



# Architecture & Deployment

2025-2026 v0.1.0 on branch main Rev: d1f684699a511826485586b2ea383cf021631add

## Version Control with Git

Learn the basics of [Git](#), one of the most popular distributed version control systems. This is a condensed version of the first chapters of the [Git Book](#), which you should read if you want more detailed information on the subject.

### Table of contents

- [Presentation](#)
- [Getting started](#)
  - [Git and the command line](#)
    - [Installing Git](#)
  - [First-time Git setup](#)
  - [Choosing a default branch name](#)
  - [Creating a new repository](#)
  - [Checking the status of your files](#)
  - [Adding new files](#)
    - [Tracking new files](#)
    - [Checking staged changes](#)
  - [Committing your changes](#)
  - [Modifying files](#)
    - [Staging modified files](#)
    - [Modifying a staged file](#)
    - [Checking staged and unstaged changes](#)
    - [Staging area versus working directory](#)
    - [Committing partially staged changes](#)
  - [Moving and removing files](#)
    - [Adding all changes](#)
- [Viewing the commit history](#)
  - [Viewing the changes in the history](#)
  - [Other log options](#)

- [Ignoring files](#)
  - [Committing the ignore file](#)
  - [Status of ignored files](#)
  - [Global ignore file](#)
- [Undoing things](#)
  - [Unmodifying a modified file](#)
  - [Unstaging a staged file](#)
  - [Changing the commit message](#)
  - [Adding changes to a commit](#)
- [Best practices](#)

## You will need

- A Unix CLI

## Recommended reading

- [Command line](#)

# Getting started

---

This is a tutorial where you will learn how to:

- Configure Git for the first time
- Create a new repository
- Check the status of your files
- Track new files
- Stage and commit modified files
- Move and remove files

- Ignore files

## Git and the command line

---

There are a lot of different ways to use Git: the original **command line tools** and various **GUIs** of varying capabilities. But the command line is the only place you can run **all** Git commands with all their options.

If you know how to run the command line version, you can easily figure out how to use the GUI version, while the opposite is not necessarily true. So the **command line** is what we will use.

Some of you may already have Git installed. Run the following command in your CLI to make sure:

```
$> git --version  
git version 2.46.0
```

## Installing Git

---

On **macOS**, Git may already be installed. If not, it is part of the command-line tools, which you can install by running the following command:

```
$> xcode-select --install
```

On **Windows** under the **Windows Subsystem for Linux (WSL)**, or on **Linux**, Git may already be installed. If not, you can install it using your distribution's package manager, for example on APT-based systems:

```
$> sudo apt install git-all
```

Otherwise, follow the [official installation instructions](#).

# First-time Git setup

---

Now that you have Git, you must configure your **identity**: your user name and e-mail address. This is important because **every Git commit uses this information**, and it's *immutably* baked into every commit you make.

Use the `git config` command to do this:

```
$> git config --global user.name "John Doe"
$> git config --global user.email john.doe@example.com
```

You can also run the command with the `--list` option to check that the settings were successfully applied:

```
$> git config --list
user.name=John Doe
user.email=john.doe@example.com
```

## More information

With the `--global` option, Git will store these settings in your user configuration file (`~/.gitconfig`), so you only need to do this **once on any given computer**. You can also change them at any time by running the commands again. Run `cat ~/.gitconfig` to display this file.

## Choosing a default branch name

---

You should also configure a default branch name, like `main`, for all your repositories:

```
$> git config --global init.defaultBranch main
```

We will talk more about branches later. If you don't perform this configuration now, you may see this warning when creating your first repository:

```
$> git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
```

## Creating a new repository

---

Let's get started by creating a directory for our new project:

```
$> cd /path/to/projects
$> mkdir hello-project
```

Go into the directory and run `git init` to create a Git repository:

```
$> cd hello-project
$> git init
Initialized empty Git repository in /path/to/projects/hello-project
```

This creates a Git directory (`.git`) with an empty object database. At this point, nothing in your project is tracked yet.

# Checking the status of your files

---

The main tool you use to determine which files are in which state is the `git status` command. If you run it in the repo you just created, you should see something like this:

```
$> git status
On branch main

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

This means you have an empty repo with no commits, and a **clean working directory** – there is nothing there.

As you can see, Git often helps you by telling you what you can do next: you need to start adding some files.

## Tip

The `git status` command is your best friend when using Git. Do not hesitate to use it at any time to check in what state you are.

# Adding new files

---

In the project's directory, write “Hello World” into a `hello.txt` file and “Hi Bob” into a `hi.txt` file:

```
$> echo "Hello World" > hello.txt
$> echo "Hi Bob" > hi.txt
```

Re-run the `git status` command:

```
$> git status  
On branch main
```

```
Initial commit
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
hello.txt
```

```
hi.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Those files are **untracked**. Git will not include them in the repository unless you **explicitly** tell it to do so.

