

GIT documentation



Tecdune Services LLP

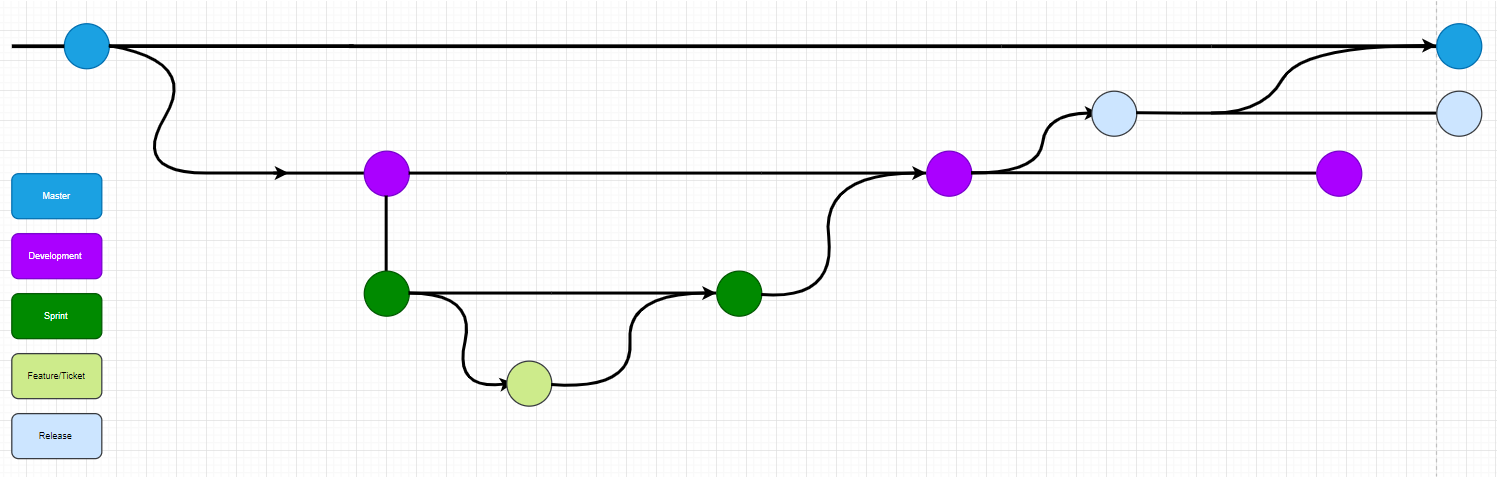
1. What is Git?

Git is a **distributed** version control system (DVCS), also known as decentralized. This means that **every developer has a full copy of the repository**, which is hosted in the cloud.

With Git, every developer has the full repository locally. So, the **developers can save the changes whenever they want**. If at certain moment the server hosting the repository is down, the developers can continue working without any problem. And the changes could be recorded into the shared repository later.

## 2. Gitflow Workflow

The Gitflow Workflow defines a strict branching model designed around the project release. This provides a robust framework for managing larger projects.

Git workflow diagram

**Develop and Master Branch**

Instead of a single master branch, this workflow uses two branches to record the history of the project. The master branch stores the official release history, and the develop branch serves as an integration branch for features. It's also convenient to tag all commits in the master branch with a version number.

**Sprint and features Branch**

As we are following agile methodology, we define our targets in small Sprints. This branch, records the history of task or feature we have done in that particular sprint.

Each new feature should reside in its own branch, which can be [pushed to the central repository](https://www.atlassian.com/git/tutorials/syncing/git-push) for backup/collaboration. But, instead of branching off of Develop, feature branches use Sprint as their parent branch. When a feature is complete, it gets [merged back into Sprint](https://www.atlassian.com/git/tutorials/using-branches/git-merge). Features should never interact directly with Develop.

**Release Branch**

Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into master and tagged with a version number. In addition, it should be merged back into develop, which may have progressed since the release was initiated.

Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also creates well-defined phases of development (e.g., it's easy to say, “This week we're preparing for version 4.0,” and to actually see it in the structure of the repository).

**Hotfix/ Live bug fixes Branch**

## 

Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on master instead of develop. This is the only branch that should fork directly off of master. As soon as the fix is complete, it should be merged into both master and develop (or the current release branch), and master should be tagged with an updated version number.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with master.

**A complete example demonstrating a Feature Branch Flow is as follows. Assuming we have a repo setup with a master branch.**

git checkout master

**#A develop branch is created from master**

git checkout -b develop

**#A Sprint branch is created from develop**

git checkout -b Sprint branch

**#A feature branch is created from Sprint branch**

git checkout -b feature branch

**# When a feature is complete it is merged into the Sprint branch after the code review**

git checkout Sprint branch  
git merge feature branch

**#After merge we should delete the feature branch**

git branch -D feature branch

**# After testing of Sprint branch we should merge the sprint branch into develop branch and cut the new release branch from develop branch**  
git checkout develop

git merge Sprint branch

git checkout -b release branch

**# When the release branch is done after testing, it is merged into master**

git checkout master

git merge release branch

**# When our release branch is deployed on production, delete the Sprint branch**

git branch -D Sprint branch

**# If an issue is detected in production, either from master or from release branch depends on which branch we are using for production  a hotfix branch is created from them.**

git checkout release branch

git checkout -b hotfix branch

**# Once work is done commits are added to the hotfix branch merged to release branch.**

git checkout release branch

git merge hotfix branch

**# Now we have to merge the release branch into both master and develop.**

git checkout master

git merge release branch

git checkout develop

git merge release branch

## 3. Download and install Git

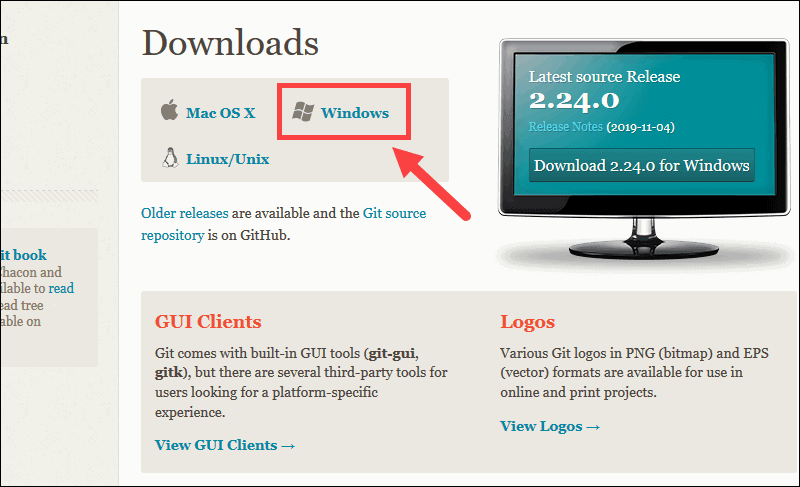
## Linux

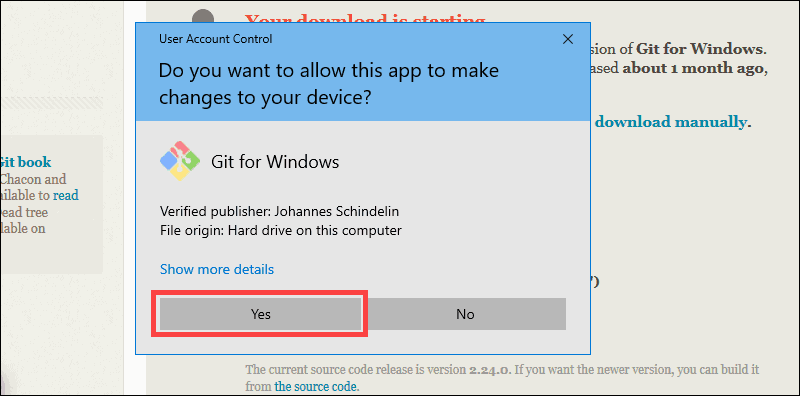
As you probably have guessed, Git can be installed in Linux executing the following commands:

