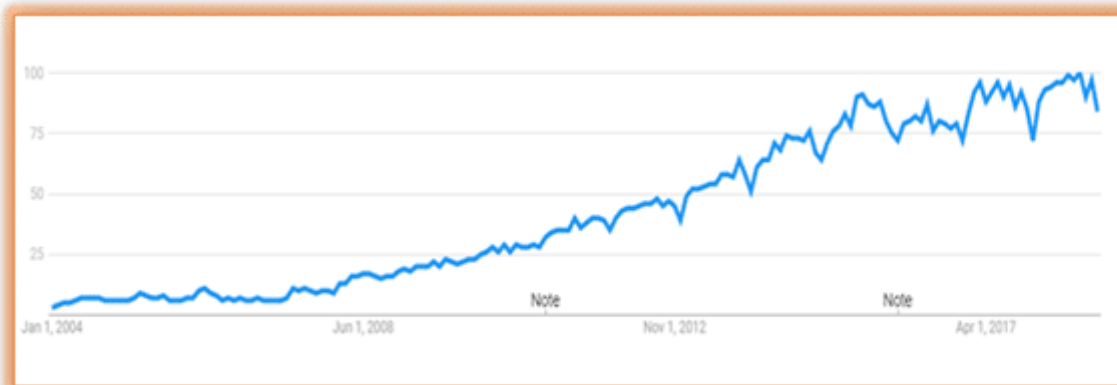


What is Git?

Git helps developers keep track of the history of their code files by storing them in different versions on its own server repository, i.e., [GitHub](#). Git has all the functionality, performance, security, and flexibility that most of the development teams and individual developers need.



The below-given graph depicts the rate of popularity that Git has gained over the years (2004–2017):



Now, let us go ahead in this Git tutorial and learn why Git is used and what the factors are that make Git so unique.

Become a master of DevOps by joining this online [DevOps Training in London!](#)

Why Git Version Control?

Below are some of the facts that make Git so popular:

- **Works offline:** Git provides users very convenient options such as allowing them to work both online and offline. With other version control systems like SVN or CVS, users need to have access to the Internet to connect to the central repository.
- **Undoes mistakes:** Git allows us to undo our commands in almost every situation. We get to correct the last commit for a minor change, and also we can revert a whole commit for unnecessary changes.
- **Restores the deleted commits:** This feature is very helpful while dealing with large projects when we try out some experimental changes.
- **Provides security:** Git provides protection against secret alteration of any file and helps maintain an authentic content history of the source file.
- **Guarantees performance:** Being a distributed version control system, it has an optimized performance due to its features like committing new changes, branching, merging, comparing

past versions of the source file, etc.

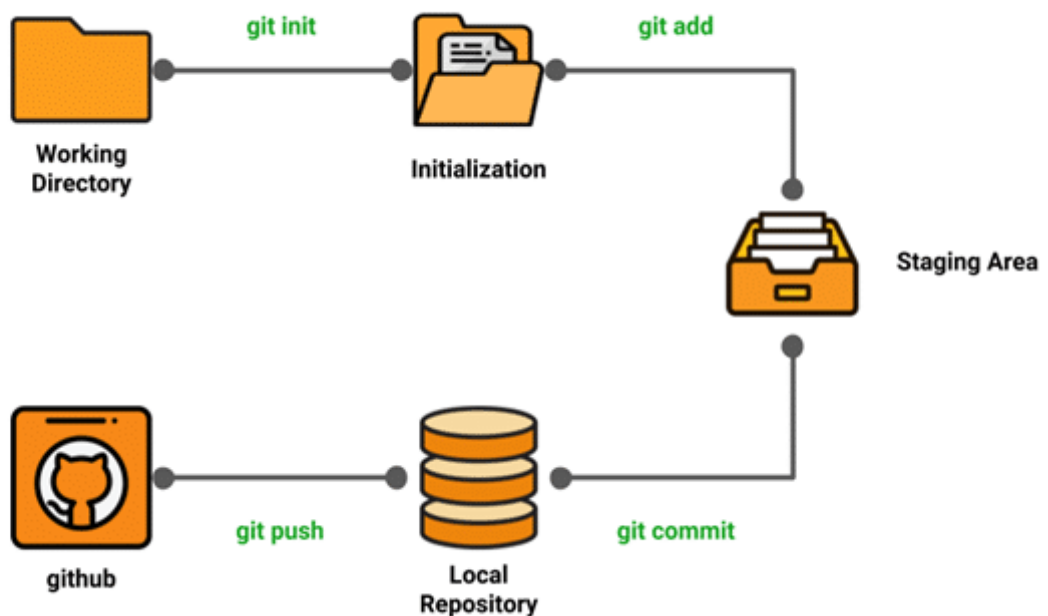
- **Offers flexibility:** Git supports different nonlinear development workflows, for both small and large projects.

How does Git work?

Now that we have learned about Git in this Git tutorial for beginners, we will move ahead and discuss how Git works. Let us start learning the Git basics, starting with the life cycle of Git.

Git Life Cycle

- **Local working directory:** The first stage of a Git project life cycle is the local working directory where our project resides, which may or may not be tracked.



- **Initialization:** To initialize a repository, we give the command `git init`. With this command, we will make Git aware of the project file in our repository.
- **Staging area:** Now that our source code files, data files, and configuration files are being tracked by Git, we will add the files that we want to commit to the staging area by the `git add` command. This process can also be called indexing. The index consists of files added to the staging area.
- **Commit:** Now, we will commit our files using the `git commit -m 'our message'` command.

We have successfully committed our files to the local repository. But how does it help in our projects? The answer is, when we need to collaborative projects, files may have to be shared with our team members.

This is when the next stage of the Git life cycle occurs, i.e., in GitHub, we publish our files from the local repository to the remote repository. And how do we do that? We do that by using the `git push` command.