## Tracking new files

---

In order to begin tracking a new file, you must use the `git add` command:

```
$> git add hello.txt
```

```
$> git add hi.txt
```

```
$> git status
```

```
On branch main
```

```
Initial commit
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file:   hello.txt
```

```
new file:   hi.txt
```

The files are now **staged**: they will be in the next commit.

## Tip

- `git add *.txt` would have added all files with the `.txt` extension in one command.
- `git add .` would have added all the files in the current directory (recursively).

## Checking staged changes

---

Git can show you what you have **staged**:

```
$> git diff --staged
```

```
diff --git a/hello.txt b`hello.txt`
new file mode 100644
index 0000000..557db03
--- /dev/null
+++ b/hello.txt
@@ -0,0 +1 @@
+Hello World

diff --git a/hi.txt b`hi.txt`
new file mode 100644
index 0000000..e5db1d9
--- /dev/null
+++ b/hi.txt
@@ -0,0 +1 @@
+Hi Bob
```

It shows you each staged file and the changes in those files.

## Committing your changes

---

Now that your staging area is set up the way you want it, you can **commit** your changes with the `git commit` command. This command takes a `--message` or `-m` option

where you should put a short description of the changes you made:

```
$> git commit -m "Add hello and hi files"
```

```
[main (root-commit) `c90aa36`] Add hello and hi files
2 files changed, 2 insertions(+)
create mode 100644 hello.txt
create mode 100644 hi.txt
```

Note that Git gives you the beginning of the new commit's SHA-1 checksum (`c90aa36` in this example, but it will be different on your machine) along with change statistics and other information.

```
$> git status
On branch main
nothing to commit, working tree clean
```

## Modifying files

---

Let's make some changes. Add one line to both files:

```
echo "You are beautiful" >> hello.txt
echo "Hi Jane" >> hi.txt
```

And see what Git tells us:

```
$> git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

modified:   hello.txt
```

```
modified:    hi.txt
```

no changes added to commit (use "git add" and/or "git commit -a")

Git has identified the **modified files** different from the last commit, but they are **not staged**, meaning that if you try to commit now, those changes will **not** be committed.

## Staging modified files

---

Stage the changes on the `hello.txt` file and check the status:

```
$> git add hello.txt
```

```
$> git status
```

On branch main

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

```
modified:    hello.txt
```

Changes not staged **for** commit:

(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes **in** working directory)

```
modified:    hi.txt
```

If you commit now, only the changes on `hello.txt` will be included in the snapshot, while the changes in `hi.txt` will remain uncommitted.

## Modifying a staged file

---

Before committing, let's make another change to `hello.txt` and check the status:

```
$> echo "I see trees of green" >> hello.txt
```

```
$> git status
```

```
On branch main
```

```
Changes to be committed:
```

```
(use "git restore --staged <file>..." to unstage)
```

```
modified:   hello.txt
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git restore <file>..." to discard changes in working directory)
```

```
modified:   hello.txt
```

```
modified:   hi.txt
```

`hello.txt` is shown both under “Changes to be committed” and “Changes not staged for commit”. What does this mean?

## Checking staged and unstaged changes

---

Use `git diff` with the `--staged` option to show **staged** changes.

```
$> git diff --staged
```

```
diff --git a/hello.txt b/hello.txt
```

```
index 557db03..2136a8e 100644
```

```
--- a/hello.txt
```

```
+++ b/hello.txt
```

```
@@ -1 +1,2 @@
```

```
    Hello World
```

```
+You are beautiful
```

You can also use it without the option to see **unstaged** changes.

```
$> git diff
```

```
diff --git a/hello.txt b/hello.txt
```

