

**Introduction**:

The next stop in our journey is the linux command line. But why linux?

*Docker has its foundation built on top of basic linux concepts*.

So if you want to be productive and troubleshoot issues easily, you need to know some of the basic linux commands.

**Linux Distributions**:

Let us start this section by talking about linux distributions also called *linux distros*.

As we know Linux is an open source software and for this reason many individuals and communities have created their own version of linux called linux distributions.



Each of these distributions are made to fit specialized needs like running servers, desktop computers, mobile phones and so on.

So we have these common Distros:



*Ubuntu*, one of the most popular linux distribution. There are more than a 1000 linux distributions out there.

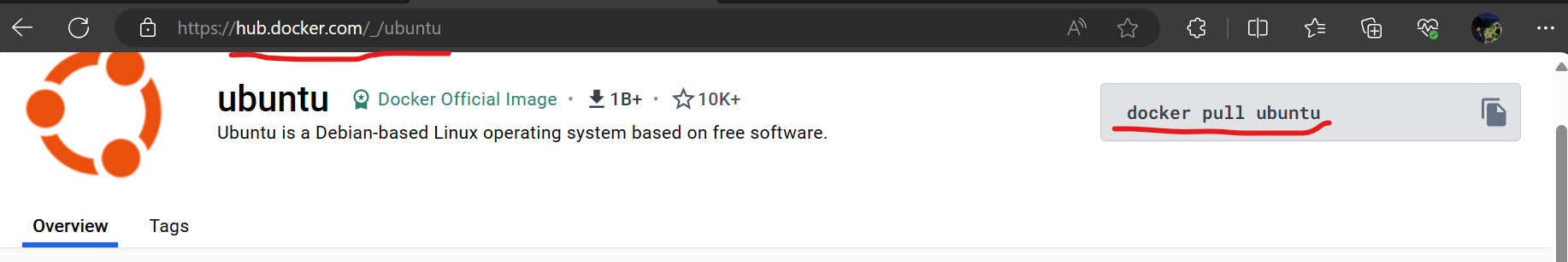
Most of these distributions support pretty much the same set of commands, but sometimes we might discover differences along the way.

In this section we will use,



**Running Linux**:

Go to hub.docker.com and search for ubuntu.



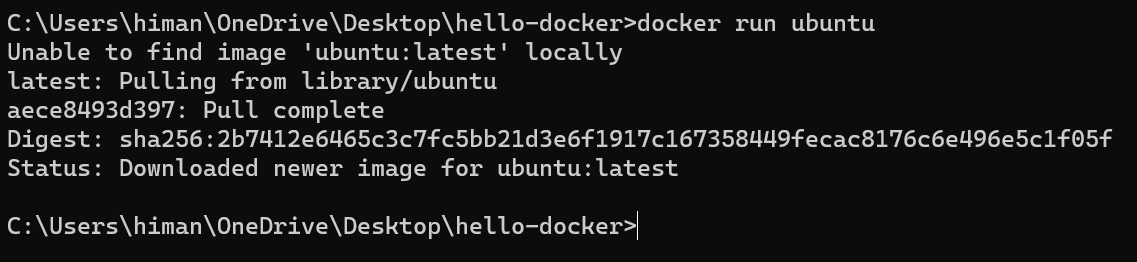
We can see the official ubuntu image.

On the right side we can see the command docker pull ubuntu to pull this image.

But in this lesson we are going to use a shortcut. So in the terminal we will learn a new command,

docker run ubuntu

*If we have this image locally on our machine, docker is going to start a container with this image otherwise it is going to pull this image behind the scene and then start a container*.

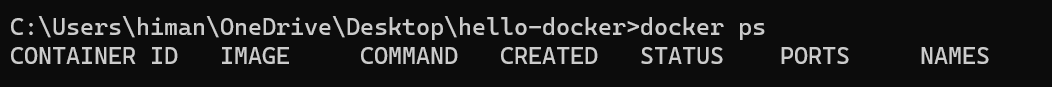


So we can see Docker is unable to find ubuntu locally, so it start pulling it from docker hub.

But what about the container? Docker started a container but since we did not interact with it, container stopped.

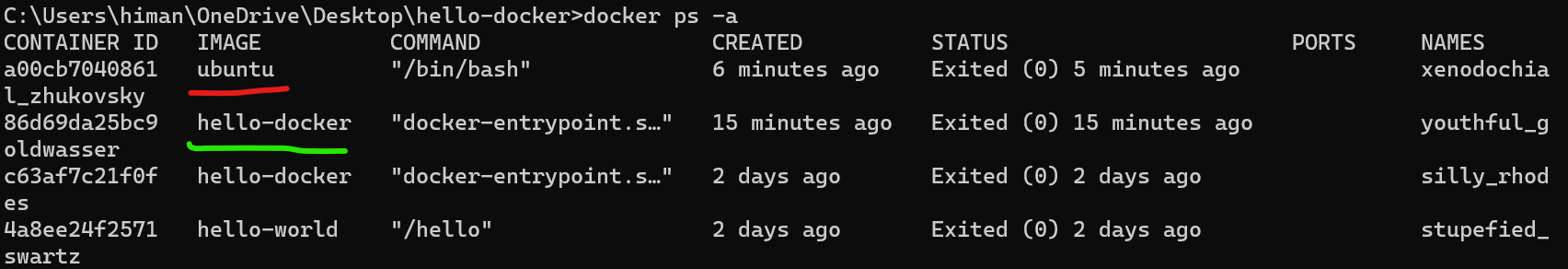
Let’s prove this,

If we run docker ps command,



We can *see the list of running processes or running containers* and we can’t see any running containers here.

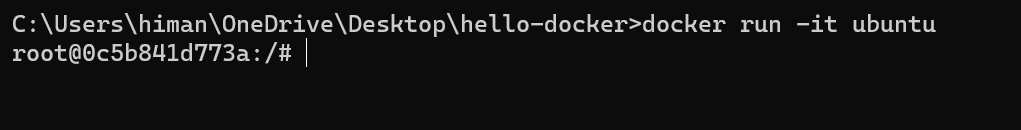
But if we run docker ps -a command, *we can see stopped containers as well*,



This ubuntu image container is the one we just started and hello-docker is the container we started earlier today.

So to start a container and interact with it, we have to type docker run *-it* ubuntu

Here *-it* *stands for interactive used for starting a container in interactive mode* and we used *ubuntu* image inside this container which we have locally.



What we are seeing here (root@0c5b841d773a:/# ) is called a *shell*. *A shell is a program that takes our commands and passes them to the operating system for execution*.

Let us break this down so it does not look mysterious,

**root** 🡪 Represents the currently logged in user (*by default we are logged in as root user, which has the highest privileges*).

**@0c5b841d773a** 🡪 After the @ sign we have the name of the machine. So this container has this id which is automatically generated by docker (*which is like the name of a machine*).

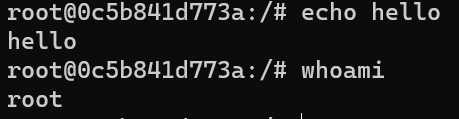
**: /** 🡪 After colon we have / (*forward slash*) which represents where we are in the file system. A plain / represents root directory (*highest directory in the file system*).

**#** 🡪 The # (*pound or hashtag symbol*) means this user has the highest privileges (*currently root user*). *If we logged in as normal user, instead of # we will see a dollar ( $ ) sign*.

In this shell we can execute couple of linux commands like

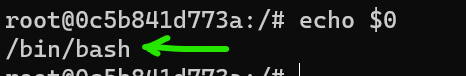
echo (*prints a string on the terminal*),

whoami (*prints the current user*),



***The shell program takes these commands that we are running inside it and passes them to kernel for execution***.

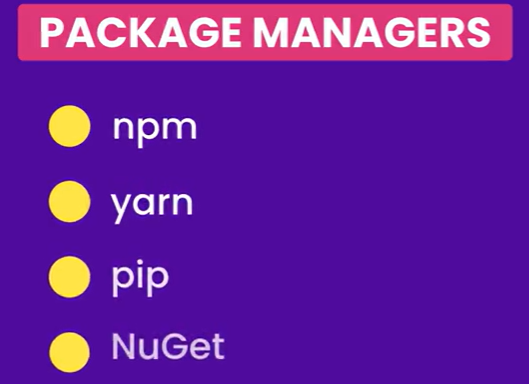
If we type echo $0 , we can see location of the shell program,

 its /bin/bash

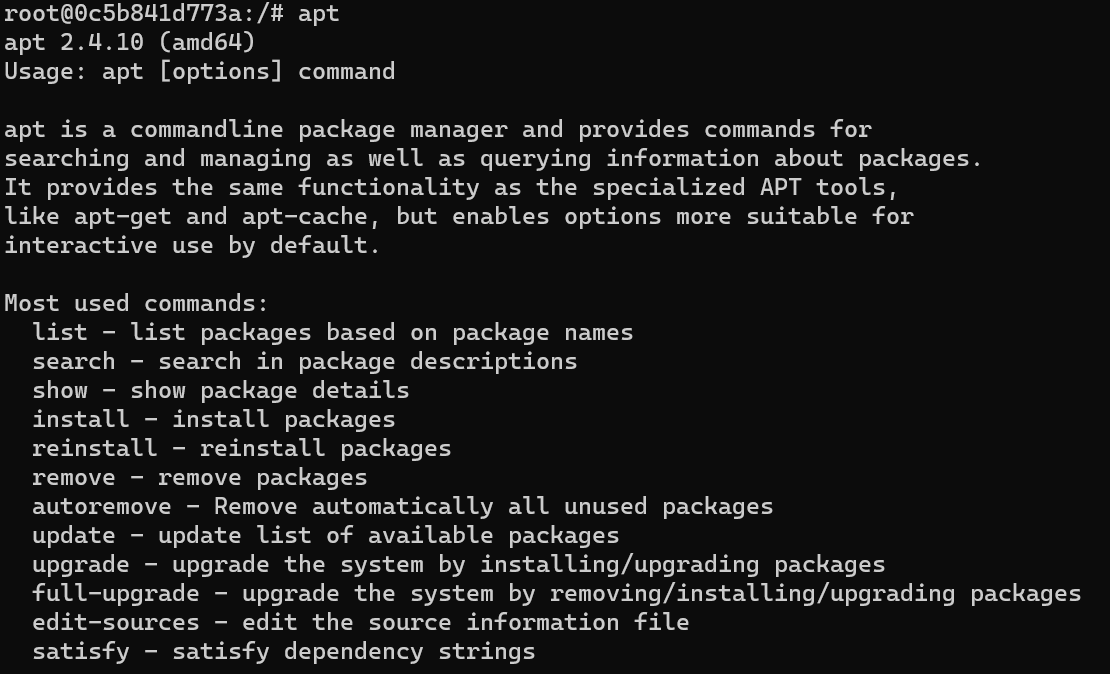
Here */bin* is a folder or directory and inside this directory, we have a program called *bash* which is short for ‘Bourne again Shell’ (*bash is an enhanced version of the original shell program*).

**Managing Packages**:

These days most operating systems and development platforms comes with a package manager. Below are some of the popular package managers,



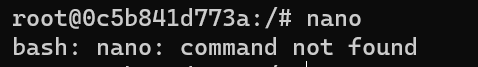
In ubuntu we also have a package manager called ***apt***. It is short for *advanced package tool*.



If we run apt command, we can see this command has a bunch of sub commands like *list*, *show*, *remove* etc.

Note: Technically apt is the newer package manager, but we also have ***apt-get*** which we see in a lot of online tutorials. Going forward we will use apt because its easier to work with.

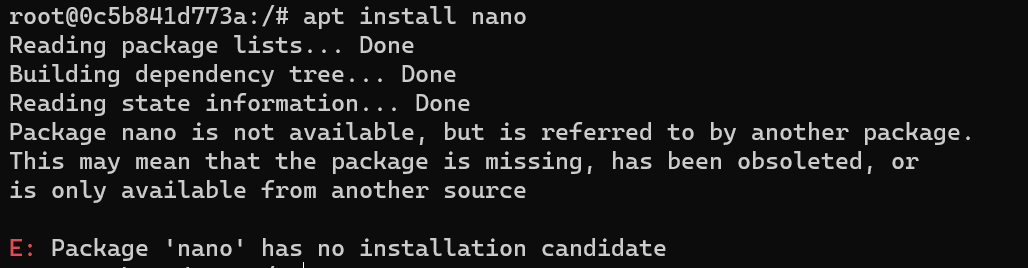
Let’s say we want to install a package called *nano* (*a basic text editor for linux*). If we simply type *nano* in our command line.



We get command not found, because this ubuntu image we are running does not have *nano*.

This is where we use *apt* to install this package,

apt install nano

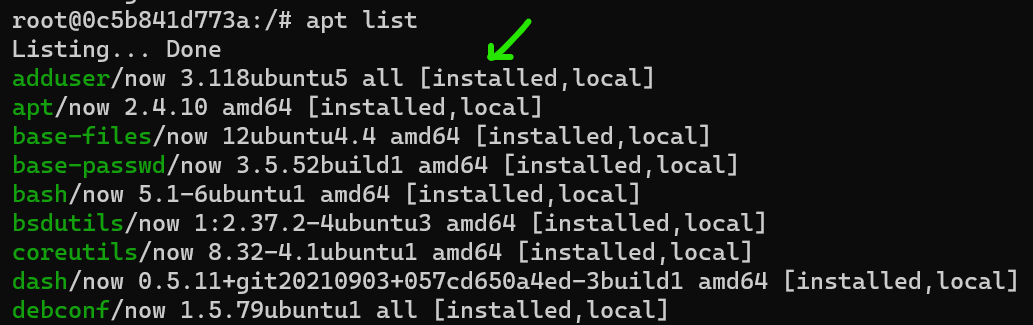


We are unable to find this nano package in our package list.

Why this is happening?

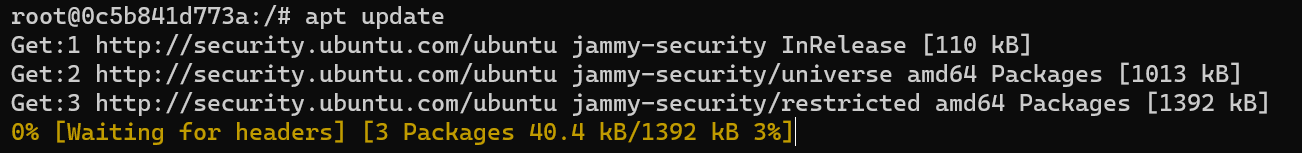
Its because *in linux we have a package database and this database might contain hundreds of packages but not all these packages are installed*.

So if we need to see all the packages in this database we type apt list command.



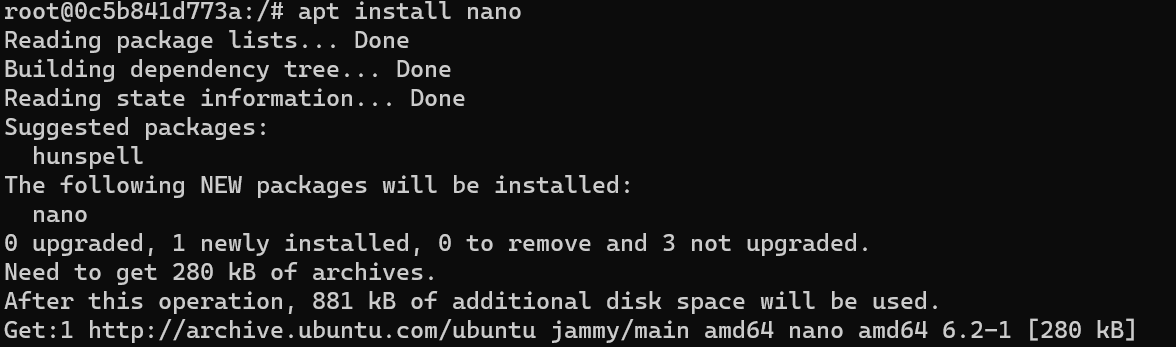
Some of these packages have ‘installed’ status, but not all packages.

So when we typed apt install nano, this command looked at the database and it cannot find a package called nano. So this is where we use the apt update command to update the database.



Linux goes to all the available sources one by one and download the list of packages. *After running this command our package database is updated and we see lot more new uninstalled packages when we run* apt list.

Now we run apt install nano command again,



The package called nano is installed now.

Note: Before installing a package always run apt update command to update package database and then install a package

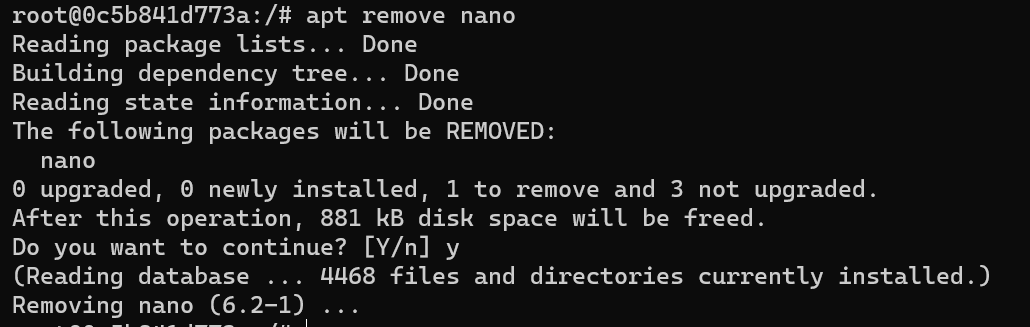
Let us make sure our nano package is installed properly. So type nano in the terminal,



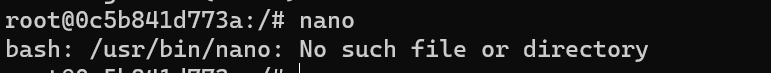
We get a text editor with shortcuts down below.

Now if we want to remove this nano, we can simply,

apt remove nano



If we type nano again after uninstall,



**Linux File system**:

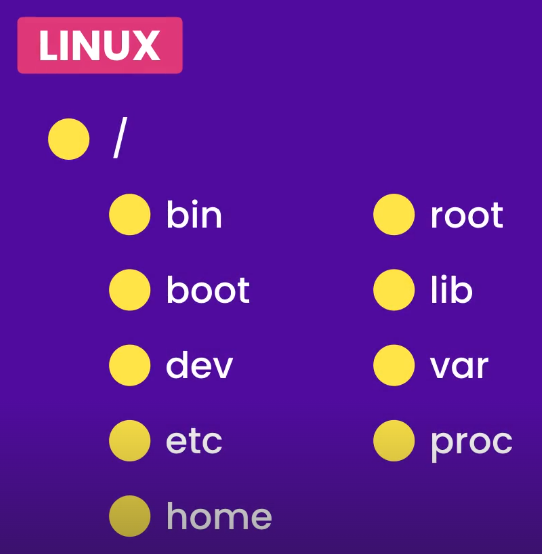
In Linux just like windows, our files and directories are organized in a tree (*hierarchical structure*).

In Windows we have a structure like this,



With C drive on top of hierarchy, then below we have directories like Program Files and Windows and so on…

In Linux we have root directory on top of hierarchy.

🡨Below root we have a bunch of standard directories. For example we have *bin* (*which include binaries or programs*) and *boot* (*includes all files related to booting*), then we have dev (*short for devices*) and so on…

In Linux,



Including devices, directories, network socket, pipes and so on…

So the files that are required to access the devices are stored in the *dev* directory. Then we have *etc* (***editable text configuration****, a place for configuration files*).

We also have *home* (*storage place for user’s home directories*). So, on a machine with multiple users, each user is going to have a home directory here.

Here *root* is the *home directory of the root user* and only root user can access this directory.

*lib* is used for keeping library files like software library dependencies.

*var* (*short for variable*) keeps the files that are updated frequently like lock – files, application data and so on.

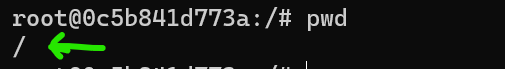
Finally we have *proc* which includes files that *represent running processes*.

So once again in Linux,

. Processes, devices and even directories are files.

**Navigating the File System**:

First command we start with is pwd, (*print working directory*).



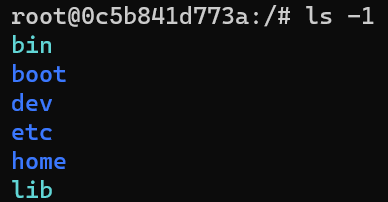
With this command we can see where we are currently in the file system (*forward slash / represents root directory*).

To see the files in the current directory we can use,

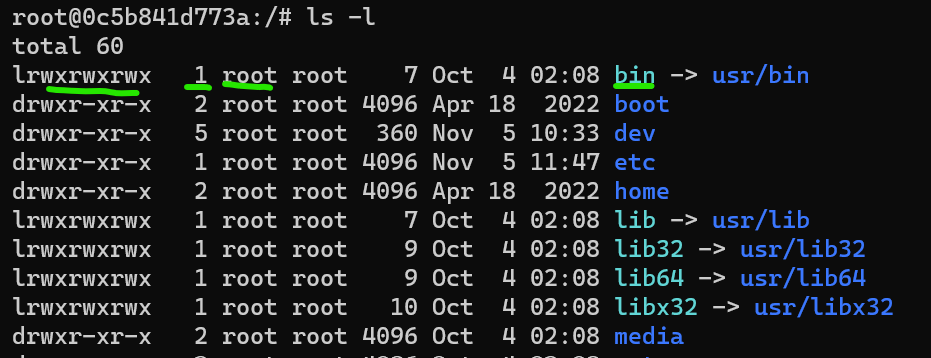
ls (*short for list*)



ls -1(*for printing one item per line*)

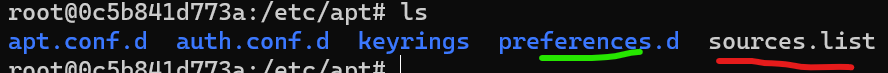


ls – l (*its -* ***L*** *for showing long listing*)

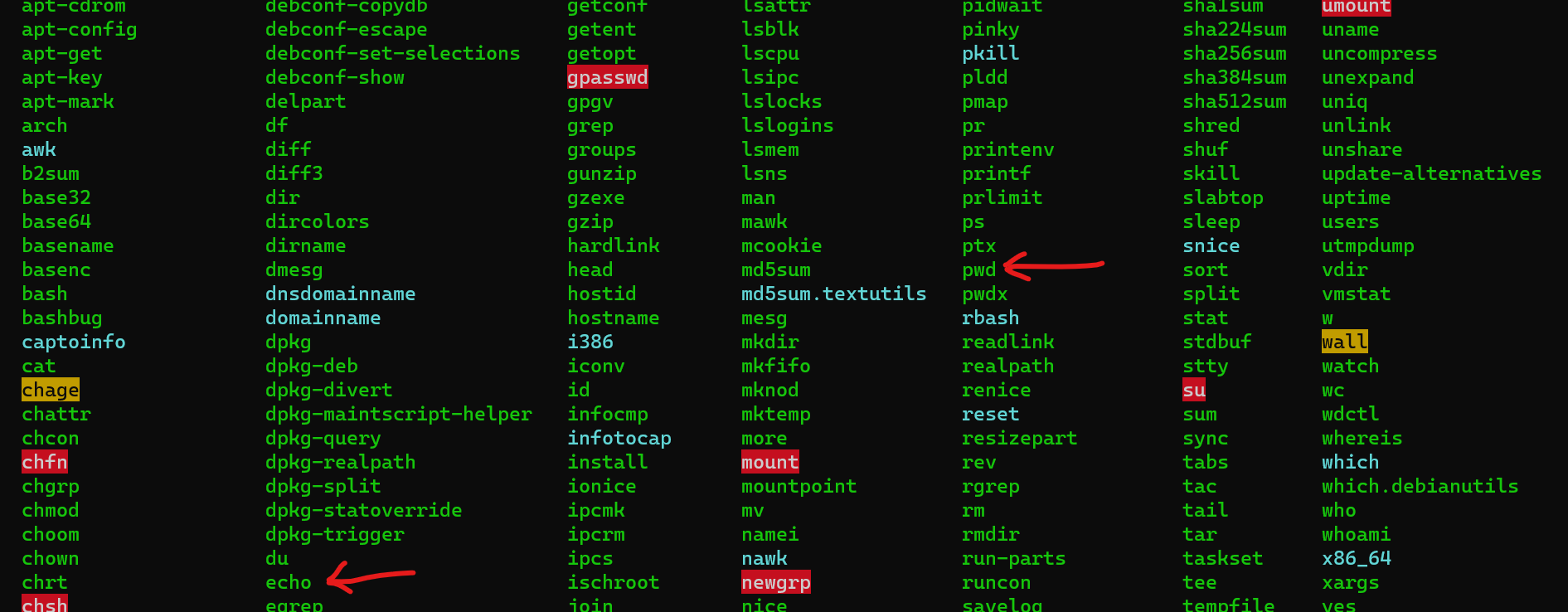


In first column *lrwxrwxrwx* , we have the permissions for this file.

Note: Blue ones are directories and other are files(*sources.list*).



Look at the content of the *bin* directory,



Here we have *pwd*, the command that we just executed which is a program in this bin directory. Another example is echo which is another program or binary in the bin directory.

Note: There is a shortcut to reach home directory of any user (*specific home for specific user and root for root user*). Use **~** sign after cd command like **cd ~**



And we are inside our root directory since we are logged in as root user automatically with this command.

**Manipulating Files and Directories**:

Currently we are in our root directory,

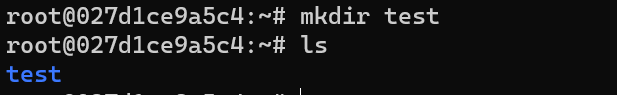


And we want to go to our home directory, so cd ~



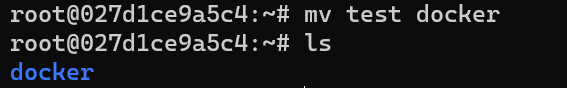
Now we are in our home directory.

In this directory we will *create another directory* called test so mkdir test



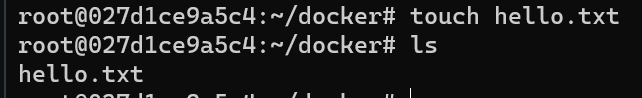
To *rename this directory* we use, mv command,

mv test docker (*will rename test to docker*)

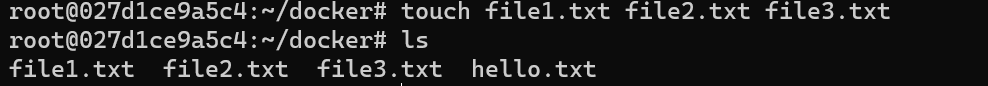


To *create a new file* we use touch command,

touch hello.txt



We can also create multiple files in one go using touch command,

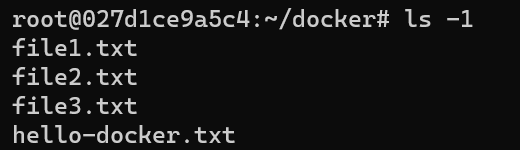


Note: If in the middle of typing a command, you want to *remove the entire word in one go* without pressing backspace.

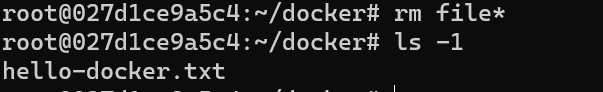
Use **ctrl + W** to do it.

We can remove a single file or multiple files in one go using rm command, but we can also use a pattern for *example remove all the files that start with the word* ***file*** *in it*.

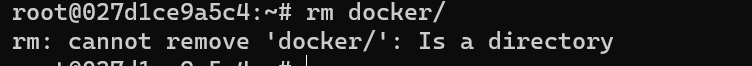
Before using rm command,



After using rm file\* command,

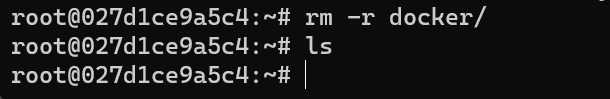


Note: We cannot remove directories with files by using only rm command, to do that we need to use rm -r, (*r is short for recursive which deletes directory and all its content recursively*).



So we use,

rm -r docker/

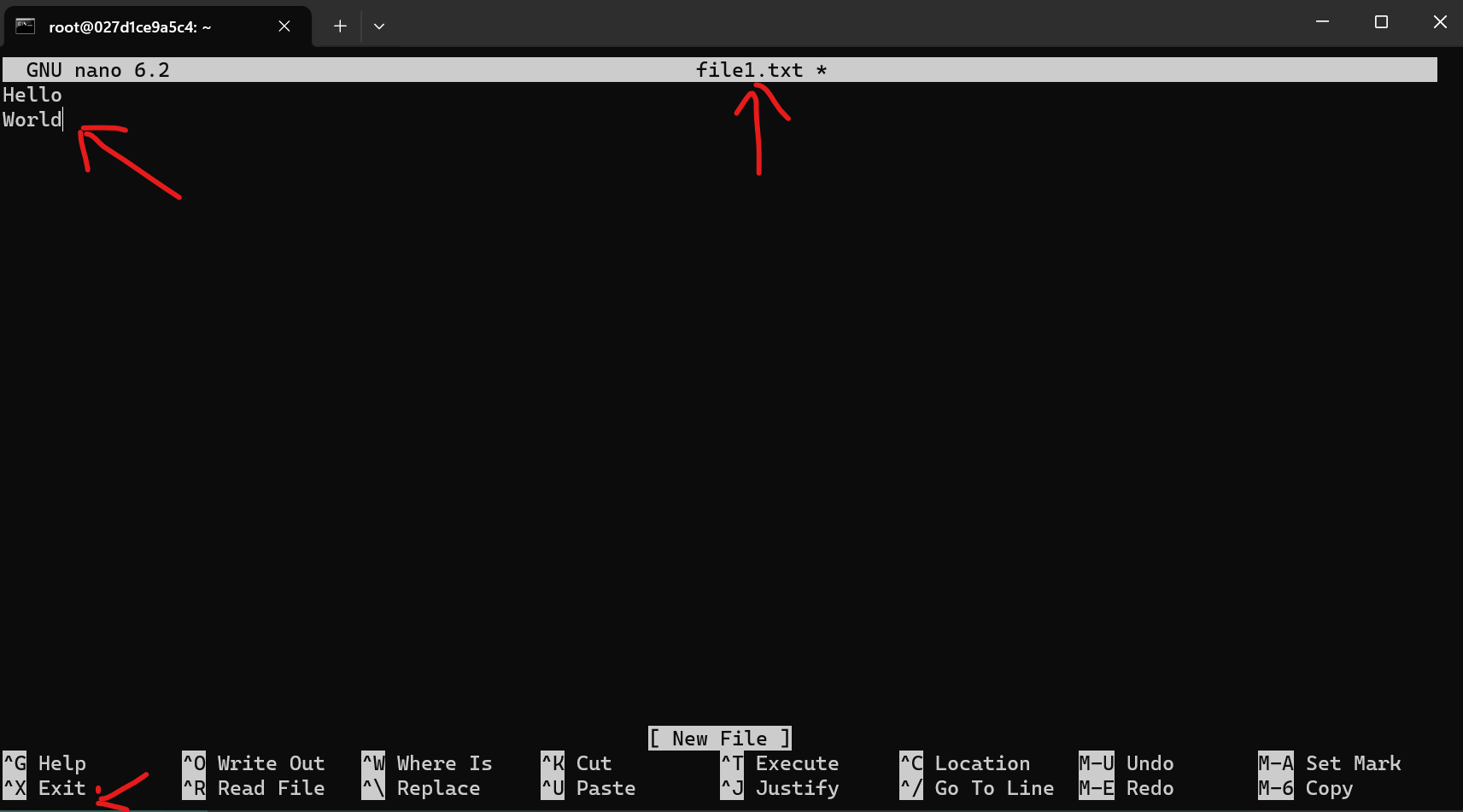


**Editing and Viewing files**:

Earlier we briefly talked about nano, which is a basic text editor for linux. So let us install it in our image.

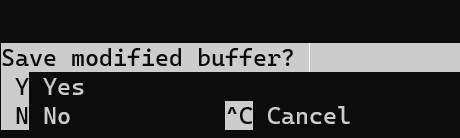
Using apt update, then apt install nano.

To launch it use nano or supply a file name optionally like nano file1.txt.

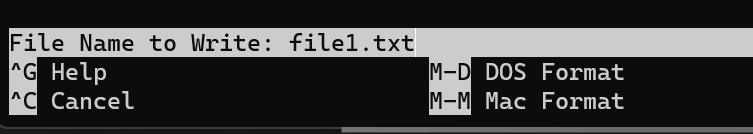


Here we have a basic text editor in which we can type whatever we want and when we are done press *ctrl + X* to exit.

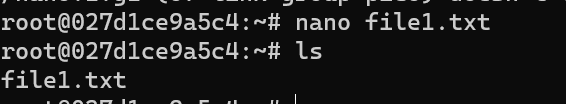
Then it will ask to save the file or not,



Then it will ask if need to edit file name before saving, press *enter* to save.



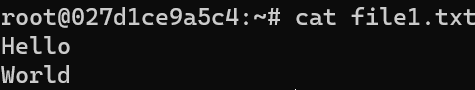
After saving, we can see that we have a new file called file1.txt inside our current directory.



To see the content of this file, we have a few different commands,

1. cat (*it is short for concatenating, with this we can concatenate contents of different files*) as well as view content of a file.

cat file1.txt



It is useful if our file is short and fits on one page.

1. If we are dealing with a large file, it is better to use more command.

We can open a file called *adduser.conf* which is inside *etc* directory.

Commands are **more**, **less**, **head** and **tail** for viewing file content. Read about them if needed.

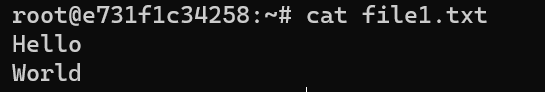
**Redirection**:

One of the important concepts in linux is the concept of standard input and output.

Standard input represents the keyboard and standard output represents the screen.

But *we can always change the source of input or output which we call* ***redirection***.

Let us see this by example using cat command.



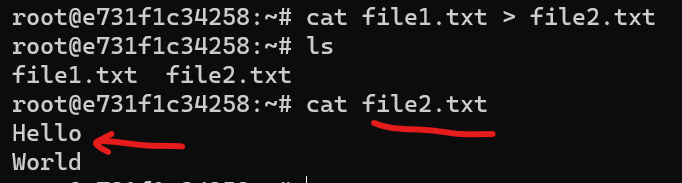
We used cat command to see the content of this file1.txt.

***“****When we execute this command or this program called cat, reads data from the file and prints it on the standard output which is the screen****”***

But using the *redirection operator* **>** , we can redirect the output from screen to let’s say a different file.

cat file1.txt > file2.txt

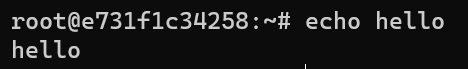
Here, cat will read the content of file1 and write it inside file2.



We see the exact same content inside file2.txt.

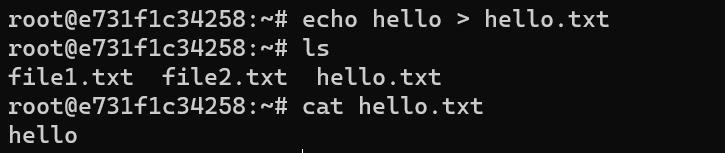
Note: The redirection operator is not just limited to cat command, we can use it pretty much anywhere. For example, earlier we talked about echo command.

If we type echo hello



We see the result printed on terminal.

But if we say echo hello > hello.txt



We have a new file called hello.txt which contains a plain hello text.

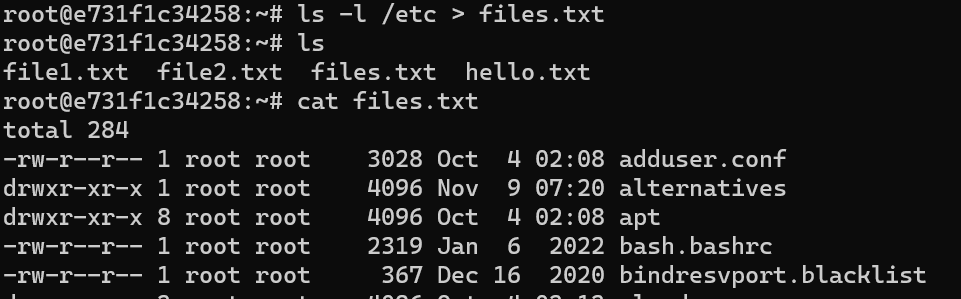
So *If we need to write a single line to a file, we do not have to use nano, we can use the echo command*.

Exercise:

Get a long listing of the files in /etc/ directory and write the output to a file.

ls -l /etc > files.txt

Here ls -l is *to get the long listing of files* then using > redirection operator we print this inside a new file.



Note: Just like using > (*greater than*) we can redirect output, similarly by using < (*less than*)redirection we can redirect input source as well.

**Searching for Text**:

Let’s see how can we search for a string in file.

We can use *grep* command which is short for *global regular expression print*.

Let us say we want to search for the word hello in file1, so we type grep hello file1.txt

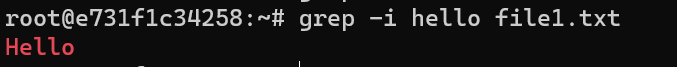


But we are not seeing any output, we clearly wrote Hello World in our file.

This is because *grep search is case sensitive by default*.

So if we want to remove case sensitivity use an option **-i** (*for case insensitive*)

grep -i hello file1.txt



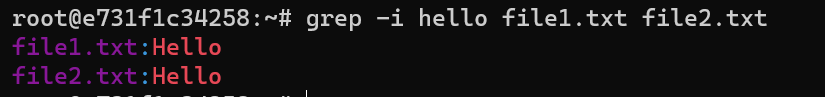
Find a word like *root* inside a file of another directory,

grep -i root /etc/passwd



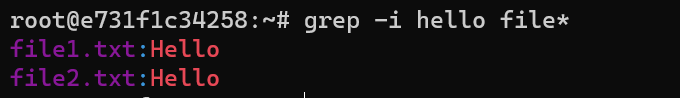
Search for same word in multiple files,

grep -i hello file1.txt file2.txt



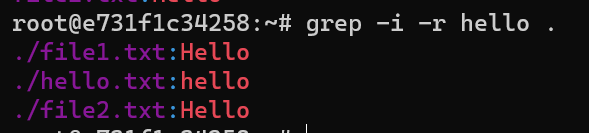
Or use a pattern, look for this word hello in all files which have names starting with file.

grep -i hello file\*



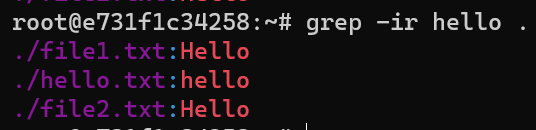
To search for a word in all files inside a directory and its subdirectory we use the recursive option -r

grep -i -r hello .



Note: In linux we can combine different options together so instead of using -i and -r separately we can combine them like this,

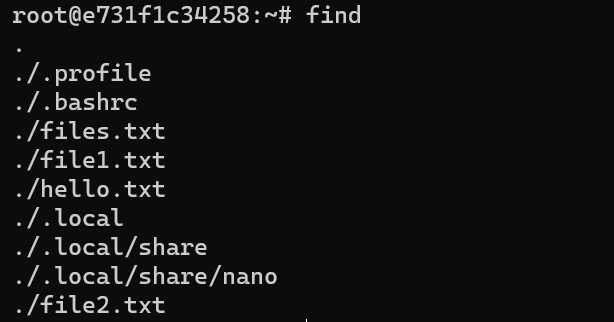
grep -ir hello .



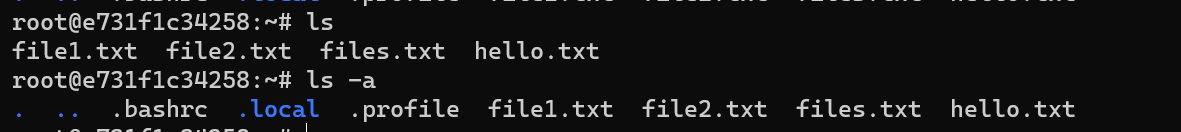
**Finding files and directories**:

In Linux we have the find command to find files and directories.

*If we execute this command without supplying any arguments, we see all the files and directories in our current directory* ***recursively***.



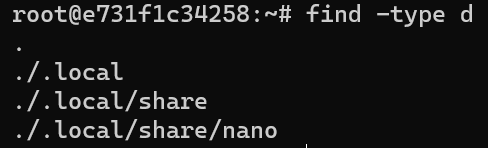
There are some files here, which we did not see earlier when we executed ls command because by default ls does not show hidden file (*we use ls -a to see those hidden files*).



We can also filter our search results in find command.

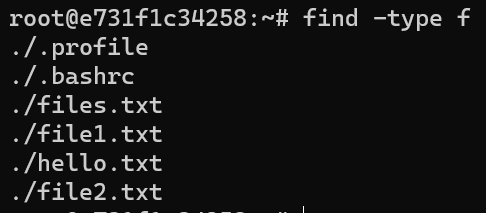
1. If need to see only directories,

find -type d



1. If need to find only the files.

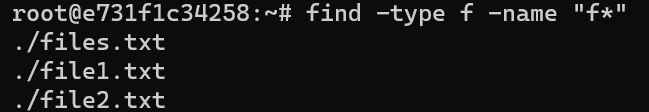
find -type f



1. Filter by a specific name.

For example Find all the files whose name starts with f.

find -type f -name "f\*"



To make the search case insensitive we use *-****i****name* as an option.

**Exercise**:

Find all the python files in this image and write the result to a file called pythonFiles.txt

*Hint* : To search across entire image first go to / to start searching from root directory.

**find** / -**type** f -name "\*.py" > python-files.txt

**Chaining commands**:

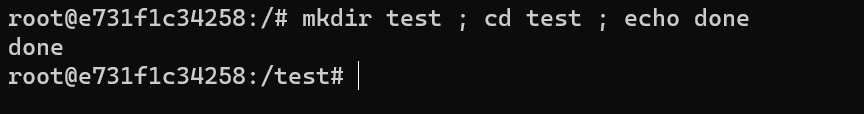
In linux we have a few different ways for chaining or combining multiple commands.

Let us say we want to create a directory called test and right after this, we need to go into this directory and echo something to the terminal.

Hint: We can separate multiple commands using ;

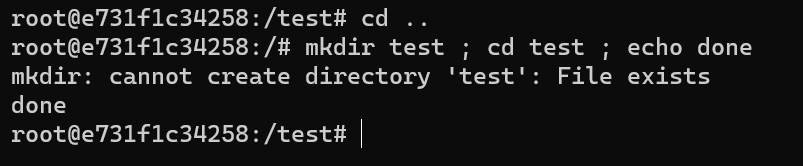
mkdir test ; cd test ; echo done

When we press enter, all these commands gets executed one after the other.



We get the done message and we are inside test directory.

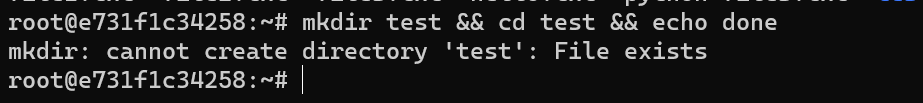
If we type the above command again, we would get error message because a directory with name test has already been created but last two commands will be executed.



What if we want to stop execution, *if one command fails other commands should not be executed*.

This is where we will use **&&** (*AND operator*)

mkdir test && cd test && echo done

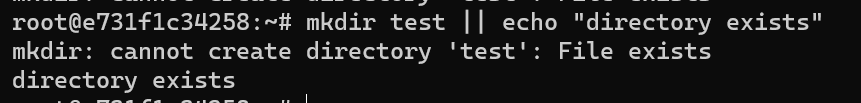


We see the error but we are still inside our home directory.

We also have the **||** (*OR operator*).

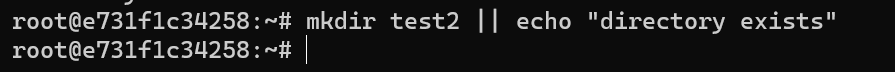
mkdir test **||** echo "directory exists"

If we say create the test directory OR echo “directory exists”. If mkdir command succeed, the echo part will not be executed. But if mkdir command fails, echo command will be executed.



If we create a new directory called test2, this time we will not see echo message, because we successfully created the new directory using mkdir command.

mkdir test2 **||** echo "directory exists"



These techniques are very powerful and we use them a lot when it comes to deploying applications using docker.

Another way to chain commands is via ***piping*** which is very powerful.

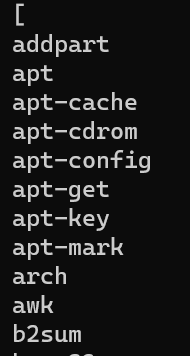
Let us look at the content of the /bin directory using ls /bin



Here we have a long list of files, but what if we want to look at this list using *less* command (***apt install less,*** *with this we can view content of the file and have the ability to scroll up/down using arrow keys🡪 less file1.txt*).

We use the ls command to get the content of the bin directory and then create a pipe, so we get the output of this command and send it to less command.

ls /bin **|** less



*Essentially we are creating a pipe, what comes out of first command goes into second command*.

This way less does not need file name as argument because it gets its input from the first command.

We can use piping in any combination,

ls /bin | head -n 5

Here we used *head* with 5 as argument instead of *less* to show first 5 lines.

Note: Sometimes when dealing with a long command our command sequence might look a little bit hard to read, so to deal with it we can split it into multiple lines using **\**.



We can keep this going as long as we want and this is how we can break up a long command into multiple lines using \(*backslash*).

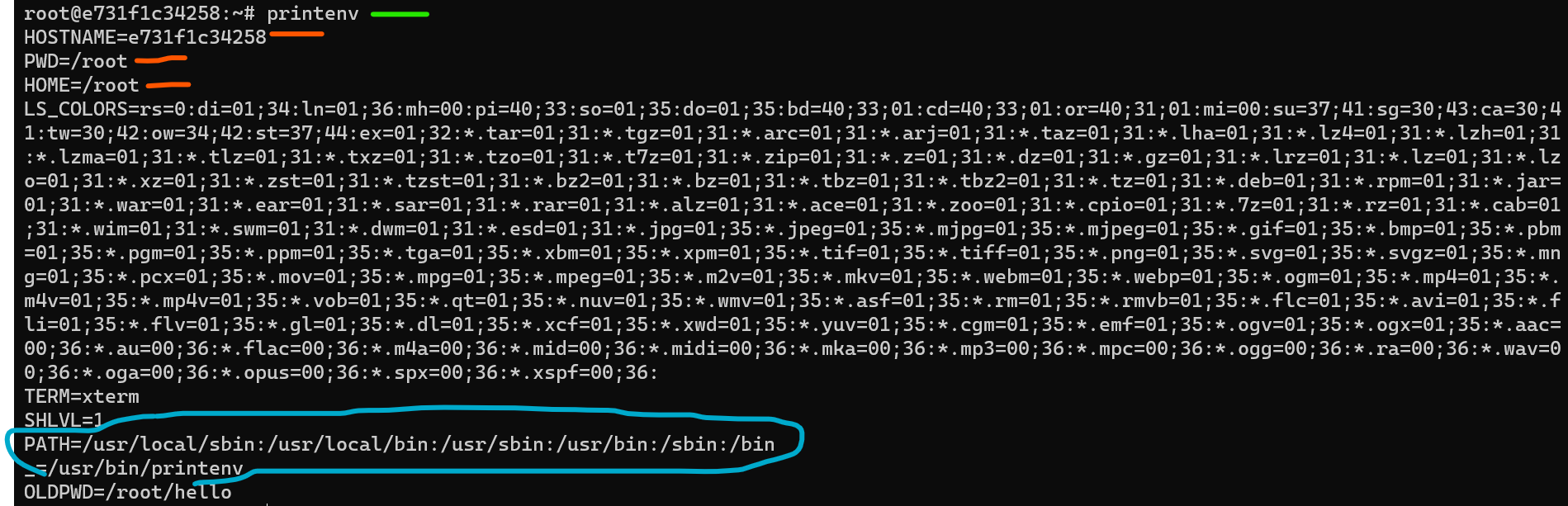
**Environment Variables**:

Just like we have variables in our programming languages, *in linux we have environment variables which we can set for storing configuration settings for our applications*.

*Our applications can read configuration settings from these environment variables*.

In this lesson we will see a few different commands for viewing all these environment variables and setting them.

1. First command is printenv and with this *we can see all the environment variables on this machine*.



Here we have a bunch of key – value pairs separated by equal sign. First variable is HOSTNAME (*it’s the id of our container*) which is generated by docker automatically. Then we have PWD, HOME, LS\_COLORS and so on…

Down below we have a very important environment variable called PATH.

*Sometimes when we run a program from command line we get an error saying the program or command was not found, even though we have installed that program on our machine.*

*Quite often it happens when our operating system whether its linux or windows cannot find that program*.

Now to find that program our operating system does not go through our entire hard drive its only going to look at specific directories and those directories are specified using the PATH variable.

So this is set to a list of directories separated by colon.

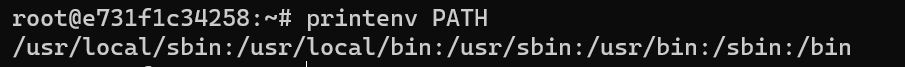
*PATH*=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

\_=/usr/bin/printenv

These are the directories linux or windows searches for, to find a program or a command.

If we want to *look at the value of any variable* say just PATH *in environment variables*,

printenv PATH



Another way is to *use echo command with $ sign as a prefix to the variable name we want to see*,

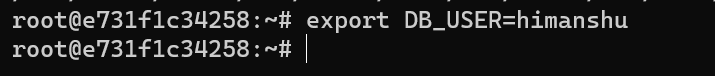
echo $PATH



When we use $ sign, linux knows that we are referencing to an environment variable.

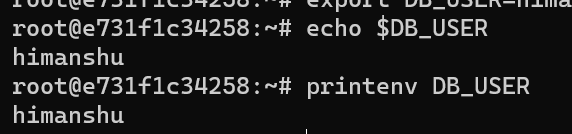
If we want *to set a new variable* we can use *export* command.

export DB\_USER=himanshu

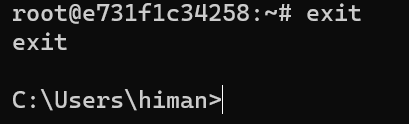


Here we have created a new variable called *DB\_USER* and set it to a value, *but this variable is stored in the current terminal session*.

So we can read it using echo $ command or using printenv,

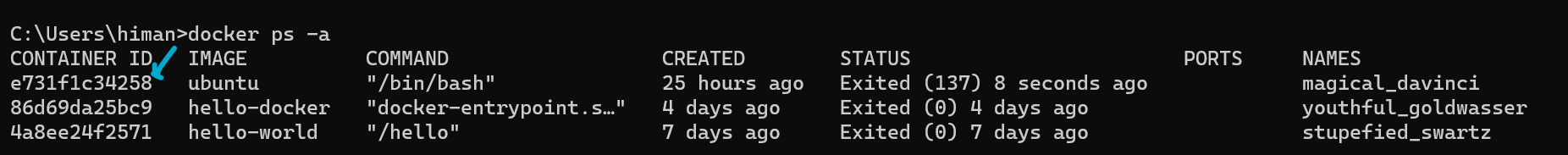


However this variable is only available in the current session, so if we close this terminal session using exit command and open a new terminal session, this variable will not exist.



After exiting we are outside our ubuntu container environment and back into windows terminal.

Now look at all the containers currently in our docker, using docker ps -a



If we want to *start this container* (*just like we start a virtual machine*)

docker start -i <*container-id*>

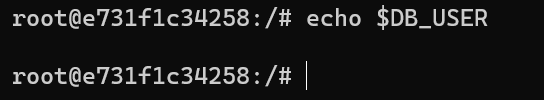
We will grab the first few letters of the container – id we want to start,

docker start -i e73



Note: *-i* means interactive (*something we can interact with*).

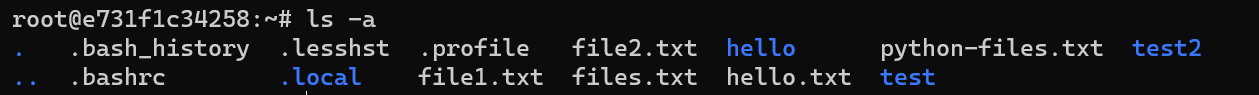
So now we are inside the same container again, and if we echo $DB\_USER, we don’t see anything.



Its because this variable does not exist anymore, it was only available in that terminal session.

To make this variable persistent, we need to write it to a special file.

If we ls -a in our home directory to see all files,



We see a file called **.bashrc** which is a file for *user’s personal startup file*. *Every time a user logs in, linux loads this command from the user’s home directory*. This is where we can write permanent environment variables.

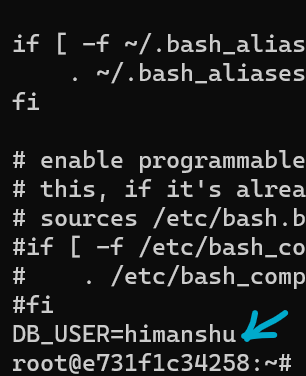
We can use *nano* or *echo* command to append this variable in our .***bashrc*** file.

echo DB\_USER=himanshu **>>** .bashrc

Note: Here we use **>>** instead of **>** because if we use single greater than sign, the entire .bashrc file will be overwritten. So *use double >> signs for appending something inside a file*.

Now to verify our work, let us look at .bashrc file.

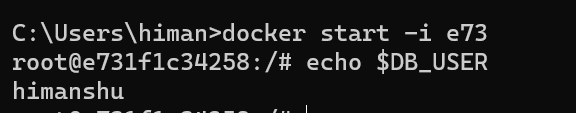
cat .bashrc



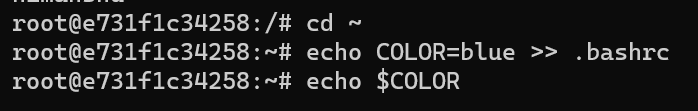
The last line contains our permanent environment variable and of course we can always come back and change its value.

Note: We should never store sensitive information inside an environment variable because at the end of the day, these variables are stored in plain text files.

Now if we exit and open a new terminal session, we can still read the value of our variable.



The changes we made in bashrc file are only effective in the next terminal session, so if we write another environment variable in bashrc, that variable is not going to be available until we open another terminal session. For example,



Here we append another variable called COLOR to our bashrc file but if we try to see it, nothing happens.

*It is because .bashrc file is loaded only once when we start a terminal session*.

There are two workarounds for this…

🡪 Exit the session and come back again.

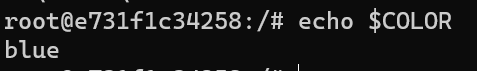
🡪 Use the *source* command *to reload .bashrc file*.(*execute it from home directory cd ~*)

source .bashrc

if not in home directory then

source ~/.bashrc

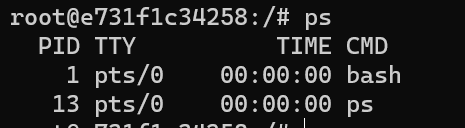
Now, if we echo $COLOR, we see result



**Managing Processes**:

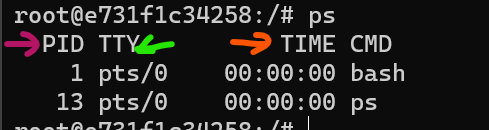
*A process is an instance of a running program*.

To see all the running programs or the processes, we can use the ps command.



Here we have two processes currently running, one is *bash* and other is *ps*. (*Technically ps is a very short lived process, it only exist while ps command produces this output*). So the only process which is running right now is *bash*.

*bash which is short for* ***‘****bourne again shell****’****, is a program that we are interacting with right now, which takes our commands and sends them to linux for execution*.



Each process has a unique identifier PID which is generated by the operating system.

**TTY** is short for teletype. Here *pts* shows the type of terminal the user is logged into (*pts is short for pseudo – terminal*).

Note: In pts / 0, /0 stands for first terminal window. So if we open this container in another terminal window simultaneously and run ps command we will see pts/1.

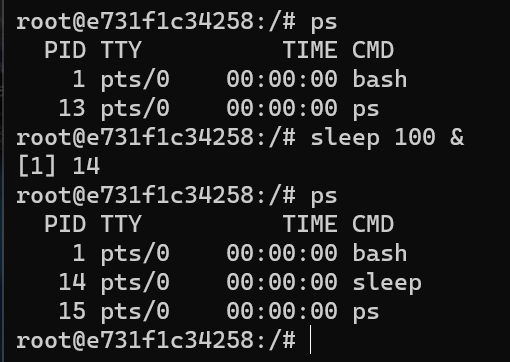
**TIME** is the amount of CPU time each process consumes. Since these processes are very light weight they are not taking much of CPU time.

In case we need to kill a process, we use the kill command.

But in order to see kill command in action, let’s create a demo process.

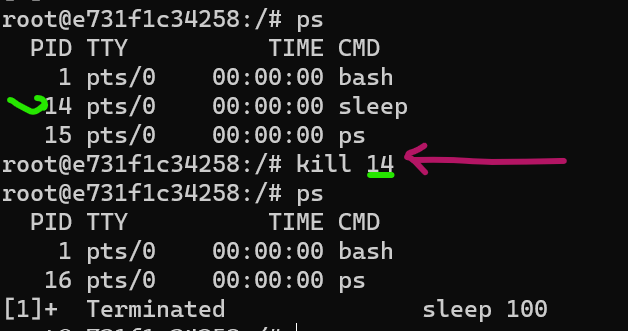
sleep 100 &

With sleep command we can make the terminal sleep for 100 seconds and with ampersand & this process runs in the background.



Now with kill <pid> we can kill the process which we referenced by process identifier.

kill 14



This process is terminated now.

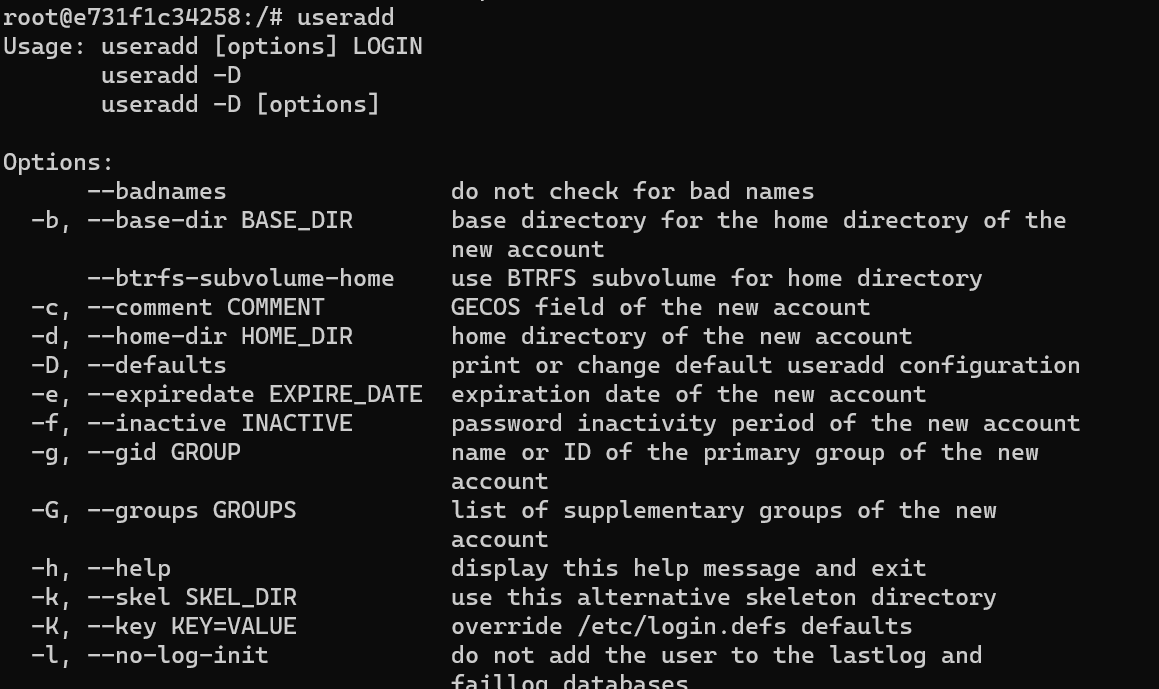
**Managing Users**:

In this lesson we will create a new user and then login as that user.

So in linux we have a command called useradd for *adding a new user*.

We also have usermod for *modifying a user* and userdel for *deleting a user*.

Let’s type useradd for now,



We see a bunch of options and none of them are mandatory, we can use them as per our needs.

Here we are going to use this option,



Note: Every option we see here, has two forms; *the short form with one hyphen -* and a *long descriptive form with two hyphens* --.

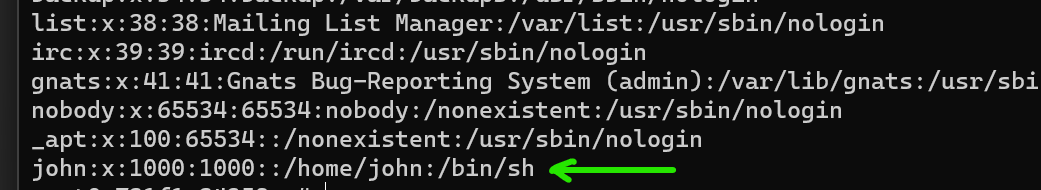
With this -m option we can create a home directory for this user.

useradd -m john

We created a new user called john which is *stored in a configuration file in the etc directory*.

To view this configuration file,

cat /etc/passwd



Note: In *passwd* we do not have user’s passwords but only their account information.

Here we have multiple fields separated by colon,

john:x:1000:1000::/home/john:/bin/sh

john 🡪 username

x 🡪 password is stored somewhere else

1000 🡪 user id

1000 🡪 group id

/home/john 🡪 home directory for this user.

/bin/sh 🡪 Is the shell program used when this user logs in.

Here /bin/sh represents old shell program, but we also have bash which is bourne again shell which is the enhanced version of this program.

Let’s say when we login instead of using shell we want to use bash, so we are going to use usermod command to modify this record.



Here with usermod we are going to use ***-s*** *option for setting the shell for this user*.

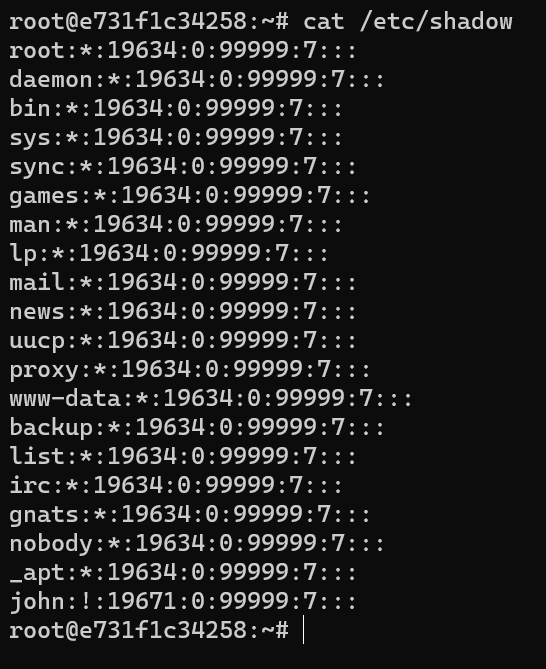
usermod -s /bin/bash john

And if look at /etc/passwd again,



To see the passwords, we have another file in the same directory called *shadow*.

cat /etc/shadow



Here passwords are stored in encrypted format. *This file is only accessible to the root user*.

Now let us say we want to login as our user john,

🡪 Open a new terminal and see the running container id’s



🡪 Then we execute a bash session inside this container using docker exec -it -u <user-name> <container-id> bash

Command breakdown:

🡪 docker

🡪 exec -- *to execute*

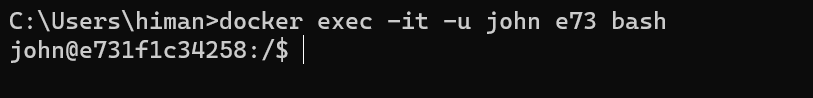
🡪 -it -- *interactive*

🡪 -u -- *supply username*

🡪 bash -- *to start a bash session*

In our case,

docker exec -it -u john e73 bash



Now we have logged in as john@e731f1c34258:/$ (*every syllable in this information is important*).

John 🡪 specifies username

@e731f1c34258 🡪 container-id which acts as a hostname

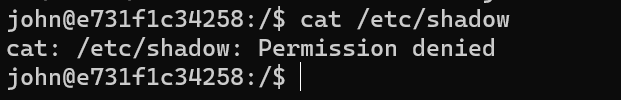
$ 🡪 This sign indicates user is not root, just a regular user.

In other window where we are logged in as root user,



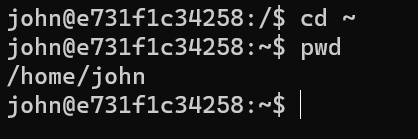
In root@e731f1c34258:~# we have a pound/hash symbol. So we have extra privileges.

Now back to john’s window. Let us see if we can access *shadow* file.



We get permission denied, so it proves that we are not a root user.

Also john has a home directory, so if we cd ~



We go to john’s home directory and if we pwd, we see path to this home directory (*/home/john*). So in this directory we can store john’s files and when we are done we can remove this by userdel john



For now, we will not delete this user because in the next lesson, we will add john to a group. So we are going to keep him for now.

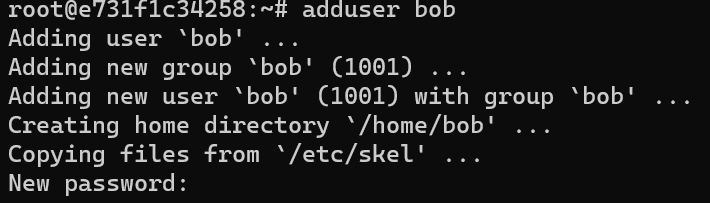
Note: We talked about useradd command, now we also have another command which is adduser. What’s the difference?

*useradd* 🡪 It’s the original API that was built.

*adduser* 🡪 It’s the Perl script which is more interactive and uses *useradd* under the hood.

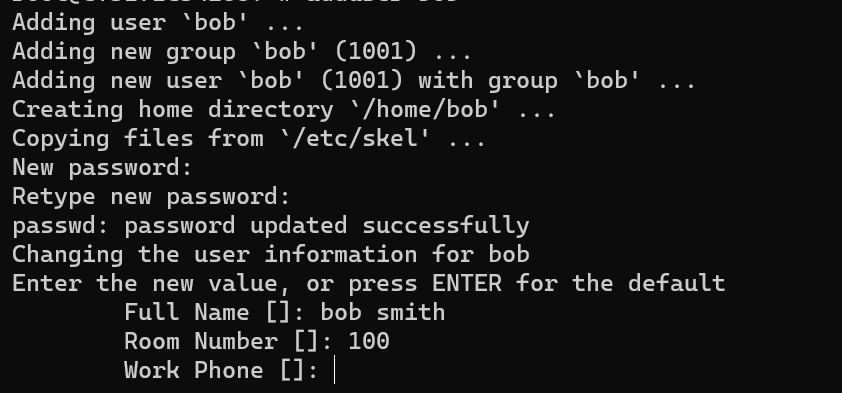
Let’s look at adduser real quick,

adduser bob



We see a few messages like adding user, adding group (*every user that is created is automatically placed inside a group with the same name*), creating home directory for this user etc.

But one new thing we see is a chance to create password for this user. This is what we meant about adduser command being more interactive.



After typing passwords, we see more input fields here. *Quite often when using docker for deploying our application we do not want to use adduser* because we do not want too much interaction with this command.

**Managing Groups**:

So we created a new user in the last lesson, now let’s talk about managing groups.

Similar to useradd, usermod and userdel, we have similar commands for managing groups which are *groupadd*, *groupmod* and *groupdel*.

Let’s add a new group called *developers*.

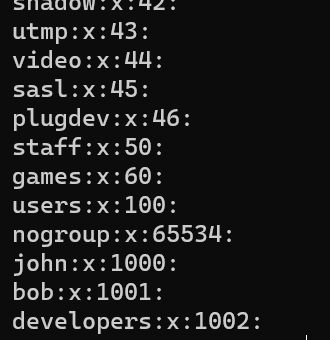
groupadd developers



Note: We use groups so that each user in a group have same type of permission.

To *locate this group*, we need to find a file called *group* inside *etc* directory.

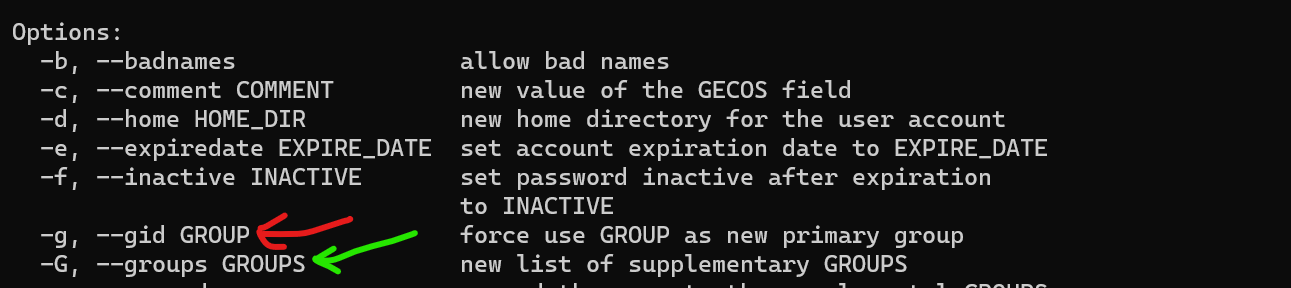
cat /etc/group



Here we see our *developers* group and 1002 is its ID.

Now if we want *a user to add in this group*, we will use usermod command.

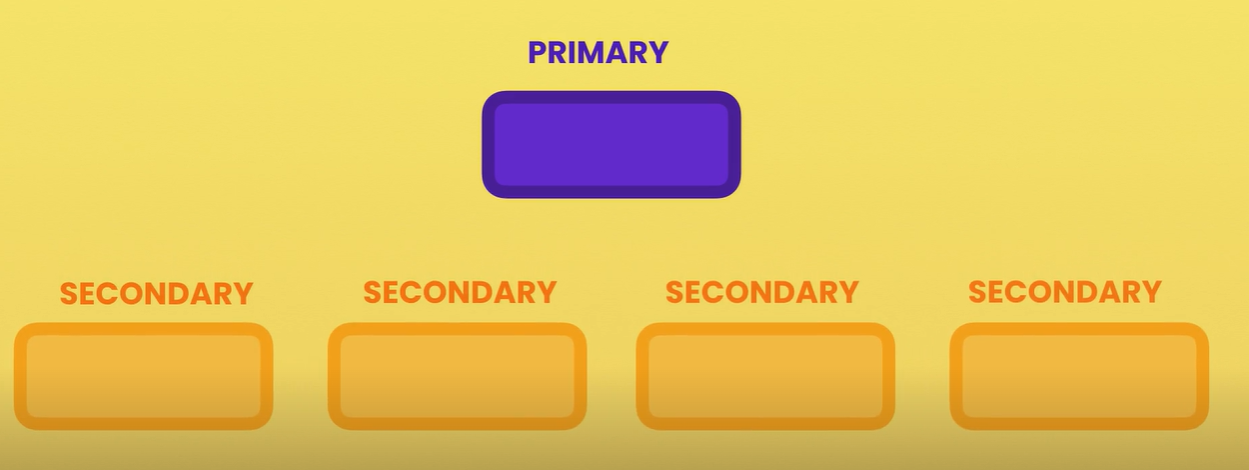
This command has an option for this purpose,



With -G option we can set the supplementary groups for this user but with -g we can set primary group.

*What’s the difference*?

🡪 Every linux user has one *primary* group and zero or more *supplementary* groups.



Suppose john is part of five groups and he wants to create a new file. Every file is owned by one user and one group.

Since john is part of five groups, which groups we use for owning the file that john is going to create? That’s why we need *primary* group (*gets automatically created when we create a new user,* ***group with same name as the user***).

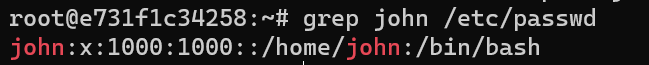
In this example we will use -G for setting supplementary groups for john.

usermod -G developers john

or usermod -G 1001 john (*here 1001 is group id we want to assign*)

If we *grep* john in our passwd file,

grep john /etc/passwd



This is the user id and primary group id for john.

If we want to *see all the groups of a specific user*, we can use *groups* command.

groups john



So john is currently part of two groups. One is john(*primary group*) and other is developers (*supplementary* *group*).

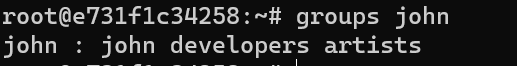
*Exercise* :

Add john to a new group called artists.

groupadd artists

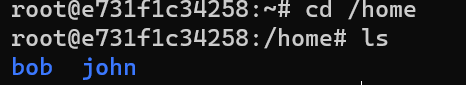
----------------------------------------------

usermod -G developers,artists john



**File Permissions**:

Here we have logged in as root and in our home directory.

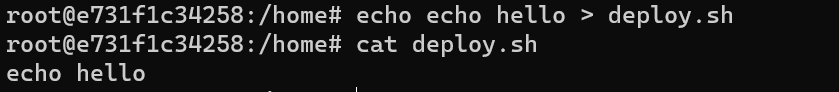


Here we will create a file called *deploy.sh*

Note: Files with .sh extension are called shell scripts and in this file we can write any of the linux commands we have learned so far. So *we can combine all these commands and create a deployment script*.

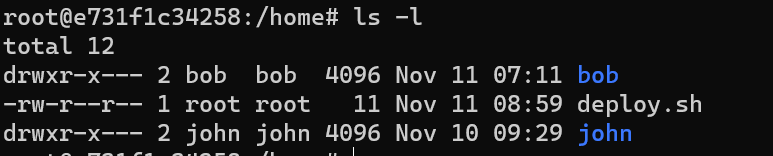
Let us write *echo hello* to our deploy.sh file.

echo echo hello > deploy.sh



So when we execute this file, we are going to see hello on our terminal.

To see the permissions for this file, we need to use *long listing* (ls -l)



In this directory we have two directories john, bob and a single file deploy.sh

In the left hand side, we see the permission for each item.

drwxr-x--- (*first letter is d means it is a directory*)

-rw-r--r--(*first letter is hyphen so it’s a file*)

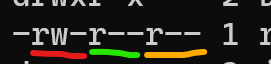
All these letters are 9 letters divided into 3 groups. For example,

rw- || r-- || r--

r (*read*), w (*write*) and x (*execute*) permissions. Notice in above example we have read and write permissions but execution permission is missing (*hyphen - in its place*).

Note: All directories by default have execute permission. It is because so that we can cd inside that directory.

We understand the permissions, but these are divided into three groups. What are these groups?



🡪 First group represents the permissions for user that owns this file.

For example, here…

-rw-r--r-- 1 **root** **root** 11 Nov 11 08:59 deploy.sh

Here it’s the root user.

🡪 Second group represents the group that own this file, which is also root group in this example.

By default every user that has been created is automatically placed inside a group with the same name.

🡪 Third group represents everyone else.

To execute this file, we need to type ./deploy.sh



We get a permission error because even the root user do not have permission to execute this file.

This is where we use *change mode* (chmod) command.

chmod u (*for changing permission for user)*

chmod g (*for changing permission for group*)

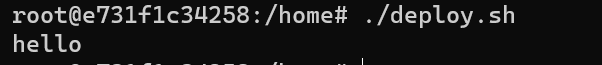
chmod o (*for changing permission for others*)

Here for the user we want to *add the execute permission* so we type chmod u+x (*if want to remove execution permission use* chmod u-x) followed by the file name.

chmod u+x deploy.sh

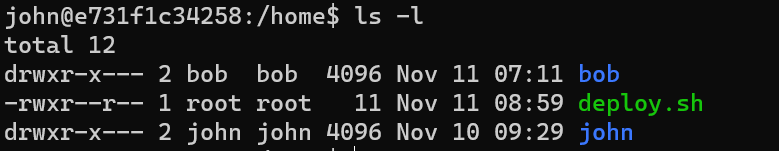


Notice the colour of our file, it turned green because now it is executable and we have rwxr--r-- (*read – write – execute permissions*).

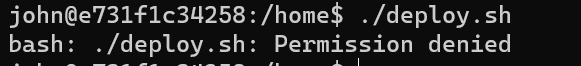


Now we can call our shell script and see hello on the terminal.

With this setup, only root user can execute this file. So if we go to another terminal window where we are logged in as john.



And try to execute this file,



We get permission error.

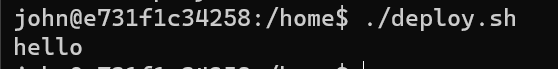
*So how do we enable john to execute this file*?

chmod o+x deploy.sh

Here we enable execution permission for others in this file.



Now back to john’s window,



This is how we can change permissions for a file.

Note: We can combine multiple permissions in this command like this , chmod og+x+w-r deploy.sh

Here we are combining others + group + execute + write – read (*removing read permission*)