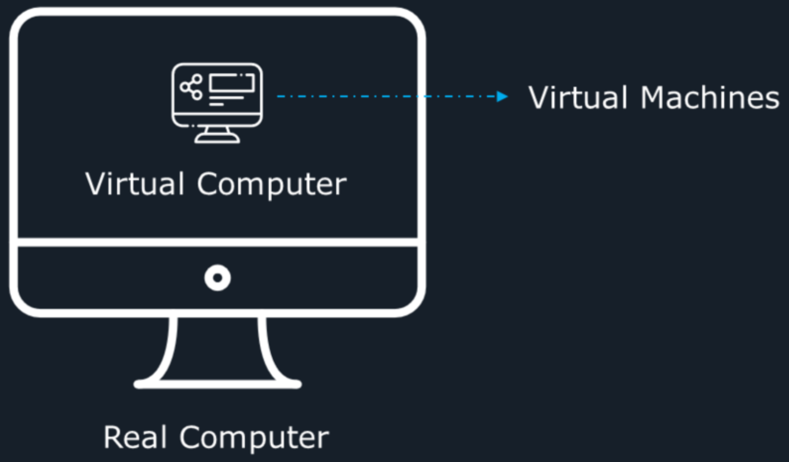
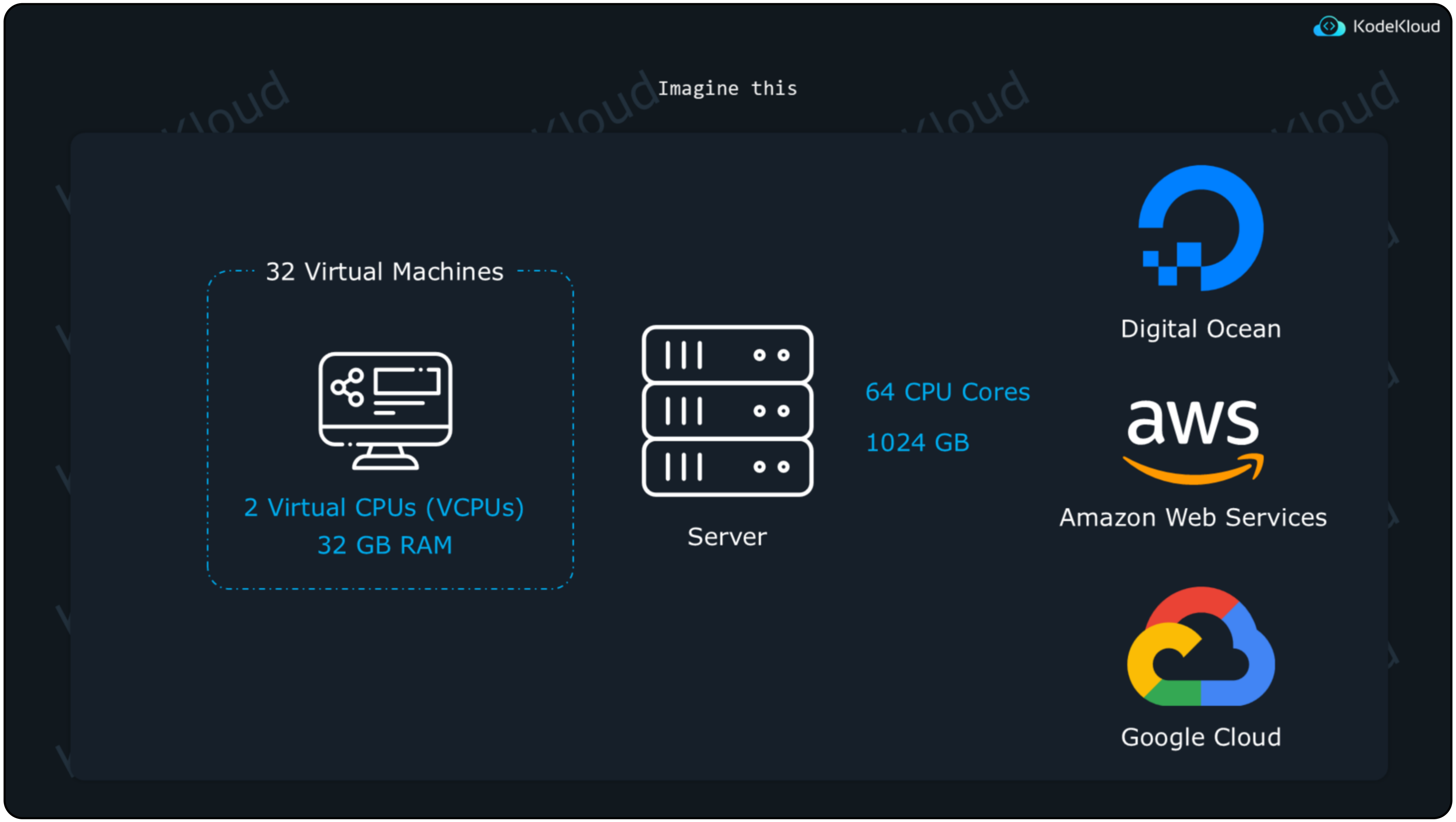
KodeKloud

Visit www.kodekloud.com to discover more.

Virtual Computer





KodeKloud

Imagine this

KodeKloud

Imagine this

Linux
(QEMU-KVM)

