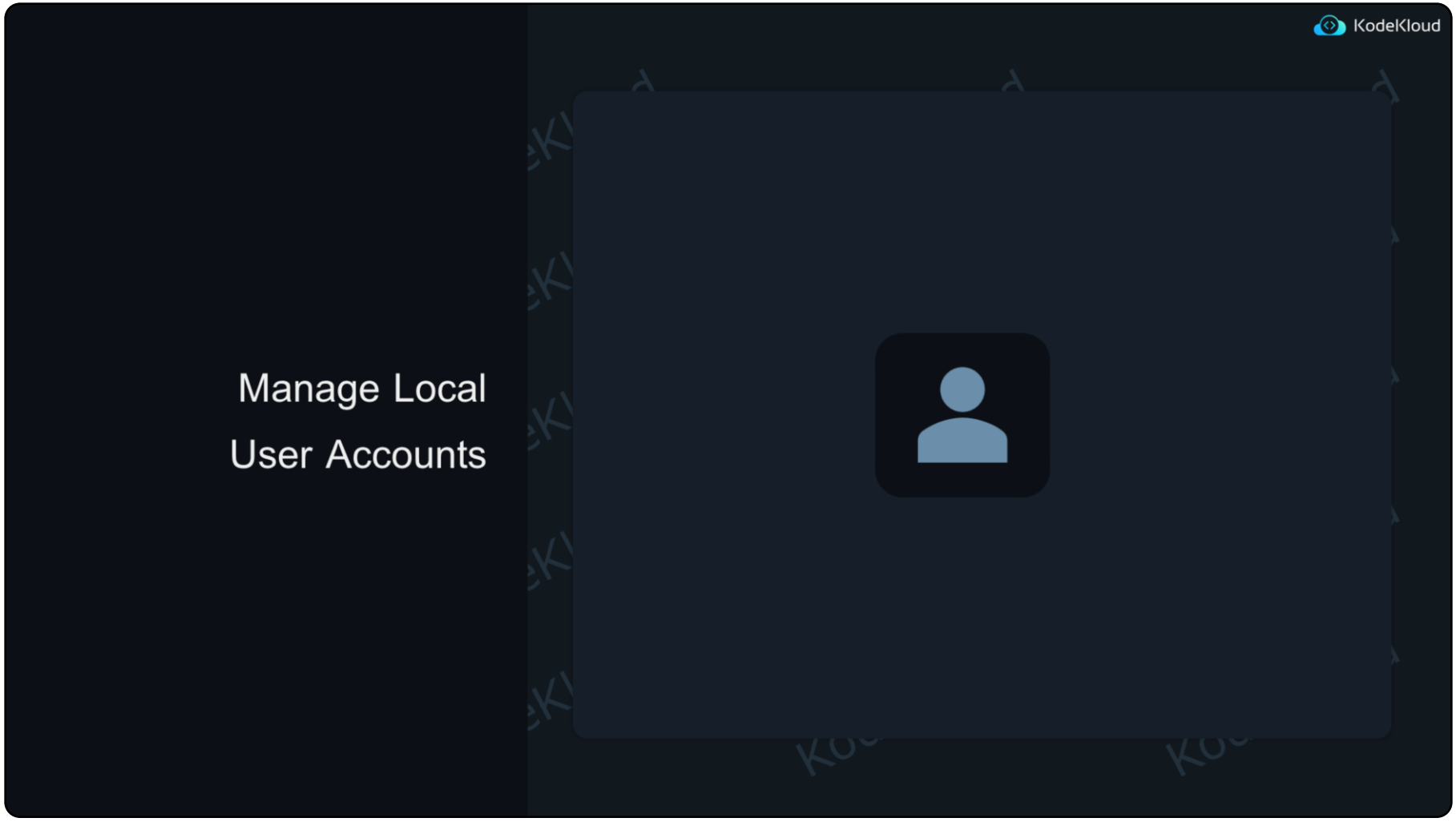
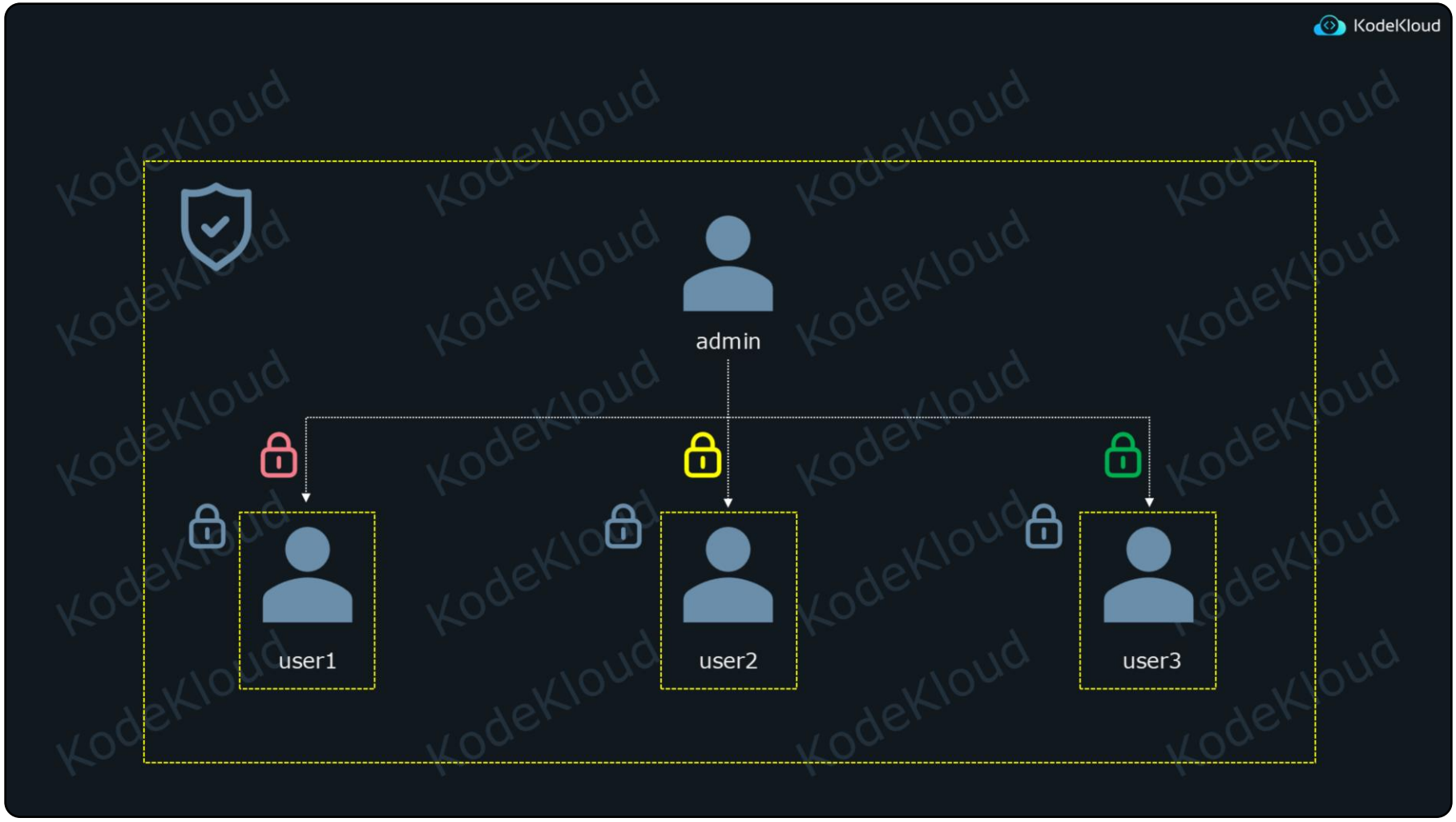
KodeKloud

Visit www.kodekloud.com to discover more.



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 based on 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 adduser. The simplest form we can use is:

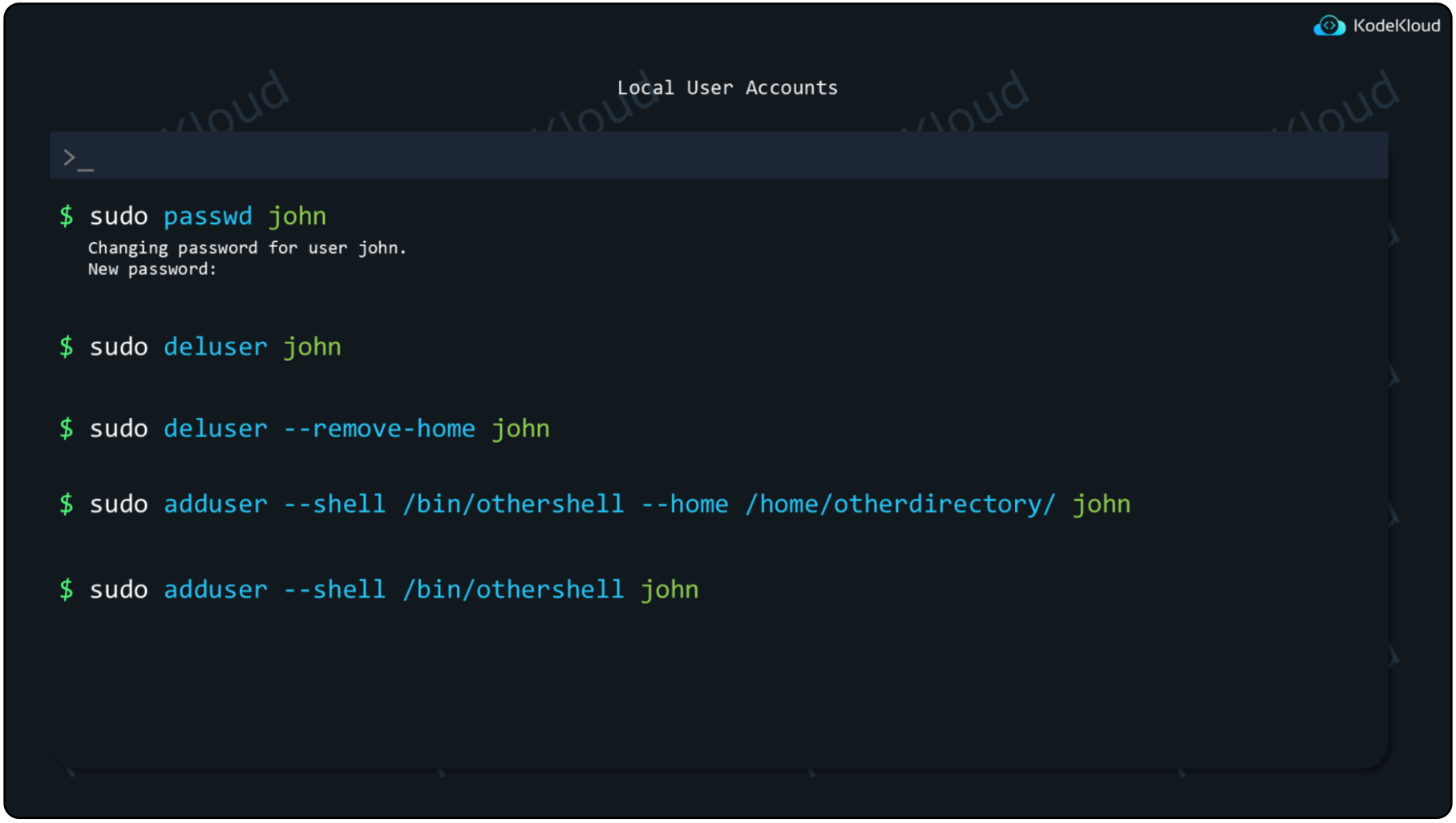
sudo adduser john

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

We'll be prompted to choose a password for this user, and then repeat the same password. Then we can type some user information, like the full name of this user, a phone number, and so on. Usually we'll just press Enter in those fields to skip adding such information. And at the end we'll have to type "y" to confirm that the information is correct and we want to create the user.

After we run this the following things happen:

A new user called "john" is added to the system
A new 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.