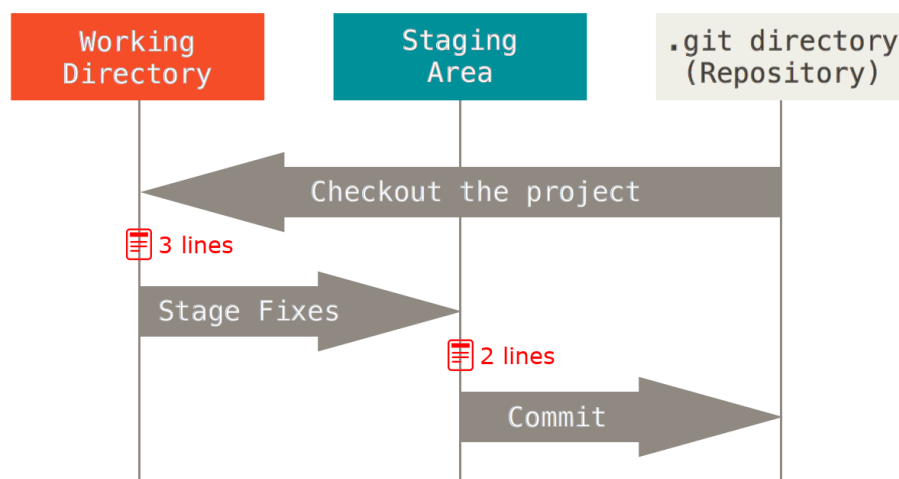
```
index 2136a8e..730ea5a 100644
```

```
--- a/hello.txt
+++ b/hello.txt
@@ -1,2 +1,3 @@
Hello World
You are beautiful
+I see trees of green
diff --git a/hi.txt b/hi.txt
index e5db1d9..f74a87a 100644
--- a/hi.txt
+++ b/hi.txt
@@ -1 +1,2 @@
Hi Bob
+Hi Jane
```

## Staging area versus working directory

This example shows you that the working directory and the staging area are really two separate steps:

- The version of `hello.txt` you have **staged** contains two lines of text (“Hello World” and “You are beautiful”). This is what will be committed.
- The version of `hello.txt` in the **working directory** has an additional line of text (“I see trees of green”) which you added later. It will not be included in the next commit unless you stage the file again.



## Committing partially staged changes

---

Commit now:

```
$> git commit -m "The world is beautiful"
[main b65ec9c] The world is beautiful
1 file changed, 1 insertion(+)
```

As expected, the changes we did not stage are still **uncommitted**.

```
$> git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

modified:   hello.txt
modified:   hi.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Let's fix that:

```
$> git add .
$> git commit -m "New lines in hello.txt and hi.txt"
[main dfc6c75] New lines in hello.txt and hi.txt
2 files changed, 2 insertions(+)
```

## Moving and removing files

---

Git has a `git mv` (**move**) and `git rm` (**remove**) command, but nobody uses them for day-to-day work on files. It's simpler to just move or remove the files yourself. Rename `hi.txt` to `people.txt` in your editor or with this command:

```
$> mv hi.txt people.txt
```

## More information

The `mv` (move) command moves a source file to a target path.

Then see what Git tells you:

```
$> git status
On branch main
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)

deleted:    hi.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

people.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

## Adding all changes

---

You can tell Git to add all changes (additions, modifications and removals):

```
$> git add .

$> git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
```

```
renamed:    hi.txt -> people.txt
```

Note that Git can now tell that the file was moved.

### Tip

Many developers simply modify and manipulate files in their favorite editor or IDE, then use the command above.

You may commit the rename now:

```
$> git commit -m "Rename hi.txt to people.txt"
```

## Viewing the commit history

---

Git has a very powerful `log` command:

```
$> git log
commit 739b7c8987d72879f79ac7979as8f9db790a82da
Author: John Doe <john.doe@example.com>
Date:   Mon Jan 23 11:50:09 2017 +0100
```

```
    Rename hi.txt to people.txt
```

```
commit e753ceb86806b285aa105a846c7295e826439637
Author: John Doe <john.doe@example.com>
Date:   Mon Jan 23 11:50:07 2017 +0100
```

```
    New lines in hello.txt and hi.txt
```

```
commit 4c56257f622c53f1ddeaf3d58b6729b01b35aedb
Author: John Doe <john.doe@example.com>
```

Date: Mon Jan 23 11:50:00 2017 +0100

The world is beautiful

...

## Viewing the changes in the history

---

With the `--patch` option, you can see that Git shows you the differences you introduced in each commit:

```
$> git log --patch
commit e753ceb86806b285aa105a846c7295e826439637
Author: John Doe <john.doe@example.com>
Date: Mon Jan 23 11:50:07 2017 +0100
```

New lines in hello.txt and hi.txt

```
diff --git a/hello.txt b/hello.txt
```

```
index 2136a8e..730ea5a 100644
```

```
--- a/hello.txt
```

```
+++ b/hello.txt
```

```
@@ -1,2 +1,3 @@
```

```
Hello World
```

```
You are beautiful
```

```
+I see trees of green
```

```
diff --git a/hi.txt b/hi.txt
```

```
index e5db1d9..f74a87a 100644
```

```
--- a/hi.txt
```

```
+++ b/hi.txt
```

```
@@ -1 +1,2 @@
```

```
Hi Bob
```

```
+Hi Jane
```

## Other log options

---

The `git log` has many options to customize its output or limit what commits it shows you. Here are some other useful options:

| Option                | Limit to                                                 |
|-----------------------|----------------------------------------------------------|
| <code>--stat</code>   | Show the list of changed files                           |
| <code>--pretty</code> | Show the commit history with a <u>custom format</u>      |
| <code>-(n)</code>     | Only the last n commits                                  |
| <code>--after</code>  | Only commits made after the specified date               |
| <code>--before</code> | Only commits made before the specified date              |
| <code>--author</code> | Only commits whose author matches the specified string   |
| <code>--grep</code>   | Only commits with a commit message containing the string |
| <code>-S</code>       | Only commits adding or removing code matching the string |

Use `git help log` or read [the documentation](#) to learn more.

