DEVOPS

SOMMAIRE

[SOMMAIRe 1](#_Toc153131286)

1. INRTODUCTION TO THE OS (operating system)

[1. Definition 2](#_Toc153131287)

[2. OS Tasks 2](#_Toc153131288)

[3. THE OS components 2](#_Toc153131289)

1. LINUX OPERATING SYSTEM

[1. linux file system 2](#_Toc153131290)

[2. INTRODUCTION TO CLI (command line interface) 3](#_Toc153131291)

[3. THE BASIC COMMANDS 3](#_Toc153131292)

[4. WORK WITH VIM EDITOR 4](#_Toc153131293)

[5. FILE OWNERSHIP/USERS PERMISSIONS 5](#_Toc153131294)

[6. PIPES AND REDIRECTS 6](#_Toc153131295)

1. SHELL SCRIPTING concepts/syntax

[1. Variables 7](#_Toc153131296)

[2. Environment Variables 13](#_Toc153131297)

1. VERSION CONTROL WITH GIT

[1. Definition 15](#_Toc153131298)

[2. GIT commands 15](#_Toc153131299)

[3. Introduction to Build Tools 16](#_Toc153131300)

1. INTRODUCTION TO ELASTIC COMPUTE CLOUD (EC2)

[1. Create an EC2 instance 17](#_Toc153131301)

[2. Connect to EC2 instance 18](#_Toc153131302)

[3. Install docker on EC2 18](#_Toc153131303)

[4. Run web application on EC2 19](#_Toc153131304)

[5. Make the app accessible from the browser 19](#_Toc153131305)

1. INTRODUCTION TO THE OS (operating system)

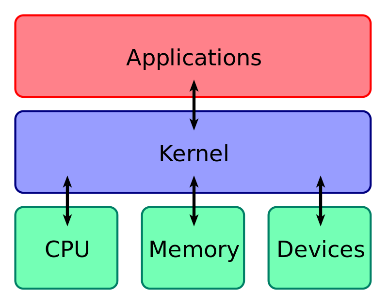
## Definition

The operating system is the **intermediary** between the software ant the hardware parts of a PC his role is to use the hardware (ram, CPU , storage…) to work with the software (browser for example). OS can also be the manager of the resources EX : CPU

## OS Tasks

* Process Management
* Memory Management
* Storage Management
* File Management
* Input/output Devices Management
* Security

## THE OS components

We can’t talk about the OS without giving the definition of the kernel, it’s the core component of an OS and serves as **the main interface** between the computer's physical hardware and the processes running on it

1. LINUX OPERATING SYSTEM

* To use Linux we have two ways, installing it directly in the pc or using a virtual machine in this course we will use the virtual machine solution (**VIRTUALBOX**)
* To do this we need to install two things :
* Virtual box
* Linux-Ubuntu

## linux file system

In the opposite of windows that works with multiple root folders , Linux use a structure called the hierarchical tree structure

* Let’s take a look inside of the Linux file system

Home : other locations **🡪** home (contains all the users)

Bin : (binary) contains all the basic commands

Sbin : (**system** binary) commands that needs a super user permission

Lib : (library) holds the libraries that some of the commands may need

/user/local : contains programs that the user install

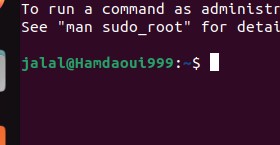
Dev : (devices) access only for apps and drivers

tmp : for temporary usage

usually the user doesn’t interact with this files the **OS** , **installer** and the **package manager** handle them

* About the hidden files, they’re files that starts with a dot mostly generated by a program or the OS itself and used to help prevent important data from being deleted

## INTRODUCTION TO CLI (command line interface)

In the GUI (graphical user interface) we can actually see the operations we’re making like creating new file or deleting it, in the CLI we can also do this operations but with **command lines**

* Username : jalal
* Computer name : Hamdaoui999
* ~ : home directory
* $ : regular user (# : root user)