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 deluser john

Note, however, that this will only delete the "john" user account. Also, the group with the same name, "john" might get auto-removed, if no other members are a part of this group. 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 deluser command also remove the user's home directory and his/her mail spool with:

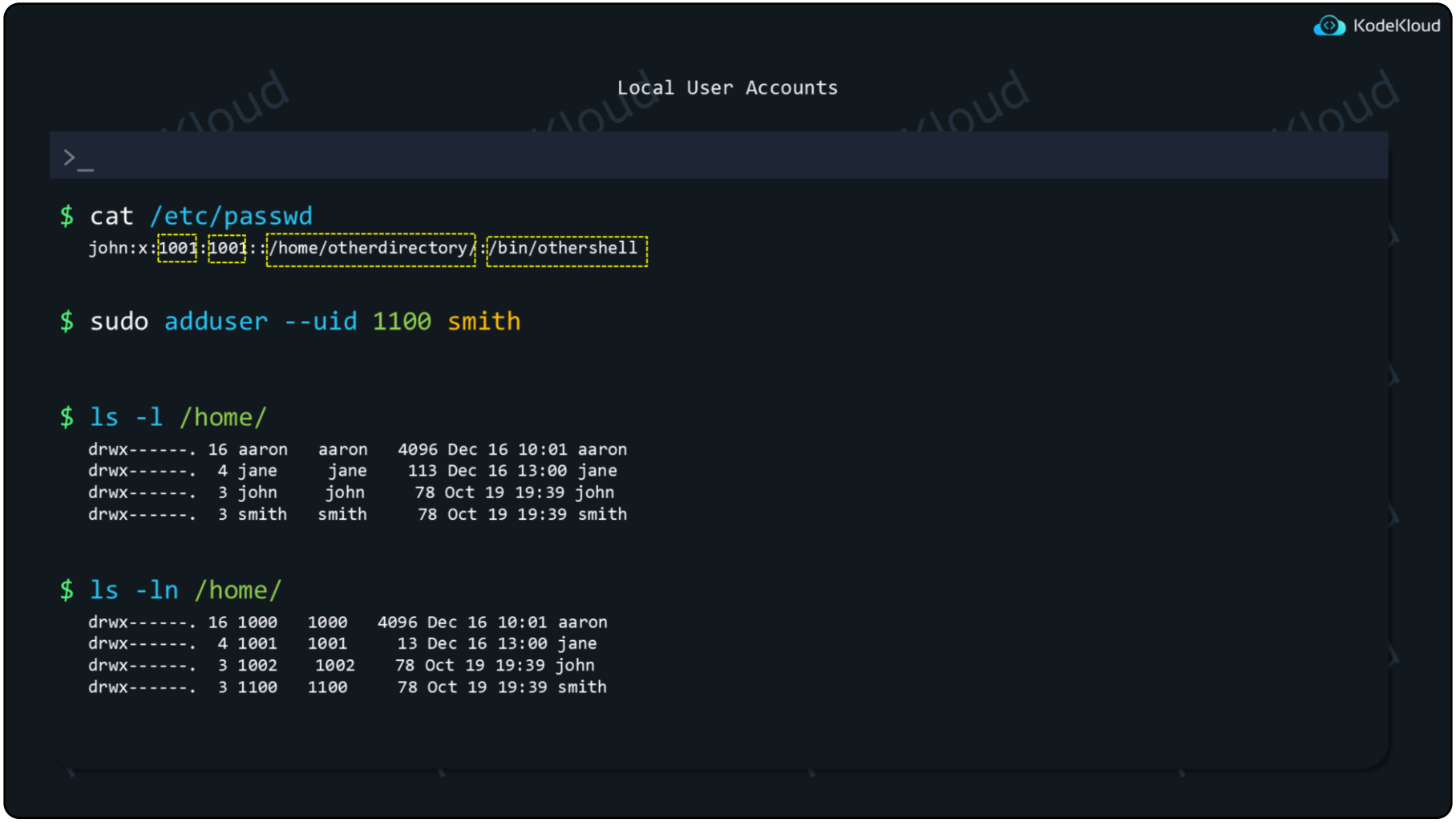
sudo deluser --remove-home john

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

sudo adduser --shell /bin/othershell --home /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, and omit the home directory option:

sudo adduser --shell /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/othershell

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.

adduser 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 adduser --uid 1100 smith

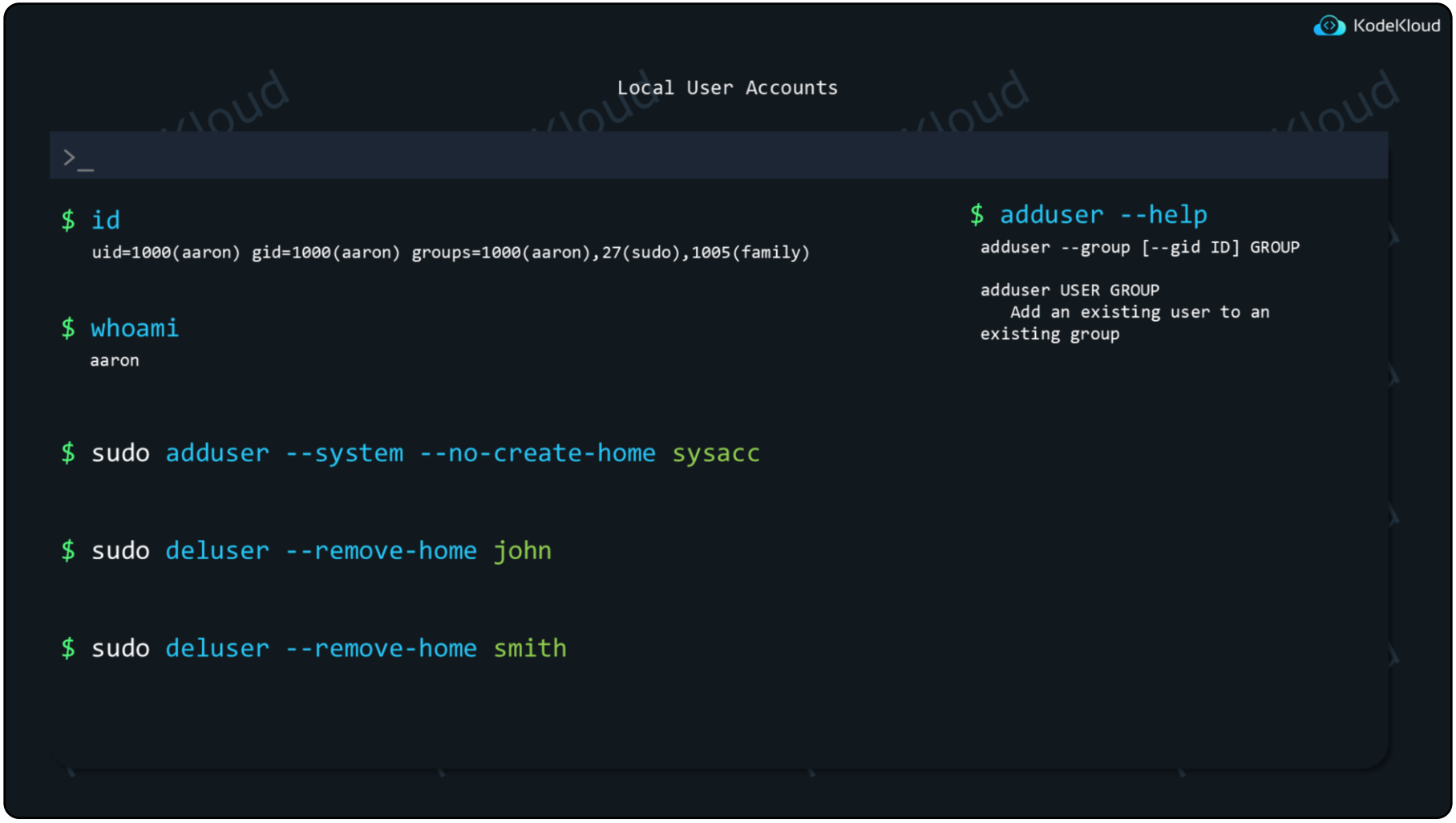
The user "smith" will have the numeric ID 1100, but also the group called "smith" will get a numeric ID of 1100.

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 adduser –system --no-create-home sysacc

We also added the --no-create-home option to tell our command not to create a home directory. Since system accounts don't usually need these.

The numeric IDs of system accounts are usually numbers smaller than 1000. So, we might see an ID like 114 or 115 for our sysacc account.

But why would we create these? User accounts are intended for people. System accounts are intended for programs. Usually, daemons use system accounts. We might see something like a database server daemon running under a system account.

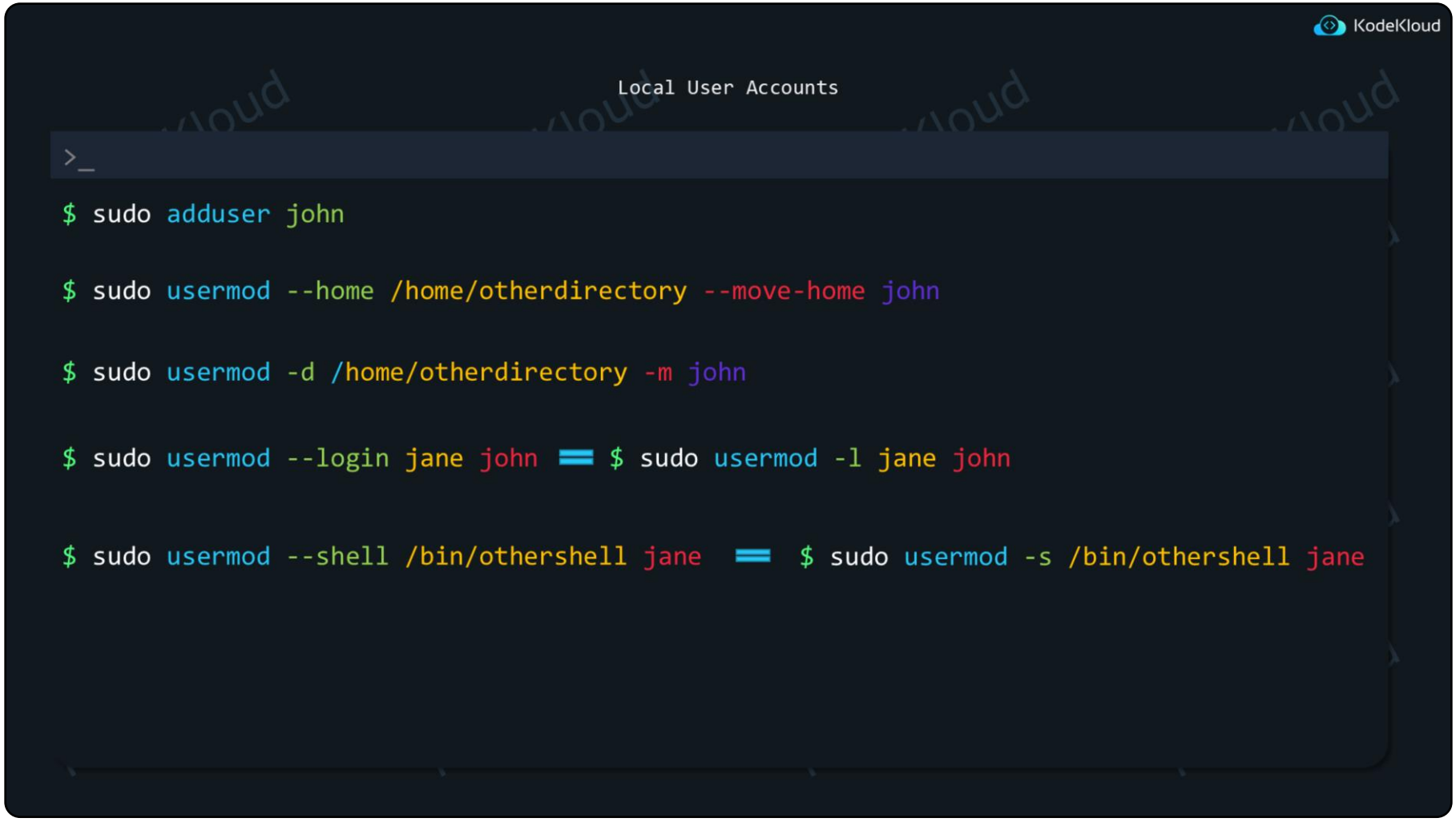
In case you're following along with these commands inside a virtual machine, remove the two user accounts we previously created:

sudo deluser --remove-home john

sudo deluser --remove-home smith

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

adduser --help



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

sudo adduser 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 type:

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

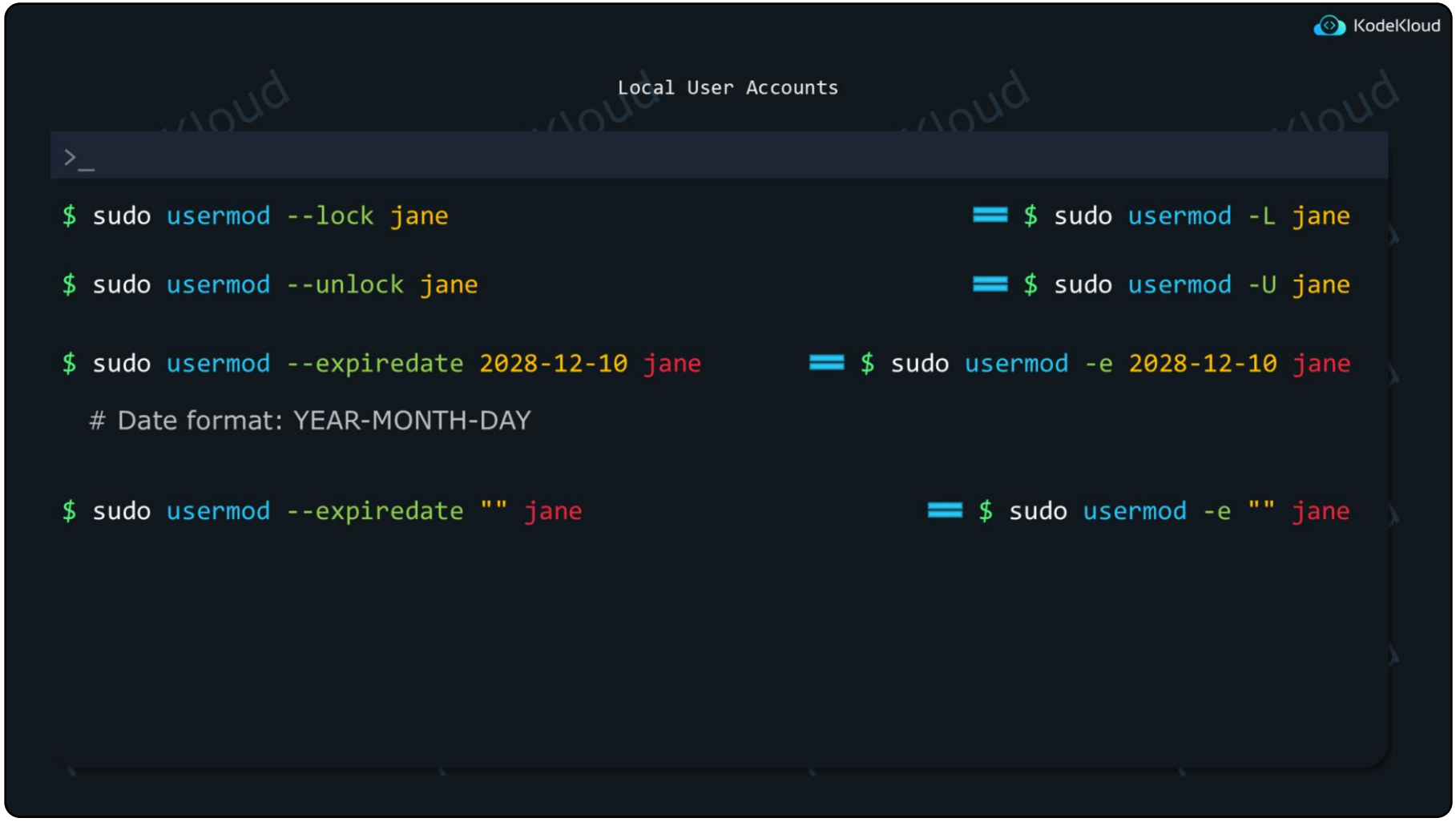
sudo usermod -l jane john

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. We'll discuss SSH keys in a future lesson.