## Ignoring files

---

Sometimes there are files you don't want to commit in your repository:

- Log files
- Dependencies
- Build artifacts

You can tell Git not to track them by adding a `.gitignore` file to your repository. Create it now with this content:

```
**.*log  
*node_modules
```

## Committing the ignore file

---

Do not forget to stage and commit the `.gitignore` file:

```
$> git add .gitignore  
$> git commit -m "Ignore file"
```

### Tip

That way, when you start collaborating with the other developers in your team, the same files will be ignored on their machine.

## Status of ignored files

---

Ignored files are no longer shown when using `git status`:

```
$> echo data > app.log  
  
$> git status  
On branch main  
nothing to commit, working tree clean
```

## Global ignore file

---

There are **some files you might want to always ignore** for all projects on your machine.

For example, macOS creates `.DS_Store` files in directories you open in the Finder. There is no reason to keep these files in your Git history, and they are useless on other operating systems.

You can create a **global ignore file** in your home directory to ignore them:

```
$> echo ".DS_Store" >> ~/.gitignore
```

Run the following command to configure Git to use this file. You only have to do it once on each machine:

```
$> git config --global core.excludesfile ~/.gitignore
```

`.DS_Store` files will no longer show up in your `git status` output, and they will not be staged or committed.

## Undoing things

---

There are several ways of undoing things with Git. We'll review a few of the tools available.



*Be careful:* you can't always undo some of these operations.

## Unmodifying a modified file

---

Sometimes you make a change and you realize it was wrong or you don't need it anymore. Git actually tells you what to do to discard that change:

```
$> echo "Hi Steve" >> people.txt
```

```
$> git status
```

```
On branch main
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
`(use "git restore <file>..." to discard changes in working directory)`
```

```
modified:   people.txt
```

no changes added to commit (use "git add" and/or "git commit -a")

Simply use `git restore` as instructed:

```
$> git restore people.txt
```

```
$> git status
```

```
On branch main
```

```
nothing to commit, working tree clean
```

Note that in this case, **the change is forever lost** as it was never committed.

## Unstaging a staged file

---

If you have staged a file but realize you don't want it in the next commit anymore, Git also tells you what to do:

```
$> echo "Hi Steve" >> people.txt
```

```
$> git add people.txt
```

```
$> git status
```

```
On branch main
```

```
Changes to be committed:
```

```
  (use "git restore --staged <file>..." to unstage)
```

```
modified:   people.txt
```

Use `git restore` as instructed:

```
$> git restore --staged people.txt
```

The changes will still be in the file in the working directory. If you want to completely get rid of them, you can use `git restore` as shown before.

## Changing the commit message

---

Commit a new change:

```
$> echo Wolf >> people.txt
$> git add people.txt
$> git commit -m "Fix teh prblme"
```

Oops, you've used the wrong commit message. Want to change it?

```
$> git commit --amend -m "Fix the problem"
```



***Be careful:*** this changes the commit and its SHA-1 hash. You should not do this if you have already shared this commit with others.

## Adding changes to a commit

---

Make two changes but only commit one of them:

```
$> echo a > a.txt
$> echo b > b.txt
$> git add a.txt
$> git commit -m "Add a & b"
```

Oops, you forgot to stage one file. Want to add it to the last commit?

```
$> git add b.txt
```

```
$> git commit --amend
```

Your editor will open to give you the opportunity to change the message if you want, but you do not have to. Simply save and exit the editor. The changes to `b.txt` will now also be in the last commit.



**Be careful:** this changes the commit and its SHA-1 hash. You should not do this if you have already shared this commit with others.

## Best practices

---

- **Commit early and often, perfect later** (Seth Robertson)

Git only takes full responsibility for your data when you commit. If you fail to commit and then do something poorly thought out, you can run into trouble. Additionally, having periodic checkpoints means that you can understand how you broke something.

- **Writing a good commit message** (GitKraken)

If by taking a quick look at previous commit messages, you can discern what each commit does and why the change was made, you're on the right track. But if your commit messages are confusing or disorganized, then you can help your future self and your team by improving your commit message practices with help from this article.

- **Conventional Commits**

If you want to go further, look at *Conventional Commits*, a specification for adding human and machine readable meaning to commit messages.