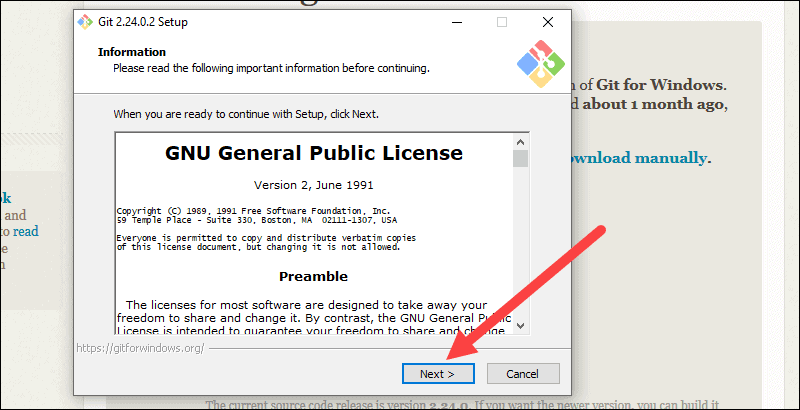
|  |  |
| --- | --- |
|  | **sudo apt-get update**  **sudo apt-get install git**  Now, press **y** and then press **<Enter>** to continue.    Git should be installed.  Now, run the following command to check whether Git is working correctly:  *$*git –version    As you can see, the version of is showing. It’s working correctly as well. |

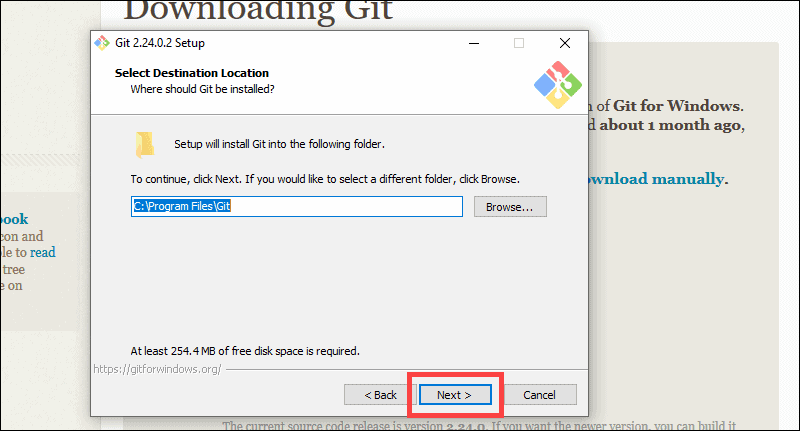
## Window

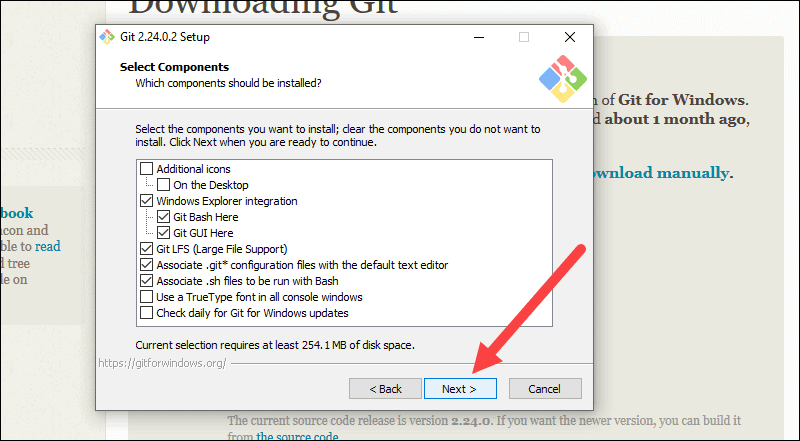
1. Browse to the official Git website: <https://git-scm.com/downloads>
2. Click the download link for Windows and allow the download to complete.



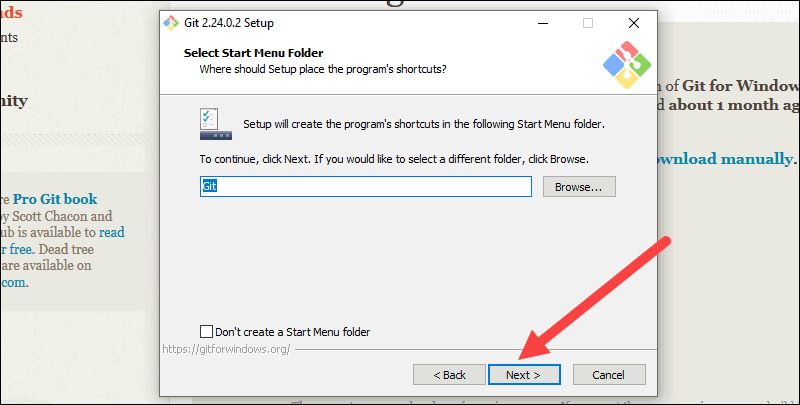
1. Browse to the download location (or use the download shortcut in your browser). Double-click the file to extract and launch the installer.
2. Allow the app to make changes to your device by clicking **Yes** on the User Account Control dialog that opens.
3. Review the GNU General Public License, and when you’re ready

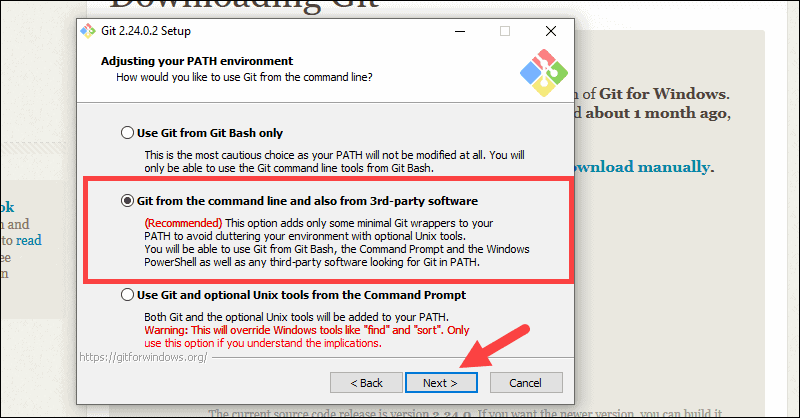
to install, click **Next**.

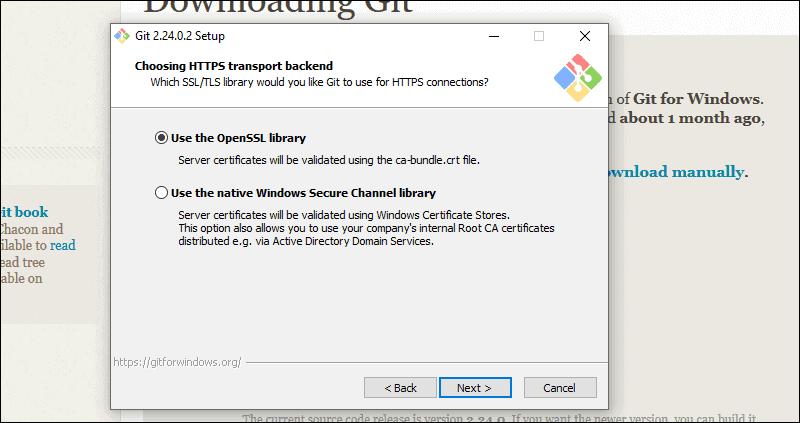
1. The installer will ask you for an installation location. Leave the default, unless you have reason to change it, and click **Next**.
2. A component selection screen will appear. Leave the defaults unless you have a specific need to change them and click **Next**.

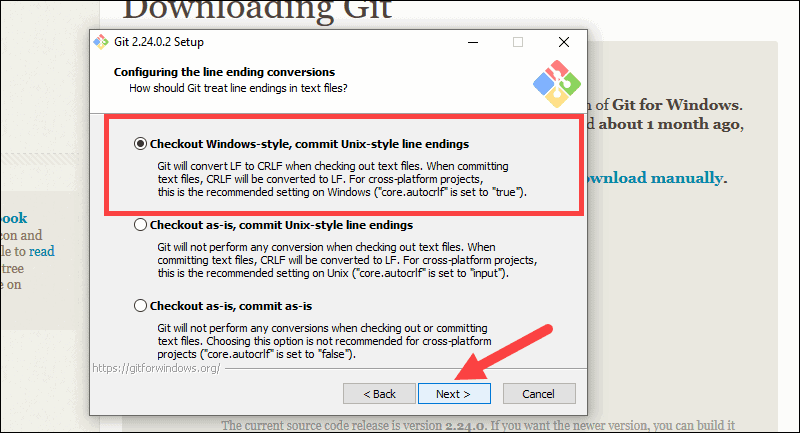


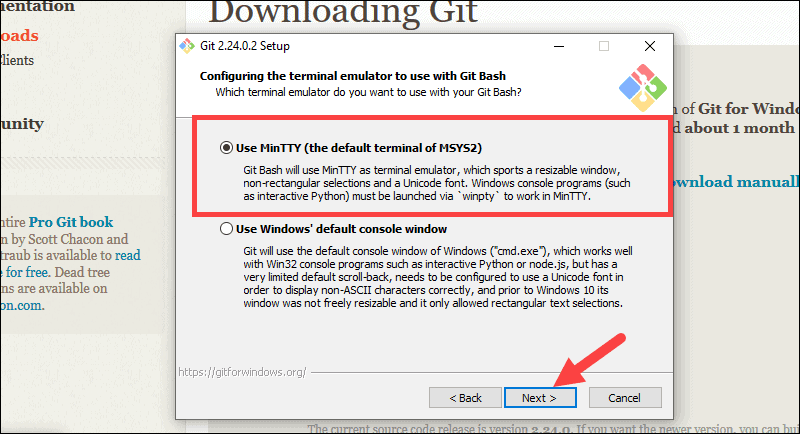
1. The installer will offer to create a start menu folder. Simply click **Next**.