So far, we have discussed the life cycle of a Git project, where we came across some commands such as `git init`, `git add`, `git commit`, `git push`, etc. Further in this Git tutorial, we will be explaining these common commands in detail.

But before that, let's go through the installation and configuration of Git.

Installing Git

For installing Git on Windows, click [here](#).

For the Git installation on macOS, click [here](#).

For the Git installation on Linux, click [here](#).

We have now successfully installed Git on our system. Let us set up the environment on our system in the next section of this Git for beginners tutorial.

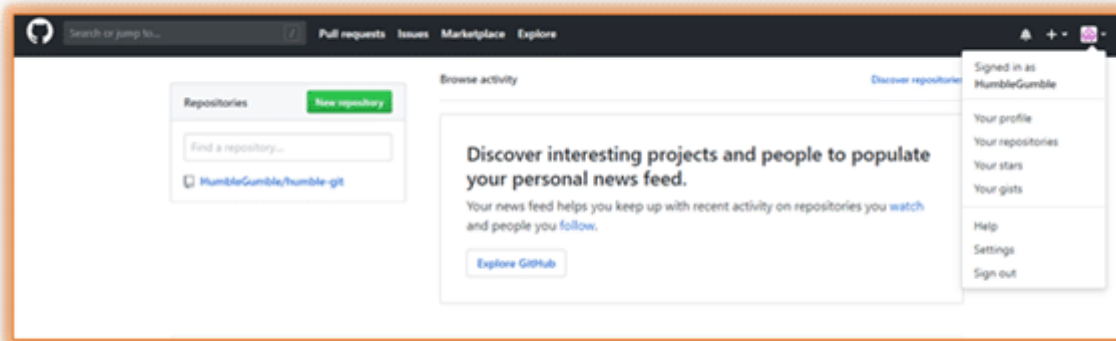
Environment Setup

- Create a GitHub account
- Configure Git
- Create a local repository

Creating a GitHub account:

- Go to <https://Github.com>
- Create a GitHub account
- Login

After accomplishing these steps, our GitHub page should look like this:



Configuring Git:

To tell Git who we are, run the following two commands:

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git config --global user.email "debashis.intellipaata@gmail.com"

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git config --global user.name "HumbleGumble"
```

Creating a local repository:

To begin with, open a terminal and move to where we want to place the project on our local machine using the `cd` (change directory) command. For example, if we have a 'projects' folder on our desktop, we would do something like below:

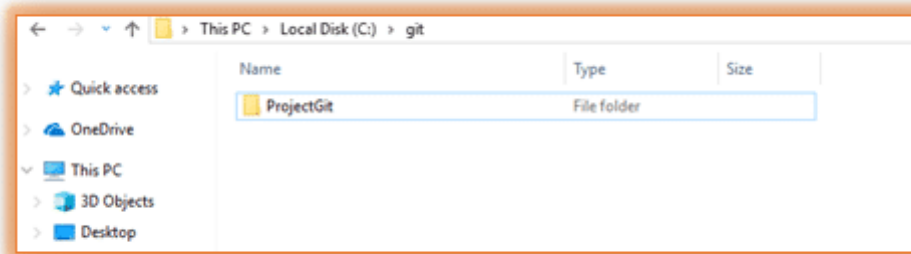
```
MINGW64:/c/git

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 ~
$ cd /c

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c
$ cd git

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git
$ mkdir ProjectGit
```

We can go to the Git folder and see the new directory that we have just created.



Finally, we are all set to start working on [Git commands](#).

Common Git Commands

We will divide the common Git commands into two primary categories:

- **Local:** git init, git touch, git status, git add, git commit, and git rm
- **Remote:** git remote, git pull, and git push

Next in Git and GitHub tutorial, we will discuss the local Git commands.

Local Git Commands

- **git init:** We use the git init command to initialize a Git repository in the root of a folder.

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git
$ cd ProjectGit/

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit
$ git init
Initialized empty Git repository in C:/git/ProjectGit/.git/
```

- **git touch:** To add files to a project, we can use git touch. Let us see how we can add a file to the Git repository we have just created

Step 1: Create a new file with the command touch

Step 2: See the files present in our master branch

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ touch humble.txt

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ ls
humble.txt
```

- **git status:** After creating a new file, we can use the git status command and see the status of the files in the master branch.

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git status
On branch master

No commits yet

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

        humble.txt

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

- **git status:** Now, we will add the humble.txt file to the staging environment by using git add, before going ahead to the staging to see the change using git status.

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   humble.txt
```

- **git commit:** Now we will use the git commit command as shown below:

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git commit -m "What is HumbleGumble?"
[master (root-commit) 169cfba] What is HumbleGumble?
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 humble.txt
```

Thus, we have successfully created our first commit.

- **Git clone:**

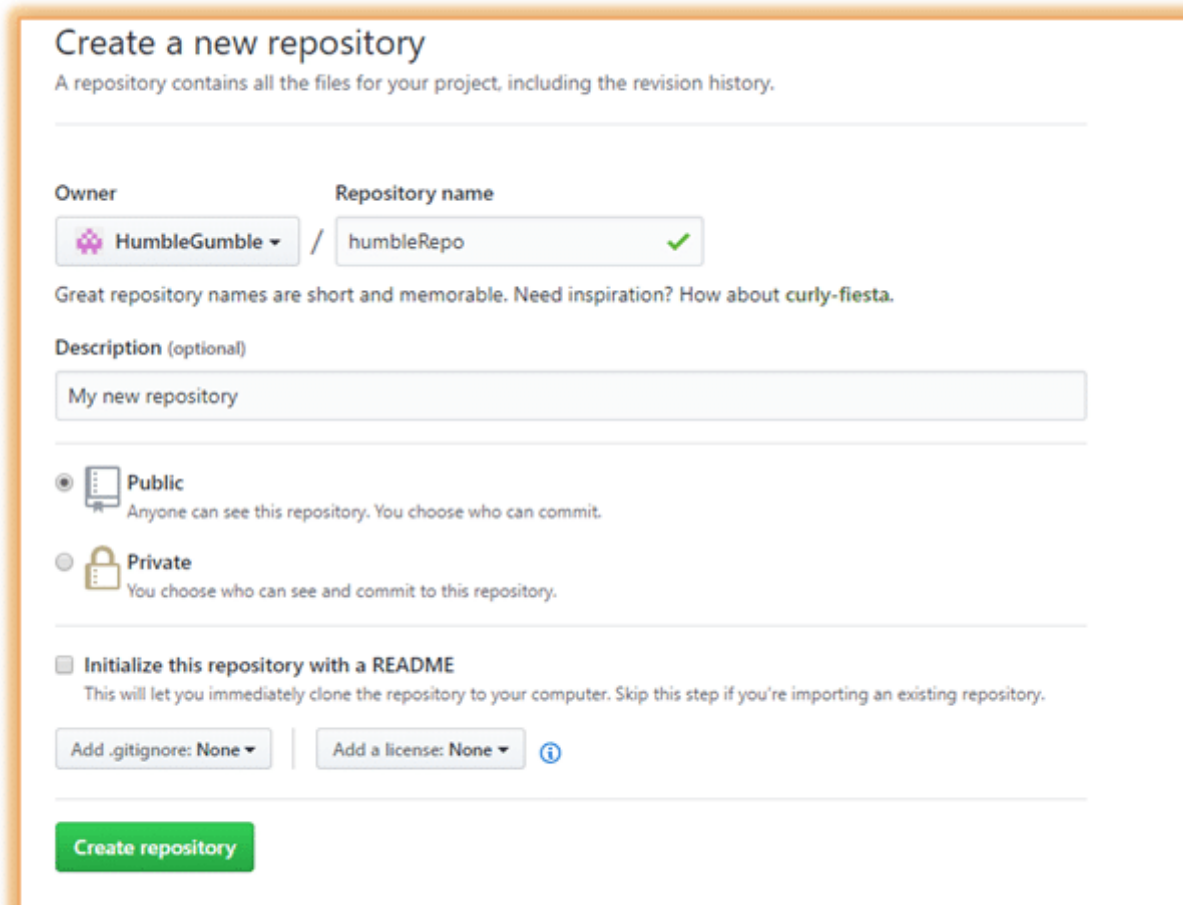
```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git clone C:\git\ProjectGit
Cloning into 'gitProjectGit'...
warning: You appear to have cloned an empty repository.
done.
```

Remote Commands

Let us now discuss the remote commands such as remote, push, and pull in Git. For that, we will create a new repository in our GitHub account. How do we do that? Follow the below steps:

Step 1: Go to the GitHub account

Step 2: Create a new repository



The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and 'A repository contains all the files for your project, including the revision history.' Below this, there are two main sections: 'Owner' and 'Repository name'. The 'Owner' is set to 'HumbleGumble' with a dropdown arrow. The 'Repository name' is 'humbleRepo' with a green checkmark. Below these, there is a note: 'Great repository names are short and memorable. Need inspiration? How about curly-fiesta.' The 'Description (optional)' field contains 'My new repository'. There are two radio button options: 'Public' (selected) and 'Private'. Below these, there is a checkbox for 'Initialize this repository with a README'. At the bottom, there are two dropdown menus: 'Add .gitignore: None' and 'Add a license: None', followed by a green 'Create repository' button.

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: HumbleGumble / Repository name: humbleRepo ✓

Great repository names are short and memorable. Need inspiration? How about curly-fiesta.

Description (optional): My new repository

☒ Public: Anyone can see this repository. You choose who can commit.

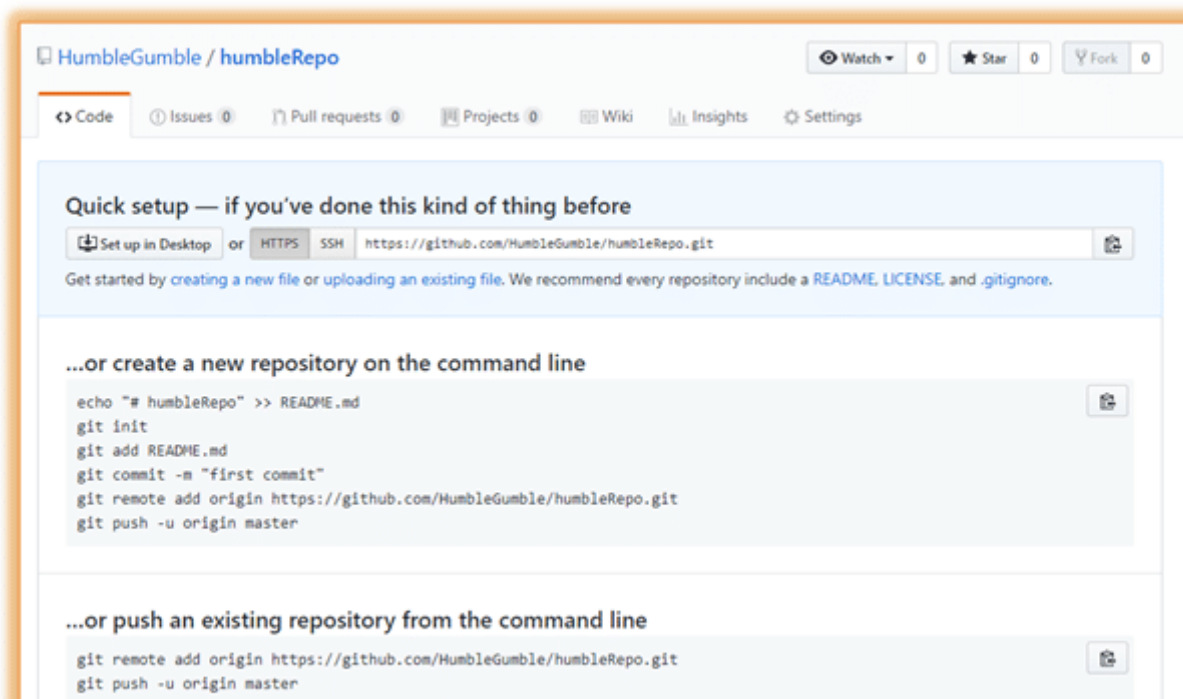
☐ Private: You choose who can see and commit to this repository.

☐ Initialize this repository with a README: This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None | Add a license: None ⓘ

Create repository

Once we are done with filling up the new repository form, we should land on a page like the following:



The screenshot shows the GitHub repository page for 'HumbleGumble / humbleRepo'. The page has a header with the repository name and a navigation bar with links to 'Code', 'Issues', 'Pull requests', 'Projects', 'Wiki', 'Insights', and 'Settings'. Below the navigation bar, there is a 'Quick setup' section with a 'Set up in Desktop' button, an 'HTTPS' button, and an 'SSH' button. The 'SSH' button is selected, and the URL 'https://github.com/HumbleGumble/humbleRepo.git' is shown. Below this, there is a section for '...or create a new repository on the command line' with a code block containing the following commands: 'echo "# humbleRepo" >> README.md', 'git init', 'git add README.md', 'git commit -m "first commit"', 'git remote add origin https://github.com/HumbleGumble/humbleRepo.git', and 'git push -u origin master'. There is also a section for '...or push an existing repository from the command line' with the same 'git remote add' and 'git push' commands.

HumbleGumble / humbleRepo

Watch 0 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights Settings

Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/HumbleGumble/humbleRepo.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# humbleRepo" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/HumbleGumble/humbleRepo.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/HumbleGumble/humbleRepo.git
git push -u origin master
```

