# The Basics

**Initialize the Git Repository**

$ mkdir myrepo

$ cd myrepo

$ git init

Git is designed to be as unobtrusive as possible. Notice that there is now a .git directory in myrepo that stores all the tracking data for our repository. The .git folder is the only difference between a Git repository and an ordinary folder, so deleting it will turn your project back into an unversioned collection of files

git init can be run without any arguments to create the local Git repository in the current directory. Under the myrepo repository directory, a .git is created with various files and directories in the current directory.

$ find .git

.git/config //Contains the configuration of the local repository

.git/HEAD //Head pointer

.git/hooks

...

.git/objects // Object storage

.git/objects/info

.git/objects/pack

.git/refs

.git/refs/heads //Contains the branch pointers

.git/refs/tags //Contains the tag pointers

**Configure Git**

Git comes with a long list of configurations options from your name to your favorite merge tool. You can set options with the git config or by manually editing a file called .gitconfig in your home directory.

The first thing you do with any Git installation is set you name an email.

$ git config --global user.name "Your Name"

$ git config --global user.email [your.email@example.com](mailto:your.email@example.com)

Once you’ve installed Git, the first thing you need to do is to tell Git your name and email (particularly before creating any commits). Rather than usernames, Git uses a name and an email address to identify the author of a commit.

We can do this with the git config command:

$ git config --global user.name "Your Name"

$ git config --global user.email [your.email@example.com](mailto:your.email@example.com)

The --global option tells Git to use this configuration as a default for all of your repositories. Omitting it lets you specify different user information for individual repositories, which will come in handy later on.

First off, Mary needs to configure her repository so that we know who contributed what to the project.

git config user.name "Mary"

git config user.email mary.example@rypress.com

You may recall from the first module that we used a --global flag to set the configuration for the entire Git installation. But since Mary’s repository is on the local filesystem, she needs a local configuration.

Use a text editor to open up the file called config in the .git folder of Mary’s project (you may need to enable hidden files to see .git). This is where local configurations are stored, and we see Mary’s information at the bottom of the file. Note that this overrides the global configuration that we set in [The Basics](http://rypress.com/tutorials/git/the-basics.html).

Git’s command line relies on a text editor for most of its input. You can forece Git to use your editor of choice with the core.editor option

$ git config --global core.editor notepad

Git supports aliasing commands

$ git config --global alias.st status

## The fundamental Git workflow



The stage/commit process

The distinction between the working directory, the staged snapshot, and committed snapshots is at the core of Git version control. Nearly all other Git commands manipulate one of these components in some way, so understanding the interplay between them is a fantastic foundation for mastering Git.

Most Git commands operate on one of the three main components of a Git repository: the working directory, the staged snapshot, or the committed snapshots.

The working directory is where the developer edit files and compile code.

The staging area is an intermediary between the working directory and the local repository

A Git *repository* is the local collection of all the files related to a particular Git version control system and contains a .git subdirectory in its root. Git keeps track of the state of the files in the repository’s directory on disk.

Git repositories store all their data on your local machine. Making commits, viewing history, and requesting differences between commits are all local operations that don’t require a network connection. This makes all these operations much faster in Git than with centralized version control systems such as Subversion.

Under the new Git repository directory, a .git subdirectory is created with various files and directories under it.

**View the Repository Status**

It would be helpful to view the status of the new repository.

$ git status

$ git status -s

It outputs the state of the working directory and staging aea

It’s a good practice to run git status to see exactly what you’re committing before running git commit

**Stage a Snapshot**

$ echo “abcd” > file1.txt

$ git add file1.txt

$ git status

git add command tells Git to start tracking file1.txt. Git add file1.txt to the snapshot for the next commit. A snapshot represents the state of your project at a given point in time. Git’s term for creating a snapshot is called **staging**. We can add or remove multiple files before actually committing it to the project history.

Git’s staging area gives you a plcae to organize a commit before adding it to the project history. Staging is the process of moving changes from the working directory to the staged snapshot. It gives you the opportunity to pick and choose related changes from the working directory, instead of committing everything all at once

To delete a file from project, you need to add it to the staging area like a new or modified file.The next command will stage the deletion and stop tracking the file but it won’t delete the file from he working directory

$ git rm --cached file1.txt

If you need more detailed information about the changes in your working directory or staging area, you can generate a diff

$ git diff

This outputs a diff of every unstaged changes in your working directory. You can also generate a diff of all staged changes with the --cached flag

$ git diff --cached

**Commit the Snapshot**

Commits represent every saved version of a project which makes them the atomic unit of Git based version control.

The next command will open a text editor and prompt you to enter a message for the commit.

$ git commit

The -m option lets you specify a commit message on the command line instead of opening a text editor. This is just a convenient shortcut.

