= Git

:toc: right

:toclevels: 5

:sectnums:

== How Git Works - PP

== GIT - JT

=== Merging

----

git branch -a

git diff main new\_feature\_branch

git checkout new\_feature\_branch

echo "new file entry"> new.txt

cat new.txt

git add .

git commit -m "commiting new file to branch"

git status

git checkout main

git diff main new\_feature\_branch

git merge new\_feature\_branch

git log --oneline --graph --decorate

output: f907392 (HEAD -> main, new\_feature\_branch) commiting new file to branch

----

Fast-forwarding is only really possible when there are no changes being made on the target branch, which is what we just did. We branched away, we did some work, and then we came back and then merged into master; Lostwe didn't do any additional work on master before doing the merge.

== GIT - PP

=== Getting to Know Git

\* Git tracks changes to source code

\* `git init` - local repository

\*Creating Snapshots\*

\* Git Tracks changes to source code.

image::img/git-launch.png[]

---

----

git init

git add .

git commit -m "comment"

git log

git diff <hash1> <hash2>

git checkout <hash2>

----

Let's say that we have a launch pad to the repository. And what we just did is we put some files on the launch pad. \*This launch pad is also called the index in git\*. In this case, we put all the files in the project in the index. And now step two, we can send the content of the index straight to the repository with this command called commit. \*So get commit means create a snapshot that includes all the files that I put on the launch pad earlier\*. And we can give a message to the commit to describe what's in it, say first commit with all the files. \*And when I send this command, fire the files, get to the repository and they become the first snapshot of the project inside the repository. So we just created a snapshot of the entire project\*. And if I use this command git log, then we see the history of the project so far that right now is \*only this single snapshot, this single commit in the language of git\*. Now this committee isn't very useful until we have at least another committee. So let's imagine that we have to do something that happens all the time in software development, we have to fix a bug. I'll use my text editor to edit a file in this project. Let's say this file and let's change this line here to fix the bug. Actually, I'm just doing a random change. Most likely I just created a bug instead of fixing it. But let's just pretend that I made this change to fix a bug. Now, this file has been changed. It's been modified since the last time we took a snapshot, a commit of the project. And in fact, if I ask git for the current status of the project, it tells me look, this file has been modified, it shows the modified file in red. This is already a mildly useful thing, right? This fact that GIT tells me which files have been modified since the last commit. So let's create another commit. But this time this commit only includes the changes since the previous commit. That is this file. So once again, two steps, first add the file, let's say it's this blue file in the diagram, put it on the launch pad. And if we do git status again, git shows me this modified file in green to mean this file is ready to go in the next commit. And second, we create the commit with a message 3 to 1, lift off. And now we have a second commit another snapshot that includes this change file. In fact, git status shows nothing because everything in the project is aligned with the latest commit. And if I ask for the log, you can see two commits one from a few seconds ago, that includes the latest changes and an earlier one that includes all the original files in the project. And it also shows us who created these commits. That's me. So we are already tracking the history of the project at this point. For example, we can ask a GIT, okay. So what changed between these two snapshots? That's a command named git diff that shows the difference between two commits. This command needs the names of two commits. And one way we identify commits in it is by using their unique code, their version. If you wish these long extra decimal codes git generates these codes automatically when we create a commit, we actually don't need to use the entire code. We can get away with just the first few digits. Like give me the diff between these, i'll copy, paste it here. And this, and this command shows that the only thing that changed between the two commits is that in this file, this line changed. So now we can find out what changed in the project and who changed it and when exactly. And that's already a very useful concept, right? Especially in a large project, but that's only the beginning it gets better. Let's see.

=== Understanding Version Control

\*Branching and Merging\*

image::img/branch-1.png[]

\*Merging\*

\* You move to the premium branch and then you say get merge main.