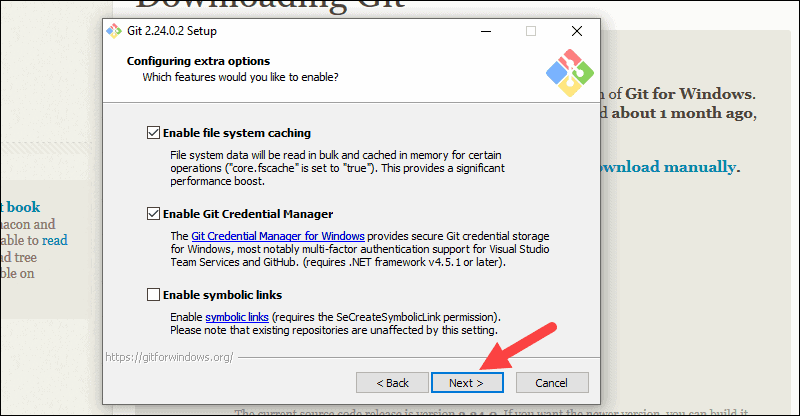
1. Select a text editor you’d like to use with Git. Use the drop-down menu to select Notepad++ (or whichever text editor you prefer) and click **Next**.
2. This installation step allows you to change the **PATH environment**. The **PATH**is the default set of directories included when you run a command from the command line. Leave this on the middle (recommended) selection and click **Next**.
3. The next option relates to server certificates. Most users should use the default. If you’re working in an Active Directory environment, you may need to switch to Windows Store certificates. Click **Next**.

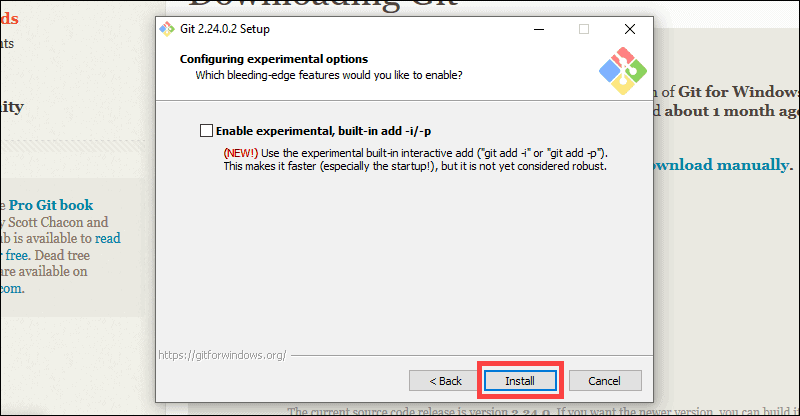
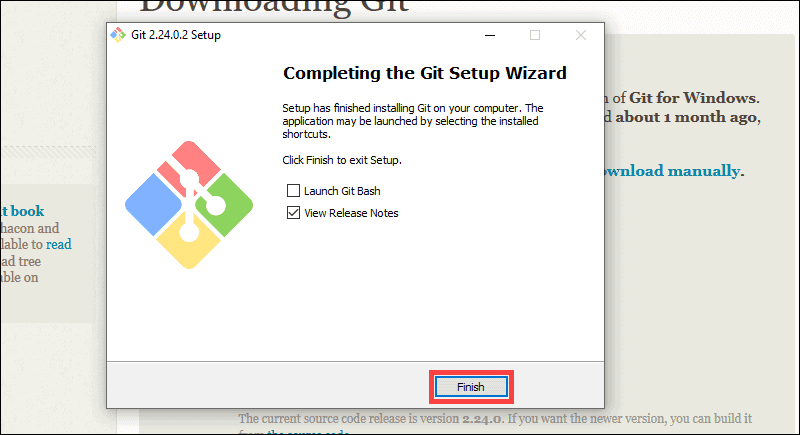


1. The next selection converts line endings. It is recommended that you leave the default selection. This relates to the way data is formatted and changing this option may cause problems. Click **Next**.
2. Choose the terminal emulator you want to use. The default MinTTY is recommended, for its features. Click **Next**.



1. The default options are recommended, however this step allows you to decide which extra option you would like to enable. If you use symbolic links, which are like shortcuts for the command line, tick the box. Click **Next**.



1. Depending on the version of Git you’re installing, it may offer to install experimental features. At the time this article was written, the option to include interactive options was offered. Unless you are feeling adventurous, leave them unchecked and click **Install**.
2.  Once the installation is complete, tick the boxes to view the Release Notes or Launch Git Bash, then click **Finish**.

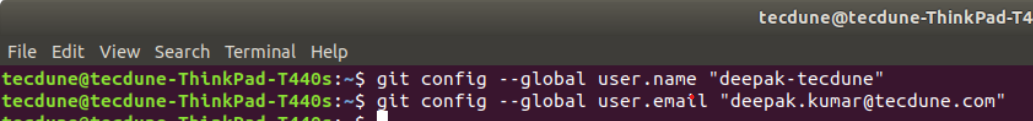
## 4. Git Basics Commands With Usage

* Linux

Configuring GitHub:

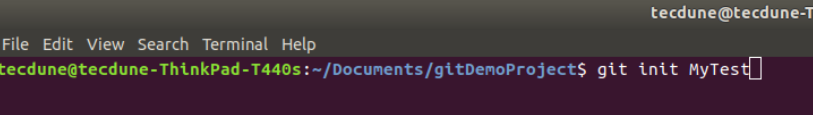
Once the installation has successfully completed, the next thing to do is to set up the configuration details of the GitHub user. To do this use the following two commands by replacing "user\_name" with your GitHub username and replacing "email\_id" with your email-id you used to create your GitHub account.

**git config --global user.name "user\_name"**

**git config --global user.email "email\_id"**

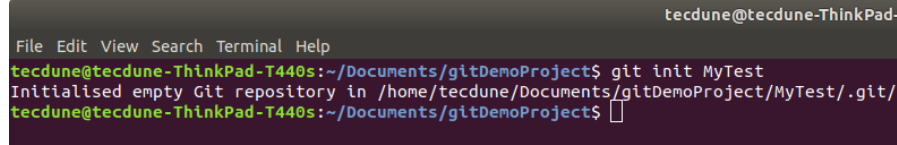
Creating a local repository

Create a folder on your system. This will serve as a local repository which will later be pushed onto the GitHub website. Use the following command:

**git init Mytest**

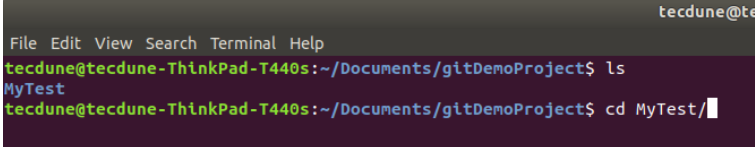
If the repository is created successfully, then you will get the following line:

***initialized empty Git repository in /home/Tecdune/Document/gitDemoProject/Mytest/.git/***

******

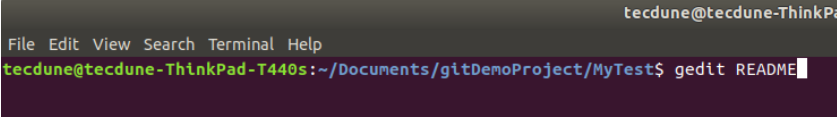
This line may vary depending on your system.

