Git Handbook

10 minute read

What's a version control system?

A version control system, or VCS, tracks the history of changes as people and teams collaborate on projects together. As the project evolves, teams can run tests, fix bugs, and contribute new code with the confidence that any version can be recovered at any

time. Developers can review project history to find out:

- Which changes were made?
- Who made the changes?
- When were the changes made?
- Why were changes needed?

What's a distributed version control system?

Git is an example of a distributed version control system (DVCS) commonly used for open source and commercial software development. DVCSs allow full access to every file,

branch, and iteration of a project, and allows every user access to a full and self-contained history of all changes. Unlike once popular centralized version control systems, DVCSs like Git don’t need a constant connection to a central repository. Developers can work anywhere and collaborate asynchronously from any time zone.

Without version control, team members are subject to redundant tasks, slower timelines, and multiple copies of a single project. To eliminate unnecessary work, Git and other VCSs give each contributor a unified and consistent view of a project, surfacing work that’s already

in progress. Seeing a transparent history of changes, who made them, and how they contribute to the development of a project helps team members stay aligned while working independently.

Why Git?

According to the latest Stack Overflow developer survey, more than 70 percent of developers use Git, making it the most-used VCS in the world. Git is commonly used for both open source and commercial software development, **with significant benefits** for individuals, teams and businesses.

- Git lets developers see the entire timeline of their changes, decisions, and progression of any project in one place. From the moment they access the history of a project, the developer has all the context they need to understand it and start contributing.
- Developers work in every time zone. With a DVCS like Git, collaboration can happen any time while maintaining source code integrity. Using branches, developers can safely propose changes to production code.
- Businesses using Git can break down

> communication barriers between teams and keep them focused on doing their best work. Plus, Git makes it possible to align experts across a business to collaborate on major projects.

What's a repository?

A *repository*, or Git project, encompasses the entire collection of files and folders associated with a project, along with each file's revision history. The file history appears as snapshots in time called *commits*, and the commits exist as a linked-list relationship, and can be organized into multiple lines

of development called *branches*. Because Git is a DVCS, repositories are self-contained units and anyone who owns a copy of the repository can access the entire codebase and its history. Using the command line or other ease-of-use interfaces, a git repository also allows for: interaction with the history, cloning, creating branches, committing, merging, comparing changes across versions of code, and more.

Working in repositories keeps development projects organized and protected. Developers are encouraged to fix bugs, or create fresh features, without fear of derailing mainline

development efforts. Git facilitates this through the use of topic branches: lightweight pointers to commits in history that can be easily created and deprecated when no longer needed.

Through platforms like GitHub, Git also provides more opportunities for project transparency and collaboration. Public repositories help teams work together to build the best possible final product.

Basic Git commands

To use Git, developers use specific commands to copy, create, change, and combine

code. These commands can be executed directly from the command line or by using an application like GitHub Desktop or Git Kraken. Here are some common commands for using Git:

- `git init` initializes a brand new Git repository and begins tracking an existing directory. It adds a hidden subfolder within the existing directory that houses the internal data structure required for version control.
- `git clone` creates a local copy of a project that already exists remotely. The clone includes all the project's

files, history, and branches.

- `git add` stages a change. Git tracks changes to a developer's codebase, but it's necessary to stage and take a snapshot of the changes to include them in the project's history. This command performs staging, the first part of that two-step process. Any changes that are staged will become a part of the next snapshot and a part of the project's history. Staging and committing separately gives developers complete control over the history of their project

without changing how they code and work.

- `git commit` saves the snapshot to the project history and completes the change-tracking process. In short, a commit functions like taking a photo. Anything that's been staged with `git add` will become a part of the snapshot with `git commit`.
- `git status` shows the status of changes as untracked, modified, or staged.
- `git branch` shows the branches being worked on locally.

- `git merge` merges lines of development together. This command is typically used to combine changes made on two distinct branches. For example, a developer would merge when they want to combine changes from a feature branch into the master branch for deployment.
- `git pull` updates the local line of development with updates from its remote counterpart. Developers use this command if a teammate has made commits to a branch on a remote, and they would like to reflect those changes in their local environment.

- `git push` updates the remote repository with any commits made locally to a branch.

Learn more from a full reference guide to Git commands.

Explore more Git commands

For a detailed look at Git practices, the videos below show how to get the most out of some Git commands.