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 2028-12-10 jane

or equivalent

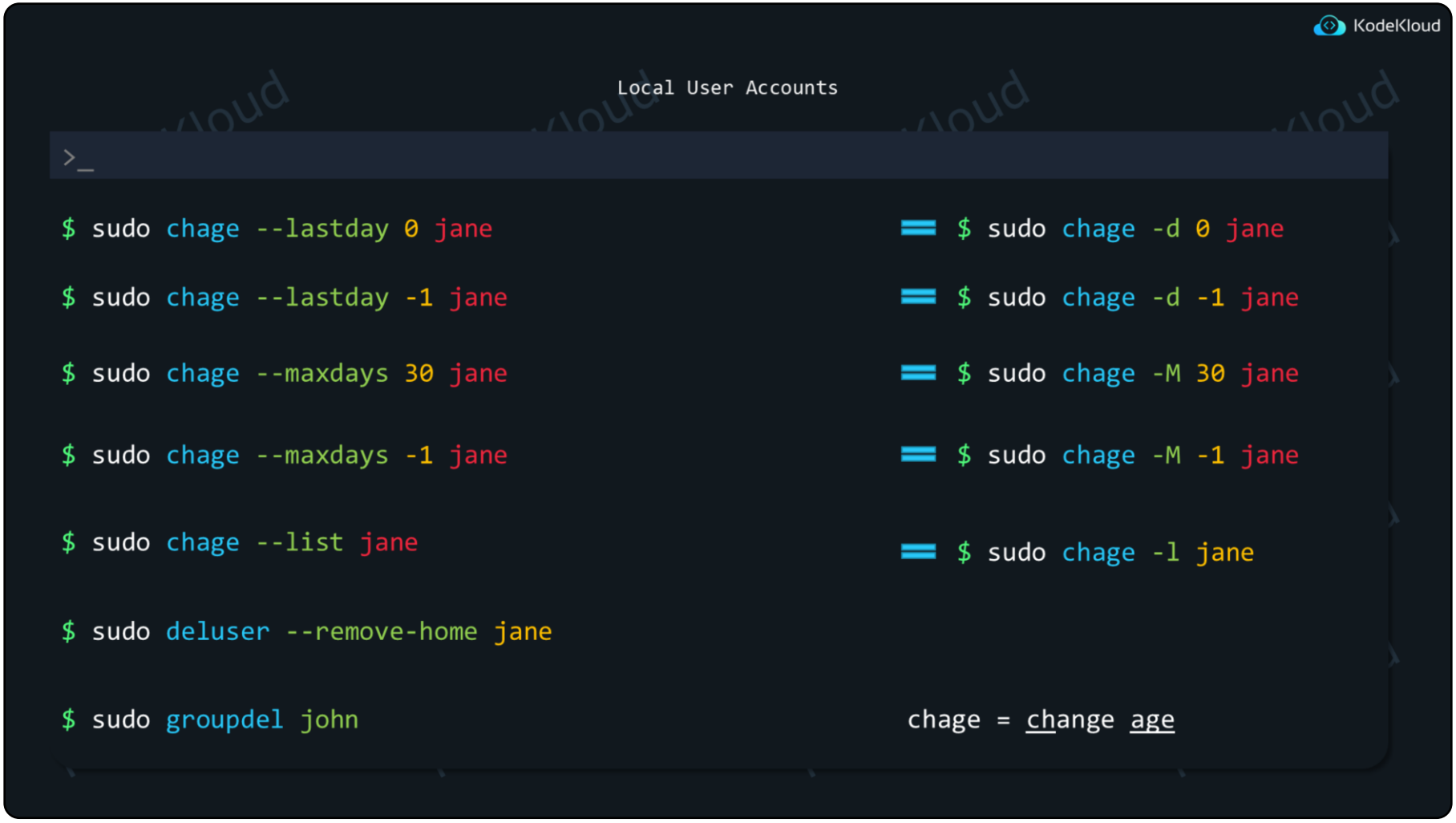
sudo usermod -e 2021-12-10 jane

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:

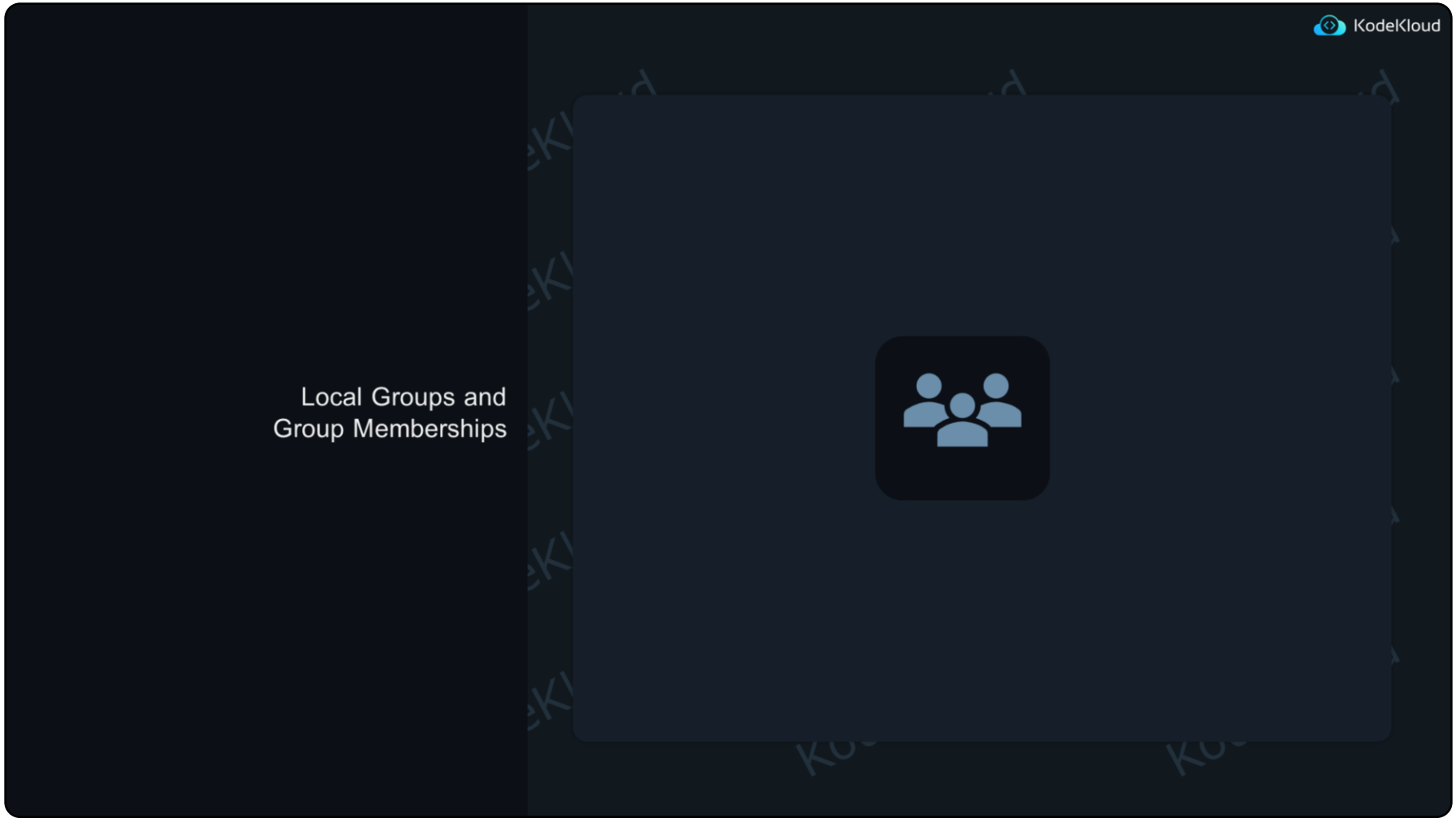
sudo chage --list jane
sudo chage -l jane

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

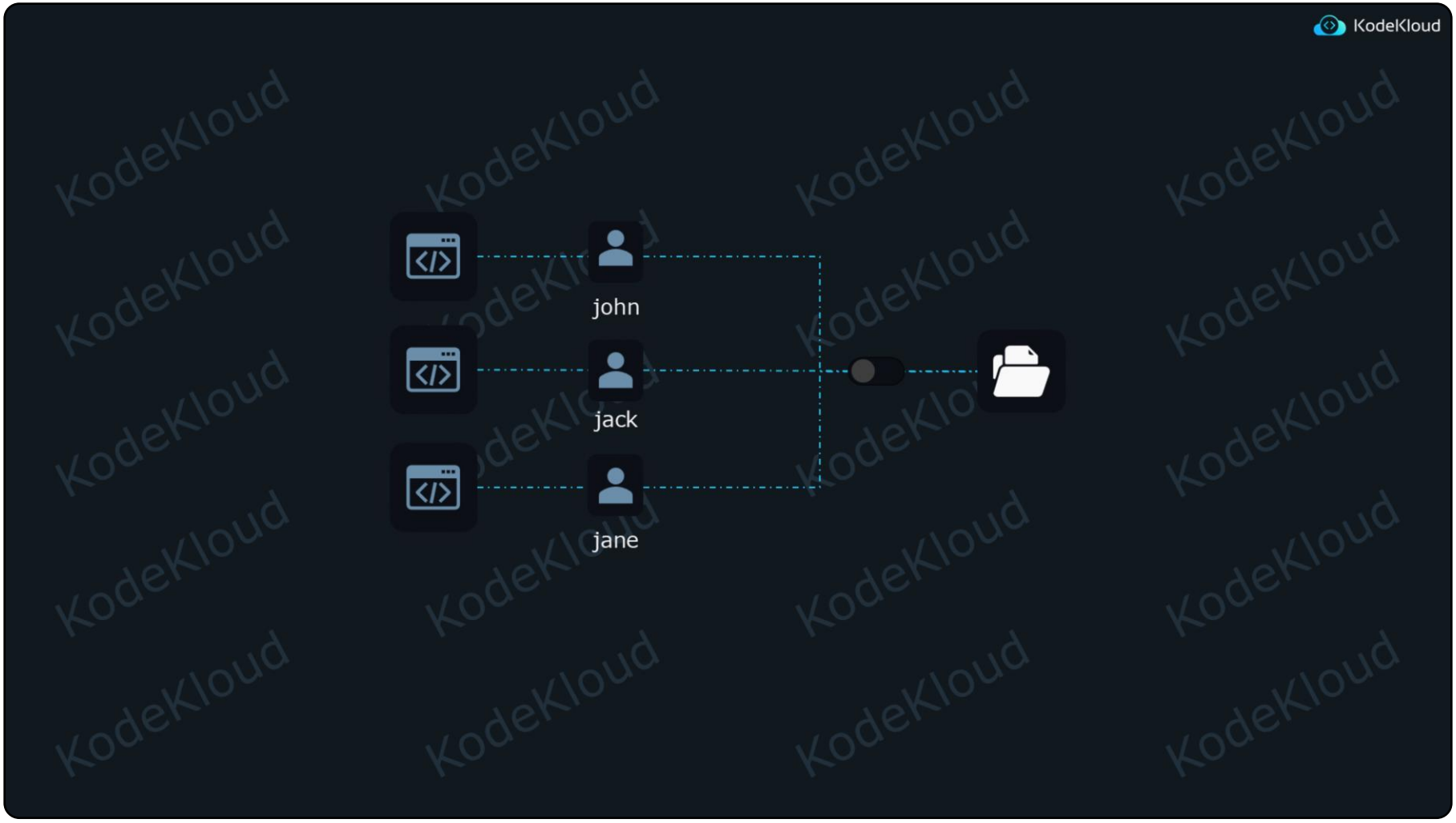
```
sudo deluser --remove-home jane
sudo groupdel john
```

KodeKloud

Visit www.kodekloud.com to discover more.



Now, let's explore how to create, delete, and modify local groups and group memberships in Linux.



Each user can belong to one or more groups. Why are these useful? Here are a few examples.

Let's say we have a directory full of files that our developers need to work on. So, they all need read-write permissions. We'd have to allow three user accounts to edit these files: john, jack and jane. An elegant solution to this problem is to create a new group called developers. Then we add our three users to the developers group. Finally, we make the developers group the owner of those files. And we change permissions so that the developers group can read and write to them. Now john, being part of the developers group can easily edit those files. And if we want to temporarily deny john access, we just remove him from the developers group. Or if a new member joins our team, we can just add their user account to the developers group

and with a simple change, they have read-write access to those files.

We can see how this makes things easier to understand from an administrator's perspective. It's like assigning roles to user accounts. Or like user accounts have a label: user X is a developer or is not a developer. All of this by simply deciding if they will be part of the developers group or not a part of that group.