Quick Emulator ← → Kernel-based Virtual Machine

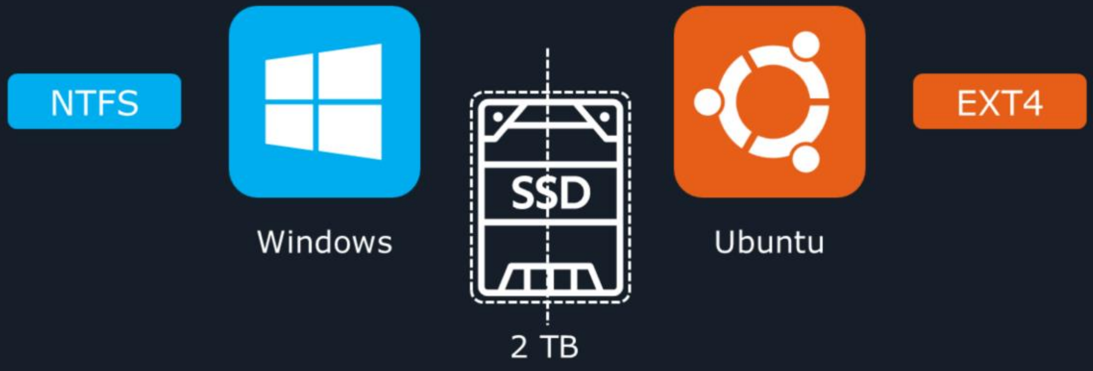
KodeKloud

VIRSH

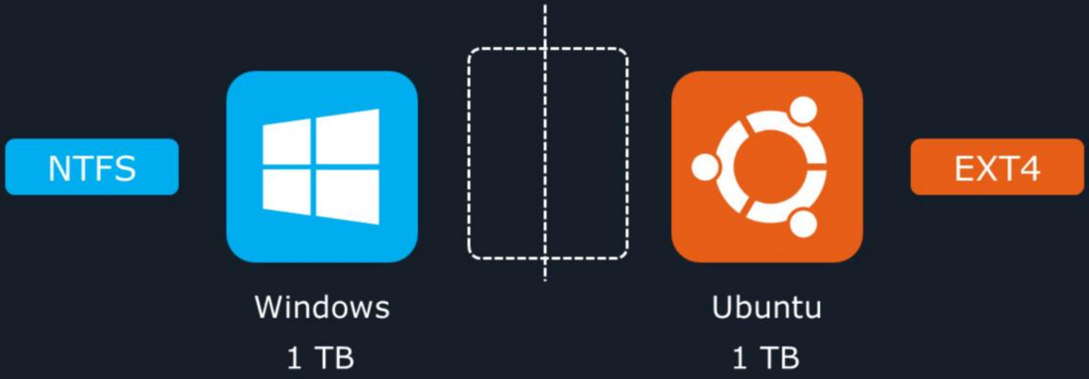
VIRSH

Manage Virtual Machines
from the Command Line

Imagine this



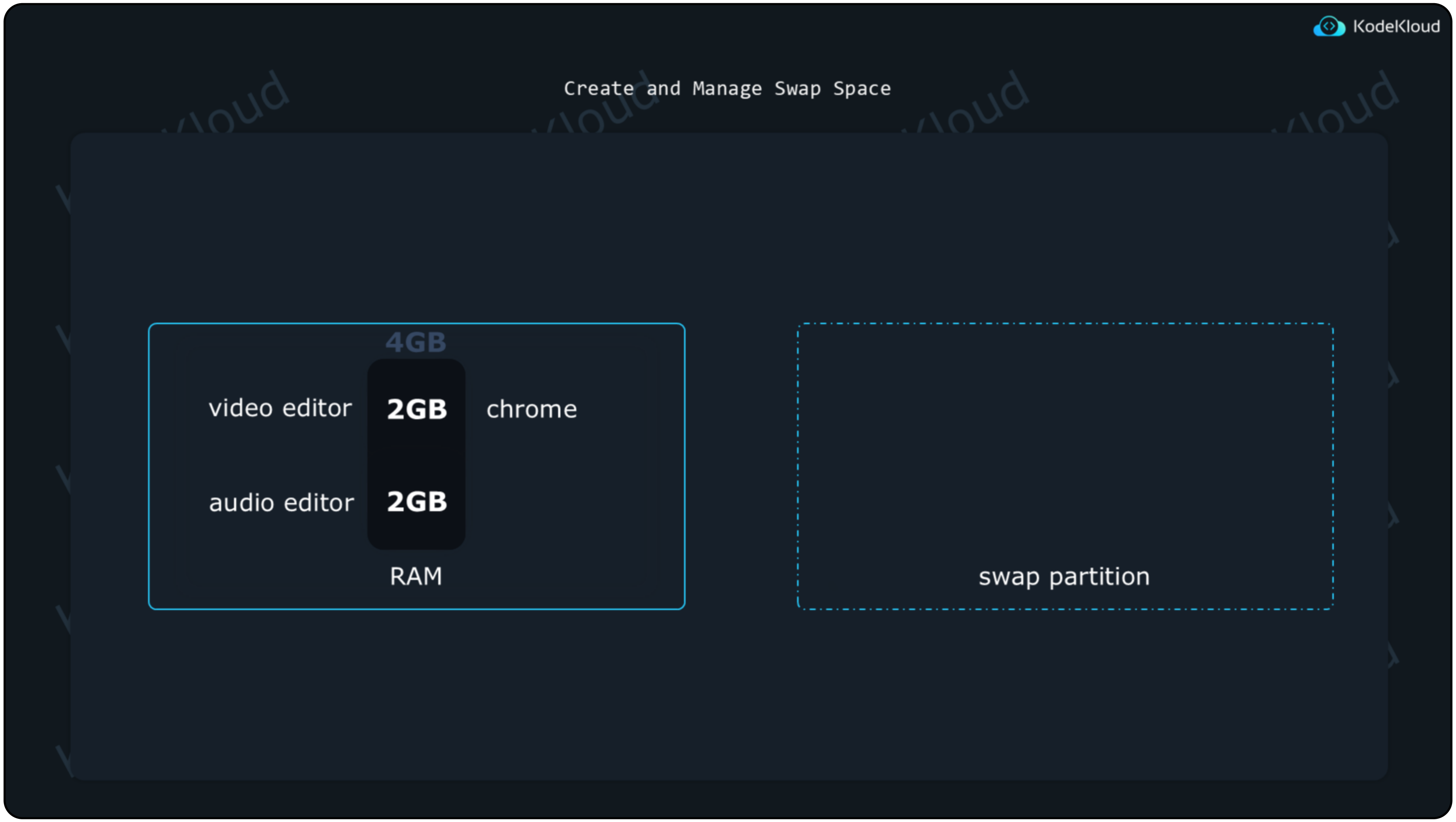
Imagine this



"Partitioning"

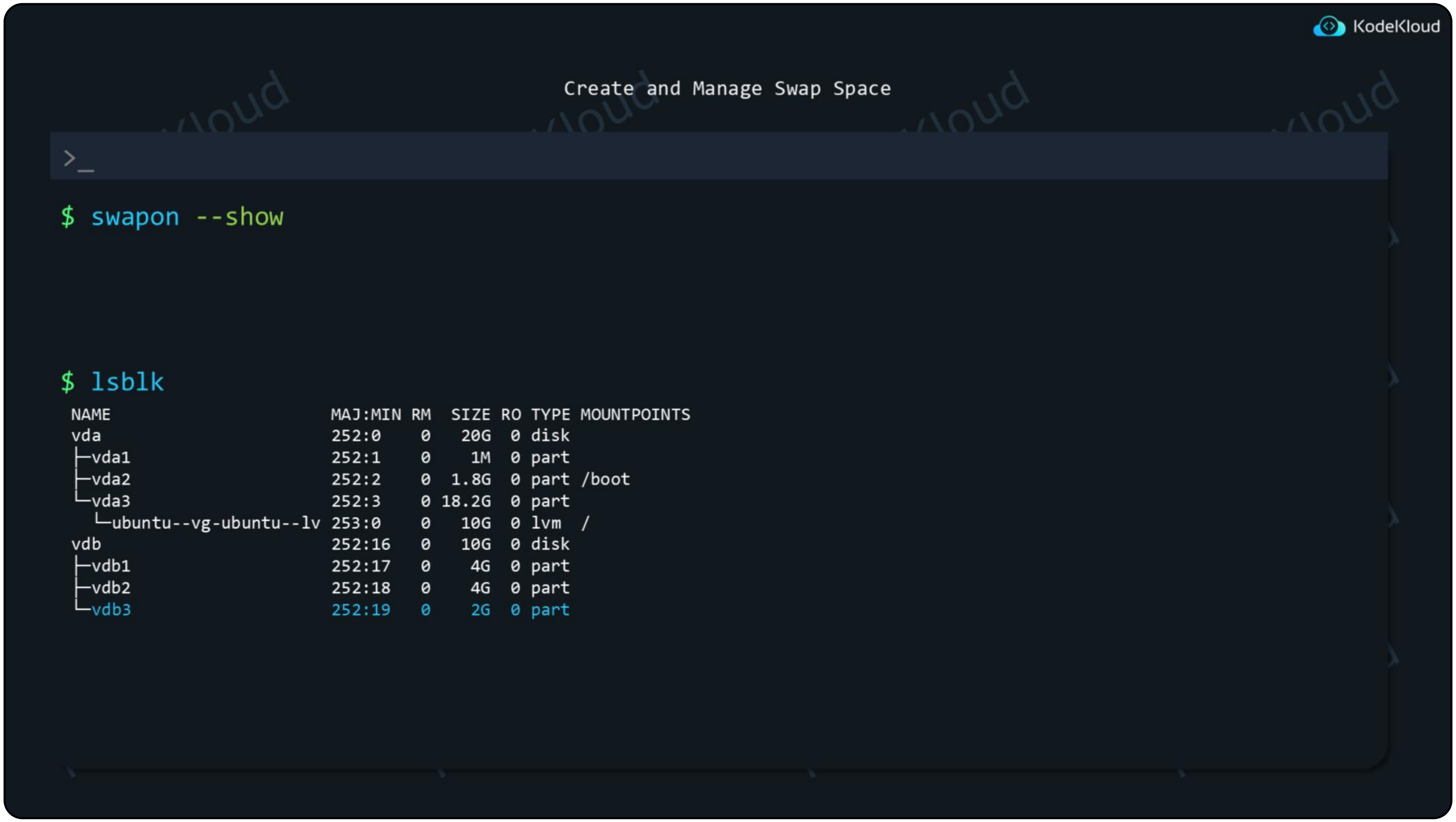
Configure and Manage Swap Space

Now, let's look at how to configure and manage swap space in Linux.



