Graphical user interface, text

Description automatically generated

And check if it is all up to date.

Text

Description automatically generated

Which branches do we have now, and where are we working from?

Graphical user interface, text

Description automatically generated

Now, open your favorite editor and confirm that the changes from the GitHub branch carried over.

**Git Push Branch to GitHub**

Let's try to create a new local branch, and push that to GitHub.

Start by creating a branch.

Text

Description automatically generated

And make some changes to the README.md file. Just add a new line as below.

Text

Description automatically generated

So now we check the status of the current branch.

Text

Description automatically generated

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

Text

Description automatically generated

Now push the branch from our local Git repository, to GitHub, where everyone can see the changes:

Text

Description automatically generated

Go to GitHub and confirm that the repository has a new branch: You will see the new branch – ‘update-branch’.

Graphical user interface, application, timeline

Description automatically generated

To the right in GitHub, locate the ‘New pull request.’

Graphical user interface, text, application, chat or text message

Description automatically generated

In GitHub, we can now see the changes and merge them into the master branch if we approve it.

If you click the "Compare & pull request", you can go through the changes made and new files added:

Click on new pull request and you will see a page as below

Graphical user interface, text, application, email

Description automatically generated

Click on create new pull request. And then on merge pull request. Confirm the merge.

A pull request is how you propose changes. You can ask someone to review your changes or pull your contribution and merge it into their branch.

Since this is your own repository, you can  merge your pull request yourself:

The pull request will record the changes, which means you can go through them later to figure out the changes made.

The result should be something like this:

Graphical user interface, text, application, email

Description automatically generated

**Git GitHub Flow**

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

**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.

**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.

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.

**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!

**Git GitHub Pages**

Host Your Page on GitHub

With GitHub pages, GitHub allows you to host a webpage from your repository. Let's try to use GitHub Pages to host our repository.

This will work on **enterprise under master.**

Create a New Repository

Start by signing in to GitHub. GitHub pages need a special name and setup to work, so we start by creating a new repository:

This repository needs **a special name** to function as a GitHub page. It needs to be your **GitHub username**, followed by **.github.io:**

Graphical user interface, text, application, chat or text message

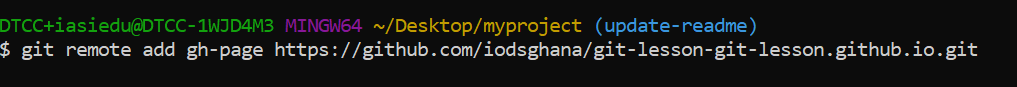
Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

Push Local Repository to GitHub Pages

We add this new repository as a remote for our local repository, we are calling it gh-page (for GitHub Pages).



Make sure you are on the master branch, then push the master branch to the new remote:

Text

Description automatically generated

**Note:** If this is the first time you are connecting to GitHub, you will get some kind of notification to authenticate this connection.

Check that the new repository has received all the files: You may have to refresh the page.

**Check Out Your Own GitHub Page**

That looks good, now click the Settings menu, and navigate to the Pages tab:

* Settings
* Pages

**Git GitHub Fork**

Add to Someone Else's Repository

At the heart of Git is collaboration. However, Git does not allow you to add code to someone else's repository without access rights.

In these next 3 chapters we will show you how to copy a repository, make changes to it, and suggest those changes be implemented to the original repository.

At the end of these chapters, you will have the opportunity to add a message to our public GitHub page:

Fork a Repository

A fork is a copy of a repository. This is useful when you want to contribute to someone else's project or start your own project based on theirs.

fork is not a command in Git, but something offered in GitHub and other repository hosts. Let's start by logging in to GitHub, and fork our repo.

<https://github.com/w3schools-test/w3schools-test.github.io>

Graphical user interface, text, application, chat or text message

Description automatically generated

**Git Clone from GitHub**

Clone a Fork from GitHub

Now we have our own fork, but only on GitHub. We also want a clone on our local Git to keep working on it.

A clone is a full copy of a repository, including all logging and versions of files.