\*\* Example: Bug Fix

image::img/branch-3.png[width=500]

---

image::img/branch-4.png[width=500]

---

image::img/branch-5.png[width=500]

---

image::img/branch-6.png[width=500]

=== Making Sense of Git

\*Client Server Architecture\*

image::img/client-server-1.png[]

---

\*Distributed Architecture\*

image::img/distributed-1.png[]

---

image::img/distributed-2.png[]

== GIT - F

########################################################################################################################

---

\*Git Configuration Hierarchy\*

image::img/git-hierarchy.png[]

########################################################################################################################

---

image::git-hierarchy2.png[]

########################################################################################################################

---

########################################################################################################################

---

\*Sample `.gitconfig`\*

----

[user]

name = Your Name

email = your.email@example.com

[core]

editor = nano

autocrlf = input

[alias]

co = checkout

ci = commit

st = status

br = branch

df = diff

----

== Git Glossary

################################################################################

---

=== Concurrency

Concurrency in the context of version control systems refers to the ability for multiple users or processes to work on the same project or files simultaneously without conflicts. In other words, it's about managing concurrent changes made by multiple developers to the same codebase.

=== Reflection

Reflection in version control systems is essential for understanding the state and history of a project, tracking changes, troubleshooting issues, and collaborating effectively with other developers.

=== Reversibility

Reversibility in version control systems refers to the capability of reverting changes made to files, code, or the entire project back to a previous state or version. It's a fundamental aspect of version control, allowing developers to undo modifications and restore earlier versions of the codebase.

################################################################################

---

=== Distributed version control systems (DVCS)

Distributed version control systems (DVCS) like Git offer several key features that differentiate them from centralized version control systems. Here are some of the main features:

1. \*\*Distributed repositories\*\*: In DVCS, every user has a complete copy of the repository, including its full history. This means developers can work offline and have access to the entire project's history without needing a constant connection to a central server.

2. \*\*Branching and merging\*\*: DVCSs excel at branching and merging, allowing developers to create branches for new features or experiments, work on them independently, and later merge them back into the main codebase. Branching is lightweight and fast, making it easy to manage parallel lines of development.

3. \*\*Peer-to-peer collaboration\*\*: DVCS enables peer-to-peer collaboration, where developers can share changes directly with each other without relying on a central server. This promotes decentralized workflows and fosters collaboration even in distributed or disconnected environments.

4. \*\*Offline support\*\*: Because each user has a complete copy of the repository, DVCSs offer robust support for offline work. Developers can commit changes, create branches, and perform other version control operations without an internet connection, making them ideal for remote or mobile development.

5. \*\*Flexible workflows\*\*: DVCSs support a wide range of workflows, from centralized to fully distributed, allowing teams to choose the workflow that best fits their needs. Common workflows include feature branching, Gitflow, and forking workflows.

6. \*\*Data integrity and redundancy\*\*: With every user having a full copy of the repository, DVCSs provide redundancy and data integrity. Even if one copy of the repository is lost or corrupted, other copies can be used to recover the data.

7. \*\*Scalability\*\*: DVCSs are highly scalable, capable of handling large repositories and thousands of users. Performance remains consistent even as the size of the project and the number of contributors grow.

8. \*\*Community and ecosystem\*\*: DVCSs like Git have vibrant communities and extensive ecosystems of tools, plugins, and integrations. This ecosystem provides support, documentation, and additional functionality to enhance the version control experience.

These features make distributed version control systems like Git powerful tools for managing and collaborating on software projects, especially in distributed or decentralized development environments.

################################################################################

---

=== Centralized version control systems (CVCS)

Centralized version control systems (CVCS) have some drawbacks compared to distributed version control systems (DVCS). Here are some disadvantages of CVCS:

1. \*\*Single point of failure\*\*: In a CVCS, the central server acts as a single point of failure. If the server goes down or becomes inaccessible, developers may not be able to access the repository, retrieve code, or commit changes, disrupting the development process.

2. \*\*Dependency on network connectivity\*\*: CVCSs rely heavily on network connectivity to access the central repository. Developers need a stable internet connection to push changes to the server or retrieve the latest updates. Limited or unreliable internet access can hinder collaboration and productivity.

3. \*\*Limited offline capabilities\*\*: Unlike DVCSs, CVCSs have limited support for offline work. Developers typically need a connection to the central server to perform version control operations, such as committing changes or creating branches. Working offline or in disconnected environments is challenging or impossible with CVCSs.

4. \*\*Slower performance\*\*: CVCSs may experience slower performance, especially for operations that involve communication with the central server, such as pushing or pulling changes. As the size of the repository or the number of users increases, the performance of the central server may degrade, leading to delays in version control operations.