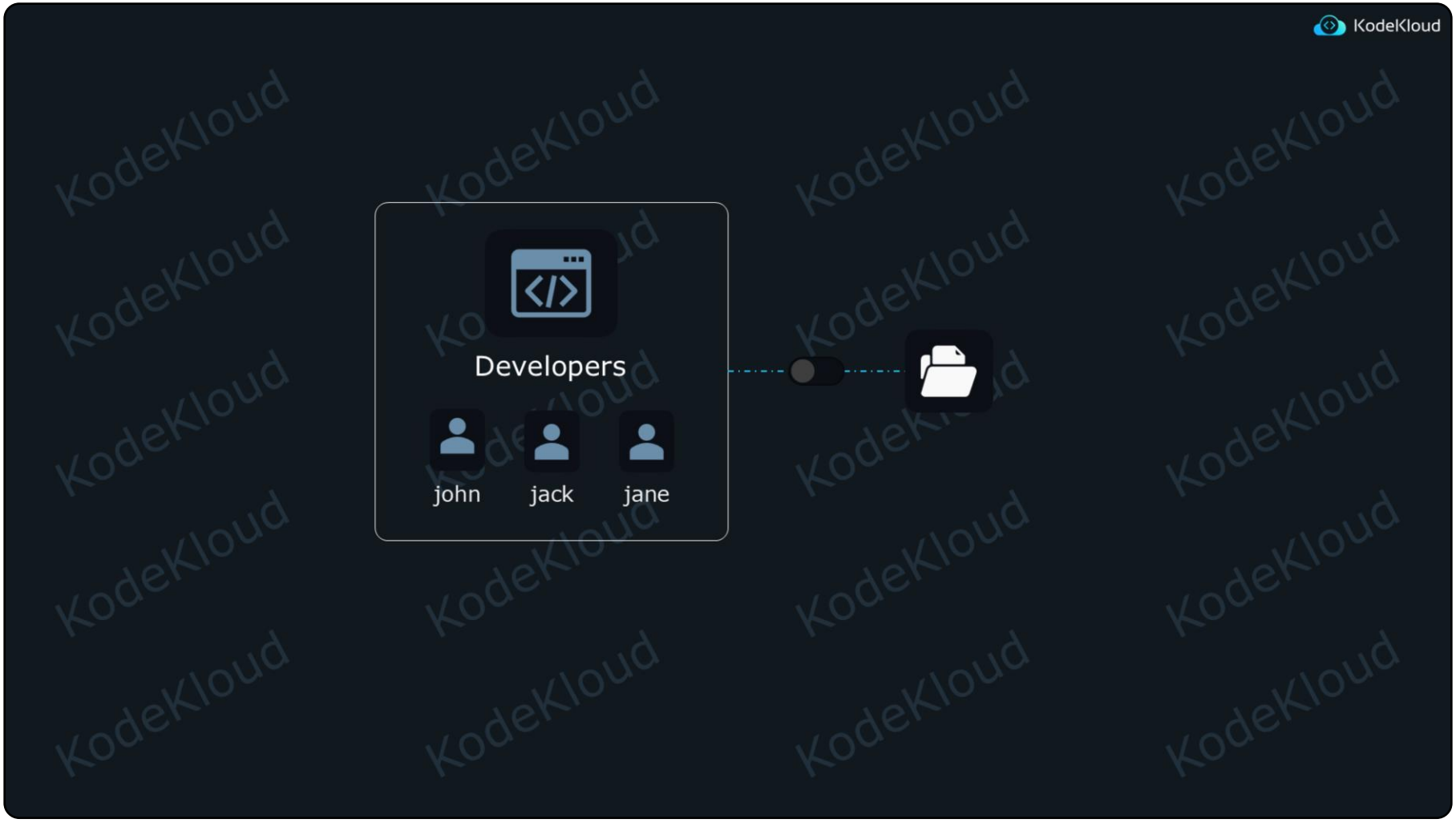
And speaking of roles for user accounts, groups can have other special effects. For example, being part of some group can grant special privileges on the system. Two common examples:

1. Users in the "wheel" group on Red Hat, or "sudo" on Ubuntu are allowed to do pretty much anything on the system. They can run any program with root privileges, root being the most powerful user account on Linux.

2. Users in the "docker" group can manage Docker containers.

We said that the user can belong to multiple groups. But one of these groups is special. One of them is the primary group, while all of the others are secondary, or supplementary groups. The primary group is also called a login group. That's because as soon as the user logs in, this becomes his/her main, active group. But it's hard to understand, with theory alone, so let's see what's so special about this primary group. Here are two practical examples:

When a user launches a program, it is said that it runs "under" that user account and group. Otherwise said, the program runs with the same privileges that the user account and its primary group have. And here's another, more visible example. When a user creates a file, this file will automatically be owned by their user account and their primary/login group.



Each user can belong to one or more groups. Why are these useful? Here are a few examples.

Let's say we have a directory full of files that our developers need to work on. So, they all need read-write permissions. We'd have to allow three user accounts to edit these files: john, jack and jane. An elegant solution to this problem is to create a new group called developers. Then we add our three users to the developers group. Finally, we make the developers group the owner of those files. And we change permissions so that the developers group can read and write to them. Now john, being part of the developers group can easily edit those files. And if we want to temporarily deny john access, we just remove him from the developers group. Or if a new member joins our team, we can just add their user account to the developers group

and with a simple change, they have read-write access to those files.

We can see how this makes things easier to understand from an administrator's perspective. It's like assigning roles to user accounts. Or like user accounts have a label: user X is a developer or is not a developer. All of this by simply deciding if they will be part of the developers group or not a part of that group.

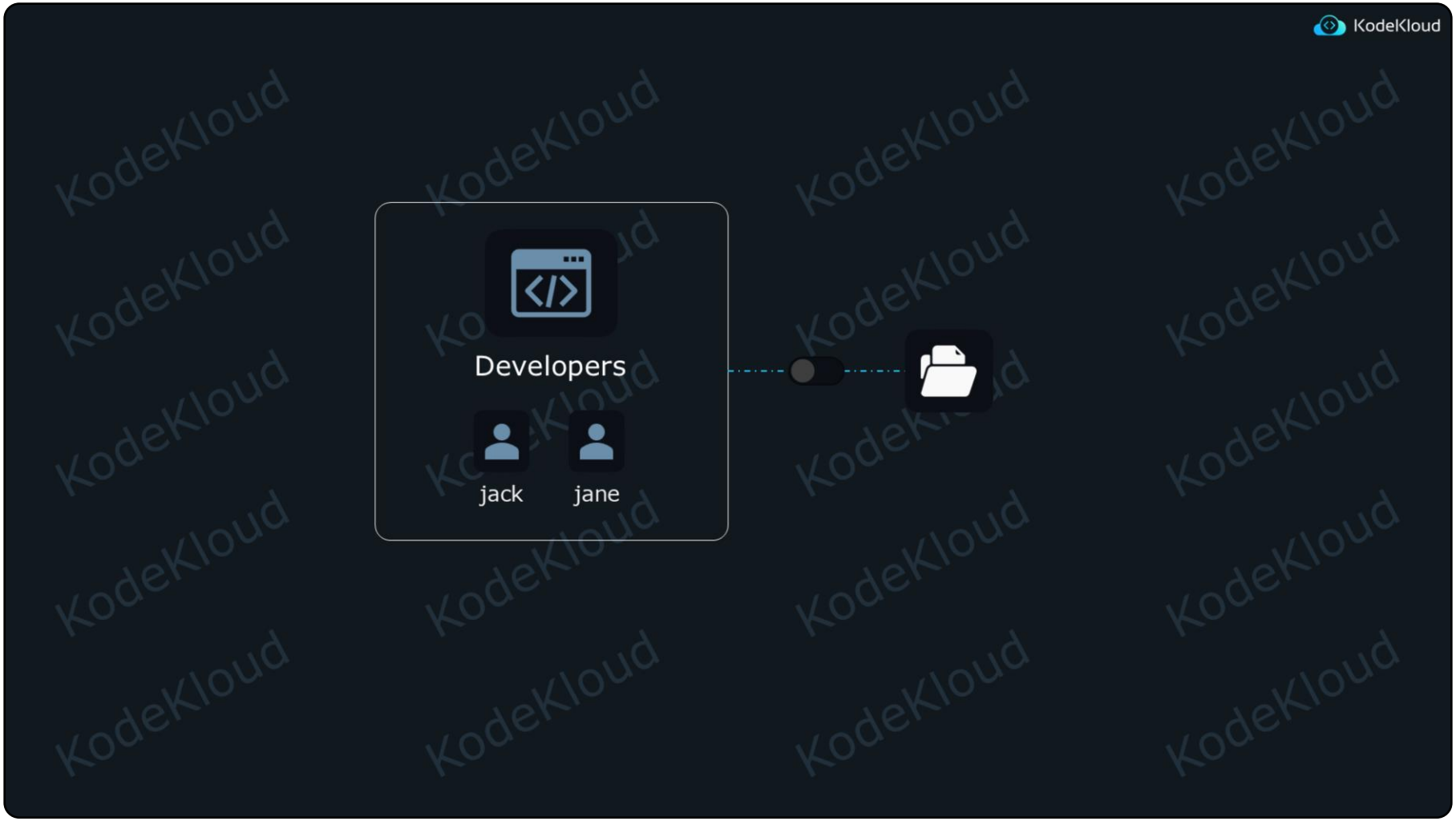
And speaking of roles for user accounts, groups can have other special effects. For example, being part of some group can grant special privileges on the system. Two common examples:

1. Users in the "wheel" group on Red Hat, or "sudo" on Ubuntu are allowed to do pretty much anything on the system. They can run any program with root privileges, root being the most powerful user account on Linux.

2. Users in the "docker" group can manage Docker containers.

We said that the user can belong to multiple groups. But one of these groups is special. One of them is the primary group, while all of the others are secondary, or supplementary groups. The primary group is also called a login group. That's because as soon as the user logs in, this becomes his/her main, active group. But it's hard to understand, with theory alone, so let's see what's so special about this primary group. Here are two practical examples:

When a user launches a program, it is said that it runs "under" that user account and group. Otherwise said, the program runs with the same privileges that the user account and its primary group have. And here's another, more visible example. When a user creates a file, this file will automatically be owned by their user account and their primary/login group.



Each user can belong to one or more groups. Why are these useful? Here are a few examples.

Let's say we have a directory full of files that our developers need to work on. So, they all need read-write permissions. We'd have to allow three user accounts to edit these files: john, jack and jane. An elegant solution to this problem is to create a new group called developers. Then we add our three users to the developers group. Finally, we make the developers group the owner of those files. And we change permissions so that the developers group can read and write to them. Now john, being part of the developers group can easily edit those files. And if we want to temporarily deny john access, we just remove him from the developers group. Or if a new member joins our team, we can just add their user account to the developers group

and with a simple change, they have read-write access to those files.

We can see how this makes things easier to understand from an administrator's perspective. It's like assigning roles to user accounts. Or like user accounts have a label: user X is a developer or is not a developer. All of this by simply deciding if they will be part of the developers group or not a part of that group.

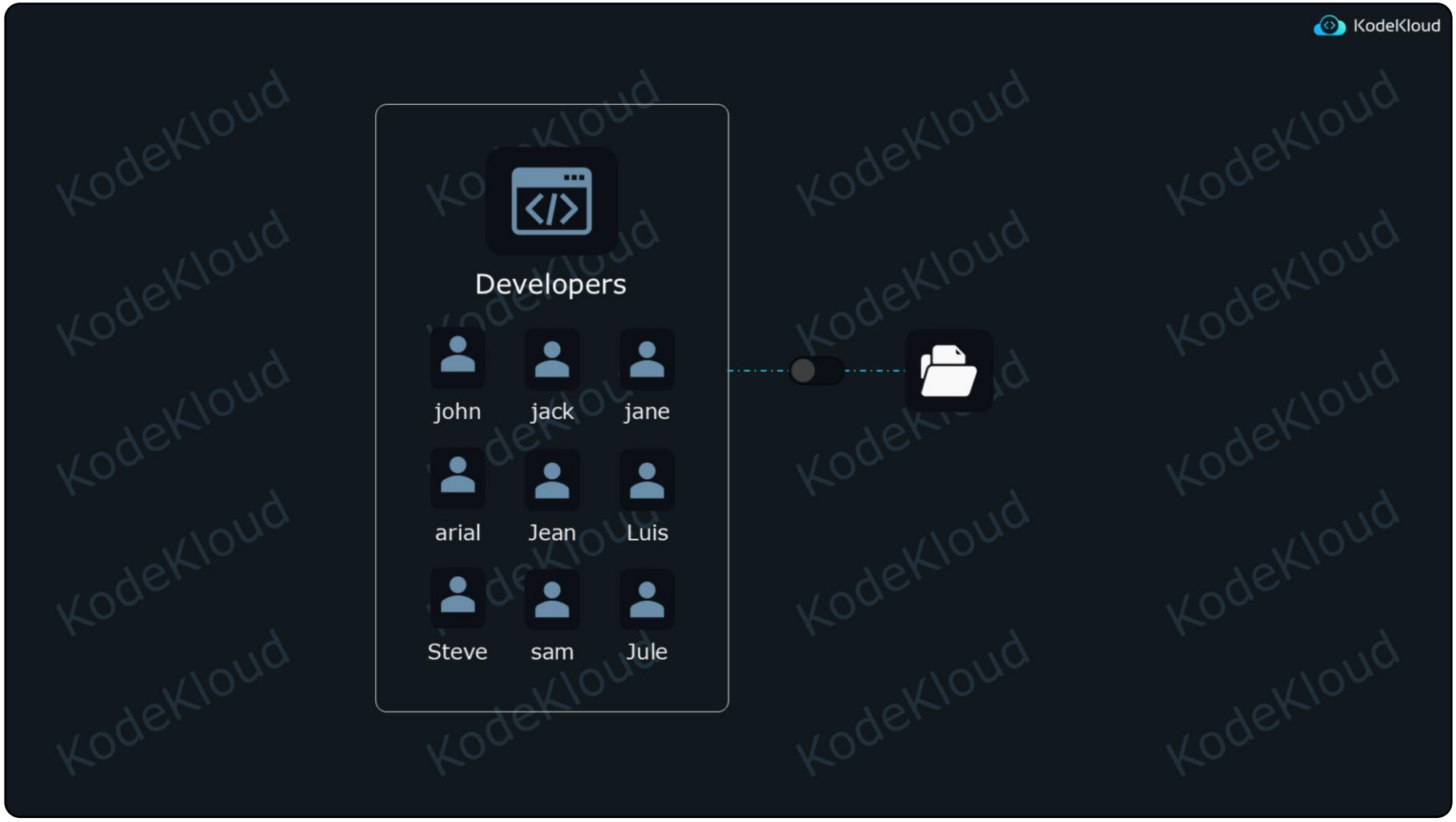
And speaking of roles for user accounts, groups can have other special effects. For example, being part of some group can grant special privileges on the system. Two common examples:

1. Users in the "wheel" group on Red Hat, or "sudo" on Ubuntu are allowed to do pretty much anything on the system. They can run any program with root privileges, root being the most powerful user account on Linux.

2. Users in the "docker" group can manage Docker containers.

We said that the user can belong to multiple groups. But one of these groups is special. One of them is the primary group, while all of the others are secondary, or supplementary groups. The primary group is also called a login group. That's because as soon as the user logs in, this becomes his/her main, active group. But it's hard to understand, with theory alone, so let's see what's so special about this primary group. Here are two practical examples:

When a user launches a program, it is said that it runs "under" that user account and group. Otherwise said, the program runs with the same privileges that the user account and its primary group have. And here's another, more visible example. When a user creates a file, this file will automatically be owned by their user account and their primary/login group.



Each user can belong to one or more groups. Why are these useful? Here are a few examples. We have a directory full of files that our developers need to work on. So, they all need read-write permissions. We'd have to allow three user accounts to edit these files: john, jack and jane. An elegant solution to this problem is to create a new group called developers. Then we add our three users to the developers group. Finally, we make the developers group the owner of those files. And we change permissions so that the developers group can read and write to them. Now john, being part of the developers group can easily edit those files. And if we want to temporarily deny john access, we just remove him from the developers group. Or if a new member joins our team, we can just add their user account to the developers group and boom, they have read-write access to those files. We can see how this makes things easier to understand from an administrator's perspective. It's like assigning roles to user accounts. Or like user accounts have a label: is a developer or is not a developer. All of this by simply

deciding if they will be part of the developers group or not a part of that group.

And speaking of roles for user accounts, groups can have other special effects. For example, being part of some group can grant special privileges on the system. Two common examples:

1. Users in the "wheel" or "sudo" group are allowed to do pretty much anything on the system. They can run any program with root privileges, root being the most powerful user account on Linux.

2. Users in the "docker" group can manage Docker containers.

We said that the user can belong to multiple groups. But one of these groups is special. One of them is the primary group, while all of the others are secondary, or supplementary groups. The primary group is also called a login group. That's because as soon as the user logs in, this becomes his/her main group. But it's hard to understand, with theory alone, so let's see what's so special about this primary group. Here are two practical examples:

When a user launches a program, it is said that it runs "under" that user account and group. Otherwise said, the program runs with the same privileges that the user account and its primary group have. And here's another, more visible example. When a user creates a file, this file will automatically be owned by their user account and their primary/login group.

If you want to follow along with this exercise, you'll need a user called "john" beforehand:

sudo adduser john

It's easy enough to create a new group, called "developers", with a command like:

sudo groupadd developers

But how do we add our user, "john", to this new group?

The easiest way to add a user to a group is with the help of the gpasswd command. This name comes from the words "group password". But don't let the name confuse you. Nowadays, group passwords are almost never used in practice. So, the main use-case for the gpasswd utility is to add or remove users from certain groups.

We can add john to our developers group with this command:

sudo gpasswd --add john developers

Or equivalent short form:

sudo gpasswd -a john developers

If we want to confirm that this worked, we can see the groups that john belongs to, with the command:

groups john

Output:
john : john developers

The first group (after :) is the primary/login group. The rest are the secondary/supplementary groups.

To remove a user from a group:

sudo gpasswd --delete john developers

Or equivalent:

```
sudo gpasswd -d john developers
```

Previously, we added john to a secondary, or supplementary group. But on rare occasions, we might want to change the user's primary/login group instead . We can do so with a command like:

sudo usermod -g developers john

It's important not to confuse this with the capital -G as that option changes the secondary groups, not the primary one. To avoid this mistake, we can make a habit to use the equivalent

long option, --gid instead of -g:

sudo usermod --gid developers john

And now

groups john

will show us that the primary group was changed.

Note the difference: gpasswd first expected the username then the group name. But usermod has a reverse order of group name, then username.

Use

gpasswd --help

to get a quick refresh. A line like this

-a, --add USER add USER to GROUP

will show that the USER name must come first when using gpasswd.

To rename the group called "developers" to "programmers", we can type:

sudo groupmod --new-name programmers developers

or equivalent

sudo groupmod -n programmers developers

To delete the "programmers" group:

sudo groupdel programmers

But we'll see this error if this is someone's primary group.

groupdel: cannot remove the primary group of user 'john'

To fix this, we can change john's primary group back to the "john" group.

sudo usermod --gid john john

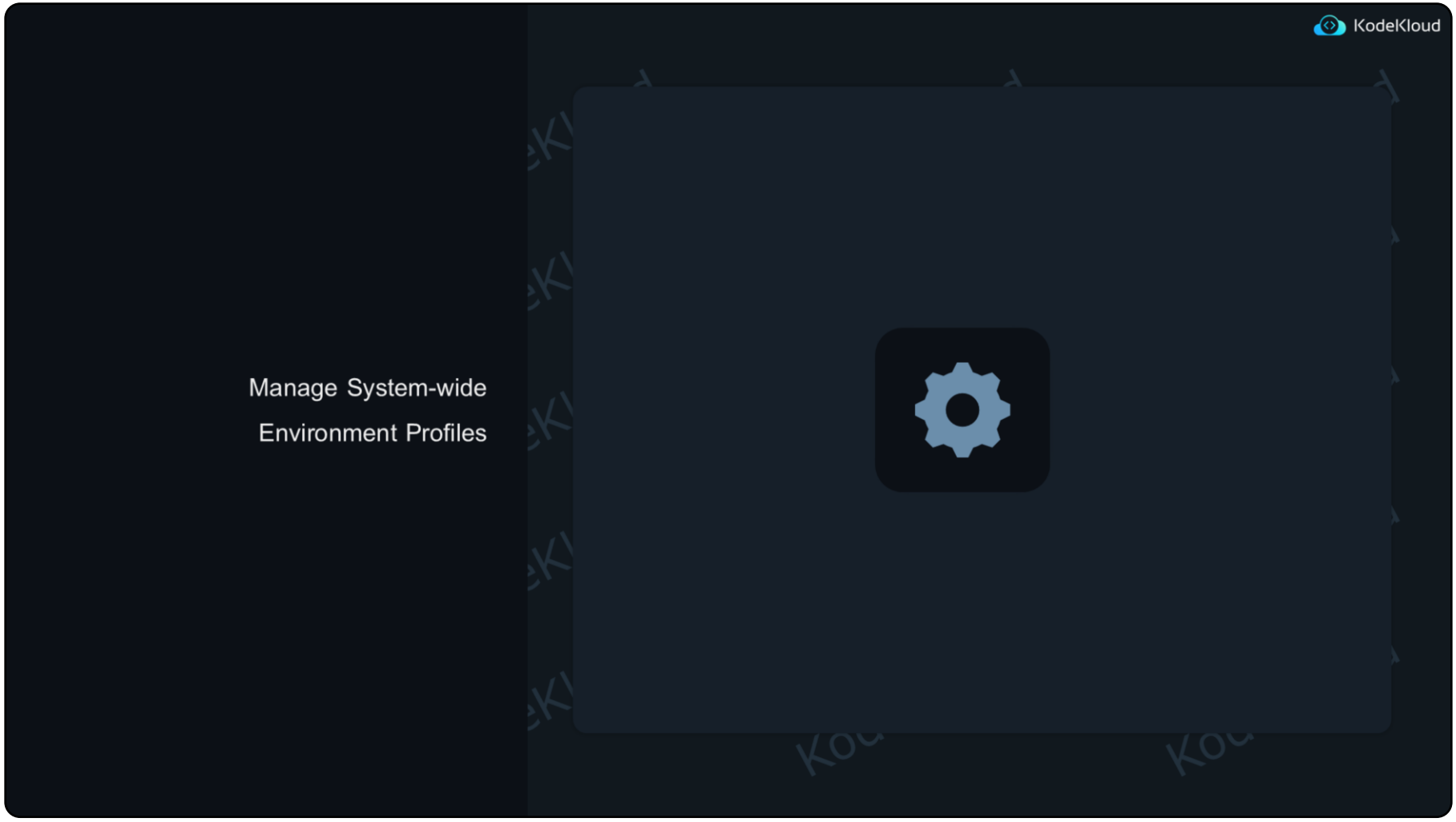
And now we can finally delete the group:

sudo groupdel programmers

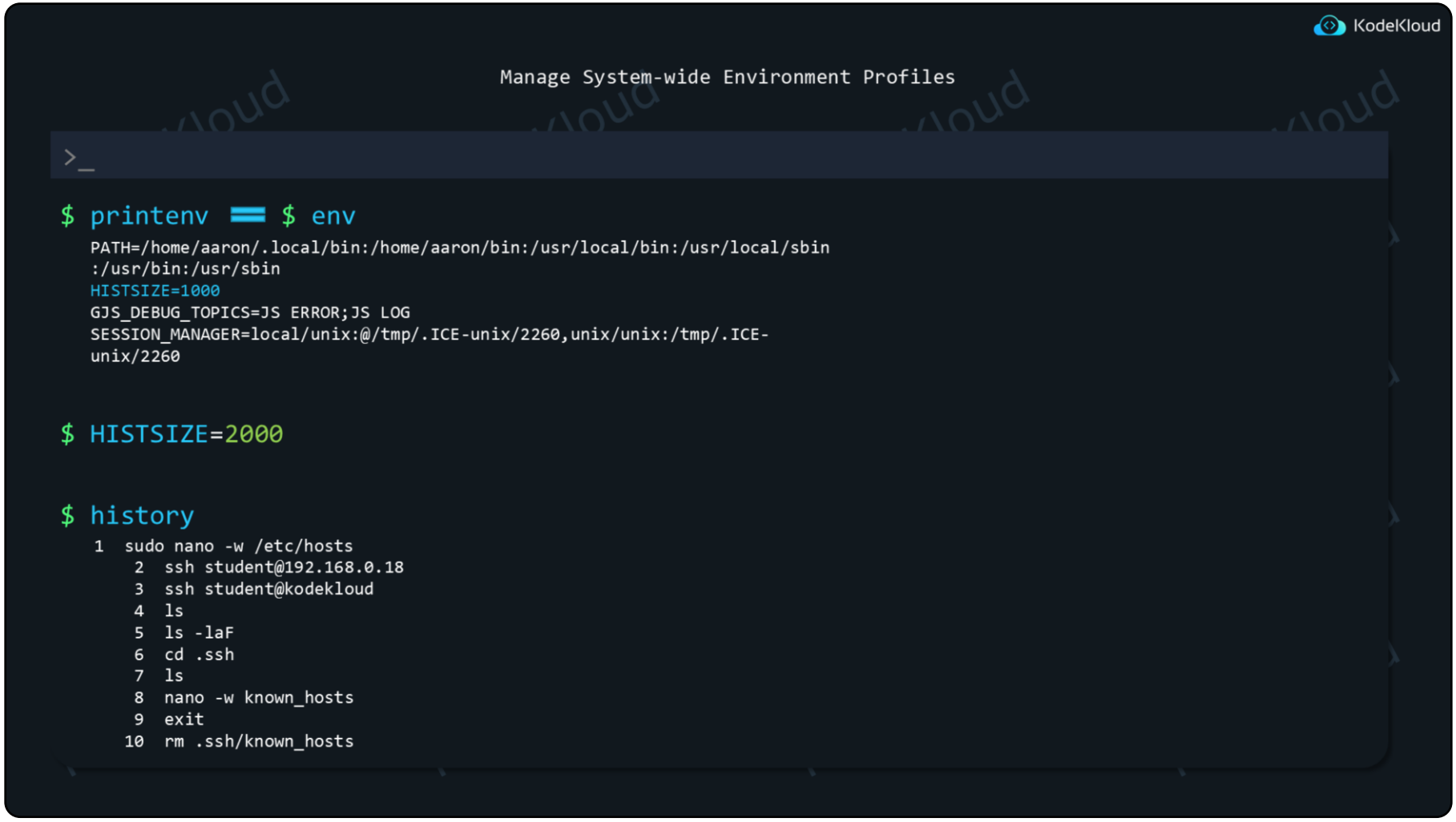
If a user is part of a secondary group, and we want to delete it, the command will work without issues. There's no need to first remove the user from that group before deleting it.

KodeKloud

Visit www.kodekloud.com to discover more.



Now, we'll look at how to manage system-wide environment profiles in Linux.



