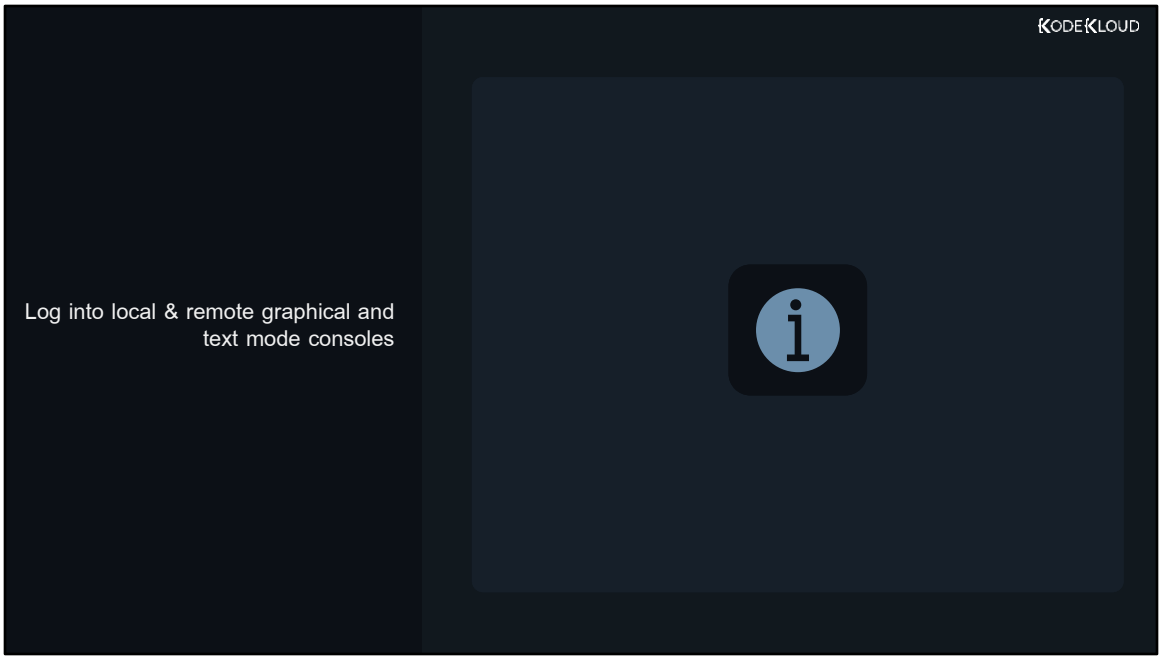
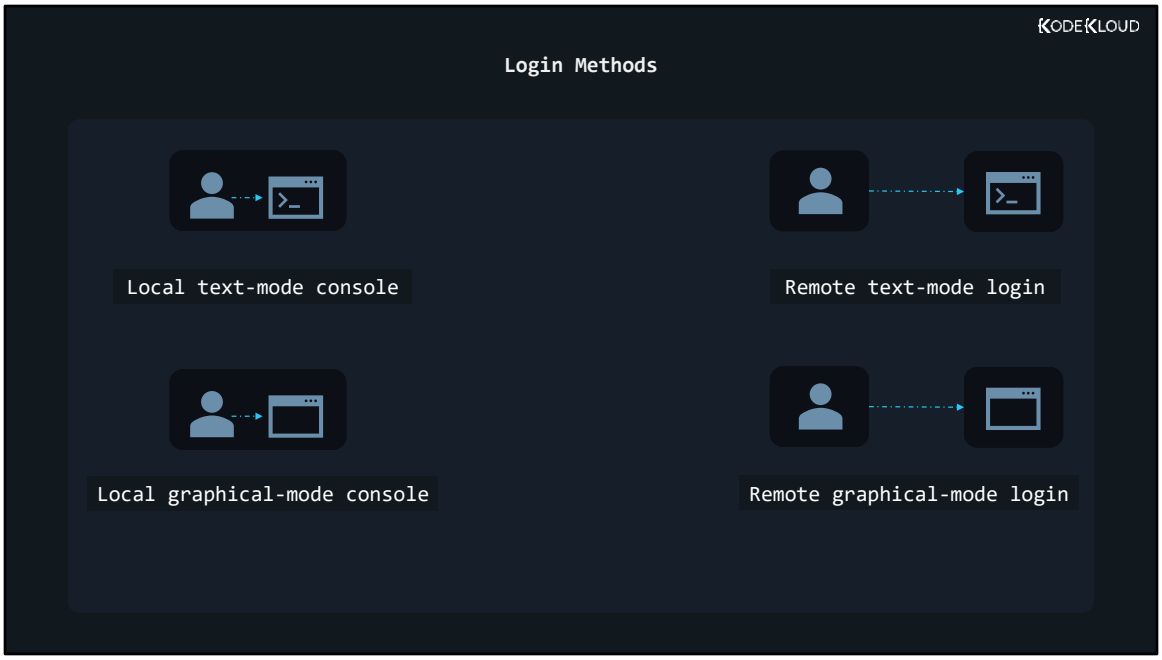

KodeKloud

Access the labs associated with this course using this link: https://kode.wiki/linux-labs



There will be many commands we will use in Linux. And each command has a lot of command line switches. How are we supposed to remember them all?

As we use a command repeatedly, we'll learn everything about it and memorize what each option does. But in the beginning, we might forget about these options after just one or two uses. That's why Linux gives you multiple ways to access "help manuals" and documentation, right at the command line.

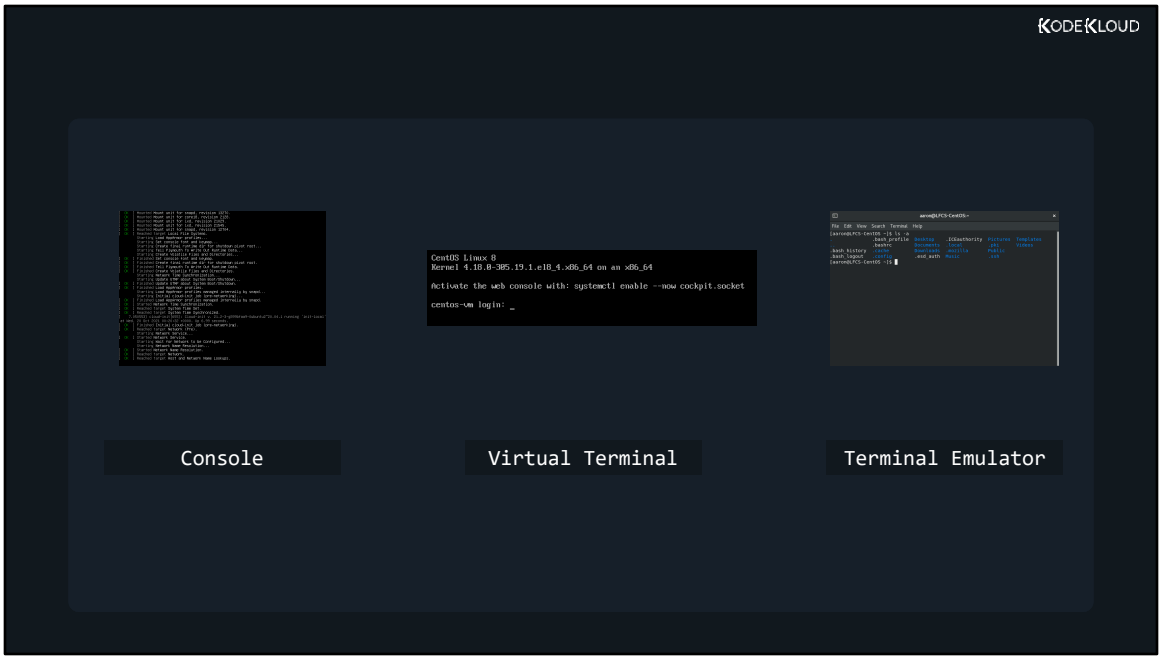


So, let's dive right in and start with some simple concepts.

We're all used to logging in to apps or websites by providing a username and password. Logging into a Linux system is pretty much the same, so there's not much mystery here. We'll look at four ways to log in:

1.Log into a local Linux system (local text-mode console).
2.Log into a local Linux system (local graphical-mode console).

3. Log into a remote Linux system (text-mode login)
4.Log into a remote Linux system (graphical-mode login)



You'll often hear terms like "console", "virtual terminal", and "terminal emulator". And it may be hard to understand why they are called this way.

Nowadays, a "console" is just a screen where your operating system displays some text and where it allows you to log in or type commands. And a terminal emulator is nothing more than a graphical app that runs in a window and does a similar thing (shows you text output and allows you to type commands). These terms come from the old days of computing.

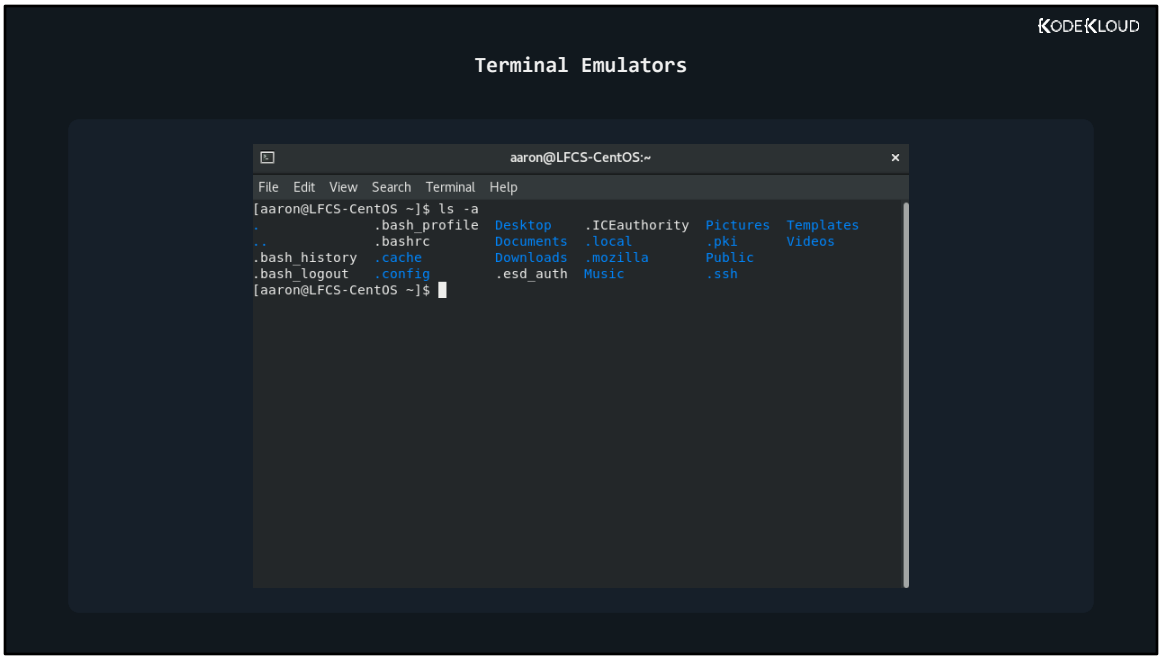
Computers were incredibly expensive, so a university may have had a single one available for their entire building.

But multiple people could connect to it and do their work by using physical devices that allowed them to type text and commands and also display on a screen what is currently happening. These devices were consoles or terminals. So instead of buying 25 super expensive computers, you could have just one, but 25 people could use it, even at the same time.

Nowadays, consoles and terminals, in Linux, are usually things that exist in software, rather than hardware. For example:

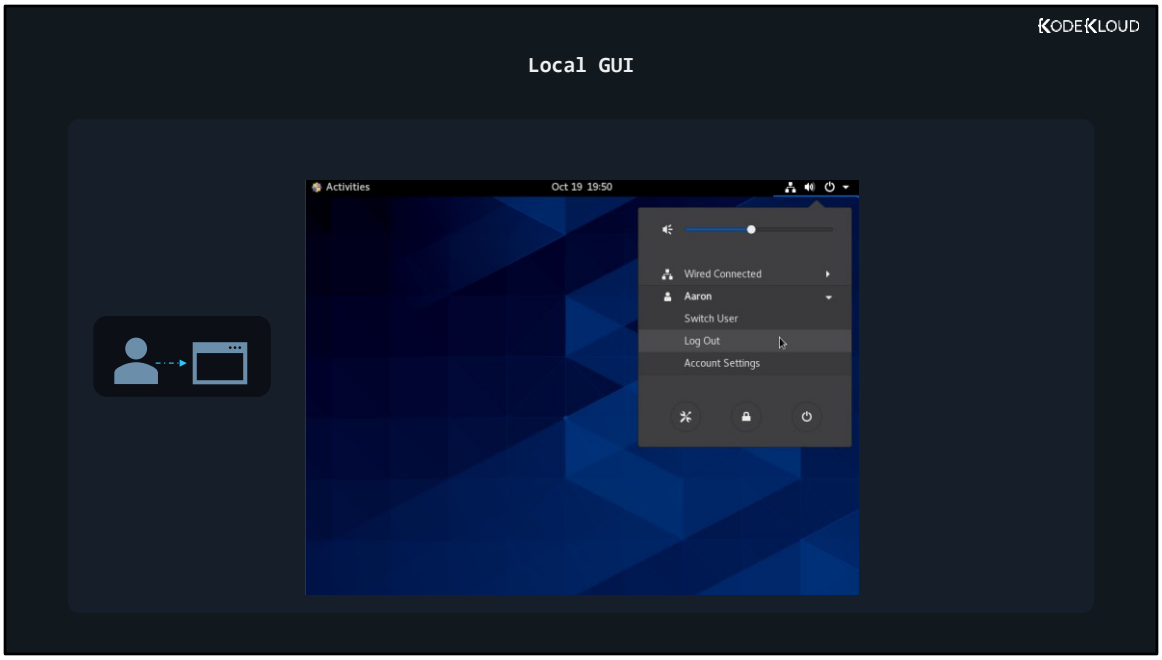
- When you see Linux boot and a bunch of text appears on screen, telling you what happens as the operating system is loading - that's the **console**.

After a Linux machine has booted, if you press CTRL+ALT+F2 on the keyboard, you'll see a **virtual terminal** (**vt2**).



If you have Linux installed on your desktop, with a graphical user interface, when you want to type commands you open a **terminal emulator**.

Let's move back to logins. In practice, most often you'll log in to remote Linux systems. But let's start with the less common scenarios.

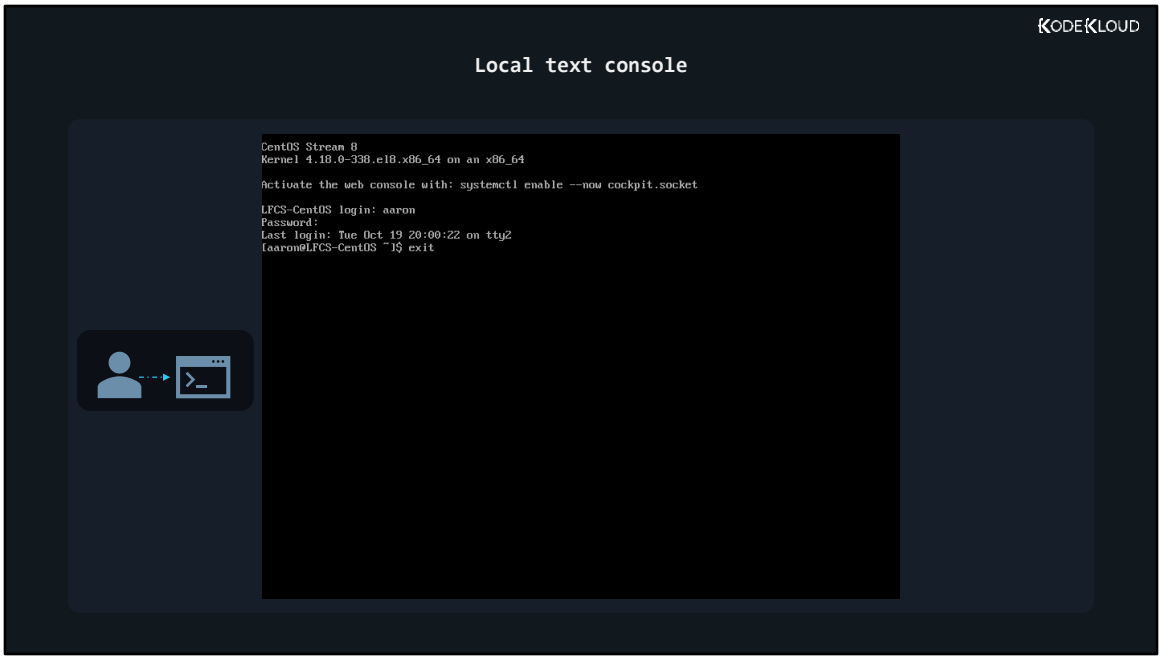


"Local" is just a tech word for "something that is in front of you" or "something you can physically access". A computer on your desk is **local**. A server running on Google Cloud is **remote**.

Usually, when Linux is installed on servers, it is installed without GUI (Graphical User Interface) components. There's no mouse pointer, no buttons, no windows, no menus, nothing of that sort, just text. But you might sometimes run across servers that include this GUI. Logging in is super easy, as it's all "in your face". You'll see a list of users you can choose from and you can then type your user's

password.

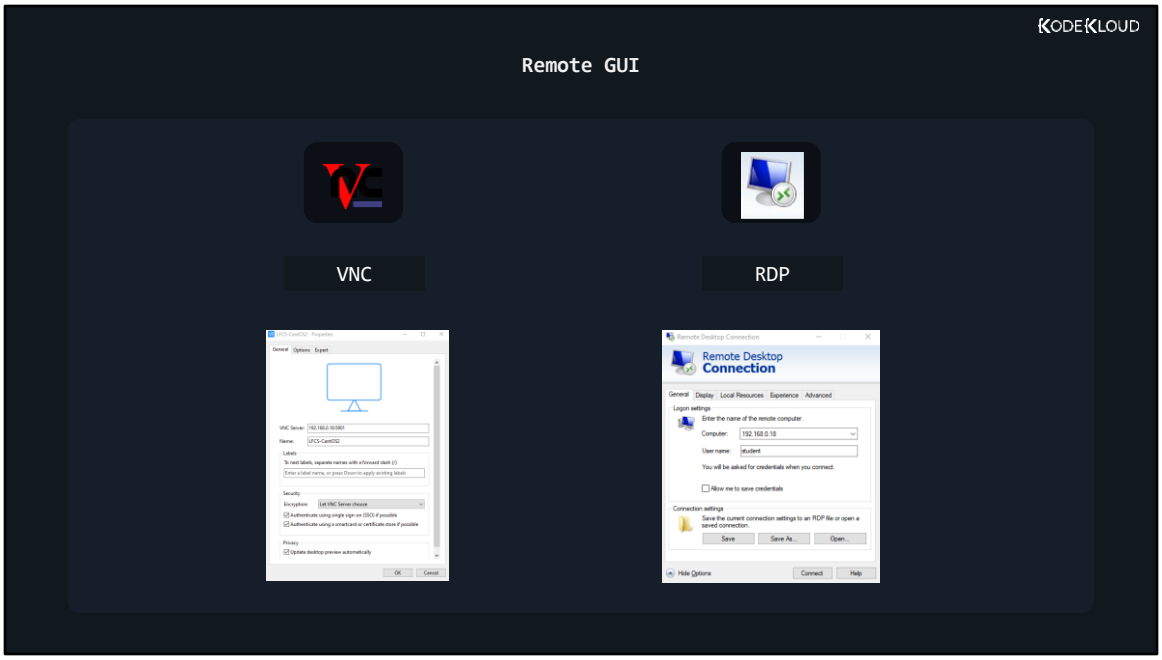
Don't forget to log out when you've finished your work.



If the device has the typical server-oriented Linux OS installed, without any GUI components, logging in (locally) is also easy. You'll usually see something like this on your screen:

There's no list of users this time, but you can just type your username and then your password. Note that you won’t see your password as you type.

When your work is done, you should type **exit** to log out.

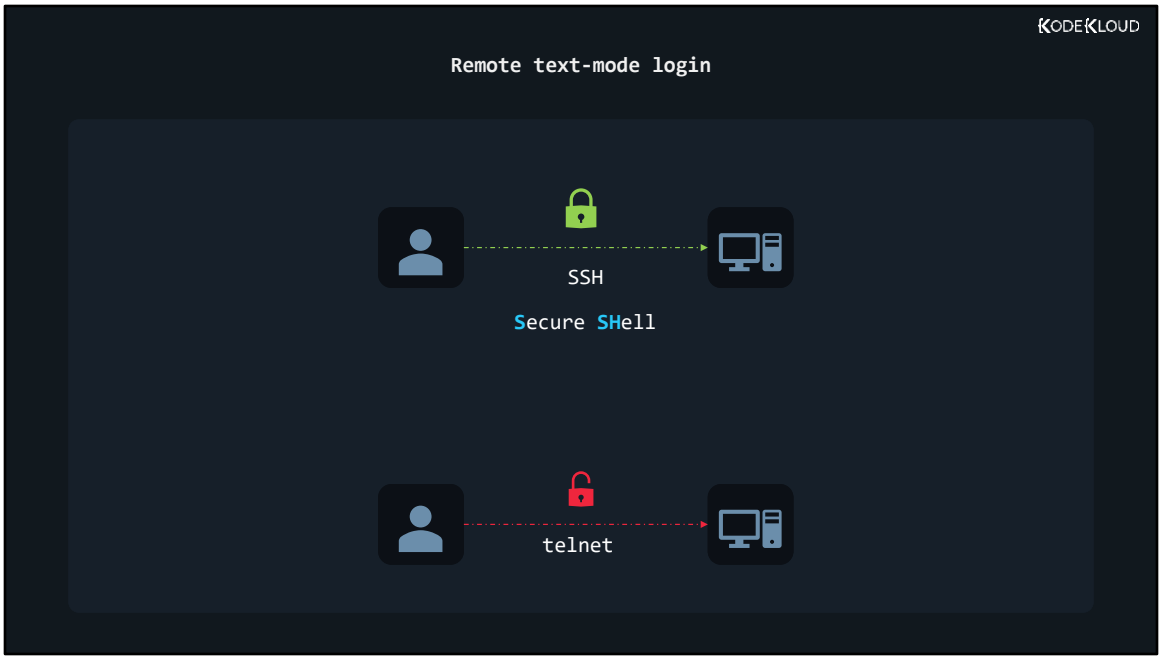


Again, most Linux operating systems running on servers will have no GUI components installed. But you will sometimes run into exceptions. Connecting to a remote server, to its graphical user interface is slightly more tricky. First of all, there is no standard set in stone. Whoever configured that server chose their preferred way of dealing with these remote graphical logins. They could have chosen to install a VNC (Virtual Network Computing) solution. In this case, you'd need to download the proper VNC client (also called "VNC viewer") to connect to it. This might be TightVNC or RealVNC or something else

entirely. It all depends on the VNC server running on the remote system and what VNC clients your local operating system supports.

If the administrator of that server wanted to let Windows users connect easily, it might mean that they used a solution allowing for RDP connections (Remote Desktop Protocol). This means you can just click on Windows' start button, type "Remote Desktop Connection", open that app and then enter the username and password you've been provided.

Whatever it might be, connecting to a remote graphical console is pretty easy. It all boils down to downloading the application that lets you do that, entering the remote system's IP address, followed by an username and a password.



Initiating a text-based remote connection to a Linux system is pretty standard. That's because almost every Linux server uses the same tool that allows for remote logins: the OpenSSH daemon (program that runs in the background, on the server, all the time). SSH comes from **S**ecure **SH**ell. Until SSH, something called telnet was the standard. telnet was highly insecure as it did not encrypt communication between you and the server you were connecting to. This meant that anyone on the same network with you could steal your Linux user password and see everything you did on that server, during your telnet

session.

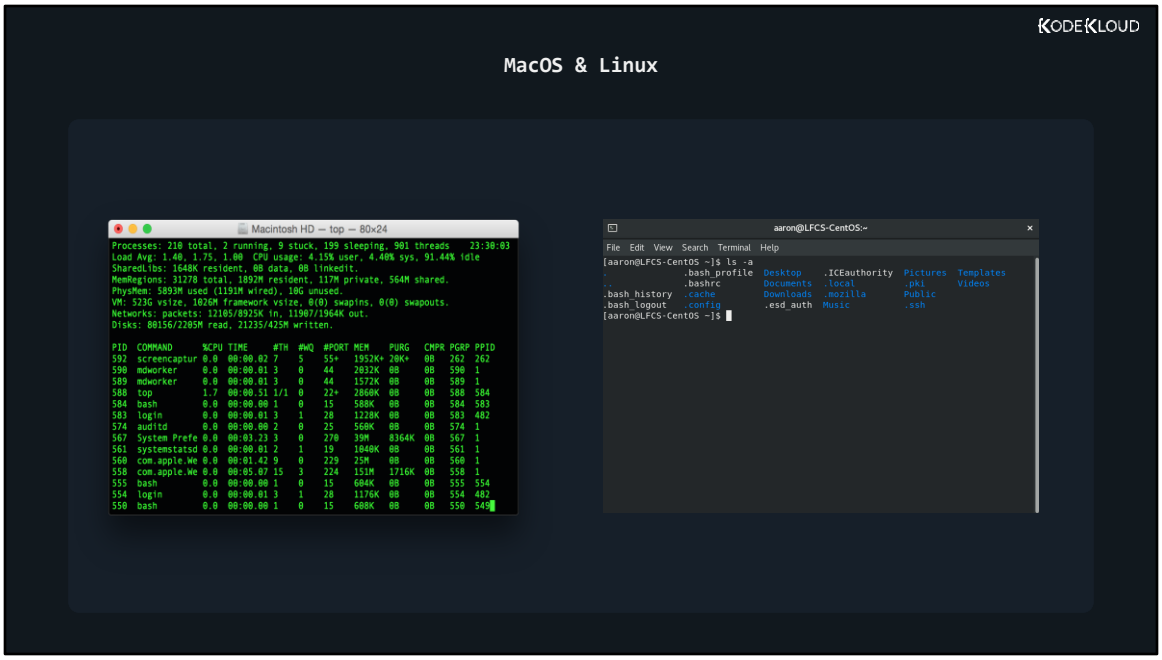
The SSH protocol uses strong encryption to avoid this and the OpenSSH daemon is built carefully to avoid security bugs as much as possible. Long story short, OpenSSH is used by millions of servers and has stood the test of time, proving to be very hard to hack. For these reasons everyone happily uses it and trusts that it can do a pretty good job at only letting authorized people log into their operating systems, while keeping bad people out.

In case you're following along on your virtual machine, log in locally (directly from the virtual machine window) and then enter this command: (**ip a**) You'll see what IP your machine uses. I've outlined the information we're looking for in yellow.

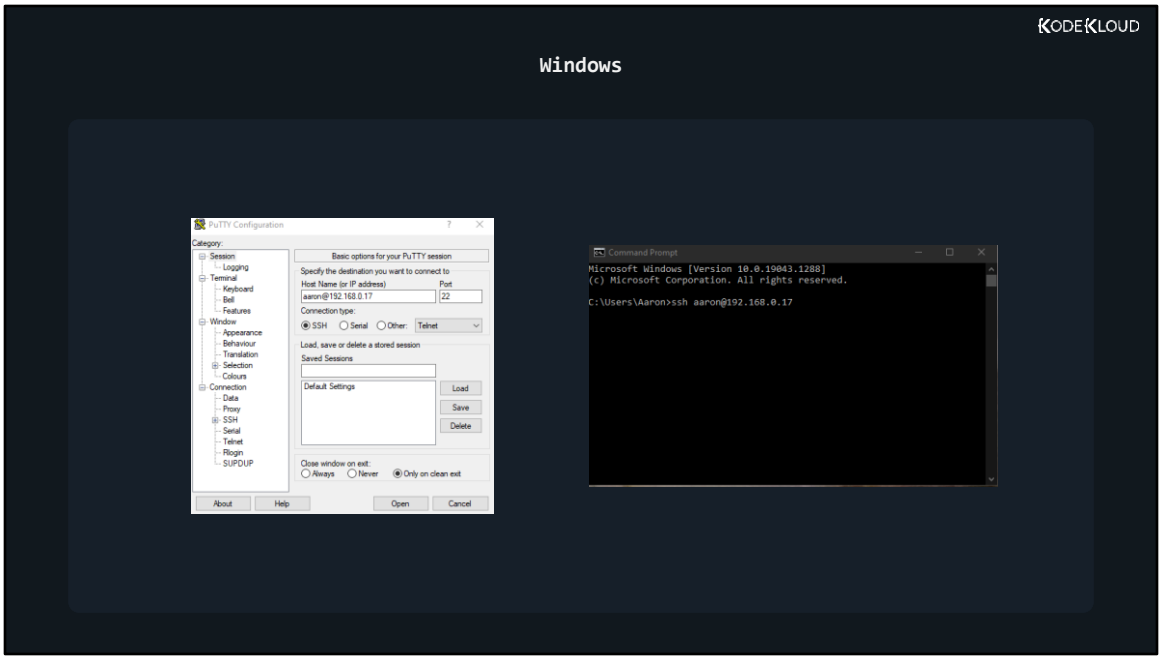
We'll use this IP – in our case 192.168.0.17 -- to simulate a situation where we have a server in a remote location.

Now to recap. We have an SSH daemon (program)

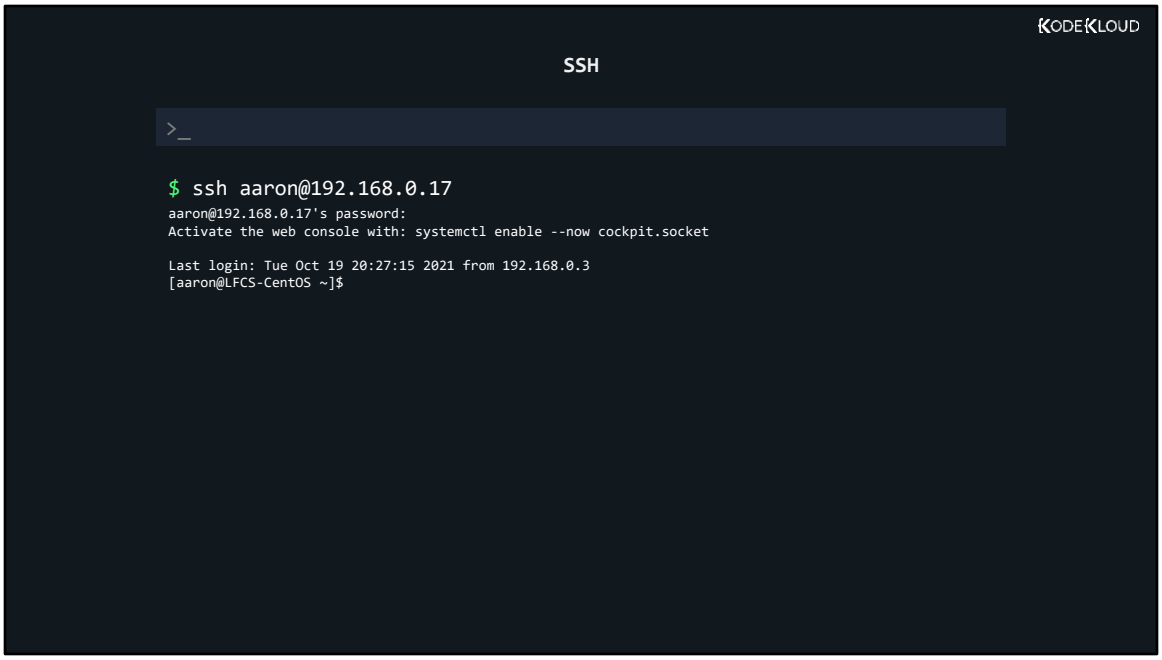
running on the server. This listens for any incoming connections. To be able to connect to this remote SSH daemon, we'll need something called an SSH client (yet another program). This client will run on our current laptop/desktop computer.



MacOS systems and Linux-based operating systems, such as Ubuntu, already have an SSH client preinstalled. If you're on MacOS or Linux, open a terminal emulator window.



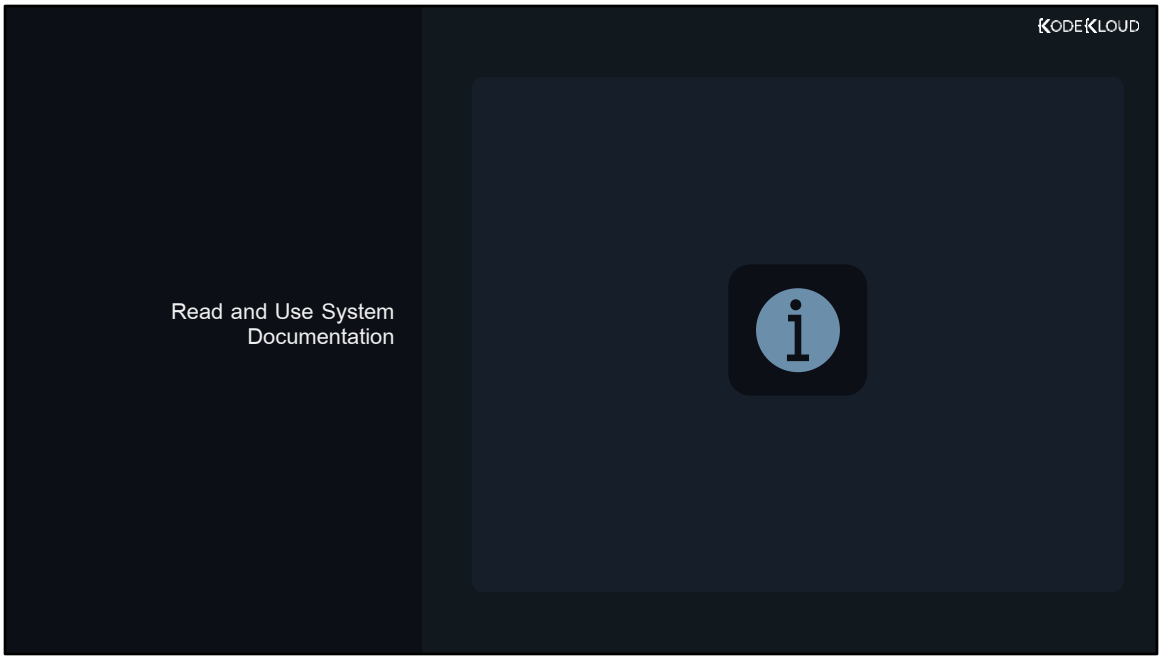
In the past, if you were running Windows, you needed to install an SSH client like PuTTY. On the latest Windows 10 this is no longer necessary as an SSH client is also preinstalled. If you're on Windows, click the Start Menu and type "cmd" to open up Command Prompt.



To connect to a remote Linux system through SSH, type:

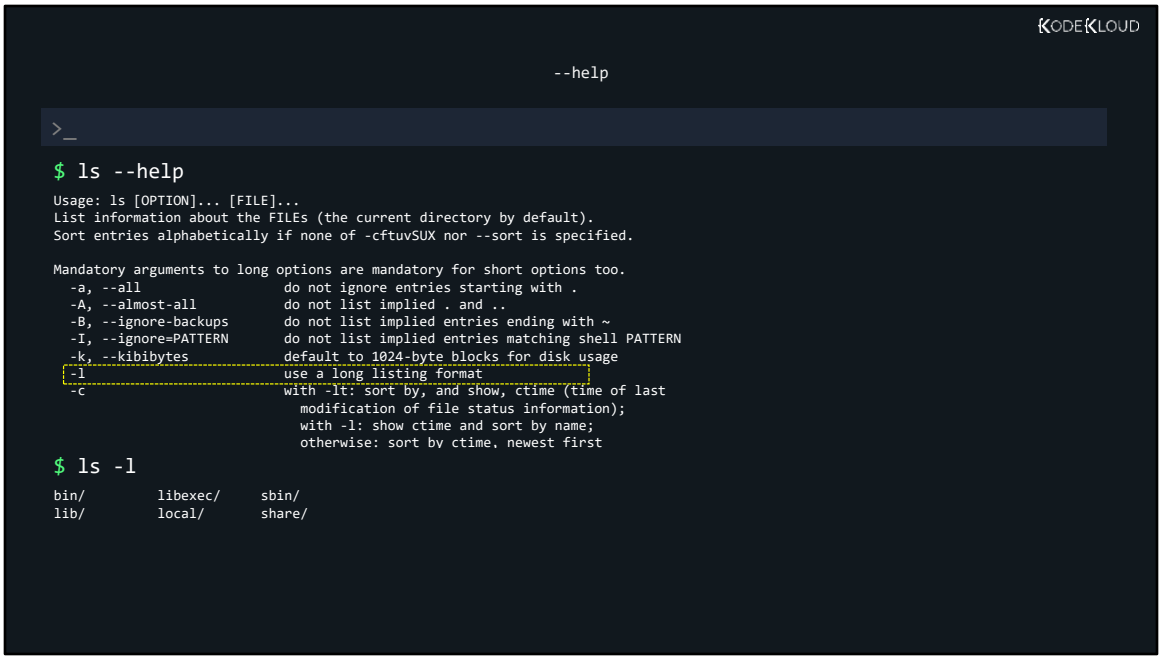
Of course, replace "aaron" with the actual username you created inside your Linux OS running in the virtual machine. Same with the IP address. From here on, we'll stay inside this SSH session to go through all the exercises in the upcoming lessons. Please join me in the demonstration video to see each of these login methods. I'll see you there.

Access the labs associated with this course using this link: https://kode.wiki/linux-labs



There will be many commands we will use in Linux. And each command has a lot of command line switches. How are we supposed to remember them all?

As we use a command repeatedly, we'll learn everything about it and memorize what each option does. But in the beginning, we might forget about these options after just one or two uses. That's why Linux gives you multiple ways to access "help manuals" and documentation, right at the command line.

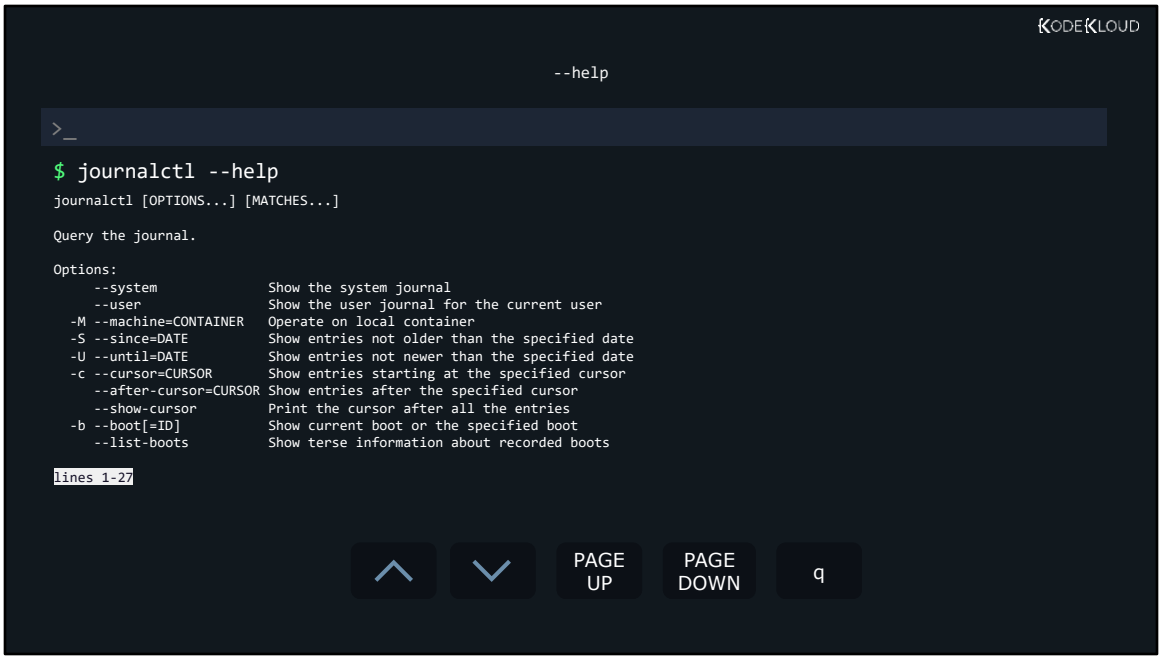


Let's say you want to see that long listing format with ls, to get a look at file permissions. But you **forgot** what the correct option was. Was it -p for permissions? We can get a quick reminder with:

ls --help

This will show us **a lot** of output. But if we scroll up, we'll find what we're looking for: **the –l flag**, in this case.

You can see how command line options are sorted **alphabetically** and **described with short text**. That's why the **--help** option for commands will **very often** be helpful when we forget about these options (and we will, as there are **so many** of them for **each command**).



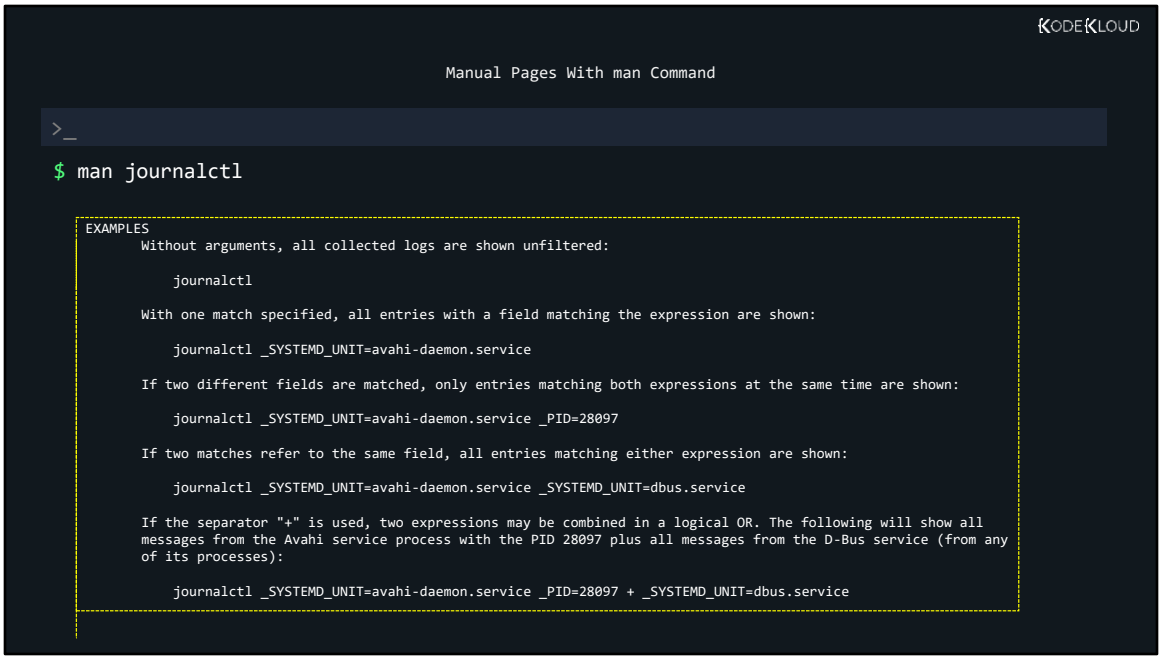
--help will usually show a condensed form of help, with very short explanations. For ls, that's ok, as it's a very simple command. Other commands, however, are very complex and we need to read longer explanations to understand what they do and how we use them.

Let's take journalctl as an example, a command that lets us read system logs.

journalctl --help

will show us this:

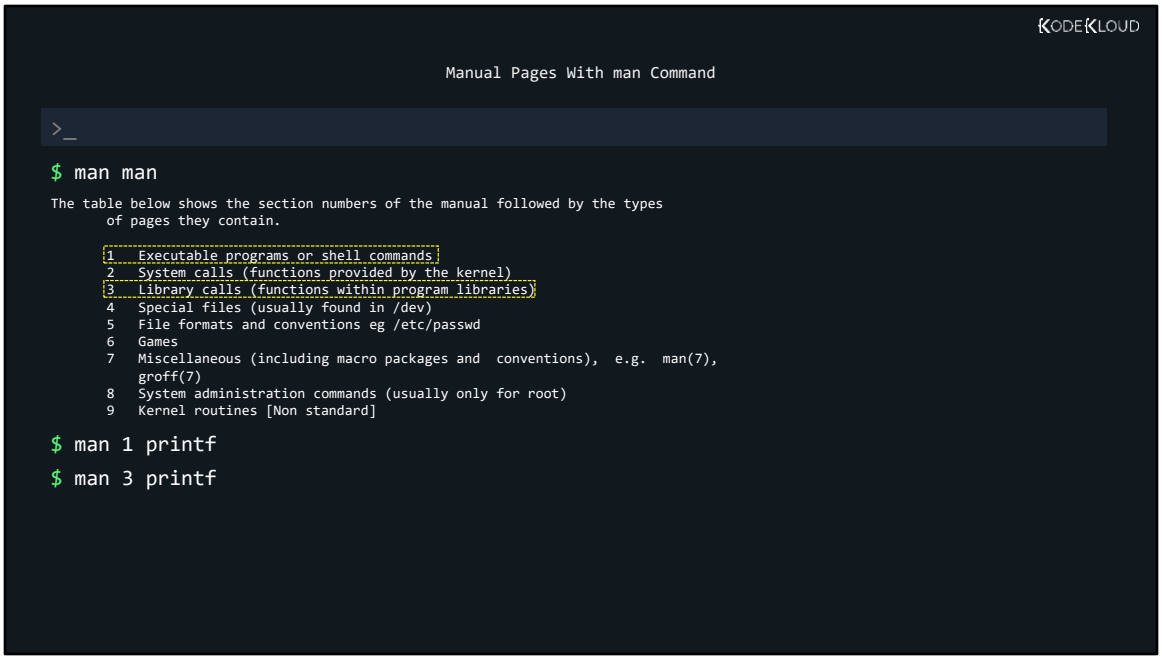
We'll notice that this opens in a slightly different way (look at "lines 1-27") in the bottom left corner. This opened in what Linux calls a "pager". It's simply a "text viewer" of sorts that lets us scroll up and down with our arrow keys or PAGE UP, PAGE DOWN. To exit this help page, press q.