5. \*\*Concurrency issues\*\*: CVCSs may encounter concurrency issues when multiple developers try to work on the same file or branch simultaneously. Locking mechanisms are often used to prevent conflicts, but they can introduce bottlenecks and hinder collaboration. Developers may experience delays or conflicts when trying to commit changes or merge branches.

6. \*\*Less flexibility in workflows\*\*: CVCSs often impose a centralized workflow, where developers must follow strict procedures for branching, merging, and code review. This can limit flexibility and autonomy, making it difficult to accommodate diverse development workflows or agile practices.

7. \*\*Limited branching and merging capabilities\*\*: CVCSs may have limited support for branching and merging compared to DVCSs. Branches are typically heavyweight and long-lived, making it challenging to manage parallel lines of development or implement feature branching effectively.

8. \*\*Less autonomy for developers\*\*: In a CVCS, developers may have less autonomy and control over their local repositories. They must rely on the central server for version control operations and may face restrictions or permissions issues when trying to create branches, commit changes, or access certain parts of the repository.

Overall, while centralized version control systems have been widely used in the past, they have certain limitations compared to distributed version control systems, especially in terms of reliability, flexibility, and support for distributed and collaborative development workflows.

################################################################################

---

=== Linear and Non-Linear workflow

1. \*\*Linear Workflow\*\*:

- In a linear workflow, changes are applied sequentially, one after the other, without any divergent branches.

- This typically involves a simple workflow where developers work on a single branch, such as the main branch (`master` or `main` in Git).

- Changes are made directly on this branch, and each change builds upon the previous one.

- Once changes are made and tested, they are committed directly to the main branch, creating a linear history of commits.

2. \*\*Non-linear Workflow\*\*:

- In a non-linear workflow, developers may work on multiple branches, allowing for parallel development and experimentation.

- This often involves creating feature branches, where developers work on specific features or fixes independently of each other.

- These feature branches can diverge from the main branch and may have their own sub-branches.

- Once a feature is complete and tested, it can be merged back into the main branch, integrating the changes into the project history.

In Git, both linear and non-linear workflows are common and can be used depending on the needs of the project and the development team. Linear workflows are simpler and easier to understand, making them suitable for smaller projects or teams. Non-linear workflows offer more flexibility and scalability, allowing for concurrent development of multiple features or bug fixes, making them suitable for larger projects with multiple developers.

=== Configure Git

Configuring Git involves setting up various parameters and options to customize your Git environment according to your preferences and requirements. Here's a basic overview of how to configure Git:

\*\*Install Git\*\*: First, you need to install Git on your system if you haven't already. You can download and install Git from the official Git website (https://git-scm.com/), or you can use a package manager for your operating system.

\*\*Set up your identity\*\*: The first configuration step is to set your username and email address, which Git will use to associate your commits with your identity. You can set these values globally for all repositories on your system or locally for a specific repository.

[source,bash]

----

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

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

----

\*\*Configure default text editor\*\*: Git uses a text editor for various tasks, such as writing commit messages or resolving merge conflicts. You can configure your preferred text editor using the following command:

```bash

git config --global core.editor "your\_text\_editor"

```

Replace "your\_text\_editor" with the command-line invocation of your preferred text editor, such as "vim", "nano", "emacs", or "code" for Visual Studio Code.

\*\*Check your configuration\*\*: You can view your Git configuration settings at any time using the following command:

```bash

git config --list

```

This command will display all the configuration settings, both global and local, set in your Git environment.

These are some of the basic configuration steps to set up Git. Depending on your specific needs and workflow, you may need to customize Git further by exploring additional configuration options and settings.

################################################################################

---

\*Sample Commands\*

----

# [home directory]/.gitconfig

git config --global user.name "Naresh C"

git config --global user.name

# [repository]/.git/config

git config --local user.email NC@example.com

git config --list

git config --global --list

git config --local --list

git config --list --show-origin --show-scope

git config --local --unset user.name

git config --local --remove-section user

----

################################################################################

== Basic Git and GitHub Essentials

\* /Users/nareshchaurasia/nc/Git-Tutes

=== What is Git and GitHub

\* Let's talk about a common software development problem and how Git helps us solve it. Let's assume you have a website with two pages: a landing page and an about page. Your boss wakes up one morning and decides that he doesn't want the company details online and he wants you to remove all traces of them. You simply remove the file, send your changes to your server and make your boss happy.

