**Git Basics**

A **distributed**[**version control system**](https://about.gitlab.com/topics/version-control/)**(DVCS)** brings a local copy of the complete repository to every team member’s computer, so they can commit, branch, and merge locally. The server doesn’t have to store a physical file for each branch — it just needs the differences between each commit.

Distributed source code management systems, such as Git, Mercurial, and Bazaar, mirror the repository and its entire history as a local copy on individual hard drives.

[Distributed version control systems](https://about.gitlab.com/blog/2020/10/02/distributed-version-control/) help software development teams create strong workflows and hierarchies, with each developer pushing code changes to their own repository and maintainers setting a [code review process](https://about.gitlab.com/topics/version-control/what-is-code-review/) to ensure only quality code merges into the main repository.

A DVCS can be puzzling, especially if a team member is accustomed to [centralized source code systems](https://about.gitlab.com/topics/version-control/what-is-centralized-version-control-system/), because a contributor can no longer rely on a server to resolve conflicts when merging and has to resolve them locally, which can result in confusing merge commits. However, despite the initial discomfort, a distributed source control system can ensure stable code development when multiple developers contribute to software development projects.

**Advantages:**

By developing a culture that prioritizes collaboration and leverages the full capabilities of modern version control tools, organizations can ensure their position at the forefront of software development innovation. Some specific advantages are as follows:

**Reliable backup copies**

An interesting way to think about distribution version control is to visualize a collection of backups. When a team member clones a repository, she essentially creates an offsite backup, so if something catastrophic happens, like a server crash, every team member’s local copy becomes a backup. Unlike a centralized version control system, a distributed version control removes the reliance on a single backup, making development more reliable. A common misconception is that multiple copies could be a waste of space, but most development includes plain text files, and many systems compress files, so the impact on hard drive storage is minimal.

**Fast merging and flexible branching**

Because systems don’t require remote server communication, code can be quickly merged. A distributed version control also allows software development teams to use different branching strategies, a feature that isn’t possible with a centralized system. Distributed version control systems accelerate delivery and business value by helping team members focus on innovation rather than become bogged down with slow builds.

**Rapid feedback and fewer merge conflicts**

A DVCS makes branching easy, because having an entire repository’s history on their local workstation ensures that they can quickly experiment and [request a code review](https://about.gitlab.com/blog/2020/06/08/better-code-reviews/). Developers benefit from fast feedback loops and can share changes with team members before merging the changeset. Merge conflicts are less likely, because contributors focus on their own piece of code. Furthermore, having easy access to the full local history helps developers identify bugs, track changes, and revert to previous versions.

**Flexibility to work offline**

A distributed version control system doesn’t require an internet connection, so most development, except pushing and pulling, can be done while traveling or away from home or an office. Contributors can view the running history on their hard drive, so any changes will be made in their own repository. This increased flexibility enables team members to fix bugs as a single changeset. Increased developer productivity

With a local copy, developers can complete common development activities rapidly. A DVCS means that developers no longer have to wait on a server run through routine tasks, which can slow down delivery and cause frustration.

**Git: An example of a distributed version control system**

Git is a distributed version control system known for its speed, workflow compatibility, and open-source foundation. With Git, software teams can experiment without fearing that they’ll create lasting damage to the source code. Teams using a Git repository can tackle projects of any size with [efficiency and speed](https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control).

**Set-up Git in your system**

**1. Install Git**

**For Windows:**

1. Download the Git installer from the [official Git website](https://git-scm.com/).
2. Run the installer and follow the steps:

* Choose the default editor for Git (e.g., Vim, Nano, or your favorite text editor).
* Select Use Git from the command line and recommended options.
* Enable Git Credential Manager if prompted.

1. Complete the installation.

**For macOS:**

1. Open Terminal and run: bash brew install git

*(Install Homebrew first if you don’t have it by visiting* [*brew.sh*](https://brew.sh/)*)*

**For Linux:**

1. Use your package manager. For example:

bash

sudo apt update

sudo apt install git For Debian/Ubuntu

sudo yum install git For RHEL/CentOS

**2. Configure Git**

After installation, configure Git with your user details (name and email) used for commits.

bash

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

git config --global user.email "your.email@example.com"

**3. Verify Installation**

Run the following commands to ensure Git is installed and configured correctly:

bash

git --version

git config –list

**4. Set Default Editor (Optional)**

If you want to set a specific editor for Git, such as Visual Studio Code, run:

bash

git config --global core.editor "code --wait"

**5. Set Up SSH (Optional for GitHub/GitLab)**

To enable secure communication with remote repositories:

1. Generate an SSH key: bash

ssh-keygen -t ed25519 -C "your.email@example.com"

1. Add the SSH key to the SSH agent: bash

eval "$(ssh-agent -s)"

ssh-add ~/.ssh/id\_ed25519

1. Copy the SSH key to your clipboard: bash cat ~/.ssh/id\_ed25519.pub
2. Add the SSH key to your Git provider (e.g., GitHub/GitLab).