Move back to the original repository, and click the green "Code" button to get the URL to clone:

Graphical user interface, text, application, chat or text message

Description automatically generated

<https://github.com/iodsghana/w3schools-test.github.io.git>

Open your Git bash and clone the repository:

git clone <https://github.com/w3schools-test/w3schools-test.github.io.git>

Take a look in your file system, and you will see a new directory named after the cloned project using **ls**:

**Note:** To specify a specific folder to clone to, add the name of the folder after the repository URL, like this:

git clone https://github.com/w3schools-test/w3schools-test.github.io.git *myfolder*

Navigate to the new directory, and check the status:

cd w3schools-test.github.io

git status

And check the log to confirm that we have the full repository data:

**git log**

**Configuring Remotes**

Basically, we have a full copy of a repository, whose origin we are not allowed to make changes to.

Let's see how the remotes of this Git is set up:

**git remote -v**

We see that origin is set up to the original "w3schools-test" repository, we also want to add our own fork.

First, we rename the original origin remote:

git remote rename origin upstream.

git remote -v

Then fetch the URL of our own fork:

Graphical user interface, text, application, chat or text message

Description automatically generated

And add that as origin:

git remote add origin <https://github.com/kaijim/w3schools-test.github.io.git>

git remote -v

**Note:** According to Git naming conventions, it is recommended to name your own repository origin, and the one you forked for upstream

Now we have 2 remotes:

* origin - our own fork, where we have read and write access
* upstream - the original, where we have read-only access

Now we are going to make some changes to the code. In the next chapter, we will cover how we suggest those changes to the original repository.

**Git GitHub Send Pull Request**

**Push Changes to Our GitHub Fork**

We have made a lot of changes to our local Git.

Now we push them to our GitHub fork:

commit the changes:

git push origin

Go to GitHub, and we see that the repository has a new commit. And we can send a Pull Request to the original repository:

**Create a pull request**.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, application

Description automatically generated

Approving Pull Requests

Now any member with access can see the Pull Request when they see the original repository:

Graphical user interface, diagram

Description automatically generated with medium confidence

Graphical user interface, text, application, chat or text message

Description automatically generated

**GIT ADVANCED**

**Git Ignore and .gitignore**

When sharing code, there are often parts of your project you do not want to share.

* Log files
* Temporary files
* Hidden files
* Personal files
* Etc.

Git can specify which files or parts of your project should be ignored by Git using a .**gitignore file.**

Git will not track files and folders specified in **.gitignore**. However, the .gitignore file itself **is tracked** by Git.

**Create .gitignore**

To create a .gitignore file, go to the root of your local Git, and create it:

First switch from the current branch to master

Text

Description automatically generated

Create .giyignore

Text

Description automatically generated

Now open the file using a text editor.

We are just going to add two simple rules:

* Ignore any files with the **.log** extension.
* Ignore everything in any directory named **temp.**

Example

# ignore ALL .log files  
\*.log  
  
# ignore ALL files in ANY directory named temp  
temp/

Now all .log files and anything in temp folders will be ignored by Git.

**Note:** In this case, we use a single .gitignore which applies to the entire repository.

It is also possible to have additional .gitignore files in subdirectories. These only apply to files or folders within that directory.

**Rules for .gitignore**

Here are the general rules for matching patterns in .gitignore files:

|  |  |  |
| --- | --- | --- |
| **Pattern** | **Explanation/Matches** | **Examples** |
|  | Blank lines are ignored |  |
| # *text comment* | Lines starting with # are ignored |  |
| *name* | All *name* files, *name* folders, and files and folders in any *name* folder | /name.log /name/file.txt /lib/name.log |
| *name*/ | Ending with / specifies the pattern is for a folder. Matches all files and folders in any *name* folder | /name/file.txt /name/log/name.log  **no match:** /name.log |
| *name*.*file* | All files with the *name.file* | /name.file /lib/name.file |
| */name*.*file* | Starting with / specifies the pattern matches only files in the root folder | /name.file  **no match:** /lib/name.file |
| *lib/name*.*file* | Patterns specifying files in specific folders are always relative to root (even if you do not start with / ) | /lib/name.file  **no match:** name.file /test/lib/name.file |
| \*\**/lib/name.file* | Starting with \*\* before / specifies that it matches any folder in the repository. Not just on root. | /lib/name.file /test/lib/name.file |
| \*\**/name* | All *name* folders, and files and folders in any *name* folder | /name/log.file /lib/name/log.file /name/lib/log.file |
| /lib/\*\**/name* | All *name* folders, and files and folders in any *name* folder within the lib folder. | /lib/name/log.file /lib/test/name/log.file /lib/test/ver1/name/log.file  **no match:** /name/log.file |
| \*.*file* | All files withe *.file* extention | /name.file /lib/name.file |
| \**name*/ | All folders ending with *name* | /lastname/log.file /firstname/log.file |
| *name*?.*file* | ? matches a **single** non-specific character | /names.file /name1.file  **no match:** /names1.file |
| *name*[a-z].*file* | [*range*] matches a **single** character in the specified range (in this case a character in the range of a-z, and also be numeric.) | /names.file /nameb.file  **no match:** /name1.file |
| *name*[abc].*file* | [*set*] matches a **single** character in the specified set of characters (in this case either a, b, or c) | /namea.file /nameb.file  **no match:** /names.file |
| *name*[!abc].*file* | [!*set*] matches a **single** character, **except** the ones specified in the set of characters (in this case a, b, or c) | /names.file /namex.file  **no match:** /namesb.file |
| \*.*file* | All files withe *.file* extention | /name.file /lib/name.file |
| *name*/ !*name*/secret.log | ! specifies a negation or exception. Matches all files and folders in any *name* folder, except name/secret.log | /name/file.txt /name/log/name.log  **no match:** /name/secret.log |
| \*.*file* !*name*.file | ! specifies a negation or exception. All files withe *.file* extention, except name.file | /log.file /lastname.file  **no match:** /name.file |
| \*.*file* !*name*/\**.file* junk.\* | Adding new patterns after a negation will re-ignore a previous negated file All files withe *.file* extention, except the ones in *name* folder. Unless the file name is junk | /log.file /name/log.file  **no match:** /name/junk.file |

**Local and Personal Git Ignore Rules**

It is also possible to ignore files or folders but not show it in the distributed .gitignore file.

These kinds of ignores are specified in the **.git/info/exclude**file. It works the same way as .gitignore but are not shown to anyone else.

**Git Security**

Up to this point, we have used HTTPS to connect to our remote repository.

HTTPS will usually work just fine, but you should use SSH if you work with unsecured networks. And sometimes, a project will require that you use SSH.

**What is SSH?**

SSH is a secure shell network protocol that is used for network management, remote file transfer, and remote system access.

SSH uses a pair of SSH keys to establish an authenticated and encrypted secure network protocol. It allows for secure remote communication on unsecured open networks.

SSH keys are used to initiate a secure "handshake". When generating a set of keys, you will generate a "public" and "private" key.

The "public" key is the one you share with the remote party. Think of this more as the lock.

The "private" key is the one you keep for yourself in a secure place. Think of this as the key to the lock.

SSH keys are generated through a security algorithm. It is all very complicated, but it uses prime numbers, and large random numbers to make the public and private key.

It is created so that the public key can be derived from the private key, but not the other way around.

**Generating an SSH Key Pair**

In the command line for Linux, Apple, and in the Git Bash for Windows, you can generate an SSH key.

Let's go through it, step by step.

Start by creating a new key, using your email as a label:

You will be prompted with the following through this creation:

**Enter file in which to save the key (/c/Users/user/.ssh/id\_rsa):**

Select a file location, or press "Enter" to use the default file location.

**Enter passphrase (empty for no passphrase):**

**Enter same passphrase again:**

Entering a secure passphrase will create an additional layer of security. Preventing anyone who gains access to the computer to use that key without the passphrase. However, it will require you to supply the passphrase anytime the SSH key is used.