Now, we are ready to operate remote commands in our repository that we have just created.

To push an existing repository to a remote repository from a command line, follow the commands given below.

- **git remote:**

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (branch1)
$ git remote add origin https://github.com/HumbleGumble/humbleRepo.git
```

After this, we will provide the following command:

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (branch1)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 224 bytes | 224.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'master' on GitHub by visiting:
remote:   https://github.com/HumbleGumble/humbleRepo/pull/new/master
remote:
To https://github.com/HumbleGumble/humbleRepo.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

- **git push**

```
git push origin EnterBranchName
```

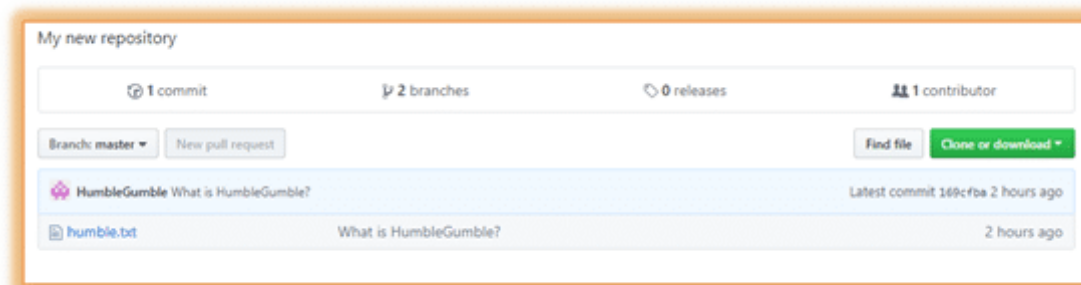
Don't get confused by the word 'origin'. What exactly is this command referring to?

Instead of writing the whole Git URL, like **git push Git@Github.com:Git/Git.Git ourbranchname**, we can just use the following command, assuming that our brach name is 'branch1':

```
git push origin branch1
```

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (branch1)
$ git push origin branch1
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'branch1' on GitHub by visiting:
remote:   https://github.com/HumbleGumble/humbleRepo/pull/new/branch1
remote:
To https://github.com/HumbleGumble/humbleRepo.git
 * [new branch]      branch1 -> branch1
```

Now, let us look at our repository:



- **git pull**

When it comes to syncing a remote repository, the pull command comes in handy. Let us take a look at the commands that can lead to the pulling operation in Git.

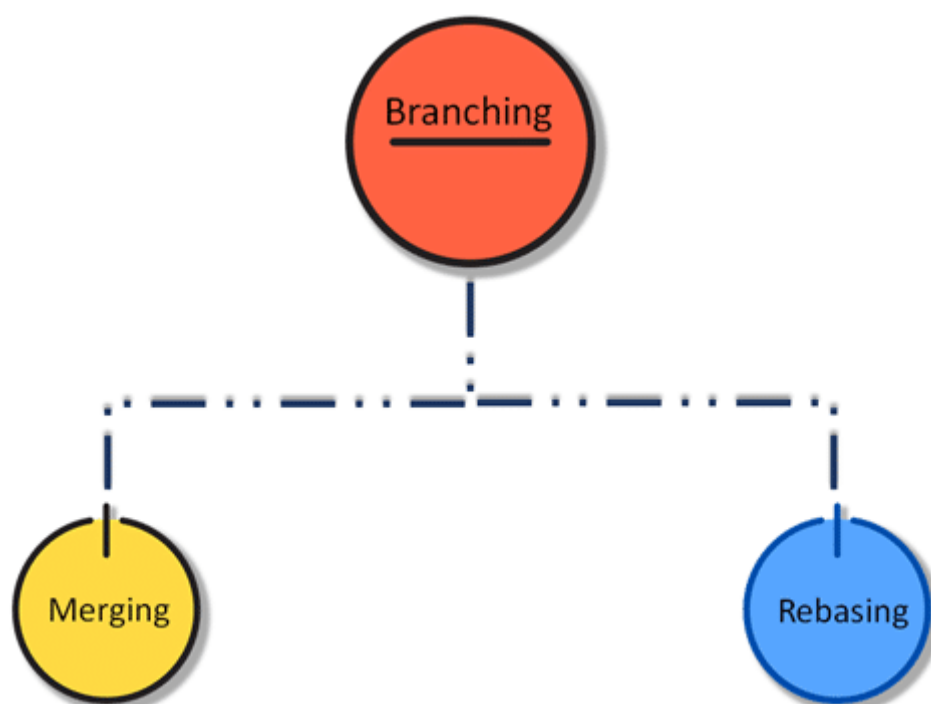
- remote origin
- pull master

We will use these as shown below:

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (master)
$ git remote set-url origin "https://github.com/HumbleGumble/humbleRepo.git"

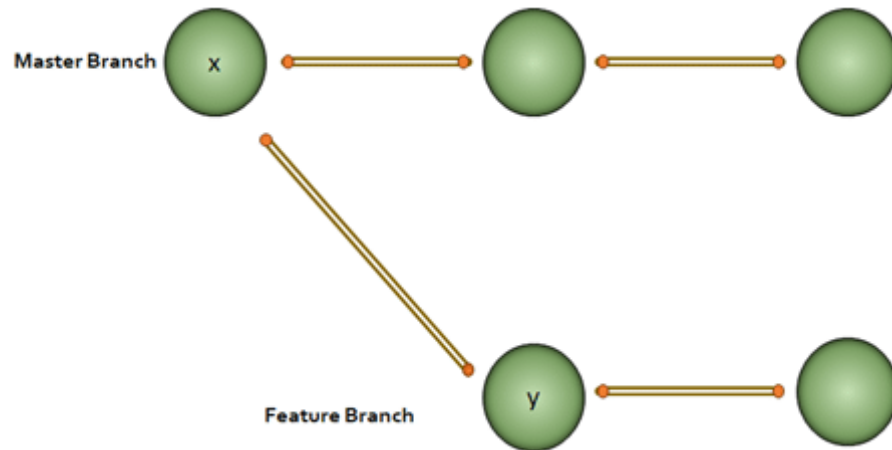
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (master)
$ git pull origin master
From https://github.com/HumbleGumble/humbleRepo
* branch          master      -> FETCH_HEAD
+ 5ad410f...169cfba master    -> origin/master (forced update)
Already up to date.
```

Now that we have learned the workflow of Git with the help of the common Git commands, let us take a look at branching, merging, and rebasing and also at how and when to use them.



Branching in Git

We are almost familiar with various Git commands now. Let's take a step further and discuss one of the most important operations in Git, namely, branching. In branching, we mainly make use of the **git checkout** and **git branch** commands.



Every node in the figure above represents commits. We have to note that for the 'y' node in the feature branch, 'x' is the base.

Why would we do branching in the first place? Say, we want to modify something but don't want to make any change in the main project. This is when we make a branch out of the master branch, i.e., if we want to create a new branch to add a feature to the main project, we will make a branch out of it with the help of the following steps:

Step 1: We will run **git checkout -b <new branch name>**

Step 2: Then, we will use the git branch command to confirm that our branch is created

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git checkout -b branch1
Switched to a new branch 'branch1'

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (branch1)
$ git branch
* branch1
  master
```

We can see from the above image that we have created a branch called 'branch1' and we automatically got landed in the new branch. Again, just to see which branch we are currently in, we run another command, git branch.

Note: The * mark before the branch name shows that it is the current branch.

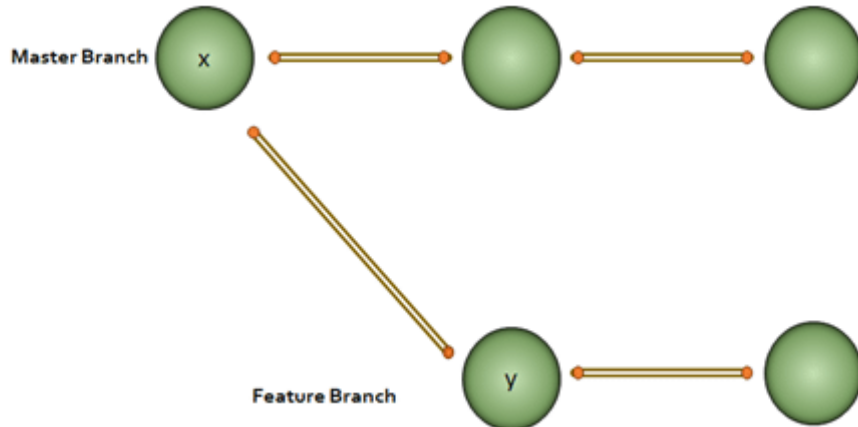
Now, how to change a branch to the master branch? For that, we use the following command:

```
git checkout master
```

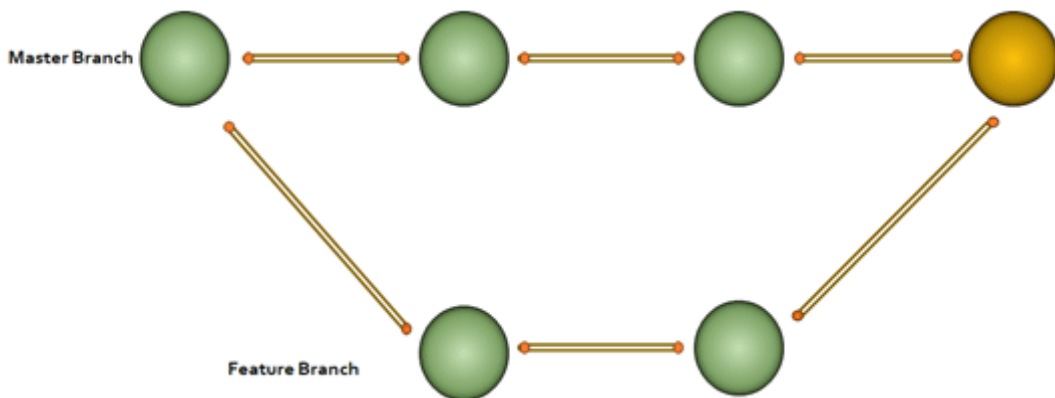
Merging in Git (git checkout, git add, git log, git merge, merging conflicts, and rebasing)

Now that we have learned how to create a branch and work on it, let us take a look at the merge feature in Git by merging the branch we created to the master branch.