So here, Mytest is the folder that is created and "init" makes the folder a GitHub repository. Change the directory to this newly created folder:

***cd Mytest***

Creating a README file to describe the repository

Now create a README file and enter some text like "this is a git setup on Linux". The README file is generally used to describe what the repository contains or what the project is all about. Example:

***gedit README***

You can use any other text editors. I use **gedit**. The content of the **README** file will be:

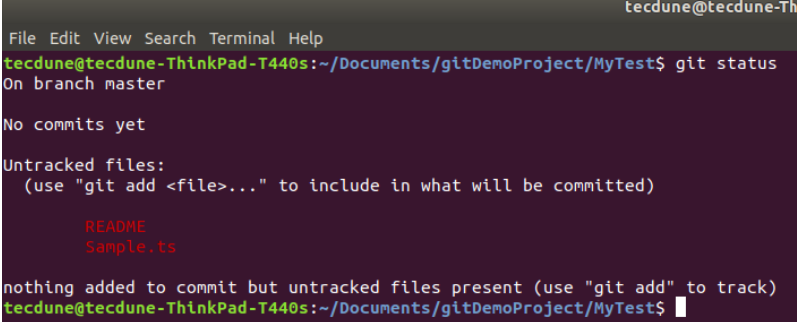
*This is a git repo*

Adding repository files to an index

This is an important step. Here we add all the things that need to be pushed onto the website into an index. These things might be the text files or programs that you might add for the first time into the repository or it could be adding a file that already exists but with some changes (a newer version/updated version).

Here we already have the **README** file. So, let's create another file which contains a simple typescript program and call it sample.ts.

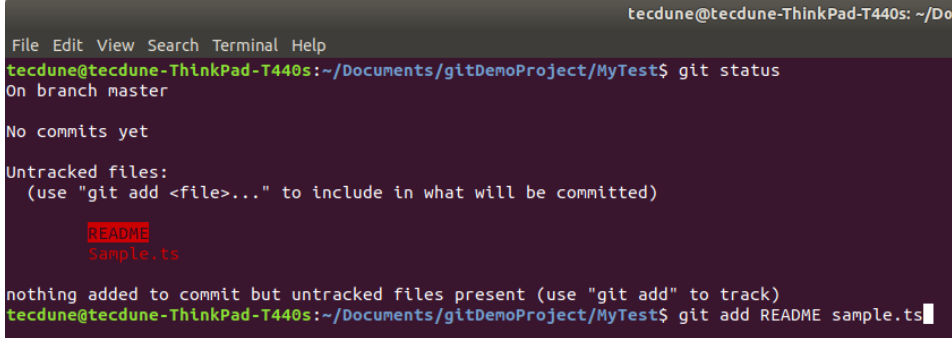
So, now that we have 2 files

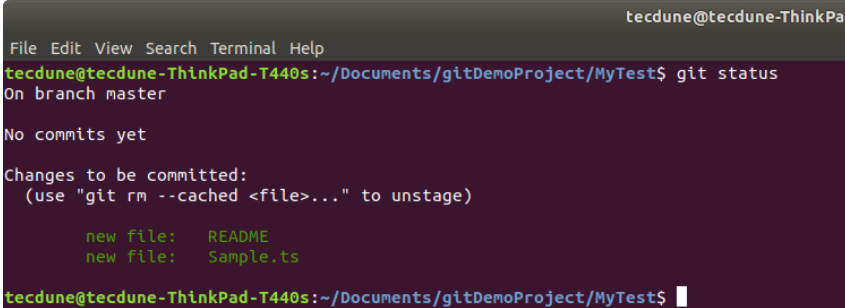
***README and sample.ts***

add it to the index by using the following 2 commands:

***git add README***

***git add sample.ts***

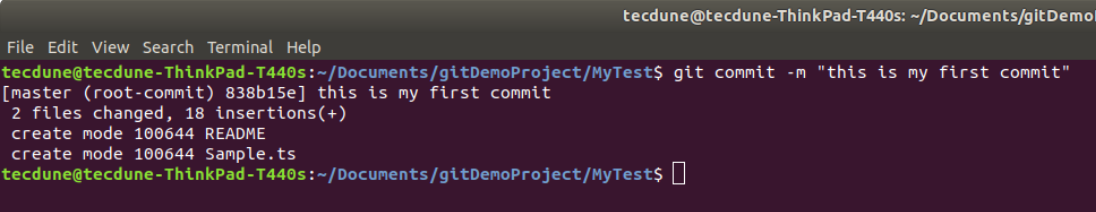
******



Note that the "**git add**" command can be used to add any number of files and folders to the index. Here, when I say index, what I am referring to is a buffer like space that stores the files/folders that have to be added into the Git repository.

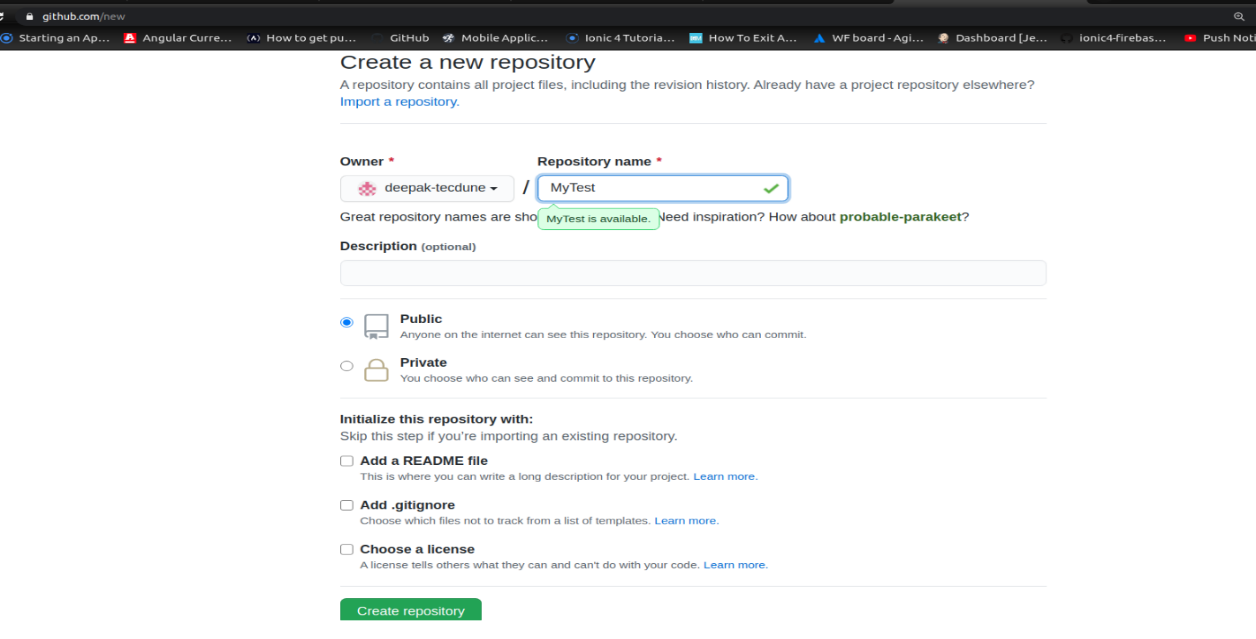
Committing changes made to the index

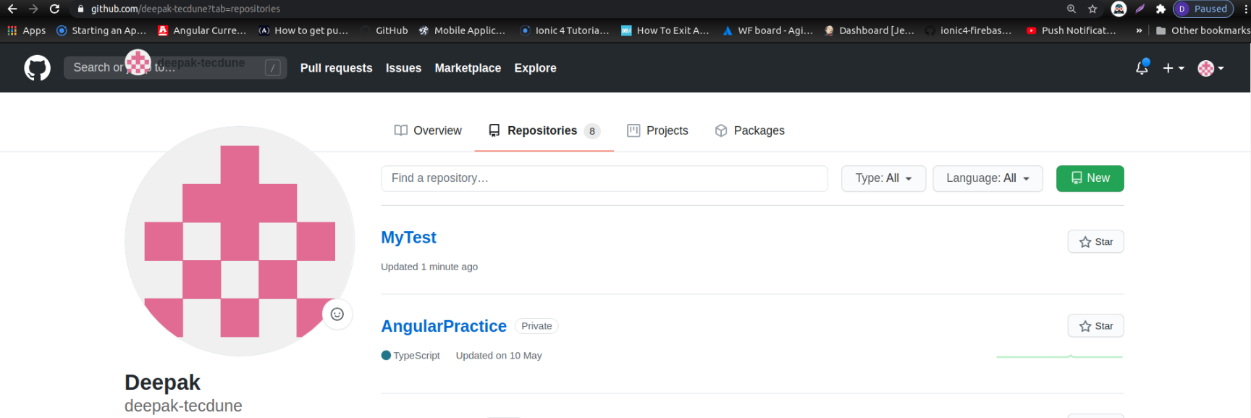
Once all the files are added, we can commit it. This means that we have finalized what additions and/or changes have to be made and they are now ready to be uploaded to our repository. Use the command :

***git commit -m "some message"***

"some message" in the above command can be any simple message like "my first commit" or "edit in readme", etc.