All important commands in Linux have their own manuals or "man pages". To access a command's manual enter "man name_of_command". In our case, we'd use:

man journalctl

Now we get:
•Short description of what the command does in NAME.
•General syntax of command in SYNOPSIS
•Detailed description of command, how it works, and so on, in DESCRIPTION.
•Detailed descriptions of command line options in OPTIONS.
•And some manual pages even have some EXAMPLES near the end of the manual.



Sometimes, you will have two man pages with the same name. Example:

printf is a command. But **printf** is also a function that can be used by programmers.

Manual pages can fall into one of these categories (sections), and we can see these by looking at the man page for man itself, by typing man man:

If you want to read the man page about printf, **the command**, you tell man you want to consult printf from section 1, like this

man 1 printf

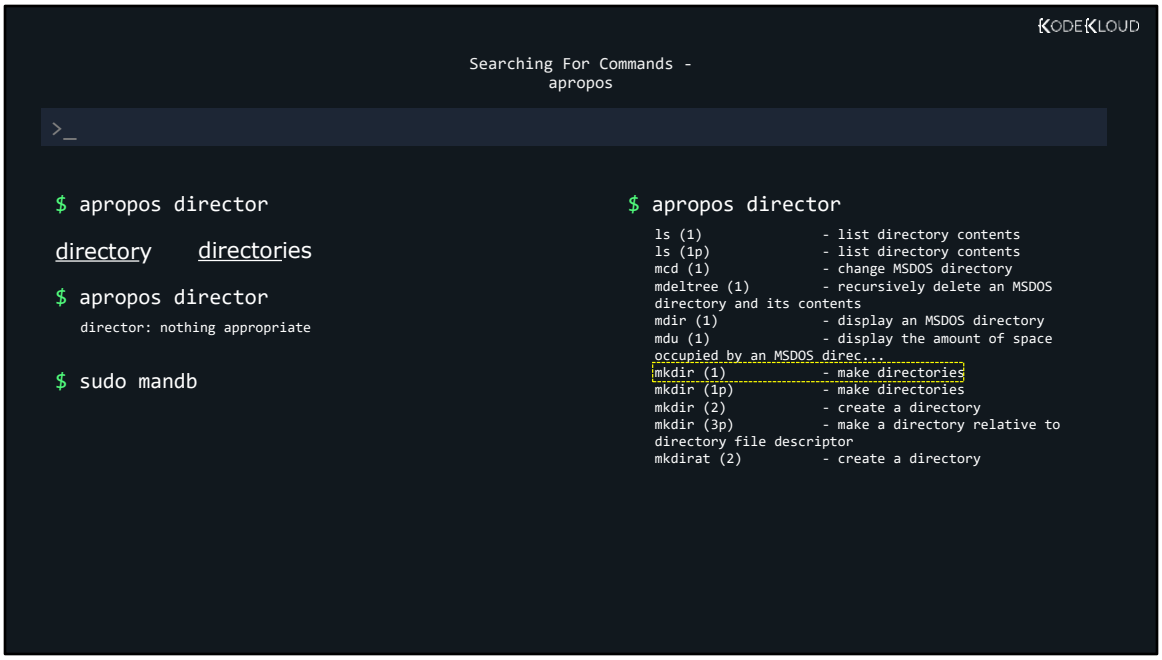
If you want to read about printf, **the function**, you tell man you want to look at section 3

man 3 printf

It's useful to know that during online exams, the Linux Foundation will let you

use man and --help. Try to use --help if you forgot a command line option as that gives you the fastest results. Diving deep into a manual page will eat up more time.

But this is all well and good when we know what command we want to explore. But what if we can't even remember the name of the command that we need to use?



Imagine you forgot the name of the command that lets you create a new directory. How would you search for it?

apropos is a command that lets you search through man pages. It looks at the short descriptions of each man page and tries to see if it matches the text we entered. For example, with the next line we can search for all man pages that have the word "director" in their short descriptions. We'll use "director" and not "directory". "director" will match commands that contain the word "**director**y" but

also the ones that contain "**director**ies". So, we keep it more generic this way.

The first time we would run apropos director, we'd get an error.

That's because apropos relies on a database. A program must refresh it periodically. Since we just started this virtual machine, the database hasn't been created yet. We can create it manually with:

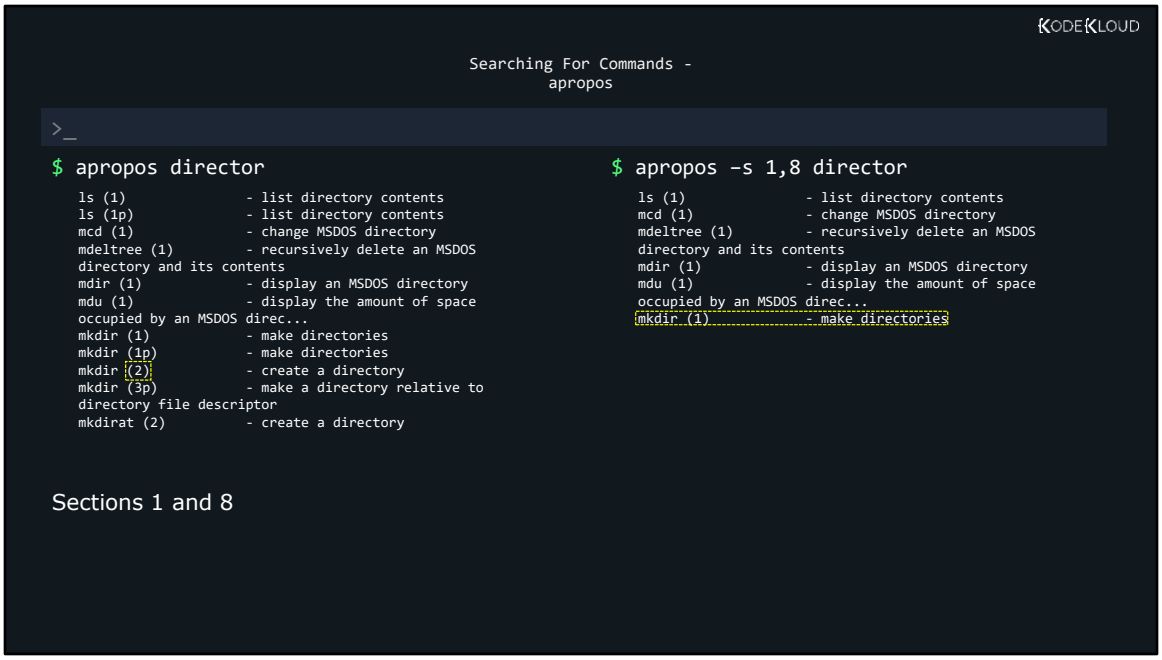
sudo mandb

On servers that have already run for days, there should be no need to do this, as it will be done automatically.

Now the apropos command should work:

apropos director

If we scroll up, we can see the entry we're looking for: mkdir.



But those are a lot of entries. Makes it hard to spot what we're looking for. You see, apropos doesn't just list commands. It also lists some other things we don't need, currently. We see stuff like (2). That signals that that entry is in section 2 of the man pages (system calls provided by the Linux kernel). That's just too advanced for our purposes. Commands will be found in sections 1 and 8. We can tell apropos to only filter out results that lead to commands from these categories. We do this by using the -s option, followed by a list of the sections we need.

apropos -s 1,8 director

And we can spot what we were looking for more easily.

Notice how mkdir's description contains the word "**director**ies". If we'd used the word "directory" in our apropos search, this command wouldn't have appeared since "directory" wouldn't have matched "directories". This is something to keep in mind when you want to make your searches as open as possible and match more stuff.

Another thing that'll save a lot of time is autocompletion. Type

systemc

press TAB

you get:

systemctl

Although this is not technically system documentation, it can still be helpful. Many commands have suggestions on what you can type next. For example, try this. Type

systemctl

add a space after the command (don't press ENTER) and now press TAB twice.

You get a huge list of suggestions. This can help you figure out what your options for that command are. Although you should not always rely on it. It's not necessary that absolutely all options are included in this suggestion list.

now add to that:

systemctl **list-dep**

press TAB

endencies will get added at the end and you get: systemctl list-dep**endencies.** This is TAB autocompletion and many commands support it. When you press TAB once, if your command interpreter can figure out what you want to do, it will automatically fill in the letters. If there are many autocomplete options and it can't figure out which one you want, press TAB again and it will show the list of suggestions we observed earlier. These will be huge timesavers in the long-run, and they might even help you in the exam, to shave off a few seconds here and there, which might add up and let you explore an extra question or two.

TAB suggestions and autocompletions also work for filenames or directory names. Try

ls /u **TAB**

ls /usr/ **TAB TAB**

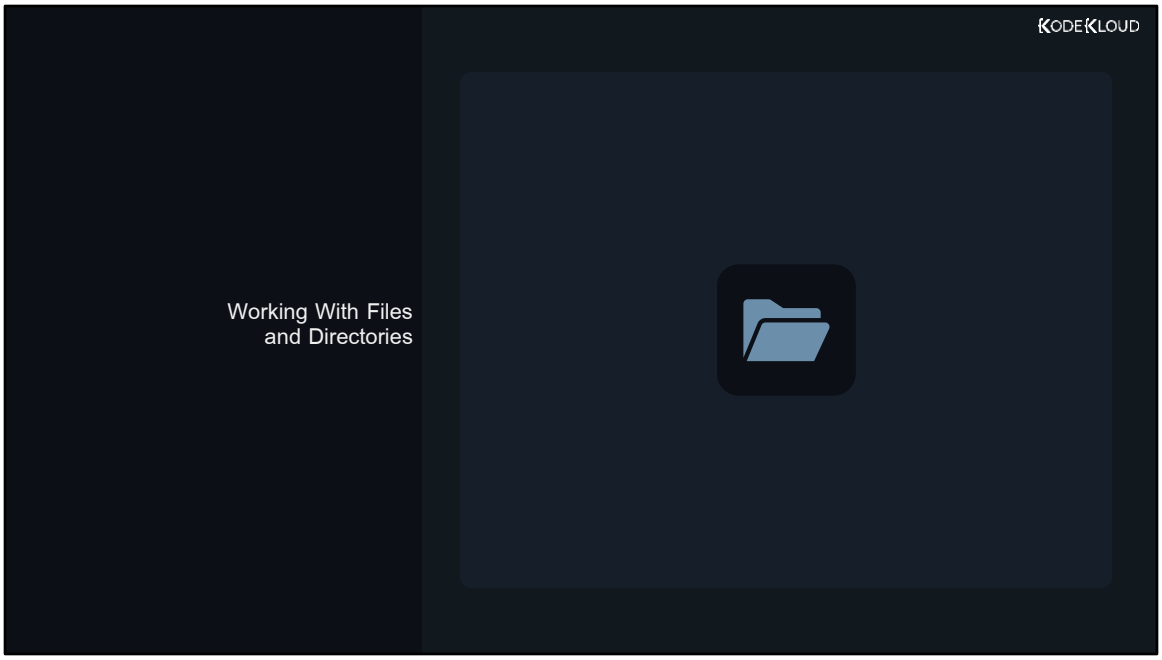
Now we can see directories available in /usr/ without even needing to explore this directory with "ls" beforehand. And if we have a long filename like "wordpress_archive.tgz" we might be able to just type "wor", press TAB and that long name will be autocompleted.

Recommendation

While manuals and --help pages are super useful, the first few times you use them, it might be hard to figure out how to do something, with that info alone. We recommend you take a command you know nothing about and try to figure out with just man and --help, how to do something. This practice will help you

develop the ability to quickly look for help when you're taking the LFCS exam. There will be questions about theory you either don't know about, or you just forgot. If you know how to quickly figure out the answer with a man page or --help, you'll be able to pass the exam much more easily.

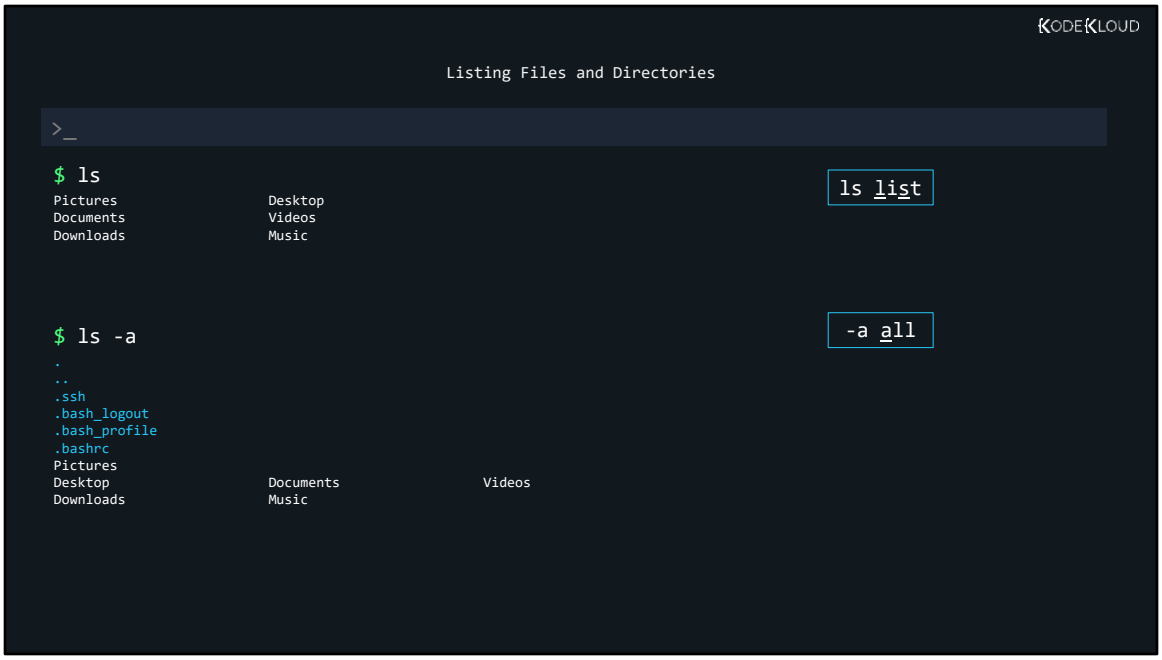
Access the labs associated with this course using this link: https://kode.wiki/linux-labs



Now we'll look at how to create, delete, copy, and move files and directories in Linux.

Before we dive into this lesson, we need to understand a few basic things:

1. What is a filesystem tree?
2. What is an absolute path?
3. What is a relative path?

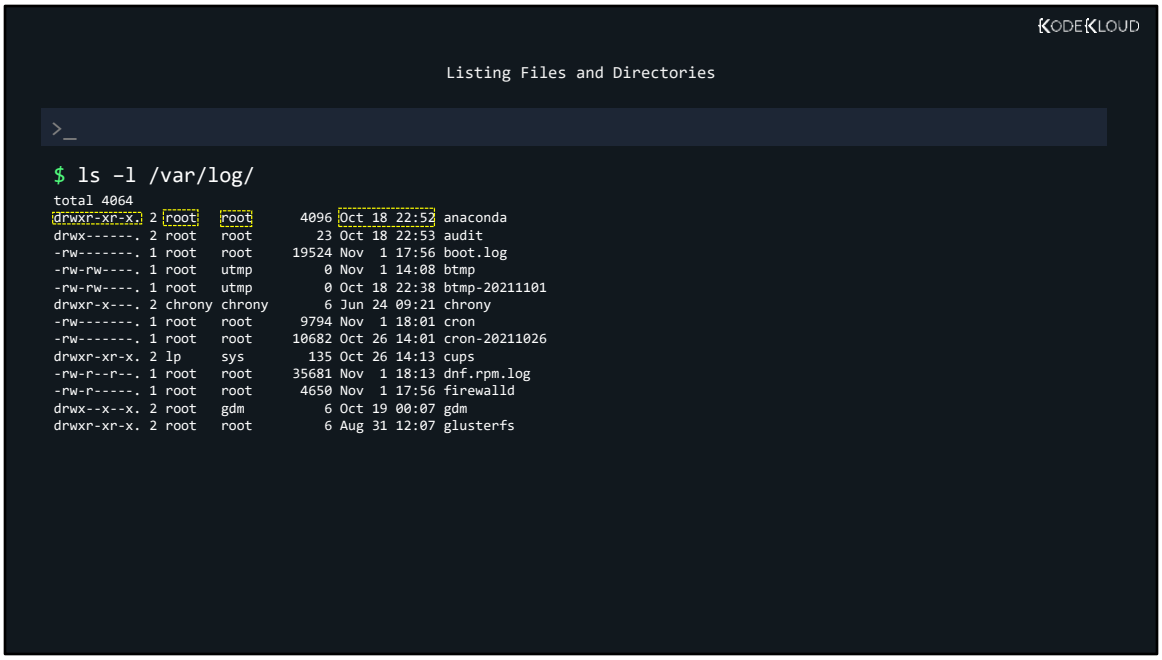


To list files and directories in your current (working) directory, we use the **ls** command in Linux. Using **ls** in your home directory might look like this:

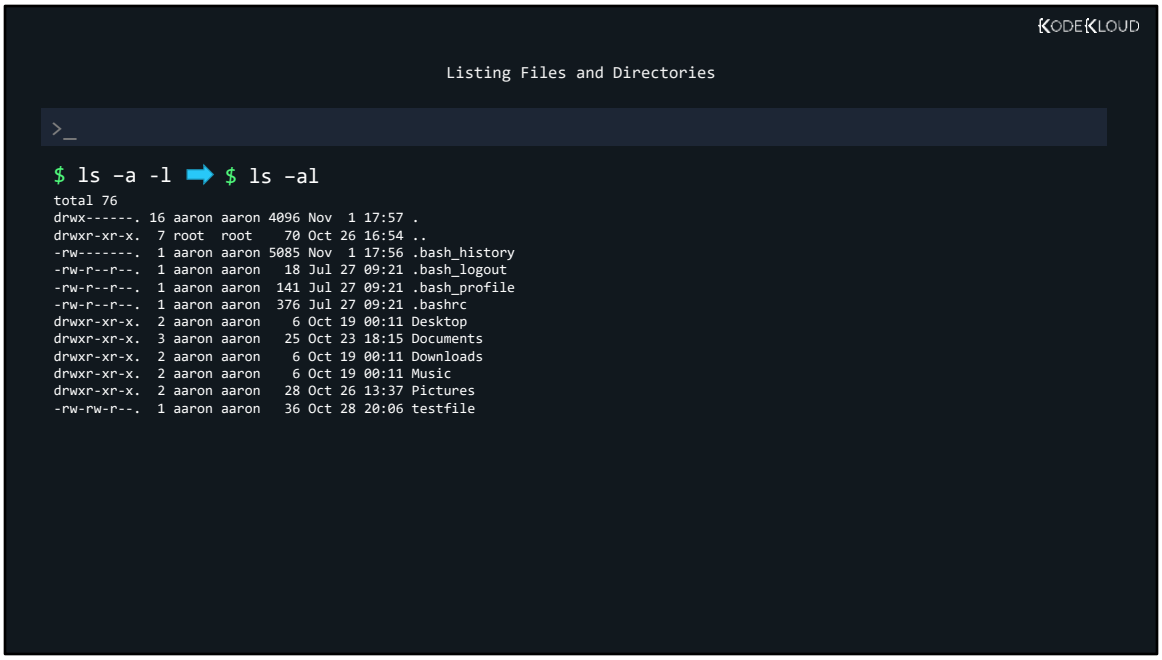
ls comes from list.

On Linux, files and directories can have a name that begins with a **.** Example: the **".ssh" directory**. These **won't be displayed** by a simple ls command. They are, in a way, **hidden**.

To list **all files and directories**, even the ones beginning with a .**,** use **ls –a** (the –a flag comes from the word **all**.)



Of course, to list files and directories from a different location, we just type the directory path at the end of ls, like **ls /var/log/** or **ls -l /var/log/** to list files and directories in a different format, called a "**long listing format,**" which shows us **more details** for each entry, like the **permissions for a file or directory**, what **user/group owns** each entry, when it was **last modified**.

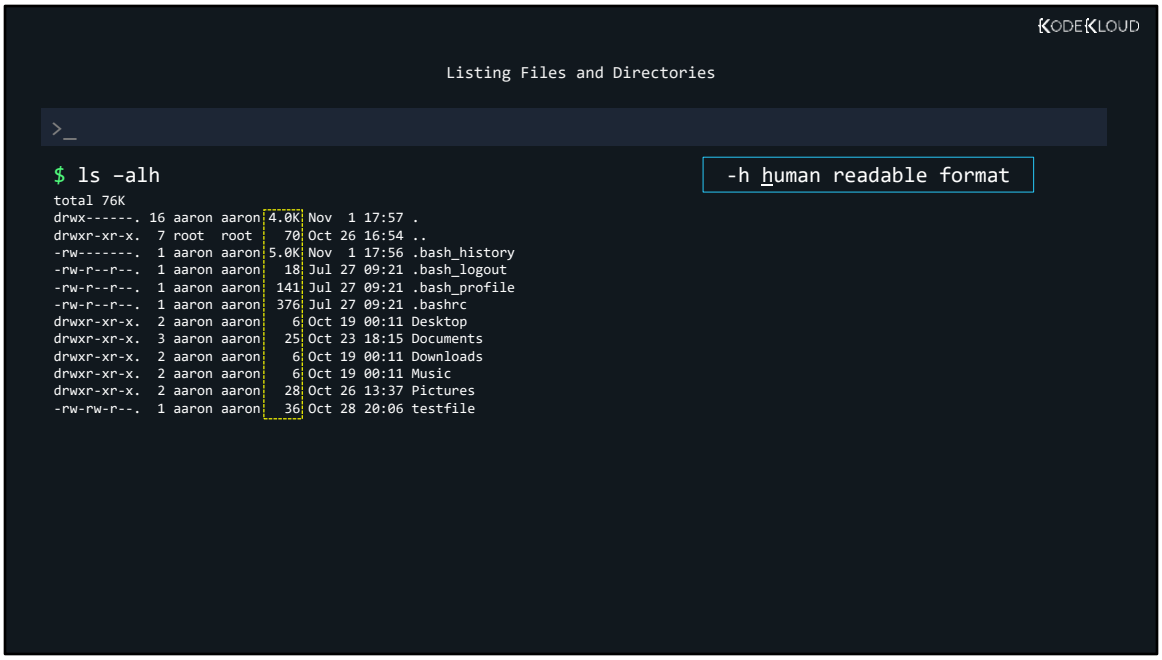


We can **combine** the -a and -l command line options like this:

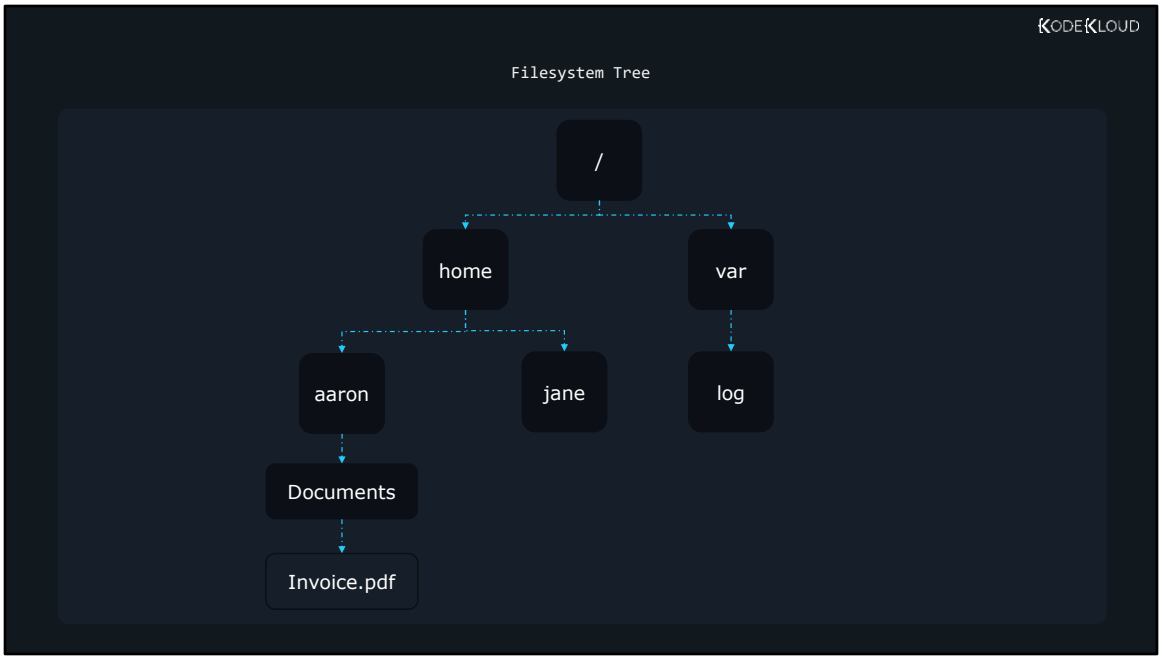
ls -a –l or like this as **ls –al**.

This will display entries in long listing format and also show us "pseudo-hidden" files and directories which have a name beginning with a **.** It doesn't matter which order you put the flags, and you don't have to put a **–** in front of each of them. However,

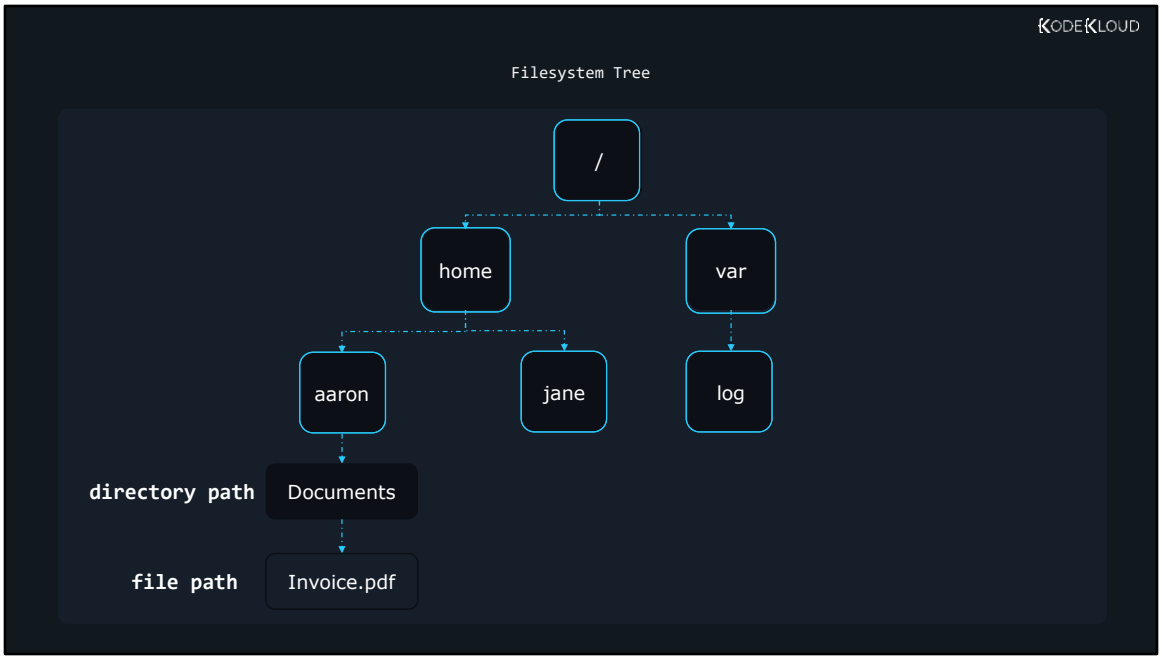
the last form is **preferred** as it's **faster to write** it.



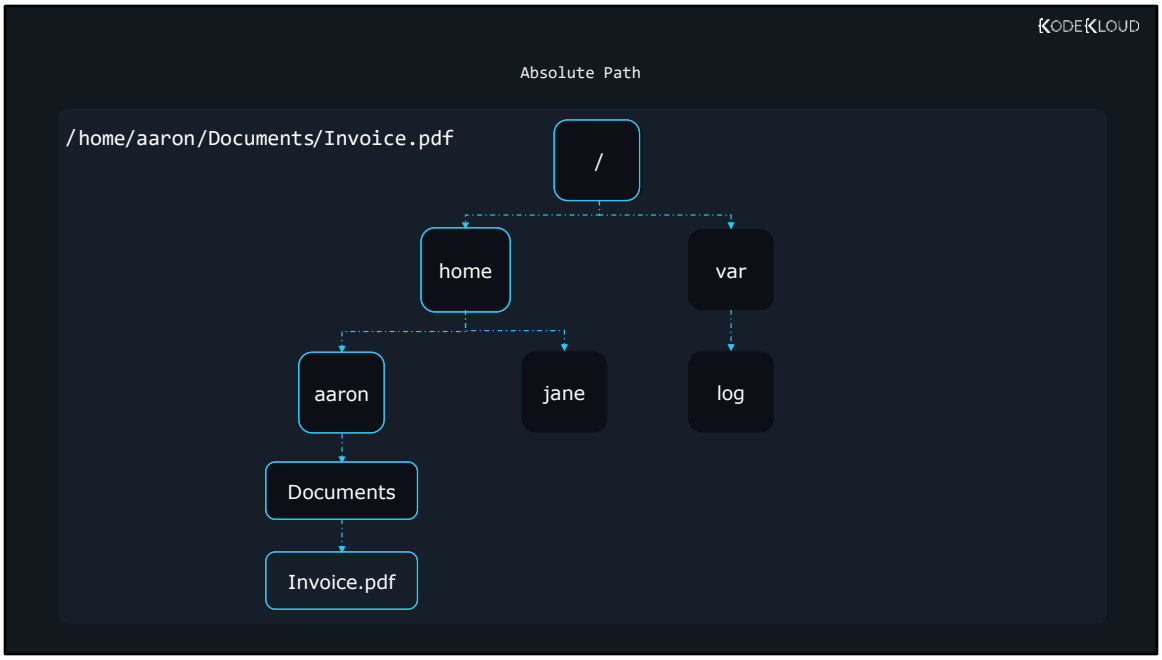
There's also a command line option, **-h,** that shows sizes in "**human readable format**": **bytes, kilobytes, megabytes**, and so on. This **must** be combined with the **-l option**. If we want to use three options, we could use **ls –alh**.



Linux organizes files and directories in what it calls the **filesystem tree**. Why is it called the filesystem tree? Because just like a tree we'd see in nature, this also has a **root**, **branches** and **leaves**. Except, Linux's filesystem tree is **inverted**. The **root** is at the **top** and its branches and leaves "grow" **downward**.



The root directory is **/**. This is the **top-level directory**, there can be no other directories above it. Under / there are a few **subdirectories** like home, var, etc, and so on. These subdirectories may also contain other subdirectories themselves. To access a file or directory on our command line, we must specify its **file path** or **directory path**. This path can be written in two different ways:

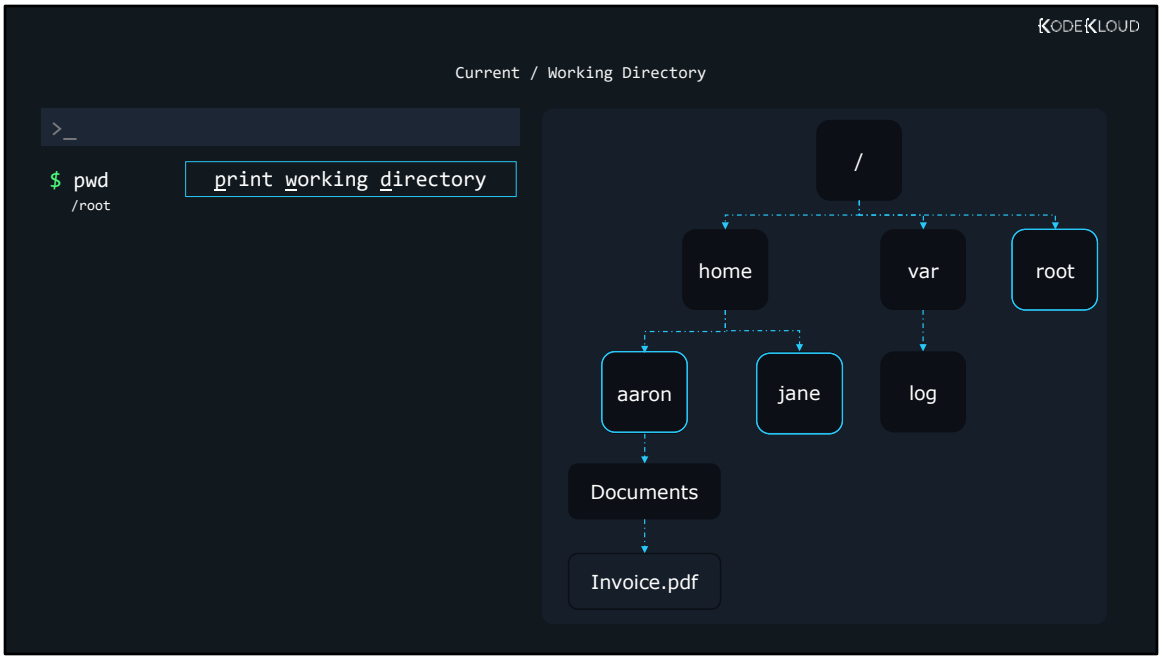


The easiest to understand is the **absolute path**.

/home/aaron/Documents/Invoice.pdf is an example of such a path.

Absolute paths always start out with the **root directory, represented by /**. Then we specify the subdirectories we want to **descend into**, in this case, first **home**, then **aaron**, then **Documents**. We can see the subdirectory names are separated by a **/**. And we finally get to the file we want to access, **Invoice.pdf**.

An absolute path can end with the name of a file, but also with the name of a directory. If we'd want to refer to the Documents directory, we'd specify a path like /home/aaron/Documents



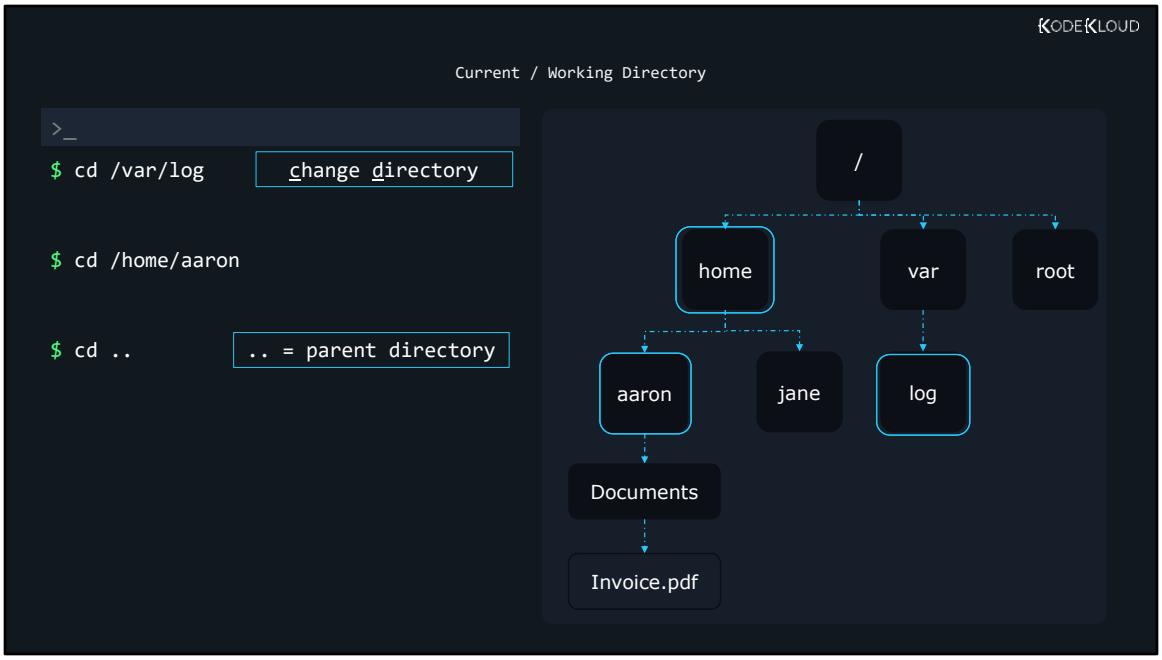
To understand a relative path, we first must explore what the **current directory** means. This is also called the **working directory**.

To see our current (working) directory we can type

pwd

pwd = Print Working Directory

When we're working at the command line, we're always "inside" a directory. For example, if we log in as the user "aaron" on some server, our starting current directory might be /home/aaron. Every user starts in its **home directory** when they log in. jane might have it at /home/jane, and root (the super user/administrator) has it at /root.



To change our current directory, we use the cd command (change directory).

cd /var/log

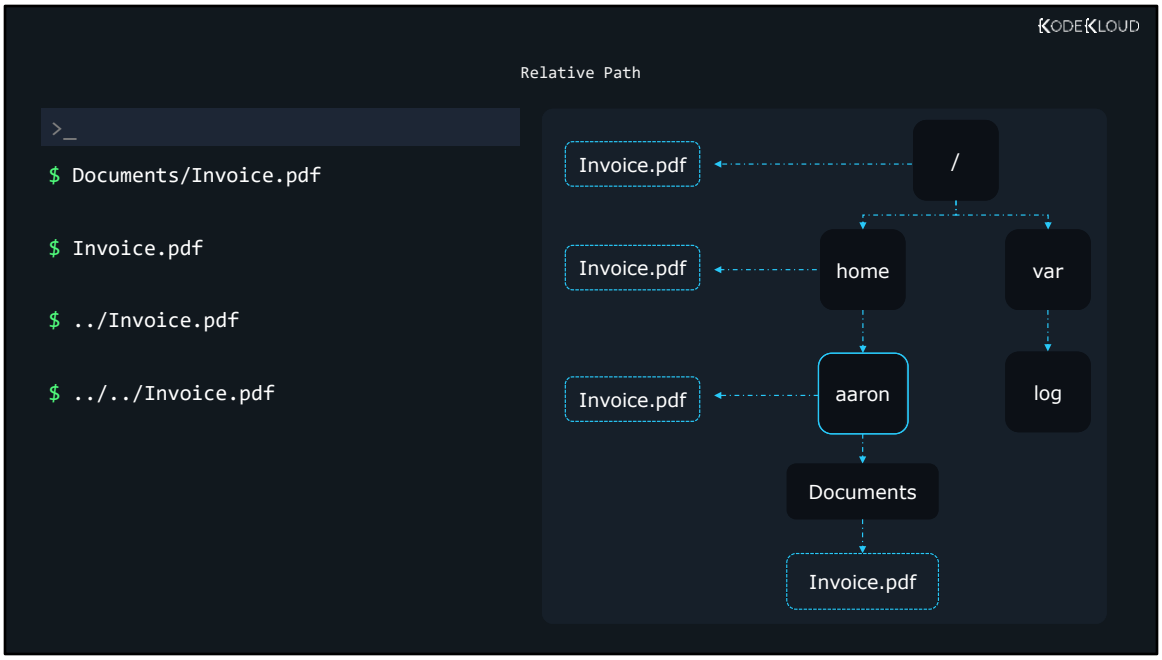
would change our current directory to /var/log. We used an absolute path here. But we can also change directory this way:

cd ..

This will take us one directory UP.

If we were cd into /home/aaron, running “**cd ..**” would take us into /home, which becomes the new current directory.

“**..**” always refers to the **parent directory** of our current directory. This was an example of using a very simple relative path. Let's dive deeper.



Let's imagine our current directory is /home/aaron. With relative paths we can refer to other places in one of three main ways

•Locations "**under**" our current

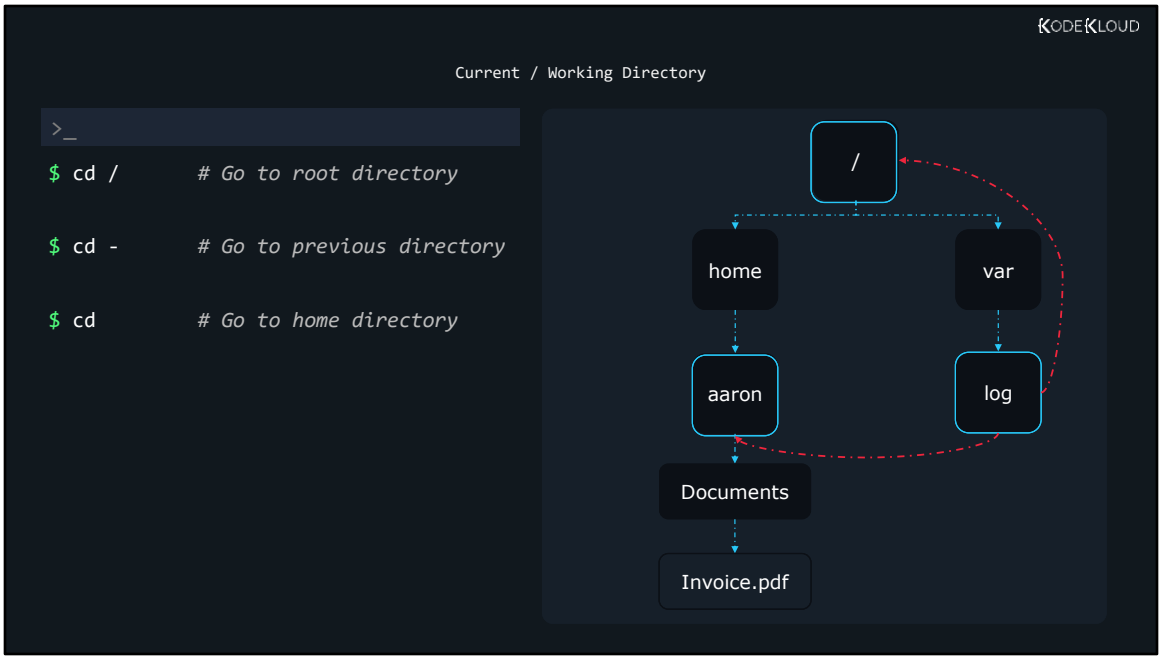
directory. E.g., Documents/Invoice.pdf Since we're in **/home/aaron**, typing a path like **Documents/Invoice.pdf** is like typing **/home/aaron/Documents/Invoice.pdf**. Our relative path "gets added" to our current directory and we get to our PDF file.

•Locations **in** our current directory. Typing Invoice.pdf will access the file at **/home/aaron/Invoice.pdf**

•Locations **above** our current directory. Typing **../Invoice.pdf**

points to the file at **/home/Invoice.pdf**. Since we used **../** we basically said, "**go one directory up**".

•We can use .. **multiple times**. **../../Invoice.pdf** points to the file at **/Invoice.pdf**. The first .. "**moved**" the relative path at **/home**, the **next ..** moved it at **/**.



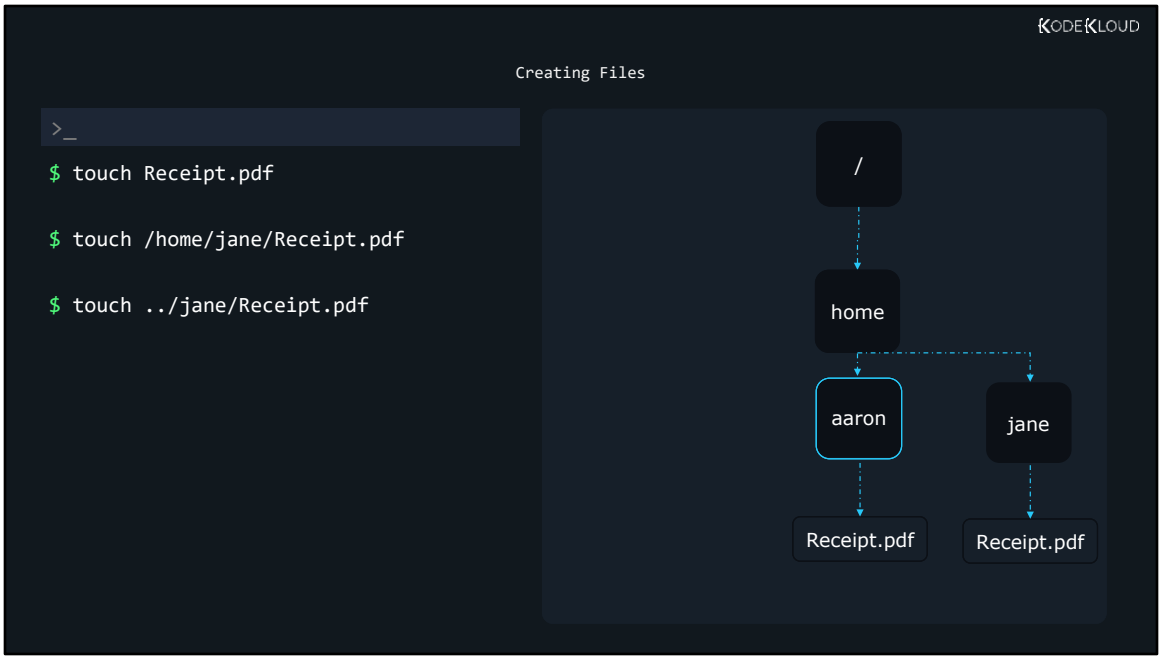
Extra tips:

If you're in **/var/log** currently and you move to **/**, you could run the command **cd /** and it will take you to the root directory.

