



A Beginner-Friendly Guide for Linux / Start Learning Linux Quickly...

## Learn the Basics of Git to Manage Projects Efficiently {Beginner's Guide}

[James Kiarie](#) | Last Updated: May 29, 2023 | [Git](#) | [Leave a comment](#)

**Git** is the most popular version control system (**VCS**) used by developers and development teams to keep track of changes made to source code. In fact, it is the most popular version control system used in **DevOps**.

A Version Control System is software that tracks or records changes made to software code over time in a special database referred to as a repository. With **Git**, you can track project history and see what changes were made and who made them.

You can also roll back the changes to get back to your original code. In addition, **Git** allows multiple people to collaborate on the same project irrespective of their location.

Version Control Systems can be categorized into two: **Centralized** and **Distributed**.

[lwptoc]

### What Is a Centralized Version Control System

In a centralized system, all users connect to a central server to get a copy of software code and share it with others. Examples of centralized Version Control Systems include [Subversion](#) which is distributed under the Apache License and Microsoft Team Foundation Server (Microsoft TFS).

The drawback of centralized **VCS** is the existence of a single point of failure. When the central server goes down, the team members can no longer collaborate on the project.

## What Is a Distributed Version Control System

---

With distributed systems, each team member gets to have a copy of the code project on their local machine. If the central server experiences downtime, users can synchronize the software code with others without a problem.

Examples of distributed Version Control Systems include [Git](#) and [Mercurial](#).

## Why Use Git?

---

Of all the Version Control Systems that we have mentioned, **Git** is the most popular and widely used VCS. And for good reasons. It's free and open-source, scalable, fast, and reliable.

You can track changes made to code and easily collaborate on a project with other team members. In addition, operations such as branching and merging are far much easier in **Git** than in **Mercurial** or any other VCS.

Nowadays, **Git** is ubiquitous and is used by both small companies and large enterprises to efficiently manage software code. For that reason, it's one of the most in-demand skills for developers or those getting into software development.

So if you are looking into getting a job as a software developer, bear in mind that **Git** is a prerequisite. You should have a thorough knowledge and understanding of this Version Control System tool.

In this tutorial, we will walk you through **Git** basics and how you can efficiently manage your code.

## Introduction to Git and GitHub

---

We cannot talk about **Git** without mentioning GitHub. These two are intertwined and often go hand in hand. In fact, a lot of people think that **Git** and **GitHub** are the same. However, these are two separate things that integrate with each other.

[GitHub](#) is a code hosting platform for Git repositories. It's essentially a website where developers and project managers host their software projects.

It makes it easy for team members or developers to coordinate and track changes made to code, as well as make updates where needed. In doing so, **GitHub** promotes collaboration, transparency and helps team members to work from anywhere and stay up to date with the latest changes made to code.

In **GitHub** packages can be published publicly, allowing access to the outside world, or privately within team members. Once shared publicly, users can download the code and try it out on their personal computers.

In summary, **GitHub** is an [online code hosting platform](#) that provides a cool UI and allows you to host remote **Git** repositories and work collaboratively with other developers from anywhere.

On the other hand, **Git** is what handles the version control operations like pushing and merging code, making commits, cloning repositories, etc. Now let us explore various **Git** commands that you can use to work with **GitHub**.

## How to Install Git on Linux

---

Before you get started with using **Git**, let's first install it. If you are running [Ubuntu or Debian-based distribution](#), run the following [apt commands](#) to install it.

```
$ sudo apt update
$ sudo apt install git -y
```

For [Red Hat-based distributions](#) such as RHEL, Fedora, CentOS Stream, AlmaLinux, and Rocky Linux execute the following [dnf commands](#):

```
$ sudo dnf update  
$ sudo dnf install git -y
```

To confirm that **Git** is installed run the command:

```
$ git --version  
  
git version 2.34.1
```

## Initializing a Git Repository

---

The initial step in using **git** is to initialize a **git** repository, which is essentially converting a directory on your local system into a git repository where all your project files and code will be stored before being pushed to **GitHub**.

To demonstrate this, we will [create a directory](#) called **test\_repo** and navigate into it.

```
$ mkdir test_repo  
$ cd test_repo
```

To initialize the repository run the following command:

```
$ git init
```

You will get a bunch of hint messages and at the end of the output, you will see a notification that the **Git** repository has been initialized on a branch which, by default, is called **master**.

```
tecmint@ubuntu:~$  
tecmint@ubuntu:~$ mkdir test_repo  
tecmint@ubuntu:~$  
tecmint@ubuntu:~$  
tecmint@ubuntu:~$ cd test_repo/  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ git init  
hint: Using 'master' as the name for the initial branch. This default branch name  
hint: is subject to change. To configure the initial branch name to use in all  
hint: of your new repositories, which will suppress this warning, call:  
hint:  
hint:   git config --global init.defaultBranch <name>  
hint:  
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and  
hint: 'development'. The just-created branch can be renamed via this command:  
hint:  
hint:   git branch -m <name>  
Initialized empty Git repository in /home/tecmint/test_repo/.git/  
tecmint@ubuntu:~/test_repo$
```