In this lesson we'll discuss how we can create a so-called swap partition. Swap is an area where Linux can temporarily move some data from the computer's Random Access Memory (RAM). To understand this mechanism let's go through an overly simplified scenario.

Let's imagine we have a computer with 4GB of RAM. We open a video editor and this uses 2GB of RAM. We open an audio editor and this needs another 2GB. Now we have no more free memory. But we have a 2GB swap partition. Although no more RAM is available, we'll see we can still open Chrome. How is this possible? When we want to open Chrome the following happens: Linux sees no more RAM is available. But it also sees that we didn't use the video editor in the last 1 hour. It's basically just sleeping there, inactive. So, it decides that it can move the data in memory used by the video editor, to our swap partition. By moving data from RAM to swap, it freed up 2GB of our RAM. So, Chrome can now use that free memory.



Create and Manage Swap Space

```
$ swapon --show




$ lsblk
NAME                        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
vda                         252:0    0   20G  0 disk
├─vda1                      252:1    0    1M  0 part
├─vda2                      252:2    0  1.8G  0 part /boot
└─vda3                      252:3    0 18.2G  0 part
  └─ubuntu--vg-ubuntu--lv 253:0    0   10G  0 lvm  /
vdb                         252:16   0   10G  0 disk
├─vdb1                      252:17   0    4G  0 part
├─vdb2                      252:18   0    4G  0 part
└─vdb3                      252:19   0    2G  0 part
```

Now let's see some commands. To check if the system uses any kind of swap areas, we can use this command:

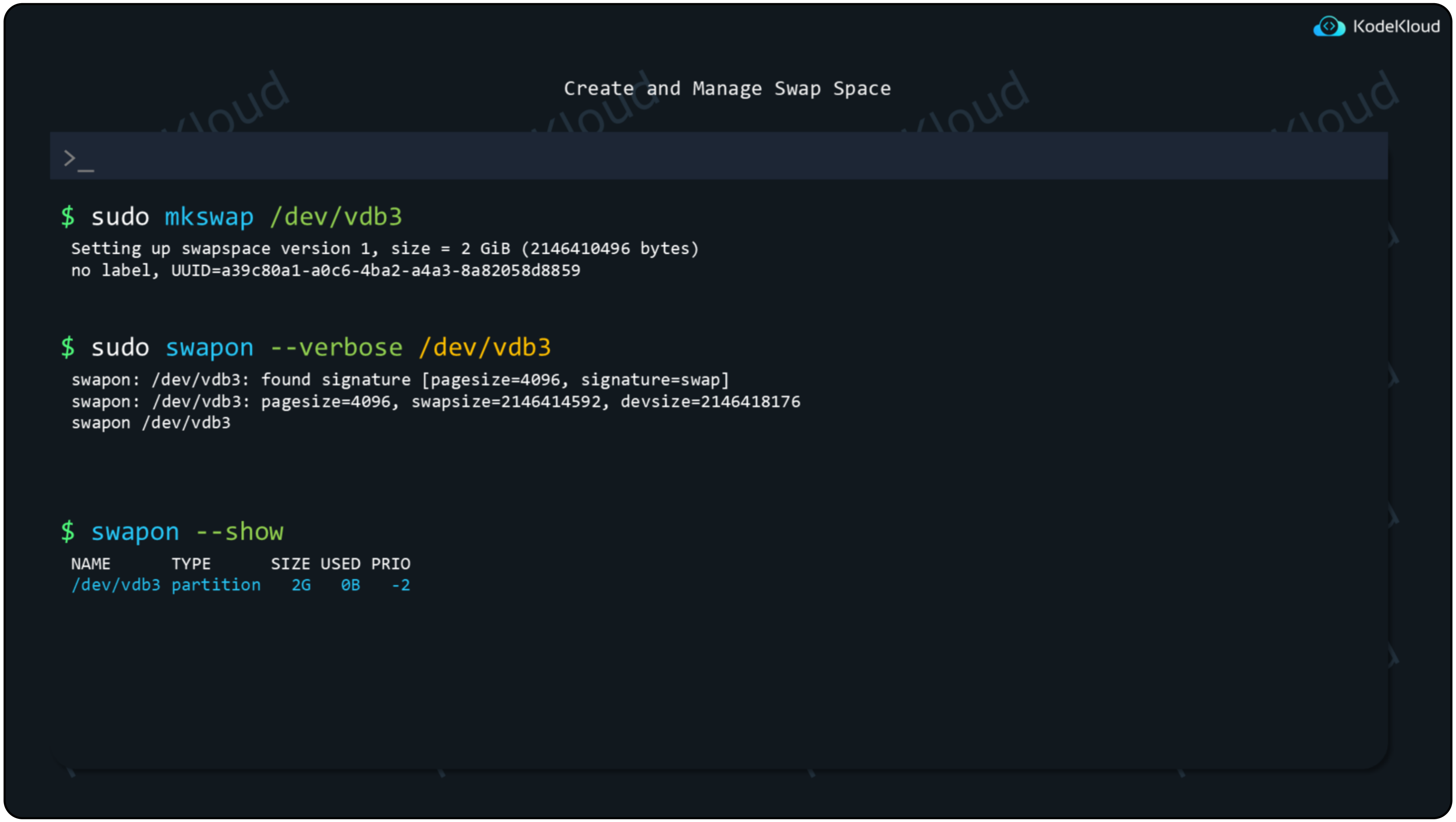
swapon --show

Since we don't have swap set up yet, the command will output nothing at this point. So let's set up a partition to be used as swap.

First, we take a look at what partitions we have available:

```
lsblk
```

In the previous lesson, we created the partition at vdb3 specifically to be used as swap. This is currently empty, with no data inside. Before it can be used as swap it has to be prepared. If you ever formatted an USB stick with a filesystem like FAT32 or NTFS, this is a similar process. It basically writes some small header data on the partition, and labels it, so that the system knows this is meant to be used as a swap area.