Let's take the above example. Say, we have a master branch and a feature branch.



The merge commit represents every change that has occurred on the feature branch since it got branched out from the master.



Note: Even after merging, we can go on with our work on both the master and the feature branches independently.

Let us see how to perform merging:

Step 1: Create a new branch called 'branch2'

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (branch1)
$ git checkout -b branch2
Switched to a new branch 'branch2'
```

Step 2: Create a new file in the branch

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (branch2)
$ touch newFile.txt
```

Step 3: Add changes from all tracked and untracked files

Note: Refer to the following **git add** attributes

```
-n, --dry-run      dry run
-v, --verbose      be verbose

-i, --interactive  interactive picking
-p, --patch        select hunks interactively
-e, --edit         edit current diff and apply
-f, --force        allow adding otherwise ignored files
-u, --update       update tracked files
--renormalize      renormalize EOL of tracked files (implies -u)
-N, --intent-to-add record only the fact that the path will be added later
-A, --all          add changes from all tracked and untracked files
--ignore-removal   ignore paths removed in the working tree (same as --no
11)
--refresh          don't add, only refresh the index
--ignore-errors    just skip files which cannot be added because of error

--ignore-missing   check if - even missing - files are ignored in dry run
--chmod (+|-)x    override the executable bit of the listed files
```

In our case, we have given the command as **git add -A** and after that, we will commit one sentence as shown below:

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch2)
$ git add -A

INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch2)
$ git commit -m "Added a file in branch2"
[branch2 59ce99d] Added a file in branch2
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 newFile.txt
```

Step 4: Check the last three logs by running the command: **git log -3**

```
INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch2)
$ git commit -m "Added a file in branch2"
[branch2 59ce99d] Added a file in branch2
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 newFile.txt

INTELLIPAAT@DESKTOP-1B6LHO3 MINGW64 /c/git/ProjectGit (branch2)
$ git log -3
commit 59ce99d0690aacba8c4ce9aaf62bcac158821f9d (HEAD -> branch2)
Author: HumbleGumble <debashis.intellipaat@gmail.com>
Date: Thu Nov 29 16:17:48 2018 +0530

    Added a file in branch2

commit 169cfba94dbbc394c895353fe6b1bfaa38890e9f (origin/master, origin/branch1,
master, branch1)
Author: HumbleGumble <debashis.intellipaat@gmail.com>
Date: Thu Nov 29 13:15:35 2018 +0530

    What is HumbleGumble?
```

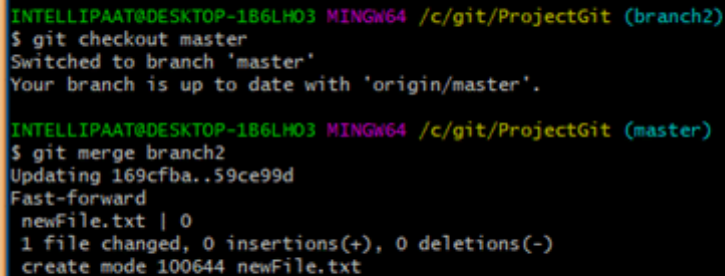
We have created another branch on our master branch. Now, we will see how to perform **merging**.

Let us get inside the master branch using the following command:

```
git checkout master
```

After that, we will perform merging with the help of the below command:

```
git merge branch2
```



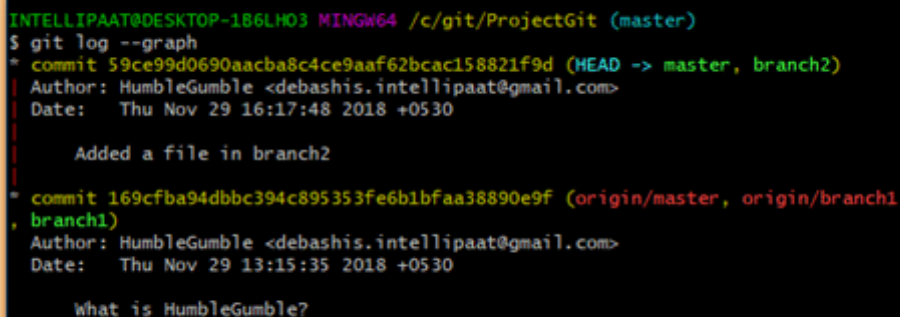
```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (branch2)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git merge branch2
Updating 169cfba..59ce99d
Fast-forward
 newFile.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newFile.txt
```

Get in touch with Intellipaate for a comprehensive [DevOps Training](#) and be a certified DevOps Engineer!

Now, we have successfully merged the feature branch into the master branch. Let us take a look at how it looks inside the master branch using the following command:

```
git log --graph
```



```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git log --graph
* commit 59ce99d0690aacba8c4ce9aaf62bcac158821f9d (HEAD -> master, branch2)
  Author: HumbleGumble <debashis.intellipaate@gmail.com>
  Date: Thu Nov 29 16:17:48 2018 +0530

    Added a file in branch2

* commit 169cfba94dbbc394c895353fe6b1bfaa38890e9f (origin/master, origin/branch1, branch1)
  Author: HumbleGumble <debashis.intellipaate@gmail.com>
  Date: Thu Nov 29 13:15:35 2018 +0530

    what is HumbleGumble?
```

We can see the graph in the above image on the left. But **why the graph is linear?**

In our ProjectGit master branch, we did branching and committing. But after the branching, the master branch had not encountered any more commits. Hence, after merging, we have a linear graph.

Let us perform **git log --graph** with more than one commits in both master and feature branches that we have created.