Initialize Git Repository

## Commit Changes to Git Repository

Once a directory has been initialized, a hidden directory called `.git` is created, which contains all the files Git requires to track changes made to code. To view this directory, run the [ls command](#) with the `-la` options as shown.

```
$ ls -la
```

```
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ ls -la  
total 12  
drwxrwxr-x  3 tecmint tecmint 4096 May 26 07:58 .  
drwxr-x--- 16 tecmint tecmint 4096 May 26 07:57 ..  
drwxrwxr-x  7 tecmint tecmint 4096 May 26 07:58 .git  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$
```

View Git Directory Files

Tracking a project only happens once changes are committed for the first time. We will come to this later.

## Configuring Git Repository

Once we have initialized the **Git** repository, the next step is to set a few configuration variables for **Git**. We will add a username and email address which will be associated with the commits.

To add a username and email address to all the commits by the current user, run the following commands. Be sure to provide your preferred name and email address.

```
$ git config --global user.name "your_name"
$ git config --global user.email "email_address"
```

To confirm the variables set, run the command:

```
$ git config --list
```

```
tecmint@ubuntu:~/test_repo$
tecmint@ubuntu:~/test_repo$
tecmint@ubuntu:~/test_repo$ git config --list
user.name=Tecmint
user.email=admin@tecmint.com
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
tecmint@ubuntu:~/test_repo$
tecmint@ubuntu:~/test_repo$
```

List Git Config Settings

## Add File to a Git Repository

To make changes to the git repository, first, run the `git add` command, followed by `git commit`. The git add command adds files to the staging area. This is essentially a special area where we propose the changes for the next commit.

In our setup, we have a Python file called **app.py**. To add this file to the staging area, we will run the following command.

```
$ git add app.py
```

If you have multiple project files, you can add all of them at a go as shown.


```
$ git add -A  
OR  
$ git add --all
```

Once you have added all the files to the staging area, commit them to the repository using the git commit command. A commit is like taking a snapshot of your project.

```
$ git commit -m "Some Message"
```

The `-m` option allows you to attach a tag or message to your commit. The string between quotation marks is the message associated with the commit. This message is essential for having a meaningful history to refer to.

The following output shows that we have successfully committed the changes to the Git repository.

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git add app.py  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git commit -m "Initial Commit"   
[master (root-commit) ebe2573] Initial Commit  
1 file changed, 1 insertion(+)  
create mode 100644 app.py  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$
```

Confirm Changes to Git Repository

To check the status of commits, run the command:

```
$ git status
```

If there are no pending commits, you will get the notification that there's nothing to

Privacy

```
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ git status ←  
On branch master  
nothing to commit, working tree clean  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$
```

Check the Status of Git Commits

Now, let us create two more files and check the status of the commits.

```
$ echo "Hello there" > file1.txt  
$ echo "Let us learn Git" > file2.txt
```

Next, display the state of the staging area.

```
$ git status
```

This time around, the newly added files will appear in red and will be marked as 'untracked'. In order for Git to track them, you will need to add them to the staging area.

```
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ echo "Hello there" > file1.txt  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ echo "Let us learn Git" > file2.txt  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ git status  
On branch master  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
    file1.txt  
    file2.txt ←  
  
nothing added to commit but untracked files present (use "git add" to track)  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$
```

Check the State of the Git Repository

To add all the files at a go, run the command:



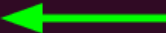
```
$ git add .
```

Note that this adds the entire directory recursively and caution should be exercised. However, since we only have two files, we will use it anyway.

Next, commit the changes made.

```
$ git commit -m "Second Commit"
```

```
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ git add .  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ git commit -m "Second Commit"  
[master 6f5c399] Second Commit  
2 files changed, 2 insertions(+)  
create mode 100644 file1.txt  
create mode 100644 file2.txt  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$
```



Commit Changes to Git Repository

Now, let's try something more ambitious. We will modify one file.

```
$ echo "Git is a popular Version Control System" >> file2.txt
```

Once again, the file appears in red since the changes made have not been committed.

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ echo "Git is a popular Version Control System" >> file2.txt  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git status  
On branch master  
Changes not staged for commit:  
  (use "git add <file>..." to update what will be committed)  
  (use "git restore <file>..." to discard changes in working directory)  
        modified:   file2.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$
```

Verify Changes to Git Repository

Once again, you need to add the modified file to the staging area.

```
$ git add file2.txt  
$ git status
```

When you check the status of the staging area, the file appears in green and is ready to be committed.

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git add file2.txt  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git status  
On branch master  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)  
        modified:   file2.txt  
  
tecmin@ubuntu:~/test_repo$
```

Check the Status of the Git Staging Area

## Remove File from Git Repository

Suppose we realize that we no longer need **file2.txt** in our project since it has unused code.

You can remove it using the [rm command](#) as shown.

