Version Control System

A version control system is software that tracks changes to a file or set of files over time so that you can recall specific versions later. It also allows you to work together with other programmers.

The version control system is a collection of software tools that help a team to manage changes in a source code. It uses a special kind of database to keep track of every modification to the code.

Developers can compare earlier versions of the code with an older version to fix the mistakes.

Benefits of the Version Control System

The Version Control System is very helpful and beneficial in software development; developing software without using version control is unsafe. It provides backups for uncertainty. Version control systems offer a speedy interface to developers. It also allows software teams to preserve efficiency and agility according to the team scales to include more developers.

Types of Version Control System

- Localized version Control System
- Centralized version control systems
- Distributed version control systems

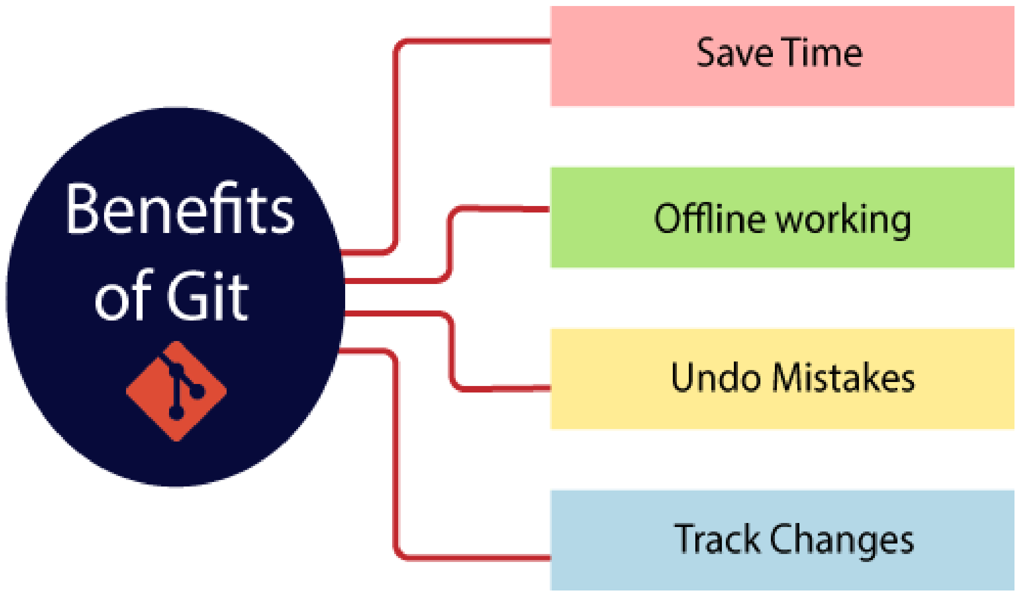
What is Git?

Git is an open-source distributed version control system. It is designed to handle minor to major projects with high speed and efficiency. It is developed to co-ordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace.

Scalability
Security
Open Source
Features of Git
Speed

- OpenSource
  Git is an open-source tool. It is released under the GPL (General Public License) license.
- Scalable
  Git is scalable, which means when the number of users increases, the Git can easily handle such situations.
- Distributed
  One of Git's great features is that it is distributed. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository. Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.
- security
  Git is secure. It uses the SHA1 (Secure Hash Function) to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.
- Speed
  Git is very fast, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a huge speed. Also, a centralized version control system continually communicates with a server somewhere. Performance tests conducted by Mozilla showed that it was extremely fast compared to other VCSs. Fetching version history from a locally stored repository is much faster than fetching it from the remote server. The core part of Git is written in C, which ignores runtime overheads associated with other high-level languages. Git was developed to work on the Linux kernel; therefore, it is capable enough to handle

large repositories effectively. From the beginning, speed and performance have been Git's primary goals.



Git is used for:

- Tracking code changes
- Tracking who made changes
- Coding collaboration

What does Git do?

- Manage projects with Repositories
- Clone a project to work on a local copy
- Control and track changes with Staging and Committing
- Branch and Merge to allow for work on different parts and versions of a project
- Pull the latest version of the project to a local copy
- Push local updates to the main project

Working with Git

- Initialize Git on a folder, making it a Repository
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered modified
- You select the modified files you want to Stage
- The Staged files are Committed, which prompts Git to store a permanent snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

Why Git?

- Over 70% of developers use Git!
- Developers can work together from anywhere in the world.
- Developers can see the full history of the project.
- Developers can revert to earlier versions of a project.

What is GitHub?

- Git is not the same as GitHub.
- GitHub makes tools that use Git.
- GitHub is the largest host of source code in the world, and has been owned by Microsoft since 2018.