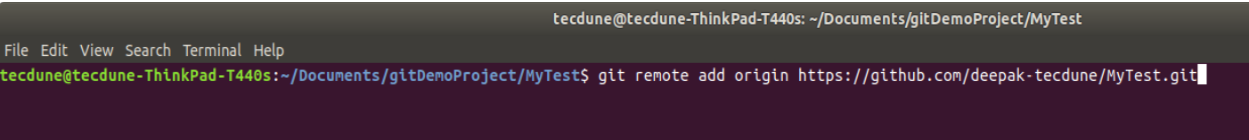
Creating a repository on GitHub

Create a repository on GitHub. Notice that the name of the repository should be the same as the repository's on the local system. In this case, it will be "Mytest". To do this login to your account on https://github.com. Then click on the "plus(+)" symbol at the top right corner of the page and select "create new repository". Fill the details as shown in the image below and click on "create repository" button.



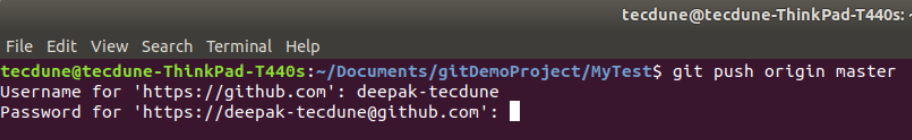
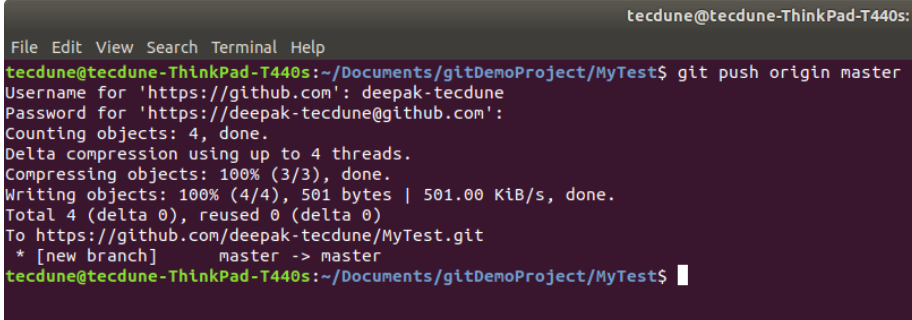
Once this is created, we can push the contents of the local repository onto the GitHub repository in your profile. Connect to the repository on GitHub using the command:

**Important Note:** Make sure you replace 'user\_name' and 'Mytest' in the path with your Github username and folder before running the command!

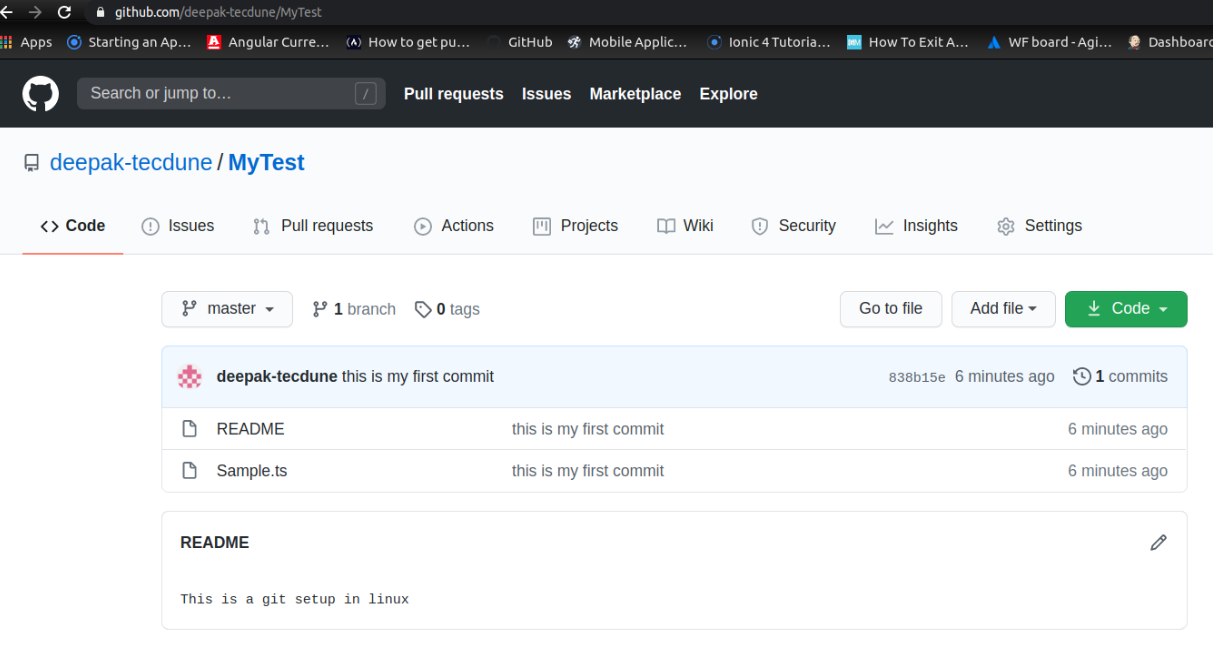
***git remote add origin*** [***https://github.com/user\_name/Mytest.git***](https://github.com/user_name/Mytest.git)

The final step is to push the local repository contents into the remote host repository (GitHub), by using the command:

***git push origin master***

Enter the login credentials [user\_name and password].

So this adds all the contents of the 'Mytest' folder (my local repository) to GitHub. For subsequent projects or for creating repositories, you can start off with step 3 directly. Finally, if you log in to your GitHub account and click on your Mytest repository, you can see that the 2 files **README** and **sample.ts** have been uploaded and are visible to all as shown in the following image.

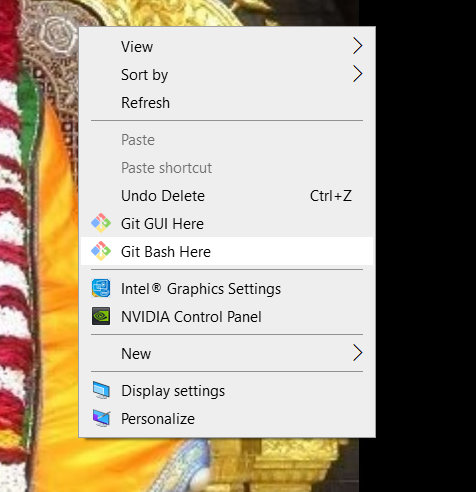


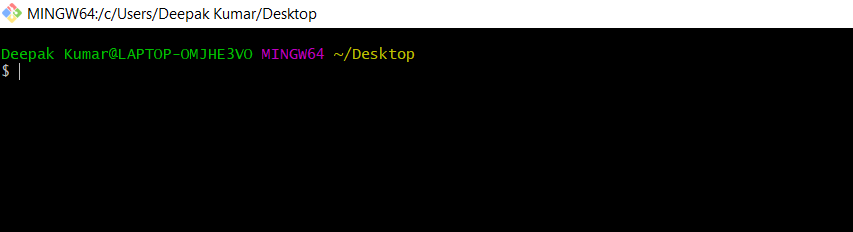
* Windows

### 

In windows we follow the same commands, from init to push the code in github repo.