Create and Manage Swap Space

```
$ sudo mkswap /dev/vdb3
 Setting up swapspace version 1, size = 2 GiB (2146410496 bytes)
 no label, UUID=a39c80a1-a0c6-4ba2-a4a3-8a82058d8859


$ sudo swapon --verbose /dev/vdb3
 swapon: /dev/vdb3: found signature [pagesize=4096, signature=swap]
 swapon: /dev/vdb3: pagesize=4096, swapsize=2146414592, devsize=2146418176
 swapon /dev/vdb3



$ swapon --show
 NAME      TYPE      SIZE USED PRIO
 /dev/vdb3 partition   2G   0B   -2
```

To "format" it as swap, we'll use this command:

sudo mkswap /dev/vdb3

Now we can tell Linux: "Use this partition as swap:"

```
sudo swapon --verbose /dev/vdb3
```

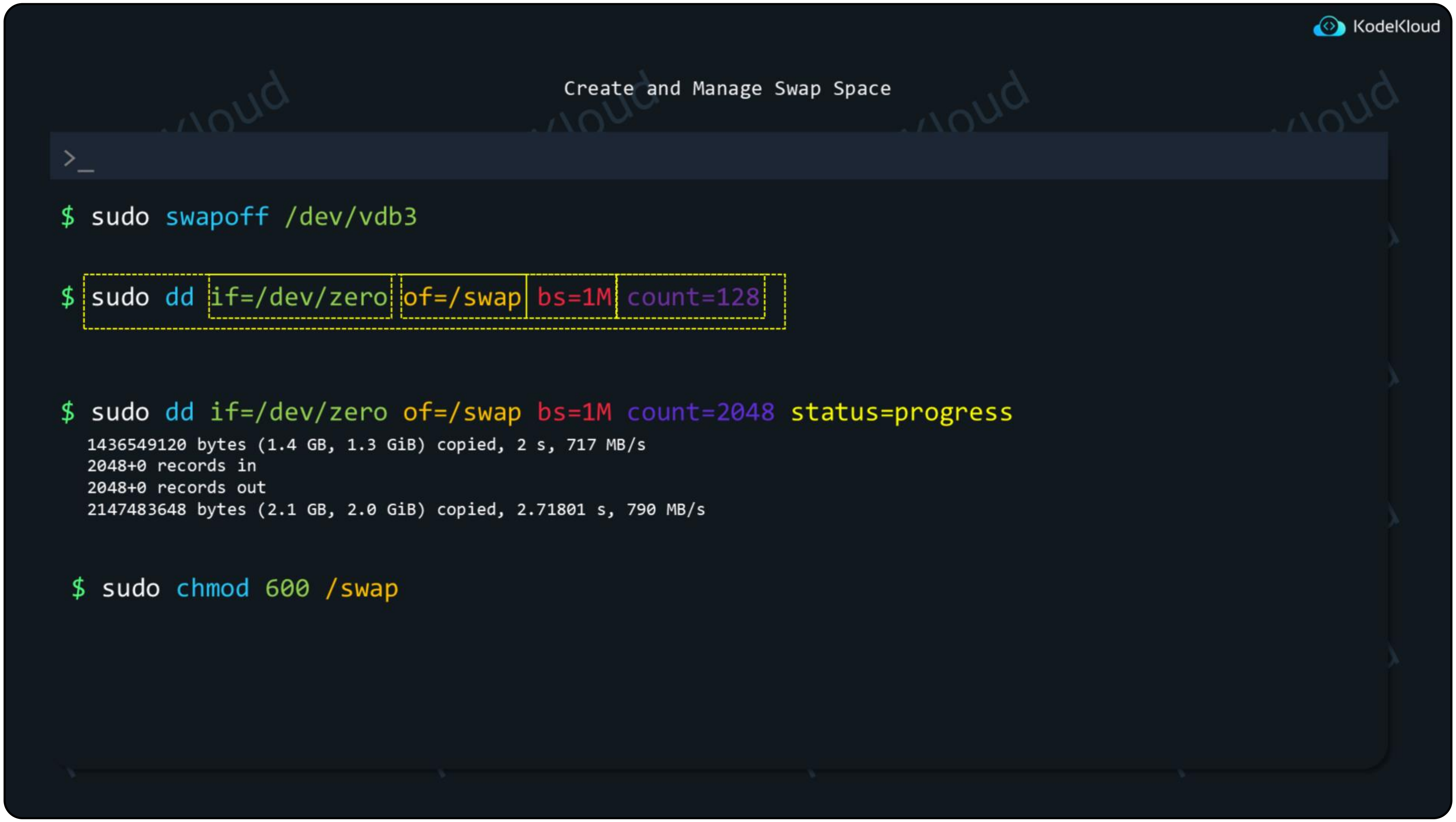
The verbose option makes the command output more details about what it's doing. An equivalent command is:

```
sudo swapon -v /dev/vdb3
```

Seems everything worked perfectly. We can check with:

```
swapon –show
```

once again. And this time the command will list our swap partition here.



Create and Manage Swap Space

```
$ sudo swapoff /dev/vdb3

$ sudo dd if=/dev/zero of=/swap bs=1M count=128

$ sudo dd if=/dev/zero of=/swap bs=1M count=2048 status=progress
  1436549120 bytes (1.4 GB, 1.3 GiB) copied, 2 s, 717 MB/s
  2048+0 records in
  2048+0 records out
  2147483648 bytes (2.1 GB, 2.0 GiB) copied, 2.71801 s, 790 MB/s

$ sudo chmod 600 /swap
```

However, there's still a problem. If we reboot this system, /dev/vdb3 won't be used as swap anymore. Because the change we made is temporary. But we'll see how we can tell Linux to automatically use this as swap, every time it boots up, in another lesson where we learn how to mount partitions.

To stop using our partition as swap space, we can enter this command:

```
sudo swapoff /dev/vdb3
```

Instead of partitions, we can also use simple files as swap. In fact, if you install Ubuntu on a personal computer, that's what it will do by default nowadays. Set up swap on a file.

To do this manually, first, we'd need to create an empty file and fill it with zeroes. That is, binary zeroes, because in binary we can have either a 0 or a 1. We'll use a utility called dd and we'll have these parameters in the command:

```
if=/dev/zero
```

This tells it to use the input file at /dev/zero. It's a special device file that generates an infinite number of zeroes when an application reads from it.

After we read from that file, we need to tell dd where to write the output. Basically, it will copy the input file to the output file. So we'll tell it to write the contents in a file we'll store at this path: /swap.

```
of=/swap
```

Then we specify the block size to be 1MB:

```
bs=1M
```

This, combined with the next parameter:

```
count=128
```

tells dd to write a 1 Megabyte block 128 times. So, we'll get 128 megabytes in total.

Our final command will look like this:

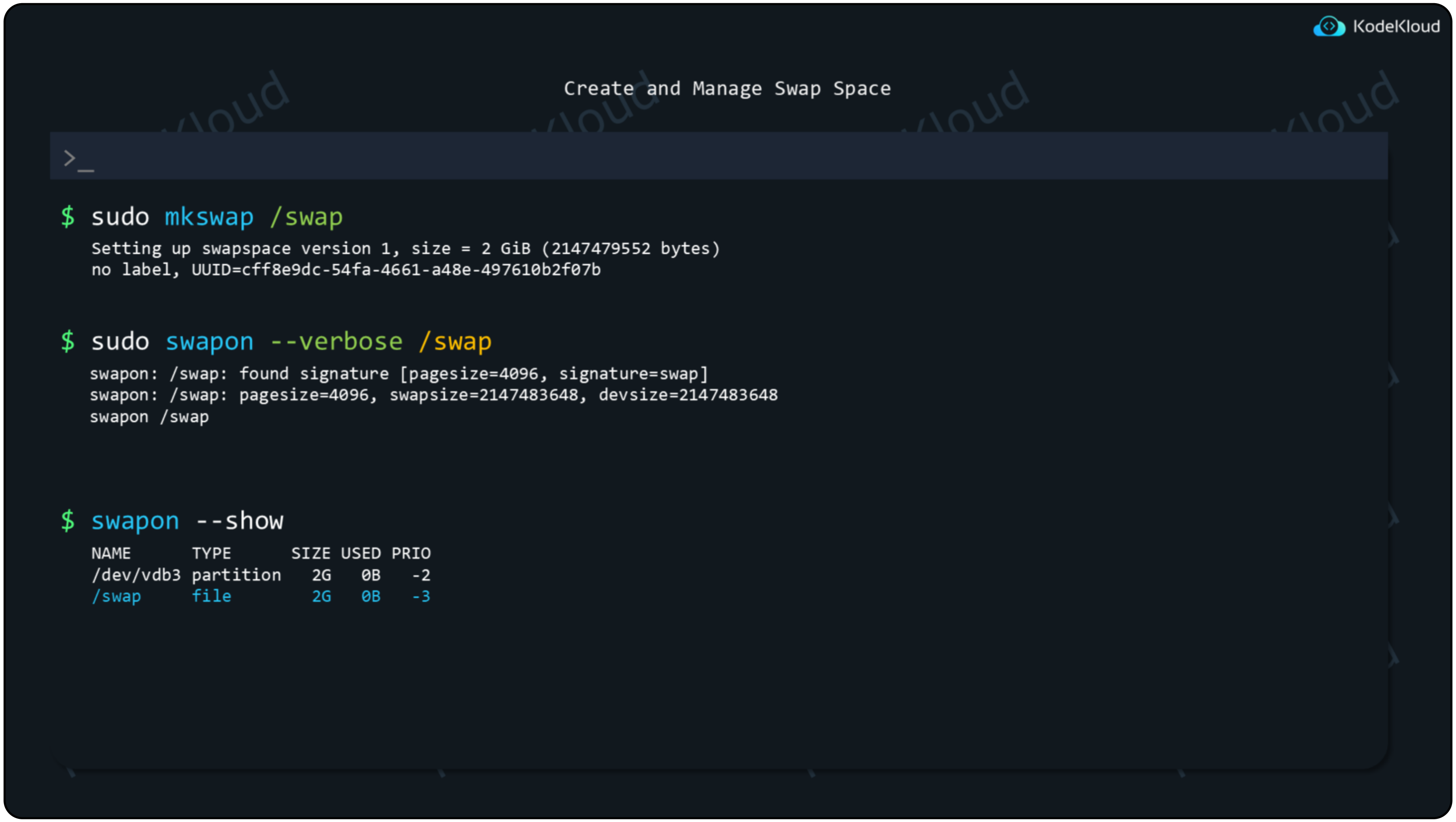
```
sudo dd if=/dev/zero of=/swap bs=1M count=128
```

This file is small so it will get written fast. But if we have a large file, we might have to wait a while. And dd won't show us how much % it has written. But we can add another parameter, status=progress. This will make it display the progress it's making while writing to that file. Here's an example command writing 2GB of data:

```
sudo dd if=/dev/zero of=/swap bs=1M count=2048 status=progress
```

Regular users should not be allowed to read this swap file. That's because this potentially gives them access to the memory contents of programs other users might be using. So, we'll set permissions to only allow the root user to read and write to this file:

```
sudo chmod 600 /swap
```



Create and Manage Swap Space

```
$ sudo mkswap /swap
  Setting up swapspace version 1, size = 2 GiB (2147479552 bytes)
  no label, UUID=cff8e9dc-54fa-4661-a48e-497610b2f07b

$ sudo swapon --verbose /swap
  swapon: /swap: found signature [pagesize=4096, signature=swap]
  swapon: /swap: pagesize=4096, swapsize=2147483648, devsize=2147483648
  swapon /swap

$ swapon --show
  NAME      TYPE      SIZE USED PRIO
  /dev/vdb3 partition   2G   0B   -2
  /swap     file        2G   0B   -3
```

Formatting this as swap is the same as before. But instead of using a file such as /dev/vdb3 that points to a partition, we just use a regular file in our command:

sudo mkswap /swap

And now we can use this as swap on our system:

```
sudo swapon --verbose /swap
```

We can check active swap areas again:

```
swapon --show
```

And we'll see it's indeed used as swap.

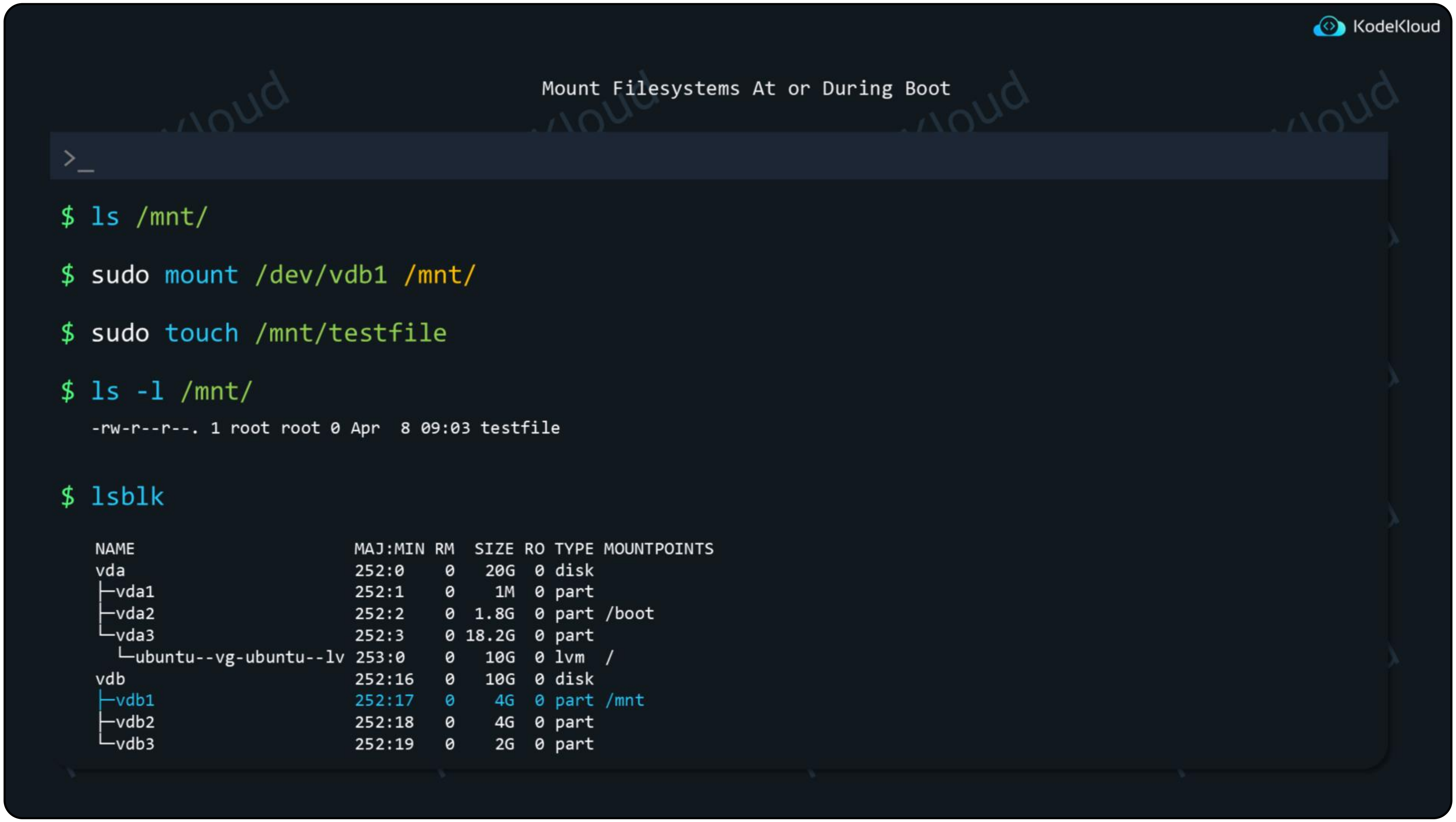
Note how we can use multiple things as swap, at the same time. In our case we use the vdb3 partition as swap, but also our file. We can also use multiple files, or multiple partitions if we want to.