$ git commit -m "Add navigation links"

The --message flag for git commit can be abbreviated to -m (all abbreviations use a single -). If this flag is omitted, Git opens a text editor (specified by the EDITOR or GIT\_EDITOR environment variable) to prompt you for the commit message

It can also take the --all (or -a) flag to add all changes to files tracked in the repository into a new commit

Saving a version of your project is a two step process:

* **Staging.** Telling Git what files to include in the next commit.
* **Committing.** Recording the staged snapshot with a descriptive message.

Staging files with the git add command doesn’t actually affect the repository in any significant way—it just lets us get our files in order for the next commit. Only after executing git commit will our snapshot be recorded in the repository. Committed snapshots can be seen as “safe” versions of the project.

Our history can now be represented as the following. Note that the red circle, which represents the current commit, automatically moves forward every time we commit a new snapshot.



Current project history

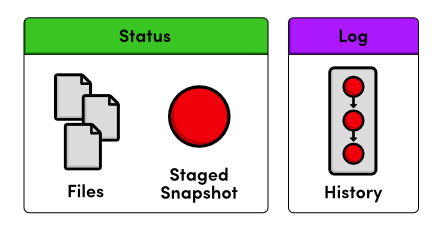
Notice that we skipped the staging step this time around. Instead of using git add, we passed the -a flag to git commit. This convenient parameter tells Git to automatically include *all* tracked files in the staged snapshot. Combined with the -m flag, we can stage and commit snapshots with a single command



Rather than require all changes in the working tree to build up new commits, git allows files to be added incrementally to the index.

**View the Repository History**

The git status command will *only* show us *staged* changes. To view our project history (*committed* changes), we need a new command git log



Status output vs. Log output

The git log command comes with a lot of formatting options. For now, we’ll just use the convenient --oneline flag. Git outputs only the first 7 characters of the checksum. These first few characters effectively serve as a unique ID for each commit

$ git log --oneline

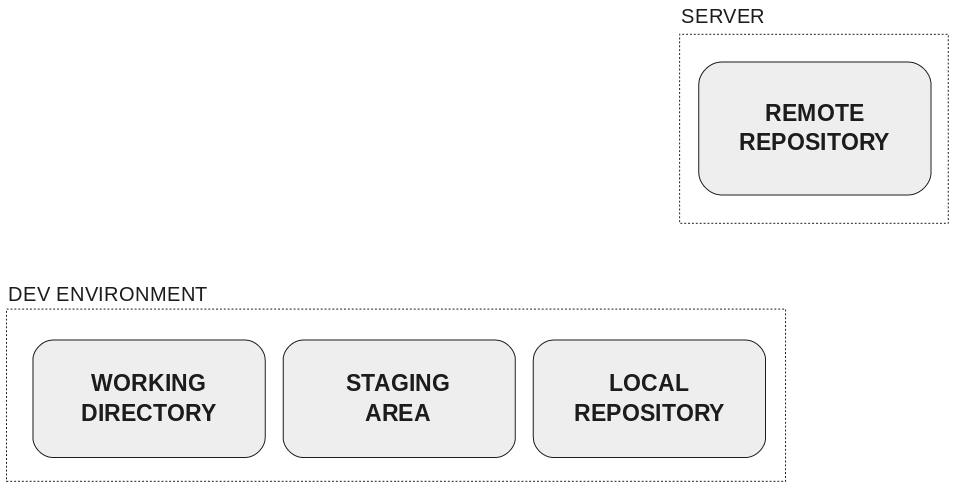
Condensing output to a single line is a great way to get a high-level overview of a repository. Another useful configuration is to pass a filename to git log:

$ git log --oneline file1.txt

This displays only the file1.txt history.



* All files present in the staging area are committed
* At a given time, different versions of a specific file can exist in any of the three areas



## Updating the remote repository

As changes are ready in the **working directory**, they must be added in the **staging area**.

When there is a set of changes with a single purpose in the **staging area**, it’s the time to create a commit with a message about that purpose in the **local repository**.

When there are one or several commits in the **local repository** ready to be shared with the rest of the world, they must be pushed to the **remote repository**.

At that time, you can talk about the different states of a file in the development environment: **modified**, **staged** and **committed**.

<https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting>

## Setting up a repository

A [Git repository](http://bitbucket-marketing.atlassian.com/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 master branch.

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>*

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.

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.

### 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.

All configurations of git init <directory> take a <directory> argument. If you provide the <directory>, 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:

### Cloning an existing repository: 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.

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

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](http://bitbucket-marketing.atlassian.com/product/version-control-software) operations are managed through their local repository.

git clone <repo url>

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.

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.

### git clone -branch

The -branch argument lets you specify a specific a branch to clone instead of the branch the remote HEAD is pointing to, usually the master 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 convince 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.

### 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).

## 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](http://bitbucket-marketing.atlassian.com/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

### 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.

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.