First, what is this so-called "environment"? We can see our current user's environment with a command like

printenv

or

env

What we see here are environment variables. Let's take this example from the output:

HISTSIZE=1000

This means that currently, the variable called HISTSIZE is set to 1000. And changing this to another value is as easy as writing

HISTSIZE=2000

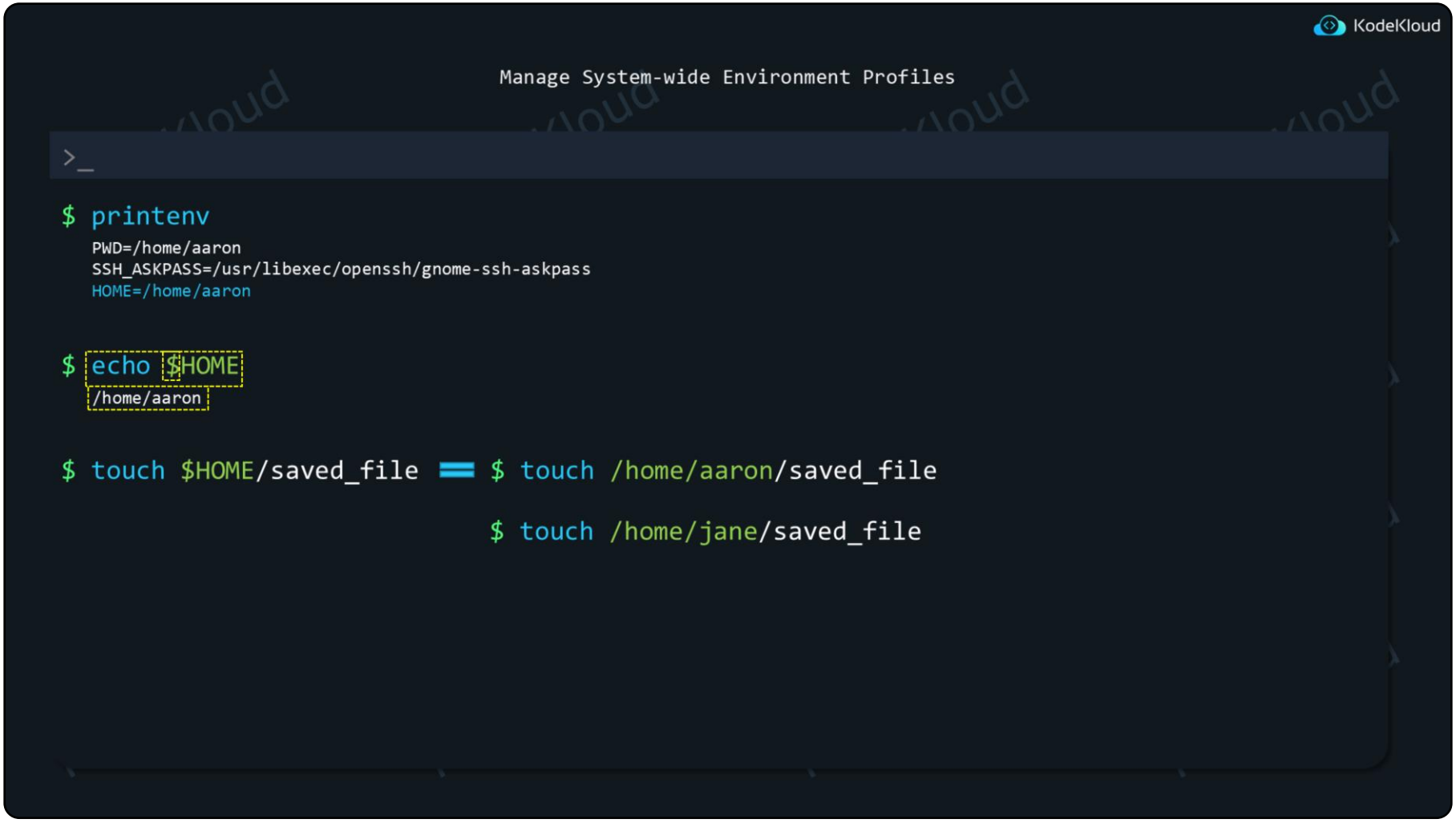
at the command line.

So, what does this do? In this case, the environment variable is used by Bash, our current login shell. Each time Bash runs, it checks out this value to know the maximum size of the command history it should save. We can check out the history of every command we ever typed in this session and previous sessions, with:

history

So, with a HISTSIZE of 1000, Bash will never save more than 1000 commands in this history.

In this case, an environment variable was used as some sort of program setting. Other times, applications look at these variables to get an idea about the sort of environment they're running in.



For example, we'll see something like this variable:

HOME=/home/aaron

This tells applications where our home directory is. If they want to save a file, they can make use of this variable to save it at the right location. To get an idea of how this works, we can run this command:

echo $HOME

Adding $ in front of a variable dumps the content of that variable in the same spot on the command line. So, writing

echo $HOME

is the same as writing

echo /home/aaron/

This can also be useful in scripts. Imagine we have a script, and we want to save a file in the user's home directory. We could write this:

touch $HOME/saved_file

Now when the user "aaron" runs it, the variable will fill in /home/aaron/ in that spot, so this command will run:

touch /home/aaron/saved_file

If Jane runs it, this command will run:

touch /home/jane/saved_file

This is useful because any user can run the script, but the command will intelligently fill in a different home directory path for each user. So, it dynamically adjusts for every user's environment.

If a user wants to add their own environment variables, they can edit the .bashrc file. We'll see something like this if we explore it:

cat .bashrc

But in this case, we want to make sure that we set some variable to a certain value, for every user that logs on to this system, not just one. To do this, we can add our changes to this file: /etc/environment.

sudo vim /etc/environment

Imagine we type this and save the file.

KODEKLOUD=https://kodekloud.com/

This variable will now be set every time a user logs in. To test this out in a virtual machine, after editing the file you can logout with this command:

logout

and then log back in. Now, if we type this:

echo $KODEKLOUD

we'll see this output

which confirms the variable has been set accordingly.

But that file is only for setting environment variables. What if we want to do more complex stuff, like run a command every time a user logs in? We can use the special directory /etc/profile.d/. Here we can create a file that ends with the .sh extension. Let's add a file called lastlogin.sh

sudo vim /etc/profile.d/lastlogin.sh

and type:

```
echo "Your last login was at: " > $HOME/lastlogin
date >> $HOME/lastlogin
```

Then save the file and exit.

This is a simple script that uses the theory we learned about in previous lessons. It basically logs the date and time when the user logged in, in a file called "lastlogin" under their home directory.

Once again, let's logout

```
logout
```

and then log back in.

If we type

```
ls
```

we'll see a new file, called lastlogin.

And if we type

```
cat lastlogin
```

we'll see our script did its job perfectly. And we even made use of the environment variable $HOME to showcase how we can actually use it in a script.

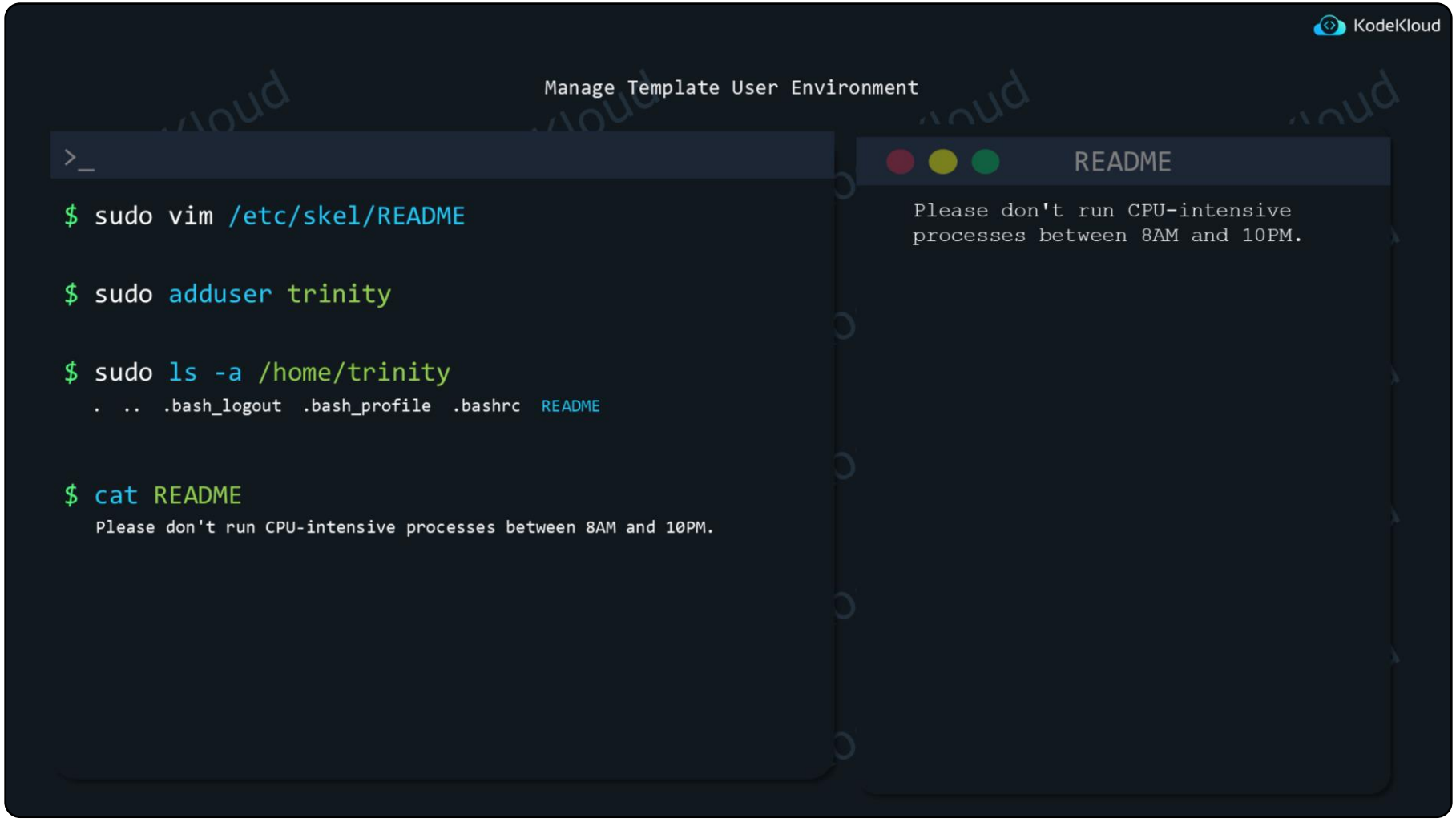
While these are Bash scripts, note that for .sh files we add to /etc/profile.d/, we don't need to add a shebang like #!/bin/bash. The system already knows that anything found in /etc/profile.d/ should be processed by our current shell, the Bash command interpreter.

KodeKloud

Visit www.kodekloud.com to discover more.

Now, let's look at managing the template user environment in Linux.

When we created new user accounts, we mentioned one action that happens by default: all files from the /etc/skel/ directory are copied to the new user's home directory. This can be very helpful when we want to create a template for the user's environment. Practical examples are easier to understand so let's jump right in and explain as we go along.



Imagine that we want to inform all new users about some default policy. Our old users already know about it. But every time we add a new user, it's likely they don't know yet. What we can do is add a custom file to /etc/skel/. Let's do that with:

sudo vim /etc/skel/README

And add this text: "Please don't run CPU-intensive processes between 8AM and 10PM." and then save our file.

Now we'll add a new user to test this out

sudo adduser trinity

Let's explore her home directory. The "-a" option makes ls display all files, including some that are hidden from the default output of the command. Files that have a name beginning with a ., like .bashrc are not shown if we don't add the "-a" option.

sudo ls -a /home/trinity/

we'll see all files were copied from the /etc/skel/ directory, including our README file. Trinity will notice this when exploring her home directory. And with a command like cat README she'll find the information that we intended to reach all our new users.

In a previous lesson, we talked about environment variables. We set them globally, for every user on the system. But what if we want to add a special variable just for our new user, Trinity? We could edit her .bashrc file. Her Bash login shell will execute instructions in this file every time she logs in.

sudo vim /home/trinity/.bashrc

We can add new instructions at the end of this or edit the existing ones. Here's a typical scenario, we can go to this line:

PATH="$HOME/.local/bin:$HOME/bin:$PATH"

and add this content:

PATH="$HOME/.local/bin:$HOME/bin:/opt/bin:$PATH"

Make sure that your new entry is preceded by : and followed by : as that is the separator for each new entry. Also make sure that you add your entry before $PATH. Now, we can save our file and exit.

So, what does this do? Every time we type a command like

ls

Bash will look for this "ls" program in the locations it sees in the $PATH variable.

If we'd list the contents of this variable

echo $PATH

we'd see locations like these:

/home/trinity/.local/bin:/home/trinity/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin

If we add some extra applications in /opt/bin we can add this PATH variable for all users or a specific user, to make sure they can use the programs in there without typing the full path. For example, if we'd have an app at /opt/bin/specialtool, they can simply type

specialtool

to use it, instead of

/opt/bin/specialtool

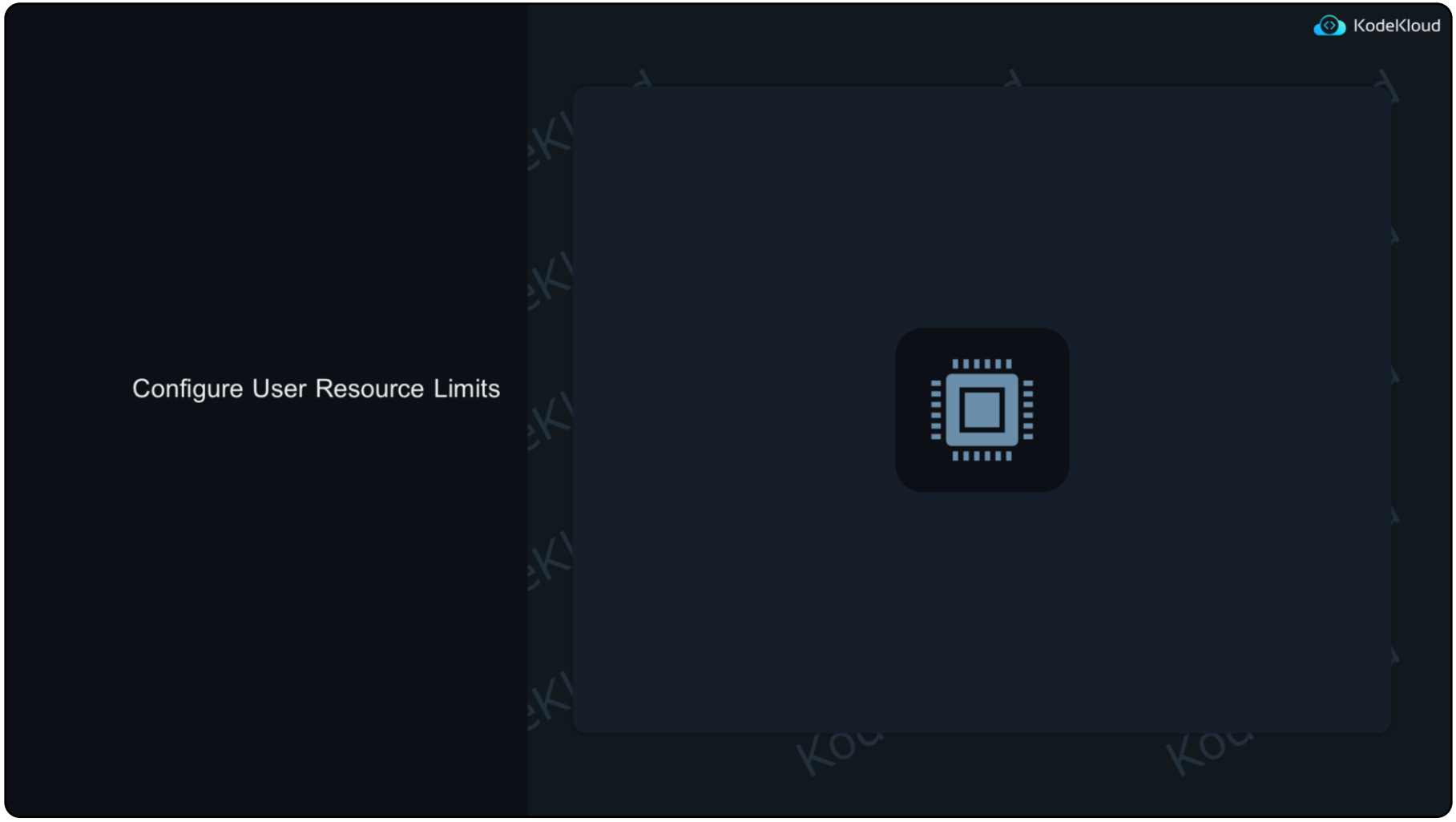
If we want to apply this change to all future users we create, we can edit the .bashrc file from /etc/skel/ instead of the personal file in a user's home directory.