KodeKloud

Visit www.kodekloud.com to discover more.

Configure Systems to Mount Filesystems At or During Boot

Now let's explore how to mount filesystems, both manually, but also automatically, at boot time.


Mount Filesystems At or During Boot

```
$ ls /mnt/

$ sudo mount /dev/vdb1 /mnt/

$ sudo touch /mnt/testfile

$ ls -l /mnt/
  -rw-r--r--. 1 root root 0 Apr  8 09:03 testfile

$ lsblk

  NAME                       MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
  vda                        252:0    0   20G  0 disk
  ├─vda1                     252:1    0    1M  0 part
  ├─vda2                     252:2    0  1.8G  0 part /boot
  └─vda3                     252:3    0 18.2G  0 part
    └─ubuntu--vg-ubuntu--lv  253:0    0   10G  0 lvm  /
  vdb                        252:16   0   10G  0 disk
  ├─vdb1                     252:17   0    4G  0 part /mnt
  ├─vdb2                     252:18   0    4G  0 part
  └─vdb3                     252:19   0    2G  0 part
```

In a previous lesson we explored how to create filesystems. But you might have noticed, there's still no way for us to access those filesystems. How do we create files and directories on them?

To make a filesystem accessible we must first mount it. Mounting basically means attaching or plugging in a filesystem to one of our directories. Here's how it works. First, we can take a look at a directory often used for temporarily mounting a random filesystem:

```
ls /mnt/
```

This is currently empty. Now let's say we want to mount the XFS filesystem created in one of our previous lessons. This is stored on the partition at /dev/vdb1. To mount it, we use this command:

```
sudo mount /dev/vdb1 /mnt/
```

With the filesystem mounted at /mnt/, we can now easily create a file on it:

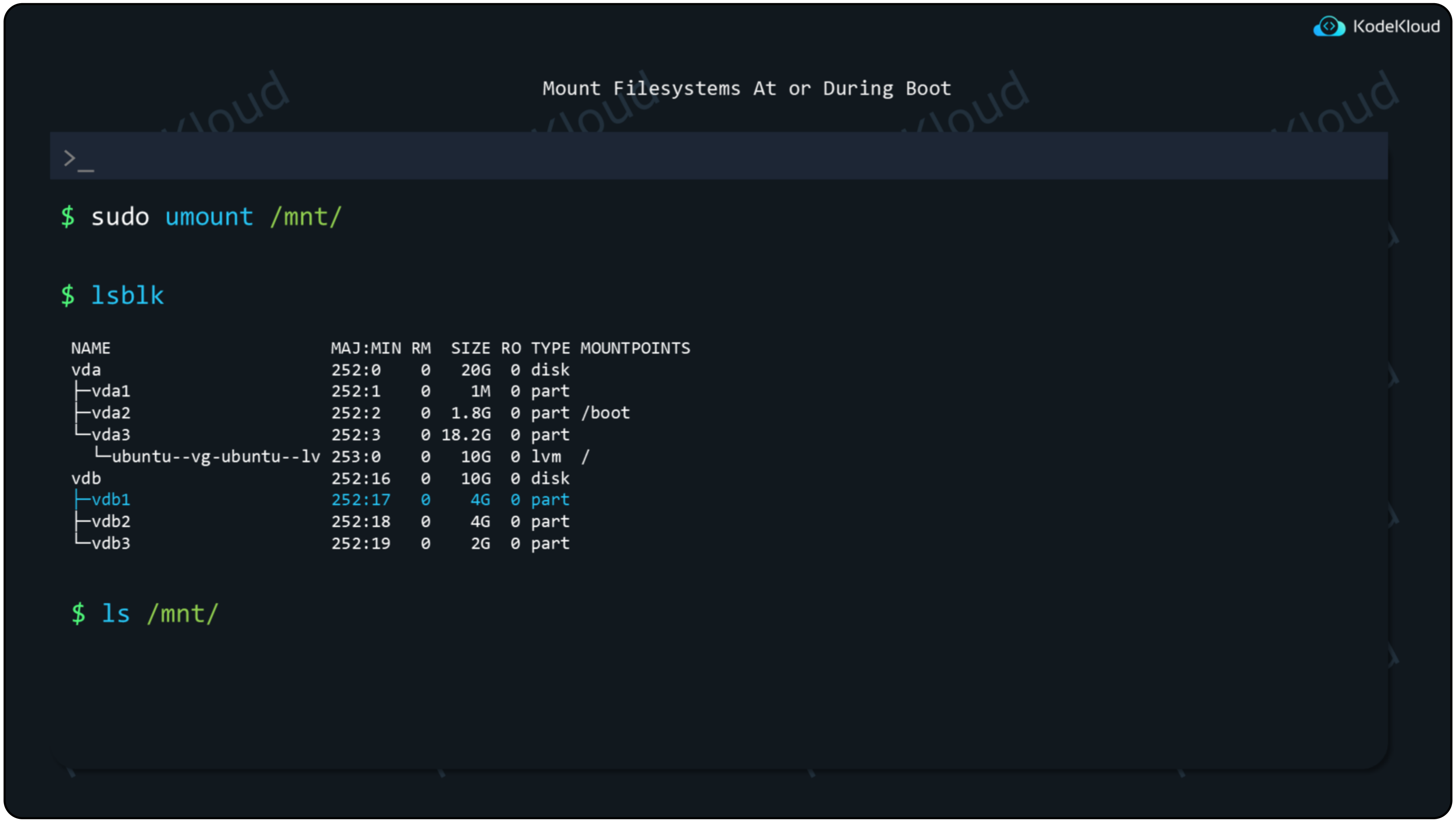
```
sudo touch /mnt/testfile
```

And we can see that our previously empty filesystem now has one file inside:

```
ls -l /mnt/
```

We can use the lsblk command to confirm that our filesystem is indeed mounted:

```
lsblk
```



Mount Filesystems At or During Boot

```
$ sudo umount /mnt/

$ lsblk

NAME                         MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
vda                          252:0    0   20G  0 disk
├─vda1                       252:1    0    1M  0 part
├─vda2                       252:2    0  1.8G  0 part /boot
└─vda3                       252:3    0 18.2G  0 part
  └─ubuntu--vg-ubuntu--lv    253:0    0   10G  0 lvm  /
vdb                          252:16   0   10G  0 disk
├─vdb1                       252:17   0    4G  0 part
├─vdb2                       252:18   0    4G  0 part
└─vdb3                       252:19   0    2G  0 part

$ ls /mnt/
```

To unmount a filesystem, we use the umount command. We'd expect it to be unmount, with an n, but it's umount, without the n letter.