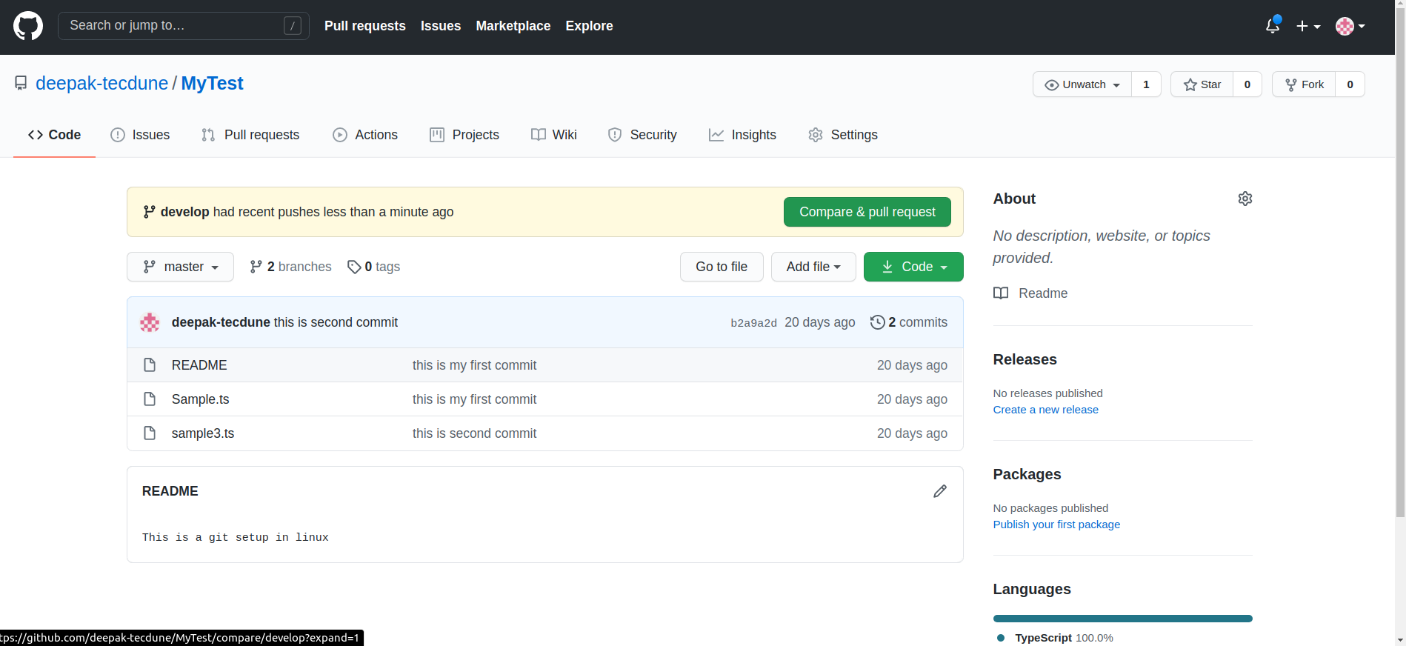
In order to do that, we type the commands in Git Bash.



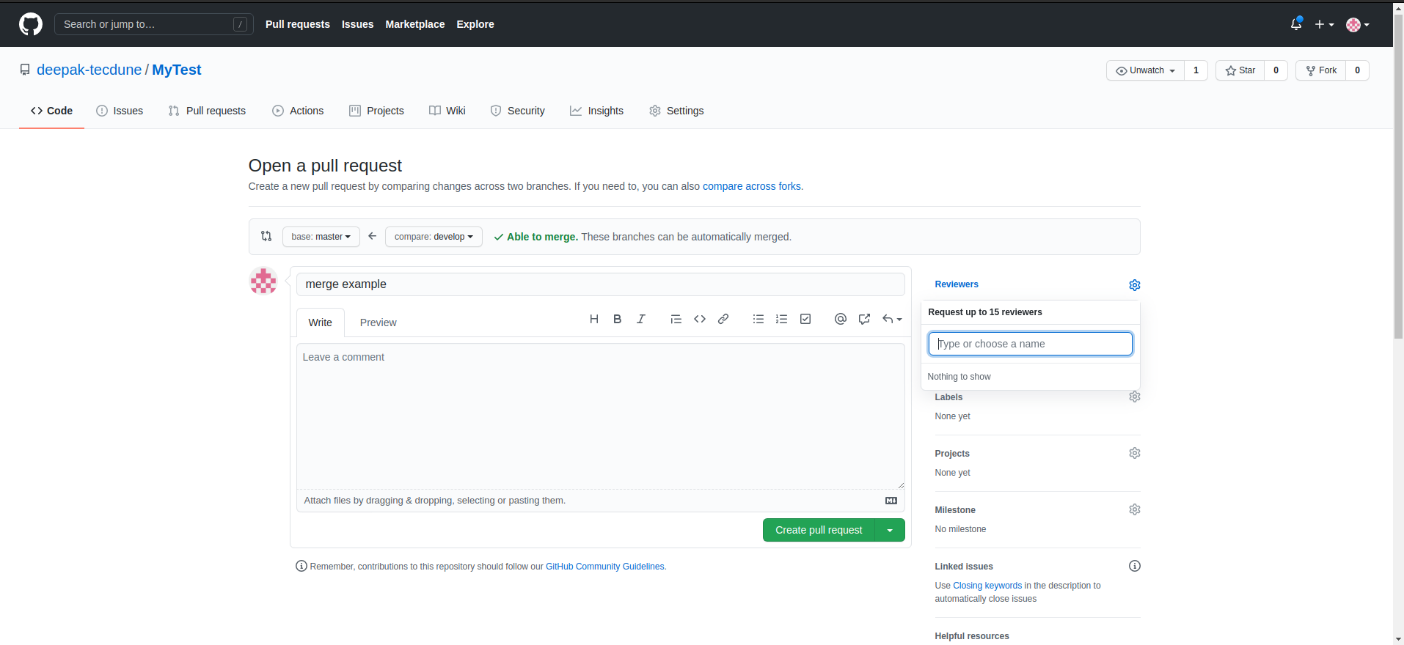


**How to create a pull request in GitHub**

Once you push the changes to your ticket branch or repo, the Compare & pull request button will appear in GitHub for merging in main or Sprint branch.



Click it and you'll be taken to this screen:



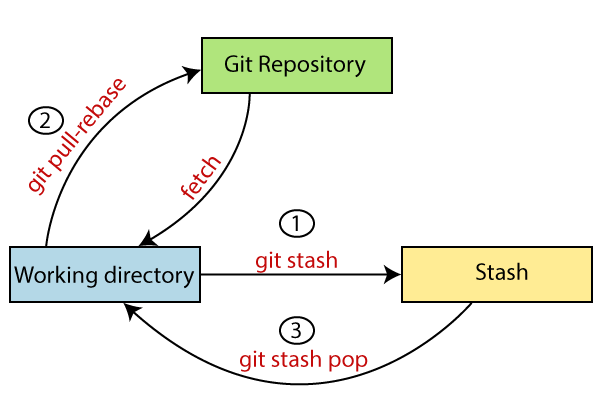
Open a pull request by clicking the Create pull request button. This allows the repo's maintainers or reviewer which you opt( show in above pic in right panel) to review your contribution. From here, they can merge it if it is good, or they may ask you to make some changes.

In summary, if you want to contribute to a project, the simplest way is to:

* Find a project you want to contribute to
* Fork it(optional)
* Clone it to your local system
* Make a new branch
* Make your changes
* Push it back to your repo
* Click the Compare & pull request button (branch in which you want to contribute)
* Click Create pull request to open a new pull request

If the reviewers ask for changes, repeat steps 5 and 6 to add more commits to your pull request.

# **How to git stash**

Sometimes you want to switch the branches, but you are working on an incomplete part of your current project. You don't want to make a commit of half-done work. Git stashing allows you to do so. The **git stash command** enables you to switch branches without committing the current branch.

The below figure demonstrates the properties and role of stashing concerning repository and working directory.

Generally, the stash's meaning is "**store something safely in a hidden place**." The sense in Git is also the same for stash; Git temporarily saves your data safely without committing.

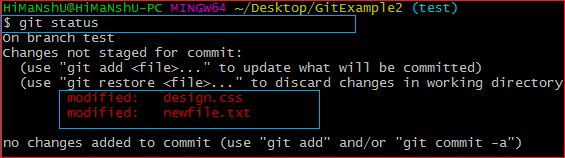
Stashing takes the messy state of your working directory, and temporarily save it for further use. Many options are available with git stash. Some useful options are given below:

* **Git stash**
* **Git stash save**
* **Git stash list**
* **Git stash apply**
* **Git stash changes**
* **Git stash pop**
* **Git stash drop**
* **Git stash clear**
* **Git stash branch**

**Syntax:**

1. $ git status

**Output:**



From the above output, you can see the status that there are two untracked file **design.css** and **newfile.txt** available in the repository. To save it temporarily, we can use the git stash command. The git stash command is used as:

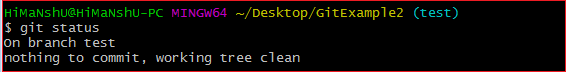
**Syntax:**

1. $ git stash

**Output:**



In the given output, the work is saved with git stash command. We can check the status of the repository.



As you can see, my work is just stashed in its current position. Now, the directory is cleaned. At this point, you can switch between branches and work on them.