Now we add this SSH key pair to the SSH-Agent (using the file location from above):

Text

Description automatically generated

Or

Text

Description automatically generated

**Checking for existing SSH Keys**

Before you generate an SSH key, you should check if you have already an existing SSH key. You can easily check for existing SSH keys using the Git Bash and entering the following command that lists the files in the .ssh directory

A picture containing text

Description automatically generated

Text

Description automatically generated

**Adding your SSH key to the ssh-agent**

You can start the ssh-agent in the background by typing:

|  |  |
| --- | --- |
| 1 | $ eval `ssh-agent -s` |

Text

Description automatically generated

And then you can add your private key file that you have generated by typing:

|  |  |
| --- | --- |
| 1 | $ ssh-add ~/.ssh/id\_rsa |

Text

Description automatically generated

**Adding the SSH key to your GitHub**

You have to copy the SSH public key. There are two options. One option is to use the cat command and copy it from the terminal such as:

|  |  |
| --- | --- |
| 1 | $ cat ~/.ssh/id\_rsa.pub |

Text

Description automatically generated

The other option is to copy it to your clipboard by typing:

|  |  |
| --- | --- |
| 1 | $ clip < ~/.ssh/id\_rsa.pub |

Then, go to your repository that you want to work and go the settings.

Some Notes

Text

Description automatically generated

**ALTERNATIVE**

ssh-add /Users/user/.ssh/id\_rsa

Enter passphrase for /Users/user/.ssh/id\_rsa:

Identity added: /Users/user/.ssh/id\_rsa (test@w3schools.com)

Text

Description automatically generated

Here the system could not open a connection so I must find a solution.

You might need to start ssh-agent before you run the ssh-add command:

Text

Description automatically generated

Text

Description automatically generated

Still did not work.

I want to find out if the private key is set up.

A screenshot of a computer

Description automatically generated with medium confidence

List the files available.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Add the Agent

Text

Description automatically generated

You will be prompted to supply the passphrase if you added one.

Now the SSH key pair is ready to use.

**Git GitHub Add SSH**

Copy the SSH Public Key

Now we will use the clip < command to copy the public key to our clipboard:

**clip < /Users/user/.ssh/id\_rsa.pub**

Go to GitHub, navigate to the top left corner, click your profile, and select: Settings:

Graphical user interface, application

Description automatically generated

**Using Multiple Repositories**

If you use multiple repositories on one server, you will need to generate a dedicated key pair for each one. You can’t reuse a deploy key for multiple repositories.

In the server’s SSH configuration file (usually ~/.ssh/config), add an alias entry for each repository. For example:

Host github.com-repo-0

Hostname github.com

IdentityFile=/home/user/.ssh/repo-0\_deploy\_key

Host github.com-repo-1

Hostname github.com

IdentityFile=/home/user/.ssh/repo-1\_deploy\_key

* Host github.com-repo-0 – The repository’s alias.
* Hostname github.com – Configures the hostname to use with the alias.
* IdentityFile=/home/user/.ssh/repo-0\_deploy\_key – Assigns a private key to the alias.

You can then use the hostname’s alias to interact with the repository using SSH, which will use the unique deploy key assigned to that alias. For example:

|  |  |
| --- | --- |
| 1 | $ git clone git@github.com-repo-1:OWNER/repo-1.git |

In case that you do not have a config file, you can generate a new one as explained [here](https://stackoverflow.com/questions/56287059/how-to-set-up-an-ssh-config-file-for-beginners). Also this post may be [helpful](https://stackoverflow.com/questions/49587932/i-dont-have-ssh-config-file-but-multiple-github-accounts-works-how)

**Git Revert**

revert is the command we use when we want to take a previous commit and add it as a new commit, keeping the log intact.

Step 1: Find the previous commit:

A picture containing text, clipart

Description automatically generated

Step 2: Use it to make a new commit:

Shape, circle

Description automatically generated

Let's make a new commit, where we have "accidentally" deleted a file: