# About Git

Learn about the version control system, Git, and how it works with GitHub Enterprise Server.

## About version control and Git

A version control system, or VCS, tracks the history of changes as people and teams collaborate on projects together. As developers make changes to the project, any earlier version of the project 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?

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.

In a distributed version control system, every developer has a full copy of the project and project history. Unlike once popular centralized version control systems, DVCSs don't need a constant connection to a central repository. Git is the most popular distributed version control system. 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.

## About repositories

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. The commits can be organized into multiple lines of development called branches. Because Git is a DVCS, repositories are self-contained units and anyone who has 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 the repository, creating branches, committing, merging, comparing changes across versions of code, and more.

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

## How GitHub Enterprise Server works

GitHub Enterprise Server hosts Git repositories and 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 Enterprise Server flow, a community of 100 million developers, and an ecosystem with hundreds of integrations, GitHub Enterprise Server changes the way software is built.

GitHub Enterprise Server 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 Enterprise Server 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. For more information, see "GitHub flow."

For GitHub plans and costs, see GitHub Pricing. For information on how GitHub Enterprise compares to other options, see Comparing GitHub to other DevOps solutions.

## GitHub Enterprise Server and the command line

### 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. 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 main 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.

For more information, see the full reference guide to Git commands.

### Example: Contribute to an existing repository

```
# download a repository on GitHub Enterprise Server to our machine
# Replace `owner/repo` with the owner and name of the repository to clone
git clone https://github.com/owner/repo.git

# change into the `repo` directory
cd repo

# create a new branch to store any new changes
git branch my-branch

# switch to that branch (line of development)
git checkout my-branch

# make changes, for example, edit `file1.md` and `file2.md` using the text editor

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

# take a snapshot of the staging area (anything that's been added)
git commit -m "my snapshot"

# push changes to github
git push --set-upstream origin my-branch
```

### Example: Start a new repository and publish it to GitHub Enterprise Server

First, you will need to create a new repository on GitHub Enterprise Server. For more information, see "Hello World." **Do not** initialize the repository with a README, .gitignore or License file. This empty repository will await your code.

```
# create a new directory, and initialize it with git-specific functions
git init my-repo

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

# create the first file in the project
touch README.md

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

# take a snapshot of the staging area
```

```
git commit -m "add README to initial commit"

# provide the path for the repository you created on github
git remote add origin https://github.com/YOUR-USERNAME/YOUR-REPOSITORY-NAME.git

# push changes to github
git push --set-upstream origin main
```

### Example: contribute to an existing branch on GitHub Enterprise Server

This example assumes that you already have a project called `repo` on the machine and that a new branch has been pushed to GitHub Enterprise Server since the last time changes were made locally.

```
# change into the `repo` directory
cd repo

# update all remote tracking branches, and the currently checked out branch
git pull

# change into the existing branch called `feature-a`
git checkout feature-a

# make changes, for example, edit `file1.md` using the text editor

# stage the changed file
git add file1.md

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

# push changes to github
git push
```

## Models for collaborative development

There are two primary ways people collaborate on GitHub Enterprise Server:

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, helps teams progress quickly when they adopt GitHub Enterprise Server.

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 a developer's personal account. Every developer has full control of their fork and is free to implement a fix or a 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. For more information, see "Contributing to projects."