eof
* cr-at-eol highlights a carriage-return at the line endings
* tabwidth= defines how many character positions a tab occupies. The default value is 8. Allowed values are 1-63

## Summary

In this article, we covered the use of the git config command. We discussed how the command is a convince method for editing raw git config files on the filesystem. We looked at basic read and write operations for configuration options. We took a look at common config patterns:

* How to configure the Git editor
* How to override configuration levels
* How to reset configuration defaults
* How to customize git colors

Overall, git config is a helper tool that provides a shortcut to editing raw git config files on disk. We covered in depth personal customization options. Basic knowledge of git configuration options is a prerequisite for [setting up a repository](https://www.atlassian.com/git/tutorials/setting-up-a-repository). See our guide there for a demonstration of the basics.

# Git Alias

 This section will focus on Git aliases. To better understand the value of Git aliases we must first discuss what an alias is. The term alias is synonymous with a shortcut. Alias creation is a common pattern found in other popular utilities like `bash` shell. Aliases are used to create shorter commands that map to longer commands. Aliases enable more efficient workflows by requiring fewer keystrokes to execute a command. For a brief example, consider the git checkout command. The checkout command is a frequently used git command, which adds up in cumulative keystrokes over time. An alias can be created that maps git co to git checkout, which saves precious human fingertip power by allowing the shorter keystroke form: git co to be typed instead.

## Git Alias Overview

It is important to note that there is no direct git alias command. Aliases are created through the use of the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) command and the Git configuration files. As with other configuration values, aliases can be created in a local or global scope.  
  
To better understand Git aliases let us create some examples.

$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status

The previous code example creates globally stored shortcuts for common git commands. Creating the aliases will not modify the source commands. So git checkout will still be available even though we now have the git co alias. These aliases were created with the --global flag which means they will be stored in Git's global operating system level configuration file. On linux systems, the global config file is located in the User home directory at /.gitconfig.

    [alias]  
        co = checkout  
            br = branch  
            ci = commit  
            st = status

This demonstrates that the aliases are now equivalent to the source commands.

## Usage

Git aliasing is enabled through the use of git config, For command-line option and usage examples please review the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) documentation.

## Examples

### Using aliases to create new Git commands

A common Git pattern is to remove recently added files from the staging area. This is achieved by leveraging options to the git reset command. A new alias can be created to encapsulate this behavior and create a new alias-command-keyword which is easy to remember:

git config --global alias.unstage 'reset HEAD --'

The preceding code example creates a new alias unstage. This now enables the invocation of git unstage. git unstage which will perform a reset on the staging area. This makes the following two commands equivalent.

git unstage fileA  
$ git reset HEAD -- fileA

## Discussion

### How do I create Git Aliases?

Aliases can be created through two primary methods:

#### Directly editing Git config files

The global or local config files can be manually edited and saved to create aliases. The global config file lives at $HOME/.gitconfig file path. The local path lives within an active git repository at /.git/config  
  
The config files will respect an [alias] section that looks like:

[alias]  
 co = checkout

This means that co is a shortcut for checkout

#### Using the git config to create aliases

As previously demonstrated the git config command is a convenient utility to quickly create aliases. The git config command is actually a helper utility for writing to the global and local Git config files.

git config --global alias.co checkout

Invoking this command will update the underlying global config file just as it had been edited in our previous example.

## Git Alias Summary

Git aliases are a powerful workflow tool that create shortcuts to frequently used Git commands. Using Git aliases will make you a faster and more efficient developer. Aliases can be used to wrap a sequence of Git commands into new faux Git command. Git aliases are created through the use of the git config command which essentially modifies local or global Git config files. Learn more on the [git config](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-config) page.

# git init

This page will explore the git init command in depth. By the end of this page you will be informed on the core functionality and extended feature set of git init. This exploration includes:

* git init options and usage
* .git directory overview
* custom git init directory environment values
* git init vs. git clone
* git init bare repositories
* git init templates

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.

Aside from the .git directory, in the root directory of the project, an existing project remains unaltered (unlike SVN, Git doesn't require a .git subdirectory in every subdirectory).

By default, git init will initialize the Git configuration to the .git subdirectory path. The subdirectory path can be modified and customized if you would like it to live elsewhere. You can set the $GIT\_DIR environment variable to a custom path and git init will initialize the Git configuration files there. Additionally you can pass the --separate-git-dir argument for the same result. A common use case for a separate .git subdirectory is to keep your system configuration "dotfiles" (.bashrc, .vimrc, etc.) in the home directory while keeping the .git folder elsewhere.