**6. Test Git with a Repository**

1. Clone a repository to verify setup: bash git clone https://github.com/user/repo.git
2. Navigate to the repository folder: bash cd repo

**Create Branches and Track Files:**

Creating branches and tracking files in Git is essential for efficient version control.

**1. Create a Branch:** Branches allow you to work on features or fixes without affecting the main codebase.

**Creating a new branch:** git branch branch\_name

**Switch to the new branch:** git checkout branch\_name

**Shortcut: Combine both steps using:** git checkout -b branch\_name

**2. View All Branches:**

To see the list of branches: git branch

The current branch is highlighted with an asterisk (`\*`).

**3. Merge Branches:** When you're done working on a branch, merge it into the main branch.

1. Switch to the branch you want to merge into (e.g., `main`): git checkout main

2. Merge the other branch: git merge branch\_name

**4. Delete a Branch:**

After merging, you can delete the branch: git branch -d branch\_name

To force delete (if the branch isn't merged): git branch -D branch\_name

**5. Track Files:** Git tracks files to include them in version control.

**Add files to tracking**

**1. Add a specific file:** git add filename

**2. Add all changes (new, modified, and deleted files):** git add.

**6. Commit Changes:** Commits save a snapshot of tracked changes.

**1. Create a commit with a message:** git commit -m "Your commit message"

**2. If you want to commit all tracked changes in one step:**

git commit -am "Your commit message"

**7. Check File Status:** View the status of files in the repository: git status

Untracked files: New files not yet added to Git.

Changes not staged for commit: Modified files not yet staged with **git add**.

**8. View Commit History:** To see the history of commits: git log

**Example Workflow**

1. Create and switch to a branch: git checkout -b feature-xyz

2. Make changes to files.

3. Stage the changes: git add.

4. Commit the changes: git commit -m "Implemented feature XYZ"

5. Switch back to the main branch and merge: git checkout main git merge feature-xyz

**Creating a repository in Git and GitHub:**

**1. Create a Repository in Git (Locally)**

1. **Initialize a Repository**: Open a terminal or command prompt, navigate to your project directory, and run: git init

This creates a. git folder, which tracks your project changes.

1. **Add Files**: Add files to your repository using: git add <

**2. Create a Repository on GitHub**

1. **Sign in to GitHub**: Go to [GitHub](https://github.com) and log in with your credentials.
2. **Create a New Repository**:

* Click the **+** icon in the top-right corner and select **New Repository**.
* Fill in the details:
* **Repository Name**: Enter a name (e.g., my-repo).
* **Description** *(optional)*: Add a short description.
* Choose **Public** or **Private** visibility.
* (Optional) Check the boxes to initialize with:
* **README**: A basic introduction file.
* **.gitignore**: To ignore unnecessary files.
* **License**: For open-source repositories.
* Click **Create Repository**.

**3. Connect Local Git Repository to GitHub**

1. **Clone (if starting from GitHub)**: If you initialized the repository on GitHub with files (e.g., README), clone it to your local machine: git clone <repository-url>

**Example:** git clone https://github.com/your-username/my-repo.git

cd my-repo

1. **Link Your Local Repository to GitHub**: If you already have a local Git repository and want to connect it to GitHub:

* Add the remote repository:
* git remote add origin <repository-url>

**Example:** git remote add origin https://github.com/your-username/my-repo.git

1. **Push Local Repository to GitHub**: Push your local repository's files to GitHub:

git branch -M main # Rename the branch to 'main' (optional)

git push -u origin main # Push files to GitHub and set the upstream branch

**4. Verify**

* Go to your GitHub repository page to see the pushed files.
* From now on, use git push to upload changes to GitHub after committing locally.

**Common Commands Recap:**

| **Action** | **Command** |
| --- | --- |
| Initialize Git locally | git init |
| Add files | git add <file-name> or git add . |
| Commit changes | git commit -m "Commit message" |
| Add remote repository | git remote add origin <repository-url> |
| Push to GitHub | git push -u origin main |

**Merging in Git:**

Merging combines the changes from one branch into another (typically, feature branches into the main branch). Here's how to do it:

**Steps for Merging**

1. **Ensure You're on the Target Branch**: Switch to the branch where you want to merge changes. git checkout main
2. **Merge the Branch**: Merge the source branch (e.g., feature-branch) into the target branch (e.g., main): git merge feature-branch
3. **Resolve Merge Conflicts** (if any):

* If Git cannot automatically merge changes, it will mark the conflicts in the files.
* Open the conflicting files, resolve the conflicts, and mark them as resolved:

git add <filename>

* Then commit the merge: git commit

1. **Push Changes to Remote (GitHub)**: git push origin main

**Fast-Forward vs. Recursive Merge**

* **Fast-Forward Merge**: When the target branch is directly ahead of the source branch, Git moves the branch pointer forward.
* **Recursive Merge**: Used when branches have diverged; Git creates a merge commit to combine the changes.

