Git is a version control system.

Git helps you keep track of code changes.

Git is used to collaborate on code.

```
git –version
```

```
For new users, using the terminal view can seem a bit complicated.
Don't worry! We will keep it really simple, and learning this way gives
you a good grasp of how Git works.
Git and GitHub are different things.
```

In this tutorial you will understand what Git is and how to use it on the remote repository platforms, like GitHub.

You can choose, and change, which platform to focus on by clicking in the menu on the right:

```
What is Git?
```

Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.

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

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!

Change Platform:

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.
- In this tutorial, we will focus on using Git with GitHub.

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:

Example

```
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:

Example

```
git config --global user.name "w3schools-test"


git config --global user.email "test@w3schools.com"
```

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.

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:

Example

```
mkdir myproject


cd myproject
```

`mkdir` **make**s a **new directory**.

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

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

Note: If you already have a folder/directory you would like to use for Git:

Navigate to it in command line, or open it in your file explorer, right-click and select "Git Bash here"

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/
```

You just created your first Git Repository!

Note: Git now knows that it should watch the folder you initiated it on.

Git creates a hidden folder to keep track of changes.

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.

If you want to learn how to create a new file using a text editor, you can visit our HTML tutorial:
HTML Editors

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

Example

```
<!DOCTYPE html>
<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:

Example

```
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:

Example

```
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.

We will cover the staging environment in the next chapter.

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:

Example

```
git add index.html
```

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

Example

```
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 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
```

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
```

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 Commit Log

To view the history of commits for a repository, you can use the `log` command:

Example

```
git log

commit 09f4acd3f8836b7f6fc44ad9e012f82faf861803 (HEAD -> master)

Author: w3schools-test
```

```
Date:   Fri Mar 26 09:35:54 2021 +0100

    Updated index.html with a new line

commit 221ec6e10aeedbfd02b85264087cd9adc18e4b26
Author: w3schools-test
Date:   Fri Mar 26 09:13:07 2021 +0100

    First release of Hello World!
```

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

Let's go over the different commands.

Git -help See Options for a Specific Command

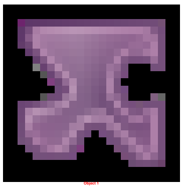
Any time you need some help remembering the specific option for a command, you can use `git command -help`:

Example

```
git commit -help

usage: git commit [] [--] ...



    -q, --quiet           suppress summary after successful commit

    -v, --verbose         show diff in commit message template



Commit message options

    -F, --file      read message from file

    --author      override author for commit

    --date          override date for commit

    -m, --message
```

```
                       commit message
    -c, --reedit-message
                       reuse and edit message from specified commit
    -C, --reuse-message
                       reuse message from specified commit
    --fixup       use autosquash formatted message to fixup
specified commit
    --squash      use autosquash formatted message to squash
specified commit
    --reset-author        the commit is authored by me now (used
with -C/-c/--amend)
    -s, --signoff         add a Signed-off-by trailer
    -t, --template
                       use specified template file
    -e, --edit            force edit of commit
    --cleanup       how to strip spaces and #comments from message
    --status              include status in commit message template
    -S, --gpg-sign[=]
                       GPG sign commit


Commit contents options
    -a, --all             commit all changed files
    -i, --include         add specified files to index for commit
    --interactive         interactively add files
    -p, --patch           interactively add changes
    -o, --only            commit only specified files
    -n, --no-verify       bypass pre-commit and commit-msg hooks
    --dry-run             show what would be committed
    --short               show status concisely
    --branch              show branch information
    --ahead-behind        compute full ahead/behind values
    --porcelain           machine-readable output
    --long                show status in long format (default)
    -z, --null            terminate entries with NUL
    --amend               amend previous commit
    --no-post-rewrite     bypass post-rewrite hook
    -u, --untracked-files[=]
                        show untracked files, optional modes: all,
normal, no. (Default: all)
    --pathspec-from-file
                        read pathspec from file
    --pathspec-file-nul   with --pathspec-from-file, pathspec
elements are separated with NUL character
```