You can return to your **previous working directory** with the **cd -** command. It will take you back to **/var/log**.

If you're in **/var/log** and you want to return to your **home directory** – in our case, **/home/aaron** – use **cd**.

cd **without any options or paths after it** will always take you **back to the home directory**.



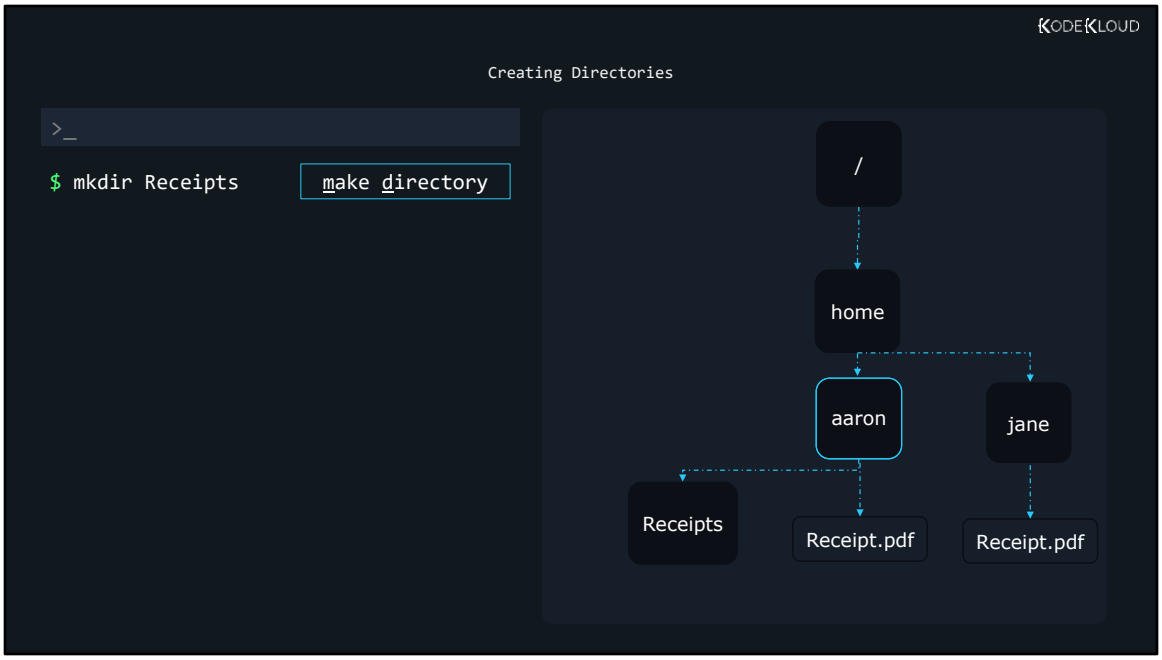
Let's assume we're in our home directory, and we want to create a new file. To do this, we can use **touch**. For example, to create a file named "Receipt.pdf," we would type **touch Receipt.pdf**.

This will create it inside the current directory. To create it at another location, we could use **touch /home/jane/Receipt.pdf**

Since we're in **/home/aaron**, we could also use the

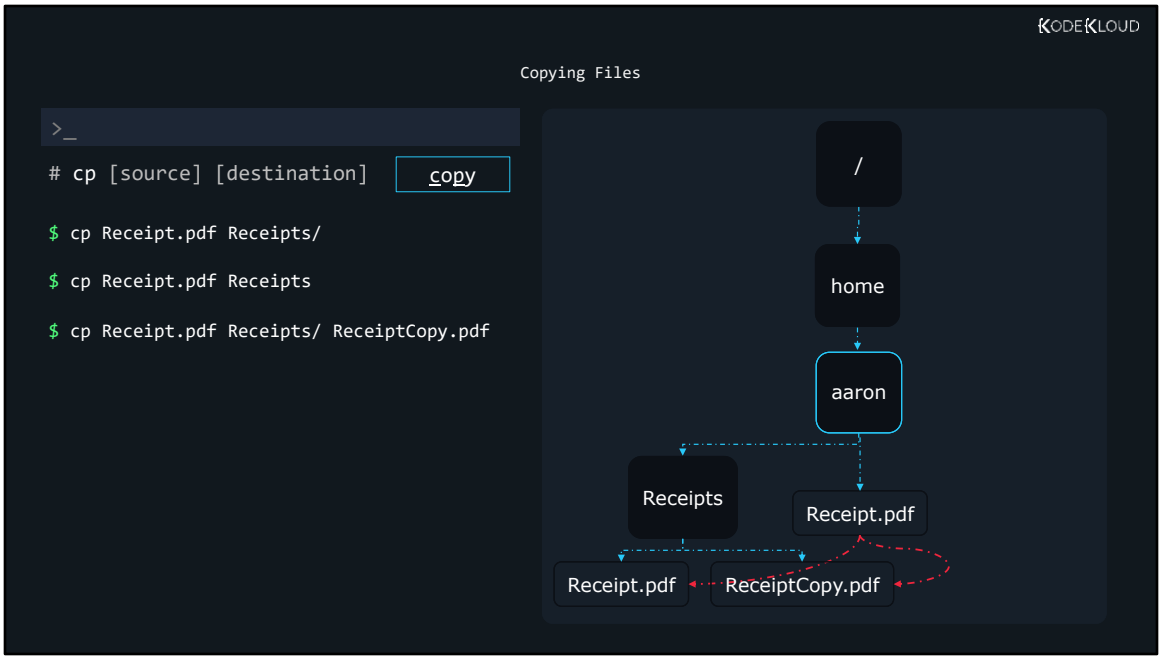
relative path to create file in **/home/jane** by typing **touch ../jane/Receipt.pdf**.

Both commands would work the same because **all the commands we'll discuss** accept both **absolute, and relative paths,** so we won't mention these alternatives for each one. Just know that after the command, you can use any kind of path you want.



To create a new directory, use **mkdir**; for example: **mkdir Receipts**

mkdir comes from **make directory**

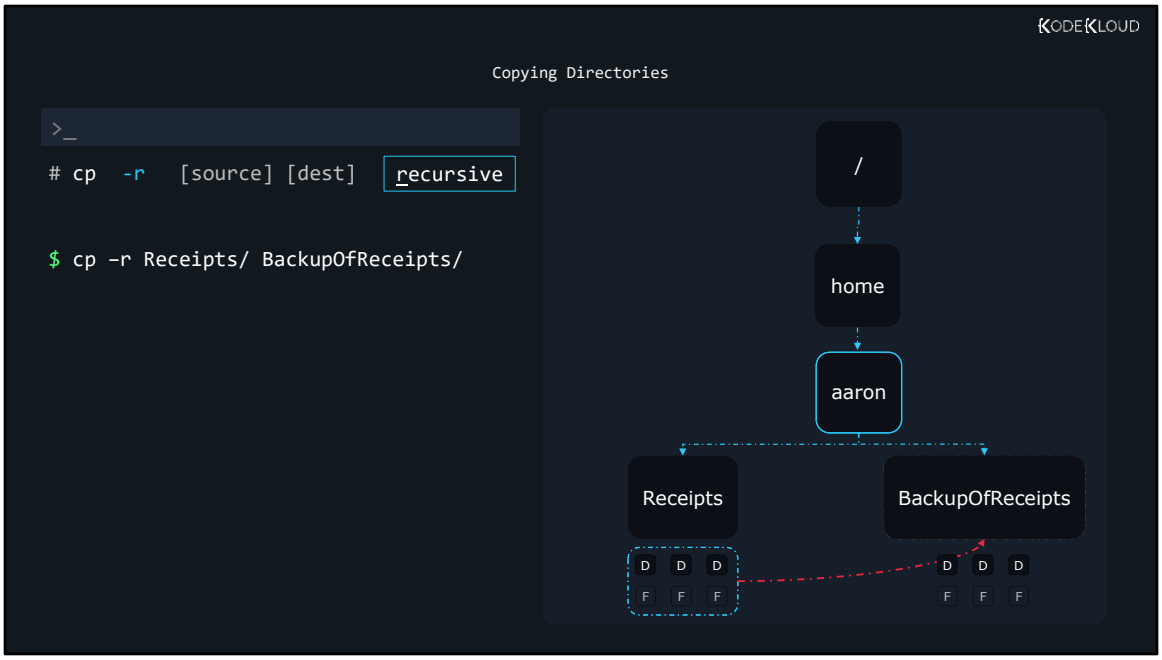


To copy a file, we use the **cp** command, which is short for copy. cp is followed by the path to the file we want to copy (**source**), then the path to the **destination** where we want to copy it. "cp source destination"

To copy Receipt.pdf to the Receipts directory, we'd use **cp** Receipt**.pdf** Receipts**/**

Notice how we terminated the path to the Receipts directory with a **/**, to make it **Receipts/**? Without the / would have worked too. But it's **good practice** to **end your directories with a /**. This way, you'll form a **healthy habit** and get a **visual indicator** that tells you when **Receipts** (**without /**) might be a **file**, and **Receipts/** might be a **directory**.

To copy Receipt.pdf to the **Receipts** directory, but **also choose a new name for it**, we could use **cp** Receipt**.pdf** Receipt**s/**Receipt**Copy.pdf**.



To copy a directory and all its contents to another directory run the cp command as before but with the –r option.

The -r is a **command line option** (also called command line **flag**)

that tells cp to copy recursively. That means, the directory itself, but also descend into the directory and copy everything else it contains, files, other subdirectories it may have, and so on.

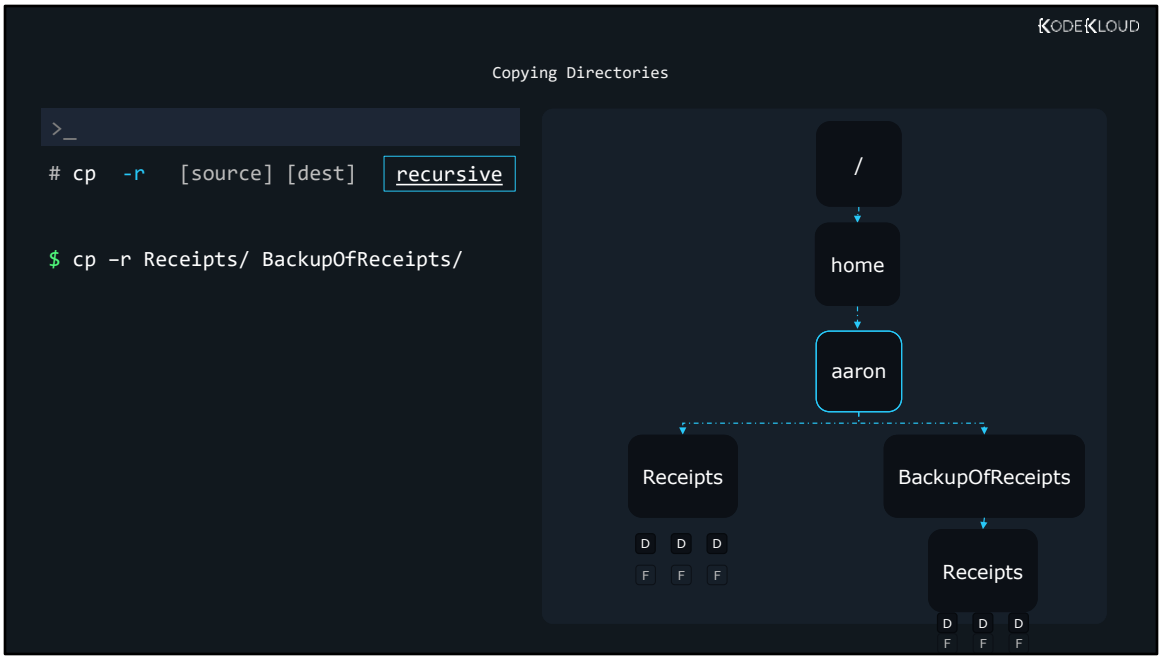
For example, say I have a lot of directories, subdirectories and

files under the receipts directory. And I'd like to back up all the contents into a backup directory named BackupOfReceipts.

Run the command – cp

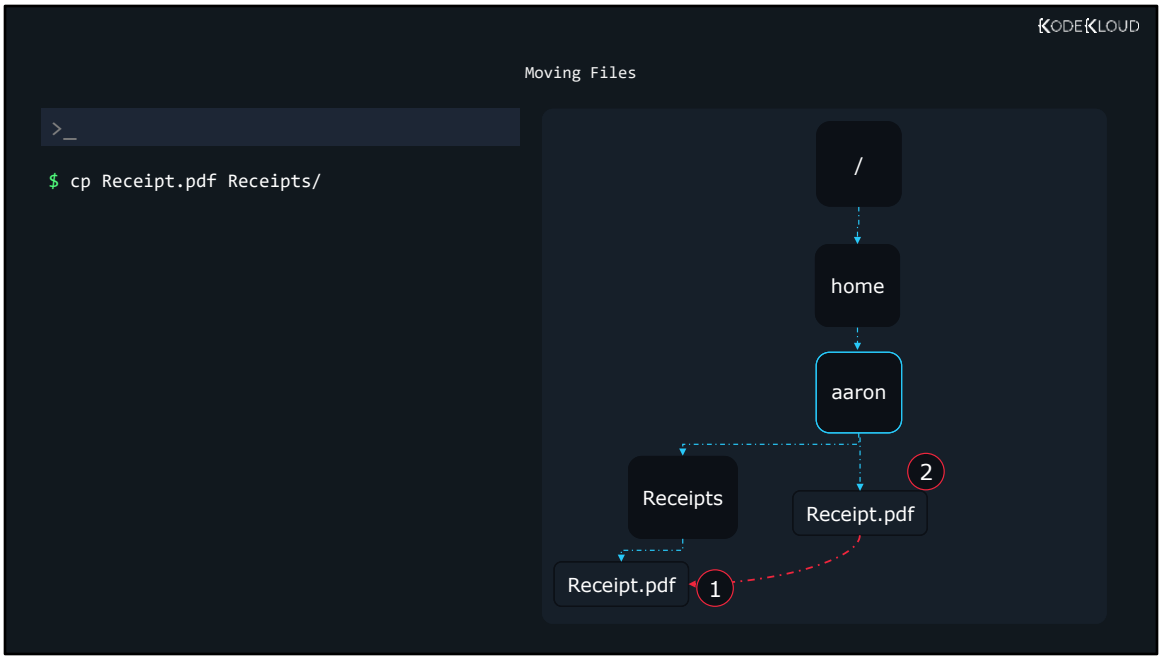
–r Receipts/ BackupOfReceipts/

This copies all subdirectories and files from the receipts folder into the backpupofreceipts folder.



The name you choose for your cloned directory **must not exist at your destination**. For example, if we'd already have a directory at **/home/aaron/BackupOfReceipts**, this will just move **Receipts** there and it would end up at

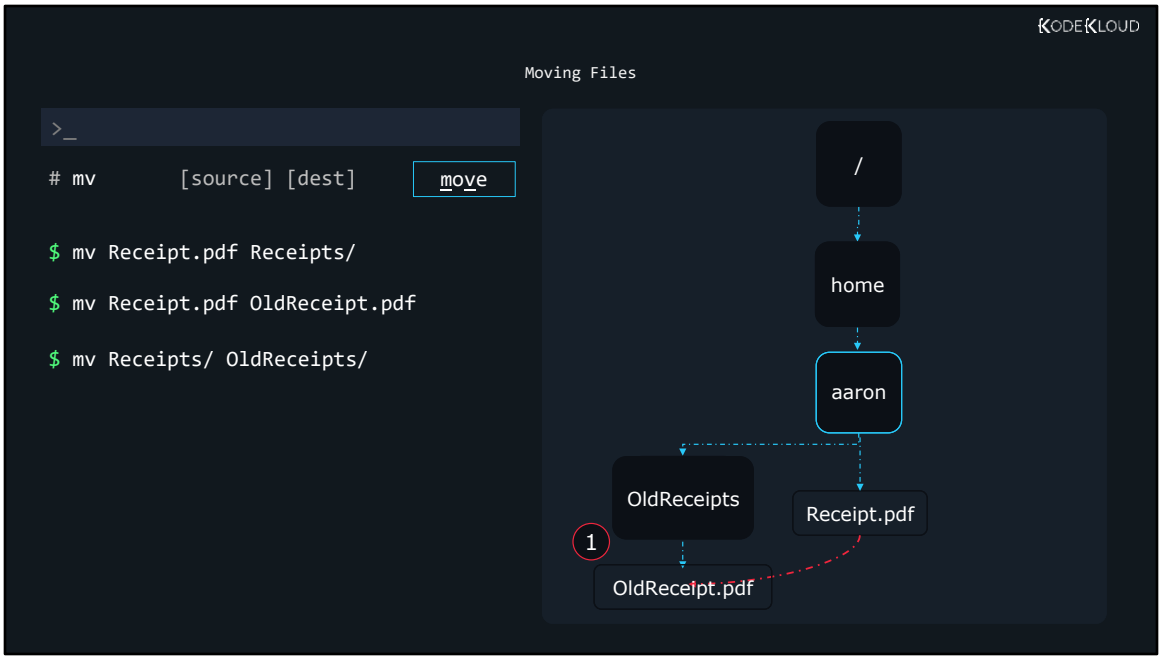
Documents/BackupOfReceipts/Receipts/.



So, we saw that the copy operation copies a file from one place to another, resulting in 2 copies of files – the original one and the new one in the new location.

But what if we want to move a file

from one location to another? So that the file is not present in the original location but is only present in the new location?



For this use the mv command. Mv stands for move.

Run the command mv Receipt.pdf Receipts/ to move the file from Receipt.pdf to the Receipts folder. The file is moved and there is only 1 copy of file

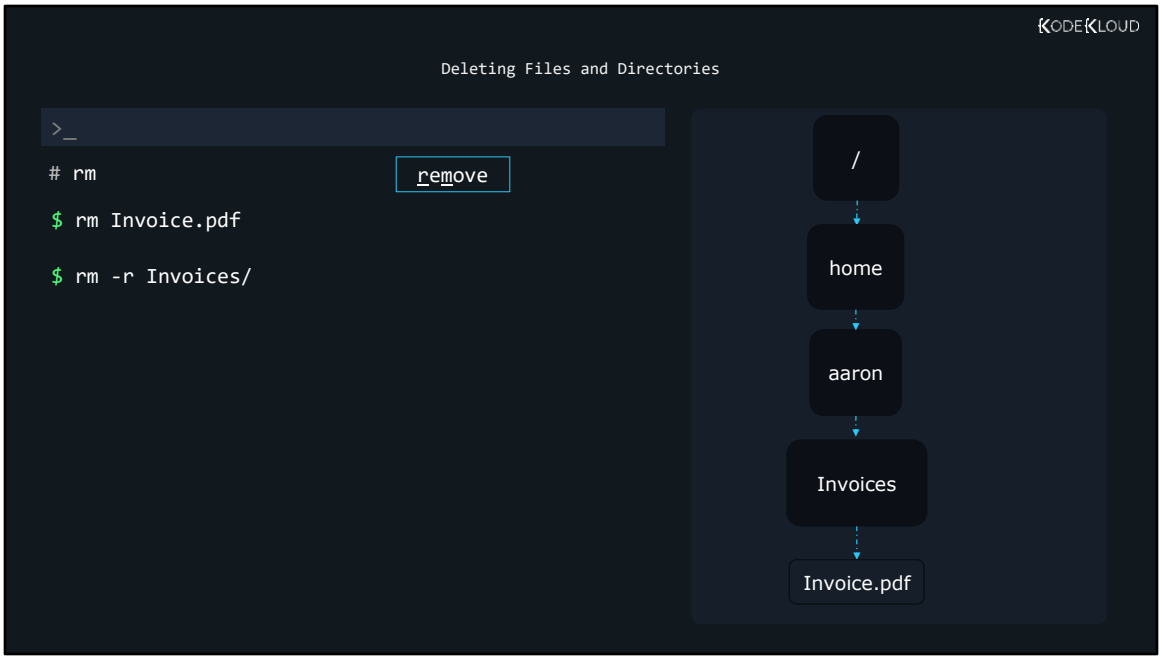
available.

To rename a file, we can use: **mv Receipt.pdf OldReceipt.pdf**

To **rename** a directory, we can use the **new name** as the **destination**, such as: **mv Receipts/**

OldReceipts/.

Notice that we did not have to use the -r flag with mv to recursively work with directories? Mv takes care of that for us.



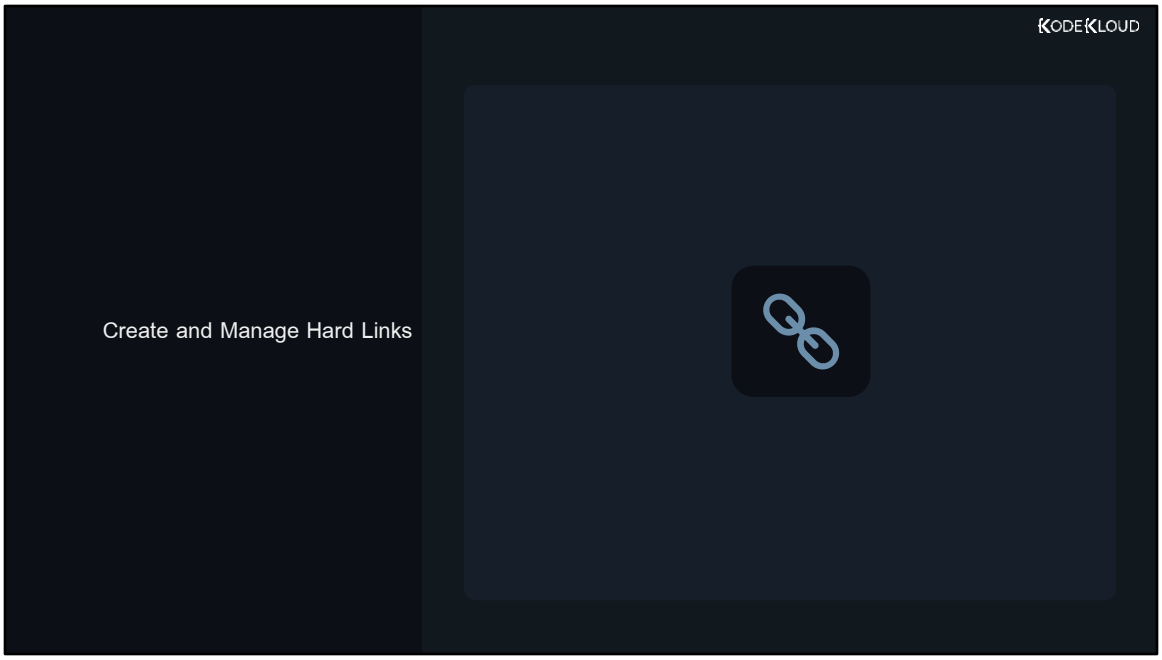
To delete a file, we use the **rm** command. rm comes from **remove**. To delete the file Invoice.pdf, we can use **rm Invoice.pdf**

To delete a directory like the Invoices directory, we would use : **rm -r Invoices/**

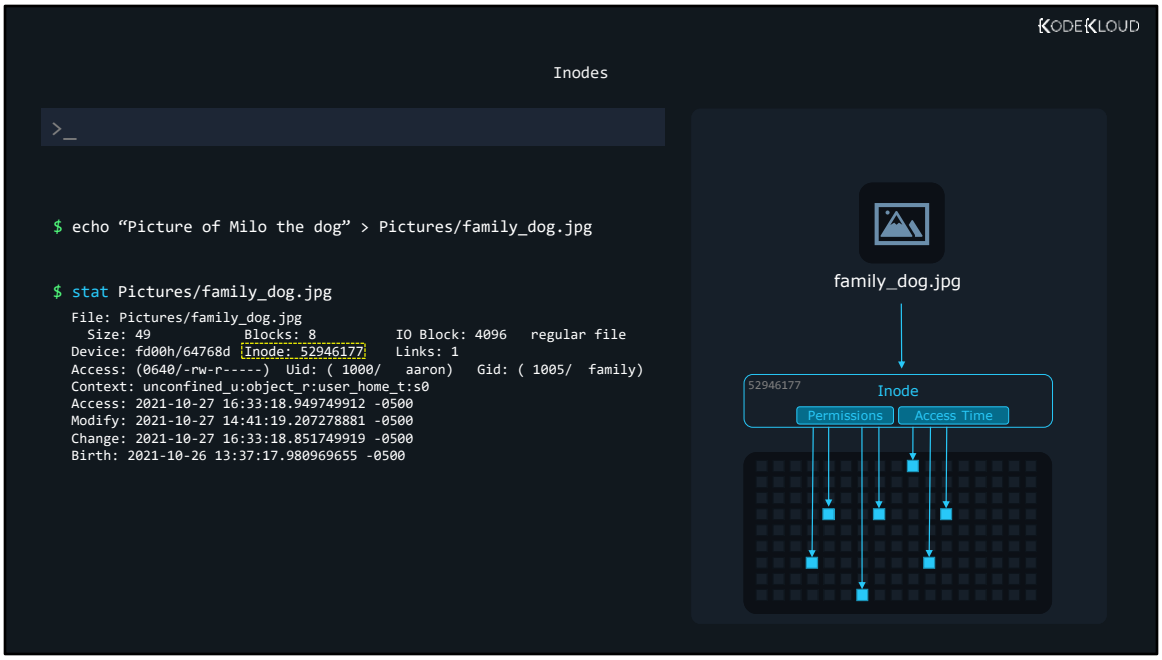
Once again, the -r option was used to do this recursively, deleting the directory, along with its subdirectories and files. When you **copy** or **delete**

directories, remember to **always** add the **-r** option.

Access the labs associated with this course using this link: https://kode.wiki/linux-labs



In this lecture, we'll look at how Linux manages hard links.



To understand hard links and soft links we first must learn some very basic things about filesystems.

Let's imagine a Linux computer is shared by two users: aaron and jane. Aaron logs in with his own username and password, Jane logs in with her own username and password. This lets them use the same computer, but have different desktops, different program settings, and so on. Now Aaron takes a picture of the family dog and saves it into /home/aaron/Pictures/family_dog.jpg.

Let's simulate a file like this.

echo "Picture of Milo the dog" > Pictures/family_dog.jpg

With this, we created a file at Pictures/family_dog.jpg and stored the text "Picture of Milo the dog" inside.

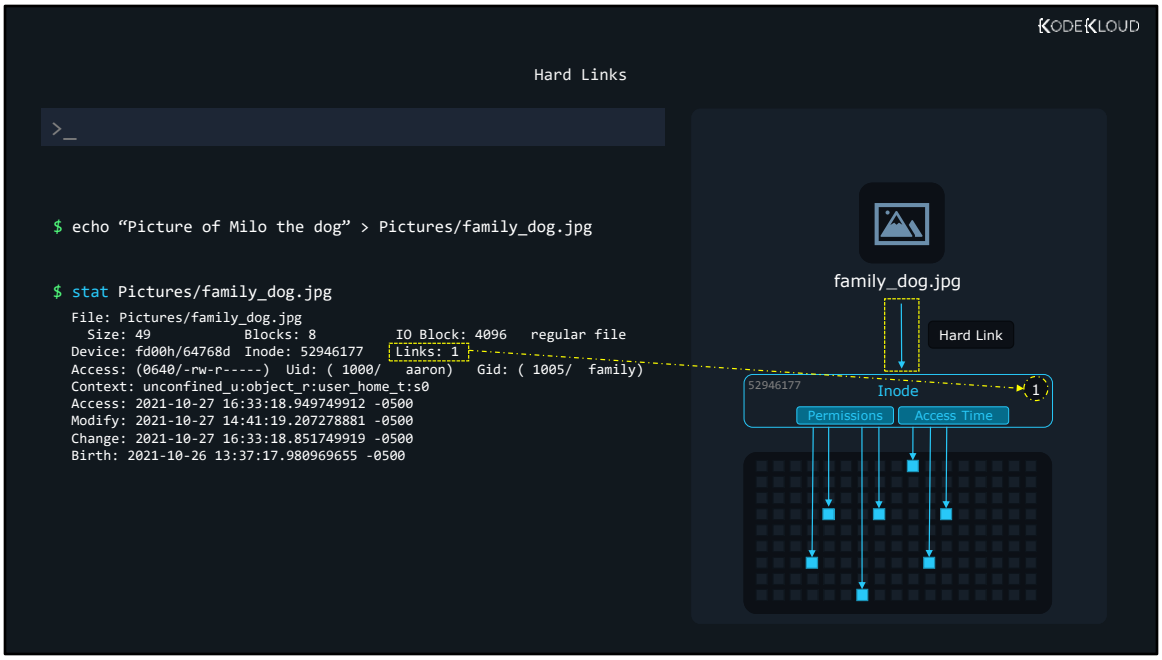
There's a command on Linux that lets us see some interesting things about files and directories.

stat Pictures/family_dog.jpg

We'll notice an Inode number. What is this?

Filesystems like xfs, ext4, and others, keep track of data with the help of inodes. Our picture might have blocks of data scattered all over the disk, but the inode remembers where all the pieces are stored. It also keeps track of metadata: things like permissions, when this data was last modified, last accessed, and so on. But it would be inconvenient to tell your computer, "Hey, show me inode 52946177". So, we work with files instead, the one called family_dog.jpg in this case. The file points to the inode, and the inode points to all the blocks of data that we require.

And we finally get to what interests us here.



We notice this in the output of our stat command.

There's already one link to our Inode? Yes, there is. When we create a file, something like this happens:

We tell Linux, "Hey save this data under this filename: family_dog.jpg"
Linux says: "Ok, we'll group all this file's data under inode 52946177. Data blocks and inode created. We'll **hardlink** file "family_dog.jpg" to Inode 52946177.

Now when we want to read the file:

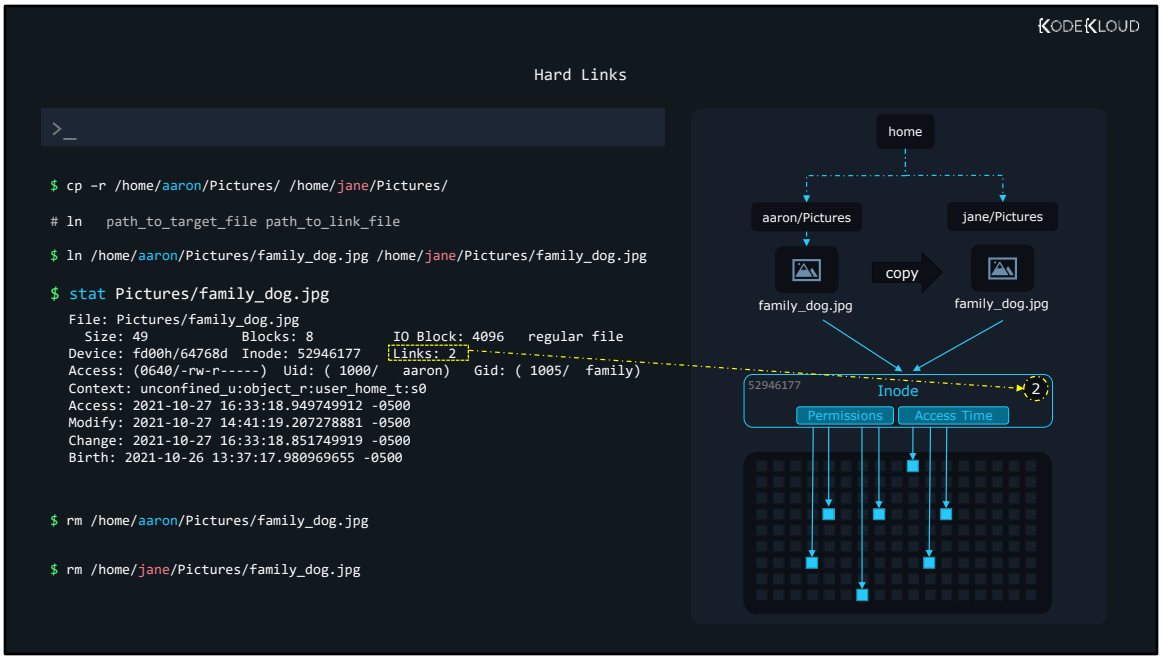
"Hey Linux, give me data for family_dog.jpg file"

And linux goes: "Ok, let me see what inode this links to. Here's all data you requested for inode 52946177"

family_dog.jpg -> Inode 52946177

So the number shown as Links in the output of the stat command is the number of hard links to this inode from files or filenames.

Easy to understand. But why would we need more than one hard link for this data?



Well, Jane has her own folder of pictures, at /home/jane/Pictures. How could Aaron share this picture with Jane? The easy answer, just copy /home/aaron/Pictures/family_dog.jpg to /home/jane/Pictures/family_dog.jpg. No problem, right? But now imagine we must do this for 5000 pictures. We would have to store 20GB of data twice. Why use 40GB of data when we could use just 20GB? So how can we do that?

Instead of copying /home/aaron/Pictures/family_dog.jpg to

/home/jane/Pictures/family_dog.jpg, we could **hardlink** it to /home/jane/Pictures/family_dog.jpg.

The syntax of the command is:

ln path_to_target_file path_to_link_file

The **target_file** is the file you want to link **with**. The **link_file** is simply the name of this new hard link we create. Technically, the hard link created at the destination is a file like any other. The only special thing about it is that instead of pointing to a new inode, it points to the same inode as the **target_file**.

In our imaginary scenario, we would use a command like:

ln /home/aaron/Pictures/family_dog.jpg /home/jane/Pictures/family_dog.jpg

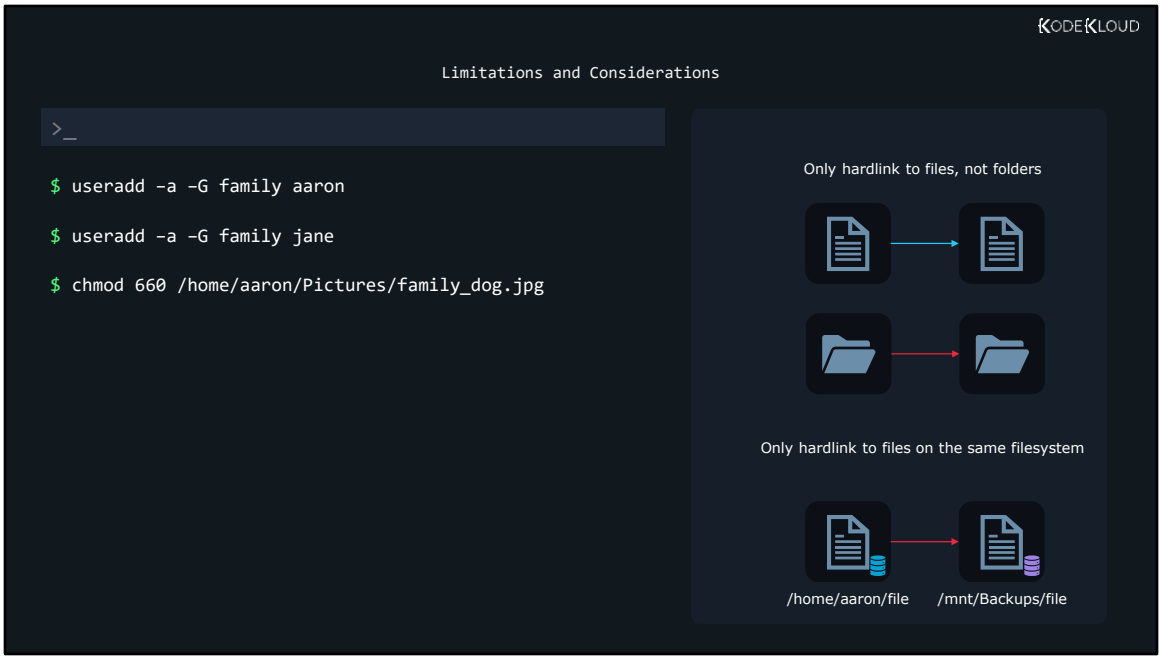
Now our picture is only stored once, but the same data can be accessed at different locations, through different filenames.

If we run the stat command now we see the Links are now 2. This is

because this Inode now has 2 hard links pointing to it.

Another beautiful thing about hard links is this: Aaron and Jane share the same 5000 pictures through hardlinks. But maybe Aaron decides to delete his hardlink of /home/aaron/Pictures/family_dog.jpg. What will happen with Jane's picture? Nothing, she'll still have access to that data. Why? Because the inode still has 1 hard link to it (it had 2, now it has 1). But if Jane also decides to delete her hard link /home/jane/Pictures/family_dog.jpg, the inode will have 0 links to it. When there are 0 links, the data itself will be erased from the disk.

The beauty of this approach is that people that share hard links can freely delete what they want, without having a negative impact on other users that still need that data. But once everyone deletes their hard links to that data, the data itself will be erased. So, data is "intelligently removed" only when EVERYONE involved decides they don't need it anymore.



Limitations of hard links:
•You can only hardlink to files, not directories.
•You can only hardlink to files on the same filesystem. If you had an external drive mounted at /mnt/Backups, you would not be able to hardlink a file from your SSD, at /home/aaron/file to some other file on /mnt/Backups since that's a different filesystem.

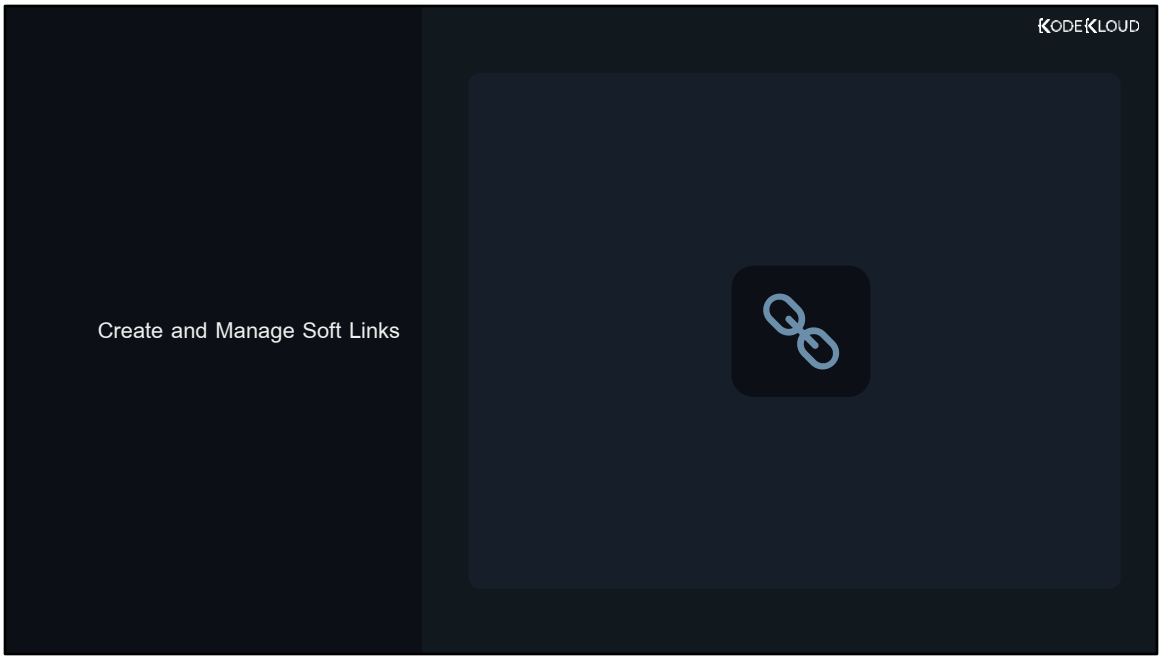
Things to take into consideration when you hardlink:

First, make sure that you have the proper permissions to **create** the link file at the destination. In our case, we need write permissions at: /home/jane/Pictures/.

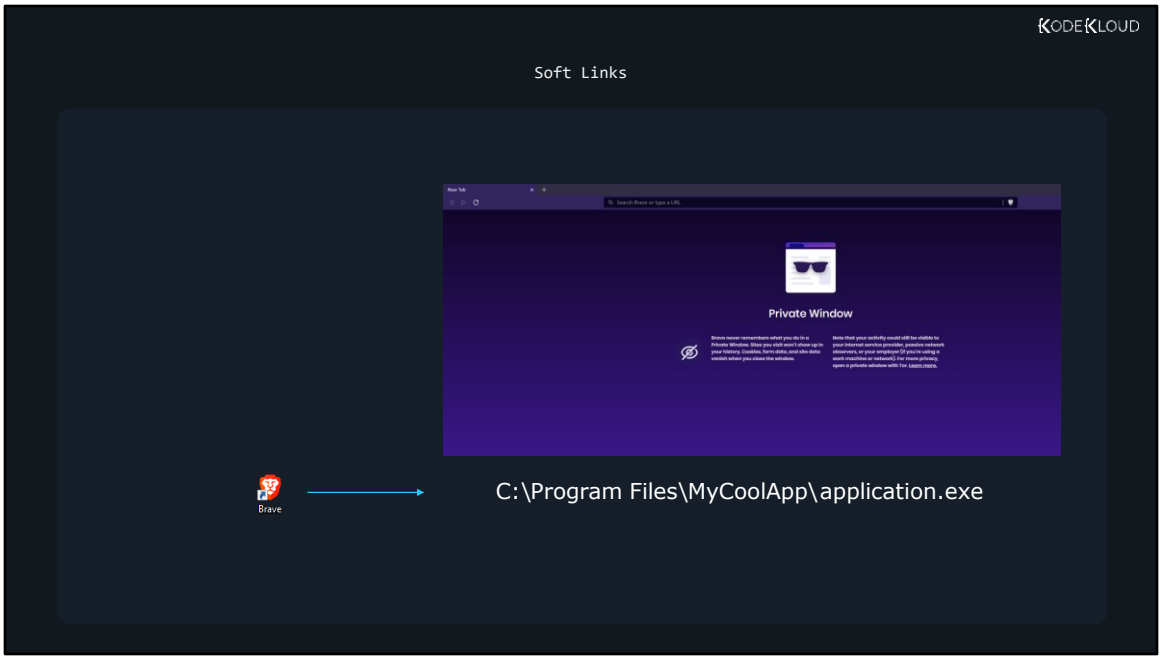
Second, when you hardlink a file, make sure that all users involved have the required permissions to **access** that file. For Aaron and Jane, this might mean that we might have to add both their usernames to the same group, for example, "family". Then we'd use a command to let the group called "family" read and write to this file. You only need to change permissions on one of the hardlinks. That's because you are actually changing permissions stored by the

Inode. So, once you change permissions at /home/aaron/Pictures/family_dog.jpg, /home/jane/Pictures/family_dog.jpg and all other hard links will show the same new sets of permissions.

Access the labs associated with this course using this link: https://kode.wiki/linux-labs



Let's look now at how Linux manages soft links.



Know how when you install a program on Windows, you might get a shortcut on your desktop? You double click on that shortcut and that application gets launched. The application is obviously not installed on your desktop. It may have its files stored in C:\Program Files\MyCoolApp directory. And when you double click the shortcut, this only **points to** an executable file at C:\Program Files\MyCoolApp\application.exe. So, the double click on that shortcut basically redirects you to the file C:\Program Files\MyCoolApp\application.exe, which gets executed.

Soft links in Linux are very similar. A hard link **pointed to an inode**. But a soft link is nothing more than a file that **points to a path** instead. It's almost like a text file, with a path to a file or directory inside.

The syntax of the command to create a soft link (also called symbolic link) is the same as before, but we add the -s or --symbolic option:

ln -s path_to_target path_to_link_file

path_to_target = our soft link will **point to this <u>path</u>** (location of a file or directory)

path_to_link_file = our soft link file will be created here

For example, to create a symbolic link that points to the Pictures/family_dog.jpg file, we can run the command:

ln -s Pictures/family_dog.jpg family_dog_shortcut.jpg

Now if we list files and directories in long listing format with the ls –l command, we'll see an output like this:

The **l** at the beginning shows us that this is a soft **l**ink. And ls -l even displays the path that the soft link points to.

If this path is long, ls -l might not show the entire path. An alternative command to see the path stored in a soft link is:

readlink path_to_soft_link

So, in our case, it would be:

readlink `family_dog_shortcut.jpg`

You may also notice that all permission bits, rwx (read, write, execute) seem to be enabled for this

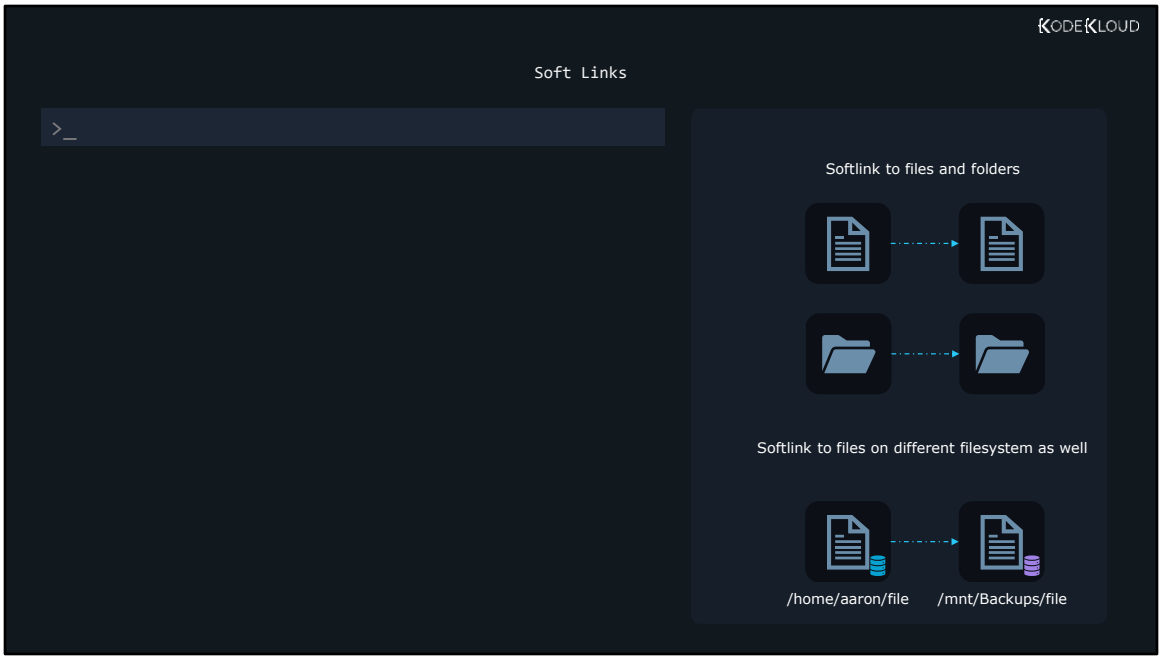
file. That's because the permissions of the soft link do not matter. If you'd try to write to "fstab_shortcut", this would be denied because the permissions of the destination file apply and /etc/fstab does not allow regular users to write here.