**Cloning in Git:** Cloning creates a local copy of a remote Git repository. This is often the starting point for working on an existing project.

**Steps for Cloning**

1. **Get the Repository URL**:

* Go to the repository's page on GitHub.
* Click the **Code** button and copy the HTTPS, SSH, or GitHub CLI URL.

1. **Clone the Repository**: In your terminal or command prompt, run:

git clone <repository-url>

**Example:** git clone https://github.com/your-username/repository-name.git

1. **Navigate to the Cloned Repository**: cd repository-name
2. **Start Working**: You now have a full copy of the repository, including its history and branches.

**Cloning and Merging Together:** When working collaboratively:

1. Clone the repository to your machine:

git clone https://github.com/your-organization/repository-name.git

1. Create a new branch for your work:

git checkout -b my-feature

1. After making changes, push your branch to GitHub:

git push origin my-feature

1. Open a **Pull Request** on GitHub to merge your changes into the main branch.

**Common Commands Recap:**

| **Action** | **Command** |
| --- | --- |
| Merge branches | git merge <branch-name> |
| Clone a repository | git clone <repository-url> |
| View merge conflicts (if any) | Open the conflicting files and resolve manually |
| Add resolved files | git add <file-name> |
| Push merged changes to remote | git push origin <branch-name> |

**Resolving Conflicts in Git**

Conflicts occur when two branches modify the same part of a file, and Git cannot automatically decide which change to apply. Here's how to resolve conflicts:

**Steps to Resolve Conflicts**

1. **Pull Changes and Trigger Conflict**: If you're merging branches (e.g., feature-branch into main), conflicts may arise: git merge feature-branch

Or during a pull: git pull origin main

1. **Identify Conflict Files**: Git will list files with conflicts in the terminal or by running:

git status

1. **Open and Resolve Conflicts**:

* Conflicting files will have markers like:

<<<<<<< HEAD

Your changes

=======

Changes from the other branch

>>>>>>> feature-branch

* Decide which changes to keep, modify, or combine, and then save the file.

1. **Mark Conflicts as Resolved**: After editing the conflicting files, mark them as resolved:

git add <file-name>

1. **Complete the Merge**: Commit the resolved conflicts:

git commit -m "Resolved merge conflicts"

1. **Push Changes to Remote**:

git push origin main

**Raising a Pull Request (PR) with Reviewers**

A Pull Request (PR) is a request to merge changes from one branch into another. It includes the option to involve reviewers for feedback.

**Steps to Raise a Pull Request**

1. **Push Your Feature Branch**: Push your branch to GitHub:

git push origin feature-branch

1. **Open a Pull Request**:

* Go to the repository on GitHub.
* Navigate to the **Pull Requests** tab and click **New Pull Request**.
* Select the base branch (e.g., main) and compare it with your feature branch (e.g., feature-branch).

1. **Fill in PR Details**:

* **Title**: Provide a concise summary of the changes.
* **Description**: Add a detailed explanation of your changes, include screenshots or links if needed.
* **Reviewers**: Add team members or collaborators who should review your PR.
* **Labels**: Optionally, assign labels (e.g., bugfix, feature, etc.).
* **Milestone**: Assign the PR to a milestone if applicable.

1. **Create the PR**: Click **Create Pull Request** to submit it.

**Reviewers' Workflow**

1. **Reviewer Comments**:

* Reviewers will review the PR and might leave comments or request changes.

1. **Address Feedback**:

* Make the necessary changes in your local branch.
* Push the changes to the same branch:
* git push origin feature-branch

1. **Resolve Conflicts (If Any)**: If conflicts arise, resolve them as described earlier and update your branch.
2. **Approval**:

* Once reviewers approve the changes, they’ll approve the PR.
* In some repositories, reviewers may also merge the PR for you.

**Merge the PR**

1. Once approved, click **Merge Pull Request** on GitHub.
2. Choose the merge strategy (default is **merge commit**):

* **Squash and Merge**: Combine all commits into one.
* **Rebase and Merge**: Rebase commits instead of merging.

1. Confirm the merge.

**Best Practices for PR Workflow**

* **Keep PRs Small**: Focus on a single feature or fix to make reviews easier.
* **Use Meaningful Commit Messages**: Ensure your commit history is clean and descriptive.
* **Write Descriptive PR Descriptions**: Provide enough context for reviewers.
* **Test Your Code**: Before raising a PR, ensure your changes are tested and working.
* **Respond Promptly to Feedback**: Communicate with reviewers to address comments efficiently.