```

commit 84fffb5c564c4363f8c7f6293a3b87956437276d2 (HEAD -> master)
Merge: fffab95 4cd5b62
Author: HumbleGumble <debashis.intellipaat@gmail.com>
Date: Thu Nov 29 20:30:13 2018 +0530

    Merge branch 'branch1'

commit 4cd5b62ef63f6145da25515a9b27dd1bccfe8115 (branch2)
Author: HumbleGumble <debashis.intellipaat@gmail.com>
Date: Thu Nov 29 20:29:35 2018 +0530

    added on branch2 2

commit fffab95d90fcff9780b2463ab5a29e7d4661ab34
Author: HumbleGumble <debashis.intellipaat@gmail.com>
Date: Thu Nov 29 20:30:07 2018 +0530

    added on master 2

commit 079fbd533578f7dcacbb89599ee35c16c6135ad3
Author: HumbleGumble <debashis.intellipaat@gmail.com>
Date: Thu Nov 29 20:28:57 2018 +0530

    added on master

```

Here, we can see the graph with commits on both master and feature branches. The red part of the graph indicates the merging operation.

Advantages of merging:

- Merging allows parallel working in collaborative projects.
- It saves the time that is consumed by manual merging.

The disadvantage of merging:

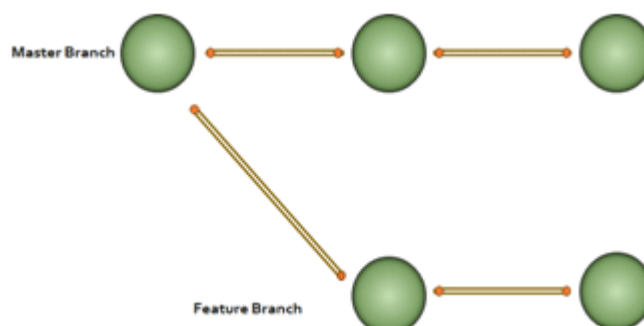
- Merging conflicts may occur while merging branches.

Now that we have successfully learned to branch and merge with Git and GitHub, further in this best Git tutorial, let us look at yet another important Git operation, i.e., rebasing.

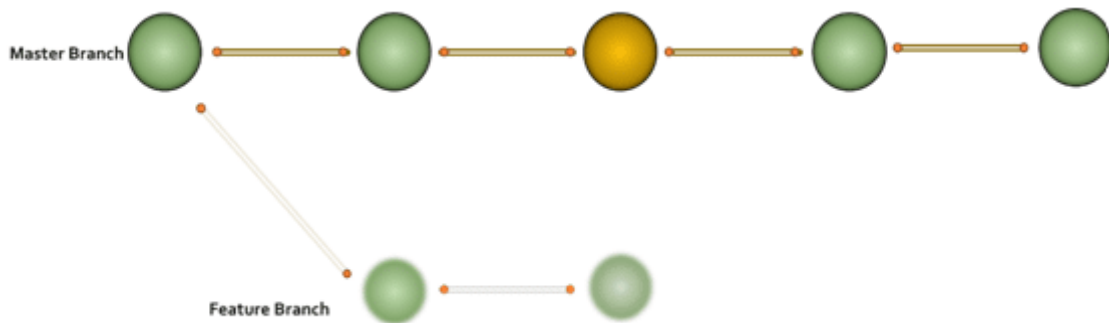
Git Rebase

When our project becomes relatively large, the commit log and the history of the repository become messy. Here, we use rebasing. Rebasing will take a set of commits, copy them, and store them outside our repository. This helps us maintain a linear sequence of commits in our repository.

Let us take the same example. Here, we will rebase the master branch and see what happens.



Note: In rebasing, the base of the feature branch gets changed and the last commit of the master branch becomes the new base of the feature branch.



Now, let us perform rebasing in Git Bash.

