University of Tehran
Engineering college

Git tutorial

Mehrshad Hekmatara

How to initialize our project(make a repository)?

The command git init is used to create a new Git repository in your project folder.

What it does:

- It creates a hidden folder called .git inside your project directory.
- That folder stores all the version history, configurations, and branches for your project.
- After running it, your project becomes a Git-enabled repository.

```
git init
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test
$ git init
Initialized empty Git repository in D:/programming/documents/gitdcvdf/test/.git/
```

Output :

Initialized empty Git repository in D:/programming/documents/git/test-project/.git/

Notes:

- You usually run git init once when starting a new project.
- If you clone an existing repository (git clone), you don't need git init (because the repo already has .git).

👉Inshort:
git init = turn this folder into a Git project.

current state of your working directory

```
git status
```

The command shows you the current state of your working directory and staging area.

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git status
On branch master

No commits yet

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

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

It helps you know:

- Which files are **untracked** (new files not yet added to Git)
- Which files are **modified** but not staged
- Which files are **staged** and ready for commit
- Which branch you're currently on

Summary

- **Untracked** → Git doesn't know about it yet
- **Modified** → File changed but not added
- **Staged** → File will be included in the next commit

👉 Think of git status as your **daily checklist** before committing.

```
git add <filename>       # add a single file
git add .                # add all changes in the current directory
git add -A               # add all changes (including deletions)
```

The command git add moves changes from the **working directory** → to the **staging area**. That means you're telling Git:

"I want this file (or these files) to be included in my next commit."

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git add new.txt

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git add -A

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   new.txt
        new file:   sfns.txt
        new file:   test.txt


Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git add .
```

Key Idea

- **Working Directory → Staging Area → Repository**
- git add is the **step in the middle**

👉 Without git add, changes stay in your working directory and won't be saved in the next commit.

```
git rm --cached <filename>
```

This command tells Git to remove a file from the staging area (and from version control) but keep it in your working directory.

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git rm --cached new.txt
rm 'new.txt'

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   sfns.txt
        new file:   test.txt

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

In simple words:

"Stop tracking this file in Git, but don't delete it from my computer."

When to use it

- You accidentally added a file with git add, but you don't want to commit it.
- You want Git to ignore a file (e.g., config.json, .env, or large temporary files).

```
git commit -m "message"
```

This command **records a snapshot of the changes in the staging area** into the repository history, with a descriptive message.

Breakdown

- **git commit** → saves everything that's in the staging area
- **-m "message"** → attaches a message describing what you changed

Why the message is important

- It explains **why** you made the changes.
- It helps you (and teammates) understand the project history later.

commit history

```
git log
```

Shows the **full commit history** of your repository. It includes:

- Commit ID (a long hash like a3b2c1d...)
- Author
- Date
- Commit message

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git commit -m 'first commit'
[master (root-commit) 39b7e61] first commit
 4 files changed, 8 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 new.txt
 create mode 100644 sfns.txt
 create mode 100644 test.txt

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git status
On branch master
nothing to commit, working tree clean

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log
commit 39b7e6143b8f46686f78174471fa1b3ef0fb6b6a (HEAD -> master)
Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
Date:   Sun Aug 17 11:07:55 2025 +0330

    first commit
```

git log --oneline

A **shorter, cleaner version** of git log.

- Each commit is shown on one line
- Uses a shortened commit hash
- Shows only the commit message

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log --oneline
39b7e61 (HEAD -> master) first commit
```

git log --stat

Shows commit history **with a summary of file changes** in each commit.

It tells you:

- Which files were changed
- How many lines were added or deleted
- A summary of total changes per commit

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log --stat
commit 39b7e6143b8f46686f78174471fa1b3ef0fb6b6a (HEAD -> master)
Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
Date:   Sun Aug 17 11:07:55 2025 +0330

    first commit

 .gitignore | 1 +
 new.txt    | 3 +++
 sfns.txt   | 1 +
 test.txt   | 3 +++
 4 files changed, 8 insertions(+)
```

git log --graph

This command shows your Git commit history **as a visual ASCII graph** alongside the log. It displays branches and merges in a tree-like structure, so you can easily see how commits relate to each other.