Git Install

You can download Git for free from the following website: https://www.git-scm.com/

Using Git with Command Line

To start using Git, we are first going to open up our Command shell.

For Windows, you can use Git bash, which comes included in Git for Windows. For Mac and Linux you can use the built-in terminal.

The first thing we need to do, is to check if Git is properly installed:

```
git --version
git version 2.30.2.windows.1
```

If Git is installed, it should show something like git version X.Y

Configure Git

Now let Git know who you are. This is important for version control systems, as each Git commit uses this information

```
git config --global user.name "Jahir"

git config --global user.email "drjahirpasha@gpcet.ac.in"
```

Change the user name and e-mail address to your own. You will probably also want to use this when registering to GitHub later on.

Creating Git Folder

Now, let's create a new folder for our project:

```
mkdir myproject
cd myproject
```

mkdir makes a new directory.

cd changes the current working directory.

Now that we are in the correct directory. We can start by initializing Git!

Initialize Git

Once you have navigated to the correct folder, you can initialize Git on that folder:

```
git init
```

```
Initialized empty Git repository in /Users/user/myproject/.git/
```

Git New Files

Git Adding New Files

You just created your first local Git repo. But it is empty.

So let's add some files, or create a new file using your favourite text editor. Then save or move it to the folder you just created

for this example, I am going to use a simple HTML file like this:

Example

```
<html>
<head>
<title>Hello World!</title>
</head>
<body>

<h1>Hello world!</h1>
<p>This is the first file in my new Git Repo.</p>

</body>
</html>
```

And save it to our new folder as index.html.

Let's go back to the terminal and list the files in our current working directory:

```
ls

index.html
```

ls will list the files in the directory. We can see that index.html is there.

Then we check the Git status and see if it is a part of our repo:

```
git status

On branch master

No commits yet

Untracked files:

  (use "git add ..." to include in what will be committed)     index.html nothing added to commit
but untracked files present (use "git add" to track)
```

Now Git is aware of the file, but has not added it to our repository!

Files in your Git repository folder can be in one of 2 states:

- Tracked - files that Git knows about and are added to the repository
- Untracked - files that are in your working directory, but not added to the repository

When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

Git Staging Environment

One of the core functions of Git is the concepts of the Staging Environment, and the Commit.

As you are working, you may be adding, editing and removing files. But whenever you hit a milestone or finish a part of the work, you should add the files to a Staging Environment.

Staged files are files that are ready to be committed to the repository you are working on. You will learn more about commit shortly.

For now, we are done working with index.html. So we can add it to the Staging Environment:

```
git add index.html
```

The file should be Staged. Let's check the status:

```
git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached ..." to unstage)     new file: index.html
```

Now the file has been added to the Staging Environment.

Git Add More than One File

You can also stage more than one file at a time. Let's add 2 more files to our working folder. Use the text editor again.

A README.md file that describes the repository (recommended for all repositories):

Example

```
# hello-world
Hello World repository for Git tutorial
This is an example repository for the Git tutorial.
```

A basic external style sheet (bluestyle.css):

Example

```
body {
background-color: lightblue;
}

h1 {
color: navy;
margin-left: 20px;
}
```

And update index.html to include the stylesheet:

Example

```
<html>
<head>
<title>Hello World!</title>
<link rel="stylesheet" href="bluestyle.css">
</head>
<body>

<h1>Hello world!</h1>
<p>This is the first file in my new Git Repo.</p>

</body>
</html>
```

Now add all files in the current directory to the Staging Environment:

Example

```
git add --all
```

Using --all instead of individual filenames will stage all changes (new, modified, and deleted) files.

Example

```
git status

On branch master



No commits yet



Changes to be committed:
 (use "git rm --cached ..." to unstage)
        new file:   README.md
        new file:   bluestyle.css
        new file:   index.html
```

Now all 3 files are added to the Staging Environment, and we are ready to do our first commit.

Note: The shorthand command for git add --all is git add -A

Git Commit

Since we have finished our work, we are ready move from stage to commit for our repo.

Adding commits keep track of our progress and changes as we work. Git considers each commit change point or "save point". It is a point in the project you can go back to if you find a bug, or want to make a change.

When we commit, we should always include a message.

By adding clear messages to each commit, it is easy for yourself (and others) to see what has changed and when.

Example

```
git commit -m "First release of Hello World!"
[master (root-commit) 221ec6e] First release of Hello World!
 3 files changed, 26 insertions(+)
 create mode 100644 README.md
 create mode 100644 bluestyle.css
 create mode 100644 index.html
```

The commit command performs a commit, and the -m "message" adds a message.