In our first command we used an absolute path - /home/aaron/Pictures/family_dog.jpg.

if we ever change the directory name "aaron" in the future, to something else, this soft link will break. You can see a broken link highlighted in red in the output of the ls –l command.

To tackle this you could create a soft link with a relative path. Say for example you were in the home directory of aaron, you could create a soft link using the relative path of the family_dog file instead of specifying the complete path.

When someone tries to read relative_picture_shortcut, they get redirected to **Pictures/family_dog.jpg**, relative to the directory where the soft link is.

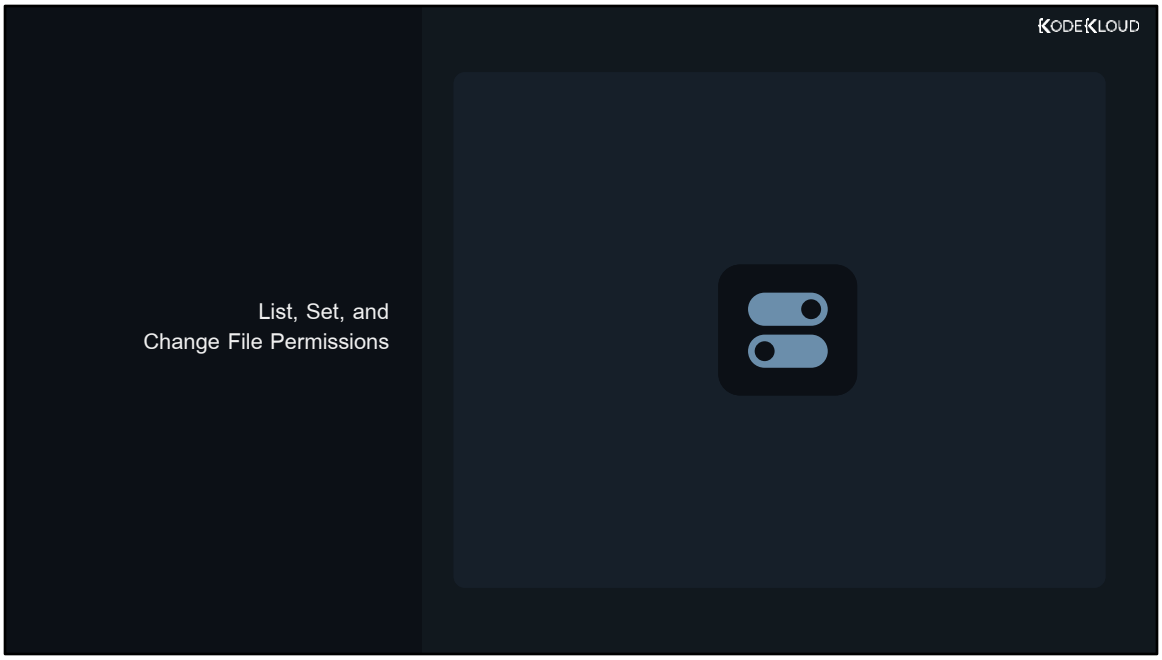


Since soft links are nothing more than paths pointing to a file, you can also softlink to directories:

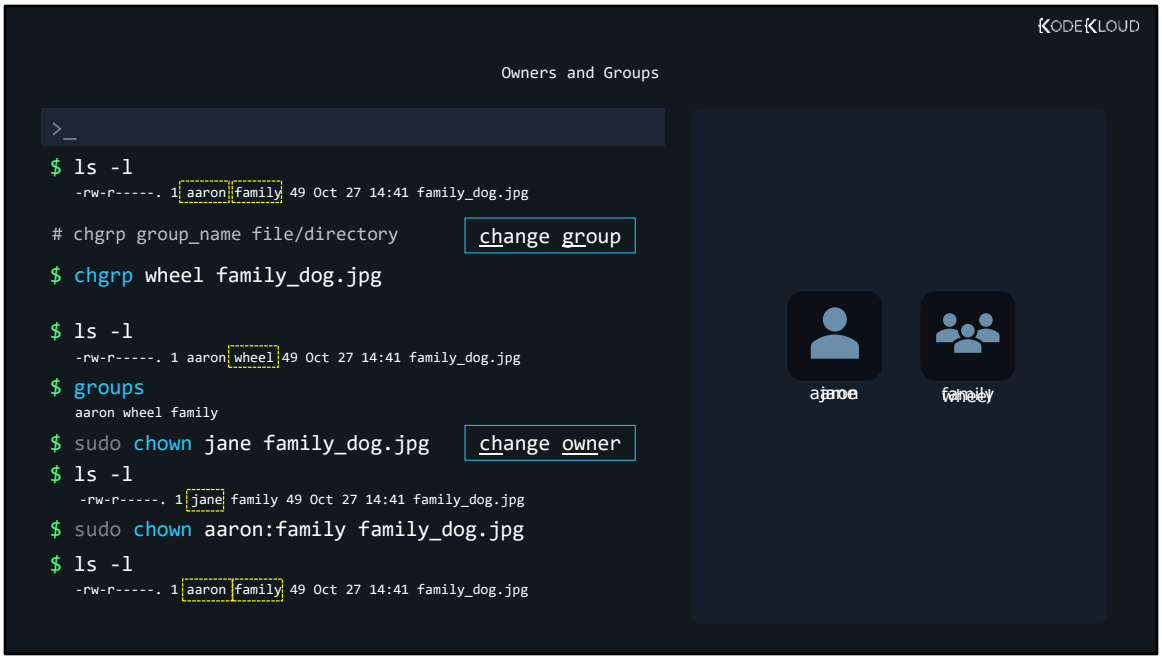
ln -s Pictures/ shortcut_to_directory

Or you can softlink to files/directories on a different filesystem.

Access the labs associated with this course using this link: https://kode.wiki/linux-labs



We'll now discuss how to list, set, and change standard file permissions in Linux.



To understand how file and directory permissions work on Linux we must first look at file/directory owners.

If we type

ls -l

we'll see something like this:

Any file or directory is owned by a **user**. In this case, we see that the file "family_dog.jpg" is owned by the user called **aaron**. Only the owner of a file or directory can change permissions, in this case,

aaron. The only exception is the **root** user (super user/administrator account), which can change permissions of any file or directory.

In the second field we can see that this file also has a group associated with it, the **family** group. We'll see later what the role of the group is.

To change the group of a file/directory, we use the chgrp command (change group).

Syntax:

chgrp group_name file/directory

For example, to change this file's group to "wheel" we'd use:

chgrp wheel family_dog.jpg

If we do another ls –l, we can see that the group has now changed to **wheel.**

We can only change to groups that our user is part of.

We can see to what groups our current user belongs with:

groups

This means we can change the group of our file to: aaron, wheel or family.

Again, the root user is the exception, which can change the group of a file or directory to whatever group exists on the system.

There's also a command to change the **user owner** of a file or directory: chown (change owner).

The syntax is:

chown user file/directory

For example, to change ownership of this file to jane, we'd use:

chown jane family_dog.jpg

But only the root user can change the user owner, so we'd have to use the sudo command to temporarily get root privileges:

sudo chown jane family_dog.jpg

With another ls –l, we can see the user has now changed to jane.

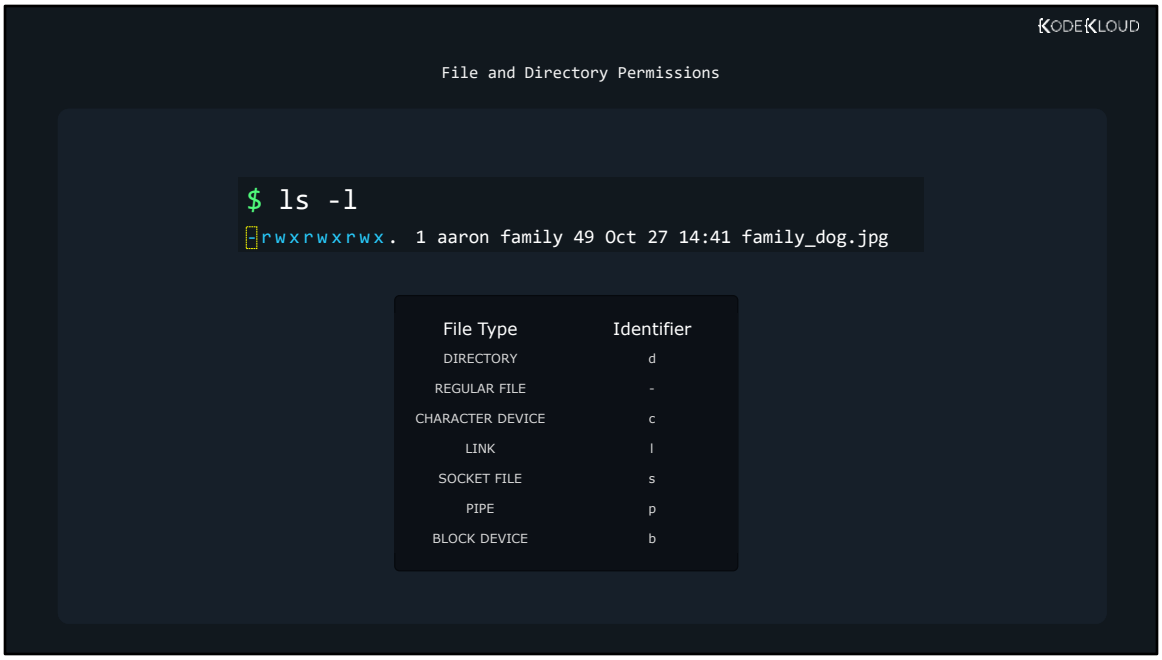
We can change both user owner and group with a different syntax of chown:

chown user:group file/directory

And since only root can change user ownership, let's set user to aaron and group to family to revert all our changes:

sudo chown aaron:family family_dog.jpg

One last ls –l will show us that the owner is aaron again, and the group is family.



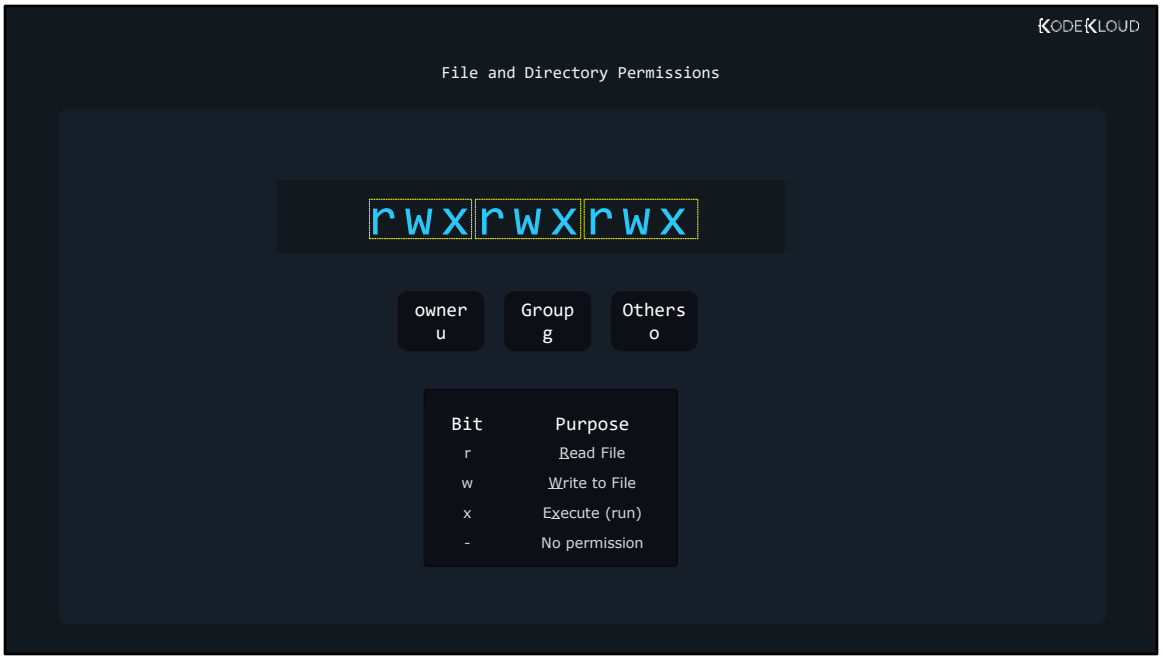
Our

ls -l

command also shows us the permissions of all files and directories in our current directory

first character on that line shows us what type of entry this is: a file, a special file, a directory and so on. For example, we'd see "d" for a directory, "l" for a soft link, or "-" for a regular file. Here’s a table that shows the different identifiers and what they stand for.

We will learn about some of these file types later in this course.



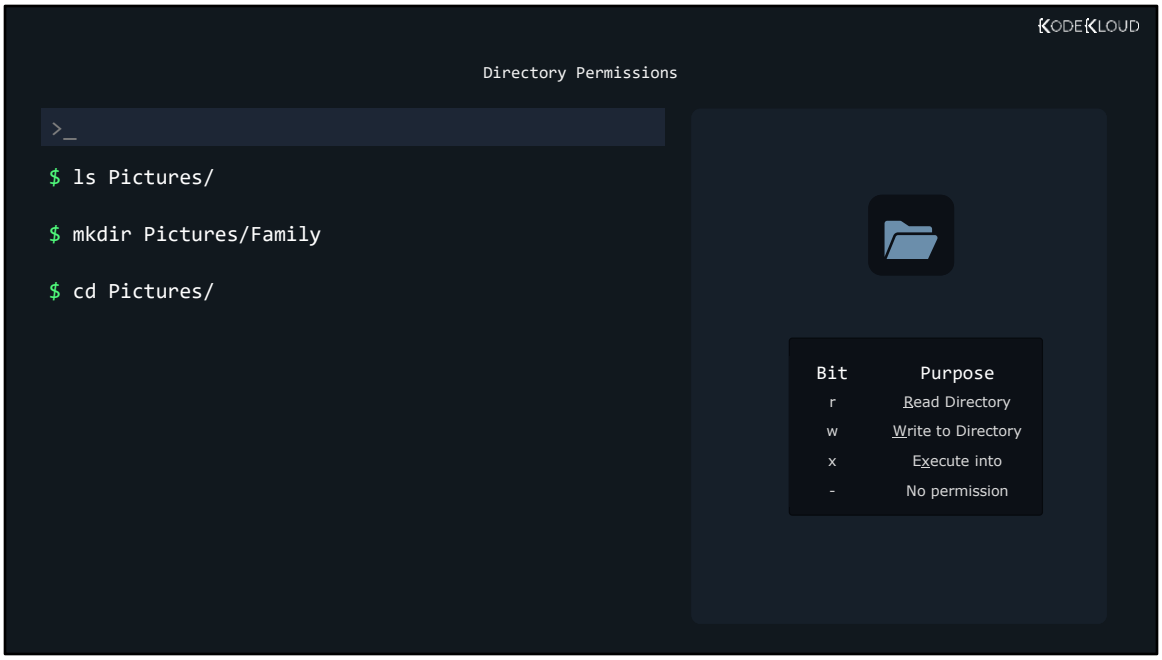
The next 9 characters show us permissions:

•First 3: permissions for the **user** that owns this file.
•Next 3: permissions for the **group** of this file.
•Last 3: permissions for **other** users (any user that is not aaron or not part of the family group).

Let's see what r, w and x mean in two different contexts, because they act in a certain way for files and have slightly different behavior for directories.

For a file:
•r means the user, group, or other users can read the contents of this file. - means they cannot read it.

•w means the user, group, or other users can write to this file, modify its contents.
x means the user, group, or other users can execute this file. Some files can be programs or shell scripts (instructions we can execute). To be able to **run** this program or shell script, we must have the x permission. A - permission here means the program or shell script cannot be executed.

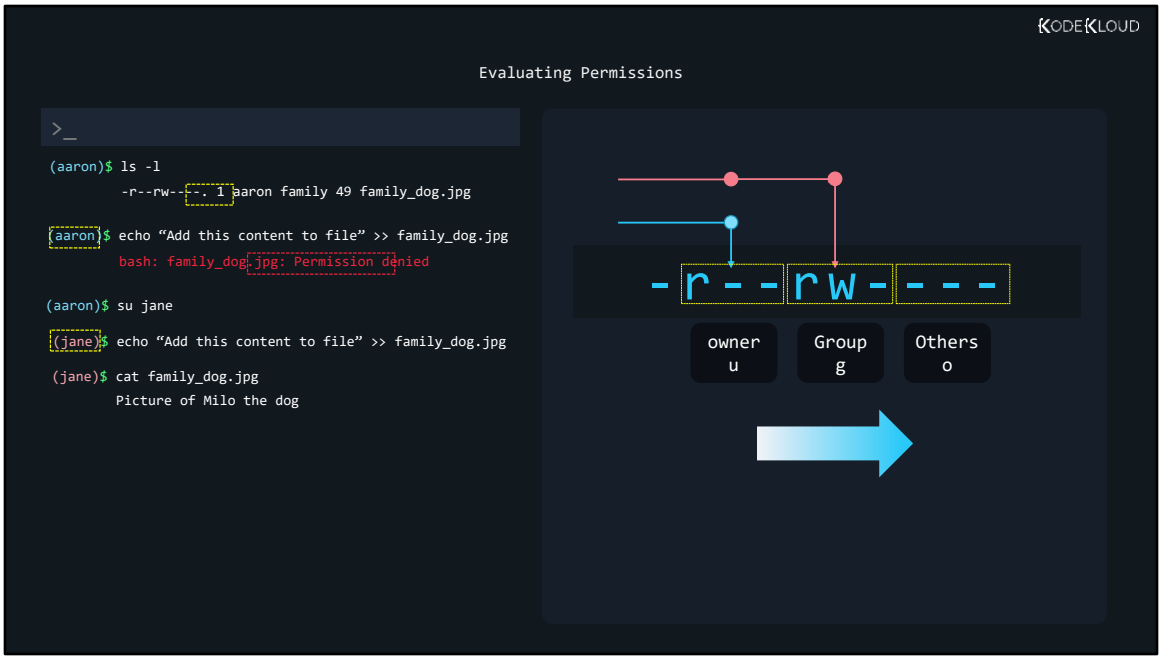


For directories, we must think differently. Unlike a file that may contain text to be read, executed, modified, directories do not have such contents. Their contents are the files and subdirectories they hold. So read, write and execute refers to these files and subdirectories they have inside.

•r means the user, group, or other users can read the contents of this directory. We need an r permission to be able to run a command like "ls Pictures/" and view what files and subdirectories we have in this directory.
•w means the user, group, or other users can write

to this directory. We need w to be able to create or delete entries in that directory (add/delete files or subdirectories), as when we use mkdir.
•x means we can "execute" into this directory. We need x to be able to do "cd Pictures/" and "enter" into the Pictures/ directory.

When directories are meant to be accessible, you'll normally find both the r and the x permissions enabled.



Whenever you're on a Linux system, you're logged in as a particular user.

We've changed permissions in an interesting way to make this easier to understand.

<C> Look at the permissions for the family_dog.jpg file. It’s set to <c> read only for owner, read write for group and no permissions for others.

<c> We see the current owner of the file is aaron.

And we know aaron is part of the family group.

Can aaron write to this file considering the fact that the owner has read-only permissions only? It might seem that he should be able to do that, as he is part of the family group, and that group has rw- (read/write) permissions.

<c> But if we try to add a line of text to this file, it fails.

Why is that? Because permissions are evaluated in a linear fashion, <c> from left to right.

With these permissions in mind:

let's see how the operating system decides if you're allowed to do something.

It goes through a logic like this:

1.Who is trying to access this file? <c>aaron
2.Who owns this file? <c> aaron
3.Ok, current user, aaron, is the owner. <c> Owner permissions apply: r--. aaron can read the file but cannot write to it. <c> Write permission denied!

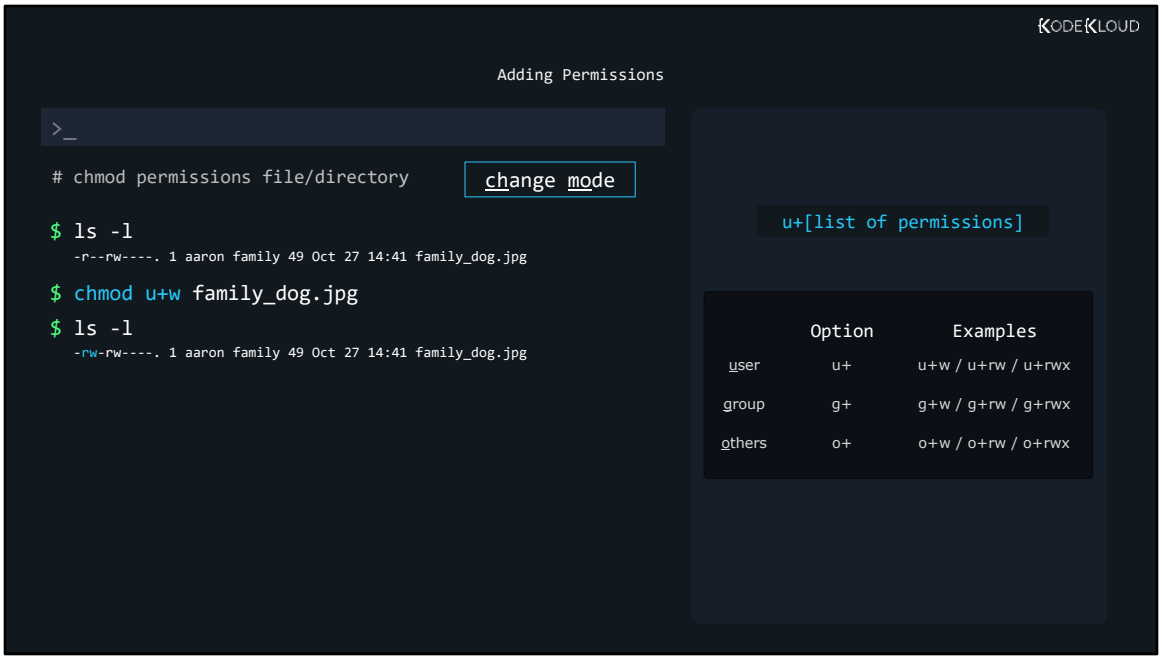
It does not evaluate the permissions of the group because it already matched you to the first set of permissions: the ones for the owner of the file.

<c If you'd be logged in as a different user, for example jane, the logic would be like this:

1.Who is trying to access this file? <c> jane
2.Who owns this file? aaron
3.Ok, owner permissions do not apply, <c> moving on to group permissions
Is jane in the family group? Yes. Ok, <c> group permissions apply: jane has rw- permissions so she can read and write to file.

If the user trying to access the file is not the owner and is also not in the "family" group, the last three permissions would apply, the permissions for **other** users.

Now that we have a basic understanding of permissions, let's move on to how we can change them to suit our needs.



To change permissions, we use the chmod command. The basic syntax of the chmod command is:

chmod permissions file/directory

We can specify these permissions in many ways. Let's start out with simple examples.

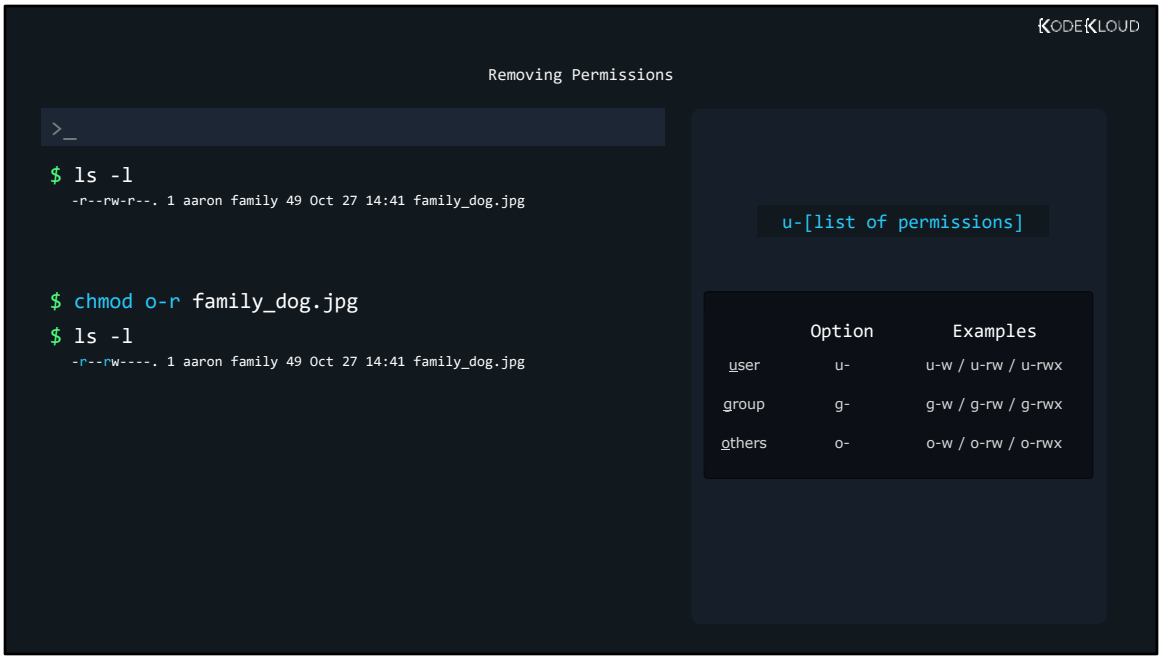
We saw that our owner, aaron, cannot write to this file. Let's fix that. To specify what permissions we want to add, **on top** of the existing ones, we use this syntax:

•To add permissions for the user (owner): u+[list of permissions]. Examples: u+w or u+rw or u+rwx.
•To add permissions for the group: g+[list of permissions].
•To add permissions for other users: o+[list of permissions].

In our case, we want to add the write permission for our user owner of the file:

chmod u+w family_dog.jpg

Now the old r-- becomes rw- with the newly added "w" permission. So we fixed our problem and aaron can write to this file.

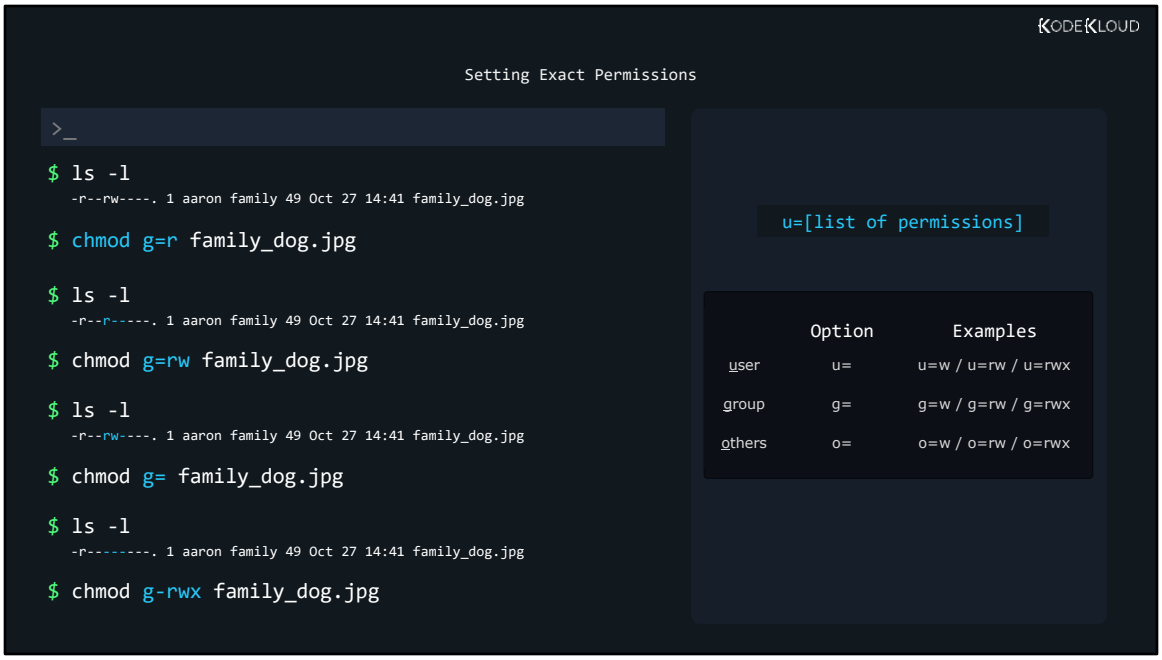


- To remove permissions for the user (owner): u-[list of permissions]. Examples: u-w or u-rw or u-wx.
- To remove permissions for the group: g-[list of permissions].
- To remove permissions for other users: o-[list of permissions].

At this point, we have the permission r-- for other users. That means anyone on this system can read our family_dog.jpg file. If we want only the user owner and group to be able to read it, but hide it from anyone else, we can remove this r permission.

chmod o-r family_dog.jpg

Now only aaron or the family group can read this file, no one else.



With + and - we saw that we can add permissions **on top** of the preexisting ones or **remove some of them** from the preexisting ones.

If a file has rwx and we remove x, we end up with rw-. If another file has r-x and we remove x, we end up with r--. If we only care about removing the execute permission and we don't care what the other permissions are, this is perfect. But, sometimes, we'll have a different requirement. We'll want to make sure that permissions are set **exactly** to certain values. We can do this with the = sign.

Just like before, this is done with the format: u=[list of permissions] or g=[list] or o=[list].

Example: we want to make sure that the group can only read this file, but not write to it or execute it. We can run

chmod g=r family_dog.jpg

We can see that, before, group permissions were rw-. We didn't tell chmod to actually **remove** the w permissions, but by saying g=r, we told it to make the group permissions **exactly**: r--. This only affects the group permissions and not the user or other permissions.

If we'd want to let the group read and write, but not execute, we'd use:

chmod g=rw family_dog.jpg

We can see that whatever letter is missing, will make chmod disable permissions for that thing. No x here means no execute permission will be present on the file.

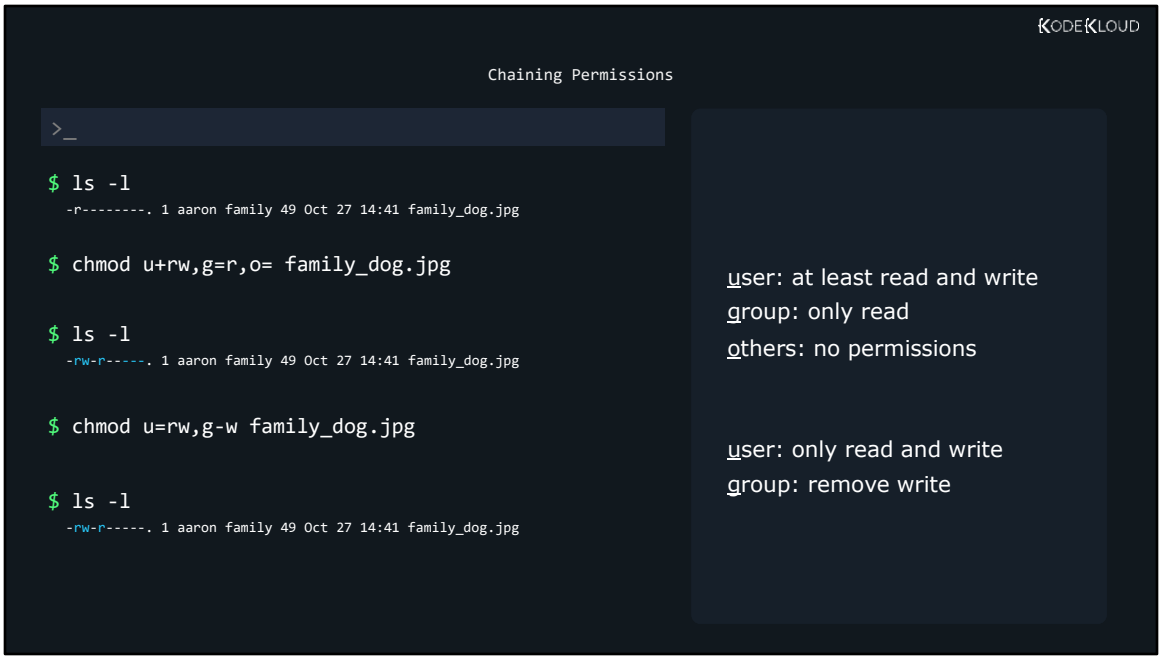
Which leads us to the next thing. What if we omit all letters? No r, no w, no x. This would disable all permissions for the group:

chmod g= family_dog.jpg

This is like saying "make group permissions all empty". Another command that does the same thing is

chmod g-rwx family_dog.jpg

It does the same thing, but following another logic - remove all these permissions for the group: r, w, and x.



We saw how to
•add permissions with +
•remove with -
•set exactly to: with =

We can group all these specifications in one single command by separating our permissions for the user, group and others, with a "," comma.

For example, let's consider this scenario:
1.We want the **user** to be able to **read** and **write** to

the file; don't care if execute permission is on or off.
2. We want the **group** to **only** be able to **read** (exactly this permission).
3.And we want **others** to have **no permissions** at all.

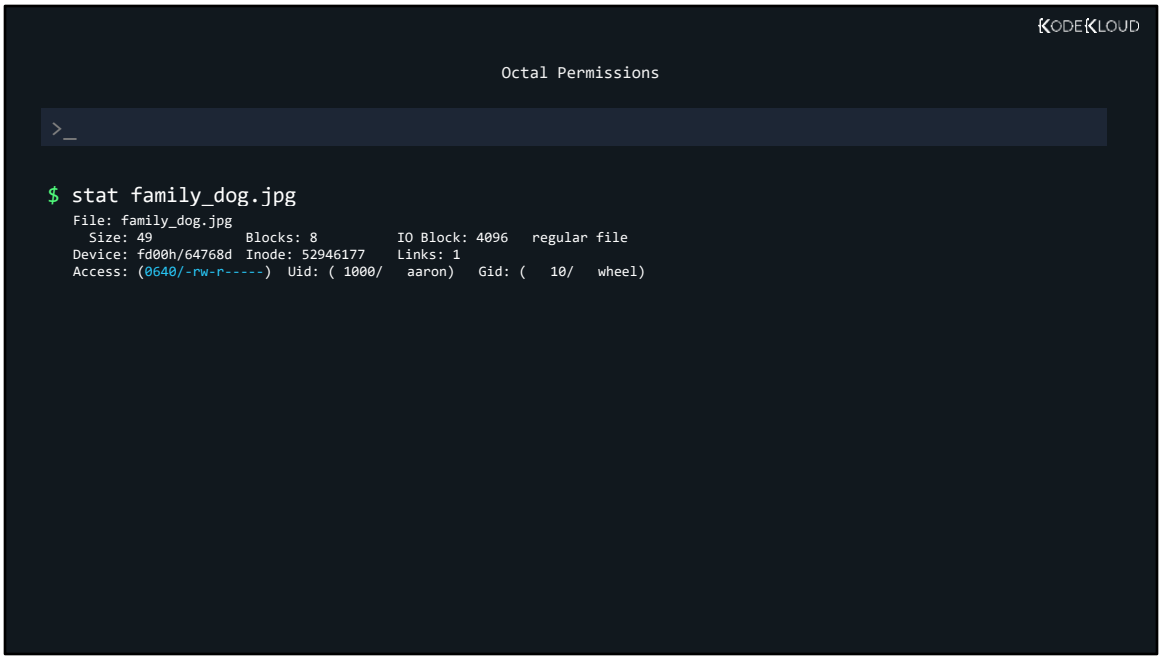
Our command could be:

chmod u+rw,g=r,o= family_dog.jpg

Or, let's say:
1.We want the **user** to **only** be able to **read** and **write.**
2.But we want to **remove** the **write** permissions for the **group** and leave all other group permissions as they were.
3.We don't care about permissions that apply to **other** users.

We would use:

chmod u=rw,g-w family_dog.jpg



chmod supports another way to set/modify permissions: through octal values.

First, let's look at another command that shows us permissions:

stat family_dog.jpg

Here's the list of permissions displayed by **stat.**

We can see rw-r----- has an octal value of 640

(ignore the first 0, that's for special permissions like setuid, setgid and sticky bit). If we break this down, 640 means the user/owner permissions are 6, the group permissions are 4 and the other permissions are 0. How are these calculated?

Let's take a closer look at this permission. We have rw for user, r for group and none set for others. Each permission is represented in binary. If it's set the binary is set to 1 or else its set to 0. In this case the first part has 110, the second part is 100 and the third part is 0. Converting this binary to decimal would give us 6 for the first part, 4 for the second part, and 0 for the third part. Here's a quick binary table for your reference.

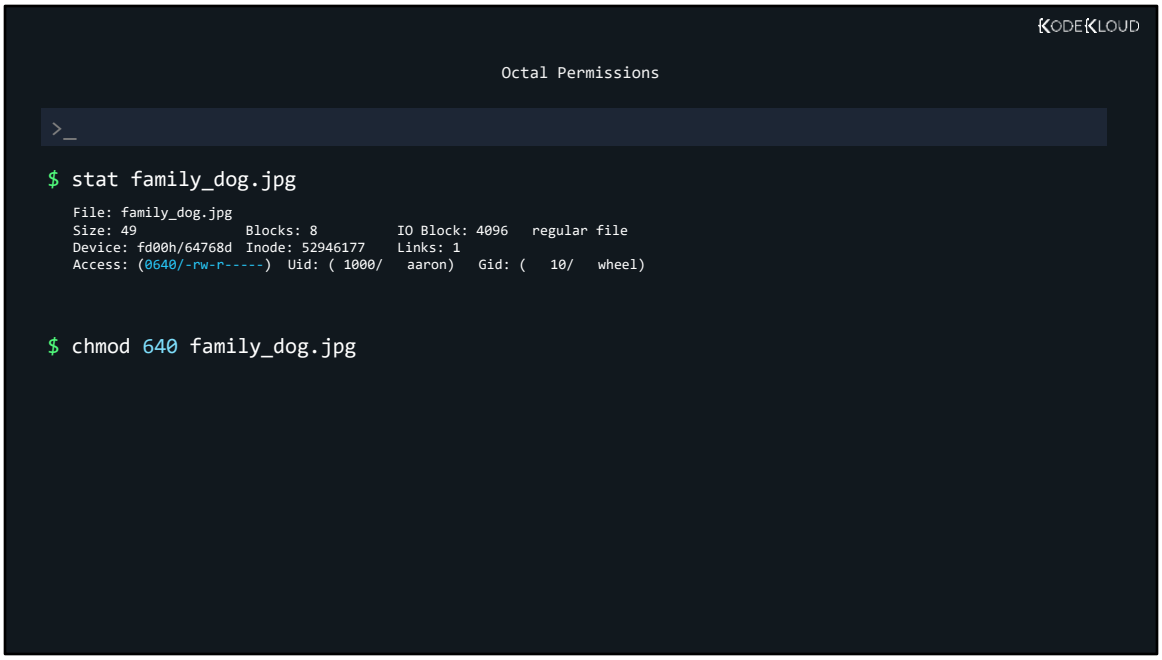
Let's take another example. This time rwx r-x and r-x. So, the binary format would be 111, 101, 101. The decimal of which is 755.

In the last example it's read write execute for all, so its 1 for all bits, and so the decimal value is 777.

if you find binary difficult another approach would be to use the octal table. It's much simpler. For each permission assign an octal value. For example 4 for read, 2 for write and 1 for execute. Then whichever permission is set, consider the respective value for that and for the permission bit not set consider 0. Once done, add up numbers within each group. 4 + 2 = 6 and 4 + 0 + 0 is 4 and the last group is 0.

Let's look at using the same approach for the other examples as well. rwx r-x and r-x gives us 755

and rwxrwxrwx gives us 777.

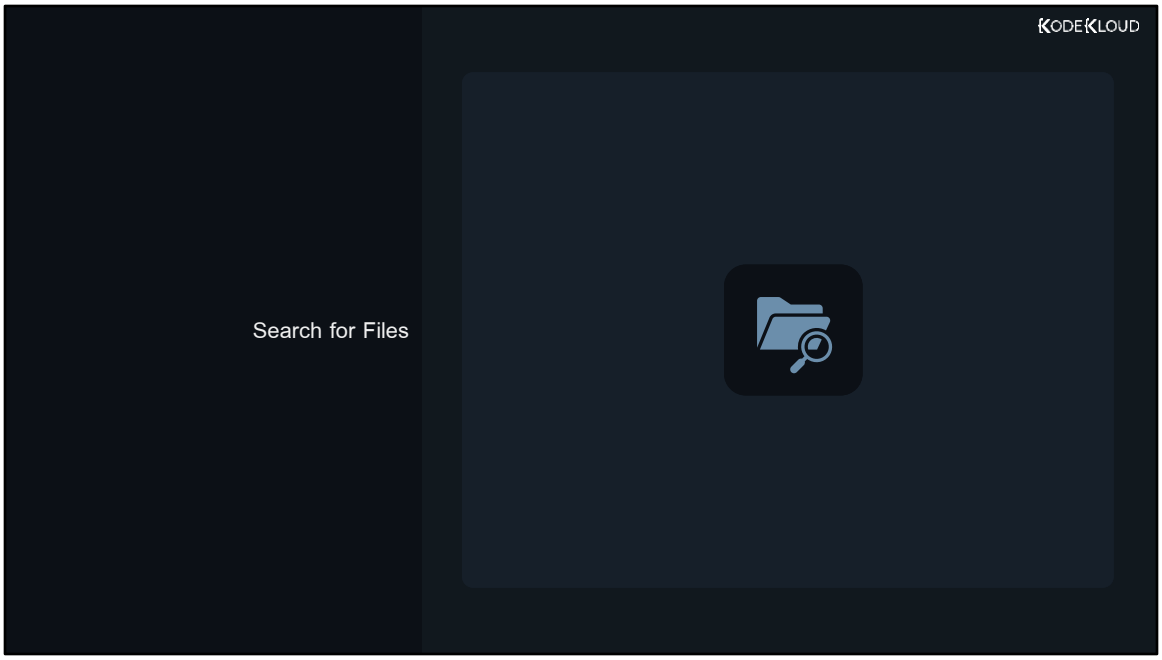


Once we identify the number we want to set to, we can use the same in chmod commands as well. Instead of specifying the permissions for each group, we could just provide a number like this.

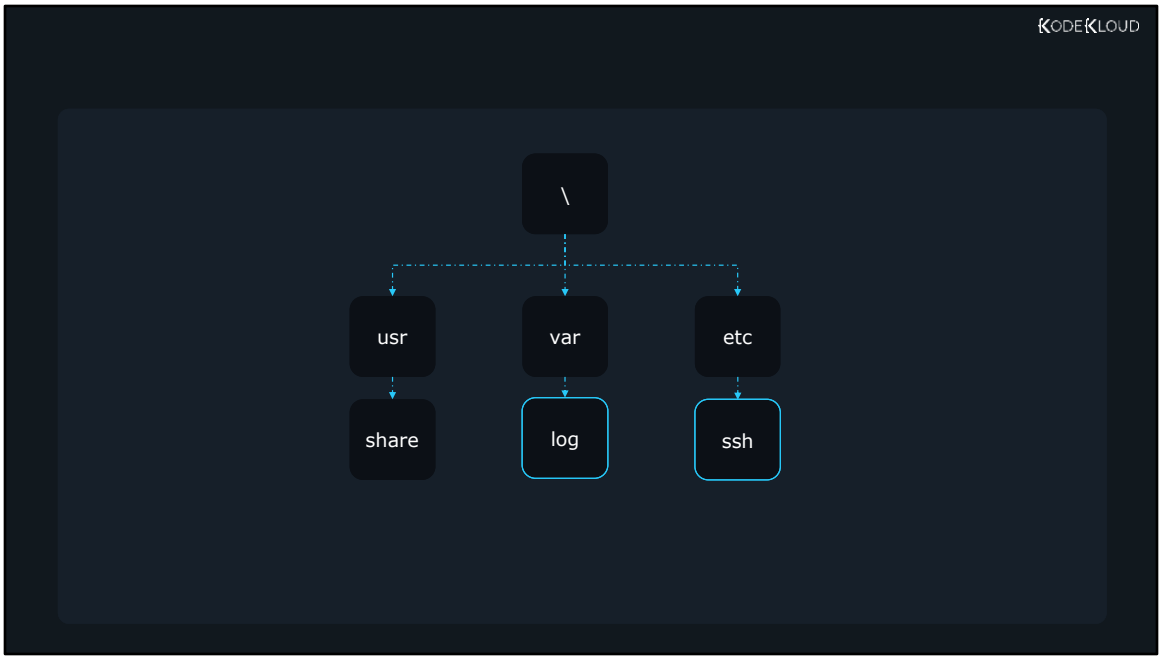
chmod 640 family_dog.jpg

Well, that's all for now, I will see you in the next one.