What you get:

- Lines and characters representing branches and merges
- Commit hashes and messages displayed next to the graph
- Clear visualization of how your project history branches and merges

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/git/test-project (mast
er)
$ git log --graph
* commit 49c19485e81e0673605ec455dae76339994e26ee (HEAD -> master)
| Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
| Date:   Wed Aug 6 12:20:02 2025 +0330
|
|     creating test variable
|
* commit 3ae365549d0698e15de1d0887baeb9d7a75c90f7
  Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
  Date:   Wed Aug 6 12:02:28 2025 +0330

      create basic files
```

Why use it?

- Makes understanding complex branching and merging easier
- Helpful when working with feature branches and pull requests
- Great for visualizing your project's history at a glance

```
git log --oneline --graph
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log --oneline --graph
* 39b7e61 (HEAD -> master) first commit
```

This command shows a **compact, one-line-per-commit** history combined with a **visual ASCII graph** of branches and merges.

What it does:

- **--graph**: Draws an ASCII art tree showing branches and merges.
- **--oneline**: Shows each commit as a single line with a short commit hash and the commit message.

Combine with `--all` to show commits from all branches:

```
git log --graph --oneline --all
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log --oneline --graph --all
* 39b7e61 (HEAD -> master) first commit
```

```
git log --before="23-10-12"
```

Shows all commits **made before** the specified date (October 12, 2023). It filters commit history to only include commits **earlier** than that date.

```
git log --after="23-10-12"
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log --before="23-10-12"

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log --after="23-10-12"
commit 39b7e6143b8f46686f78174471fa1b3ef0fb6b6a (HEAD -> master)
Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
Date:   Sun Aug 17 11:07:55 2025 +0330

    first commit
```

Shows all commits **made after** the specified date (October 12, 2023). It filters commit history to only include commits **later** than that date.

```
git log --author="Mehrshad Hekmatara"
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git log --author="Mehrshad Hekmatara"
commit 39b7e6143b8f46686f78174471fa1b3ef0fb6b6a (HEAD -> master)
Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
Date:   Sun Aug 17 11:07:55 2025 +0330

    first commit
```

This command filters the commit history to show **only the commits made by the author whose name matches "Mehrshad Hekmatara"**.

How it works:

- Git looks at the **author name** metadata in each commit.

- Only commits where the author's name exactly or partially matches the string "Mehrshad Hekmatara" will be displayed.

```
git commit -am "another test code"
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git status
On branch master
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:   new.txt

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

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git commit -am "modified new "
[master c6abf31] modified new
 1 file changed, 3 insertions(+), 1 deletion(-)

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Breakdown of the command:

- **git commit**: Create a new commit with your changes.
- **-a** : Automatically **stage** all tracked files that have been modified or deleted.
    - Note: It **does NOT** add new untracked files.
- **-m "another test code"**: Adds the commit message "another test code" directly in the command (no editor opens).

What happens when you run this:

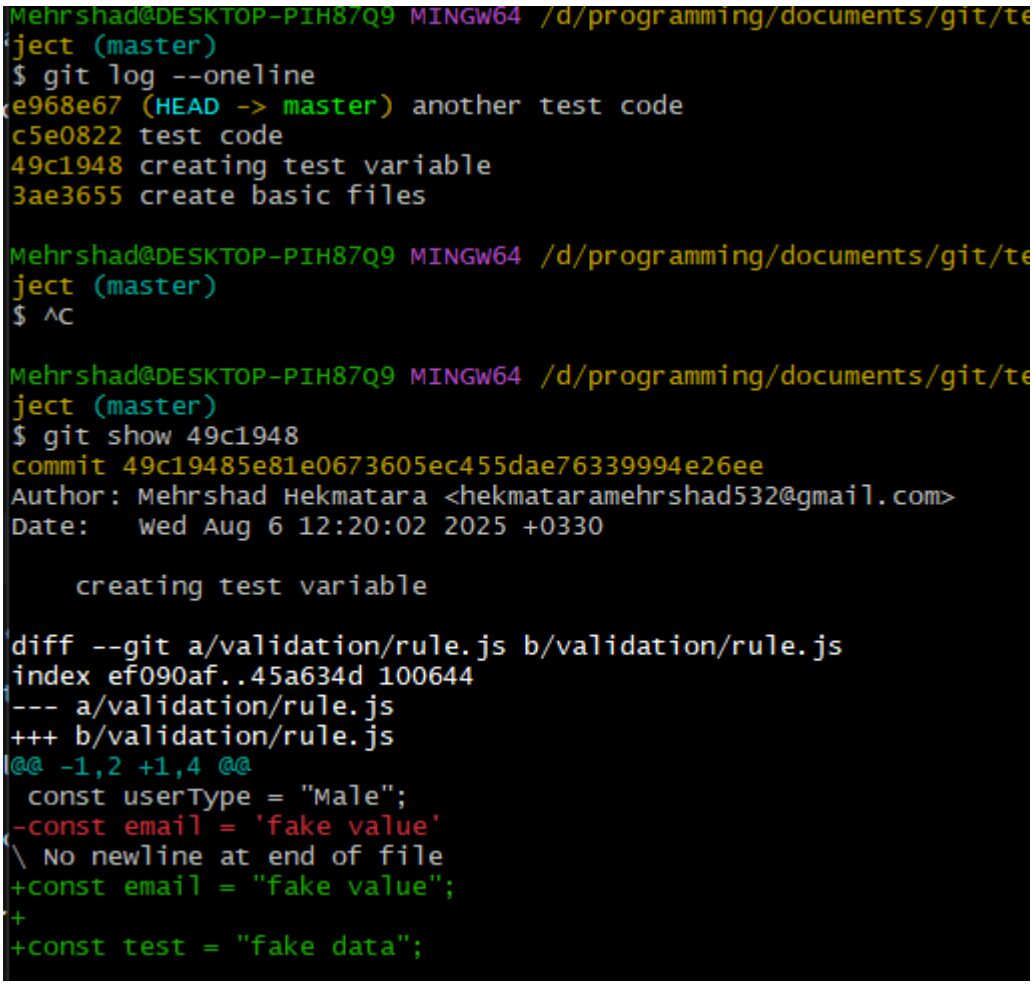
1. Git stages all changes to files that are already being tracked (modified or deleted).
2. Git skips the staging step for new files—you must add new files separately with git add.
3. Git creates a commit with the given message "another test code".

When to use:

- When you want to quickly commit all your changes to tracked files without running git add manually.
- Good for fast commits during development if you don't have new files to add.

New files (untracked) are NOT included; you must `git add` them first.

git show <commit id>



Shows detailed information about a specific commit identified by <commit id> (usually the commit hash).

Output includes:

- Commit metadata (author, date, commit message)
- The diff (changes) introduced by that commit — what lines were added, removed, or modified.

git show

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git show
commit c6abf31cda4a4f09786435071b01d6fbbe8736c3 (HEAD -> master)
Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
Date:   Sun Aug 17 11:20:03 2025 +0330

    modified new

diff --git a/new.txt b/new.txt
index 533eecc..4f2819a 100644
--- a/new.txt
+++ b/new.txt
@@ -1,3 +1,5 @@
 hello

-fef
\ No newline at end of file
+fef
+
+ml;;ll
\ No newline at end of file
```

Without a commit id, git show shows details about the **latest commit** (the HEAD commit).

Useful to quickly see what was changed in the most recent commit.

```
git show <commit-id> <filename>
```

See only one specific file changes in one specific commit.

Shows the **contents of the specified file (<filename>) exactly as it was in the given commit (<commit-id>)**.

Instead of showing the entire commit details or full diff, it extracts and displays just that file's snapshot from that commit.

Use case:

- You want to see how a specific file looked at a particular commit, without browsing the entire project history or diff.
- Helpful for comparing past versions of a file or recovering old code snippets.

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/git/test-project (master)
$ git show e968e67 scroll.js
commit e968e679672847a7ab6c5f5d2ca4900b4d80246e (HEAD -> master)
Author: Mehrshad Hekmatara <hekmataramehrshad532@gmail.com>
Date:   Wed Aug 6 15:28:02 2025 +0330

    another test code

diff --git a/scroll.js b/scroll.js
index 06457a1..8e8dd0d 100644
--- a/scroll.js
+++ b/scroll.js
@@ -1 +1,3 @@
-// test code
\ No newline at end of file
+// test code
+
+// another test code
\ No newline at end of file
```

alias

```
git config --local alias.lgo "log --oneline"
```

This creates a **Git alias** called lgo for the command git log --oneline.

So after running it, you can type:

git lgo instead of git log --oneline .

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git config --local alias.lgo "log --oneline"

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git lgo
c6abf31 (HEAD -> master) modified new
39b7e61 first commit
```

```
git config --global alias.lgo "log --oneline"
```

Creates a **Git alias** called lgo that runs: git log –oneline

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git config --global alias.lgo "log --oneline"

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git lgo
c6abf31 (HEAD -> master) modified new
39b7e61 first commit
```

The --global flag means this alias will be available in **all repositories** for your current user account.

git config → The Git command for setting configuration values.