Git Stash Save (Saving Stashes with the message):

In Git, the changes can be stashed with a message. To stash a change with a message, run the below command:

**Syntax:**

1. $ git stash save "**<Stashing** Message**>**"

**Output:**

Git Stash

The above stash will be saved with a message

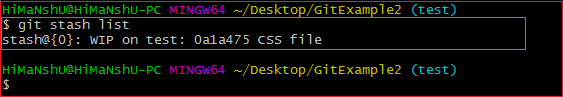
Git Stash List (Check the Stored Stashes)

To check the stored stashes, run the below command:

**Syntax:**

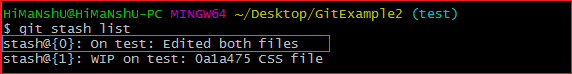
1. $ git stash list

**Output:**



In the above case, I have made one stash, which is displayed as "**stash@{0}: WIP on the test: 0a1a475 CSS file**".

If we have more than one stash, then It will display all the stashes respectively with different stash id. Consider the below output:



It will show all the stashes with indexing as **stash@{0}: stash@{1}:** and so on.

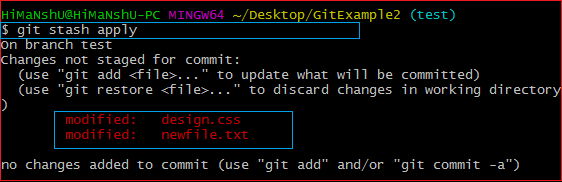
Git Stash Apply

You can re-apply the changes that you just stashed by using the git stash command. To apply the commit, use the git stash command, followed by the apply option. It is used as:

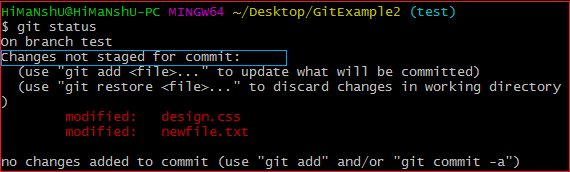
**Syntax:**

1. $ git stash apply

**Output:**



The above output restores the last stash. Now, if you will check the status of the repository, it will show the changes that are made on the file. Consider the below **output:**



From the above output, you can see that the repository is restored to its previous state before stash. It is showing output as "**Changes not staged for commit**."

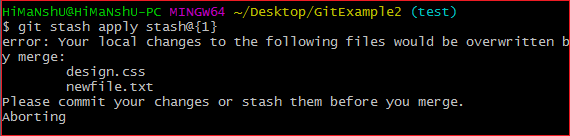
In case of more than one stash, you can use "git stash apply" command followed by stash index id to apply the particular commit. It is used as:

**Syntax:**

1. $ git stash apply **<stash** id**>**

Consider the below output:

**Output:**



If we don't specify a stash, Git takes the most recent stash and tries to apply it.

Git Stash Changes

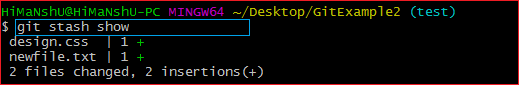
We can track the stashes and their changes. To see the changes in the file before stash and after stash operation, run the below command:

**Syntax:**

1. $ git stash show

The above command will show the file that is stashed and changes made on them. Consider the below output:

**Output:**



The above output illustrates that there are two files that are stashed, and two insertions performed on them.

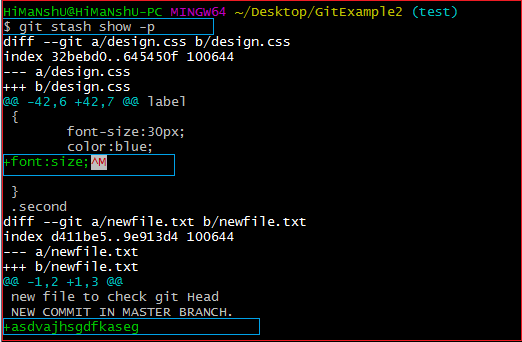
We can exactly track what changes are made on the file. To display the changed content of the file, perform the below command:

**Syntax:**

1. $ git stash show -p

Here, -p stands for the partial stash. The given command will show the edited files and content, consider the below output:

**Output:**



The above output is showing the file name with changed content. It acts the same as git diff command. The **git diff** command will also show the exact output.

Git Stash Pop (Reapplying Stashed Changes)

Git allows the user to re-apply the previous commits by using git stash pop command. The popping option removes the changes from stash and applies them to your working file.

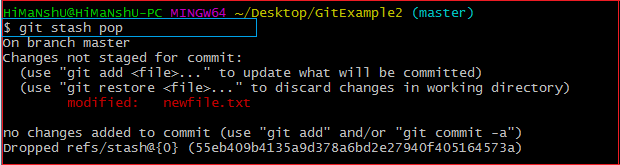
The git stash pop command is quite similar to git stash apply. The main difference between both of these commands is stash pop command that deletes the stash from the stack after it is applied.

**Syntax:**

1. $ git stash pop

The above command will re-apply the previous commits to the repository. Consider the below output.

**Output:**



Git Stash Drop (Unstash)

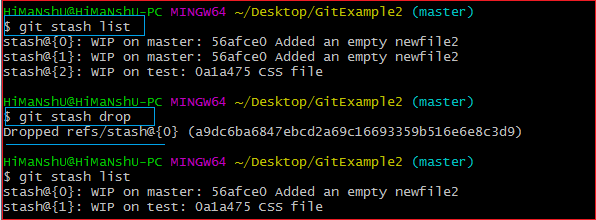
The **git stash drop** command is used to delete a stash from the queue. Generally, it deletes the most recent stash. Caution should be taken before using stash drop command, as it is difficult to undo if once applied.

The only way to revert it is if you do not close the terminal after deleting the stash. The stash drop command will be used as:

**Syntax:**

1. $ git stash drop

**Output:**



In the above output, the most recent stash **(stash@{0})** has been dropped from given three stashes. The stash list command lists all the available stashes in the queue.

We can also delete a particular stash from the queue. To delete a particular stash from the available stashes, pass the stash id in stash drop command. It will be processed as:

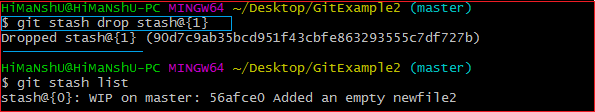
**Syntax:**

1. $ git stash drop **<stash** id**>**

Assume that I have two stashes available in my queue, and I don't want to drop my most recent stash, but I want to delete the older one. Then, it will be operated as:

1. $ git stash drop stash@{1}

Consider the below output:



In the above output, the commit **stash@{1}** has been deleted from the queue.

Git Stash Clear

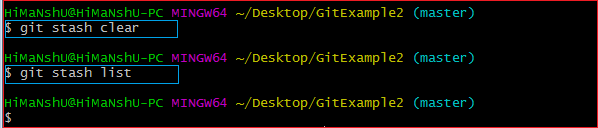
The **git stash clear** command allows deleting all the available stashes at once. To delete all the available stashes, operate below command:

**Syntax:**

1. $ git stash clear

it will delete all the stashes that exist in the repository.

**Output:**



All the stashes are deleted in the above output. The git stash list command is blank because there are no stashes available in the repository.

Git Stash Branch

If you stashed some work on a particular branch and continue working on that branch. Then, it may create a conflict during merging. So, it is good to stash work on a separate branch.

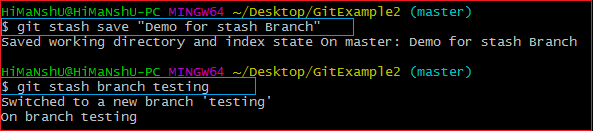
The git stash branch command allows the user to stash work on a separate branch to avoid conflicts. The syntax for this branch is as follows:

**Syntax:**

1. $ git stash branch **<Branch** Name**>**

The above command will create a new branch and transfer the stashed work on that. Consider the below output:

**Output:**



In the above output, the stashed work is transferred to a newly created branch testing. It will avoid the merge conflict on the master branch.

# **Git Checkout**

In Git, the term checkout is used for the act of switching between different versions of a target entity. The **git checkout** command is used to switch between branches in a repository. Be careful with your staged files and commits when switching between branches.

Operations on Git Checkout

We can perform many operations by git checkout command like the switch to a specific branch, create a new branch, checkout a remote branch, and more. The **git branch** and **git checkout** commands can be integrated.

Checkout Branch

You can demonstrate how to view a list of available branches by executing the git branch command and switch to a specified branch.

To demonstrate available branches in repository, use the below command:

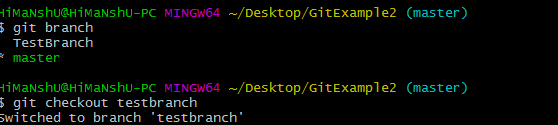
1. $ git branch

Now, you have the list of available branches. To switch between branches, use the below command.