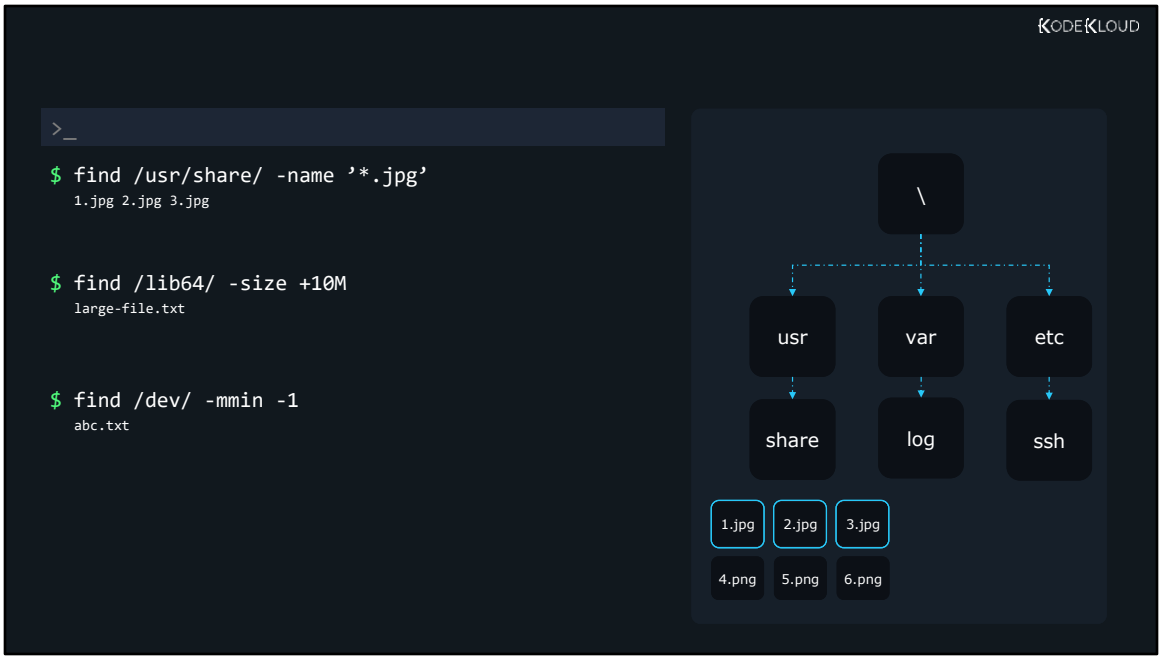
Access the labs associated with this course using this link: https://kode.wiki/linux-labs



Let’s now look at how to search for files in Linux.



After you get a little bit familiar with a Linux OS you learn that files are very nicely organized. If you want to configure your SSH daemon, you'll know you'll find relevant config files in /etc/ssh/. Need to find logged errors? You go to /var/log. Most of the time, you'll know where everything is, at least, approximately. So why would you need to search for files? Let's look at some typical scenarios.



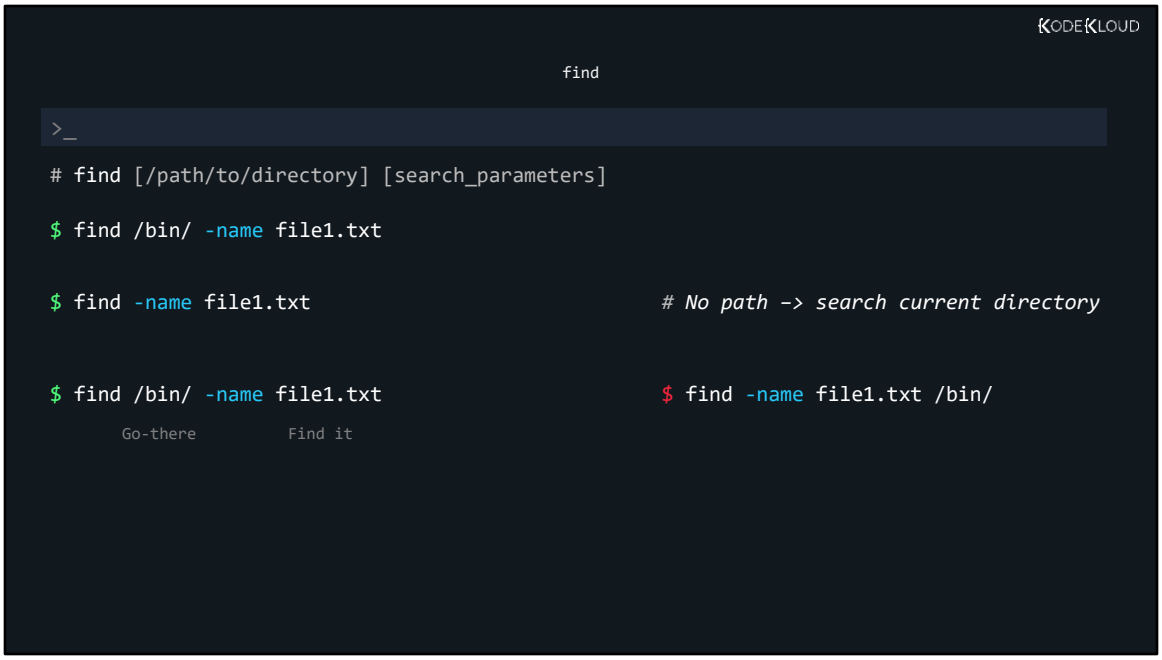
Imagine you have a website. You may want to find all your image files. If your website's directory would be /usr/share/, you could quickly get a list of all .jpg files with a command like:

In a different scenario, you're almost running out of disk space. This server is hosting virtual machines. You notice that most of the virtual machines require files under 20GB. You figure that you can search for files that are larger than 20GB to filter out the abnormally large ones.

We don't have such large files available, but here's how we would look for files larger than 10 megabytes:

Or let's say you've just updated an application and you're curious to see what files were changed. You can quickly look at all files that have been modified in the last minute, with a command like:

Of course, this applies to many other scenarios. Like you could use a similar command to see what configuration files your system administration team changed in the last hour.



From these examples, it's clear that the command to search for files is **find**. Let's take a look at the syntax we'll use throughout this lesson:

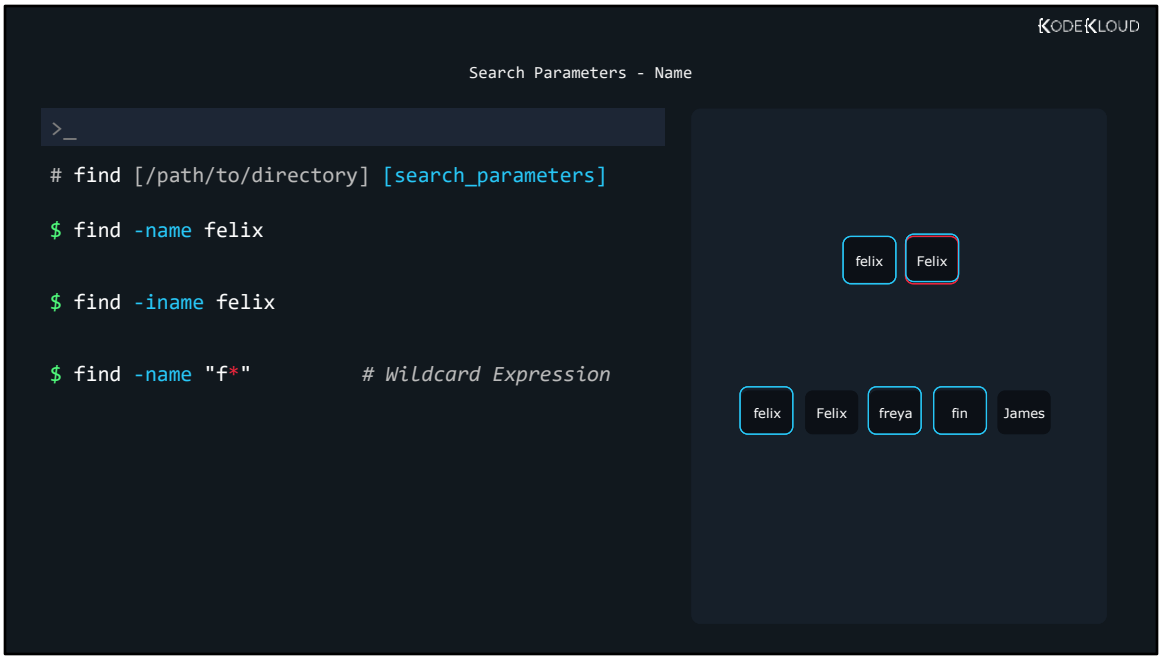
For example to find a file named file1.txt in the directory /bin run the command find /bin –name file1.txt . –name is the search parameter used to specify the name of the file you are looking for.

You can sometimes skip specifying the path to the directory you want to search through. And when you do that it searches in the current directory.

The first few times you'll use this command, it may happen quite often that you mix up the directory path with the search parameters. Meaning, instead of writing find /bin/ -name file1.txt, you may write find –name file1.txt /bin/. If you find yourself falling into this trap, just think about it this way, "First I have to go there, then I will find it". You have to enter your room, and only after you can search for your keys. This will remind you that you first have to specify the search location and then the search parameters.

With this basic knowledge out of the way, let's focus on what makes the real magic

happen, the search parameters.



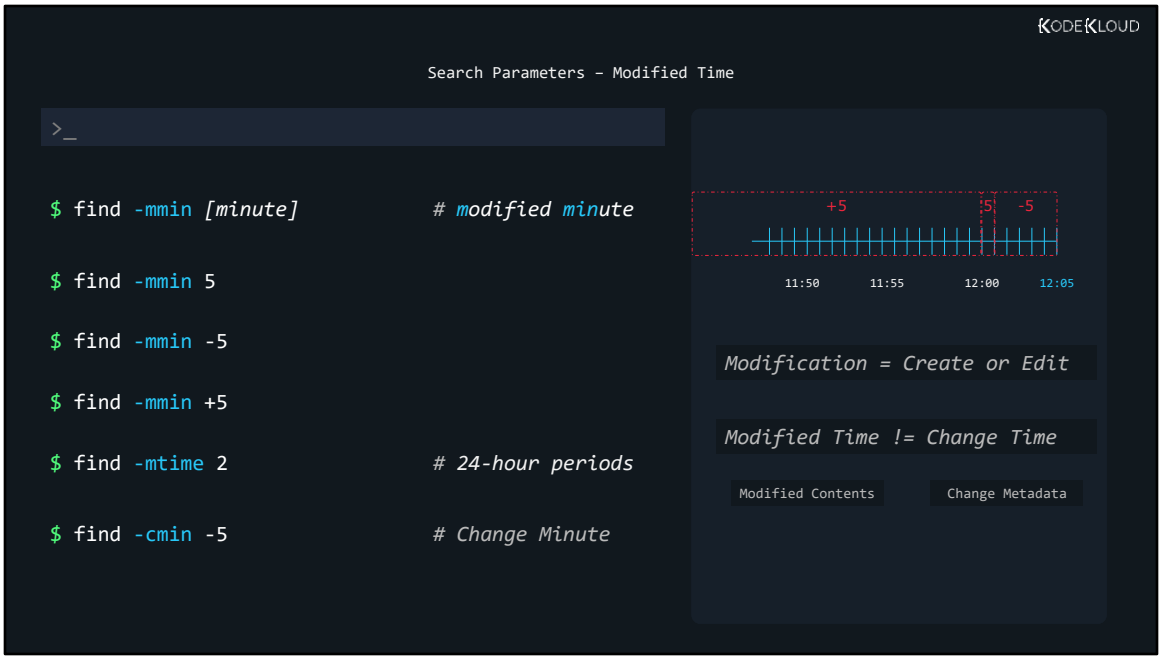
Let's look at some other parameters.

We just saw the name parameter being used already. It is used to find files with a specific name in this case felix.

This however is case sensitive. Meaning it won't find a file named Felix with a capital F.

If you'd like the find command to not be case sensitive, or case insensitive add an i infront of the option to make it iname.

At times you may want to find multiple files that have a pattern in their names. For example, I want to find all files that start with a lowercase f. For this use a wildcard expression, which is a starting expression, followed by a star. The * is like a joker card, for text. It will match anything even if it's 0 characters or 100. In this case it matches all names starting with f.



We already saw, in the examples, a command that looks for files modified in the last minute. It uses the mmin option. To remember "-mmin" think about "**m**odified **min**ute". Let's understand the options in a bit more detail.

Let's say the current time is 12:05. To find files modified 5 minutes ago – that is files modified at the minute 12:01 run the find command with the mmin parameter set to 5. This is going to list files modified in that minute only.

To list all files modified in the last 5 minutes set the minute parameter to -5.

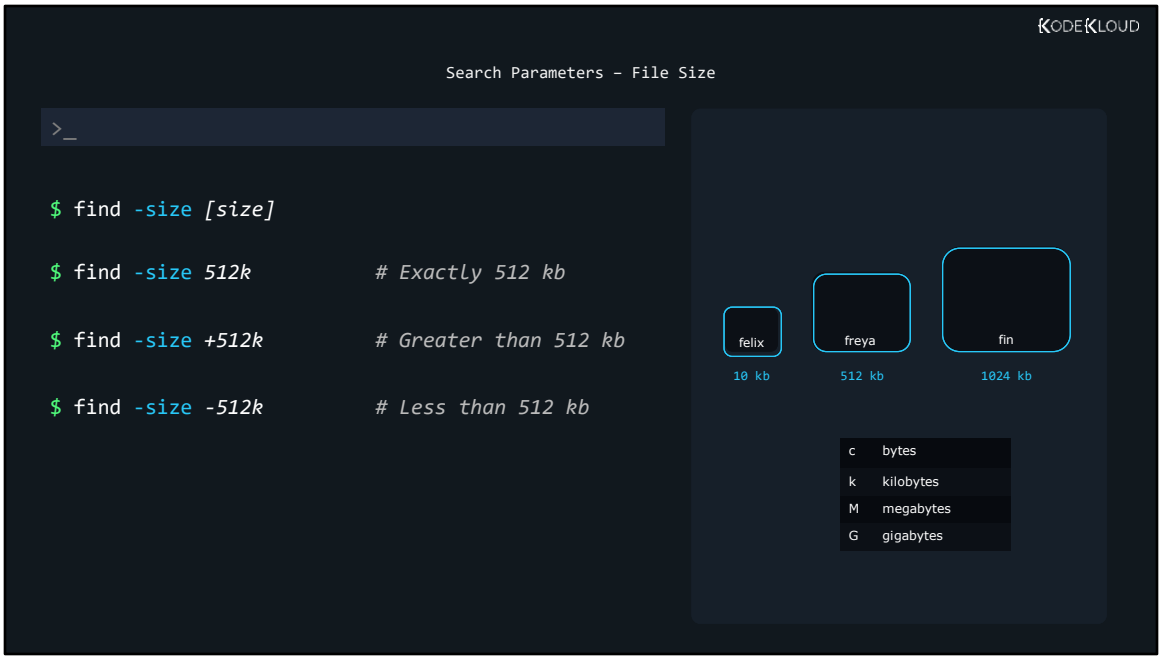
So if there is a -5, there's surely a +5. What do you think that does? I hope you are not thinking its going to list files modified 5 minutes into the future.

With the parameter set to +5, the command lists all files modified before 5 minutes and unto infinity. So any file modified more than 5 minutes ago will be listed.

Another similar option is mtime and it helps search for files modified in days or past 24 hour periods. 0 lists past 24 hours, 1 lists files modified between 24 and 48 hours and so on.

Its worth noting that modification means creation or edition of files.

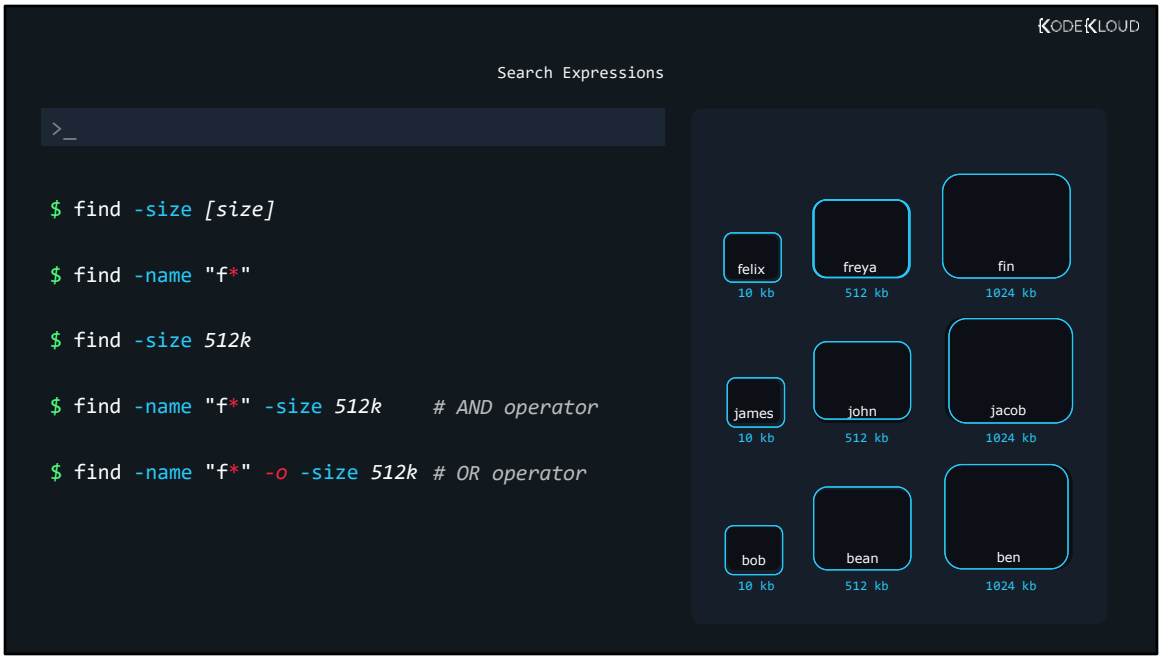
Linux also has a thing called "change" time for files. Which might sound like the same thing as a "modify" time, but it's actually different. Modify time refers to time when contents have been modified. Change time refers to the time when metadata has been changed. Metadata is "data about data", so in this case, "data about your file". This might mean something like file permissions. And this is where change time could be useful. Imagine you suddenly get errors with some app and you suspect it's because someone changed some file permissions in the wrong way. You could find files with permissions changed in the last 5 minutes, with a command like:



In our initial exercises, we used -size to search for files, based on their size.

To find files of size exactly 512 KB run the find command with the size parameter set to 512k. K stands for kilobytes. Here's a quick table showing the different values.
C stands for bytes, k for kilobytes, m for megabytes and g for gigabytes. Note that M and G are capital letters.

To search for files greater than 512 kb use +512 kb and for files less than 512 kb use –512 kb.



The parameters are also at times referred to as search expressions. This is because you can extend the parameter and add more parts to it to create an expression - like in Mathematics.

So we learned that we could find files that start with a letter using the wildcard format like this. So all files starting with the letter f are found.

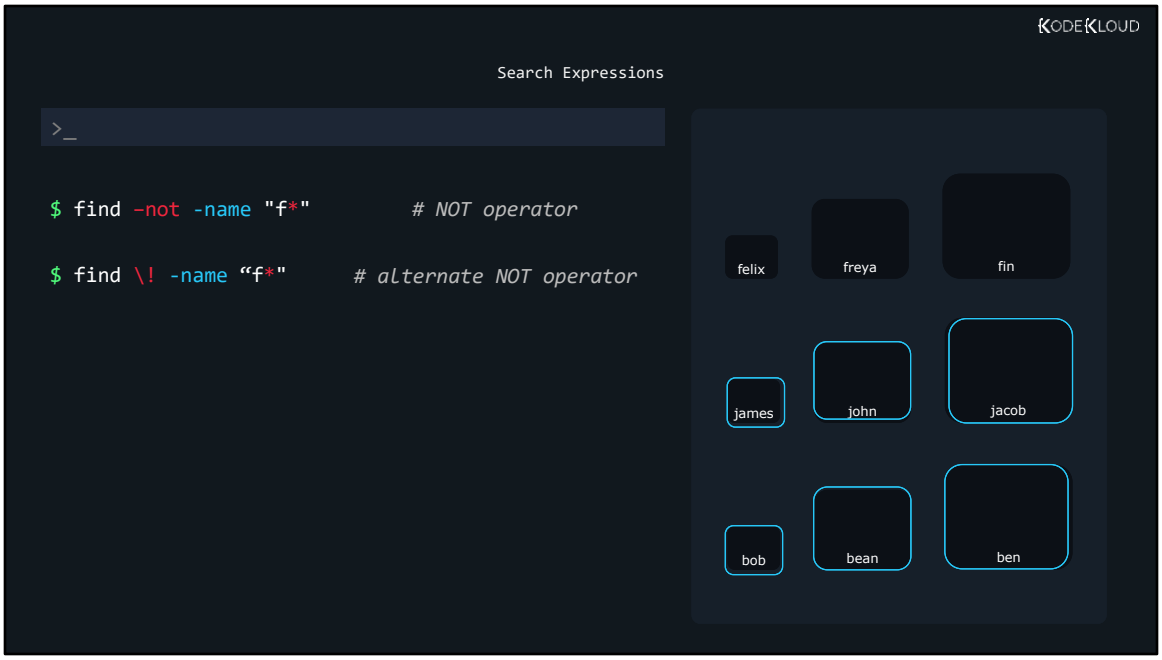
We also learned we can list files by a size using the size parameter like this. All files of size 512kb are listed.

However what if I want to find files that start with the filename f and are also of size 512 kb?

For this you can specify multiple options together in a single command like this. Here I have the name option and the size option. This works like an AND operator. It finds files that match both of these criteria. In our case the file that starts with the letter f and is also 512kb is the file named freya.

But what if we want an OR expression? For example I'd like to find files that match

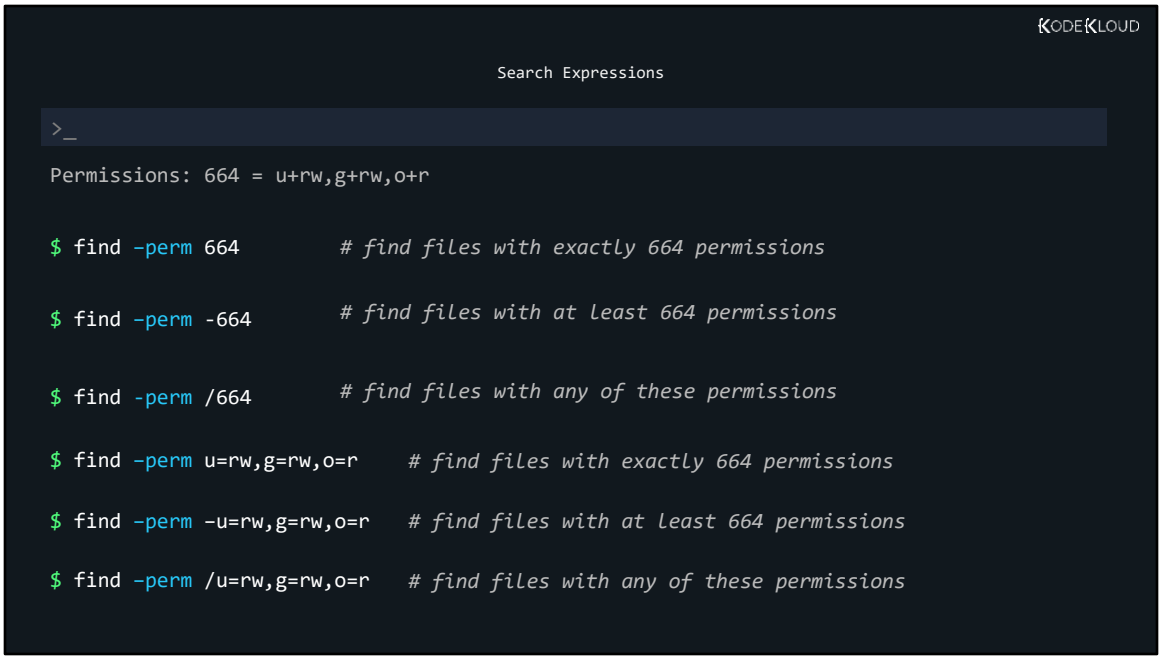
either of these criteria. All files that either start with f or are of size 512kb. For this add the –o flag to the command like this.



Another interesting thing you could insert into an expression is the NOT operator. To make it easy to understand, let's look at another example. Say you want to find all files that do not begin with the letter f. To exclude files beginning with the letter f from our results, we would use the “-not” flag before the “-name” flag, followed by “f*.” This would return a list of file names that do not begin with the letter f.

Another way to write the NOT operator is to use the “!.” Since One important note, however. Our command interpreter, bash, when we write "!" it will

think we want to do some special things, as that's a special character for it. To tell it "Hey bash, ignore this special character and just consider it a regular thing I typed, don't take any special actions" we **escape** it. To escape a character, we just add a backslash "\" in front of it. Our command becomes:
find \! –name “f*”.



We can also search for files based on their permissions. We'll use "664" for our permissions. "664" means this permission: user can read and write, group can read and write, others can read (u+rw,g+rw,o+r).

To search for files based on their permissions, we can use:

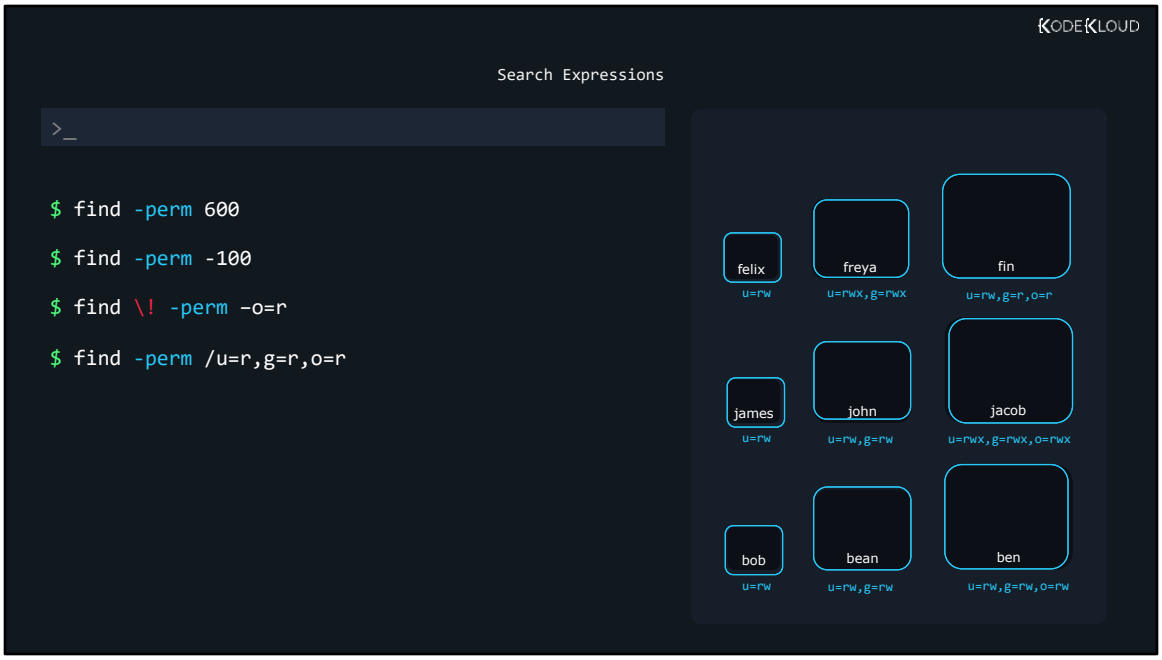
- `-perm 664` to look for files which have exactly these permissions
- `-perm -664` to look for files which have **at least**

these permissions. Which means that even if the file has some extra permissions set, it will still show up in the search results. But if it has less than these permissions, it won't show up. For example, 664 denotes that a user should have read and write permissions. If they only have read permissions but no write, then find will not show this in the search result. Think of it as "bare minimum permissions are these:"

- `-perm /664` to look for files which have any of these permissions. Unlike the "bare minimum" condition above, this is more inclusive. For example, if a user can read the file, but cannot write to it, it will still show up in search results, as one permission has been matched, u=r, so it does not matter if other permissions exist or don't exist.

An alternative way to write each of these is:

- `-perm u=rw,g=rw,o=r`
- `-perm -u=rw,g=rw,o=r`
- `-perm /u=rw,g=rw,o=r`



Suppose we have a group of files.

We want to find files which only the **owner** can **read** and **write**, and **no other permissions** are set, we would run **find –perm 600.** This would match the files, “felix,” “james,” and “bob.”

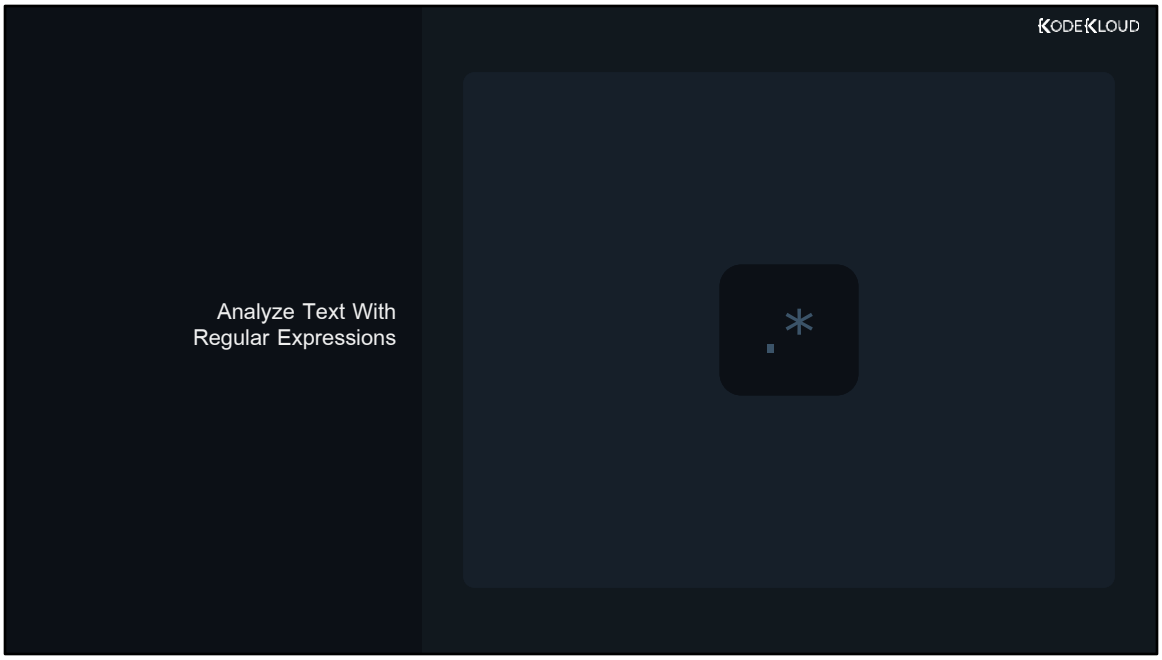
To find files that the **owner** can **execute at least**, but rest of permissions **can be anything**, we would run **find –perm -100**, which would match only “freya” and “jacob.”

Now, imagine we want to make sure that **nobody**

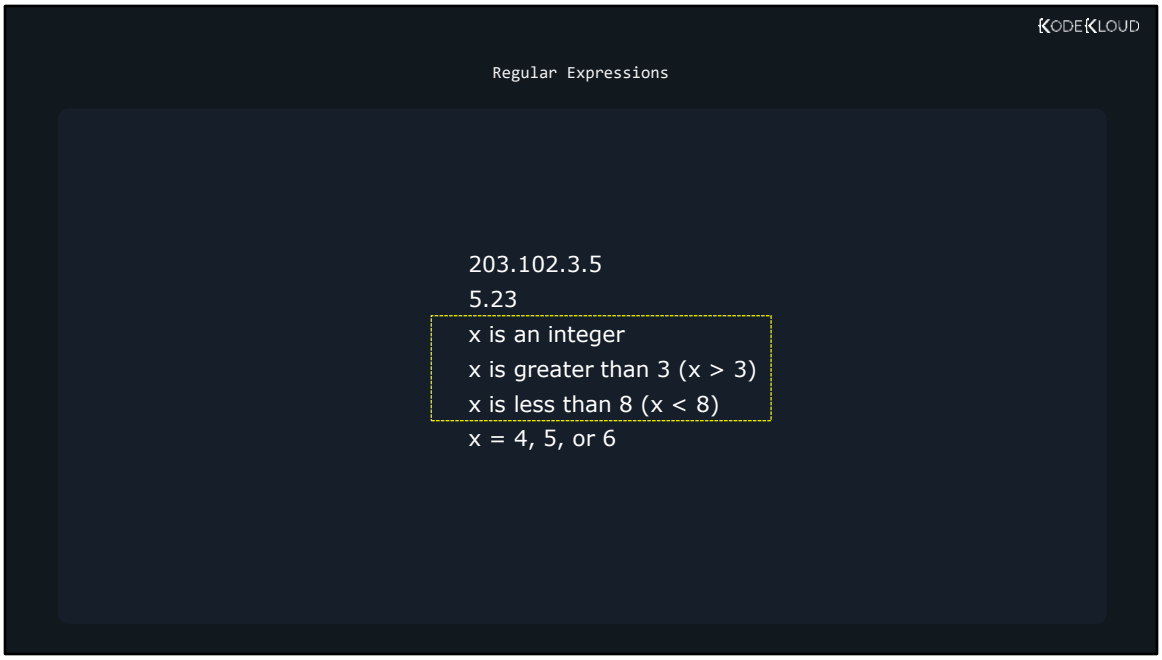
else can **read** these files, except **users and groups that own them**. In this case, we use the **NOT operator**. To look for files that **others can NOT read**, we would run **find \! –perm –o=r**, which matches “felix,” “james,” “bob,” “freya,” “john,” and “bean.”

Finally, to find files that can be **read by either the user, or the group, or others** -- does not matter who it is -- but **at least one of them should be able to read**. To do this, we would run **find –perm /u=r,g=r,o=r.** In this case, all our files match the condition. **If no one can read it, it won't show up in the results**.

Access the labs associated with this course using this link: https://kode.wiki/linux-labs



Let’s look at analyzing text using basic regular expressions in Linux.



In our previous commands, we used simple search patterns, looking for some specific pieces of text, like "centos". But what if we need more complex search conditions?

Imagine we have some application code scattered in hundreds of files. And we need to extract all IP addresses used in this app. That would require more advanced search instructions. An IP has a form like 203.102.3.5. But we can't just make a search pattern look for numbers with a . between them, as this would also match numbers like "5.23", which are not IP addresses.

In math, we can say something like:

•

x is an integer

•x is bigger than 3, x>3

•x is smaller than 8, x<8

And this would mean x is either 4, 5, 6 or 7. Regular expressions work in a similar way. We specify some conditions, tie all of them together, and our search pattern only matches what perfectly fits within those conditions.

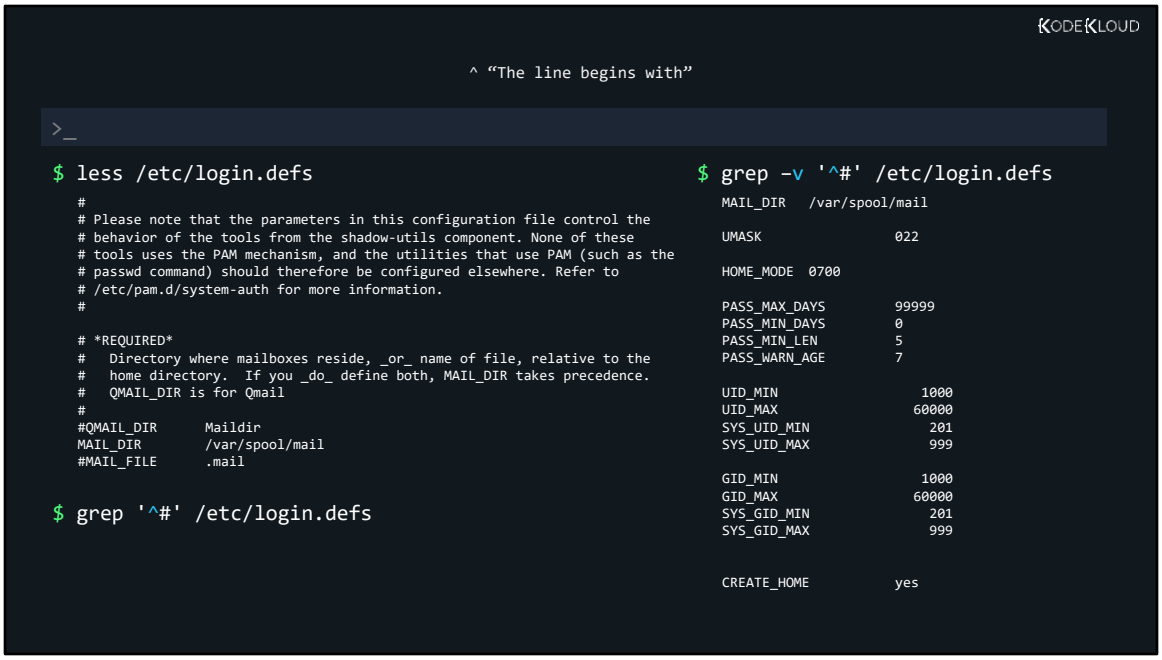
Let's start out with some super simple examples and then build up to slightly more advanced expressions.

All regular expressions are built with the help of operators like:

- ^ (caret)
- $ (dollar sign)
- . (period)
- * (asterisk)
- + (plus sign)
- { } (braces)
- question mark

- vertical pipe
- brackets
- parenthesis
- brackets with caret

Let's see what each of them does.



In Linux, configuration files can have lines that begin with a # sign. These are called "commented lines". They are inactive. The program looking for settings in such a file will ignore all lines that begin with a #. But comments are useful for humans, as they let us see examples of config settings in that file, and descriptions for what they do, without interfering with the program that reads them.

This means that we can search for commented lines, specifically, by creating a regular expression that looks for all lines that begin with a "#".

The regular expression would be:

^#

And we can use it in grep like this:

grep '^#' /etc/login.defs

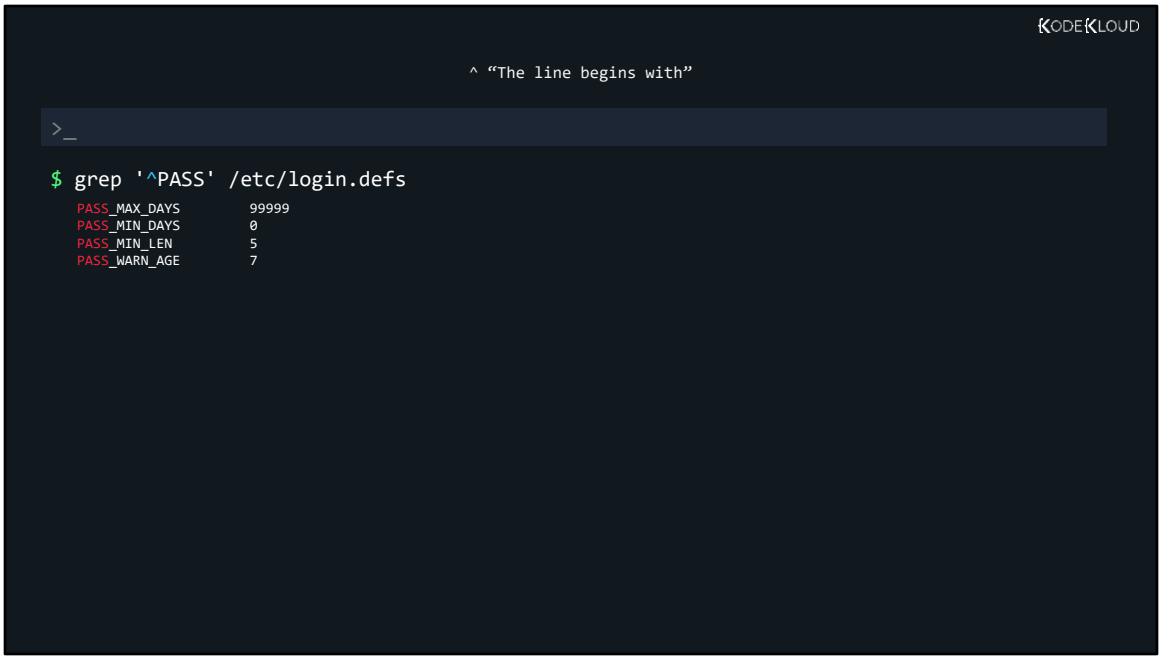
But this doesn't seem to be terribly useful. However, combined with grep's option to invert results, it becomes so.

By inverting we tell grep to show us lines that **don't begin with a # sign**.

grep **-v** '^#' /etc/login.defs

And boom! Now we can see exactly what we wanted: settings actively used.

Imagine how useful this would be in a very long file with hundreds of comments that make it hard to spot what you're looking for!



And just to show another example, we could look for **lines** that start exactly with these four letters: PASS.

grep '**^PASS**' /etc/login.defs

Now let's imagine a different scenario. We need to change a setting that is currently set to "7" days. Easy enough, we could look for a 7, right?

grep '7' /etc/login.defs
But this shows us some stuff we don't need.

However, we know that this file uses this syntax: VARIABLE NAME [space] VARIABLE VALUE. The variable value is last. Which means that if some variable is set to have a value of 7, this number will

be the last character on the line.

We can tell our regex to look for a line that ends with "7", with this expression:

7$

In grep, we'd use it like this:

grep '7$' /etc/login.defs

Clean result!

Just like with ^, with $ we can look for lines that end with a sequence of characters. To look for all lines that end with the text "mail":

grep '**mail$**' /etc/login.defs

Please take note how these operators are placed differently

mail**$**
^PASS

If you mix up their location you won't get any results, which can lead to confusion why your regex is not working. To easily remember their locations, think like this:

•The "line **begins** with" operator, **^**, should be placed at the **beginning** of my search pattern.
The "line **ends** with" operator, **$**, goes at the end of my pattern.

Anywhere you add a . in your expression, it will match any character in that spot. For example:

c.t will match cat, cut, cit, cot, and even c1t or c#t. But it won't match ct. There must be exactly one random character between c and t. With c..t there have to be two characters.

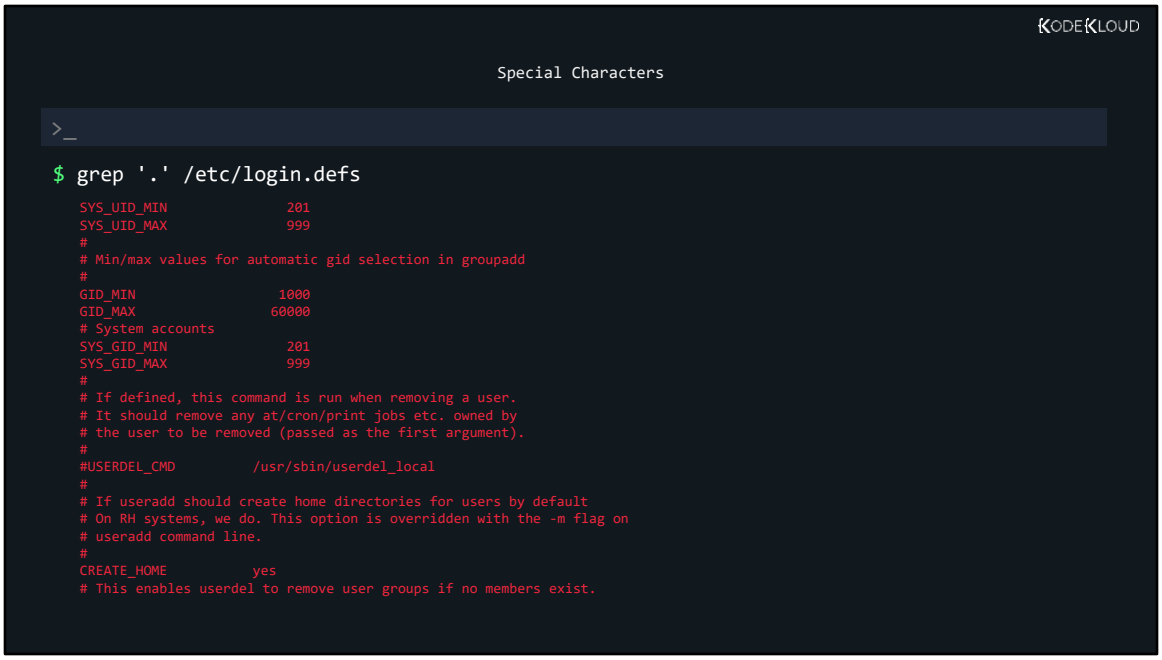
Example grep command:

grep -r 'c.t' /etc/

We can see that even "exe**cut**e" is a match because that sequence fits inside that word

. If we'd only want to match whole words with this and not parts of words, we can use grep's -w option

grep **-w** -r 'c.t' /etc/

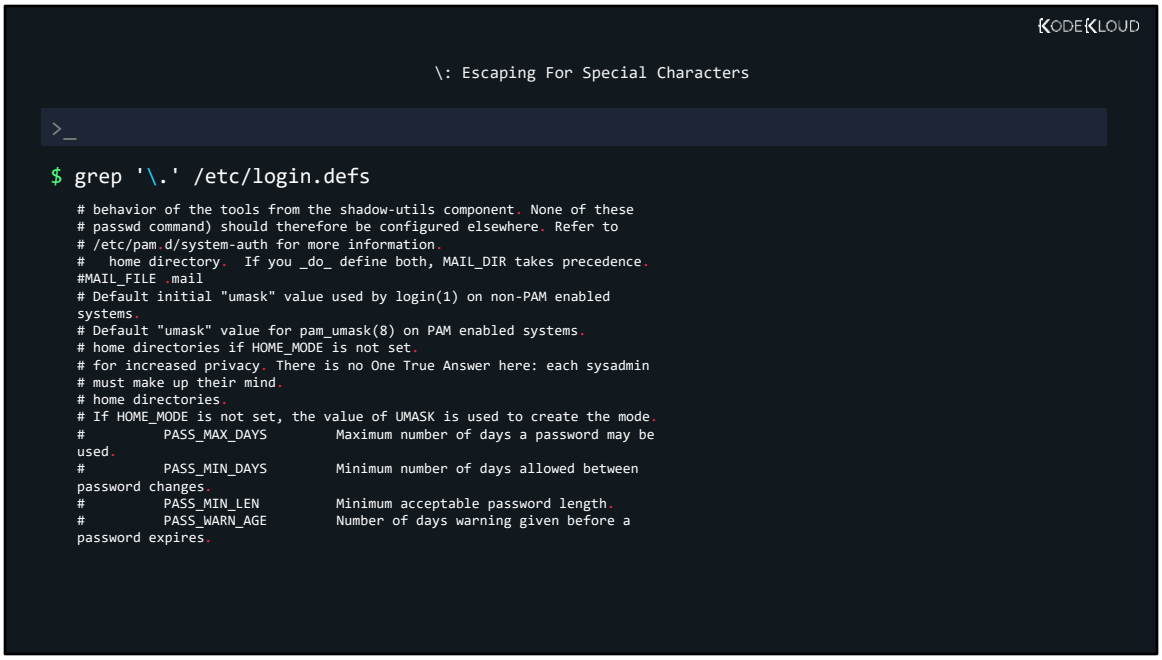


And this brings us to an interesting problem. This . has a special meaning in regex. But what if we need to search for an actual . in our text?