```
$ rm file2.txt
```

When you check the status, you see that the deleted file is marked in red and that the change has not been committed yet.

```
$ git status
```

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ rm file2.txt  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git status  
On branch master  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)  
        modified:   file2.txt  
  
Changes not staged for commit:  
  (use "git add/rm <file>..." to update what will be committed)  
  (use "git restore <file>..." to discard changes in working directory)  
        deleted:    file2.txt  
tecmin@ubuntu:~/test_repo$
```

Confirm File Deletion in Git

We have removed **file2.txt** from our working directory. However, it still exists in the staging area. To confirm this, run the command:

```
$ git ls-files
```

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git ls-files  
app.py  
file1.txt  
file2.txt  
tecmin@ubuntu:~/test_repo$
```

Check the File in Git Staging Area

To stage the deletion, run the **git add** command as shown

```
$ git add file2.txt
```

Once again, list the files in the staging area.

```
$ git ls-files
```

This time, you see that **file2.txt** no longer exists in the staging area. When you check the status, you can see that the change is ready to be committed.

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git add file2.txt  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git ls-files ←  
app.py  
file1.txt  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git status ←  
On branch master  
Changes to be committed:  
  (use "git restore --staged <file>..." to unstage)  
    deleted:   file2.txt ←
```

Confirm the Status of the Git Staging Area

So now let us commit to the change made.

```
$ git commit -m "Remove unused code files"
```