--global → Saves the setting in your **global config file**.

Viewing your global aliases

```
git config --global --get-regexp alias
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git config --global --get-regexp alias
alias.lgo log --oneline
```

Removing the alias

```
git config --global --unset alias.lgo
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git config --global --unset alias.lgo

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git config --global --get-regexp alias
```

What is a Branch in Git?

A **branch** is like an independent **line of development** in your project's history. Think of it as a **parallel timeline** where you can make changes without affecting other timelines until you decide to merge them.

Core Ideas

1. **Separate Workflows**
    - Each branch contains its own version of the code and its own commit history.
    - You can experiment, add features, or fix bugs without touching the main code.
2. **Pointer to a Commit**
    - Technically, a branch is just a pointer (a reference) to the latest commit in that branch's history.
    - As you make new commits, the branch pointer moves forward.
3. **Collaboration Tool**

- Multiple people can work on different branches at the same time without overwriting each other's work.

4. **Merging and Integration**
   - When a branch's work is ready, it can be merged back into another branch (e.g., main or develop).
5. **Default Branch**
   - Every repository starts with a default branch (commonly called main or master).
   - Other branches are created from this or other branches as needed.

- **Why Branches Are Important**
- **Isolation** → Work on features, fixes, or experiments safely.
- **Parallel Development** → Multiple developers can work at once.
- **Organized History** → Changes are grouped logically by feature or task.
- **Safe Rollback** → If something breaks, you can simply abandon a branch without harming the main code.

Analogy:
Imagine your project's history as a river. The **main branch** is the main flow of water. When you create a new branch, it's like creating a side channel where you can work independently. Later, you can merge that channel back into the main river.

```
git branch
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch
* master
```

This command show us **List** of existing branches.

```
git branch <branchName>
```

What it does

This creates a **new branch** in your local repository with the name you provide.

Local only:

The branch exists only in your local repository until you push it to a remote.

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch newBranch

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch
* master
  newBranch
```

Does not switch to the new branch:

- You stay on your current branch.
- To start working on the new branch, you must explicitly **switch** to it:

```
git switch <branchName>
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git switch newBranch
Switched to branch 'newBranch'

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (newBranch)
$ git branch
  master
* newBranch
```

```
git switch -c <branchName>
```

Creates a **new branch and** immediately switches to it — all in one step.

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git switch -c newbranch
Switched to a new branch 'newbranch'

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (newbranch)
$ git branch
  master
* newbranch
```

-c means "create":

- If the branch doesn't exist, -c tells Git to create it.
- If it already exists, Git will throw an error unless you use other options to handle it.

```
git branch -d <branchName>
```

Deletes a local branch (not remote) — but only if it has already been merged into the branch you are currently on (or another specified branch).

This is Git's safe delete mode.

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch -d newbranch2
Deleted branch newbranch2 (was c6abf31).

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch
* master
  newBranch
```

If you really want to delete regardless of merge status, use:

```
git branch -D <branchName>
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch -D newBranch
Deleted branch newBranch (was c6abf31).
```

It's best practice to:

1. Merge your feature branch into main/develop.
2. Delete the branch locally (git branch -d ...).
3. Delete it remotely if no longer needed.

Hint:

For example, we have two branches: master and cart.
We switch to the cart branch, and then create a new branch using the appropriate command.
The newly created branch is a branch off of cart, not master.

```
git branch -m branchname
```

is used to **rename the current branch** you're on.

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (newbranch)
$ git branch
  master
* newbranch

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (newbranch)
$ git branch -m new

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (new)
$ git branch
  master
* new
```

Breakdown:

- **git branch** → deals with branches.
- **-m** → means "move" (rename) the branch.
- **branchname** → the new name you want to give your current branch.

Important Notes

- You must **already be on** the branch you want to rename if you only provide the new name.
- If you want to rename a different branch without switching to it, you can use:

```
git branch -m oldName newName
```

```
Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch
* master
  new

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch -m new newbr

Mehrshad@DESKTOP-PIH87Q9 MINGW64 /d/programming/documents/gitdcvdf/test (master)
$ git branch
* master
  newbr
```

Renaming a branch only changes the **local branch name**. If it's already pushed to a remote, you'll need to update the remote branch separately.