Git and GitHub Introduction.

It 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

### **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.

## **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.

Git Version-

git –-version

## **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 "indra kumar "

git config --global user.email “[indrakumarbanshkar.jd@gmail.com](mailto:indrakumarbanshkar.jd@gmail.com)”

**Note:** Use global to set the username and e-mail for **every repository** on your computer.

If you want to set the username/e-mail for just the current repo, you can remove global

## **Creating Git Folder**

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

mkdir myproject

cd myproject

mkdir **make**s a **new directory**.

cd **changes** the **current working directory**.

## **Initialize Git**

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

### **Example**

git init

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

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.

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

### **Example**

git add index.html

## **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 tutoial on https://www.w3schools.com  
  
This repository is built step by step in the 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**

<!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>  
  
</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

## **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

**Note:** Short status flags are:

* ?? - Untracked files
* A - Files added to stage
* M - Modified files
* D - Deleted files

We see the file we expected is modified. So let's commit it directly:

### **Example**

git commit -a -m "Updated index.html with a new line"

[master 09f4acd] Updated index.html with a new line

1 file changed, 1 insertion(+)

**Warning:** Skipping the Staging Environment is not generally recommended.

Skipping the stage step can sometimes make you include unwanted changes.

## **Git Help**

If you are having trouble remembering commands or options for commands, you can use Git help.

There are a couple of different ways you can use the help command in command line:

* git *command* -help -  See all the available options for the specific command
* git help --all -  See all possible commands

## **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.

**Note:** Using the -b option on checkout will create a new branch, and move to it, if it does not exist

ADVERTISEMENT

## **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.

See how easy it is to work with branches? And how this allows you to work on different things?

## **Emergency Branch**

Now imagine that we are not yet done with hello-world-images, but we need to fix an error on master.

I don't want to mess with master directly, and I do not want to mess with hello-world-images, since it is not done yet.

So we create a new branch to deal with the emergency:

### **Example**

git checkout -b emergency-fix

Switched to a new branch 'emergency-fix'

Now we have created a new branch from master, and changed to it. We can safely fix the error without disturbing the other branches.

Let's fix our imaginary error:

### **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>This line is here to show how merging works.</p>  
  
</body>  
</html>

We have made changes in this file, and we need to get those changes to the master branch.

Check the status:

### **Example**

git status

On branch emergency-fix

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

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

stage the file, and commit:

### **Example**

git add index.html

git commit -m "updated index.html with emergency fix"

[emergency-fix dfa79db] updated index.html with emergency fix

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

Now we have a fix ready for master, and we need to merge the two branches.

# Git Branch Merge

[❮ Previous](https://www.w3schools.com/git/git_branch.asp?remote=github)[Next ❯](https://www.w3schools.com/git/git_remote_getstarted.asp?remote=github)

## **Change Platform:**

[[](https://www.w3schools.com/git/git_branch_merge.asp?remote=github)GitHub](https://www.w3schools.com/git/git_branch_merge.asp?remote=github)[[](https://www.w3schools.com/git/git_branch_merge.asp?remote=bitbucket)Bitbucket](https://www.w3schools.com/git/git_branch_merge.asp?remote=bitbucket)[GitLab](https://www.w3schools.com/git/git_branch_merge.asp?remote=gitlab)

## **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).

Now you have a better understanding of how branches and merging works. Time to start working with a remote repository!

# Git GitHub Flow

[❮ Previous](https://www.w3schools.com/git/git_branch_push_to_remote.asp?remote=github)[Next ❯](https://www.w3schools.com/git/git_remote_pages.asp?remote=github)

## **Change Platform:**

[[](https://www.w3schools.com/git/git_github_flow.asp?remote=github)GitHub](https://www.w3schools.com/git/git_github_flow.asp?remote=github)[[](https://www.w3schools.com/git/git_github_flow.asp?remote=bitbucket)Bitbucket](https://www.w3schools.com/git/git_github_flow.asp?remote=bitbucket)[[](https://www.w3schools.com/git/git_github_flow.asp?remote=gitlab)GitLab](https://www.w3schools.com/git/git_github_flow.asp?remote=gitlab)

## **Working using the GitHub Flow**

On this page, you will learn how to get the best out of working with GitHub.

The GitHub flow is a workflow designed to work well with Git and GitHub.

It focuses on branching and makes it possible for teams to experiment freely, and make deployments regularly.

The GitHub flow works like this:

* Create a new Branch
* Make changes and add Commits
* Open a Pull Request
* Review
* Deploy
* Merge

You should already have a good understanding of how this works from the previous chapters. This chapter focuses on understanding how the flow makes it easy for you to work together.

## **Create a New Branch**

Branching is the key concept in Git. And it works around the rule that the master branch is ALWAYS deployable.

That means, if you want to try something new or experiment, you create a new branch! Branching gives you an environment where you can make changes without affecting the main branch.

When your new branch is ready, it can be reviewed, discussed, and merged with the main branch when ready.

When you make a new branch, you will (almost always) want to make it from the master branch.

**Note:** Keep in mind that you are working with others. Using descriptive names for new branches, so everyone can understand what is happening.

## **Make Changes and Add Commits**

After the new branch is created, it is time to get to work. Make changes by adding, editing and deleting files. Whenever you reach a small milestone, add the changes to your branch by commit.

Adding commits keeps track of your work. Each commit should have a message explaining what has changed and why. Each commit becomes a part of the history of the branch, and a point you can revert back to if you need to.

**Note:** commit messages are very important! Let everyone know what has changed and why. Messages and comments make it so much easier for yourself and other people to keep track of changes.

ADVERTISEMENT

## **Open a Pull Request**

Pull requests are a key part of GitHub. A Pull Request notifies people you have changes ready for them to consider or review.

 You can ask others to review your changes or pull your contribution and merge it into their branch.

## **Review**

When a Pull Request is made, it can be reviewed by whoever has the proper access to the branch. This is where good discussions and review of the changes happen.

Pull Requests are designed to allow people to work together easily and produce better results together!

If you receive feedback and continue to improve your changes, you can push your changes with new commits, making further reviews possible.

**Note:** GitHub shows new commit and feedback in the "unified Pull Request view".

## **Deploy**

When the pull request has been reviewed and everything looks good, it is time for the final testing. GitHub allows you to deploy from a branch for final testing in production before merging with the master branch.

If any issues arise, you can undo the changes by deploying the master branch into production again!

**Note:** Teams often have dedicated testing environments used for deploying branches.

## **Merge**

After exhaustive testing, you can merge the code into the master branch!

Pull Requests keep records of changes to your code, and if you commented and named changes well, you can go back and understand why changes and decisions were made.

**Note:** You can add keywords to your pull request for easier searching!