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git commit -m "Remove unused code files"  
[master 0975f31] Remove unused code files  
1 file changed, 1 deletion(-)  
delete mode 100644 file2.txt  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$
```

Commit Changes to Git

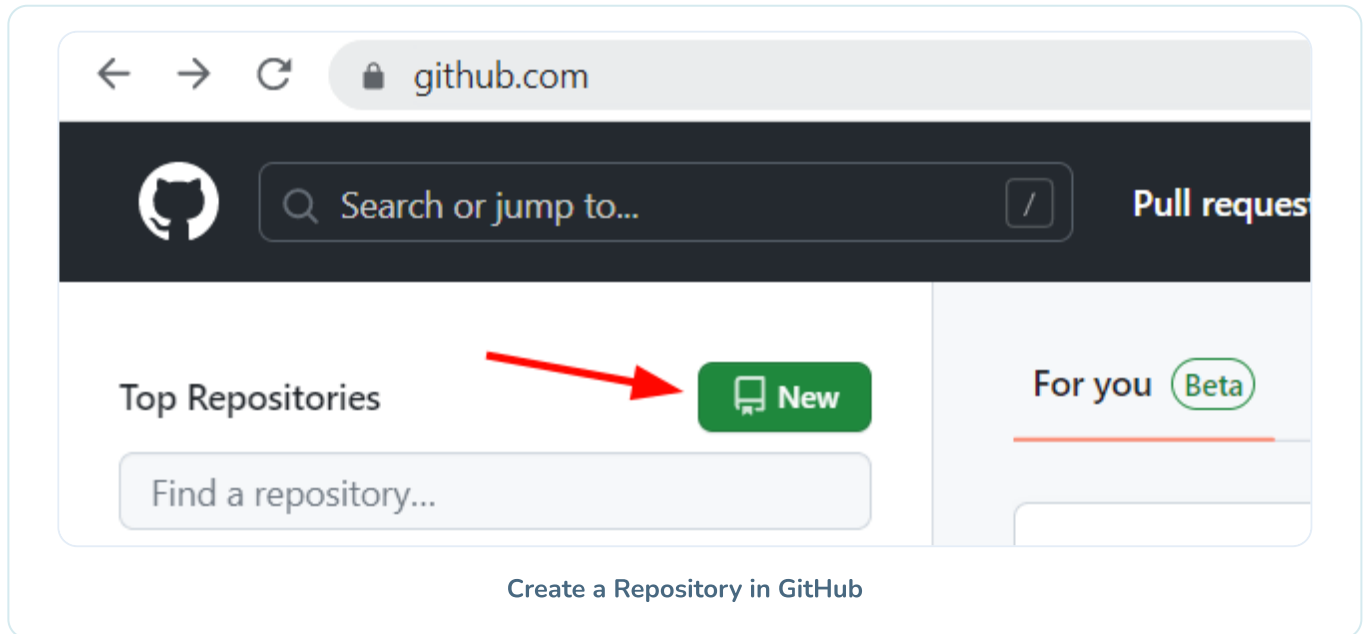
## Pushing Local Repository to GitHub

Privacy

With the local repository ready, you can now opt to push your project to GitHub for collaboration and for other users to access your project.

But first things first. You need to [create a GitHub account](#) if you don't have one already.

Next, you need to create a new empty repository. This is the repository that you will push your local repository to. So, click the '**New**' button.



Fill in the repository name and set it to '**Public**'. You can either opt to initialize a README file or not. This is a file that briefly describes what your code is all about. This doesn't really matter for now.

## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner \*



Repository name \*

my-test-project

✓ my-test-project is available.

Great repository names are short and memorable. Need inspiration? How about [animated-waddle?](#)

Description (optional)



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:



Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None

### GitHub Repository Details

Then click the **'Create repository'** button to create the repository with the details provided. In this example, we have created a repository called **'my-test-project'**.

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: None

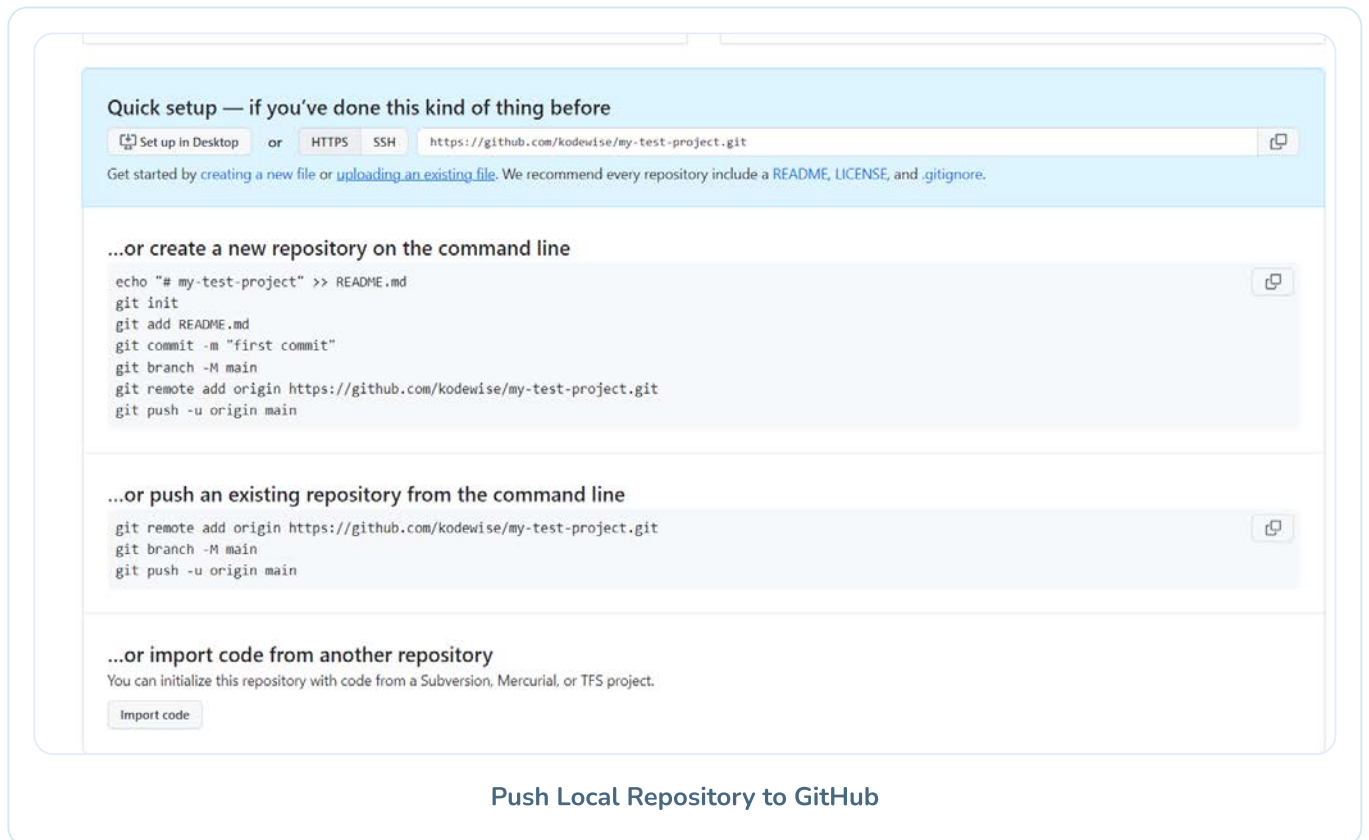
A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

ⓘ You are creating a public repository in your personal account.

Create repository

### Create Github Repository

A quick setup will be provided which includes a link to your newly created repository and instructions on how you can push your local repository to GitHub.



The screenshot shows the GitHub 'Quick setup' page for a new repository. At the top, it says 'Quick setup — if you've done this kind of thing before'. Below this, there are three tabs: 'Set up in Desktop', 'HTTPS', and 'SSH'. The 'SSH' tab is selected, and the URL 'https://github.com/kodewise/my-test-project.git' is displayed in a text box. Below the tabs, it says 'Get started by creating a new file or [uploading an existing file](#). We recommend every repository include a README, LICENSE, and .gitignore.' There are three main sections for creating or pushing a repository:

- ...or create a new repository on the command line**: This section contains a code block with the following commands:

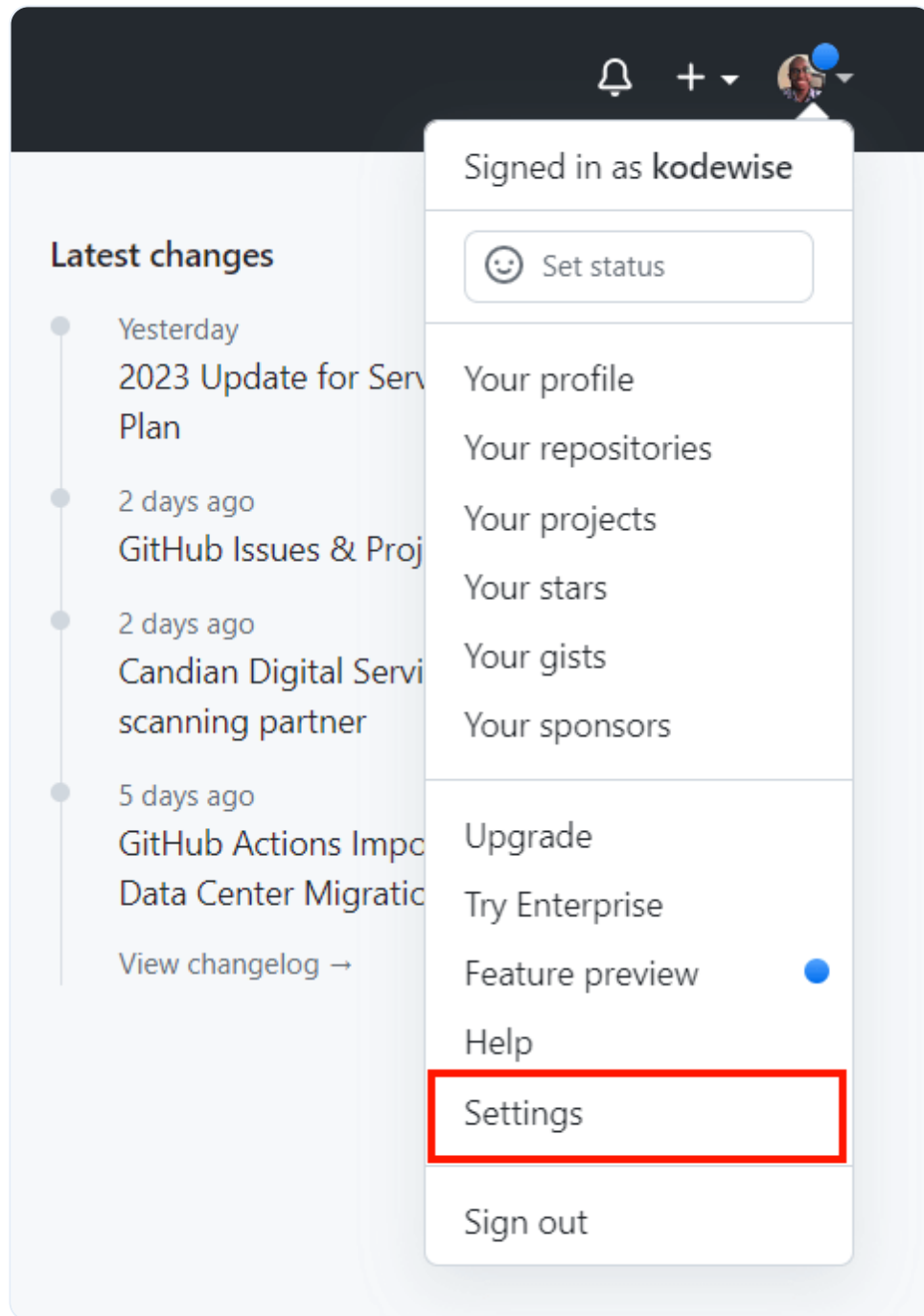
```
echo "# my-test-project" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/kodewise/my-test-project.git
git push -u origin main
```
- ...or push an existing repository from the command line**: This section contains a code block with the following commands:

```
git remote add origin https://github.com/kodewise/my-test-project.git
git branch -M main
git push -u origin main
```
- ...or import code from another repository**: This section says 'You can initialize this repository with code from a Subversion, Mercurial, or TFS project.' and has an 'Import code' button.

At the bottom of the page, there is a button labeled 'Push Local Repository to GitHub'.

The next step is to enable authentication to **GitHub** on the [command line using SSH keys](#).

To do so, click on your profile icon at the right-hand corner, and select '**Settings**'.



GitHub Settings

Next, navigate to the '**SSH and GPG**' keys, then click '**New SSH Key**'.



The screenshot shows the GitHub 'SSH and GPG keys' settings page. On the left sidebar, the 'SSH and GPG keys' option is highlighted with a red box and a red circle with the number '1'. In the main content area, the 'SSH keys' section has a 'New SSH key' button highlighted with a red box and a red circle with the number '2'. Below this, the 'GPG keys' section has a 'New GPG key' button. The 'Vigilant mode' section is also visible.

### GitHub SSH and GPG Keys

Next, copy the public SSH key and provide a meaningful title to your key. Then press 'Add SSH Key'.

The screenshot shows the 'SSH keys / Add new' page. The 'Title' field is filled with 'My Public SSH Key'. The 'Key type' is set to 'Authentication Key'. The 'Key' field contains a public SSH key. The 'Add SSH key' button is highlighted with a red box and a red arrow points to it.

### GitHub Add SSH Key

All is set now to push your repository to GitHub. Add the remote GitHub repository using the `git remote add` command syntax.

```
$ git remote add origin [REMOTE REPOSITORY URL HERE]
```

In the above syntax, '**remote**' stands for the remote repository version while '**origin**' is the name given to a remote server.

In our case, the command will appear as shown.

```
$ git remote add origin https://github.com/kodewise/my-test-project.
```

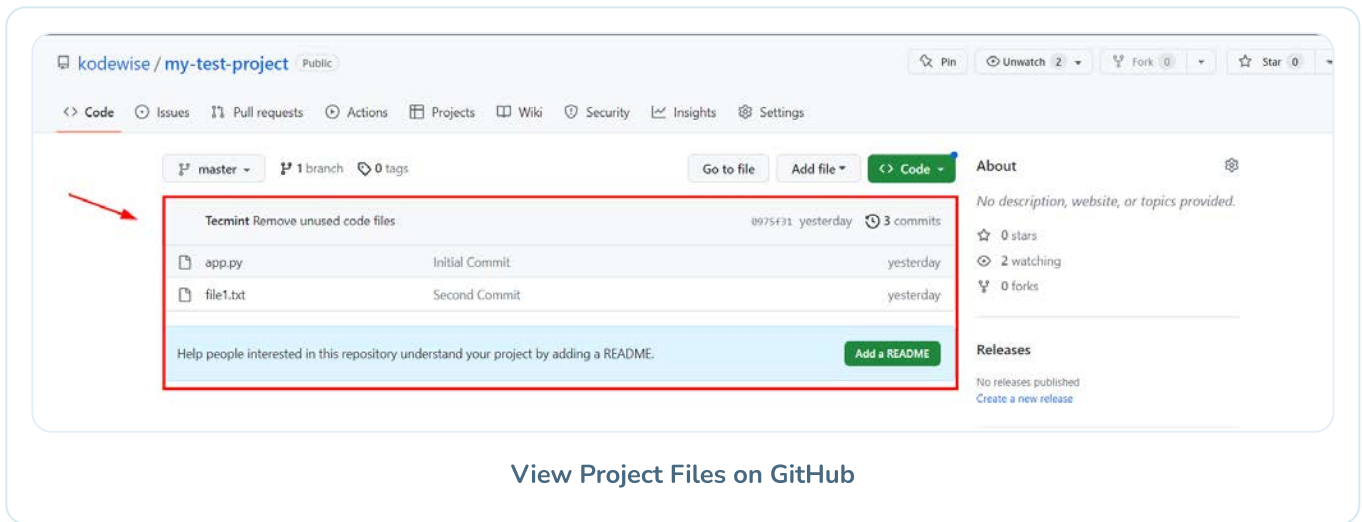
Thereafter, run the **git push** command to push the local repository to GitHub. Here, the master is the default branch name.

```
$ git push -u origin master
```

```
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ git remote add origin git@github.com:kodewise/my-test-project.git  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$ git push -u origin master  
Enumerating objects: 9, done.  
Counting objects: 100% (9/9), done.  
Delta compression using up to 2 threads  
Compressing objects: 100% (5/5), done.  
Writing objects: 100% (9/9), 697 bytes | 116.00 KiB/s, done.  
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0  
remote: Resolving deltas: 100% (1/1), done.  
To github.com:kodewise/my-test-project.git  
* [new branch]      master -> master  
Branch 'master' set up to track remote branch 'master' from 'origin'.  
tecmint@ubuntu:~/test_repo$  
tecmint@ubuntu:~/test_repo$
```

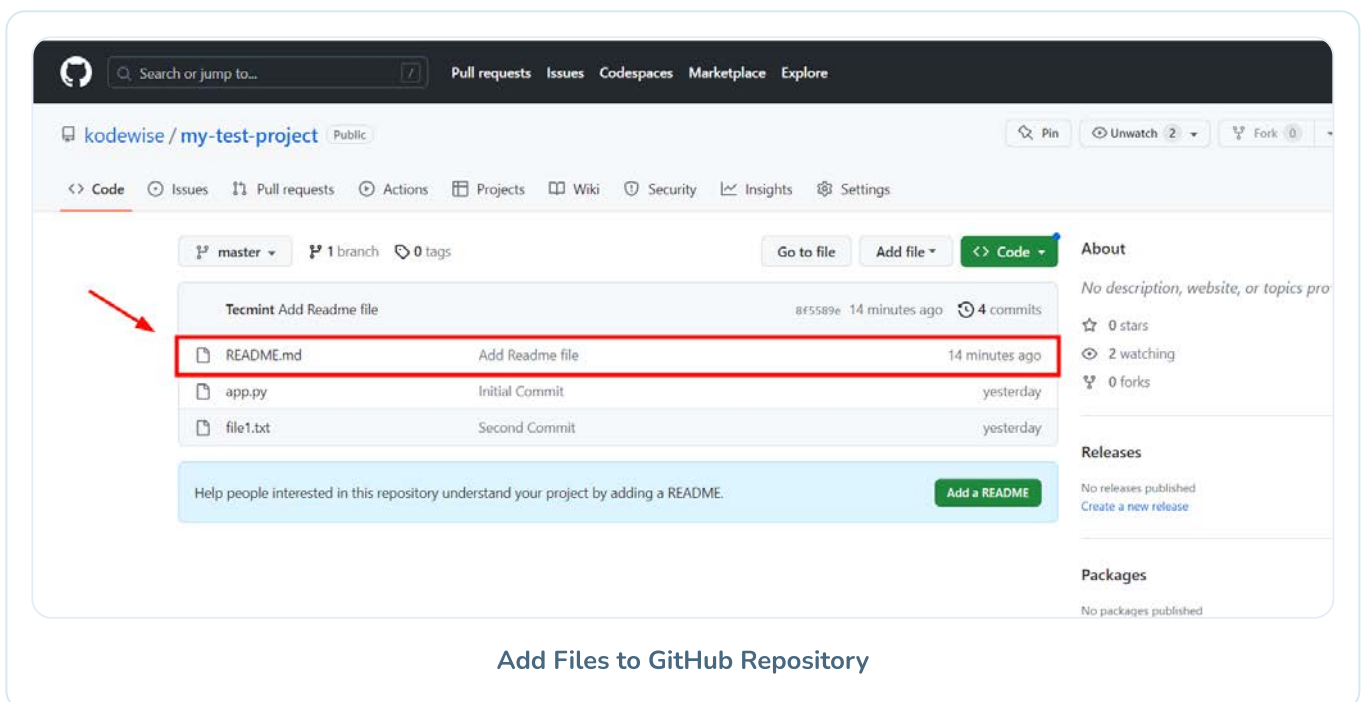
Push Git Local Repository to GitHub

When you visit your repository on Github, you will see all your project files.



Going forward, if you want to add a new file to your repository, just repeat the same steps of adding the file to the staging area, committing it, and pushing the changes to the repository.

```
$ git add -A
$ git commit -m 'Added my project'
$ git push -u origin main
```



## Update Local Repository with Changes Made on GitHub

The `git pull` command is used to update the local repository with upstream changes on the remote repository. It essentially downloads the changes made to code on **GitHub** and

updates the local repository on your system accordingly. It combines `git fetch` and `git merge` commands.

In this example, we have edited the **README** file on the remote repository hosted on **GitHub**. To ensure that we have the same copy on our local repository, we will run the command:

```
$ git pull https://github.com/kodewise/my-test-project.git
```

```
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$ git pull https://github.com/kodewise/my-test-project.git  
remote: Enumerating objects: 5, done.  
remote: Counting objects: 100% (5/5), done.  
remote: Compressing objects: 100% (3/3), done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), 742 bytes | 247.00 KiB/s, done.  
From https://github.com/kodewise/my-test-project  
* branch      HEAD      -> FETCH_HEAD  
Updating f86d29d..3934735  
Fast-forward  
 README.md | 1 +  
 1 file changed, 1 insertion(+)  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$  
tecmin@ubuntu:~/test_repo$
```

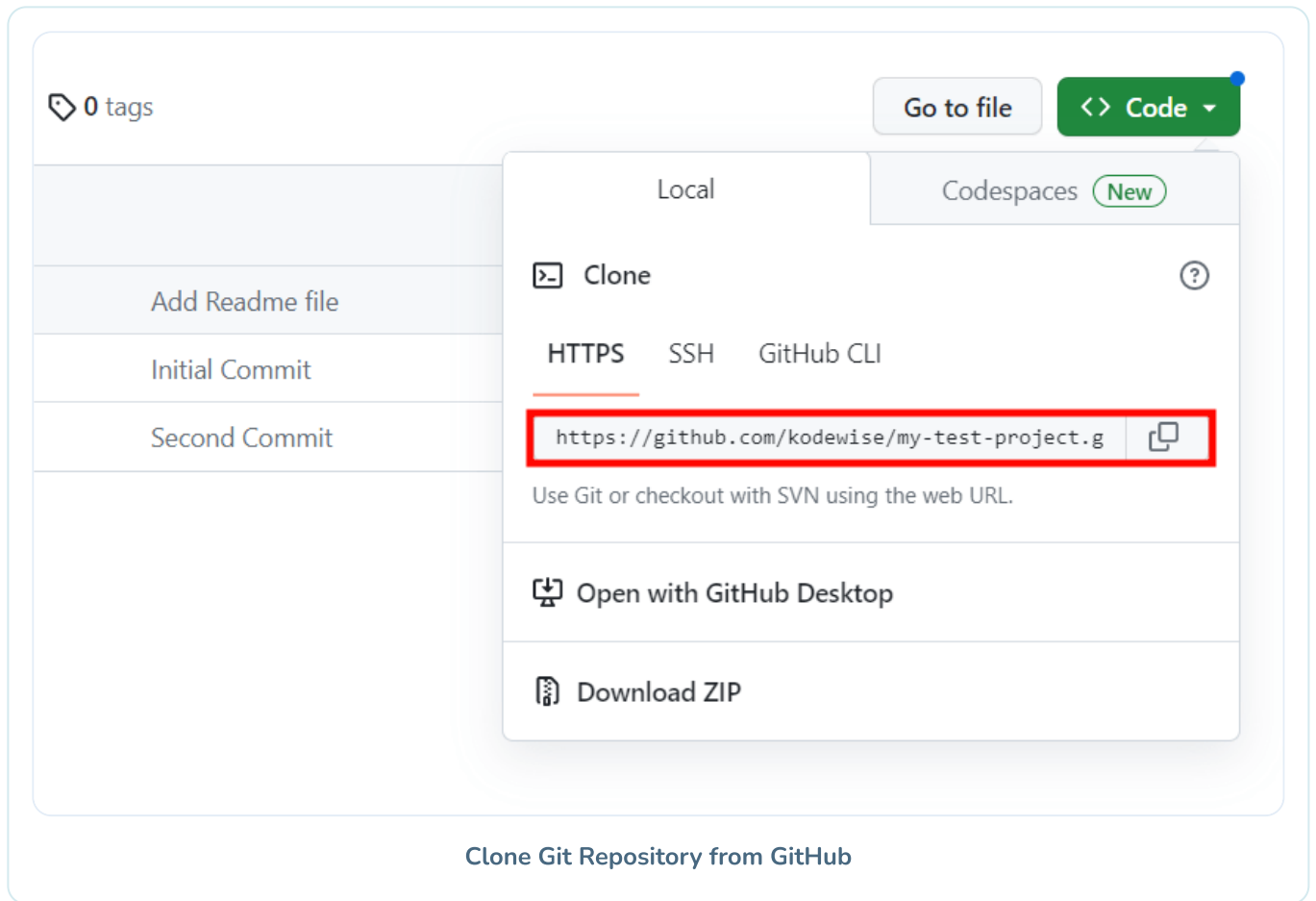
Update Local Git Repository with GitHub

## Cloning a Git Repository

The `git clone` command downloads an exact copy of the project files on a GitHub repository. You can clone a public repository on any system or directory using the following syntax.

```
$ git clone [REMOTE REPOSITORY URL HERE]
```

On the GitHub project's page, click '**Code**' and copy the URL as highlighted below.



Then clone the **Github** repository as shown.

```
$ git clone https://github.com/kodewise/my-test-project.git
```

This downloads the entire repository along with the project files.

```
tecmin@ubuntu:~$  
tecmin@ubuntu:~$ git clone https://github.com/kodewise/my-test-project.git  
Cloning into 'my-test-project'...  
remote: Enumerating objects: 12, done.  
remote: Counting objects: 100% (12/12), done.  
remote: Compressing objects: 100% (6/6), done.  
remote: Total 12 (delta 1), reused 12 (delta 1), pack-reused 0  
Receiving objects: 100% (12/12), done.  
Resolving deltas: 100% (1/1), done.  
tecmin@ubuntu:~$  
tecmin@ubuntu:~$  
tecmin@ubuntu:~$ ls  
Desktop Downloads my-test-project Public Templates  
Documents Music Pictures snap Videos  
tecmin@ubuntu:~$  
tecmin@ubuntu:~$  
tecmin@ubuntu:~$ ls -l my-test-project/  
total 8  
-rw-rw-r-- 1 tecmin tecmin 22 May 28 17:23 app.py  
-rw-rw-r-- 1 tecmin tecmin 12 May 28 17:23 file1.txt  
-rw-rw-r-- 1 tecmin tecmin 0 May 28 17:23 README.md  
tecmin@ubuntu:~$  
tecmin@ubuntu:~$
```

Download GitHub Repository

### You might also like:

- [How to Use Git in Linux \[Comprehensive Guide\]](#)
- [10 Best GitHub Alternatives to Host Projects](#)
- [How to Work With GitHub Flavored Markdown in Linux](#)
- [11 Best Git Clients and Git Repository Viewers for Linux](#)

## Conclusion

In this **Git** tutorial, we have introduced **git** and demonstrated basic commands used to create and manage files in a Git repository. Lastly, we walked you through how to push your repository to **GitHub** for collaboration. That's it for now. We hope that you found this tutorial useful.

If you read this far, tweet to the author to show them you care. [Tweet a thanks](#)