2024

Git & GitHub

A Version Control System

Ahmed Magdy

Windows PowerShell

Command Rule	Example	Notes
Changing Directory		
cd /		بترجع المسار لـ **root** (مثلا C:\)
cd ..		بترجع ورا فولدر واحد
cd *folderName*	**cd** Ahmed	بتدخل ع الفولدر Ahmed
cd *"path"*	**cd** "E:\Data"	بتروح ع المسار E:\Data
Making Directory		
md *newFolderName*	**md** Ali	انشأنا فولدر اسمه Ali
md *Folder1, Folder2*	**md** Amr, Ali	انشأنا فولدرين Amr, Ali
md *"path\newFolderName"*	**md** "E:\Data\New"	انشأنا فولدر New في مسار معين
Creating New Files		
new-item *fileName.extention*	**new-item** index.html	نفس الـ **md** بس المرة دى مع فايلز مش فولدرز
new-item *file1.ext, file2.ext*	**new-item** index.html, readme.txt	
new-item *"path\newfile.ext"*	**new-item** "E:\Data\readme.txt"	
Other Commands		
ls		بيعرض الملفات والفولدرات الموجودة في المسار الحالى
dir		
move *(file/folder) (folder/"path")*	**move** Ali E:\Data	الـ **move** بتنقل بس لو المسار E:/Data ده مش موجود ، الـ **move** هتتعامل كأنها **rename** وتسمى الفولدر بالاسم E:\Data
copy *(file/folder) (folder/"path")*	**copy** Ali F:\Data	نسخ فولدر Ali للمسار F:\Data
rm *(file/folder)*	**rm** Ali, Amr, Ahmed	حذف (ممكن تحذف اكتر من حاجة)
echo *text* **>** *file*	**echo** Hello **>** text.txt	عمل ملف مكتوب فيه Hello
echo *text* **>>** *file*	**echo** World **>>** text.txt	زود على نفس الفايل كلمة World
type *file*	**type** text.txt	عرض اللى موجود ف الفايل
type *file1* **>** *file2*	**type** text.txt **>** new.txt	نسخ المحتوى لفايل جديد
get-help *command*	**get-help** copy	يقولك تفاصيل الأمر copy

Loops:

```
for ( $i = 0; $i -lt 10; $i++ ) { md $i }
```

Git & GitHub Introduction

Git is a version control system that let you manage and keep track of source code history.
Git is used for tracking code changes, tracking who made changes, coding collaboration.
GitHub is a cloud-based hosting service that makes tools that use Git and let you manage Git repositories.

Git Getting Started

In order to use Git, you need to **download and install Git** from https://git-scm.com.
Open your command shell and check if Git is properly installed **`> git --version`**
Now let Git know who you are.

```
> git config --global user.name "Your Name"
> git config --global user.email "Your Email"
```

Creating Git Folder

Navigate to your project directory in command line and run this command **`> git init`**
Git now should watch the folder you initiated it on and creates a hidden folder to keep track of changes.

Repository Status

The command **`> git status`** displays the status of our repository files.
for example, if you change or create a new file (index.html) and check the repo status, you get:

```
> git status
Untracked files:
index.html
```

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

To get Git to track them, you need to **stage them**, or **add them to the staging environment**.

Git Staging Environment

Staged files are files that are ready to be committed to the repository you are working on.
For now, we are done working with index.html. So we can add it to the Staging Environment:
`> git add index.html` (If more than a file **`git add --all`** or **`git add .`**)
Now if you check the status:

```
> git status
Changes to be committed:
new file: index.html
```

Use the **`--short`** option to see the changes in a more compact way: **`> git status --short`**
(?? - Untracked files), (A - Files added to stage), (M - Modified files), (D - Deleted files)

Git Commit

Since we have finished our work, we are ready to 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.
When we commit, we should always include a message.
> git commit -m "*First release of Hello World!*"
To view the history of commits for a repository, you can use the log command **> git log**

Git Branches

In Git, a **branch** is a new/separate version of the main repository.
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.
To create a new branch **> git branch adding-images**
Let's confirm that we have created a new branch:

```
> git branch
    adding-images
  * master
```

We can see the new branch with the name "adding-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:

```
> git checkout adding-images
    Switched to branch 'adding-images'
```

Now you are in a new version (adding-images branch) of your project, and you can work on it, stage, commit changes without impacting the main version (master branch).