sudo vim /etc/skel/.bashrc

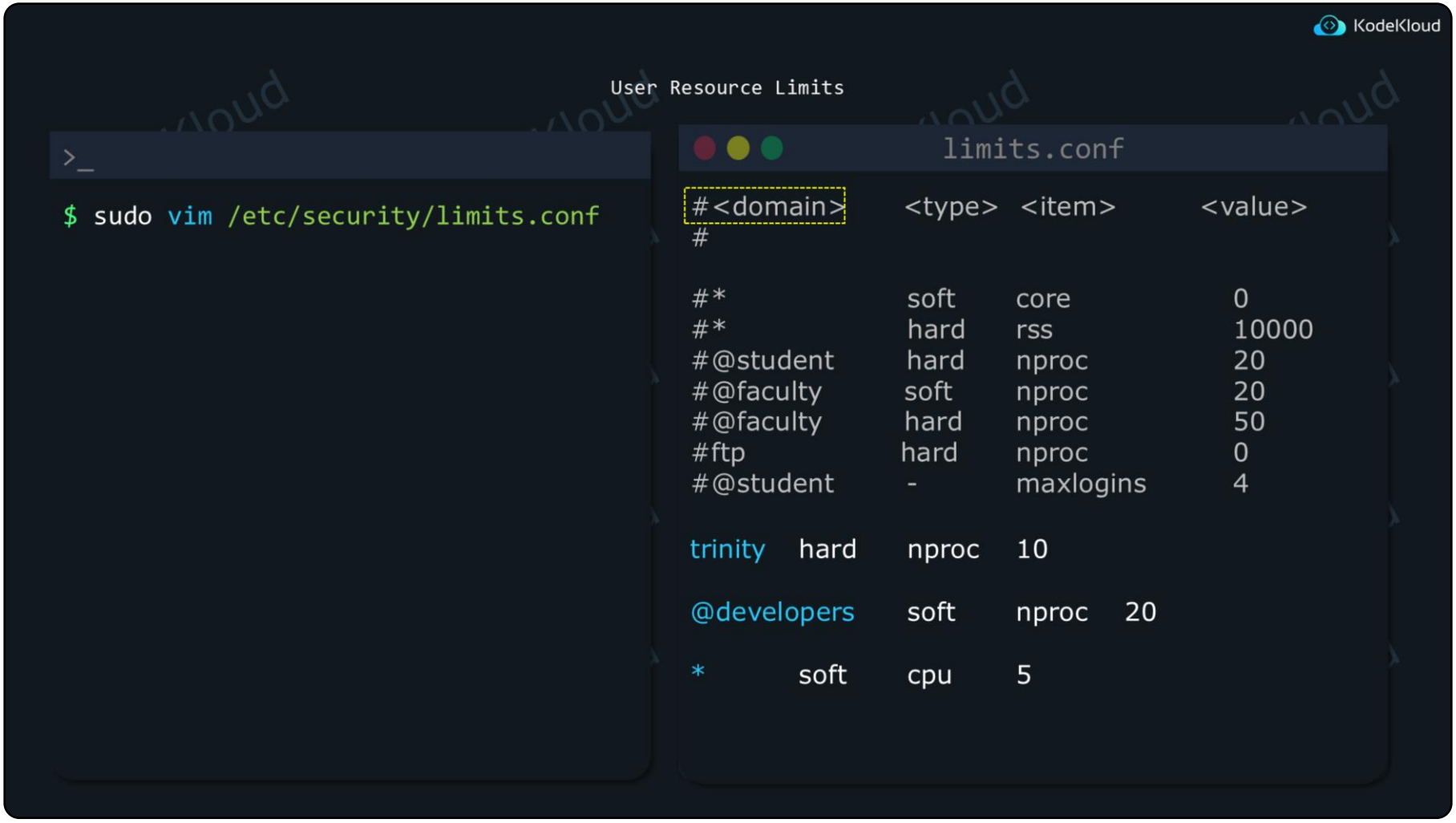
Now every time a new user is added to the system, the .bashrc file from /etc/skel/ will be copied to the new account. Effectively applying the new .bashrc instructions to all extra users we add from now on.

KodeKloud

Visit www.kodekloud.com to discover more.



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, for example, that a user 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'll see a lot of comments in that file explaining how we should use it, the syntax required, and so on.

If we move further down in that file, we'll see something like this:

We can observe 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. A 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 limit for the "developers" group.

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. Note however that this limit will only apply to every user that is not mentioned in this list. A user-specific limit overrides a global * 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, the default one. But for trinity, it

will be set to 20.

Example default limit set with *:

```
*               soft    cpu             5
```

Next is the type field which can take three different values:

hard

soft

-

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
```

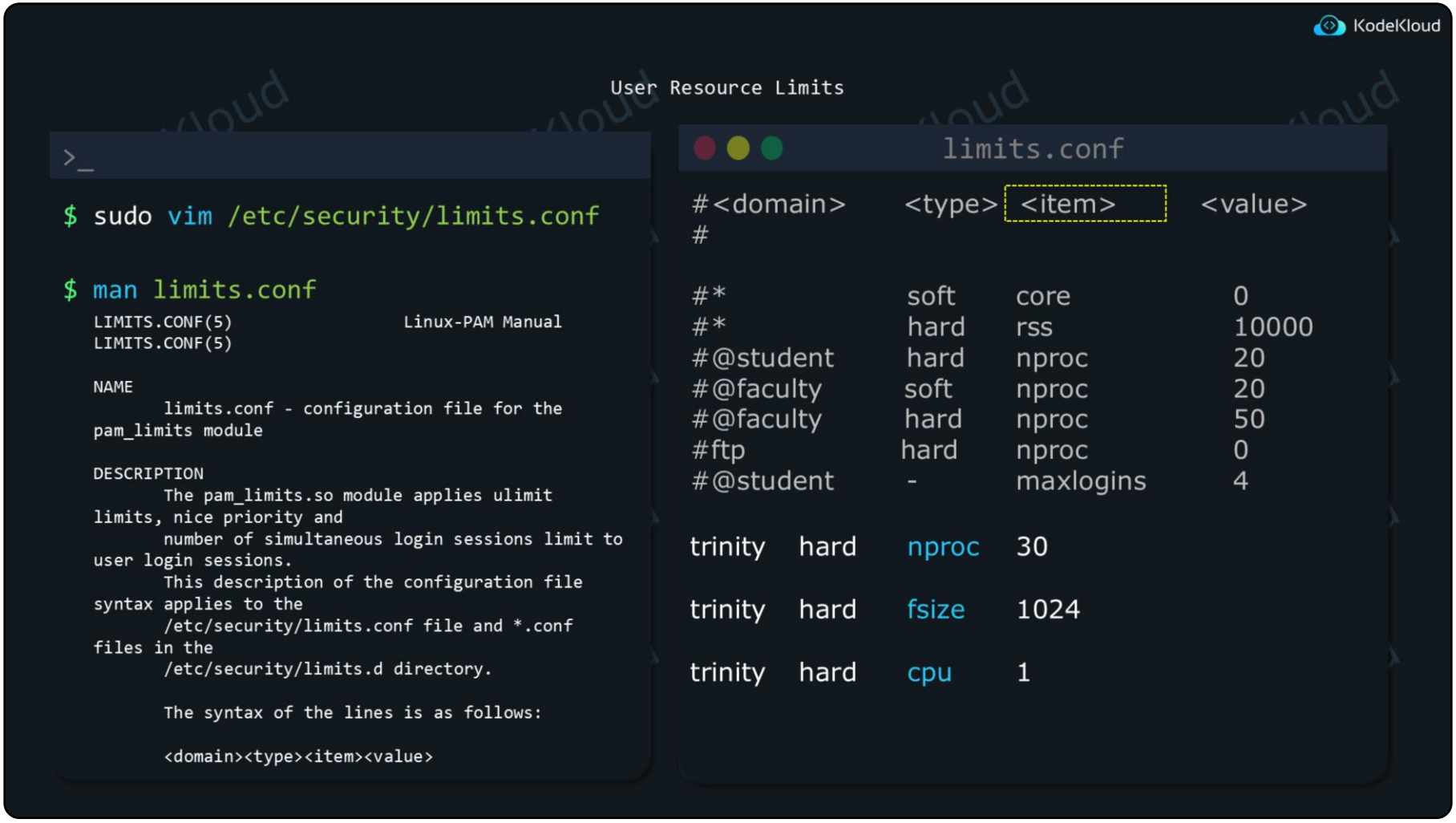
A soft limit on the other hand is different. Instead of an absolute maximum value, this is more like the "startup limit", the initial value for the limit when the user logs in. If a user has a soft limit of 10 max processes and a hard limit of 20, the following happens. When they log in, the limit will 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 need, but never above the hard limit. It basically gives them a bit of extra breathing room when they need it.

```
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. Because that's the time it actively utilized the CPU. The rest of the time, it was just sitting idle, basically sleeping, and not using the CPU.

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

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

We'll 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.

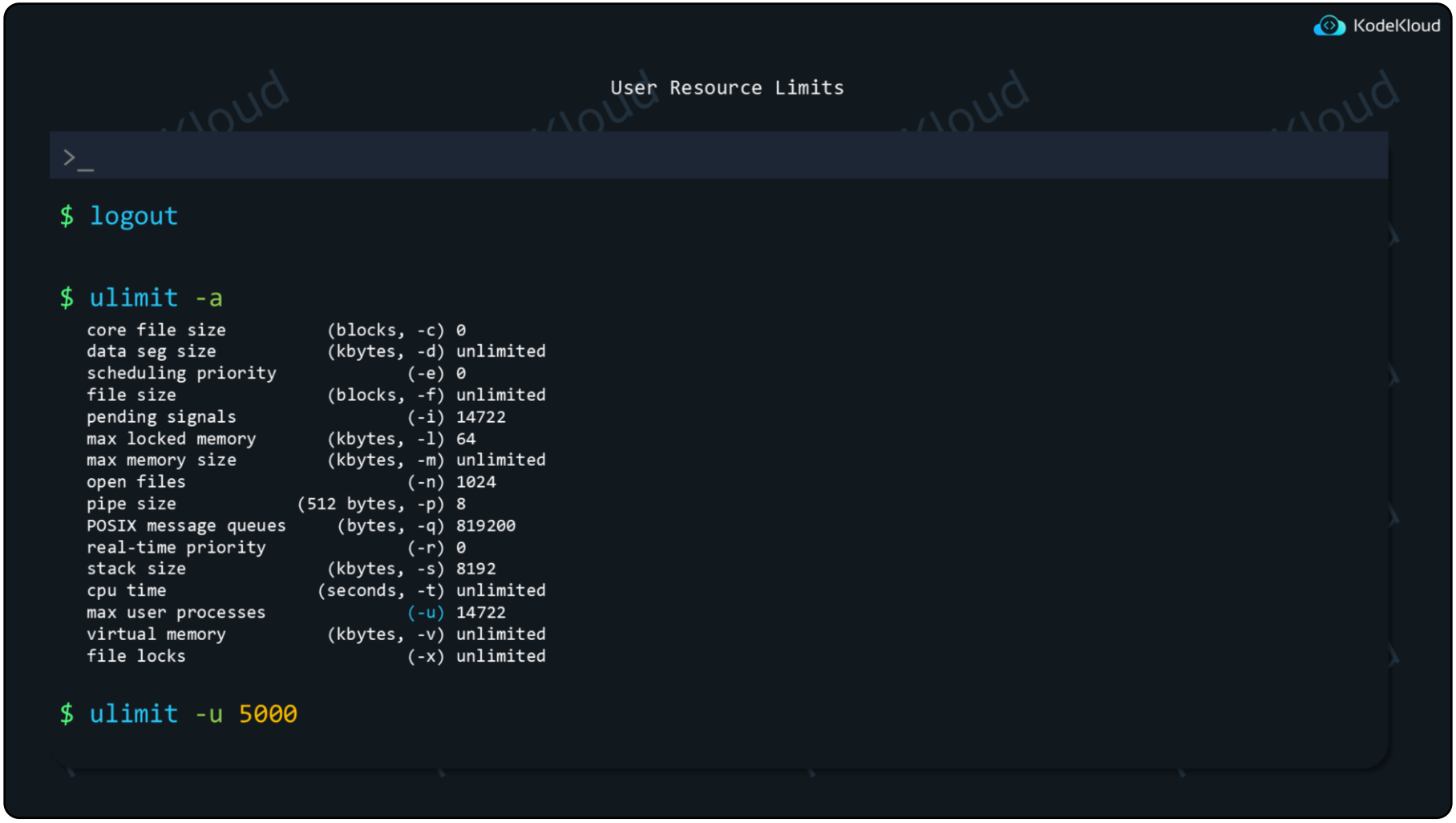
ps | less

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 command:

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. So the 3 process limit is enforced correctly.