**Syntax:**

1. $ git checkout <branchname>

**Output:**



As you can see in the given output that master branch has switched to TestBranch.

Create and Switch Branch

The git checkout commands let you create and switch to a new branch. You can not only create a new branch but also switch it simultaneously by a single command. The git checkout -b option is a convenience flag that performs run git branch <new-branch>operation before running git checkout <new-branch>.

**Syntax:**

1. $ git checkout -b <branchname>

**Output:**

Git Checkout

As you can see in the given output, branch3 is created and switched from the master branch.

Checkout Remote Branch

Git allows you to check out a remote branch by git checkout command. It is a way for a programmer to access the work of a colleague or collaborator for review and collaboration. Each remote repository contains its own set of branches. So, to check out a remote branch, you have first to fetch the contents of the branch.

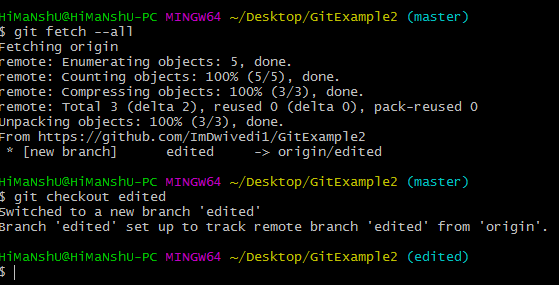
1. $ git fetch --all

In the latest versions of Git, you can check out the remote branch like a local branch.

**Syntax:**

1. $ git checkout <remotebranch>

**Output:**



In the above output, first, the fetch command is executed to fetch the remote data; after that, the checkout command is executed to check out a remote branch.

Edited is my remote branch. Here, we have switched to edited branch from master branch by git command line.

The earlier versions of Git require the creation of a new branch based on the remote. In earlier versions, below command is used to check out the remote branch.

1. $ git checkout <remotebranch> origin/<remotebranch>

# **Git Revert**

In Git, the term revert is used to revert some changes. The git revert command is used to apply revert operation. It is an undo type command. However, it is not a traditional undo alternative. It does not delete any data in this process; instead, it will create a new change with the opposite effect and thereby undo the specified commit. Generally, git revert is a commit.

It can be useful for tracking bugs in the project. If you want to remove something from history then git revert is a wrong choice.

Moreover, we can say that git revert records some new changes that are just opposite to previously made commits. To undo the changes, run the below command:

**Syntax:**

1. $ git revert

Git Revert Options:

Git revert allows some additional operations like editing, no editing, cleanup, and more. Let's understand these options briefly:

**< commit>:** The commit option is used to revert a commit. To revert a commit, we need the commit reference id. The git log command can access it.

1. $ git revert **<commit-ish>**

**<--edit>:** It is used to edit the commit message before reverting the commit. It is a default option in git revert command.

1. $ git revert -e **<commit-ish>**

**-m parent-number /--mainline parent-number:** it is used to revert the merging. Generally, we cannot revert a merge because we do not know which side of the merge should be considered as the mainline. We can specify the parent number and allows revert to reverse the change relative to the specified parent.

**-n/--no edit:** This option will not open a text editor. It will directly revert the last commit.

1. $ git revert -n **<commit-ish>**

**--cleanup=<mode>:** The cleanup option determines how to strip spaces and comments from the message.

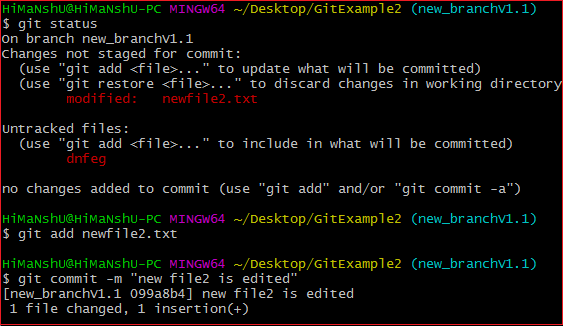
**-n/--no-commit:** Generally, the revert command commits by default. The no-commit option will not automatically commit. In addition, if this option is used, your index does not have to match the HEAD commit.

The no-commit option is beneficial for reverting more than one commits effect to your index in a row.

Let's understand how to revert the previous commits.

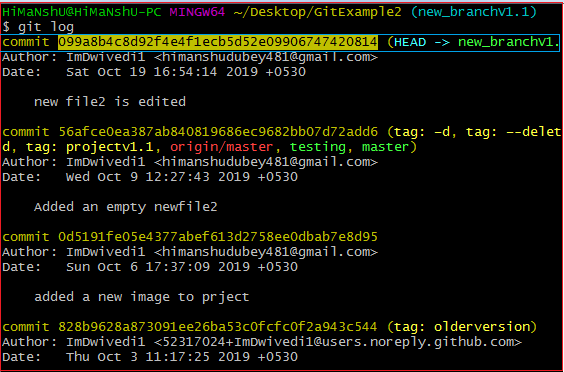
Git Revert to Previous Commit

Suppose you have made a change to a file say **newfile2.txt** of your project. And later, you remind that you have made a wrong commit in the wrong file or wrong branch. Now, you want to undo the changes you can do so. Git allows you to correct your mistakes. Consider the below image:



As you can see from the above output that I have made changes in newfile2.txt. We can undo it by git revert command. To undo the changes, we will need the commit-ish. To check the commit-ish, run the below command:

1. $ git log

Consider the below output:

In the above output, I have copied the most recent commit-ish to revert. Now, I will perform the revert operation on this commit. It will operate as:

1. $ git revert 099a8b4c8d92f4e4f1ecb5d52e09906747420814

The above command will revert my last commit. Consider the below output:



As you can see from the above output, the changes made on the repository have been reverted.

Git Revert Merge

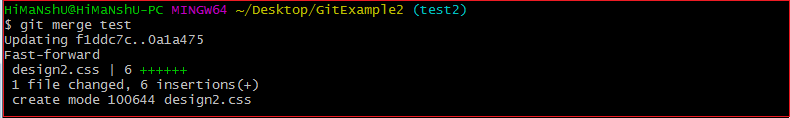
In Git, merging is also a commit that has at least two parents. It connects branches and code to create a complete project.

A merge in Git is a commit that has at least two parents. It brings together multiple lines of development. In a work-flow where features are developed in branches and then merged into a mainline, the merge commits would typically have two parents.

How to Revert a Merge

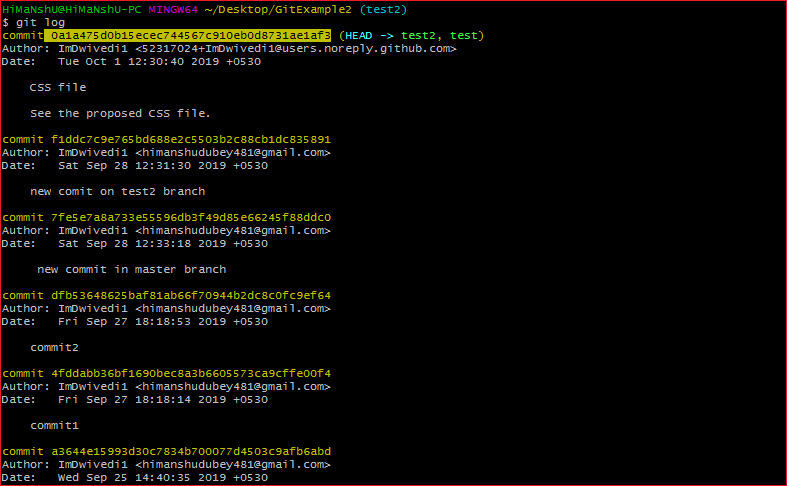
Usually, reverting a merge considered a complicated process. It can be complex if not done correctly. We are going to undo a merge operation with the help of git revert command. Although some other commands like git reset can do it. Let's understand how to revert a merge. Consider the below example.

I have made some changes to my file **design2.css** on the test and merge it with **test2**. Consider the below output:



To revert a merge, we have to get its reference number. To check commit history, run the below command:

1. $ git log

The above command will display the commit history. Consider the below output:

From the above output, copy your merging commit that you to want to revert and run the below command:

1. $ git revert **<commit** reference**>** -m 1

The above command will revert the merging operation. Here, -m 1 is used for the first parent as the mainline. Merge commit has multiple parents. The revert needs additional information to decide which parent of the merge shall be considered as the mainline. In such cases, the parameter -m is used. Consider the below output:



From the above output, we can see that the previous merge has been reverted.

References:

<https://www.javatpoint.com/git>

<https://www.atlassian.com/git>

<https://git-scm.com/docs/gittutorial>