This won't work:

grep '.' /etc/login.defs

as this regex will basically match each character, one by one.



The solution, however, is simple. We look for a regular . by **escaping** this. Escaping is how we tell our regular expression "Hey, don't consider this . a **match any one character** operator. Instead, interpret it as a regular ".".

To escape some special character we just add a backslash \ before it. Instead of

.

we write

\.

So our grep command becomes:

grep '\.' /etc/login.defs

An expression like:

let*

will match le, let, lett, lettt, and so on, no matter how many "t"s at the end. Another way of saying this is that the * allows the previous element to:

- be omitted entirely
- appear once
- appear two or more times

In a grep command, we'd use it like this:

grep -r 'let*' /etc/

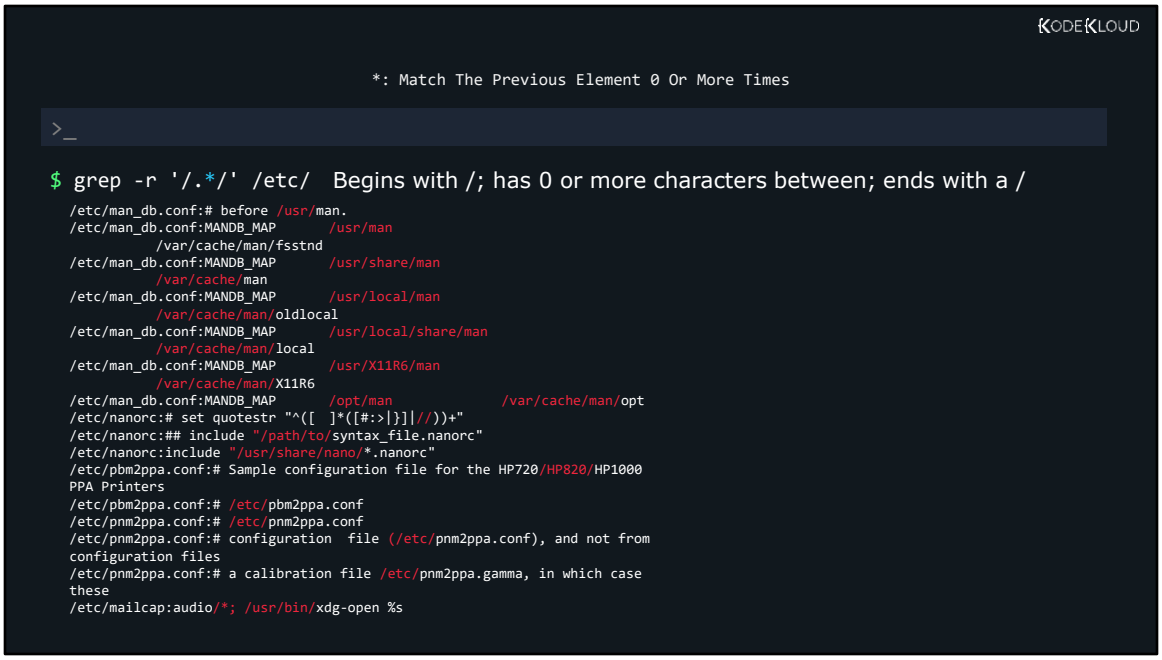
The * operator can be paired up with other operators. For example, to look for something for sequences that begin with a /, have 0 or more characters and between, and end with another /, we could use:

/.*/

Since . matches any ONE character and * says "previous element can exist 0, 1, 2 or many more times" we basically allow any sequence of characters to exist between / and /.

We can now use this in grep:

grep -r '/.*/' /etc/

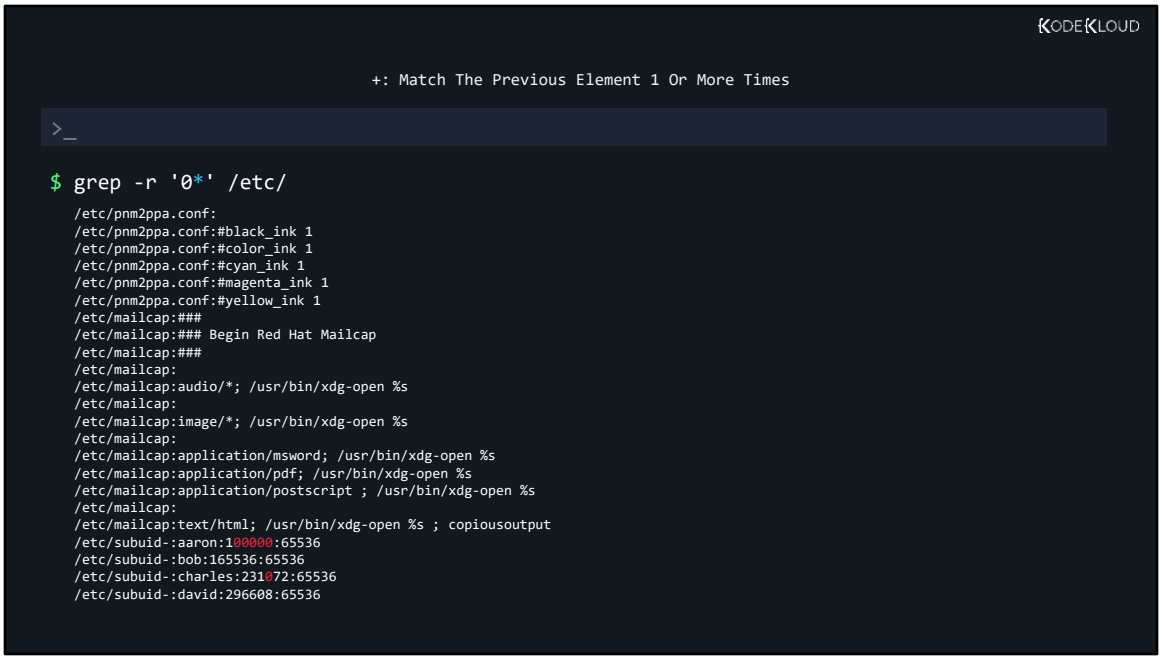


The * operator can be paired up with other operators. For example, to look for something for sequences that begin with a /, have 0 or more characters and between, and end with another /, we could use:

/.*/

Since . matches any ONE character and * says "previous element can exist 0, 1, 2 or many more

times" we basically allow any sequence of characters to exist between / and /.



Let's say we want to find all sequences of characters where 0 appears one or more times. We might be tempted to use:

grep -r '0*' /etc/

But this also matches lines that contain no zeroes at all. Why is that? Because * lets the previous character exist one or more times, but also ZERO times. It basically allows that element to be optional in our search. So, we need another operator that

forces the element to exist **at least** one time, or many more. + does this:

0+

would find strings like:

0
00
000
0000

and so on

We might think we can write this in grep like this:

grep -r '0+' /etc/

But this doesn't look like the result we want. Our +

works like a literal + instead of an operator. Why is this? By default, grep uses "basic regular expressions".

Its manual page has this to say: "In basic regular expressions the meta-characters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \)"

That means, to use "+" as an operator here, we have to add a \ before it, make it "\+". Our command becomes:

```
grep -r '0\+' /etc/
```

But this can become confusing really fast. We saw we already use something like \. to turn the . operator into a regular . Now we use \ to turn a regular + into the + operator. It will be hard to keep track of what to backslash and what not to. So we can go the easier route, use "extended regex" instead, which doesn't require us to backslash ?, +, {, |, (, and).

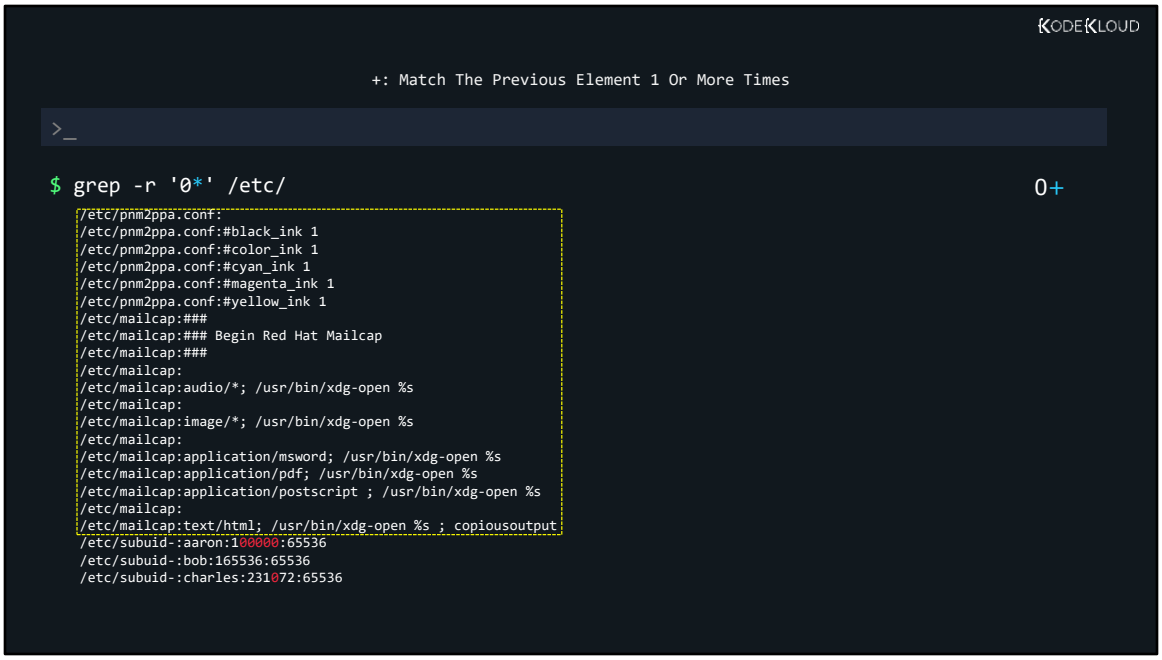
We use extended regex by adding the -E option to grep

grep -E -r '0+' /etc/

Or even easier, we use the equivalent egrep command. Using "egrep" is the same as typing "grep -E".

egrep -r '0+' /etc/

So you can make it a habit to always use egrep instead of grep, to avoid mistakes where you forgot to backslash one of the regex operators.



Let's say we want to find all sequences of characters where 0 appears one or more times. We might be tempted to use:

grep -r '0*' /etc/

But this also matches lines that contain no zeroes at all. Why is that? Because * lets the previous character exist one or more times, but also ZERO times. It basically allows that element to be optional in our search. So, we need another operator that

forces the element to exist **at least** one time, or many more. + does this:

0+

would find strings like:

0
00
000
0000

and so on

We might think we can write this in grep like this:

grep -r '0+' /etc/

But this doesn't look like the result we want. Our +

works like a literal + instead of an operator. Why is this? By default, grep uses "basic regular expressions".

Its manual page has this to say: "In basic regular expressions the meta-characters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \)"

That means, to use "+" as an operator here, we have to add a \ before it, make it "\+". Our command becomes:

```
grep -r '0\+' /etc/
```

But this can become confusing really fast. We saw we already use something like \. to turn the . operator into a regular . Now we use \ to turn a regular + into the + operator. It will be hard to keep track of what to backslash and what not to. So we can go the easier route, use "extended regex" instead, which doesn't require us to backslash ?, +, {, |, (, and).

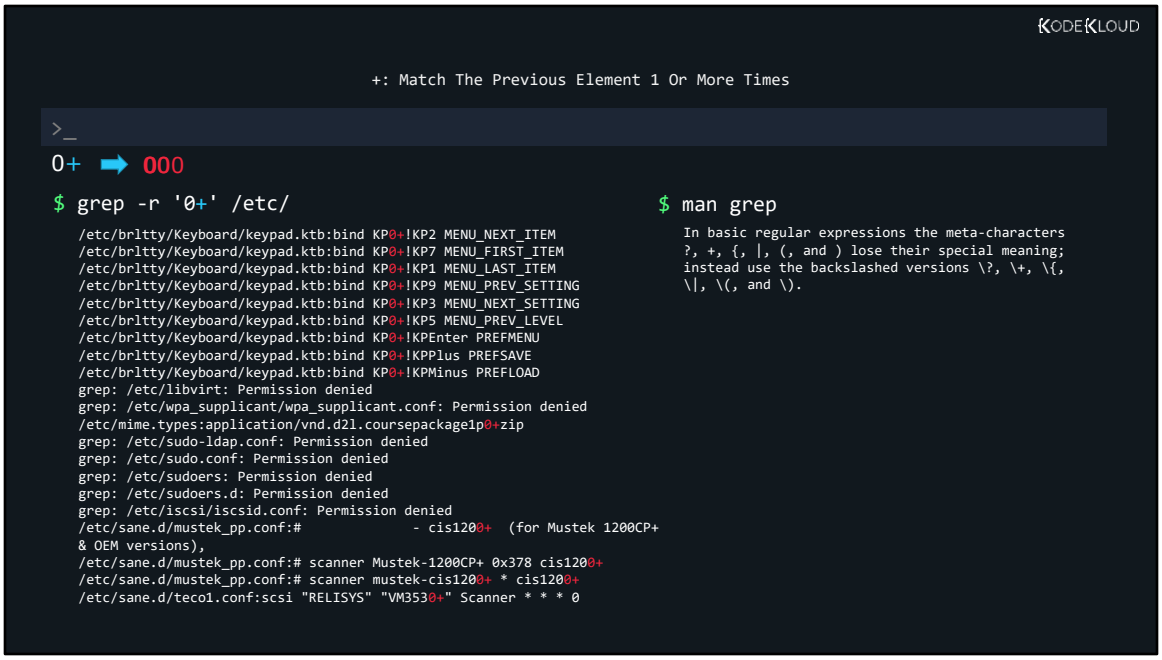
We use extended regex by adding the -E option to grep

grep -E -r '0+' /etc/

Or even easier, we use the equivalent egrep command. Using "egrep" is the same as typing "grep -E".

egrep -r '0+' /etc/

So you can make it a habit to always use egrep instead of grep, to avoid mistakes where you forgot to backslash one of the regex operators.



0+

would find strings like:

0
00
000

and so on.

We might think we can write this in grep like this:

```
grep -r '0+' /etc/
```

But this doesn't look like the result we want. Our + works like a literal + instead of an operator. Why is this? By default, grep uses "basic regular expressions".

Its manual page has this to say: "In basic regular expressions the meta-characters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \)"

That means, to use "+" as an operator here, we have to add a \ before it, make it "\+". Our command becomes:

```
grep -r '0\+' /etc/
```

But this can become confusing really fast. We saw we already use something like \. to turn the . operator into a regular . Now we use \ to turn a regular + into the + operator. It will be hard to keep track of what to backslash and what not to. So we can go the easier route, use "extended regex" instead, which doesn't require us to backslash ?, +, {, |, (, and).

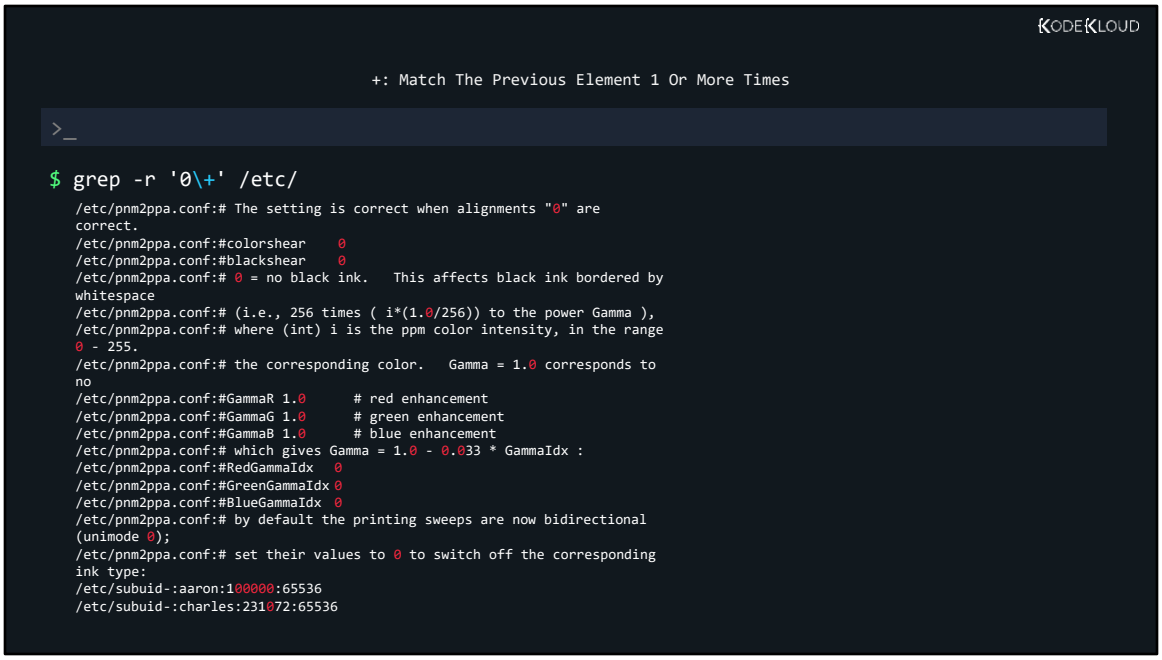
We use extended regex by adding the -E option to grep

```
grep -E -r '0+' /etc/
```

Or even easier, we use the equivalent egrep command. Using "egrep" is the same as typing "grep -E".

```
egrep -r '0+' /etc/
```

So you can make it a habit to always use egrep instead of grep, to avoid mistakes where you forgot to backslash one of the regex operators.



That means, to use "+" as an operator here, we have to add a \ before it, make it "\+". Our command becomes:

grep -r '0\+' /etc/

But this can become confusing fast. We saw we already use something like \. to turn the . operator into a regular . Now we use \ to turn a regular + into the + operator. It will be hard to keep track of what

to backslash and what not to. So, we can go the easier route, use "extended regex" instead, which doesn't require us to backslash ?, +, {, |, (, and).

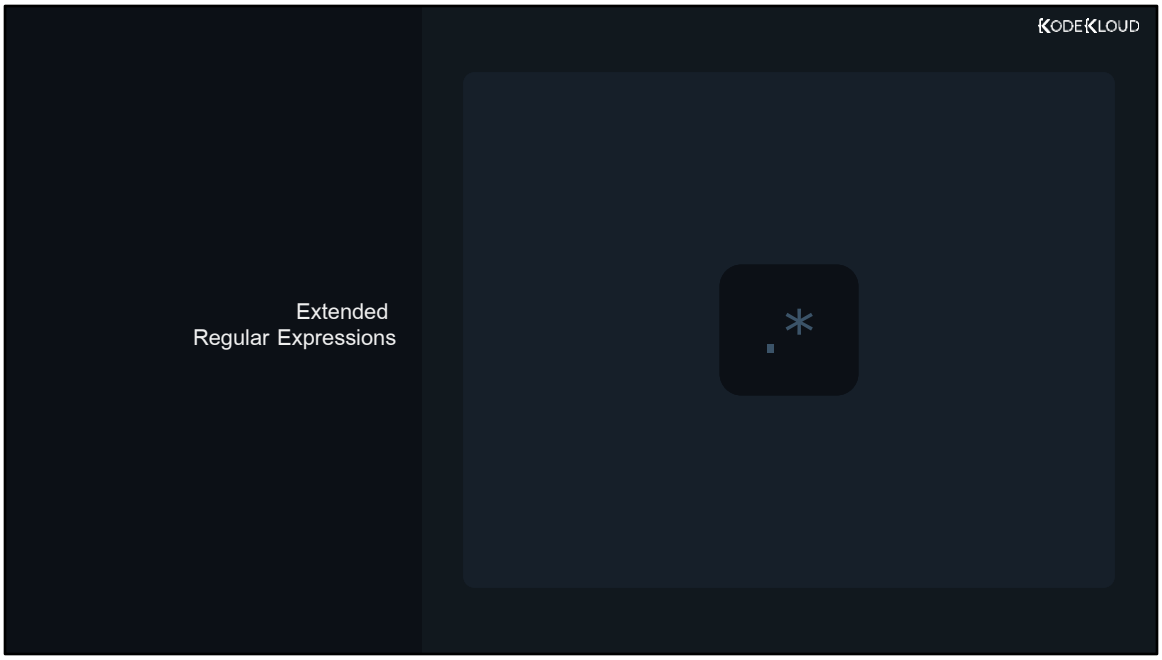
We use extended regex by adding the -E option to grep

grep -E -r '0+' /etc/

Or even easier, we use the equivalent egrep command. Using "egrep" is the same as typing "grep -E".

egrep -r '0+' /etc/

So, you can make it a habit to always use egrep instead of grep, to avoid mistakes where you forgot to backslash one of the regex operators.



Let’s look at analyzing text using basic regular expressions in Linux.

We saw we already use something like \. to turn the . operator into a regular . Now we use \ to turn a regular + into the + operator. It will be hard to keep track of what to backslash and what not to. So, we can go the easier route, use "extended regex" instead, which doesn't require us to backslash ?, +, {, |, (, and).

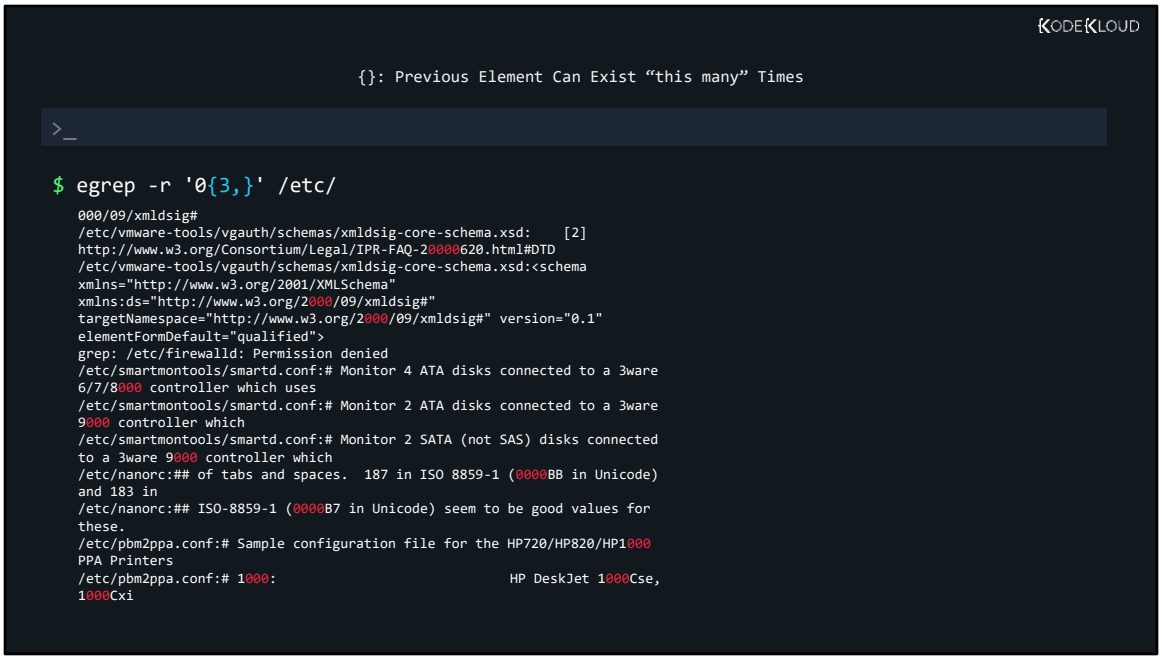
We use extended regex by adding the -E option to grep

grep -E -r '0+' /etc/

Or even easier, we use the equivalent egrep command. Using "egrep" is the same as typing "grep -E".

egrep -r '0+' /etc/

So, you can make it a habit to always use egrep instead of grep, to avoid mistakes where you forgot to backslash one of the regex operators.



To find all strings that contain **at least 3** zeros:

0{3,}

egrep -r '0{3,}' /etc/

To find all strings that contain "1" followed by **at most** 3 zeroes:

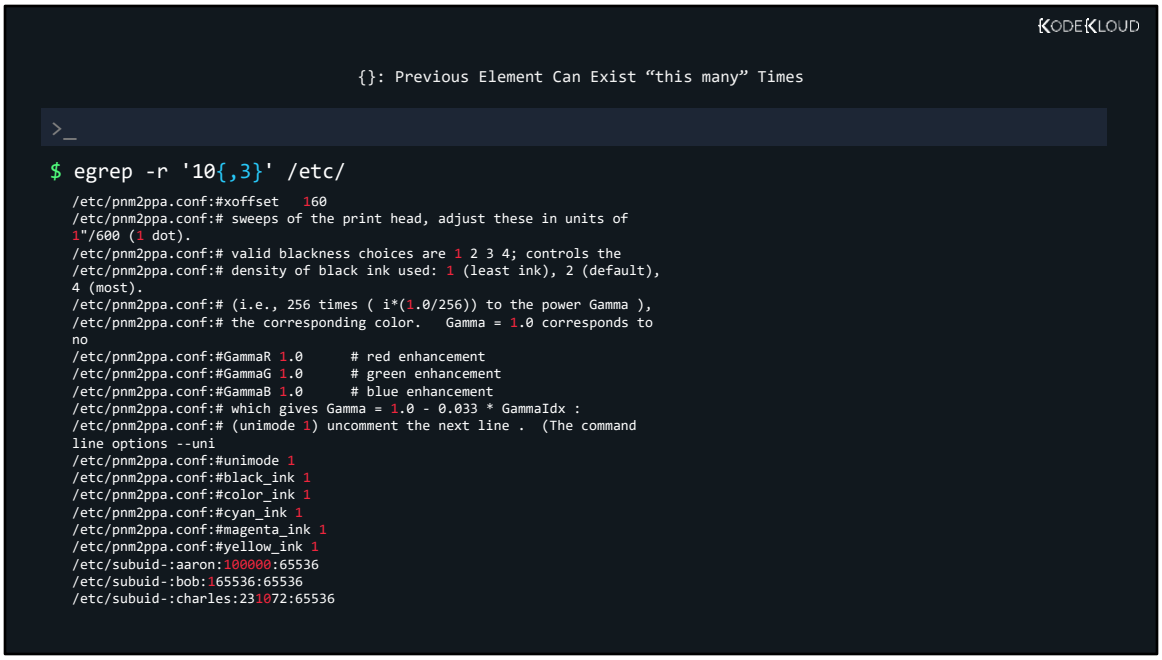
10{,3}

egrep -r '10{,3}' /etc/

And to find all strings that contain **exactly** three zeroes:

0{3}

egrep -r '0{3}' /etc/



To find all strings that contain "1" followed by **at most** 3 zeroes:

10{,3}

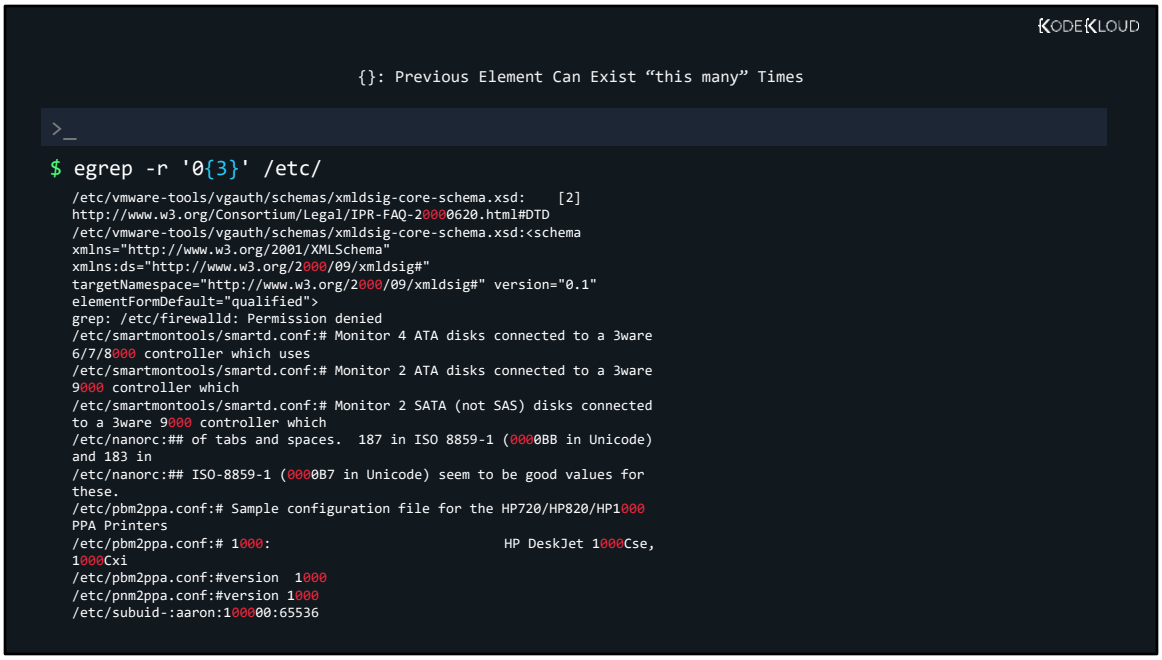
egrep -r '10{,3}' /etc/

Note: This will also match 1s followed by no zeroes.

And to find all strings that contain **exactly** three zeroes:

0{3}

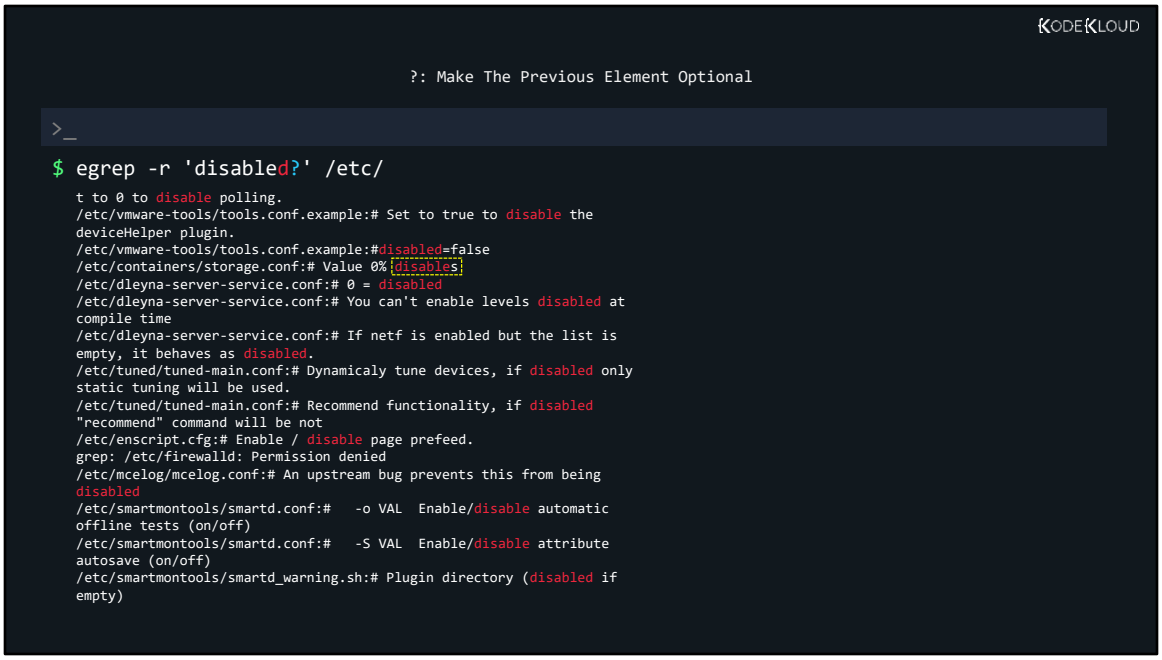
egrep -r '0{3}' /etc/



And to find all strings that contain **exactly** three zeroes:

0{3}

egrep -r '0{3}' /etc/



? will let the previous element exist precisely 0 or 1 times. This basically makes it optional: it can exist once, or not at all.

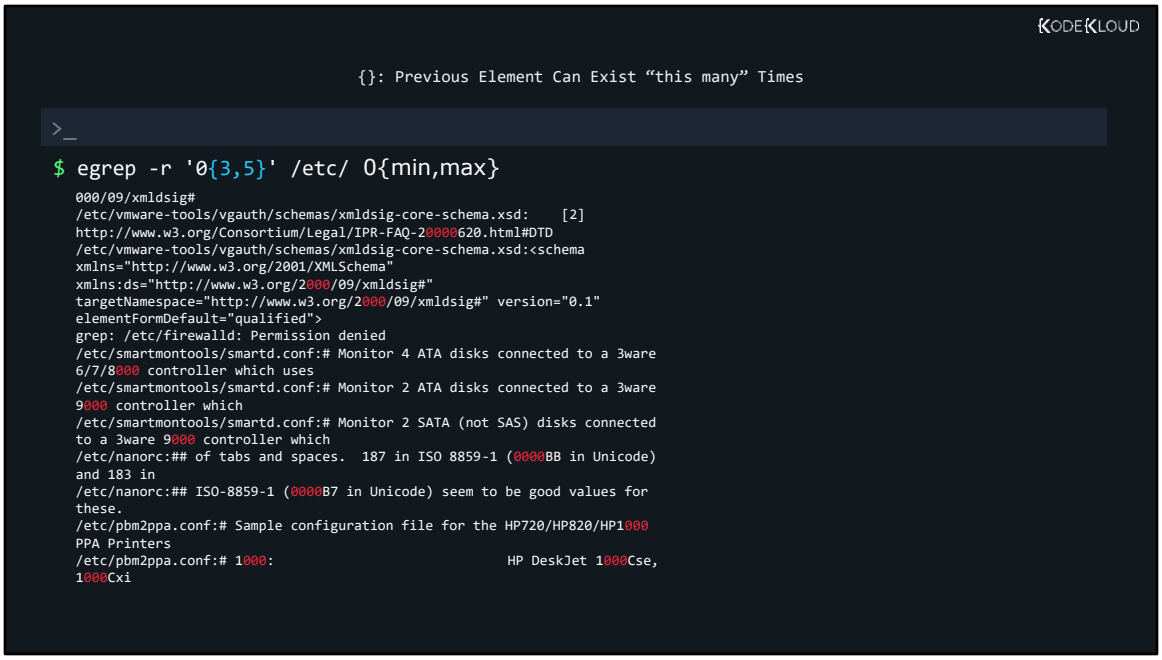
Let's say we're trying to find all text that says "disable**d**" or "disable". This means the last "d" is optional, so we can write an expression like:

disabled?

To use in grep:

```
egrep -r 'disabled?' /etc/
```

Note that this also matches the word “disables.” This is a case where the letter “d” did not come at the end, and “disable” still matches.



In an expression like

0{min,max}

to find a match, zero has to exist at least **min** times and at most **max** times.

To find all strings that contain 3, 4 or 5 zeroes:

0{3,5}

egrep -r '0{3,5}' /etc/

To find all strings that contain **at least 3** zeros:

0{3,}

egrep -r '0{3,}' /etc/

To find all strings that contain "1" followed by **at most** 3 zeroes:

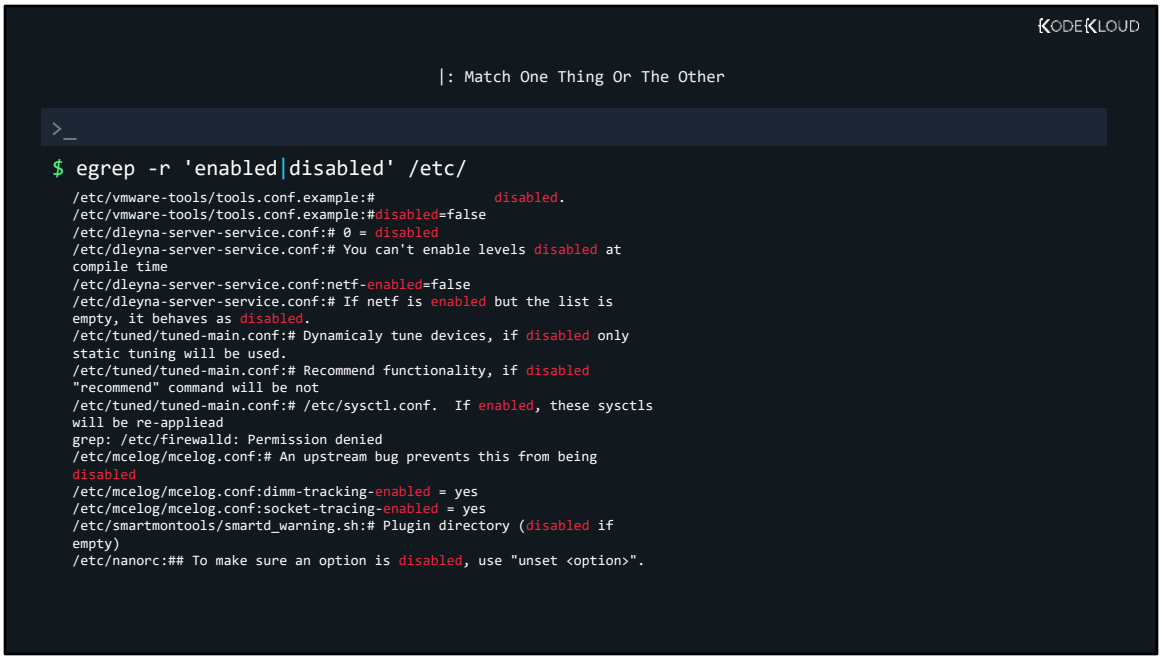
10{,3}

egrep -r '10{,3}' /etc/

And to find all strings that contain **exactly** three zeroes:

0{3}

egrep -r '0{3}' /etc/

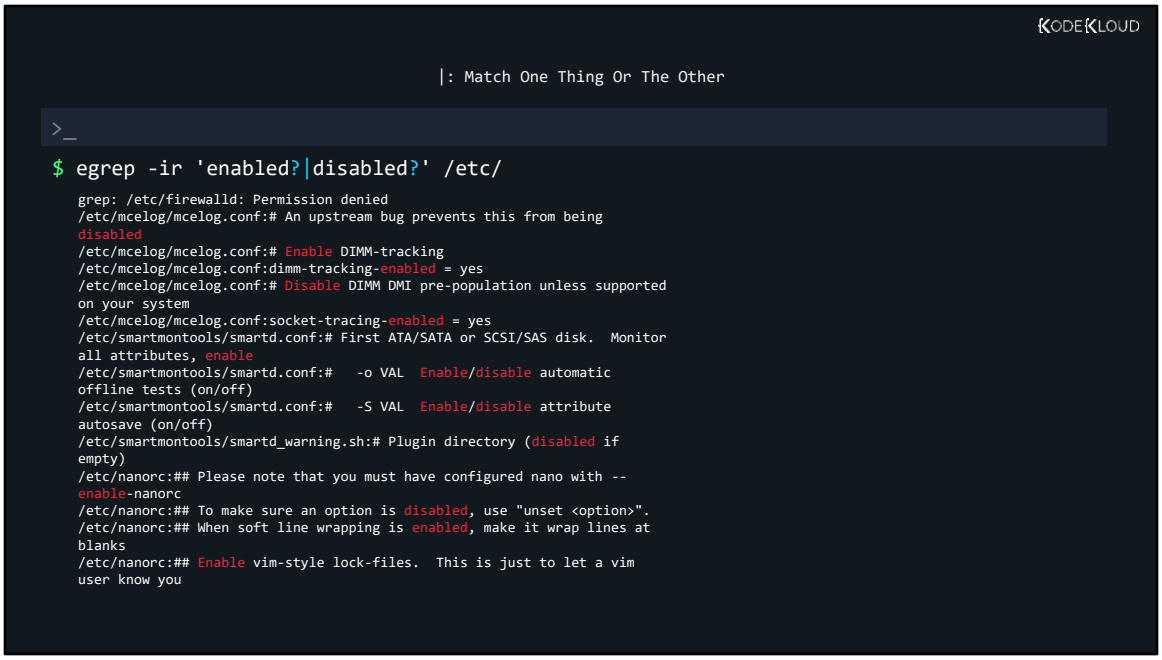


If we'd want to match "enabled" or "disabled", we could use

enabled|disabled

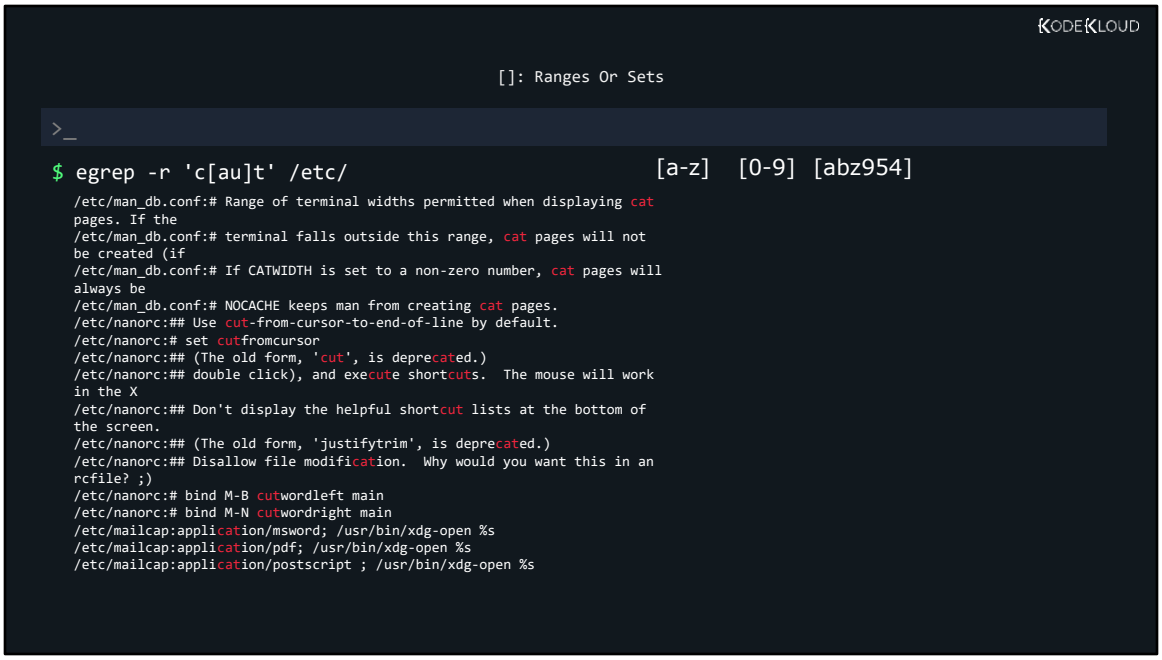
So this basically matches what it finds on its left side or its right side.

egrep -r 'enabled|disabled' /etc/



And we could combine this with our previous trick (make last "d" letter optional), to also find variations like enable/enabled, disable/disabled:

egrep -r 'enabled?|disabled?' /etc/



Now it's time to see how we can put all this knowledge to use and combine multiple regex operators to fine-tune our searches.