The Staging Environment has been committed to our repo, with the message:
"First release of Hello World!"

Git Commit without Stage

Sometimes, when you make small changes, using the staging environment seems like a waste of time. It is possible to commit changes directly, skipping the staging environment. The -a option will automatically stage every changed, already tracked file.

Let's add a small update to index.html:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World!</title>
<link rel="stylesheet" href="bluestyle.css">
</head>
<body>

<h1>Hello world!</h1>
<p>This is the first file in my new Git Repo.</p>
<p>A new line in our file!</p>

</body>
</html>
```

And check the status of our repository. But this time, we will use the --short option to see the changes in a more compact way:

Example

```
git status --short

 M index.html
```

Working with Git Branches

In Git, a branch is a new/separate version of the main repository.

Let's say you have a large project, and you need to update the design on it.

How would that work without and with Git:

Without Git:

- Make copies of all the relevant files to avoid impacting the live version
- Start working with the design and find that code depend on code in other files, that also need to be changed!
- Make copies of the dependant files as well. Making sure that every file dependency references the correct file name
- EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
- Save all your files, making a note of the names of the copies you were working on
- Work on the unrelated error and update the code to fix it
- Go back to the design, and finish the work there
- Copy the code or rename the files, so the updated design is on the live version
- (2 weeks later, you realize that the unrelated error was not fixed in the new design version because you copied the files before the fix)

With Git:

- With a new branch called new-design, edit the code directly without impacting the main branch
- EMERGENCY! There is an unrelated error somewhere else in the project that needs to be fixed ASAP!
- Create a new branch from the main project called small-error-fix
- Fix the unrelated error and merge the small-error-fix branch with the main branch
- You go back to the new-design branch, and finish the work there
- Merge the new-design branch with main (getting alerted to the small error fix that you were missing)

Branches allow you to work on different parts of a project without impacting the main branch.

When the work is complete, a branch can be merged with the main project.

You can even switch between branches and work on different projects without them interfering with each other.

Branching in Git is very lightweight and fast!

New Git Branch

Let add some new features to our index.html page.

We are working in our local repository, and we do not want to disturb or possibly wreck the main project.

So we create a new branch:

Example

```
git branch hello-world-images
```

Now we created a new branch called "hello-world-images"

Let's confirm that we have created a new branch:

Example

```
git branch

  hello-world-images

* master
```

We can see the new branch with the name "hello-world-images", but
the * beside master specifies that we are currently on that branch.

checkout is the command used to check out a branch. Moving us from the current branch, to the one specified at the end of the command:

Example

```
git checkout hello-world-images

Switched to branch 'hello-world-images'
```

Now we have moved our current workspace from the master branch, to the new branch

Open your favourite editor and make some changes.

For this example, we added an image (img_hello_world.jpg) to the working folder and a line of code in the index.html file:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World!</title>
<link rel="stylesheet" href="bluestyle.css">
</head>
<body>

<h1>Hello world!</h1>
<div><img src="img_hello_world.jpg" alt="Hello World from Space"
style="width:100%;max-width:960px"></div>
<p>This is the first file in my new Git Repo.</p>
```

```
<p>A new line in our file!</p>

</body>
</html>
```

We have made changes to a file and added a new file in the working directory (same directory as the main branch).

Now check the status of the current branch:

Example

```
git status

On branch hello-world-images

Changes not staged for commit:

  (use "git add ..." to update what will be committed)

  (use "git restore ..." to discard changes in working directory)

        modified:   index.html



Untracked files:

  (use "git add ..." to include in what will be committed)

        img_hello_world.jpg



no changes added to commit (use "git add" and/or "git commit -a")
```

So let's go through what happens here:

- There are changes to our index.html, but the file is not staged for commit
- img_hello_world.jpg is not tracked

So we need to add both files to the Staging Environment for this branch:

Example

```
git add --all
```

Using --all instead of individual filenames will Stage all changed (new, modified, and deleted) files.

Check the status of the branch:

Example

```
git status

On branch hello-world-images

Changes to be committed:

  (use "git restore --staged ..." to unstage)

    new file: img_hello_world.jpg

    modified: index.html
```

We are happy with our changes. So we will commit them to the branch:

Example

```
git commit -m "Added image to Hello World"

[hello-world-images 0312c55] Added image to Hello World

2 files changed, 1 insertion(+)

create mode 100644 img_hello_world.jpg
```

Now we have a new branch, that is different from the master branch.

Switching Between Branches

Now let's see just how quick and easy it is to work with different branches, and how well it works.