## Usage

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. Additionally, Git does not require any pre-existing server or admin privileges. All you have to do is cd into your project subdirectory and run git init, and you'll have a fully functional Git repository.

git init

Transform the current directory into a Git repository. This adds a .git subdirectory 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 subdirectory called containing nothing but the .git subdirectory.

If you've already run git init on a project directory and it contains a .git subdirectory, you can safely run git init again on the same project directory. It will not override an existing .git configuration.

### git init vs. git clone

A quick note: git init and git clone can be easily confused. At a high level, they can both be used to "initialize a new git repository." However, git clone is dependent on git init. git clone is used to create a copy of an existing repository. Internally, git clone first calls git init to create a new repository. It then copies the data from the existing repository, and checks out a new set of working files. Learn more on the [git clone page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-clone).

## Bare repositories --- git init --bare

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.

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. You would create a bare repository to git push and git pull from, but never directly commit to it. 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, as 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.

The most common use case for  git init --bare is to create a remote central repository:

ssh <user>@<host> cd path/above/repo git init --bare my-project.git

First, you SSH into the server that will contain your central repository. Then, you navigate to wherever you’d like to store the project. Finally, you use the --bare flag to create a central storage repository. Developers would then clone my-project.git to create a local copy on their development machine.

## git init templates

git init <directory> --template=<template\_directory>

Initializes a new Git repository and copies files from the   into the repository.

Templates allow you to initialize a new repository with a predefined .git subdirectory. You can configure a template to have default directories and files that will get copied to a new repository's .git subdirectory. The default Git templates usually reside in a `/usr/share/git-core/templates` directory but may be a different path on your machine.

The default templates are a good reference and example of how to utilize template features. A powerful feature of templates that's exhibited in the default templates is Git Hook configuration. You can create a template with predefined Git hooks and initialize your new git repositories with common hooks ready to go. Learn more about Git Hooks at the [Git Hook page](https://www.atlassian.com/git/tutorials/git-hooks).

## Configuration

All configurations of git init take a  argument. If you provide the , the command is run inside it. If this directory does not exist, it will be created. In addition to the options and configuration already discussed, Git init has a few other command line options. A full list of them follows:

-Q

--QUIET

Only prints "critical level" messages, Errors, and Warnings. All other output is silenced.

--BARE

Creates a bare repository. (See the "Bare Repositories" section above.)

--TEMPLATE=

Specifies the directory from which templates will be used. (See the "Git Init Templates" section above.)

--SEPARATE-GIT-DIR=

Creates a text file containing the path to . This file acts as a link to the .git directory. This is useful if you would like to store your .git directory on a separate location or drive from your project's working files. Some common use cases for --separate-git-dir are:

* To keep your system configuration "dotfiles" (.bashrc, .vimrc, etc.) in the home directory while keeping the .git folder elsewhere
* Your Git history has grown very large in disk size and you need to move it elsewhere to a separate high-capacity drive
* You want to have a Git project in a publicly accessible directory like `www:root`

You can call git init --separate-git-dir on an existing repository and the .git dir will be moved to the specified  path.

--SHARED[=(FALSE|TRUE|UMASK|GROUP|ALL|WORLD|EVERYBODY|0XXX)]

Set access permissions for the new repository. This specifies which users and groups using Unix-level permissions are allowed to push/pull to the repository.

## Examples

### Create a new git repository for an existing code base

cd /path/to/code \   
git init \   
git add . \   
git commit

### Create a new bare repository

git init --bare /path/to/repo.git

### Create a git init template and initialize a new git repository from the template

mkdir -p /path/to/template \   
echo "Hello World" >> /absolute/path/to/template/README \  
git init /new/repo/path --template=/absolute/path/to/template \   
cd /new/repo/path \   
cat /new/repo/path/README

# git clone

Here we'll examine the git clone command in depth. git clone is a Git command line utility which is used to target an existing repository and create a clone, or copy of the target repository. In this page we'll discuss extended configuration options and common use cases of git clone. Some points we'll cover here are:

* Cloning a local or remote repository
* Cloning a bare repository
* Using shallow options to partially clone repositories
* Git URL syntax and supported protocols

On the [setting up a repository guide](https://www.atlassian.com/git/tutorials/setting-up-a-repository), we covered a basic use case of git clone. This page will explore more complex cloning and configuration scenarios.