## THE BASIC COMMANDS

* These are the basic command s that we will need to work with the terminal to do some basic operations :
* **ls :** shows all the files in the current working directory
* **ls -l :** list of all the current working directories
* **ls –al :** shows the list with the hidden files
* **pwd :** shows you where you are ex : /home/jalal
* **cd :** to specify the location to go
* **cd.. :** go back a step
* **cd (space) :** go back to home
* **touch file name :** create a file
* **touch file fileone filetwo filethree :** create file fileone filetwo filethree
* **touch file{1..10} :** createfile1 file2 file3… file10
* **echo MESSAGE! :** gives you back the message HELLO!
* **echo “STUFF!” > newfile.awsome :** create a file inside it STUFF!
* PRO TIPS :
* **nano filename (enter) :** enable you to edit the file

to save it hit (ctrl + x + y + enter)

* **vim filename (enter) :** edit the file text

to save it hit (esc + :wq + enter)

* **cat filename :** to see what’s inside of a file
* **shred filename :** make the file in binary
* **mkdir directoryname :**  create a new directory
* **cp filename destination :** copy the file to the destination
* **mv filename destination :** move the file to the destination
* **rm filename :** delete a file
* **rmdir directoryname :** delete an empty directory
* **rm –r directoryname :** delete a directory that contains files
* **ln –s filename linkname :** gives a link to a file
* **whoami :** gives you your username
* **useradd jalal2 :** usually it will give you permission denied we use
* **sudo useradd jalal2 :** type your password and you’ll add a user
* **su jalal2 :** type your password to access to jalal2
* **exit :** to go back to jalal
* **sudo passwd :** to change your password
* **sudo apt install :** to install
* **man (some command):** to explain what the command does
* **whatis (some command):** shorter way of man
* **wget link :** help you get some stuff from the internet
* **curl link > directory :** install something from the internet to a directory
* **zip zipedfile.zip zippingfile.awsome :** to zip a file
* **unzip zipedfile.zip :** to unzip a file
* **cmp file1 file2 :**  compare the file1 and file2
* **diff file1 file2 :**  tell us the exact difference between file1 and file2
* **sudo find / -name jalal :** gives you the files that contains the name jalal
* **ifconfig :**  to show your IP address
* **cat /etc/resolv.conf :** shows your DNS server address
* **resolvectl status :** shows your DNS server address
* **neofetch :** gives you all the infos about your machine
* **df –H :** shows you how much space you have in each directory
* **history :** to show you the history of the commands
* **sudo shutdown –h now :**  shutdown your machine instantly

## WORK WITH VIM EDITOR

**VIM** usage is for small modifications , faster to create and edit at the same time and when working with a remote server , writing the git commit message , display kubernetes configuration files

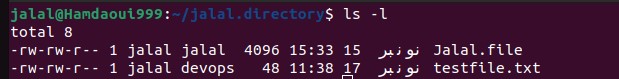
* sudo apt install vim
* vim filename 🡪 to execute

by entering to vim you’ll get by default the command mode you can navigate search delete undo the content etc

to convert to insert mode wish allows you to enter text you need to type i and start editing your file. When your done editing you have to go back to command mode, to do this you have to hit **esc** , to save and quit hit **:wq** or to quit without saving **q!**

* here are some useful vim commands :
* **esc + dd :** to delete a line
* **esc + d10 + d :** remove 10 lines
* **u :** undo
* **shift + a :** go to the end of the line and switching to insert mode
* **0 :** to jump to the beginning of the line
* **$ :** go to the end of the line without switching to insert mode
* **LineNumber + shift + g :** jump straight to a line
* **esc + / + searchedWord :** to search for a certain word
* **n :** to pass to the next one
* **%s/old/new :** replaces the old word with the new one

## FILE OWNERSHIP/USERS PERMISSIONS



Permission – username – group - space – date – filename

1. **Ownership permissions**

Ownership of files and directories comes down to two main groups: users and groups. Users are individual owners with specific access, while groups are collections of users with shared ownership and access. This setup makes it easier to manage permissions, ensuring that both individual users and groups can effectively control digital access and permissions.