We are currently on the branch hello-world-images. We added an image to this branch, so let's list the files in the current directory:

Example

```
ls

README.md  bluestyle.css  img_hello_world.jpg  index.html
```

We can see the new file img_hello_world.jpg, and if we open the html file, we can see the code has been altered. All is as it should be.

Now, let's see what happens when we change branch to master

Example

```
git checkout master

Switched to branch 'master'
```

The new image is not a part of this branch. List the files in the current directory again:

Example

```
ls

README.md  bluestyle.css  index.html
```

img_hello_world.jpg is no longer there! And if we open the html file, we can see the code reverted to what it was before the alteration.

Merge Branches

We have the emergency fix ready, and so let's merge the master and emergency-fix branches.

First, we need to change to the master branch:

Example

```
git checkout master

Switched to branch 'master'
```

Now we merge the current branch (master) with emergency-fix:

Example

```
git merge emergency-fix

Updating 09f4acd..dfa79db

Fast-forward

 index.html | 2 +-

 1 file changed, 1 insertion(+), 1 deletion(-)
```

Since the emergency-fix branch came directly from master, and no other changes had been made to master while we were working, Git sees this as a continuation of master. So it can "Fast-forward", just pointing both master and emergency-fix to the same commit.

As master and emergency-fix are essentially the same now, we can delete emergency-fix, as it is no longer needed:

Example

```
git branch -d emergency-fix

Deleted branch emergency-fix (was dfa79db).
```

Merge Conflict

Now we can move over to hello-world-images and keep working. Add another image file (img_hello_git.jpg) and change index.html, so it shows it:

Example

```
git checkout hello-world-images

Switched to branch 'hello-world-images'
```

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World!</title>
<link rel="stylesheet" href="bluestyle.css">
</head>
<body>

<h1>Hello world!</h1>
<div><img src="img_hello_world.jpg" alt="Hello World from Space" style="width:100%;max-width:960px"></div>
<p>This is the first file in my new Git Repo.</p>
<p>A new line in our file!</p>
<div><img src="img_hello_git.jpg" alt="Hello Git" style="width:100%;max-width:640px"></div>

</body>
</html>
```

Now, we are done with our work here and can stage and commit for this branch:

Example

```
git add --all

git commit -m "added new image"

[hello-world-images 1f1584e] added new image
```

```
 2 files changed, 1 insertion(+)
 create mode 100644 img_hello_git.jpg
```

We see that index.html has been changed in both branches. Now we are ready to merge hello-world-images into master. But what will happen to the changes we recently made in master?

Example

```
git checkout master
git merge hello-world-images
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

The merge failed, as there is conflict between the versions for index.html. Let us check the status:

Example

```
git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   img_hello_git.jpg
        new file:   img_hello_world.jpg

Unmerged paths:
  (use "git add ..." to mark resolution)
        both modified:   index.html
```

This confirms there is a conflict in index.html, but the image files are ready and staged to be committed.

So we need to fix that conflict. Open the file in our editor:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World!</title>
<link rel="stylesheet" href="bluestyle.css">
</head>
<body>

<h1>Hello world!</h1>
<div><img src="img_hello_world.jpg" alt="Hello World from Space" style="width:100%;max-
width:960px"></div>
<p>This is the first file in my new Git Repo.</p>
<<<<<<< HEAD
<p>This line is here to show how merging works.</p>
=======
<p>A new line in our file!</p>
<div><img src="img_hello_git.jpg" alt="Hello Git" style="width:100%;max-
width:640px"></div>
>>>>>>> hello-world-images

</body>
</html>
```

We can see the differences between the versions and edit it like we want:

Example

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World!</title>
<link rel="stylesheet" href="bluestyle.css">
</head>
<body>

<h1>Hello world!</h1>
<div><img src="img_hello_world.jpg" alt="Hello World from Space" style="width:100%;max-
width:960px"></div>
<p>This is the first file in my new Git Repo.</p>
<p>This line is here to show how merging works.</p>
```

```
<div><img src="img_hello_git.jpg" alt="Hello Git" style="width:100%;max-
width:640px"></div>

</body>
</html>
```

Now we can stage index.html and check the status:

Example

```
git add index.html

git status

On branch master

All conflicts fixed but you are still merging.

  (use "git commit" to conclude merge)



Changes to be committed:

        new file:   img_hello_git.jpg

        new file:   img_hello_world.jpg

        modified:   index.html
```

The conflict has been fixed, and we can use commit to conclude the merge:

Example

```
git commit -m "merged with hello-world-images after fixing conflicts"

[master e0b6038] merged with hello-world-images after fixing conflicts
```

And delete the hello-world-images branch:

Example

```
git branch -d hello-world-images

Deleted branch hello-world-images (was 1f1584e).
```