\* However a week later your boss changes their mind, as they often do. He decides he wants an about page and not only that but he wants the exact page back online as it was before. So your only solution is now to start writing that page again and hope that you get it exactly like the previous one. A lot of extra work for nothing.

\* Now let's look at how Git handles that problem. Git maintains a copy of every file in your project at any time. \*Every time you modify any file in any way and commit your changes Git will track it and both versions are never lost\*. Commits, in this context, means simply telling Git that there is a new version of a file or multiple files.

\* So with every change, Git maintains a trace of all files and you can easily switch between different versions of each file. Coming back to our previous example, we have version 1 where we have two pages a landing page and an about page. after your boss tells you they want the About page removed. We have version 2 with only one file. When the boss comes back and decides they actually want the About page we simply revert to Version 1. With a single line of code we have solved the problem.

\* This is of course a very basic example that can be solved in many other ways. However in a real world situation where you have thousands of files with hundreds of versions of each file and teams of developers working on the same project. This versioning system becomes the only way to handle changes in requirements or issues with newer versions of files.

\* Here are a few other problems that Git helps us solve easily.

\*\* \*Easily retrieving changes\* made by other people in our team on the same project. If multiple people are working on the same project we need a way to retrieve their changes quickly and easily.

\*\* \*Solving conflicts\*. It sometimes happens that two or more people change the same file at the same time. When this file is retrieved, there is a conflict because the system cannot know which version of the file is correct. Git gives us a way to analyze and solve these conflicts.

\*\* \*Working on multiple versions of the project at the same time\*. If one developer takes a long time to develop a feature the rest of the team can carry on working without waiting for her. When she has finished Git provides a way for her to reintegrate her changes into the main project.

\*\* \*Never lose any changes you have made to your project ever\*. Anyone who makes changes to a project can subsequently access those changes anywhere. No more accidental file deletes or overwritten data.

\*\* \*Simple light weight and fast system\*. Git is a fast lightweight and as you shall soon see, easy to learn.

\*\* \*GitHub\* is an online service that provides Git repositories. It is free to use for public repositories but paid for private ones.

\*\* There are a few alternatives that provide more or less the same functionality such as \*GitLab, BitBucket or sourceforge\*. I have chosen to use GitHub because of its popularity and its openness to open source projects.

################################################################################

---

=== What is Git Repository

A Git repository is simply a folder that Git manages. It's a folder where we want to track file changes and have file versioning. If we create a Git repository on a local computer we create a folder. If we create it on the GitHub website or any other online service this will simply be a link.

=== Git Commands

\*How to use help in Git\*

----

git clone -h

git log --graph --decorate --oneline --all

export PS1="Master $:"

----

################################################################################

---

==== Cloning

\*Cloning GitHub Repository\*

[source, shell]

----

git clone https://github.com/Naresh-Chaurasia/FullStack-Development-1

git remote -v

----

################################################################################

---

==== Create File

----

echo hello > hello.txt

ls

cat hello.txt

pwd

----

==== Commits

In Git, a commit is a snapshot of your code at a specific point in time. When you commit changes, you're essentially saving those changes to the Git repository. Each commit typically represents a logical unit of work, like adding a new feature, fixing a bug, or making some improvements. Commits include a commit message that describes the changes made, providing context for anyone looking at the commit history.

In the context of Git, a "snapshot" refers to a complete copy of the state of your project at a specific point in time.

----

git init

git status

git add hello.txt

git status

git commit -m "first commit"

git status

git log

----

==== Staging

NOTE: StagedFile.txt and UnStageFile.txt

################################################################################

---

----

echo "hello" > hello.txt

echo "hai" > hai.txt

git status

git add hai.txt

git status

git reset hai.txt

git status

----

################################################################################

---

image::img/git-workflow.png[]

################################################################################

---

Think of the \*staging area in Git\* as a middle ground between your working directory (where you make changes) and your Git repository (where your committed changes are stored).

When you make changes to your files, Git doesn't automatically save them to the repository. Instead, you first need to add these changes to the staging area. This is like preparing your changes to be committed. You can think of the staging area as a place where you gather up all the changes you want to include in your next commit.