```
INTELLIPAAT@DESKTOP-1B6LH03 MINGW64 /c/git/ProjectGit (master)
$ git rebase master
Current branch master is up to date.
```

The advantage of rebasing:

- Rebasing provides a cleaner project history.

The disadvantage of rebasing:

- In a collaborative workflow, re-writing the project history can be potentially catastrophic.

Now that we understood what branching, merging, and rebasing are, next in this Git tutorial, we will see where to use merging and rebasing.

The golden rule of Git merging and rebasing:

- **When to use git merge:** We can use git merge while working on the public branches.
- **When to use git rebase:** We will use git rebase while working on the local branches.

As we have already mentioned earlier in this Git commands tutorial, one major disadvantage of merging is merge conflicts, which can be backbreaking for a team project. Let us understand how merge conflicts occur and how to resolve them.

Git Merge Conflict and Rebase Conflict

This drawback of the merging operation in Git can be explained with a simple example shown below:

<pre>main() { program statements calling function 1 } Function1() { statements } Function2() { statements }</pre> <p>Developer A</p>	<pre>main() { program statements calling function 1 } Function1() { statements } Function3() { statements }</pre> <p>Developer B</p>
---	---

Say, we have two branches of the master branch, branch1 and branch2, and two developers are working on these two branches independently but on the same code file.

As we have seen in the above image, the developers have made the below-mentioned modifications:

- Developer A added a function called 'function2' to the main code.
- Developer B added a different function called 'function3' to the same code file.

How to merge these two modifications? That is where the merging conflict occurs.

Similarly, git rebase also exhibits conflicts

Solving the Conflicts

We can manually resolve the merge conflict using the merging tool. We can use **file locking** which doesn't allow different developers to work on the same piece of code simultaneously. It helps avoid merge conflicts but slows down the development process.

The rebasing conflict can also be solved with the help of the Git merging tool.

Now, let us look at a few Git workflows briefly before we end this Git tutorial so that we get an idea of which Git workflow to choose for our team project.

Git Workflows

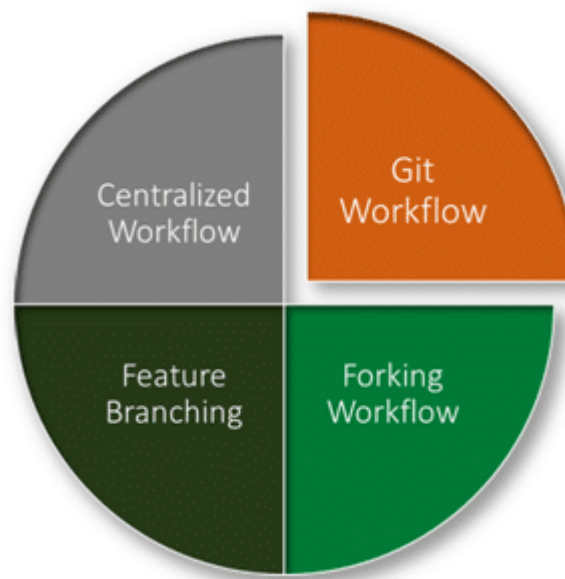
In this section, we will get introduced to different workflow options available in Git. Once we get an idea of these workflows, we can choose the right one for our team project.



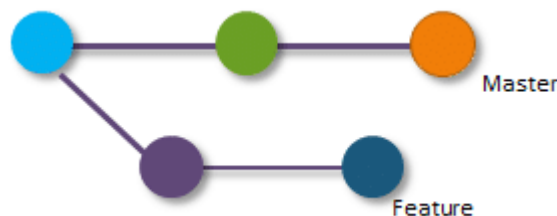
But, why is it important to choose the right Git workflow?

Depending upon the team size, choosing the right Git workflow is important for a team project to increase its productivity.

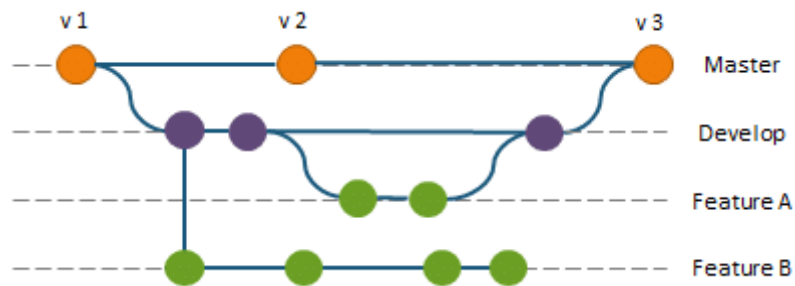
Now in this Git tutorial, let us look at the different Git workflows:



- **Centralized workflow:** In the Git centralized workflow, only one development branch is there, which is called 'master' and all changes are committed into this single branch.
- **Feature branching workflow:** In the feature branching workflow, feature development takes place only in a dedicated feature branch. The below-given image in this Git tutorial depicts the feature branching workflow:



- **Git workflow:** Instead of a single master branch, the Git workflow uses two branches. Here, the master branch stores the official release history, whereas the second 'develop' branch acts as an integration branch for features. The below-given image depicts the Git workflow:

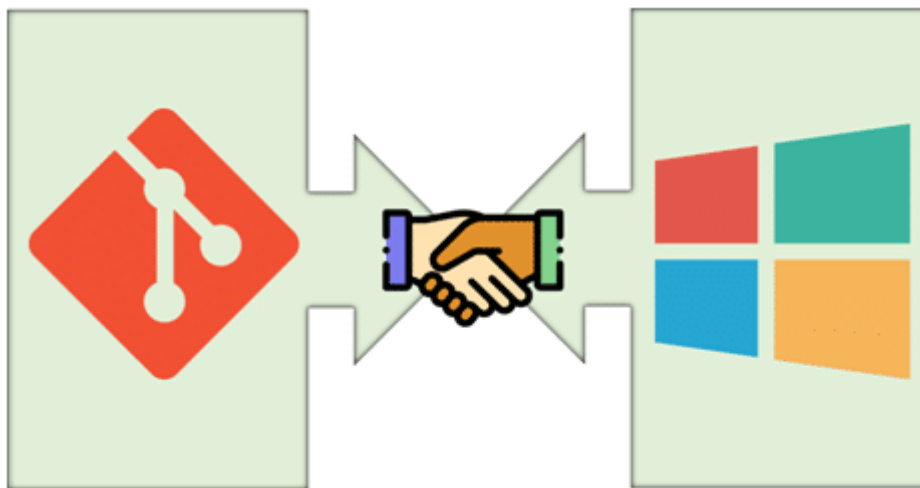


- **Forking workflow:** In the case of the forking workflow, the contributor has two Git repositories: one private local repository and the other public server-side repository.

This brings us to the end of the Git tutorial. In this Git tutorial, we have gone through the version control systems and its different types, the basics of Git, terminologies related to Git, Git installation in Windows, Linux, and on macOS systems, setting up and working on the GitHub repository, and various commands used in Git. We have also discussed some important operations in Git toward the end of this Git tutorial.

Git and Its Popularity

Git is one of the most important version control systems that has experienced a significant growth rate and popularity over the years. Git plays a key role in both developers' and non-developers' world. If we are part of the software developers' world, it is expected that we know Git. So, having a good grasp of Git is very beneficial for us to get into a lucrative career.



Even Microsoft has started leveraging Git very recently. This opens up more opportunities for the developers' community. Hence, it is the right time to learn Git.