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

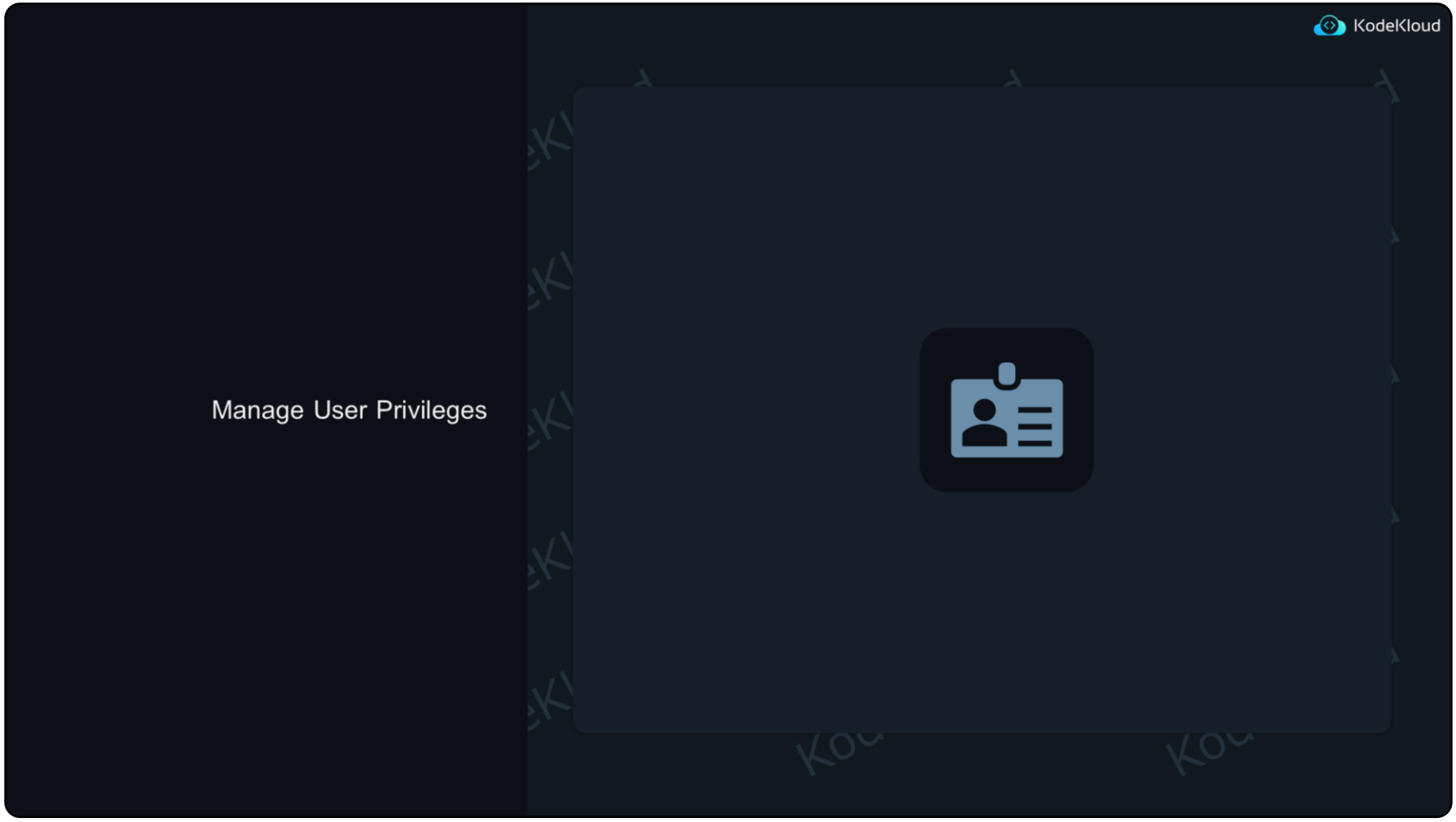
We have small hints between parentheses. For example, we can see "-u" displayed for max user processes. This means that we could type

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.

KodeKloud

Visit www.kodekloud.com to discover more.



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 "sudo" 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 "sudo" group. To add our user "trinity" to the "sudo" group:

sudo gpasswd -a trinity sudo

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 sudo group, to make sure she can't use sudo anymore, and instead, define a different sudo policy for her, later.

sudo gpasswd -d trinity sudo

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 sudo to run all commands
%sudo   ALL=(ALL)       ALL
```

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

Let's break down this line into 5 different parts and analyze what they do:

1.%sudo 2.ALL=3.(ALL:4.ALL) 5.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_user 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. And, by default, it runs it as the "root" user. But sudo can also be used so that "aaron" can run commands as some other regular user, like "jane". 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". In these fields, if we want to specify multiple values, we just list them one by one, separated by commas.

4. is the run_as_group field. Same as before. If we type ALL here, then users that belong to the "sudo" group can use sudo to run commands as any group. If we type a group name, then they can only run commands under the group listed here.

5. 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:run_as_group) command_list

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

trinity ALL=(ALL) ALL

We can see, in this case, between parantheses we just specified that jane can run commands as ALL users. But we did not specify any groups. This will mean, implicitly, that she can run commands as any group too, even if we didn't add the second ALL field here.

To specify a policy for all users in the developers group, we just add the % sign in front of the group name:

%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 this command 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=(aaron, john) ALL

Also, this is wrapped in ( ) parentheses which hints 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 kodekloud.

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:

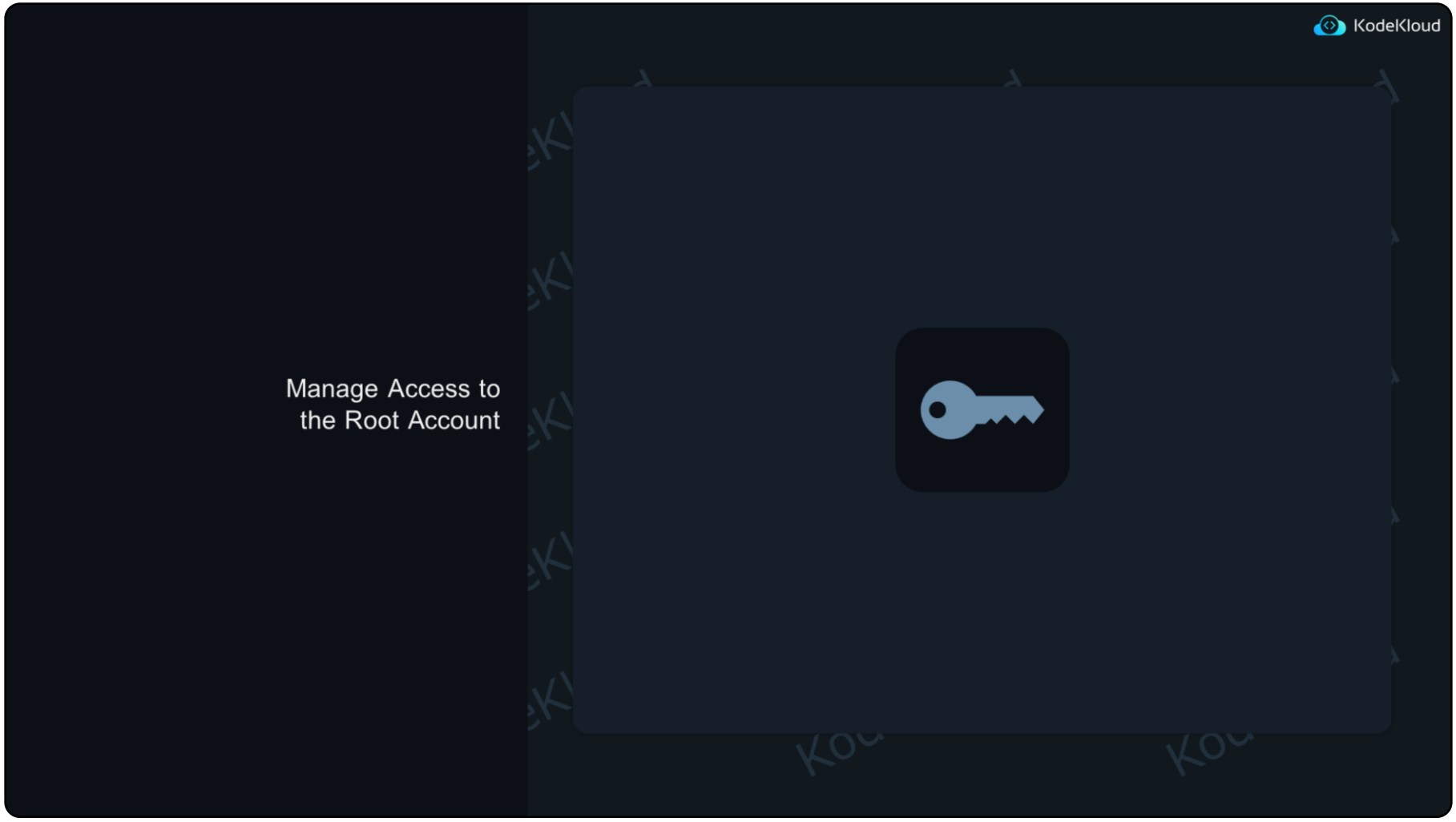
%sudo ALL=(ALL) NOPASSWD: ALL

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

KodeKloud

Visit www.kodekloud.com to discover more.



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

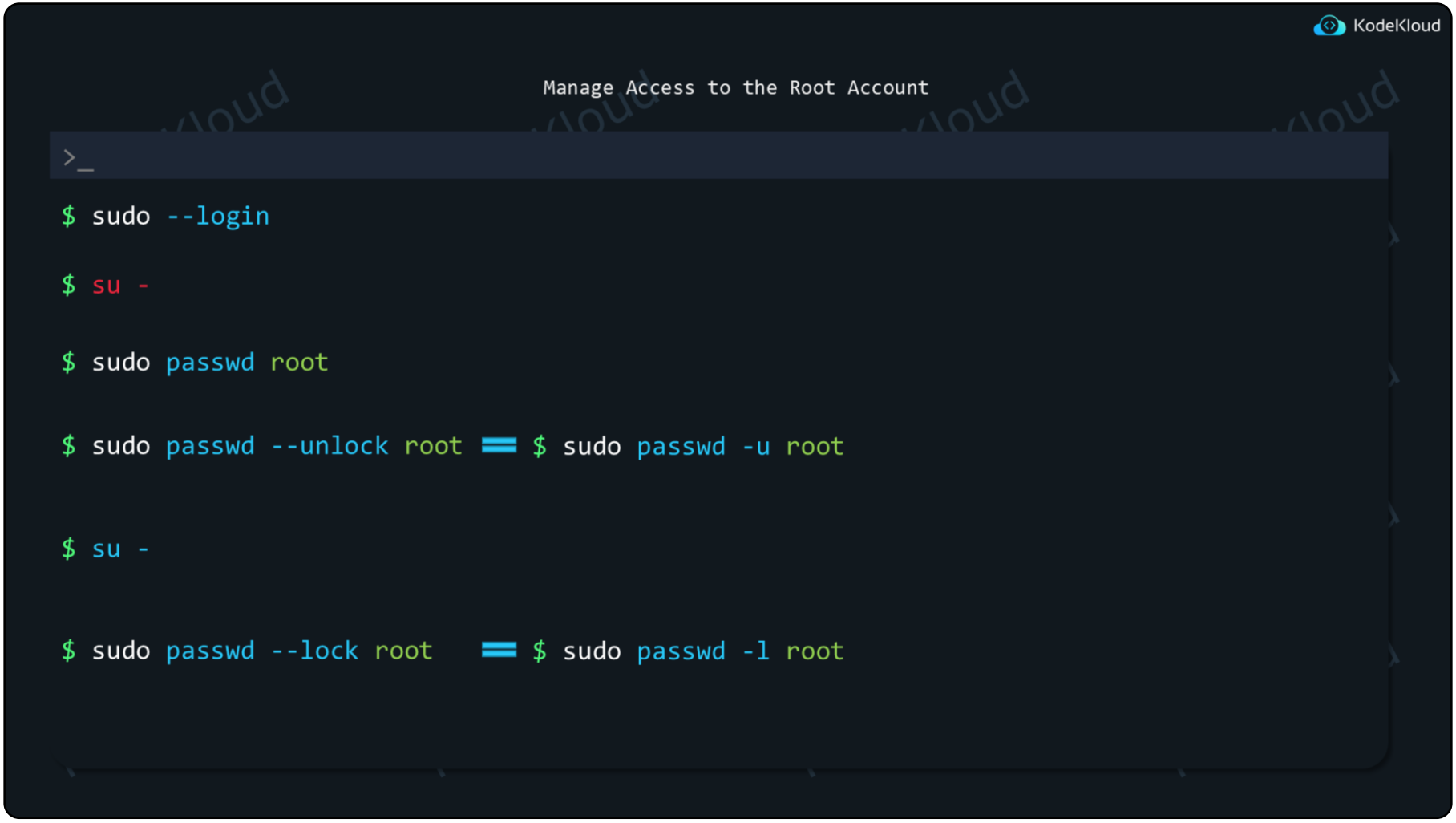
And that's it, we're logged in as root. To exit from root's session, we'll type:

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. But instead of typing our current user's password, as is the case for "sudo", we have to type the password for the "root" user instead, when we use "su".



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. Because the password for our current user is not locked. 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. Let's say that, currently, people can log in as "root". But we decide that this is a bit insecure for our purposes. We can lock password-based logins to the root account with:

sudo passwd --lock root
sudo passwd -l root

Other login methods 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. Since that method does not use a password, but a cryptographic key instead.

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

KodeKloud

Visit www.kodekloud.com to discover more.