Once your changes are in the staging area, you can review them and make sure you're happy with what you're about to commit. If you realize you made a mistake, you can modify your changes in the working directory and then add the corrected version to the staging area again.

Once you're satisfied with the changes in the staging area, you commit them to the repository. This saves the staged changes as a snapshot in the Git history, creating a permanent record of your work.

################################################################################

---

The staging area serves several important purposes:

1. \*\*Selective Committing\*\*: It allows you to selectively choose which changes you want to include in your next commit. This is helpful when you've made modifications to multiple files but want to commit them separately or in groups.

2. \*\*Review Changes\*\*: Before committing your changes to the repository, you can review them in the staging area. This allows you to carefully examine what you're about to commit and make any necessary adjustments or corrections.

3. \*\*Commit Preparation\*\*: The staging area acts as a preparation area before committing changes. It gives you a chance to organize and refine your changes before making them permanent in the repository.

4. \*\*Collaboration\*\*: In collaborative environments, the staging area provides a way for team members to share changes before they are committed. This can help prevent conflicts and ensure that only the intended changes are included in each commit.

Overall, while it's possible to directly add changes to the repository without using the staging area, incorporating it into your workflow offers more control, flexibility, and clarity in managing your changes.

################################################################################

---

==== Branches

TIP: Make a repository in Github and add README.md file.

\* Let's say you're working on Feature A for your project. In the middle of development, a new high priority requirement for Feature B comes in. You're now forced to pause development on feature A and start feature B. Then once Feature B is finished, resume development on feature A.

\* Git accomplishes this with the concept of branches. A branch is simply a series of commits that can be manipulated independently. Git maintains a master branch that is in some sense the source of truth for the project, meaning it is the code that is currently the most up to date. Any commit on a branch can be the origin commit where a new branch can be initiated at that point. Any commit on master will not be seen on the second branch and any commit on the second branch will not affect master.

\* When the development of a feature is complete or at any arbitrary point that we as developers choose, we can merge two branches effectively bringing all the commits on one branch into another branch. In our example, our feature branch will bring all the committed parts into the main branch.

---

image::img/branch-2.png[width=500]

---

----

git branch

git branch featureBranch

git checkout featureBranch

echo "feature" > feature.txt

git add .

git commit -m "feature commit"

git log

----

==== Checking status and inspecting commits

----

git log

git show --name-only hash

#gives info about all commits on all brances

git reflog

----

==== Push and pull

\* So far we have worked with commits and branches only on our local machine. Now it's time to explore the \*collaboration\* features of Git that make it such a flexible system.

\* In order to collaborate with other team members, we want to be able to send the commits that we have made on our local machine to the repository so that they have access to them.

\* Similarly, we want to be able to retrieve commits that our colleagues have published to the central repository. Git push sends all our local changes to the GitHub repository that we have set up at the start.

\*push\*

----

git checkout main

git push

git checkout featureBranch

git push

(git push --set-upstream origin featureBranch)

----

\*pull\*

----

git clone <url>

echo new-file > new-file.txt

git add .

git commit -m "adding new file"

git push

----

\*switch terminals\*

----

git pull

----

==== Merging branches

image::img/merge.png[]

Once our work has been done, we need to be able to merge the branch back into the project so that it is integrated into the rest of the work. In this lecture we'll see how we can do that and what happens when two or more people update the same file from different branches?

In our project we have this situation. We have two branches master and feature branch. Feature branch has one extra commit that we created earlier. When we merge feature branch into master, we will have a new commit that adds the changes we introduced in our feature branch. Let's see this in practice.

----

git init

echo hello > hello.txt

git add .

git commit -m "first master commit"

git log --graph --decorate --oneline --all

git branch -a

###

git branch branch1

git checkout branch1

echo branch > branch.txt

git add .

git commit -m "branch commit"

git status

git log --graph --decorate --oneline --all

###

git diff master branch1

git checkout master

git merge branch1

##Fast-forward##:: Fast-forwarding is only really possible when there are no changes being made on the target branch

----

==== Happy Path - Disable Fast Forward.