* Users who owns the file/directory
* Group who owns the file/directory
* Usually the owner is the user who created the file
* The owning group is the primary group of that user
* We can create or change the owner user/group of a file
* **Both at ones :** sudo chown username:groupname filename
* **Just the user :** sudo chown NewUsername filename
* **Just the group :** sudo chgrp NewGroup filename

1. **File permissions**

Like the first image shows the first part is the permission part , to give more details about it we can represent it like this :

**d** stands for directory d rwx rwx r-x r : read

- For normal file - rwx rwx r-x w : write

x : **execute**

* First part of rwx is for the owner , rwx part two is for the group , third part is for all the other users that usually are allowed to read only
* Any “-” in the three parts that take a letters place stands for **no permission**

- **Editing permissions**

Before heading to editing the permissions we have to know first certain things

* U : Owner
* G : Group
* O : other
* A : all

So to edit a permission we use the following command :

**sudo chmod –thepermission(r,w,x) filename**

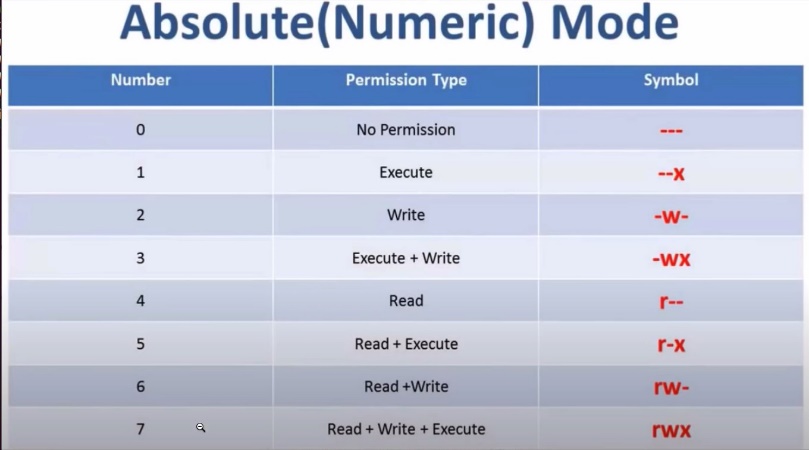
To specify the owner group or others permission use **:**

**sudo chmod (u,g,o,a)–thepermission(r,w,x) filename**

To add a permission we use “+” instead of “-” :

**sudo chmod (u,g,o,a)+thepermission(r,w,x) filename**

To add multiple permission we use “=” :

**sudo chmod (u,g,o,a)= thepermissions (r,w,x,-) filename**

* or we can do all of the above in a numeric way :
* **for example :**

**chmod 777 filename :** will allow the user , group and others to read write and execute the file

## PIPES AND REDIRECTS

1. **Pipes**
2. **Input/output and pipes in Linux**

The output of a program can be the input of another command

**“­|” pipe :** pipes the output of the previous command as an input to the next command

1. **Pipe and less**

**Command | less :** enable you to show the content of a file for example page by page using the pipe and less

1. **Pipe and grep**

Let’s say we wanted to look at the history of commands (or any other file) but we want to specify a word to appear in the output we use :

**history | grep searchedWord**

in case we had words separated **: history | grep “searched Word”**

* We can also use more than one pipe we use :

**Command1 | command2 | command3**

1. **Redirects in Linux**

* Redirection is putting the output of a command in a file that you can save in case you need it
* Using >> for the reason that the file we add will overwrite the previous one

To cut the content of file1 to file2 : **cat file1 > file2**

To copy the content of file1 to file2 : **cat file1 >> file2**

1. SHELL SCRIPTING consepts/syntax

## Variables

1. **Definition**

In programming, variables are placeholders or containers that store and represent data so instead of just retyping the name of a file that could be long or hard to type we could just type it for once and use the variation to change it to an easier one . Basically what variables do is that they allow you to label and manipulate information within a program, making it dynamic and adaptable.

file\_name (fileName) =test.txt

camel case

that’s how we declare that we have a variable, now we want to use it in our code ! to do so we need to call that variable :

**echo”using text.test with its new name $file\_name”**

what if we wanted to assign values to variables that are the output of a command execution , to clarify that take for example a file that contains some configurations let’s name it config for example to show what config has inside of it we type **ls config** give that a variable name for example **config\_files** combine these like this **config\_files=$(ls config)** to call that variable we do :

**echo”here are all config files : $config\_files”**

1. Conditional statements if/else

if/else is a conditional statement that you will find in many programing languages , it gives you a condition, **if** it’s true it will be executed if not (**else**) it pass to the next statement. It’s a between brackets built in command […]

* To understand it more we will write a code for a program that searches for a directory if it’s already in the pc else the program will create that directory :

**if [ -d “directory\_name”] - then** is the first to

**then** come after if **.**

**echo”reading config directory ” -** we use **fi** to end if

**config\_files=$(ls config)** statement .

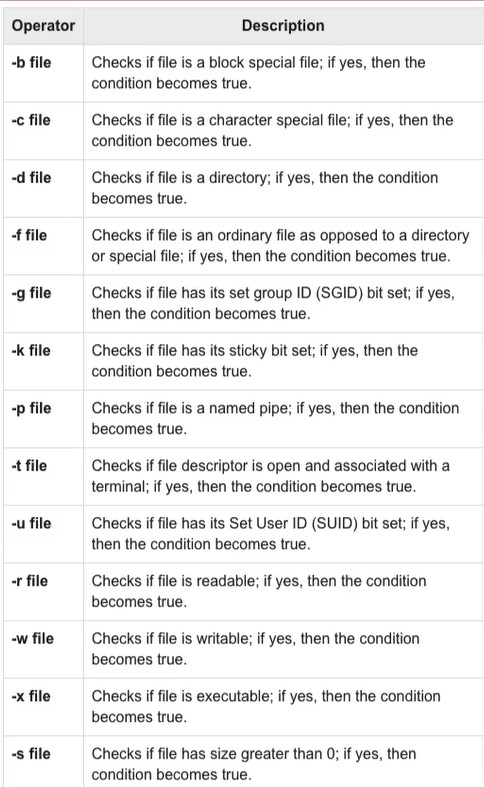
**else” config directory wasn’t found, Creating one ”**

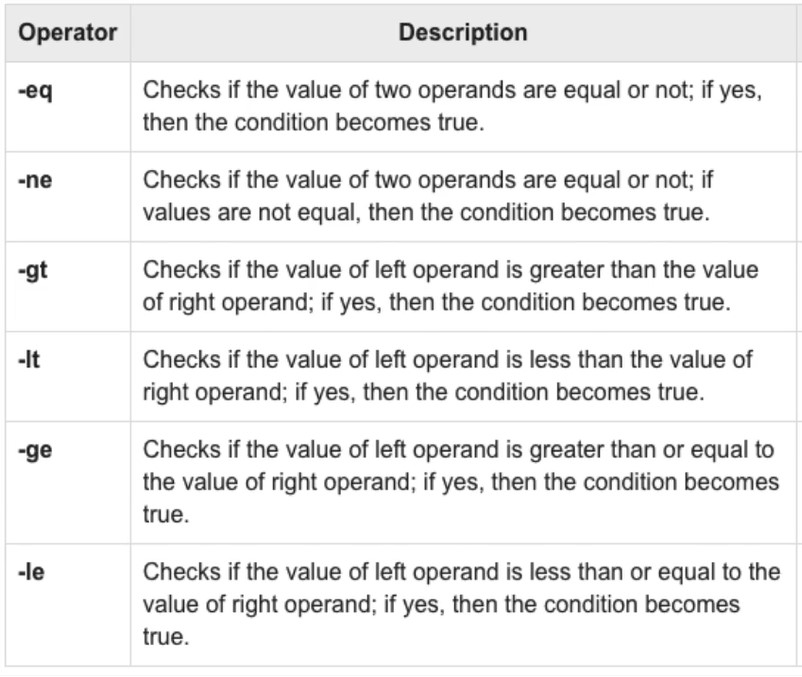
**mkdir config**

**fi**

**echo” here are all the config files : $config\_files”**

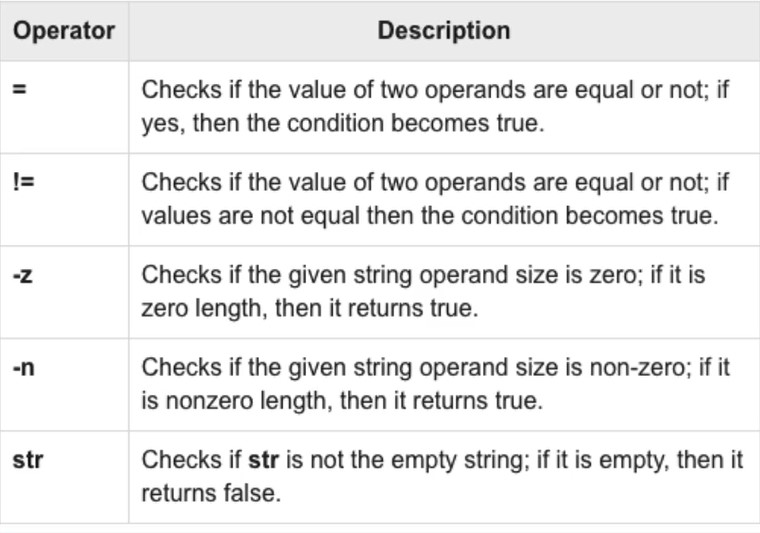
about the first part of the code [ **-d** “directory\_name”] checks if directory\_name is a directory using **-d** we call that file operators we also have another types of basic operators



🡨 file conditions that you can execute in file types and it’s called **file operators**

Helps us compare between operands 🡪

and its called **relational operators**

🡨 we use the **string operators** to compare characters or strings for example. “=” is a bash specific and the “==” is a posix standard it’s cross compatible with all shell programs

1. elif (else if)

We use else if or what we call elif to check more than one value. To understand this statement will write a code that checks more than one value :

**user\_group=xx**

**If** **[ user\_group == “nana”]**

**then**

**echo “configure the server”**

**elif [ user\_group == “admin”]**

**then**

**echo “administer the server”**

**else**

**echo “No permission to configure server. Wrong user group”**

* **So if the first block is true which is user\_group is nana it shows the message configure the server if not it checks if the second block is true and shows it’s message, if none of them is true it basically shows whatever is after the last else which is for this example No permission to configure server. Wrong user group.**

1. Passing arguments to a script :

Until now we have learned some basic codes that we can configure them from the outside, now we will learn a new thing that we call passing parameters to our code (script)

To read a parameter that is provided in your bash script we can access using the syntax **$** followed by **(1-9)** so that will be the first parameter that we pass in our script.

To get a better knowledge of how to read and accept a parameter we will use a modified version of the last script :

**User\_group=$1**

**If** **[ “$user\_group” == “user”]**

**then**

**echo “configure the server”**

**elif [ “$user\_group” == “admin”]**

**then**

**echo “administer the server”**

**else**

**echo “No permission to configure server. Wrong user group”**

**save and quit :**

**now if we did 🡪 ./dir.sh user**

**we will get the message 🡪 “configure the server”**

**now 🡪 ./dir.sh admin**

**we will get the message 🡪 “administer the server”**

* To provide two parameters we do the same steps with a little modifications :

Now we have to call two different parameters

**./dir.sh config\_dir admin 🡪 “reading config directory contents”**

**“administer the server”**

**./dir.sh setup-script user 🡪 “config directory not found, Creating one”**

**“configure the server”**

**Config\_dir=$1**

**If** **[ -d “$config\_dir”]**

**then**

**echo “reading config directory contents”**

**config\_files=$(ls “$config\_dir”)**

**else**

**echo “config directory not found, Creating one”**

**mkdir “$config\_dir”**

**User\_group=$1**

**If** **[ “$user\_group” == “user”]**

**then**

**echo “configure the server”**

**elif [ “$user\_group” == “admin”]**

**then**

**echo “administer the server”**

**else**

**echo “No permission to configure server. Wrong user group”**

what’s also cool about this is that we can reuse the same script many times just using different parameters

1. Read user Input

To store the data that the user writes as an input or let’s say to give the user a place where he can put certain information to be used as an input to our code we can use the command read.

**read** -p “type your age please : ” **user\_age**

**echo** “you are $user\_age years old !”

* Another thing is that how we can read all the parameters that the user provides using a special variable

**echo “$\*”**

* And to know the total number of the parameters the user passed we use the special variable

**echo “$#”**

1. Shell loops

Whenever we are working with a list of numbers or characters ex… we are working with loops that allows us to execute a set of commands repeatedly, and there are different types of loops :

* While loop
* For loop
* Until loop
* Select loop

About the for loop, this loop helps us do a list repeatedly until there is no more values left, to understand it more we will do the following script :

**for** param **in** $\*

**do**

**echo** $param

**done**

so the loop will assign the parameters that the user will provide as a list in the output

For the while loop, it executes a set of commands repeatedly until some condition is matched , there is a ton of use cases to this loop like a program that ping a service that currently started until the service becomes accessible and prints at the and a success message

sum=0

**while true**

**do**

**read -p “enter a score” score**

**if [ “$score” == “q”]**

**then**

**break**

**fi**

**sum=$sum+$score**

**echo “total score : $sum”**

**done**

to explain this code , it add the number that the user puts as an input to the previous value and repeats that over and over and to break this loop we used the if statement and **break** and quit the loop

## Environment Variables

Till now we know that the OS we’re using (LINUX) like every other OS can accept many users and each user has his own environment/account and has the right to configure it by setting preferences being isolated from other user environments .

So how does this OS keep storing this user’s configuration informations

1. Definition

Environment variables is where the OS stores all the users informations, they’re basically informations defined by key value pairs and are available for the whole environment

1. List all Env Variables

To see the list of the environment that the system has set for us we use the command **printenv**

We can also access to a specific variable using the same command followed by the name of the value we’re searching for : **printenv USER**

here are other important commands that does pretty much the same job as printenv :

**printenv | grep USER :** gives you all the USERS in your device

**echo $USER :** gives you the USER you’re working with

1. Application Env Variables

* Let’s say I want to run a python app or java app in my Linux server and this application need to connect to a database or a google maps API for example usually this kind of apps needs an authentication

* What can we do in this case knowing that we just can’t write our host, user and password in the code that will be easy to everyone in the planet to scratch it just from the web site
* So to store this kind of sensitive data we have to set them as env vars on server so that apps can read those env vars **DB\_PWD=secretpassword / DB\_USER=musqluser .** by making this env vars, we make them available in the environment so that they become accessible to all the apps and processes
* The question is how we can access to that env var . every programing language has its own library that can take this variables and use them
* It also makes the app more flexible, the apps should have more than one server each one has his own DB so instead of hard coding the user the URL and the password we can simply use the library in the programing language

1. Creating Env Variables

To create the env var we simply use the command :

**export DB\_USERNAME=dbuser**

**echo $DB\_USERNAME 🡪** to see the DB\_USERNAME

**echo $DB\_USERNAME=newdbname 🡪** to change DB\_USERNAME value

1. Deleting Env Variables

To delete an env var we do the command :

**unset** **DB\_USERNAME**

1. Persisting Env Variables

An important thing to know is that whenever we set a env var using the export command it will only be saved in the current session of the terminal once we close it or open a new one we will not be able to found the env vars again

To save the data permanently we use a shell specific configuration file with the command **vim .bashrc** then inside of it we write our env vars

And to tell the program that we have changed something in the bashrc we do

**source .bashrc**

1. **Persisting Env Variables (system wide)**

Earlier we created an env vars inside of the bashrc but this is a user specific solution, what if we wanted to do an env var that we can use with all the users in the system

To do that we go to a file where the system configurations is stored and it’s called **/etc/environment**

1. Add a custom command/program

To do so , we first have to explain what a path env variable is. It’s a file that stores the locations of all the commands in our system so whenever we execute a command we don’t have to write it’s path we simply type it and the program searches for it in this file called **/etc/environment**

Let’s see how we can add our executable program

**Step 1:**

Go to the bashrc the type this line : **PATH=$PATH:/home/jalal**

**Step 2:**

Add a vim file named whatever you want this command to be named **hello** example and write the code you want this command to do, you have to make the file executable by all

**Step 3:**

**Source .bashrc** to let the system know the changes you did

**Step 4(final step):**

Execute the command “hello” or basically the command you created in your user area

1. VERSION CONTROL WITH GIT

## Definition

Git is a distributed version control system that helps you track changes in your source code during software development. It allows multiple developers to collaborate on a project, keeping a history of changes, and facilitating team coordination. Git is widely used for its efficiency in managing branches, enabling parallel development, and providing a reliable mechanism for code collaboration and version management.

## GIT commands

|  |  |
| --- | --- |
| Command | Function |
| $ git init | Initialize loco repo |
| $ git add <file>[.,\*] | Add files to starting area |
| $ git status | Check status of file in staging area |
| $ git commit -m “message” | Commit changes to local repo |
| $ git commit, | to insert, :wq to commit | Commit changes to local repo |
| $ git push | Push local repo to remote repo |
| $ git pull | Push changes from remote repo |
| $ git clone | Copy remote repo into new directory |
| $ git rm -cached <file> | Remove file from repo |
| $ git config --global user.name “name“ |  |
| $ git config --global user.email “email“ |  |
| $ git branch <name> | Make a new branch |
| $ git checkout <branch name> | Change branch |
| $ git merge <branch> | Merges branch name to master branch |
| $ git remote add <name><url> | Adds a new remote repo |
| $ git push -u <name><branch> | Pushes changes to remote repo |

1. **Rebase**

Imaging working with more than one person in a project and you want to make changes every time you do this you have to pull your changes before pushing them to the remote repo that makes the history less cleaner and not easy to understand and track. Instead of that we can use the **git pull -r**  (r : rebase) to solve us this problem

1. **Gitignore**

Don’t track certain files or what we call (.gitignore) is a file we create that’s only available for the creator or the developer who created it to exclude certain files or folders from git to be tracked .

To create this file is actually easy first we go to the folder of the project that we want our gitignore to be in then we enter the files or other folders we want to hide followed by <file name>**/\*** ,

And to let the program know that we don’t want to track this files anymore we do the command **git rm -r - - cached <file/folder name>**

1. **Going back in history**

If we’re working on a specific project and we faced an issue that we didn’t know where it came from, we can actually go back to a specific project version by simply doing : **git checkout <commit hash>** and we grab the commit hash from the **git log** that becomes after that the branch we will work in

It let you in the condition you have been back then in the hash it maybe a week or a month

But to go to the latest commit state we do the same command but this time using the branch in the head position

1. **Merging branches**

Let’s say that the branch xx hasn’t been updated for a few weeks and you’ve added some commits to your master branch these commits now are missing in the xx branch. So what you will have to do now is to merge the changes that you committed in the master branch to xx, to do so you need to **git pull** in the master branch to make sure everything is up to date the **git checkout xx** after that comes the command that merges the master branch to the xx : **git merge master**

1. **Git for DevOps**

As a DevOps engineer you’ll probably be working with some developers and of course dealing with so many files such as

* Bash and python files
* Terraform and Ansible configuration file to communicate with AWS
* Many Kubernetes configuration files

And these type of files require so many things like a secure place to be stored at being tracked in history… etc

Theres also another use cases of git for you as a DevOps engineer like checkout codes, test and build apps. You also need integration for the build automation tool with application git repository and set it up. Or checking if the changes happened in the frontend or the backend.

## Introduction to Build Tools

When you’re done implementing your app it must be deployed on a production server it means that the app you built that been leaving in a repository must end-up in a live server

So how to move the app that you want to run in a live server so it will be accessible to the users

To do so we have to package the app to a single movable file and that file is called **Artifact** to make this type of files we have to do **packaging** 🡪 building the code

- Compiling the file

- Compressing the file

hundreds to 1 file

We need to keep the artifact in the storage because we will need it in other environment like the dev server, test server or production server. Also to have a backup in case we lost the file somewhere and that place we store the artifact in is called the **Artifact repository** like NEXUS or JFROG

You probably wondering how does this artifact look like, well it depends on the programming language you’re using for example if you’re using java the file format will be **JAR** or **WAR (**jar 🡺 java archive**)** this file will contain the whole code and dependencies , spring framework , datetime libraries , PDF processing libraries…

1. Introduction to Elastic Compute Cloud (EC2)

The EC2 is a virtual server in the AWS cloud, it’s an amazon made service that’s job is to provide us as users with a compute capacity

In this part of the course we’re going to learn how to deploy an application on EC2 instance, we’re going to do the following steps :

* **Create an EC2 instance on AWS**
* **Connect to EC2 instance with SSH**
* **Install docker on remote EC2 instance**
* **Run docker container (docker login, pull, run) from privet repository**
* **Configure EC2 fire wall to access app externally from browser**

## Create an EC2 instance

. Search AWS on your browser 🡺 Service 🡺 EC2 🡺 Launch instance

. Select the image you want to work with, in our case we’re going to use Amazon Linux 2AMI

. Choose an instance capacity, we are going to choose the free tier

. Do the network configuration “leave it at default”

. Add your storage “also going to let the default settings”

. Add tags “meta data as key value pairs”, it gives more information about the instance

. Change the security group name

. Configure security group “set the rules”

**LUNCH :**  - Create key pair on AWS

- Download the privet key locally on your machine

-Public key stored by AWS “you can’t change it ones you start”

Ones we are on the dashboard we can see :

* Instance ID
* State
* Subnet ID
* Public/Privet IP address
* Public/Privet IP DNS
* Security group name, storage , tags …

## Connect to EC2 instance

We first have to move the key that we downloaded previously :

SSH 🡺 **move “key path” ~/.ssh/**

Remove permissions except read for user

Now to access to the virtual server we do the following command

**ssh -i ~/.ssh/”key name” ec2-user@”puplic IP address ”**

## Install docker on EC2

We first have to update our package manager tool : sudo yam update

Then install docker : sudo yam install docker

Start docker daemon : sudo service docker start

Now we can execute docker command but with sudo , in order to execute docker commands without writing sudo first we have to add the user to docker group

**sudo usermod -aG docker $USER**

save the changes by logging out then logging in , now we can run the docker commands without sudo

## Run web application on EC2

* Get the image
* Know the port that the app will start the docker container at it
* Log the repo by the command : docker login “docker hub”
* Type the username and the password “the authentication is now saved for later”
* Run the image from the repo using : “the image full name“:tag
* **docker run -d -p 3000:3080 “image full name”**

## Make the app accessible from the browser

Our instance is actually attached to a security group wich has the firewall rules already configured and all we have to do is allow port 22 to be opened in the server : edit the inbound rules “rules for the requests coming into your server” outbound rules “rules going out from your server”

Now to make the website accessible we add **custom TCP + port 3000 + 0.0.0.0/0”to make it accessible to all users ”**

**To access to it we type in the search bar : “public IP address”:”the port number”**

Or in another way we do : **“DNS name”:”port number”**