But first, let's learn about ranges and sets. A range is specified in the form of:

[a-z] - this will match any **one** lowercase letter, from a,b,c,d,e… to z

[0-9] - will match any **one** digit from 0,1,2… to 9

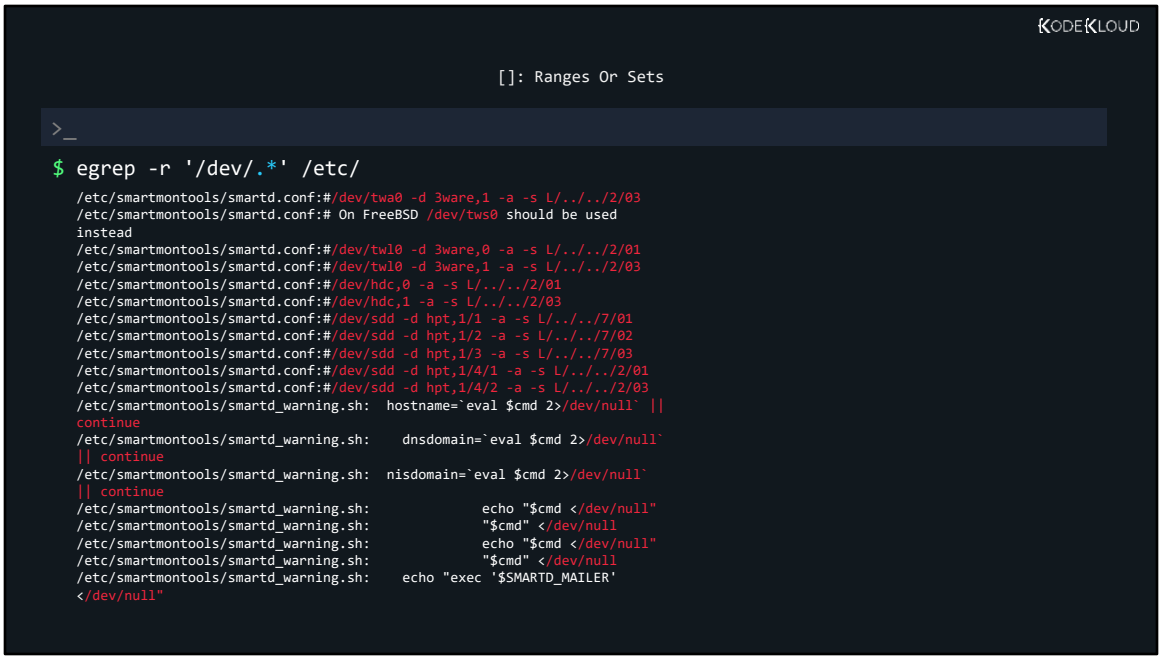
A set is specified in this form:

[abz954] will match any **one** character within, a, b, z, 9, 5 or 4

So, to find all strings that contain the text cat or cut, we'd use:

c[au]t

egrep -r 'c[au]t' /etc/'



With ranges and sets we can make our searches both wide, and specific, even at the same time. For example, let's ask ourselves: how would we find all special device files which have names like /dev/sda1 or similar? We could think like this: find all strings that contain "/dev/" followed by any random characters:

/dev/.*

egrep -r '/dev/**.***' /etc

But this matches weird stuff. **.*** is "greedy" matching way too many things after it captures what we're looking for. So, we can make our search wide enough to catch all /dev devices, but specific enough to only capture the parts we need. We do this with ranges.

We can say: "after /dev/ match any number (*) of lowercase letters, from a to z.

/dev/**[a-z]***

egrep -r '/dev/[a-z]*' /etc/

Looks a little bit better, but we see some things are still missed. /dev/twl is matched instead of the entire /dev/twl**0**. How can we catch the digits at the end too? Easy, we specify that a digit from 0 to 9 should exist there

/dev/[a-z]***[0-9]**

egrep -r '/dev/[a-z]***[0-9]**' /etc/

But now we run into another problem. Only things that have a digit at the end are matched with this new regex. We'll only find **/dev/sda1** but not **/dev/sda**. This is an easy fix, we just make the digit at the end optional with ?.

egrep -r '/dev/[a-z]*[0-9]**?**' /etc/

Looks much better now.

We can say: "after /dev/ match any number (*) of lowercase letters, from a to z.

/dev/**[a-z]***

egrep -r '/dev/[a-z]*' /etc/

Looks a little bit better, but we see some things are still missed. /dev/twa is matched instead of the

entire /dev/twa**0**. How can we catch the digits at the end too? Easy, we specify that a digit from 0 to 9 should exist there

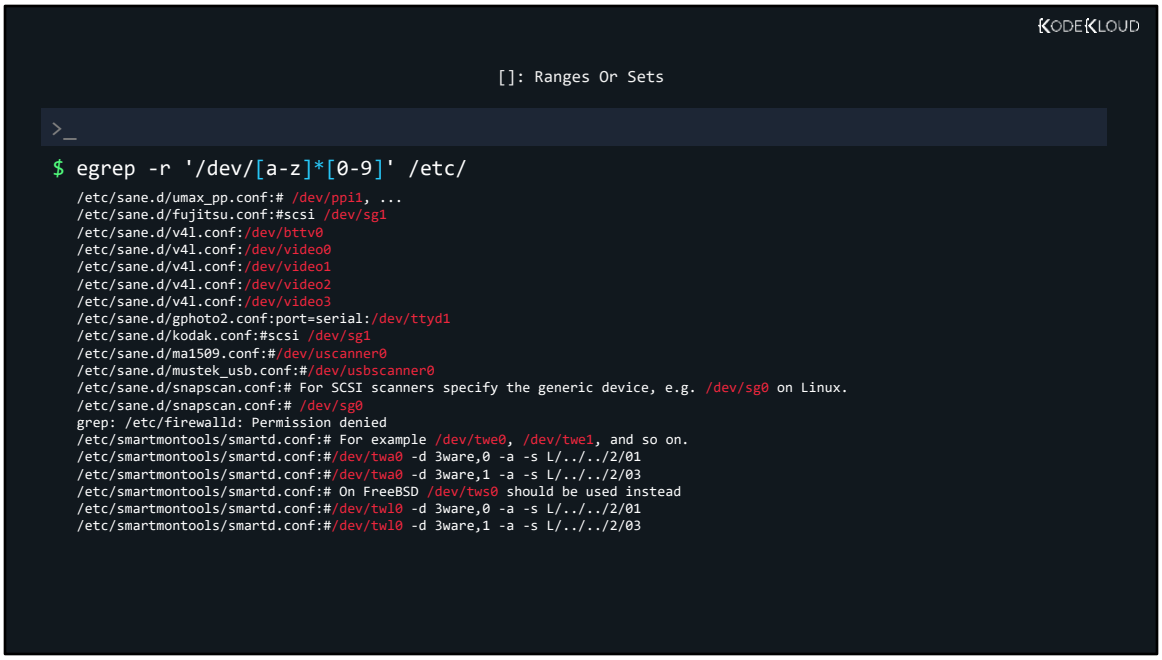
/dev/[a-z]***[0-9]**

egrep -r '/dev/[a-z]***[0-9]**' /etc/

But now we run into another problem. Only things that have a digit at the end are matched with this new regex. We'll only find **/dev/sda1** but not **/dev/sda**. This is an easy fix, we just make the digit at the end optional with ?.

egrep -r '/dev/[a-z]*[0-9]**?**' /etc/

Looks much better now.



Easy, we specify that a digit from 0 to 9 should exist there

/dev/[a-z]*[0-9]

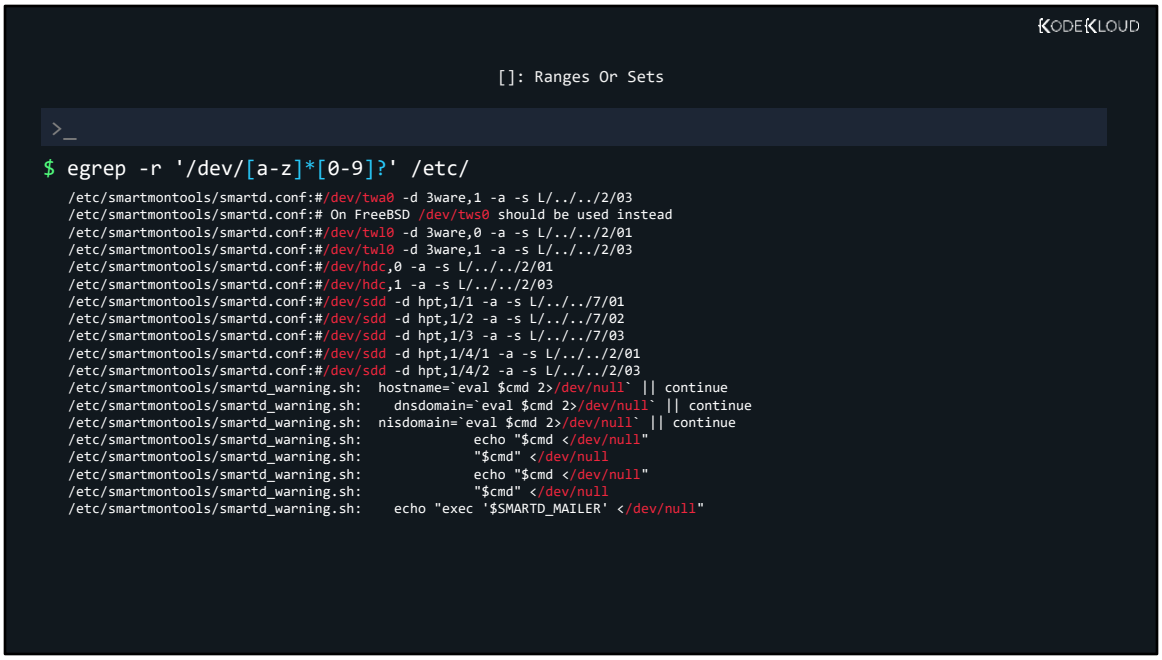
egrep -r '/dev/[a-z]*[0-9]' /etc/

But now we run into another problem. Only things that have a digit at the end are matched with this new regex. We'll only find **/dev/sda1** but not

/dev/sda. This is an easy fix, we just make the digit at the end optional with ?.

egrep -r '/dev/[a-z]*[0-9]**?**' /etc/

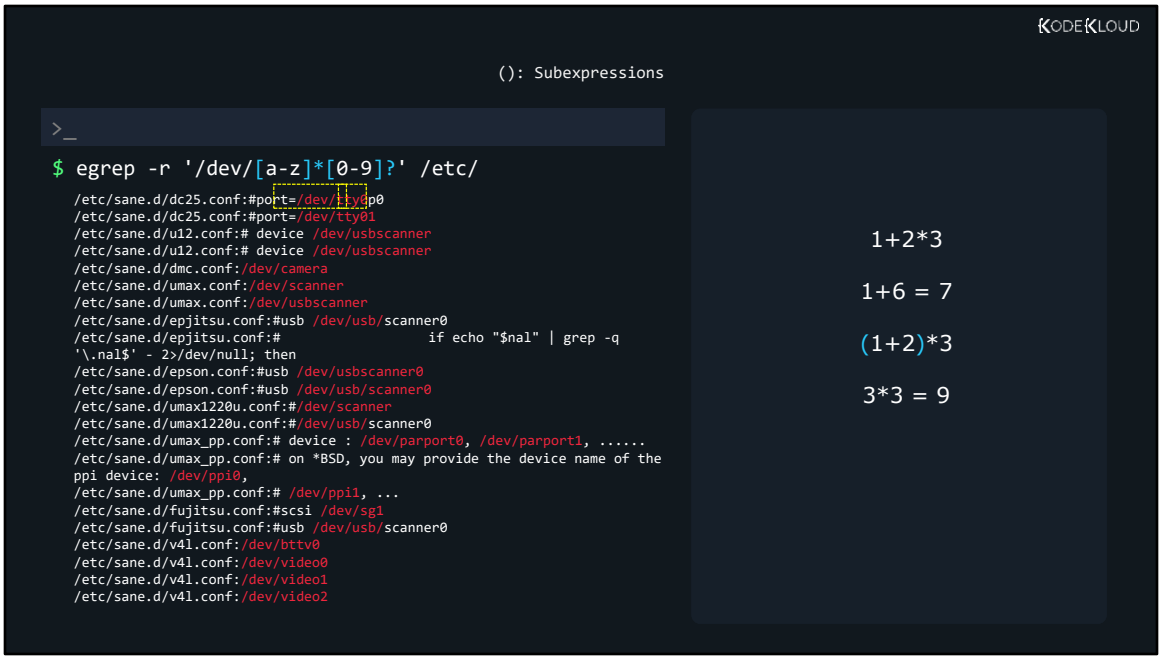
Looks much better now.



This is an easy fix, we just make the digit at the end optional with ?.

egrep -r '/dev/[a-z]*[0-9]**?**' /etc/

Looks much better now.



In math we can see this:

1+2*3

This is 1+6=7. That's because, first, multiplication will be done, and then, addition. But what if we first want to add 1+2 and then multiply by 3? We write:

(1+2)*3

This will be 3*3=9.

In regex we can do a very similar thing.

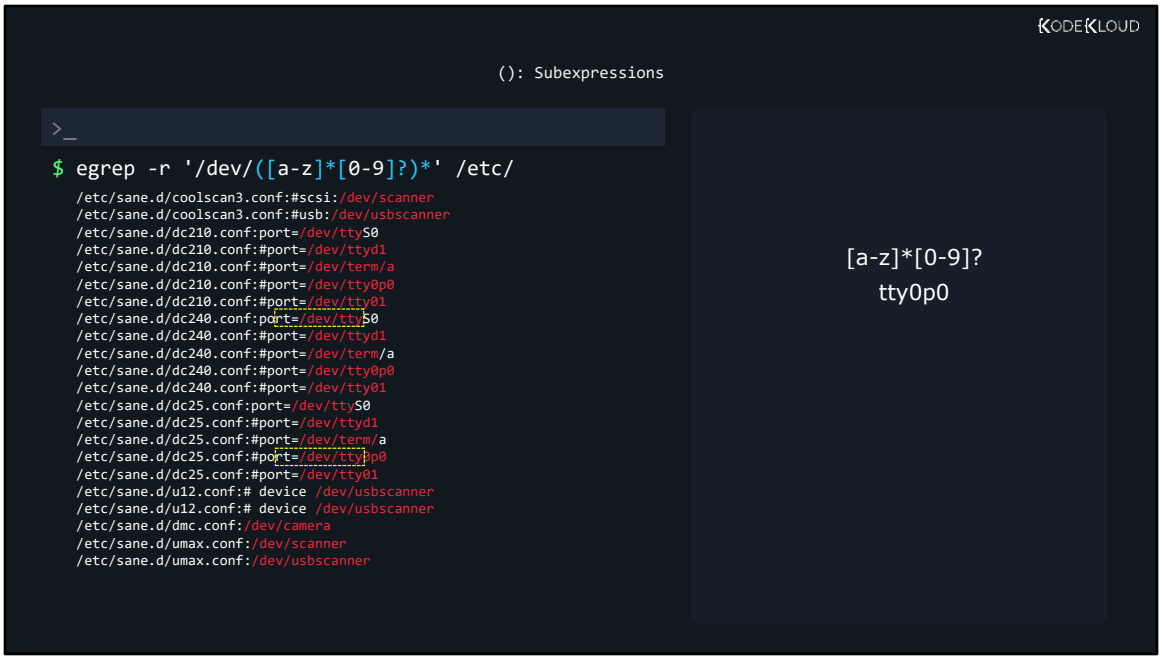
Let's take a look at our last expression:

egrep -r '/dev/[a-z]*[0-9]?' /etc/

If we scroll up, we'll see we still don't match everything we need perfectly:

/dev/tty0p0

p0 is left out. Why is that? Because our expression, after it finds /dev/ matches any number of a to z characters, then a digit at the end. And that's it, that's where the match ends. So, in /dev/tty0p0 after that first 0 is hit our regex is happy with the partial result. How could we correct this?



We could tell it that after /dev/ we have some letters, and a digit at the end, **but after that**, the same thing can **repeat** 0,1,2,3 or more times. There can be other sequences of letters followed by a digit. This way, /dev/tty0 would match first, then p0 will be added to this match by that **repetition**.

So, we would basically want to say that this part of the regex:

[a-z]*[0-9]?

should look for this pattern existing 0, 1, 2, 3 or many more times, so it can match things like **tty0**p0. What makes regex look for something to exist 0 or more times? The *. But if we add it at the end, we get

[a-z]*[0-9]?*

This isn't good, as the * would apply to the **previous element only**, and we want to apply it to **our whole construct** here. Again, easy solution. We just wrap our construct in ( ) and this way, * will apply to our **entire subexpression wrapped** in parentheses, instead of the last element only.

([a-z]*[0-9]?)*

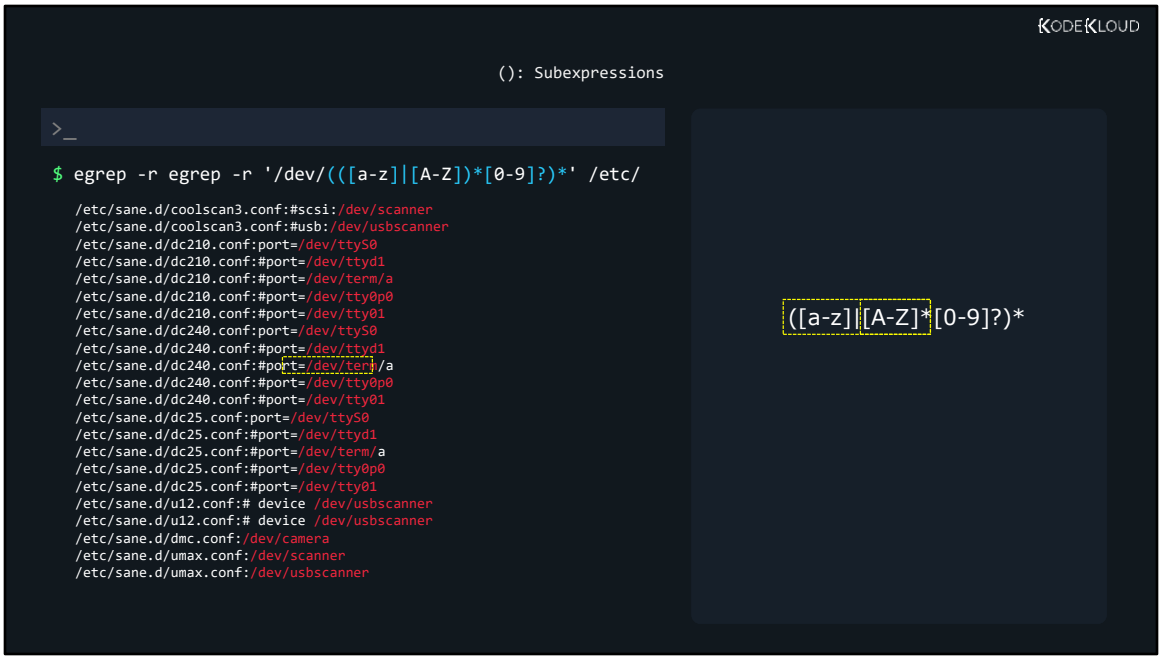
egrep -r '/dev/**([a-z]*[0-9]?)***' /etc/

And now we get a full match for strings like **/dev/tty0p0**.

And if we scroll up in our result list, we'll still find

some things that don't quite work

like /dev/ttyS0 with the S0 not matching because we didn't include uppercase letters in our regex.



So, we could tell our expression to look for "lowercase letters **OR** uppercase" with the | operator.

But writing it like this would be a mistake:

([a-z]|[A-Z]*[0-9]?)*

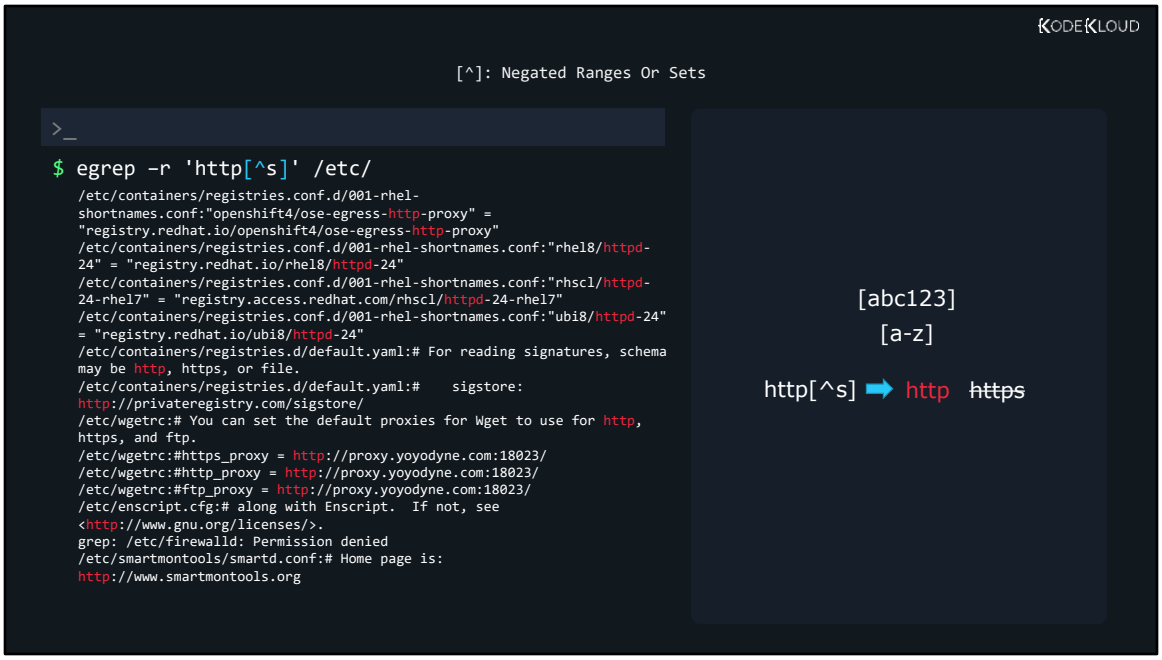
Because now the * would only apply to [A-Z] and we need to apply it to our entire [a-z]|[A-Z]. Once again,

we can wrap in parentheses to fix this.

(([a-z]|[A-Z])*[0-9]?)*

egrep -r '/dev/(([a-z]|[A-Z])*[0-9]?)*' /etc/

Now ttyS0 matches. And if we would go on, we could fix things like /dev/term/a not matching, because our regex stops when it encounters the next /, and so on. This is the kind of logic and fine-tuning we would go through when fixing our regular expressions or making them laser-focused on what we need to find.



Imagine we want to search for links to website addresses that don't use encryption. This means we would want to search for "http" strings, but exclude "http**s**".

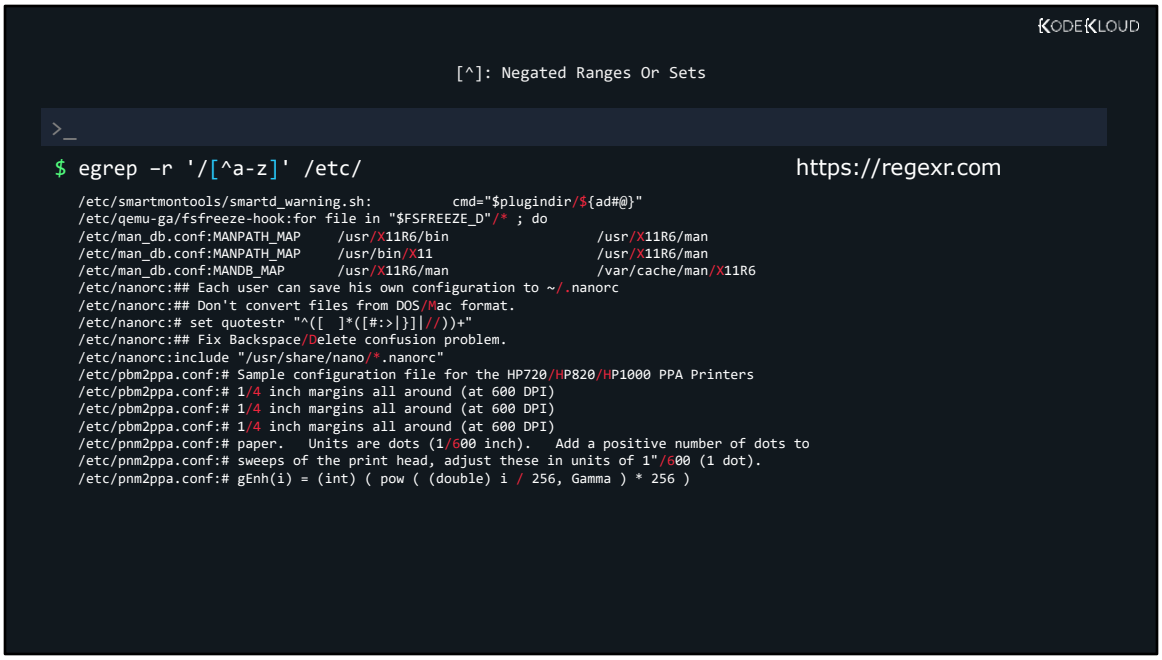
We saw sets are in the form of [abc123] and ranges [a-z]. If we add a ^ in here, we can **negate** them, tell regex "the elements in this set or range **should not exist at this position**"

So to look for http links, we could have a regex that

makes sure http is not followed by the **s** letter:

http[^s]

egrep -r 'http[^s]' /etc/



In this case, we used a set with only one character, but we can use multiple if we want.

For example, we could tell our pattern: "After a /, there should not be any lowercase letter":

egrep -r '/[^a-z]' /etc

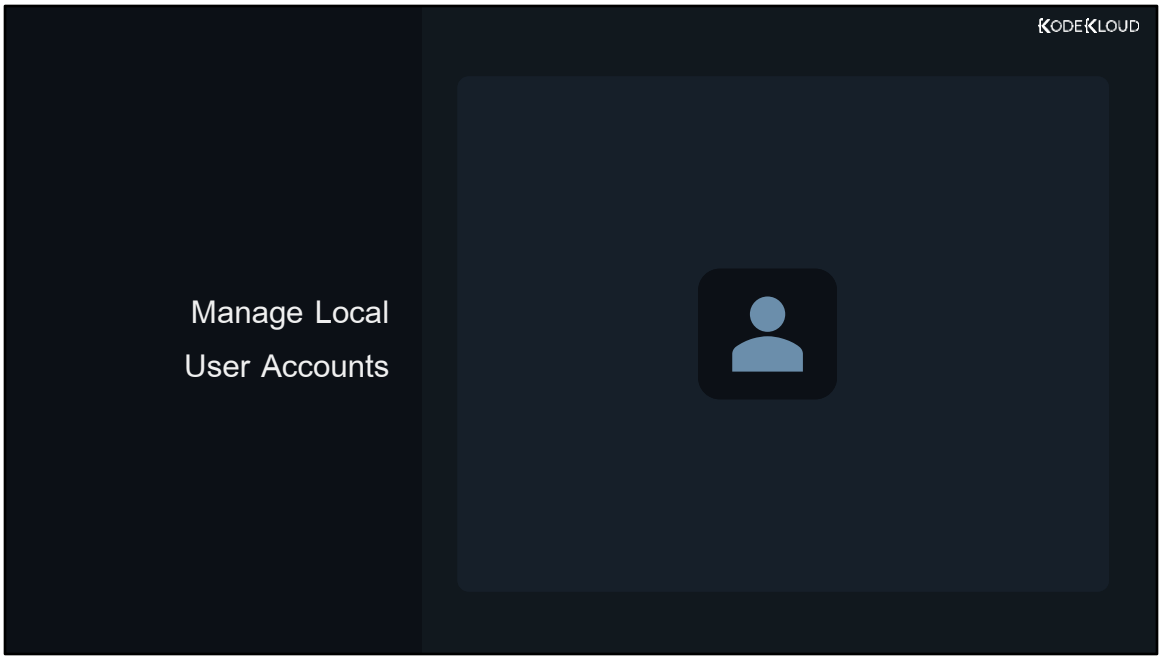
Keep in mind that for any pattern you're trying to match, there are multiple regex solutions you may find. To get this right, you should practice until you

feel comfortable with regular expressions.

It's also worth noting that regex is not limited to grep. You can use regular expressions in a lot of programs that deal with search patterns. For example, the sed utility also supports regular expressions.

Additional Resources

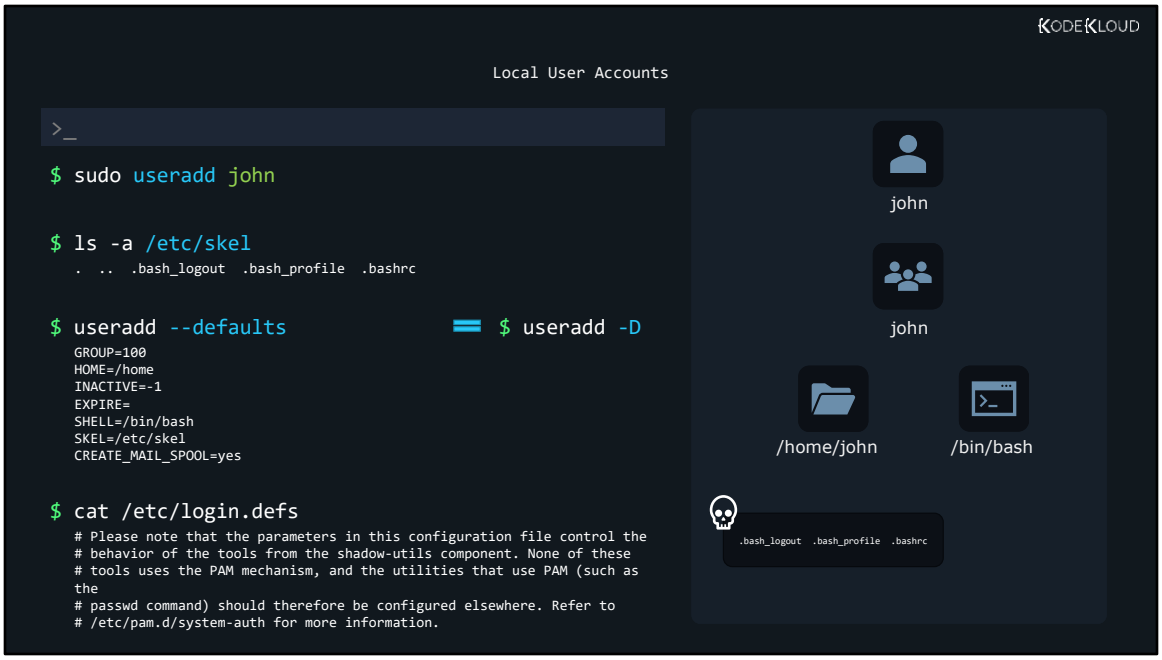
https://regexr.com/



Now let's look at how to create, delete, and modify local user accounts in Linux.

Each person that needs to log in to our Linux server should have their own, separate, user account. This allows them to have personal files and directories, protected by proper permissions. They also get to choose their own settings for whatever tools they use. And it also helps us as administrators. We can limit the privileges of each user to only what they require to do their job. This can sometimes reduce or prevent the damage when someone accidentally

writes the wrong command. And it can help with the overall security of the system.



It will be up to us to manage these user accounts, which are sometimes simply called "users". So, let's dive right in and see how we create a new user on a Linux system. The command that lets us add a new user is intuitively called **useradd**. The simplest form we can use is:

sudo useradd john

where john can be replaced with whatever username we want to choose for this specific account.

After we run this the following things happen:

•A **new user** called "john" is added to the system
•A n**ew group** also called "**john**" is automatically created. The **group** "john" will be set to be the primary group of the user "john".
•A **home directory** is created for this account at **/home/john/**. This is where John can store his personal files and subdirectories, plus his program settings.
•Their default **shell** will be set to be the program found at **/bin/bash**. Whenever John logs in, this is the application he'll be "dropped into". Effectively, his entire login session will run inside this app.
•All files from **/etc/skel** will be copied to the user's home directory **/home/john/**. You can explore it with **ls -a /etc/skel/** if you're curious to see what's inside. We'll see why this so-called "skeleton directory" is useful, in one of the next lessons.
•The account will **never expire**. We'll see what this means, later in this lesson.

All these things happen because the operating system is configured to take some default actions for each newly added account. We can explore these defaults with the following commands:

useradd --defaults

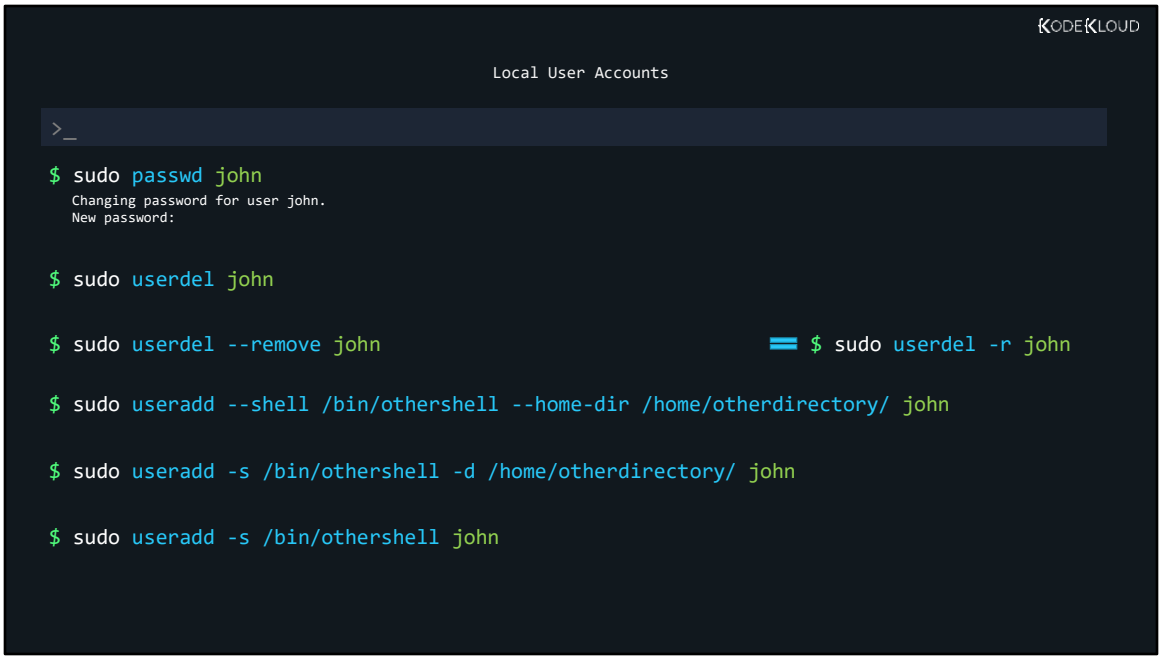
or equivalent command

useradd -D

Other defaults related to account creation can be seen by exploring this file:

cat /etc/login.defs

The comments explain what each setting does.



Ok, at this point we have an account for "john". But how does he log in? His account has no password now. To set a password for him, we can run

sudo passwd john

If later, we want to delete an account:

sudo userdel john

Note, however, that this will only delete the "john" user account. Also, the group with the same name, "john" might get auto-removed. But john's home directory at /home/john/ will remain. And that's

normal, because his personal files might still be needed. But if we're certain that those files aren't necessary anymore, we can make the userdel command also remove the user's home directory and his/her mail spool with:

sudo userdel --remove john

or equivalent

sudo userdel -r john

Coming back to the useradd command, if we're not happy with the defaults, we could choose a different shell and home directory with a command such as:

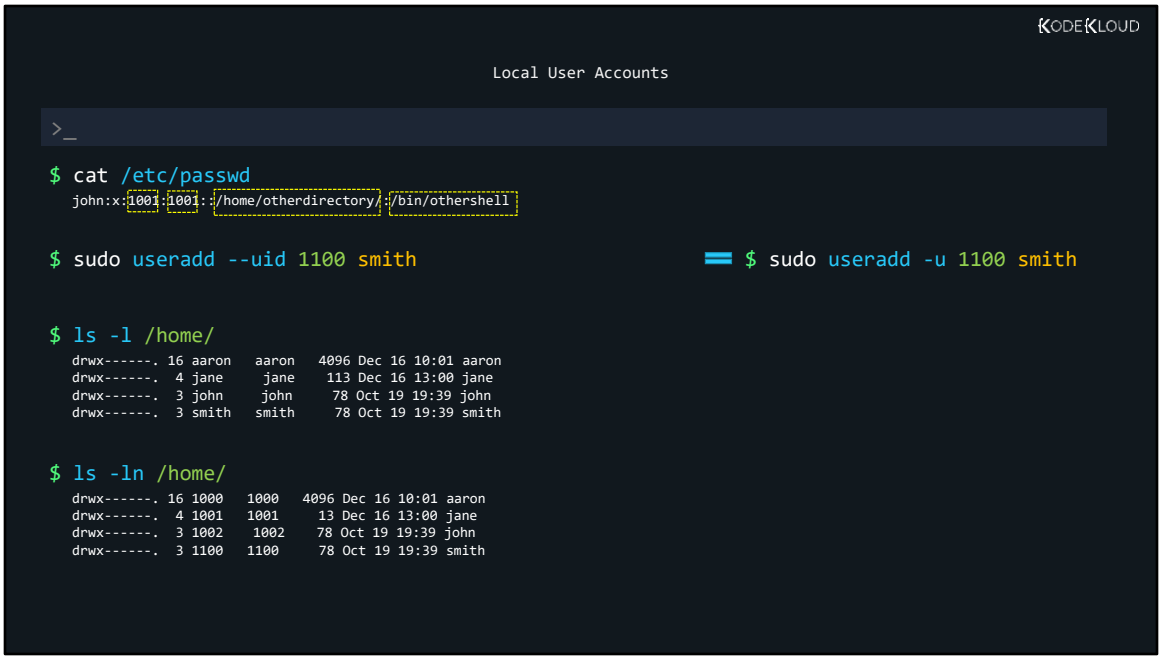
sudo useradd --shell /bin/othershell --home-dir /home/otherdirectory/ john

or equivalent

sudo useradd -s /bin/othershell -d /home/otherdirectory/ john

Of course, if we only want to choose a different shell, but keep the default location for the home directory, we can just pass the shell option:

sudo useradd -s /bin/othershell john



These account details, such as usernames, user IDs, group IDs, preferred shells, home directories are stored in the file at **/etc/passwd**. We can see them if we type:

cat /etc/passwd

We'll see a line like this:

john:x:1001:1001::/home/otherdirectory/:/bin/othersh ell

The first number, 1001 is the ID number associated with john's username. The next 1001 is the numeric

ID of its primary group, also called "john" in this case. Then we can see the home directory and the preferred login shell.

useradd will automatically select a proper numeric ID available, incrementally. For the first user, the ID will be 1000, for the next one 1001, and so on. If we want to manually select a different ID, we can use a command such as:

sudo useradd --uid 1100 smith

or equivalent

sudo useradd -u 1100 smith

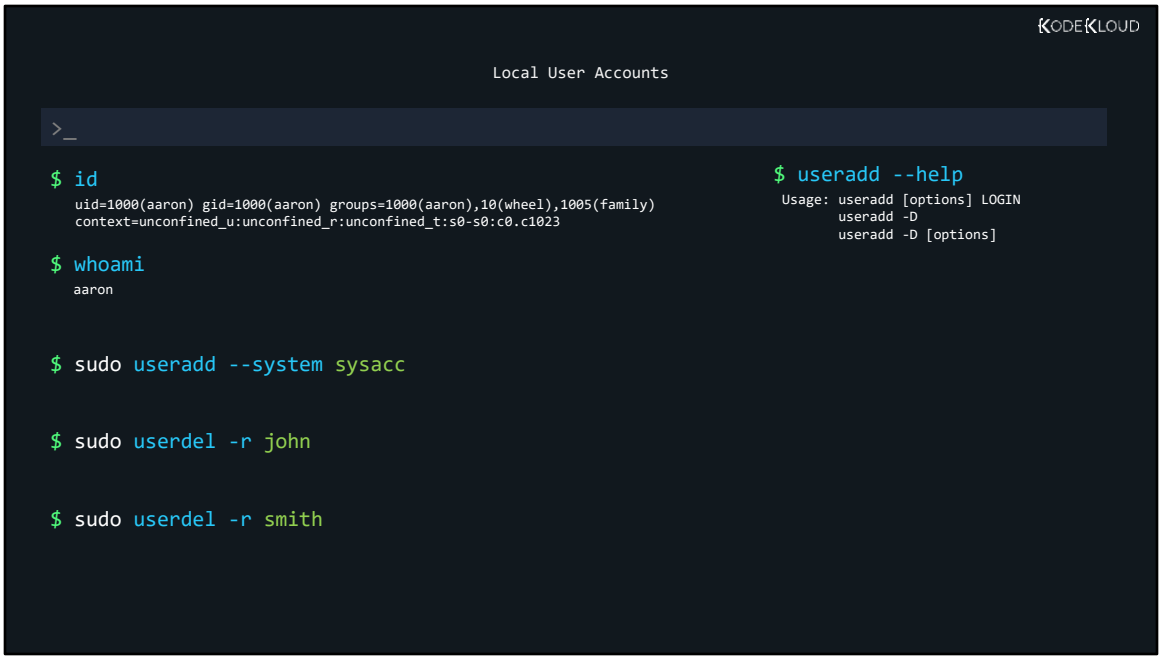
The user "smith" will have the numeric ID 1100, but also the group called "smith" will get a numeric ID of 1100.

If we want to see what username and group owns files or directories, we can do so with the usual

ls -l /home/

But if we want to see the numeric IDs of the user and group owners, we can add the -n (numeric ID) option:

ls -ln /home/



It might also be useful sometimes to find out more about the user we're currently logged in as. We can see the username we're logged in as, plus groups we're members of, alongside with the respective IDs, with this command:

id

To just print out the username:

whoami

Up until now, we've created user accounts. But there's another type we can create, called system

accounts. To create a system account called **sysacc**, we just add the --system option:

sudo useradd --system sysacc

The numeric IDs of system accounts are usually numbers smaller than 1000. So, we might see an ID like 976 or 978 for our sysacc account.

Why would we create these? User accounts are intended for people. System accounts are intended for programs. So, there will be no home directory created since it's not needed. Usually, daemons use system accounts. We might see something like a database program running under a system account.

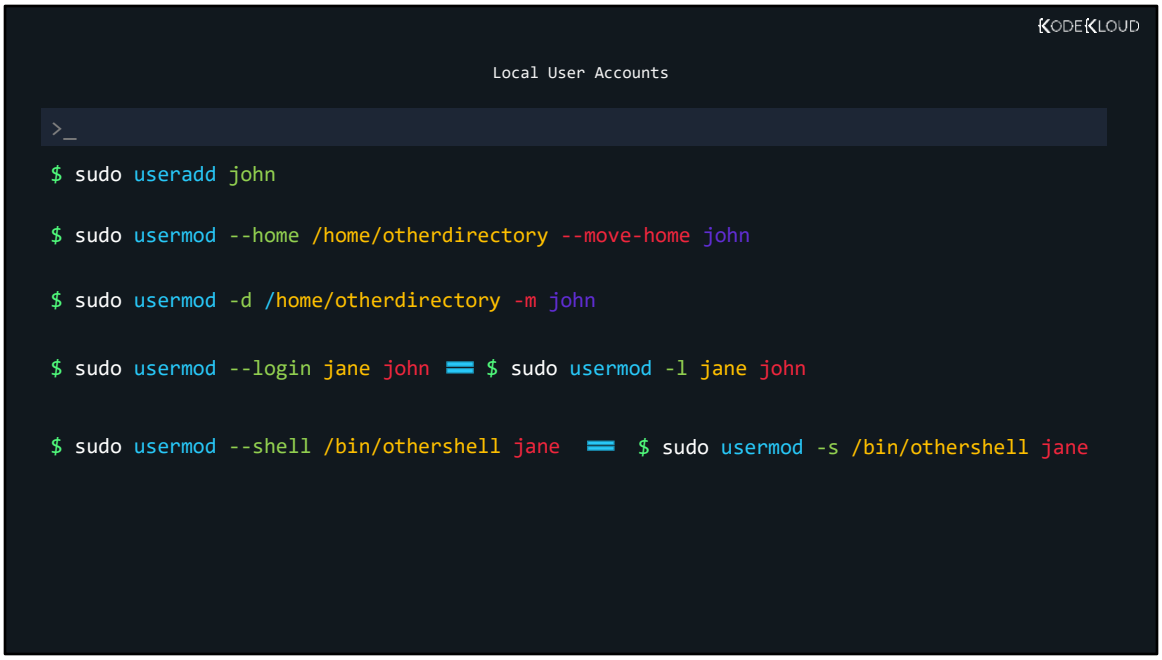
Now let's remove these users and their personal files:

sudo userdel -r john

sudo userdel -r smith

If we ever forget the options for the useradd command, we can get a quick reminder with:

useradd --help



Now let's say we create the user "john" again:

sudo useradd john

But later, we decide that we want to change some details for this account. The command **usermod** (user modify) is used for this purpose.

For example, if we want to change john's home directory, we can use:

sudo usermod --home /home/otherdirectory/ --move-home john

or equivalent

sudo usermod -d /home/otherdirectory/ -m john

The --move-home option ensures that the old directory will be moved or renamed so that John can still access his old files. In our case, /home/john/ was renamed to /home/otherdirectory/.