----

- Create Branch

- Commit1

- Commit2

- Switch to master branch

- git merge branch1 --no-ff

----

==== Automatic Merge

==== Git Rebase

Certainly! Git rebase is a powerful tool used to integrate changes from one branch into another. It works by moving or reapplying commits from one branch (the source branch) onto another branch (the target branch). This process effectively rewrites the commit history of the target branch.

\* make change in branch1, commit

\* make change in master, commit

\* rebase master to branch1

\* git log

\* make change to branch1

Here's a step-by-step explanation of how git rebase works:

1. \*\*Start with a Clean Working Directory:\*\*

Before starting a rebase operation, ensure that your working directory is clean, meaning there are no uncommitted changes.

2. \*\*Checkout the Target Branch:\*\*

First, switch to the branch where you want to apply the changes. This is typically the branch you want to update with changes from another branch.

```bash

git checkout target-branch

```

3. \*\*Start the Rebase:\*\*

Run the rebase command, specifying the source branch from which you want to reapply commits. This is typically done with the `-r` or `--rebase` flag.

```bash

git rebase source-branch

```

Git will identify the common ancestor of both branches and then replay each commit that is unique to the source branch on top of the target branch.

==== Pull Request and Code Reviews

\* Any team larger than one developer should use pull requests and if they don't, there is an issue with the project. Pull requests are a way for one contributor to have a look at another contributors code, give opinions and reviews and approve or reject the change. They also help solve merge conflicts. Pull requests are basically a request to merge a branch into another branch.

\* Let's create a new commit on feature branch and push it to the origin repository. Right. So at the moment we have a difference between the two branches.

Let's log in to the GitHub website, click on the project link. Let's switch to our feature branch by clicking on the button that says Branch. And from here, create a new pull request by clicking on the button that says New pull request. And click create pull request. On this screen, we can see a few different things. We can view the code changes that were introduced by this PR. Reviewers or anyone can add comments or suggestions to the code right at the line where the comment applies. We can add reviewers to the PR in the top right corner of conversation tab. Reviewers can also approve or reject the PR. Also, if a PR introduces any merge conflicts, we will not be able to merge it until the conflicts are resolved. You can always add new changes to the pull request by uploading new commits to the branch. Once the PR is approved and everyone is happy with the changes, the PR can be merged by clicking on the merge PR button on the main screen.

################################################################################

---

=== Local Repository, New Repository

\*Creating Repository Locally, and Pushing it Remotely on GitHub\*

\*\*Step 1: Creating a New Repository:\*\*

[source,bash]

----

echo "# Temp-Del" >> README.md

git init

git add README.md

git commit -m "first commit"

----

\*\*Step 2: Setting Up the Main Branch:\*\*

[source,bash]

----

git branch -M main

----

\*\*Step 3: Linking to GitHub:\*\*

[source,bash]

----

git remote add origin git@github.com:Naresh-Chaurasia/Temp-Del.git

----

\*\*Step 4: Pushing to GitHub:\*\*

[source,bash]

----

git push -u origin main

----

################################################################################

---

=== Connect Git Bash to GitHub using SSH

Connecting Git Bash to GitHub using SSH involves several steps. SSH (Secure Shell) is a cryptographic network protocol that allows secure communication between two systems. Here's a step-by-step guide:

\*Step 1: Install Git Bash\*

If you haven't already, download and install Git Bash from the official website: [Git](https://git-scm.com/).

\*Step 2: Generate SSH Key\*

\* Open Git Bash.

\* Use the following command to generate a new SSH key:

```bash

ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"

```

Replace "your\_email@example.com" with the email address associated with your GitHub account.

\* Press Enter to accept the default file location.

\* You may be prompted to enter a passphrase. You can either set one or press Enter to skip.

Here's a sample command and output:

----

Generating public/private rsa key pair.

Enter file in which to save the key (/home/your\_username/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/your\_username/.ssh/id\_rsa.

Your public key has been saved in /home/your\_username/.ssh/id\_rsa.pub.