- Working locally
- `git status`
- Two-step commits
- `git pull` and `git pu`

How GitHub fits in

GitHub is a Git hosting repository that provides developers with tools to ship better code through command line features, issues (threaded discussions), pull requests, code review, or the use of a collection of free and for-purchase apps in the GitHub Marketplace. With collaboration layers like the GitHub flow, a community of 15 million developers, and an ecosystem with hundreds of integrations, GitHub changes the way software is built.

How GitHub works

GitHub builds collaboration directly into the development

process. Work is organized into repositories, where developers can outline requirements or direction and set expectations for team members. Then, using the GitHub flow, developers simply create a branch to work on updates, commit changes to save them, open a pull request to propose and discuss changes, and merge pull requests once everyone is on the same page.

The GitHub flow

The GitHub flow is a lightweight, branch-based workflow built around core Git commands used by teams around the globe—including ours.

The GitHub flow has six steps, each with distinct benefits when implemented:

1. **Create a branch:** Topic branches created from the canonical deployment branch (usually `master`) allow teams to contribute to many parallel efforts. Short-lived topic branches, in particular, keep teams focused and results in quick ships.
2. **Add commits:** Snapshots of development efforts within a branch create safe, revertible points in the project's history.
3. **Open a pull request:** Pull requests publicize a project's ongoing efforts

and set the tone for a transparent development process.

4. **Discuss and review code:**Teams participate in code reviews by commenting, testing, and reviewing open pull requests. Code review is at the core of an open and participatory culture.
5. **Merge:** Upon clicking merge, GitHub automatically performs the equivalent of a local ‘git merge’ operation. GitHub also keeps the entire branch development history on the merged pull request.
6. **Deploy:** Teams can choose the best release cycles or incorporate

continuous integration tools and operate with the assurance that code on the deployment branch has gone through a robust workflow.

Learn more about the GitHub flow

Developers can find more information about the GitHub flow in the resources provided below.

- Interactive guide
- GitHub Flow video

GitHub and the command line

For developers new to the command line, the GitHub Training team has put

together a series of tutorials on Git commands to guide the way. Sometimes, a series of commands can paint a picture of how to use Git:

Example: Contribute to an existing repository

```
# download a repository on Gi
git clone https://github.com/

# change into the `repo` dire
cd repo

# create a new branch to stor
git branch my-branch

# switch to that branch (line
git checkout my-branch

# make changes, for example,

# stage the changed files
git add file1.md file2.md

# take a snapshot of the stag
git commit -m "my snapshot"

# push changes to github
git push --set-upstream origi
```

Example: Start a new repository and publish it to GitHub

First, you will need to create a new repository on GitHub. You can learn how to create a new repository in our Hello World guide. **Do not** initialize the repository with a README, .gitignore or License. This empty repository will await your code.

```
# create a new directory, and
git init my-repo

# change into the `my-repo` d
cd my-repo

# create the first file in th
touch README.md

# git isn't aware of the file
git add README.md

# take a snapshot of the stag
```

```
git commit -m "add README to

# provide the path for the re
git remote add origin https:/

# push changes to github
git push --set-upstream origi
```

Example: contribute to an existing branch on GitHub

```
# assumption: a project calle

# change into the `repo` dire
cd repo

# update all remote tracking
git pull

# change into the existing br
git checkout feature-a

# make changes, for example,

# stage the changed file
git add file1.md

# take a snapshot of the stag
git commit -m "edit file1"

# push changes to github
git push
```

Models for collaborative development

There are two primary ways people collaborate on GitHub:

1. Shared repository
2. Fork and pull

With a *shared repository*, individuals and teams are explicitly designated as contributors with read, write, or administrator access. This simple permission structure, combined with features like protected branches and Marketplace, helps teams progress quickly when they adopt GitHub.

For an open source project, or for projects to which anyone can contribute, managing individual permissions can be challenging, but a *fork and pull*model allows anyone who can view the project to contribute. A fork is a copy of a project under an developer's personal account. Every developer has full control of their fork and is free to implement a fix or new feature. Work completed in forks is either kept separate, or is surfaced back to the original project via a pull request. There, maintainers can review the suggested changes before they're merged. See

the Forking Projects Guide for more information.

Learn at your own pace

The GitHub team has created a library of educational videos and guides to help users continue to develop their skills and build better software.

- Beginner projects to explore
- GitHub video guides
- GitHub on-demand training
- GitHub training guides
- GitHub training resources