To change the username, from **john** to **jane** we can enter:

sudo usermod --login jane john

or equivalent

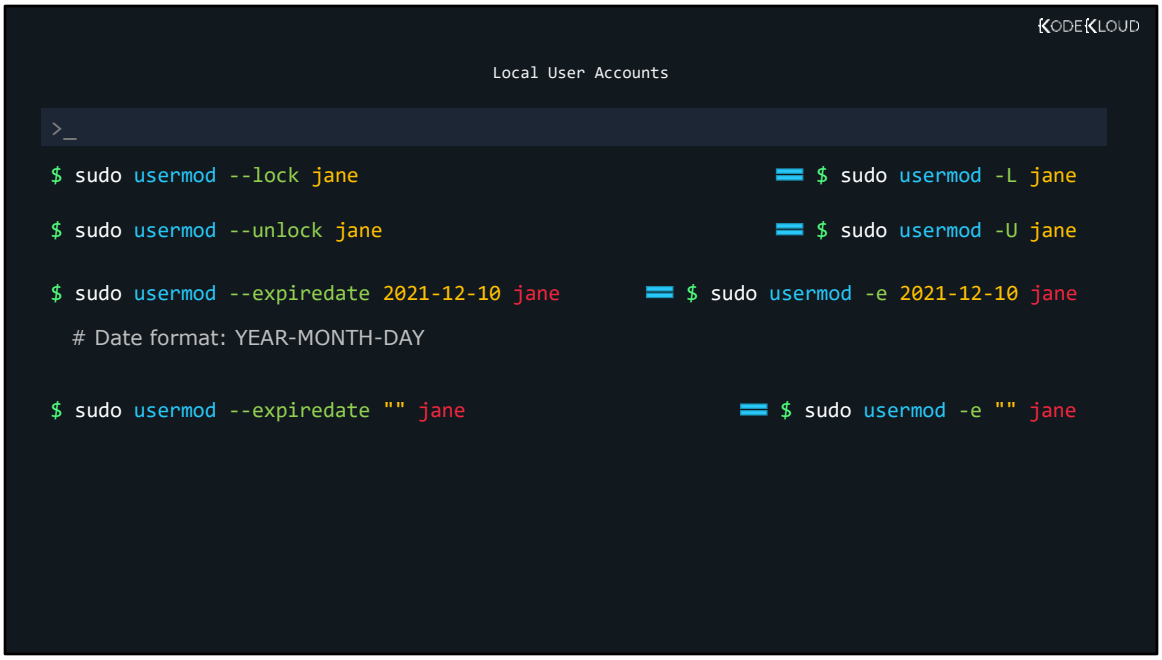
sudo usermod -l jane john

To change the user's login shell:

sudo usermod --shell /bin/othershell jane

or equivalent:

sudo usermod -s /bin/othershell jane



An often-used option with usermod is --lock (or equivalent option -L). This effectively disables the account, but without deleting it. The user will not be able to log in with his/her password anymore. However, they might still be able to log in with an SSH key, if such a login method has been previously set up.

sudo usermod --lock jane

sudo usermod -L jane

To cancel this and unlock the account:

sudo usermod --unlock jane

or equivalent

sudo usermod -U jane

To set a date at which a user's account expires, we

can use

sudo usermod --expiredate 2021-12-10 jane

or equivalent

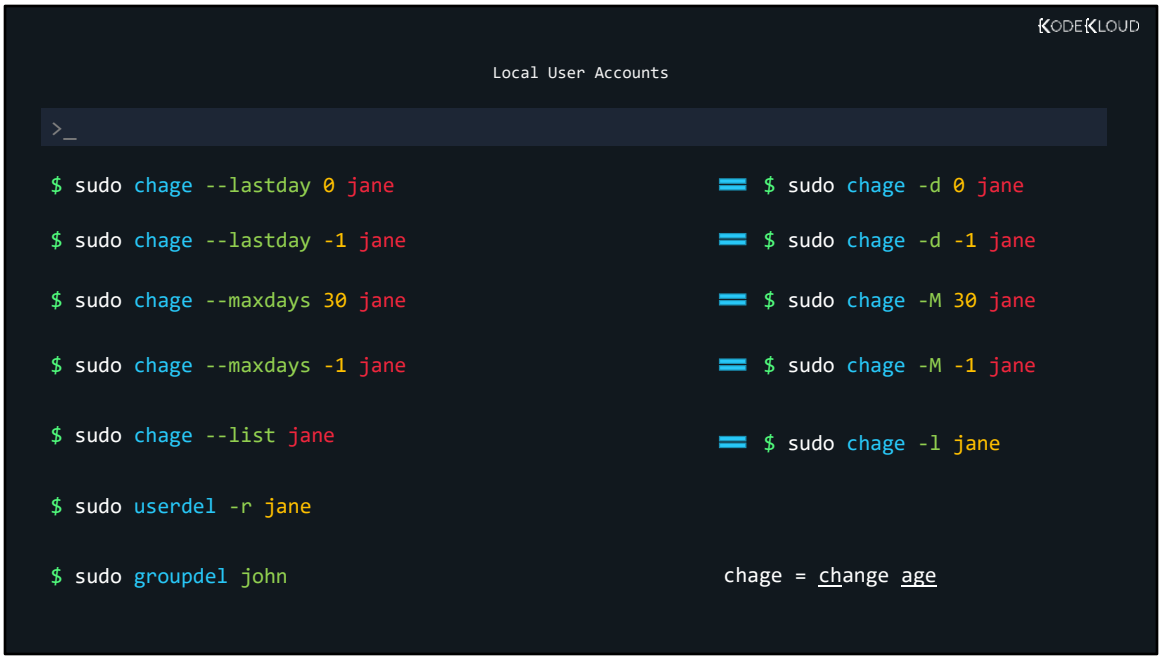
sudo usermod -e 2021-12-10 jane

After expiration, they won't be able to log in and need to contact a system administrator to re-enable their account. If we want to immediately set an account as expired, we can just choose a date that is in the past.

This date is in the format YEAR-MONTH-DAY.

To remove the expiration date, just specify an empty date. Use two quotes " with nothing inside.

sudo usermod --expiredate "" jane
sudo usermod -e "" jane



We can also set an expiration date on the password. **Please keep in mind that this is not the same as account expiration.** Account expiration completely disables user logins. Password expiration forces the user to change their password next time they log in. They can still use the account.

If we want to immediately set password as expired, we can enter this command:

sudo chage --lastday 0 jane

or equivalent

sudo chage -d 0 jane

"chage" stands for "change age"

Next time Jane logs in, she'll have to change her password.

If we want to cancel this, unexpire the password:

sudo chage --lastday -1 jane
sudo chage -d -1 jane

If we want to make sure that a user changes their password once every 30 days, we can use this command:

sudo chage --maxdays 30 jane
sudo chage -M 30 jane

If we want to make sure their password never expires, we set maxdays to -1:

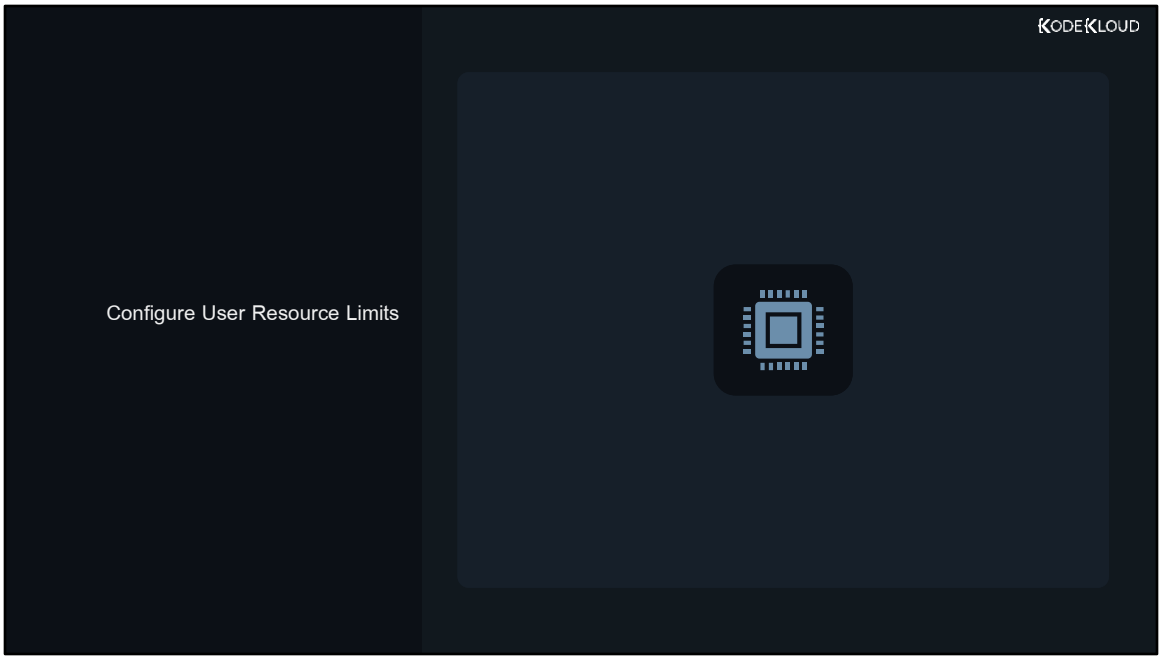
sudo chage --maxdays -1 jane
sudo chage -M -1 jane

To see when the account password expires:

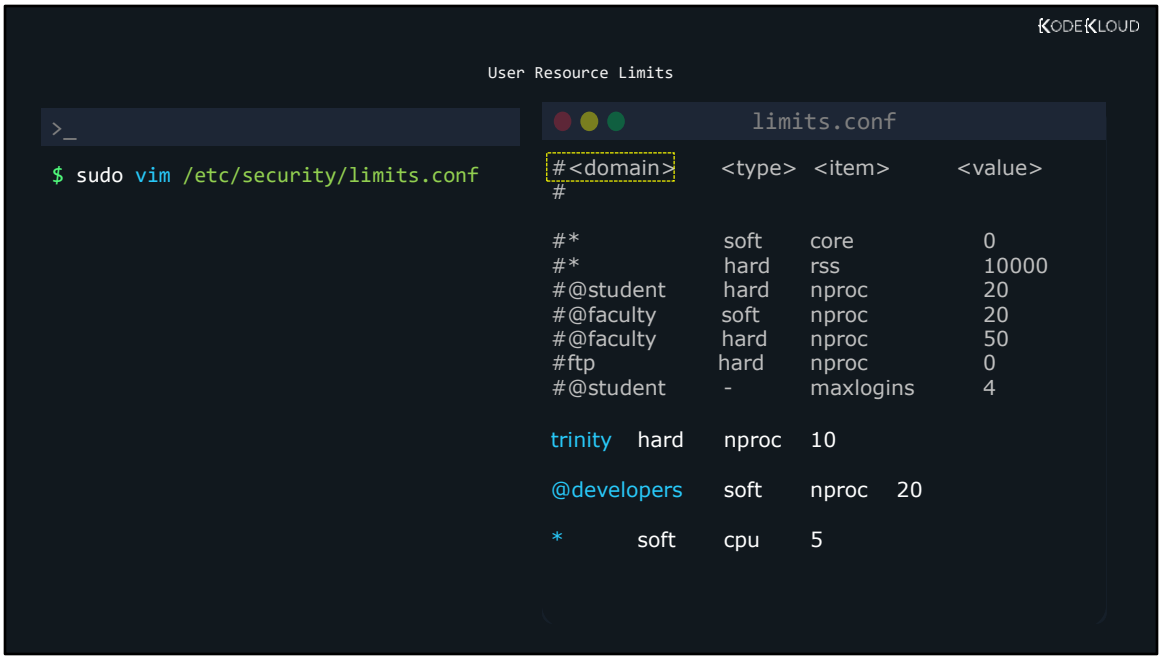
sudo chage --list jane
sudo chage -l jane

In case you followed along with this exercise, delete the user called "jane" and the group called "john".

sudo userdel -r jane
sudo groupdel john



Now, let's look at managing user resource limits in Linux.



When we have a lot of users logging in to the system, we may want to impose limits on what resources they can use. This way, we can ensure that user A does not use 80% of the CPU leaving very little to spare for the others.

To set such a limit, we can edit this file:

sudo vim **/etc/security/limits.conf**

We can see this is well-documented.

Let's move down until we see this:

We can see that the syntax for setting a limit is

domain type item value

Let's break this down into easy-to-understand parts.

First, the **domain**; what can we specify here? Usually, one of these three things:

1. Username. In this case, we just simply type the name of the user, such as **trinity**.

Example limit for the **trinity** user:

trinity hard nproc 0

2. Group name. To set a limit for everyone in the **developers** group, we just add @ in front of its name. So we'd write **@developers** to set such a group limit.

Example limit for the **developers** group:

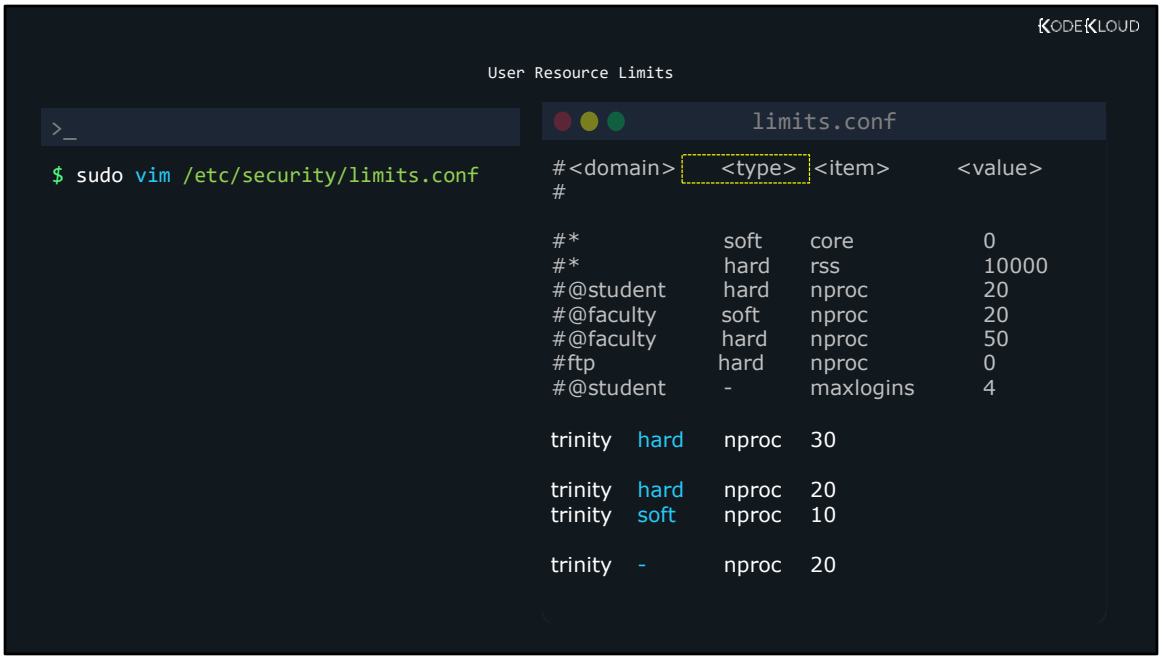
@developers soft nproc 20

3. * will match all. Setting a limit for * basically says "set this limit for every user on the system". So it's a way to set a default limit. Why default? Because this limit will only apply to every user that is not mentioned in this list. A user limit overrides a * limit.

For example, one * limit can specify that everyone can only launch 10 processes. But then another limit, for the user **trinity**, says she can launch 20 processes. In this case, the limit for everyone will be 10 (default), but for trinity, it will be set at 20.

Example default limit set with *:

```
*               soft    cpu             5
```



Next is **type** which can take three different values:

1.hard
2.soft
3.-

A **hard** limit cannot be overridden by a regular user. If a hard limit says they can only run 30 processes, they cannot go above that. It's basically, the top, the max value of a resource someone can use.

trinity **hard** nproc 30

A **soft** limit on the other hand is different. Instead of

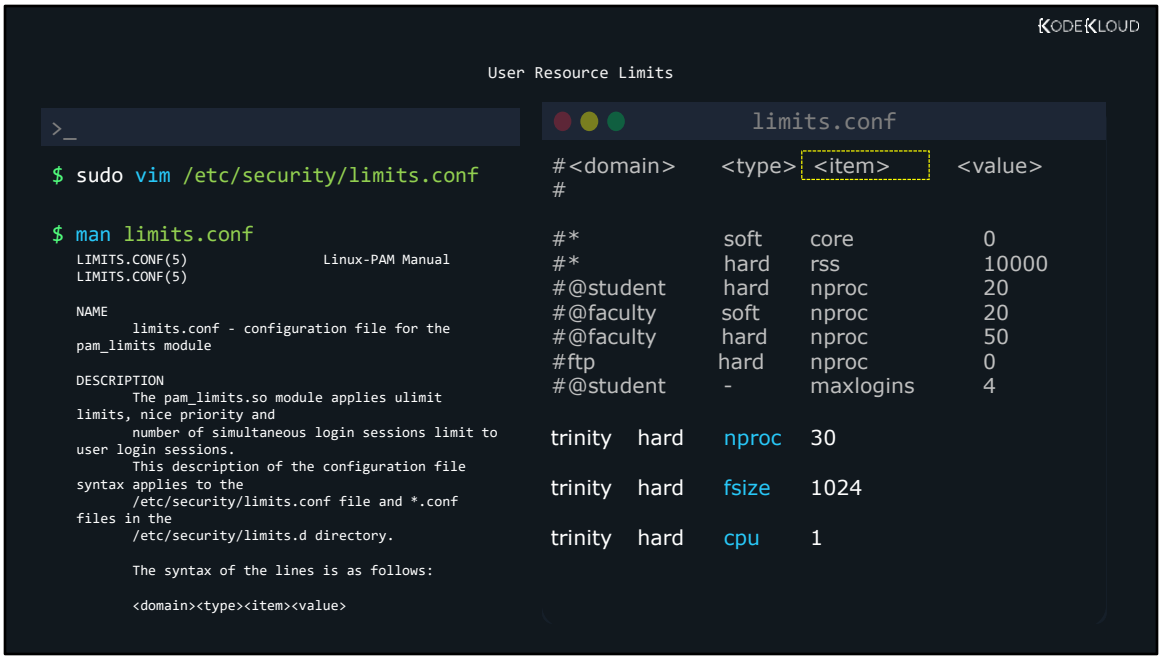
a max value, this is more like the "startup limit", the initial value for the limit when the user logs in. If a user has a soft limit of 10 max processes and a hard limit of 20, the following happens. When they log in, the limit will be set to 10 processes. But if the user has some temporary need to increase this, they can raise it to 11, 12, 15 or 20 processes. This way they can get a slight increase when absolutely required. So, they can manually raise it to anything they require, but never above the hard limit.

```
trinity      hard      nproc      20
trinity      soft      nproc      10
```

Last, we have the **-** sign. This specifies that this is both a hard and a soft limit.

```
trinity      -      nproc      20
```

With this we're saying "Trinity should be able to run 20 processes at most. When she logs in, she should be able to use up her entire allocation, without needing to manually raise her limit."



Next up, the **item** value. This decides what this limit is for. We can have things such as:

trinity hard **nproc** 20

nproc sets the maximum number of processes that can be open in a user session.

trinity hard **fsize** 1024

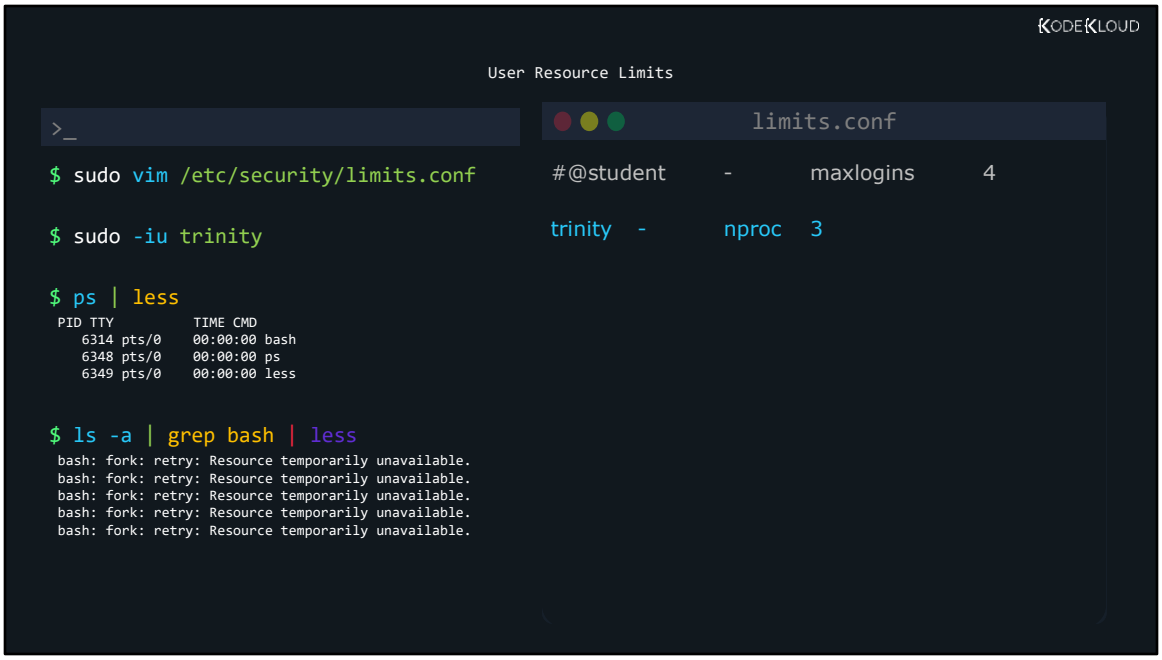
fsize sets the maximum filesize that can be created in this user session. The size is in KB so 1024 here means that the maximum file size is 1024KB which is exactly one Megabyte.

trinity hard **cpu** 1

cpu sets the limit for the CPU time. This is specified in minutes. When a process uses 100% of a cpu core for 1 second, it will use up 1 second of its allocated time. If it uses 50% of one core for one second, it will use up 0.5 seconds of its allocation. Even if a process was open 3 hours ago, it might have only used 2 seconds of CPU time.

If you want to see more stuff that can be limited just consult the user manual for this limits.conf file:

man limits.conf



Now let's test our knowledge and add a limit for our user called trinity, to ensure she can open a maximum number of three processes

Under this line

#@student - maxlogins 4

Add this:

trinity - nproc 3

Make sure there's no **#** at the beginning of this line. The vim editor might automatically add it when you press ENTER to add a new line here. Make sure to delete the preceding # otherwise the line would be commented and have no effect. Now, let's save our file and exit.

To log in as trinity, we can enter this command:

sudo -iu trinity

-i instructs sudo to do a real log in
-u specifies the user we want to log in as

At this moment, only one process is permanently running in her session, the Bash shell. So, we should be able to run two more processes. Let's launch ps and pipe the output to the less pager.

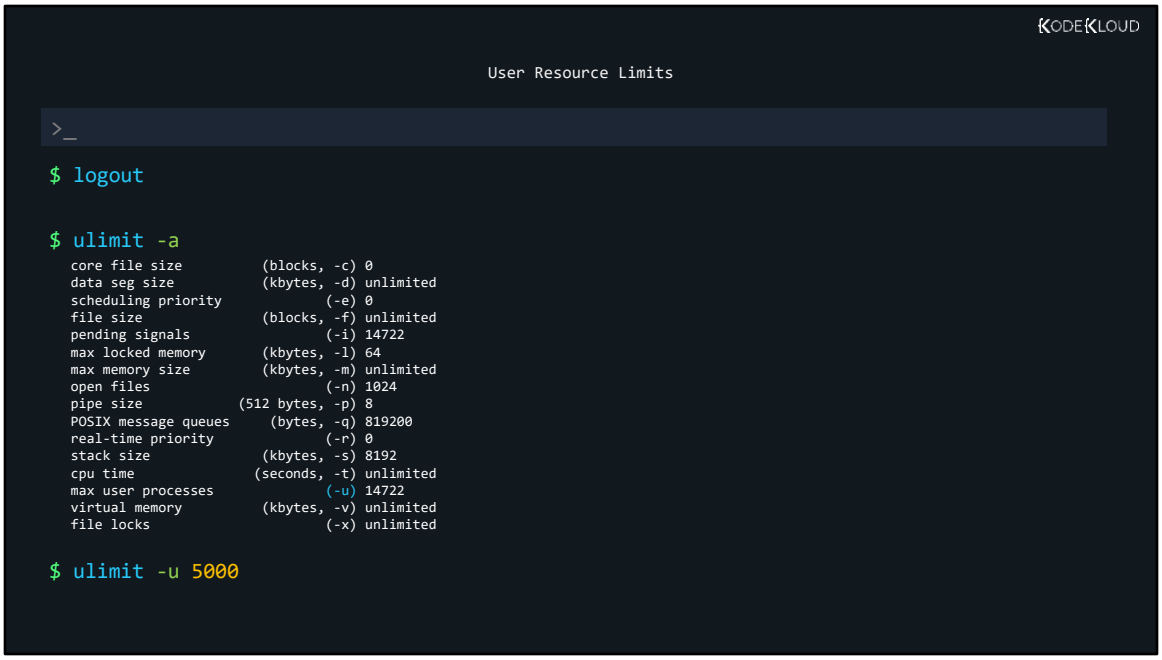
ps | less

We can see it works and it got us to running three processes, the max limit. Now what would happen if we'd try to launch the fourth? Let's press q to quit the less pager and then try the following:

ls -a | grep bash | less

This would try to launch three new processes, ls, grep and less, plus Bash already running, would total 4 processes:

And we'll see this failing, as expected. We cannot run more than three processes:



Let's type

logout

to exit from trinity's session.

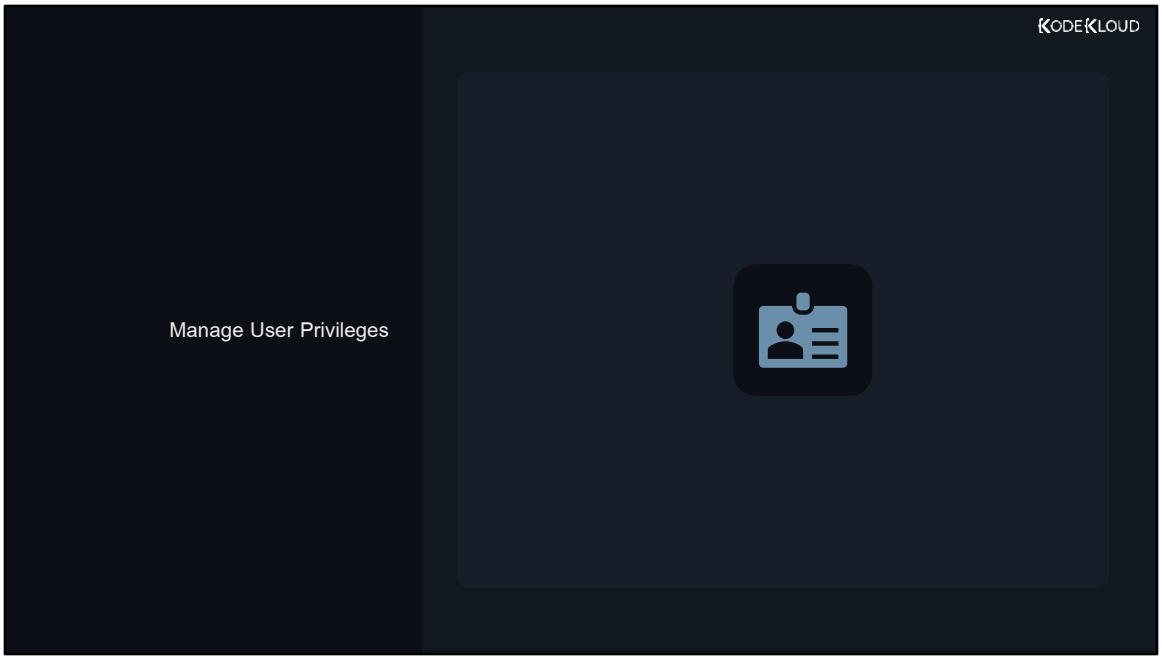
If we want to see the limits for our current session, we can type:

ulimit -a

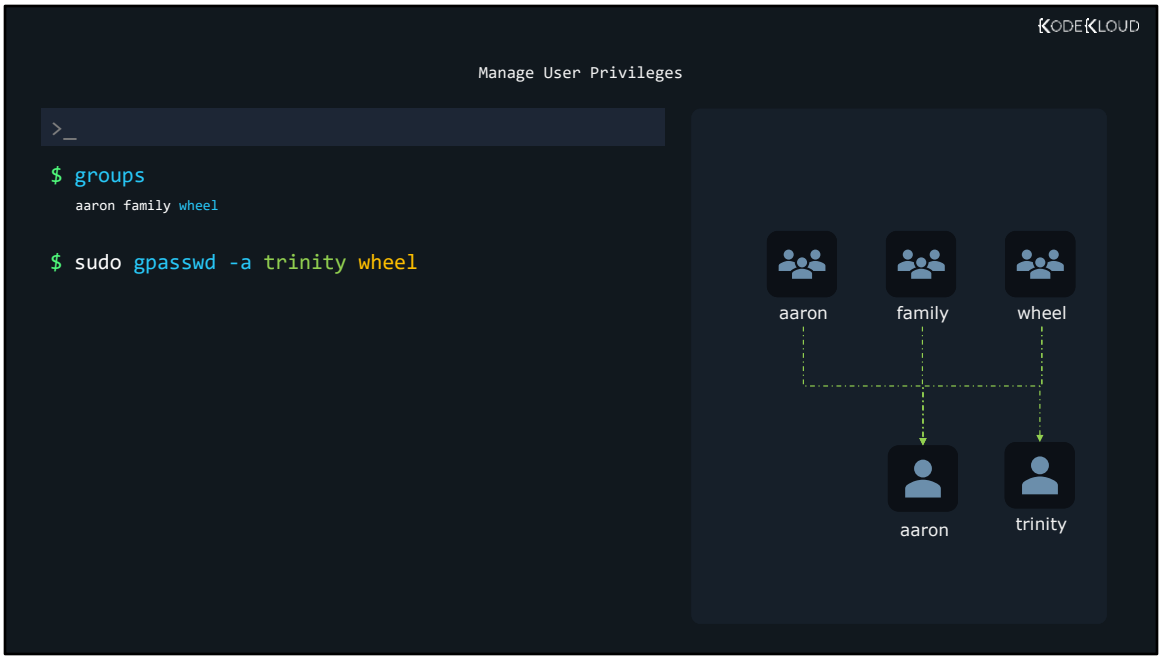
We have small hints between parentheses. For example, we can see "-u" displayed for **max user processes**. This means that we could type

ulimit -u 5000

to lower our limit to 5000 processes. By default, a user can only lower his limits, not raise them. The exception is when there are hard and soft limits. In that case, the user can raise his/her limit all the way up to the hard value, but only once. After the limit is raised with a ulimit command, the next command can only lower it. It cannot be raised the second time, even if the hard limit would allow it.



Now, let's examine how to manage user privileges in Linux.



Every time we had to make some important changes to the system, we used "sudo" in our commands. That's because only the root user, also called "superuser" can make changes to important areas of the operating system. Whenever we put "sudo" in front of a command, that command runs as if the root user executed it. So how come our user is allowed to use sudo?

If we type this command

groups

we'll see our user is part of the "wheel" group.

Whoever is part of this group is automatically allowed to use sudo.

This means that the easiest way to give another user sudo privileges is to add them to the wheel group. To add our user "trinity" to the "wheel" group:

sudo gpasswd -a trinity wheel

And that's it. Now this user can get administrator privileges whenever they want. But this gives them power to do anything they want on our system. What if we want more fine-tuned control? Then we could take a different approach.

There is a special file at **/etc/sudoers** that defines who can use sudo and under what conditions, what commands they can run, and so on. But we should not edit this file directly. We use a utility called **visudo**. This utility can check if our edits are correct to help us avoid mistakes in this file.

First, let's remove trinity from the wheel group, to make sure she can't use sudo anymore, and instead, define a different sudo policy for her, later.

sudo gpasswd -d trinity wheel

To start editing the **/etc/sudoers** file we run:

sudo visudo

This opens in the vim editor. The file is thoroughly commented, but we're not interested in the first few parts. So, let's navigate to the end. We'll notice this line

Allows people in group wheel to run all commands
%wheel ALL=(ALL)   ALL

Now we see why any user added to the "wheel" group can run any command with sudo.

Let's break down this line into 4 different parts and analyze what they do:

1.%wheel 2.ALL=3.(ALL)   4.ALL

1. is the **user/group**. Here we define who this policy is for.

2. is the **host**. Here we could specify that these rules only apply if our **server's** hostname or IP address has a specific value. Not useful for our purposes, so we'll just type ALL for this host field.

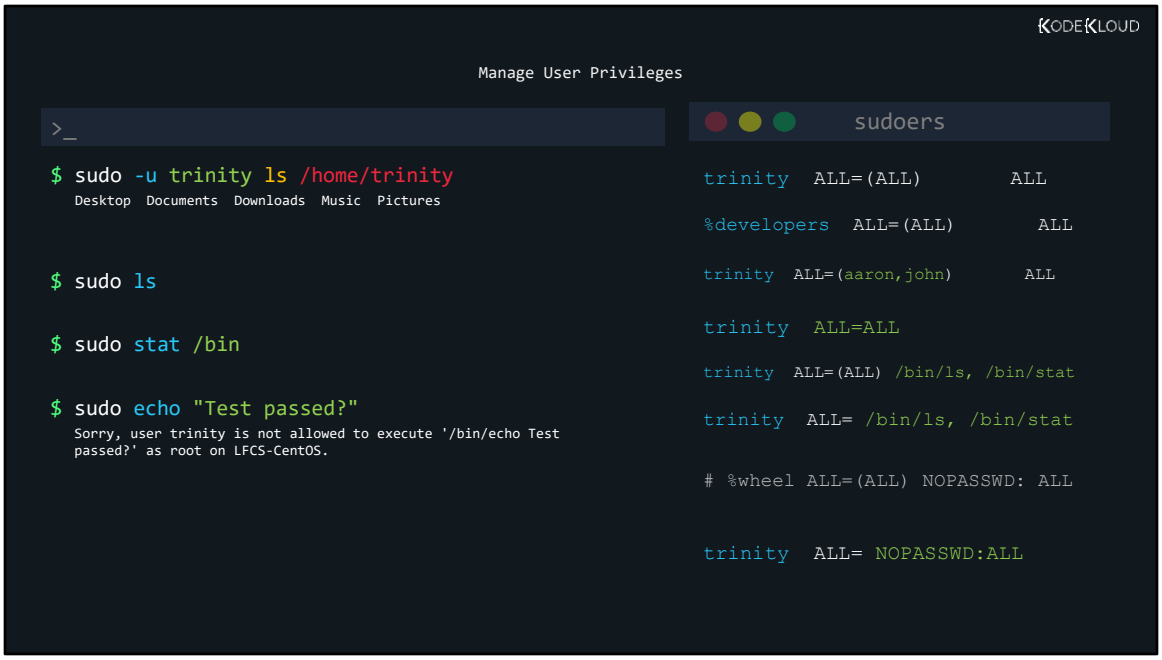
3. is the **run_as** field. Here, we could type a list of

usernames. Normally, "sudo ls" will run the "ls" command as root. Because that's what sudo does, it runs the command after it **as a different user**. But sudo can also be used so that "aaron" can run commands as "jane" or vice versa. We'll see more about this later. So, if we list "aaron, jane" in this "run_as" field, then sudo can only be used to run commands as the user "aaron" or "jane", but not "root".

4. is the list of commands that can be executed with sudo.

So we could say the syntax for a policy defined in the sudoers file is:

user/group host=(run_as_user) command_list



Now let's go through some examples. To define a policy for our trinity user and let her run any sudo command:

trinity ALL=(ALL) ALL

To specify a policy for all users in the **developers** group:

%developers ALL=(ALL) ALL

We mentioned sudo lets us run commands as root, but also as non-root, regular users. For example, to run the **ls /home/trinity/** command as the user

called trinity we could write:

sudo -u trinity ls /home/trinity/

After -u we specify the username we want to run as.

If this third field is (ALL) then this policy allows someone to run sudo commands as any user. But if we'd want **trinity** to only be able to run sudo commands as the users **aaron** or **john**, we would write:

trinity ALL=(alex, john) ALL

Also, this is wrapped in ( ) parentheses which hints us that the field is optional. So, a line like:

trinity ALL= ALL

is also valid.

We mentioned that in the fourth field we can specify a list of commands. With our previous entries, the user or group granted sudo privileges could execute any command. But we could limit them like this:

trinity ALL=(ALL) /bin/ls, /bin/stat

Now trinity could run commands such as:

sudo ls /

sudo stat /bin/

Only "ls" and "stat" commands will work. If trinity tries a command such as:

sudo echo "Test passed?"

she will get this error:

Sorry, user trinity is not allowed to execute '/bin/echo Test passed?' as root on centos-vm.

And since we specified the third field is optional, this line

trinity ALL=(ALL) /bin/ls, /bin/stat

could also be written like this:

trinity ALL= /bin/ls, /bin/stat

We know that the first time we run a sudo command in a session, it asks for our current user's password. In our sudoers file, we see a hint about how we could get rid of this requirement.

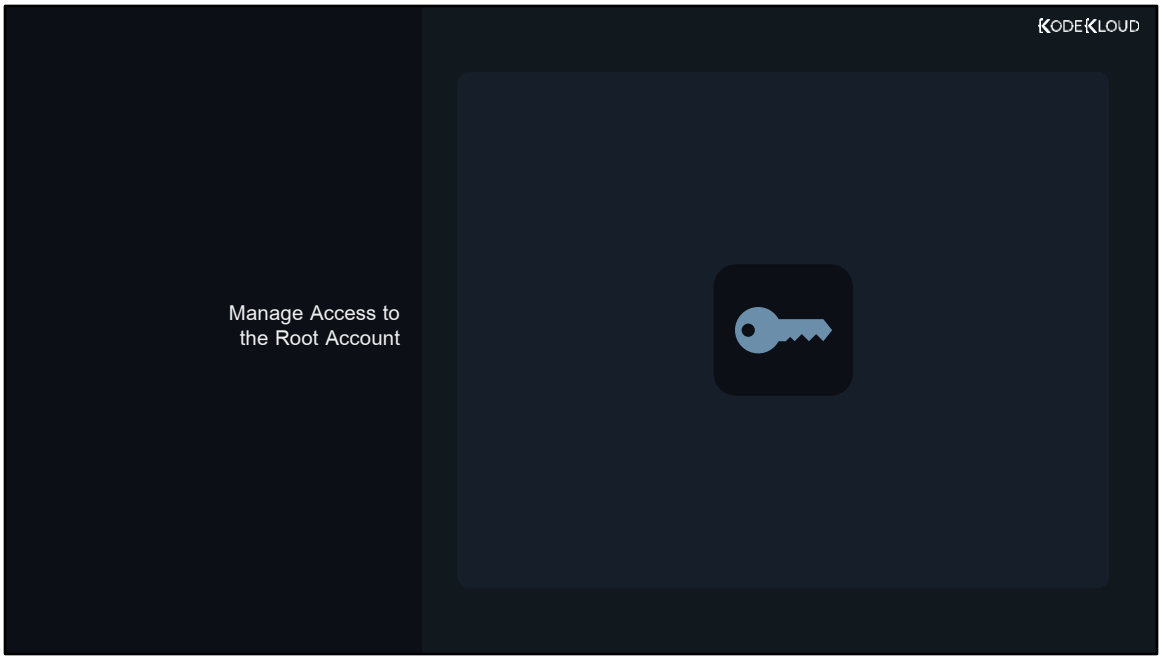
So, we could use the example in the comments:

%wheel ALL=(ALL) NOPASSWD: ALL

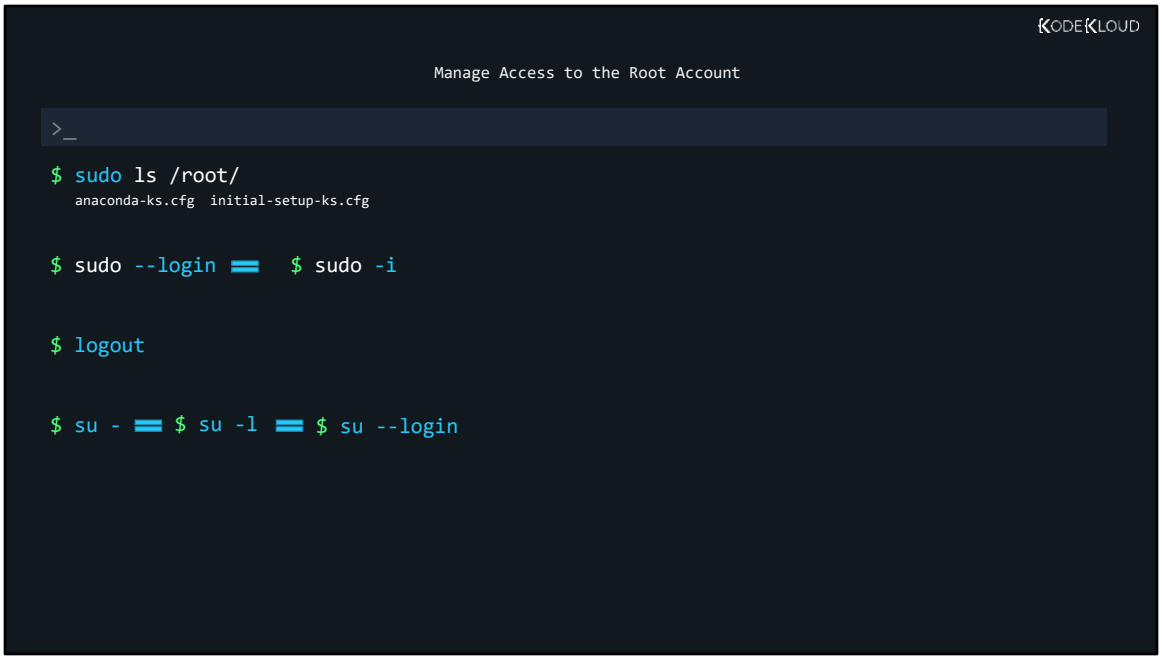
And figure out how to apply this for our user trinity. If we want her to be able to run sudo commands,

without providing her password, we could write this line in the sudoers file:

trinity ALL=(ALL) NOPASSWD: ALL



Now, let's examine how to manage access to the Root account in Linux.



We already saw one method to temporarily become root whenever needed. When we run a command such as

sudo ls /root/

it's basically the same as if the root user would execute "ls /root/".

But what if we want to log in as root? For a user with sudo access, we can enter this command:

sudo --login

or equivalent

sudo -i

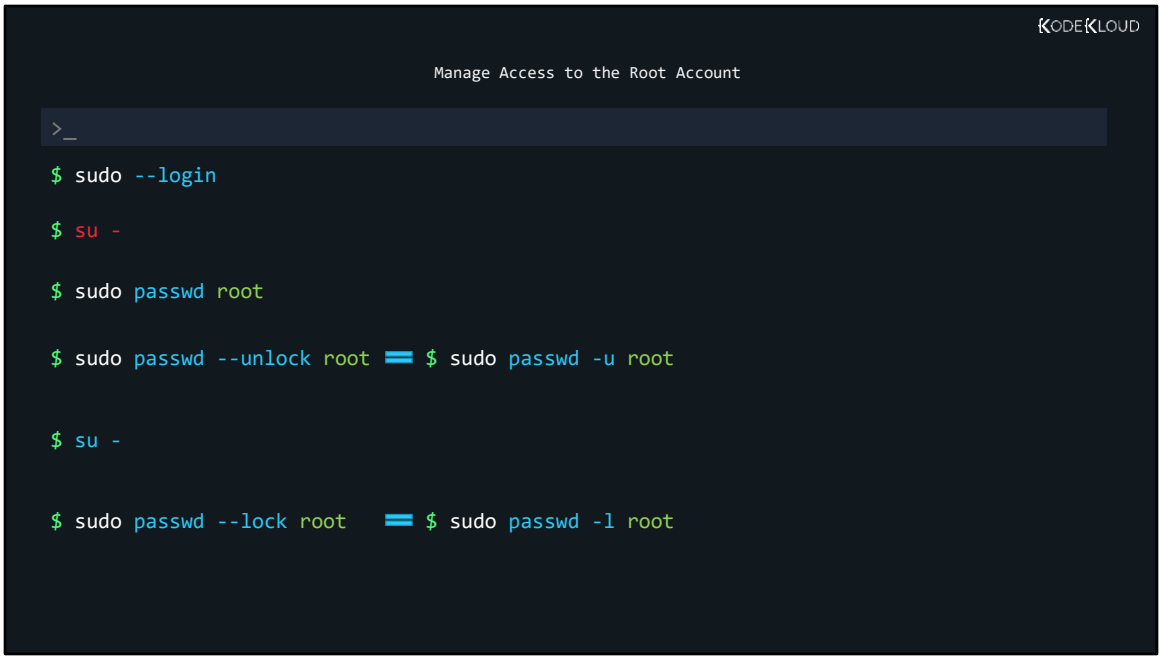
And that's it, we're logged in as root. To exit from root's session, we'll type:

logout

If the user does not have sudo privileges, but knows root's password, they can use:

su -
su -l
su --login

All these commands do the same thing: log you in as root.



Some systems might have the root account locked. This does not mean that we cannot use the root user. It just means that we cannot do a regular log in, with a password. When root is locked, we can still use

sudo --login

to log in as root. But we cannot use

su -

as that would ask for root's password, which is currently locked.

If we want to allow people to log in as root, with a password, we have two options:

1. If root never had a password set, we just choose a new password for it:

sudo passwd root

2. If root had a password set in the past, but then, the account was locked for some reason, we can unlock it with:

sudo passwd --unlock root
sudo passwd -u root

After one of these steps, we can run

su -

and type the password for root to log in.

Of course, we could also find ourselves in the reverse scenario. Imagine this: currently, people can log in as "root". But we figure that this is a bit insecure. So, we can lock password-based logins to the root account with:

sudo passwd --lock root
sudo passwd -l root

Other logins might still be possible if they were previously set up. For example, if an administrator has set up logins with an SSH private key, they'll still be able to log in even if the root account is locked.

Make sure to only lock root if your user can use sudo commands. With no root login and no sudo, you'll find yourself in the situation of not being able to become root at all, effectively locking yourself out, not able to change important system settings anymore.

Access the labs associated with this course using this link: https://kode.wiki/linux-labs


KodeKloud