----

\*Step 3: Add SSH Key to SSH Agent\*

\* Start the SSH agent by running:

```bash

#Linux or Mac

eval "$(ssh-agent -s)"

#Windows

eval `ssh-agent -s`

```

\* Add your SSH private key to the SSH agent:

```bash

ssh-add ~/.ssh/id\_rsa

```

\*Step 4: Add SSH Key to GitHub\*

\* Copy the SSH key to your clipboard:

```bash

clip < ~/.ssh/id\_rsa.pub # For Windows

```

or

```bash

cat ~/.ssh/id\_rsa.pub # For Linux/Mac, copy manually

```

\* Go to your GitHub account settings.

\* Navigate to "SSH and GPG keys."

\* Click on "New SSH key" or "Add SSH key."

\* Paste your SSH key into the provided field and give it a meaningful title.

\*Step 5: Test the Connection\*

\* In Git Bash, run the following command to test the SSH connection:

```bash

ssh -T git@github.com

```

You might see a warning the first time; type "yes" to continue.

\* If successful, you should see a message confirming the authentication.

Now, your Git Bash is connected to GitHub using SSH. When you interact with your repositories, Git will use the SSH key for authentication. This setup is more secure than using passwords and allows for smoother, password-less access to your GitHub repositories.

---

\*What is SSH Agent\*

The SSH Agent is a program that runs in the background on your local machine and manages SSH keys. Its primary purpose is to securely store and handle private keys, eliminating the need for users to enter passphrases every time they connect to a remote server or perform operations that require authentication.

Here's a breakdown of what `eval "$(ssh-agent -s)"` command does:

\* `ssh-agent -s`: This command starts the SSH agent and outputs the necessary environment variables to initialize the agent.

\* `eval "$(ssh-agent -s)"`: The `eval` command is used to execute the output of the `ssh-agent -s` command, setting up the environment variables in the current shell session.

\* By running this command, you ensure that the SSH agent is running and properly configured in your shell session.

\* The agent holds your private SSH key and provides it to Git when needed, eliminating the need to enter your SSH key passphrase repeatedly.

\* It enhances security and convenience by managing the SSH key for you.

\* This is especially useful when working with multiple repositories and avoiding the need to re-enter your passphrase for each interaction with your remote repositories.

---

The main difference between using SSH and HTTPS with Git lies in how authentication and data transfer are handled:

\*\*SSH (Secure Shell):\*\*

1. \*\*Authentication:\*\* SSH uses public-key cryptography to authenticate with the remote repository. When you connect to a remote repository via SSH, you generate a key pair: a public key and a private key. The public key is stored on the remote server, while the private key remains on your local machine. This allows you to authenticate without needing to enter a username and password each time you interact with the remote repository.

2. \*\*Protocol:\*\* SSH uses the SSH protocol for communication between your local machine and the remote server. This protocol provides encryption and ensures secure communication over the network.

3. \*\*URL format:\*\* When using SSH, the URL format for the remote repository typically starts with `git@`, followed by the hostname of the server, and then the path to the repository. For example:

```

git@github.com:user/repo.git

```

\*\*HTTPS (Hypertext Transfer Protocol Secure):\*\*

1. \*\*Authentication:\*\* HTTPS uses basic authentication or token-based authentication to authenticate with the remote repository. When using basic authentication, you provide your username and password each time you interact with the remote repository. With token-based authentication (such as personal access tokens), you generate a token that serves as a credential for accessing the repository.

2. \*\*Protocol:\*\* HTTPS uses the HTTP protocol over an encrypted connection (HTTPS) for communication between your local machine and the remote server. Like SSH, HTTPS also provides encryption and ensures secure communication over the network.

3. \*\*URL format:\*\* When using HTTPS, the URL format for the remote repository typically starts with `https://`, followed by the hostname of the server, and then the path to the repository. For example:

```

https://github.com/user/repo.git

```

\*\*Which one to use?\*\*

- \*\*SSH:\*\* Using SSH is often preferred for its convenience and security. Once you've set up SSH keys, you don't need to enter your credentials each time you interact with the remote repository, which can streamline your workflow. Additionally, SSH can be more secure as it uses public-key cryptography for authentication.

- \*\*HTTPS:\*\* HTTPS may be preferred in certain environments where SSH connections are blocked or restricted. Additionally, HTTPS can be more straightforward to set up initially, especially for users who are less familiar with SSH key management.

In summary, both SSH and HTTPS are secure ways to interact with remote Git repositories, and the choice between them depends on factors such as convenience, security, and the requirements of your specific environment.