To unmount, we just specify the directory where our filesystem is mounted:

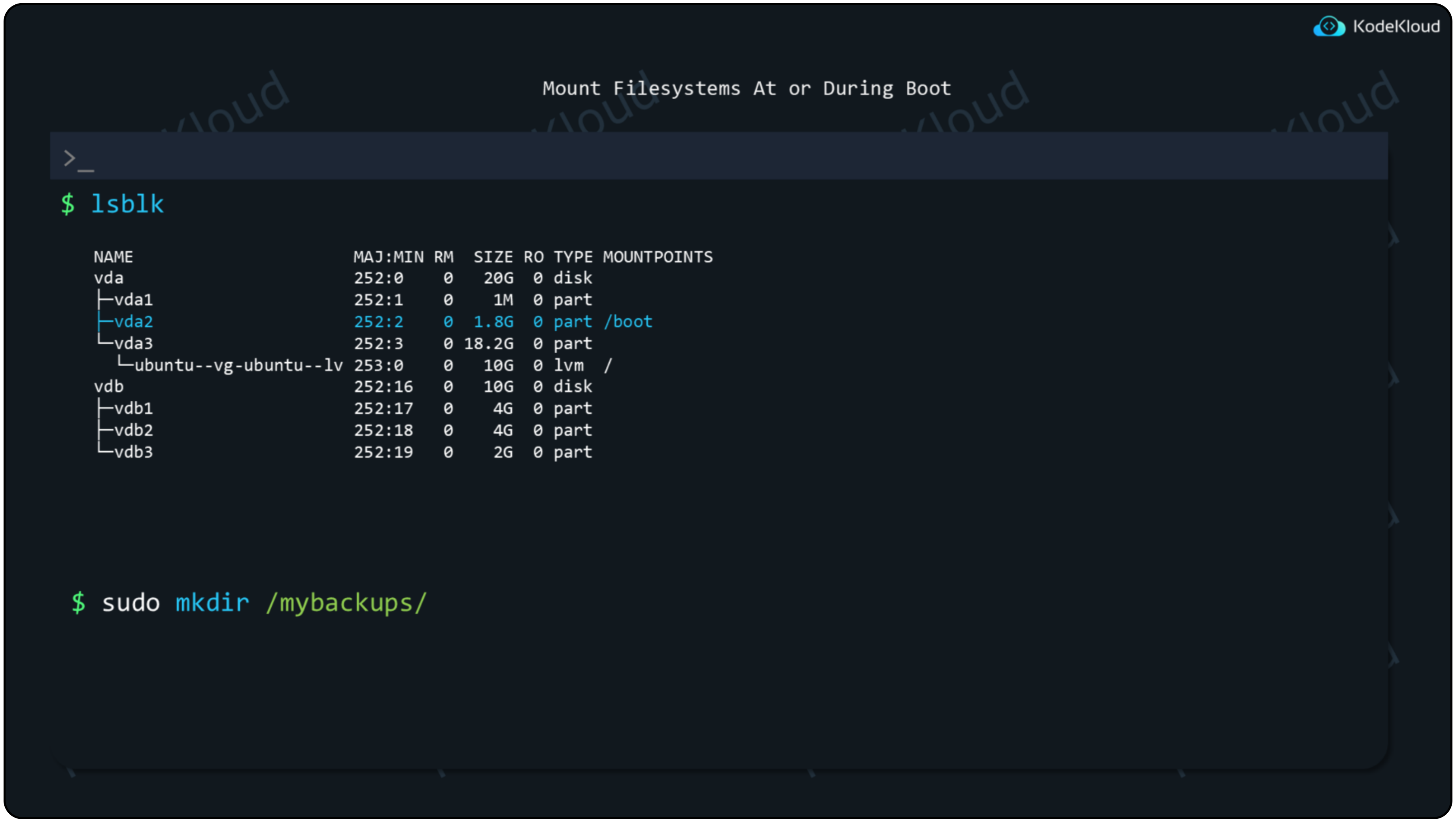
sudo umount /mnt/

Now lsblk confirms this is unmounted:

```
lsblk
```

And we can see /mnt/ is once again an empty directory, with no contents inside:

```
ls /mnt/
```



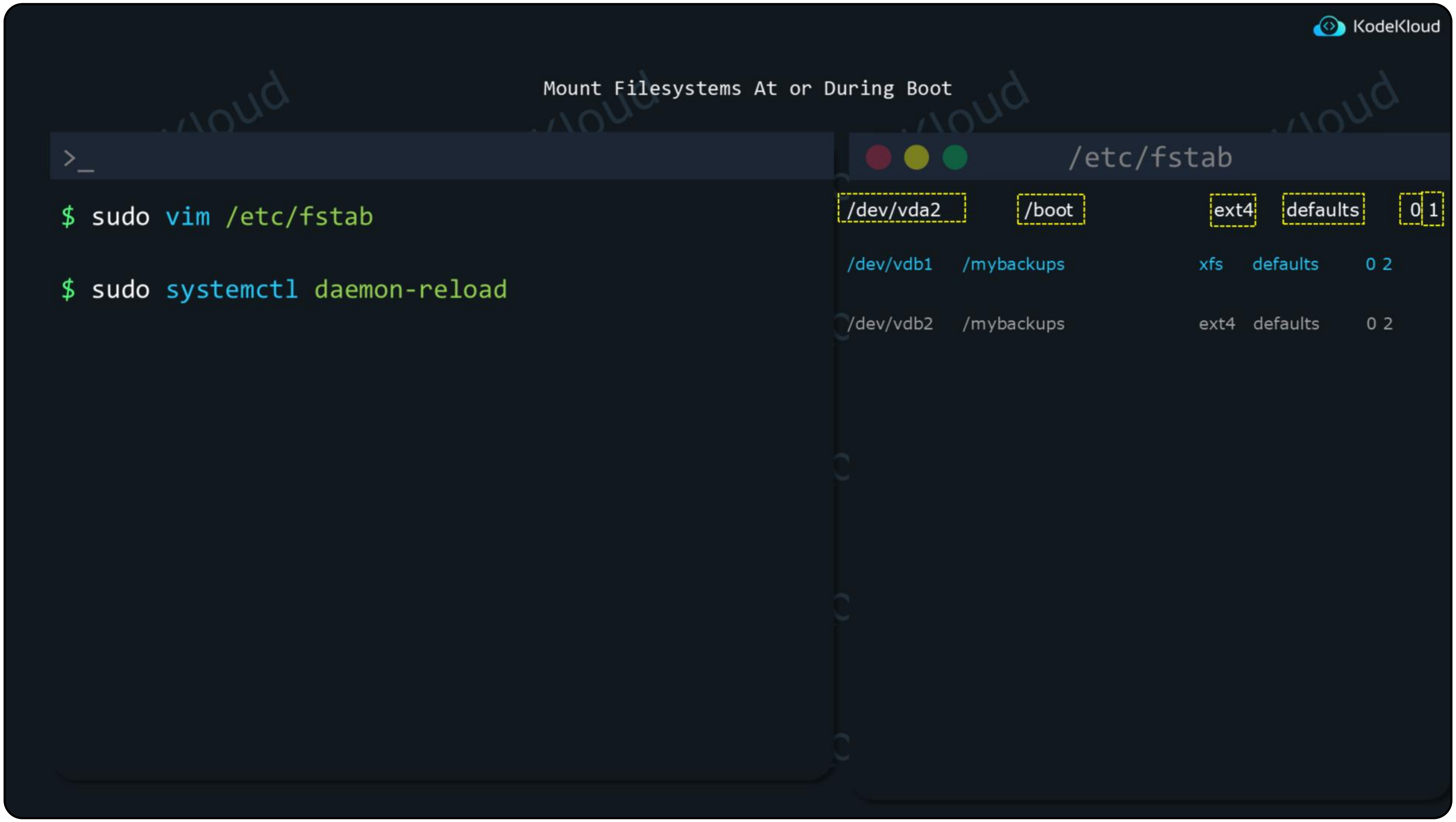
Mount Filesystems At or During Boot

```
$ lsblk

  NAME                         MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
  vda                          252:0    0   20G  0 disk
  ├─vda1                       252:1    0    1M  0 part
  ├─vda2                       252:2    0  1.8G  0 part /boot
  └─vda3                       252:3    0 18.2G  0 part
    └─ubuntu--vg-ubuntu--lv    253:0    0   10G  0 lvm  /
  vdb                          252:16   0   10G  0 disk
  ├─vdb1                       252:17   0    4G  0 part
  ├─vdb2                       252:18   0    4G  0 part
  └─vdb3                       252:19   0    2G  0 part
```

```
$ sudo mkdir /mybackups/
```

We notice in the lsblk command that some filesystems on some partitions are already mounted. For example, /dev/vda2 is mounted in the /boot/ directory. When a Linux operating system boots up, it automatically mounts some filesystems. It does this according to instructions in a file. Let's see how we can make it mount our XFS filesystem at /dev/vdb1. First, we'll create a directory that we intend to use as our mounting point:

sudo mkdir /mybackups/



Mount Filesystems At or During Boot

```
$ sudo vim /etc/fstab

$ sudo systemctl daemon-reload
```

/etc/fstab

```
/dev/vda2    /boot          ext4   defaults   0 1
/dev/vdb1    /mybackups     xfs    defaults   0 2
/dev/vdb2    /mybackups     ext4   defaults   0 2
```

/etc/fstab is the file where we can define what should be mounted automatically when the system boots up. Let's edit it:

sudo vim /etc/fstab

A line in fstab has 6 fields. Let's take a look at what each one represents.