Note: You can also use `--help` instead of `-help` to open the relevant Git manual page



Git help --all See All Possible Commands

To list all possible commands, use the `help --all` command:

Warning: This will display a very long list of commands

Example

```
$ git help --all

See 'git help ' to read about a specific subcommand



Main Porcelain Commands

   add                         Add file contents to the index

   am                          Apply a series of patches from a mailbox
```

```
█   archive             Create an archive of files from a named tree

█   bisect              Use binary search to find the commit that
█introduced a bug

█   branch              List, create, or delete branches

█   bundle              Move objects and refs by archive

█   checkout            Switch branches or restore working tree
█files

█   cherry-pick         Apply the changes introduced by some
█existing commits

█   citool              Graphical alternative to git-commit

█   clean               Remove untracked files from the working tree

█   clone               Clone a repository into a new directory

█   commit              Record changes to the repository

█   describe            Give an object a human readable name based
█on an available ref

█   diff                Show changes between commits, commit and
█working tree, etc
```

```
   fetch                Download objects and refs from another
repository

   format-patch         Prepare patches for e-mail submission

   gc                   Cleanup unnecessary files and optimize the
local repository

   gitk                 The Git repository browser

   grep                 Print lines matching a pattern

   gui                  A portable graphical interface to Git

   init                 Create an empty Git repository or
reinitialize an existing one

   log                  Show commit logs

   maintenance          Run tasks to optimize Git repository data

   merge                Join two or more development histories
together

   mv                   Move or rename a file, a directory, or a
symlink

   notes                Add or inspect object notes
   pull                    Fetch from and integrate with another
repository or a local branch

   push                    Update remote refs along with associated
objects

   range-diff              Compare two commit ranges (e.g. two versions
of a branch)

   rebase                  Reapply commits on top of another base tip

   reset                   Reset current HEAD to the specified state

   restore                 Restore working tree files

   revert                  Revert some existing commits

   rm                      Remove files from the working tree and from
the index

   shortlog                Summarize 'git log' output

   show                    Show various types of objects

   sparse-checkout         Initialize and modify the sparse-checkout

   stash                   Stash the changes in a dirty working
directory away
```

```
   status                Show the working tree status

   submodule             Initialize, update or inspect submodules

   switch                Switch branches

   tag                   Create, list, delete or verify a tag object
signed with GPG

   worktree              Manage multiple working trees



Ancillary Commands / Manipulators

   config                Get and set repository or global options

   fast-export           Git data exporter

   fast-import           Backend for fast Git data importers

   filter-branch         Rewrite branches

   mergetool             Run merge conflict resolution tools to
resolve merge conflicts
```

```
   pack-refs               Pack heads and tags for efficient repository
access

   prune                   Prune all unreachable objects from the
object database

   reflog                  Manage reflog information

   remote                  Manage set of tracked repositories

   repack                  Pack unpacked objects in a repository

   replace                 Create, list, delete refs to replace objects



Ancillary Commands / Interrogators

   annotate                Annotate file lines with commit information

   blame                   Show what revision and author last modified
each line of a file

   bugreport               Collect information for user to file a bug
report

   count-objects           Count unpacked number of objects and their
disk consumption
```

```
   difftool              Show changes using common diff tools

   fsck                  Verifies the connectivity and validity of
the objects in the database

   gitweb                Git web interface (web frontend to Git
repositories)

   help                  Display help information about Git

   instaweb              Instantly browse your working repository in
gitweb

   merge-tree            Show three-way merge without touching index

   rerere                Reuse recorded resolution of conflicted
merges

   show-branch           Show branches and their commits

   verify-commit         Check the GPG signature of commits

   verify-tag            Check the GPG signature of tags

   whatchanged           Show logs with difference each commit
introduces
```

```
Interacting with Others

   archimport            Import a GNU Arch repository into Git

   cvsexportcommit       Export a single commit to a CVS checkout

   cvsimport             Salvage your data out of another SCM people