## Purpose: repo-to-repo collaboration development copy

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, 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're all full-fledged [Git repositories](https://bitbucket.org/product/code-repository).

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.

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

## Usage

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.

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. This automatic connection is established by creating Git refs to the remote branch heads under refs/remotes/origin and by initializing remote.origin.url and remote.origin.fetch configuration variables.

An example demonstrating using git clone can be found on the [setting up a repository guide](https://www.atlassian.com/git/tutorials/setting-up-a-repository). The example below demonstrates how to obtain a local copy of a central repository stored on a server accessible at example.com using the SSH username john:

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

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

### Cloning to a specific folder

git clone <repo> <directory>

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

### Cloning a specific tag

git clone --branch <tag> <repo>

Clone the repository located at ＜repo＞ and only clone the ref for ＜tag＞.

### Shallow clone

git clone -depth=1 <repo>

Clone the repository located at ＜repo＞ and only clone the   
history of commits specified by the option depth=1. In this example a clone of ＜repo＞ is made and only the most recent commit is included in the new cloned Repo. Shallow cloning is most useful when working with repos that have an extensive commit history. An extensive commit history may cause scaling problems such as disk space usage limits and long wait times when cloning. A Shallow clone can help alleviate these scaling issues.

## Configuration options

### git clone -branch

The -branch argument lets you specify a specific branch to clone instead of the branch the remote HEAD is pointing to, usually the main branch. In addition you can pass a tag instead of branch for the same effect.

git clone -branch new\_feature git://remoterepository.git

This above example would clone only the new\_feature branch from the remote Git repository. This is purely a convenience utility to save you time from downloading the HEAD ref of the repository and then having to additionally fetch the ref you need.

### git clone -mirror vs. git clone -bare

#### git clone --bare

Similar to git init --bare, when the -bare argument is passed to git clone, a copy of the remote repository will be made with an omitted working directory. This means that a repository will be set up with the history of the project that can be pushed and pulled from, but cannot be edited directly. In addition, no remote branches for the repo will be configured with the -bare repository. Like git init --bare, this is used to create a hosted repository that developers will not edit directly.

#### git clone --mirror

Passing the --mirror argument implicitly passes the --bare argument as well. This means the behavior of --bare is inherited by --mirror. Resulting in a bare repo with no editable working files. In addition, --mirror will clone all the extended refs of the remote repository, and maintain remote branch tracking configuration. You can then run git remote update on the mirror and it will overwrite all refs from the origin repo. Giving you exact 'mirrored' functionality.

### Other configuration options

For a comprehensive list of other git clone options visit the [official Git documentation](https://git-scm.com/docs/git-clone). In this document, we'll touch on some other common options.

#### git clone --template

git clone --template=<template\_directory> <repo location>

Clones the repo at ＜repo location＞ and applies the template from ＜template directory＞ to the newly created local branch. A thorough refrence on Git templates can be found on our [git init page](https://www.atlassian.com/git/tutorials/setting-up-a-repository/git-init).

## Git URLs

Git has its own URL syntax which is used to pass remote repository locations to Git commands. Because git clone is most commonly used on remote repositories we will examine Git URL syntax here.

### Git URL protocols

**-SSH**

Secure Shell (SSH) is a ubiquitous authenticated network protocol that is commonly configured by default on most servers. Because SSH is an authenticated protocol, you'll need to establish credentials with the hosting server before connecting. ssh://[user@]host.xz[:port]/path/to/repo.git/

**- GIT**

A protocol unique to git. Git comes with a daemon that runs on port (9418). The protocol is similar to SSH however it has NO AUTHENTICATION. git://host.xz[:port]/path/to/repo.git/

**- HTTP**

Hyper text transfer protocol. The protocol of the web, most commonly used for transferring web page HTML data over the Internet. Git can be configured to communicate over HTTP http[s]://host.xz[:port]/path/to/repo.git/

## Summary

In this document we took a deep look at git clone. The most important takeaways are:  
  
1. git clone is used to create a copy of a target repo

2. The target repo can be local or remote

3. Git supports a few network protocols to connect to remote repos

4. There are many different configuration options available that change the content of the clone

For further, deeper reference on git clone functionality, consult the [official Git documentation](https://git-scm.com/docs/git-clone). We also cover practical examples of git clone in our [setting up a repository guide](https://www.atlassian.com/git/tutorials/setting-up-a-repository).