Merging Branches

After completing work in this branch, let's merge the **master** and **adding-images** branches.
First, we need to change to the master branch **> git checkout master**
Now merge the current branch with adding-images: **> git merge adding-images**

As master and adding-images are essentially the same now, we can delete adding-images.
> git branch -d adding-images

When working on many branches concurrently, a conflict may occur when merging.
If a conflict occurs, fix it manually in the text editor then stage and commit the final results.

If you want to rename the current branch:
> git branch -m new-name

GitHub

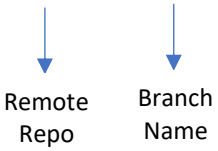
GitHub is a cloud-based hosting service that makes tools that use Git and let you manage Git repositories.
Create an account on GitHub with the same e-mail address you used in the Git config.
Create a repository on the website and copy its URL.

Add a remote repository, with the specified URL, as an origin to your local Git repo using the command:

```
> git remote add remoteRepo URL
> git remote add origin https://github.com/Username/RepoName.git
```

Now we are going to push our **master** branch to the **origin** url, and set it as the default remote branch:

```
> git push origin master
```



Push to GitHub

Let's try making some changes to our local git and pushing them to GitHub.

```
> git add .
> git commit -m "index.html updated"
> git push origin master
```

Go to GitHub and confirm that the repository has a new commit.

Pull from GitHub

Pulling to Keep up-to-date with Changes, if there are updates on the remote repo and not in your local Git.
`pull` is a combination of 2 commands: **`fetch`** and **`merge`**.
`fetch` gets all the change history of a tracked branch/repo.
`merge` combines the current branch with a specified branch.

So instead of doing:

```
> git fetch origin
> git merge origin/master
```

We just need:

```
> git pull origin
```

That is how you keep your local Git up to date from a remote repository.

GitHub Branches

If there is a new branch on the remote repo that doesn't appear in your local Git, you can use:
`> git fetch` or **`> git branch -a`**

Pull Request

A pull request is how you propose changes.
You can ask some to review your changes or pull your contribution and merge it into their branch.
If there is a branch that you accept its changes then you can merge it to another branch.

GitHub Cloning

GitHub cloning refers to the process of creating a local copy of a remote GitHub repo to your computer.

```
> git clone https://github.com/Username/RepoName.git
> git clone https://github.com/Username/RepoName.git ProjectFolder
```

In the last example, we specify a folder to clone to.

GitHub Fork

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.

After forking a repo, we also want a **clone** of your GitHub fork on your local Git to keep working on it.

```
> git clone https://github.com/Username/RepoName.git
```

Make some changes to the project then push them to your GitHub fork:

```
> git add .
> git commit -m "My Changes"
> git push origin master
```

Now go to your GitHub fork and send a **Pull Request** to the original repository.
Now any member with access can see the Pull Request when they see the original repository.
They can see the proposed changes, comment on the changes and merge.

Git Security 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**.

Git Revert

revert is the command we use when we want to **take a previous commit and add it as a new commit**.

First thing, we need to find the point we want to return to. To do that, we need to go through the log.
> git log or **> git log --oneline**

Copy the commit hash that you want to return to it then:
> git revert abcd123 -m "*Undo changes from abcd123 commit*"

In case you want the same message as in the old commit:
> git revert abcd123 --no-edit

You can use the word HEAD to revert the latest change, and then commit:
> git revert HEAD --no-edit

Git Reset

reset brings the repository back to an earlier state in the commits without making a new commit.

First thing, we need to find the point we want to return to. To do that, we need to go through the log.
> git log or **> git log --oneline**

Copy the commit hash that you want to return to it then:
> git reset abcd123

Warning:
Messing with the commit history of a repository can be dangerous if you are working with a team.

To undo changes that are not staged locally:
> git reset --hard

Warning:
reset --hard should be used with caution as it permanently deletes changes and can't be undone.

Git Amend

commit --amend is used to modify the most recent commit. It combines changes in the staging environment with the latest commit.

For example, if the last commit contained unexpected errors:
> git commit -m "*Adding plines to reddme*"

The solution using **--amend**:
> git commit --amend -m "*Added lines to README.md*"

Warning:
Messing with the commit history of a repository can be dangerous if you are working with a team.

Git .gitignore

When sharing your code with others, there are often files or parts of your project you do not want to share. For examples: 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.

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

```
> new-item .gitignore
```

Now open the file using a text editor. We are just going to add two simple rules as an example:

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

```
*.log
temp/
```

Rules for .gitignore

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 specifiing files in specific folders are always realative 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 numberic.)	/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 spesified in the set of characters (in this case a, b, or c)	/names.file /namex.file **no match:** /namesb.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