love to hate

   cvsserver             A CVS server emulator for Git

   imap-send             Send a collection of patches from stdin to
an IMAP folder

   p4                    Import from and submit to Perforce
repositories

   quiltimport           Applies a quilt patchset onto the current
branch

   request-pull          Generates a summary of pending changes

   send-email            Send a collection of patches as emails

   svn                   Bidirectional operation between a Subversion
repository and Git
```

```
█Low-level Commands / Manipulators

█   apply                    Apply a patch to files and/or to the index

█   checkout-index           Copy files from the index to the working
█tree

█   commit-graph             Write and verify Git commit-graph files

█   commit-tree              Create a new commit object

█   hash-object              Compute object ID and optionally creates a
█blob from a file

█   index-pack               Build pack index file for an existing packed
█archive

█   merge-file               Run a three-way file merge

█   merge-index              Run a merge for files needing merging

█   mktag                    Creates a tag object

█   mktree                   Build a tree-object from ls-tree formatted
█text

█   multi-pack-index         Write and verify multi-pack-indexes
█   pack-objects          Create a packed archive of objects

█   prune-packed          Remove extra objects that are already in
█pack files

█   read-tree             Reads tree information into the index

█   symbolic-ref          Read, modify and delete symbolic refs

█   unpack-objects        Unpack objects from a packed archive

█   update-index          Register file contents in the working tree
█to the index

█   update-ref            Update the object name stored in a ref
█safely

█   write-tree            Create a tree object from the current index



█Low-level Commands / Interrogators

█   cat-file              Provide content or type and size information
█for repository objects

█   cherry                Find commits yet to be applied to upstream
```

```
   diff-files           Compares files in the working tree and the
index

   diff-index           Compare a tree to the working tree or index

   diff-tree            Compares the content and mode of blobs found
via two tree objects

   for-each-ref         Output information on each ref

   for-each-repo        Run a Git command on a list of repositories

   get-tar-commit-id    Extract commit ID from an archive created
using git-archive

   ls-files             Show information about files in the index
and the working tree

   ls-remote            List references in a remote repository

   ls-tree              List the contents of a tree object

   merge-base           Find as good common ancestors as possible
for a merge

   name-rev             Find symbolic names for given revs

   pack-redundant       Find redundant pack files
   rev-list              Lists commit objects in reverse
chronological order

   rev-parse             Pick out and massage parameters

   show-index            Show packed archive index

   show-ref              List references in a local repository

   unpack-file           Creates a temporary file with a blob's
contents

   var                   Show a Git logical variable

   verify-pack           Validate packed Git archive files



Low-level Commands / Syncing Repositories

   daemon                A really simple server for Git repositories

   fetch-pack            Receive missing objects from another
repository

   http-backend          Server side implementation of Git over HTTP
   send-pack            Push objects over Git protocol to another
repository

   update-server-info   Update auxiliary info file to help dumb
servers



Low-level Commands / Internal Helpers

   check-attr           Display gitattributes information

   check-ignore         Debug gitignore / exclude files

   check-mailmap        Show canonical names and email addresses of
contacts

   check-ref-format     Ensures that a reference name is well formed

   column               Display data in columns

   credential           Retrieve and store user credentials

   credential-cache     Helper to temporarily store passwords in
memory

   credential-store     Helper to store credentials on disk
   fmt-merge-msg          Produce a merge commit message

   interpret-trailers     Add or parse structured information in
commit messages

   mailinfo               Extracts patch and authorship from a single
e-mail message

   mailsplit              Simple UNIX mbox splitter program

   merge-one-file         The standard helper program to use with git-
merge-index

   patch-id               Compute unique ID for a patch

   sh-i18n                Git's i18n setup code for shell scripts

   sh-setup               Common Git shell script setup code

   stripspace             Remove unnecessary whitespace



External commands

   askyesno
```

```
credential-helper-selector

flow

lfs
```

Note: If you find yourself stuck in the list view, `SHIFT + G` to jump the end of the list, then `q` to exit the view.

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

---

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.

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 stagedto 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!