The first usually points to the block device file that represents a partition or some kind of storage space. In the line we will add, this will be the block device file /dev/vdb1 representing the first partition on our second virtual disk. To be more specific, by writing /dev/vdb1 in our first field, we tell Linux "mount the filesystem contained in the partition at /dev/vdb1".

The second field specifies the mount point. In our case, this will be the directory where we want to mount our filesystem, /mybackups.

On the third field we specify the filesystem type. This is xfs in our case. But if we'd want to automount /dev/vdb2 instead of /dev/vdb1 we'd type ext4 since we have an ext4 filesystem on the other partition.

The fourth field will contain the mount options. We'll type defaults here to use default mounting options. But we'll take a brief look at custom mount options in one of our next lessons.

The fifth field determines if an utility called "dump" should backup this filesystem. 0 means backup disabled, 1 means backup enabled. dump is rarely used these days so we can usually just set the fifth field to 0.

Filesystems can sometimes get corrupted. And the last field decides what happens when errors are detected. We can have three values here: 0, 1, or 2. 0 means the filesystem should never be scanned for errors. 1 means this filesystem should be scanned first for errors, before the other ones. 2 means this filesystem should be scanned after the ones with a value of 1 have been scanned. In practice, we should often write "1" here for the root filesystem (where the operating system is installed). And we should write "2" for all the other filesystems.

We'll end up with this line in our fstab file:

```
/dev/vdb1 /mybackups xfs defaults 0 2
```

If we would have wanted to mount our other filesystem, stored on the second partition, we would have used a line like this:

```
/dev/vdb2 /mybackups ext4 defaults 0 2
```

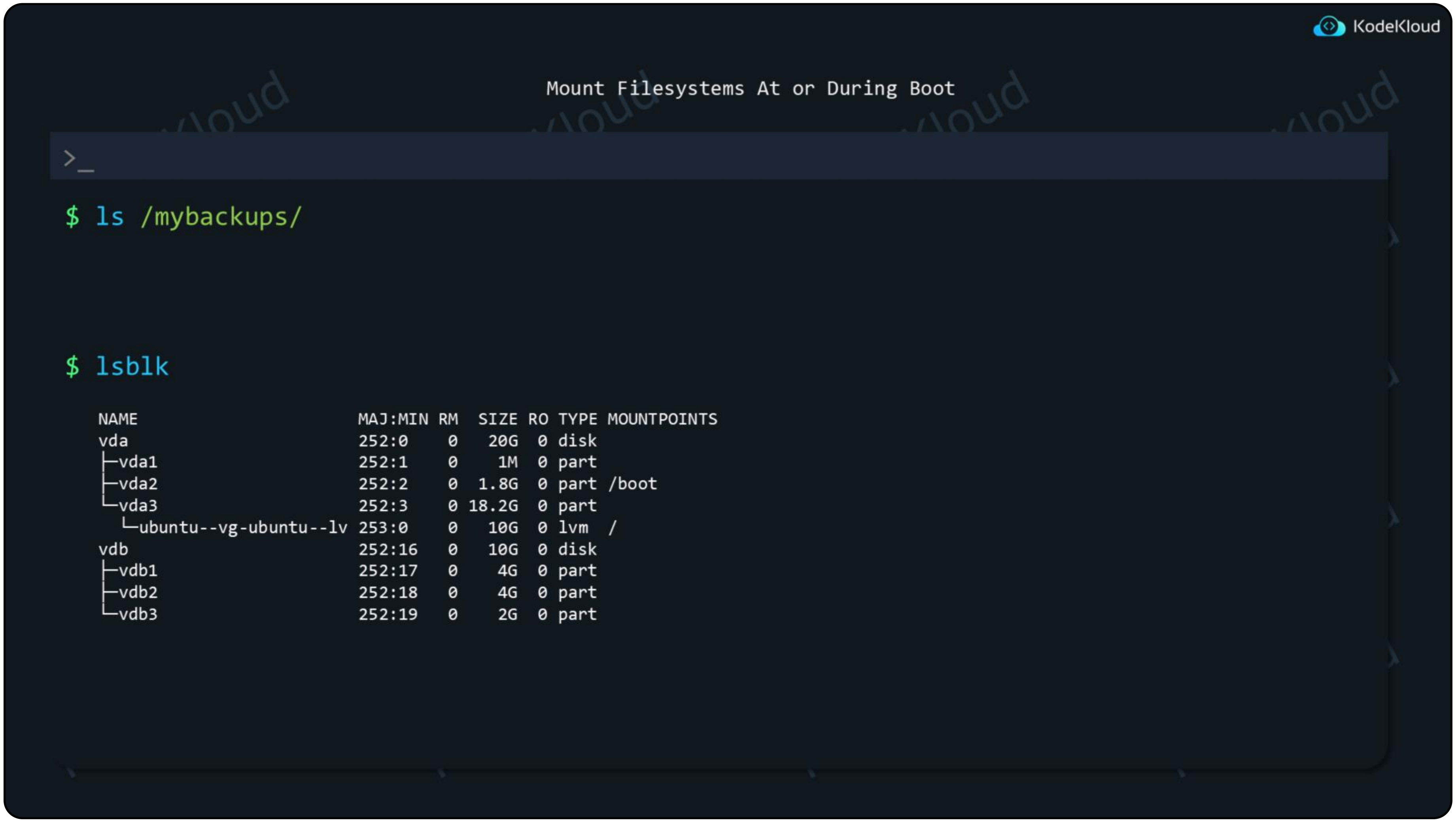
But we won't use this here since it would interfere with our previous line that also automounts in the /mybackups directory.

Let's save this file.

If we edit fstab and don't intend to reboot the system, systemd, the daemon that takes care of important things on the operating system won't know about our changes. With the next command, we can force it to pick up on any changes we made to important configuration files, fstab included:

```
sudo systemctl daemon-reload
```

If we reboot, it will pick up on these changes automatically.



Mount Filesystems At or During Boot

```
$ ls /mybackups/



$ lsblk

NAME                       MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
vda                        252:0    0   20G  0 disk
├─vda1                     252:1    0    1M  0 part
├─vda2                     252:2    0  1.8G  0 part /boot
└─vda3                     252:3    0 18.2G  0 part
  └─ubuntu--vg-ubuntu--lv  253:0    0   10G  0 lvm  /
vdb                        252:16   0   10G  0 disk
├─vdb1                     252:17   0    4G  0 part
├─vdb2                     252:18   0    4G  0 part
└─vdb3                     252:19   0    2G  0 part
```

Now if we look at this directory:

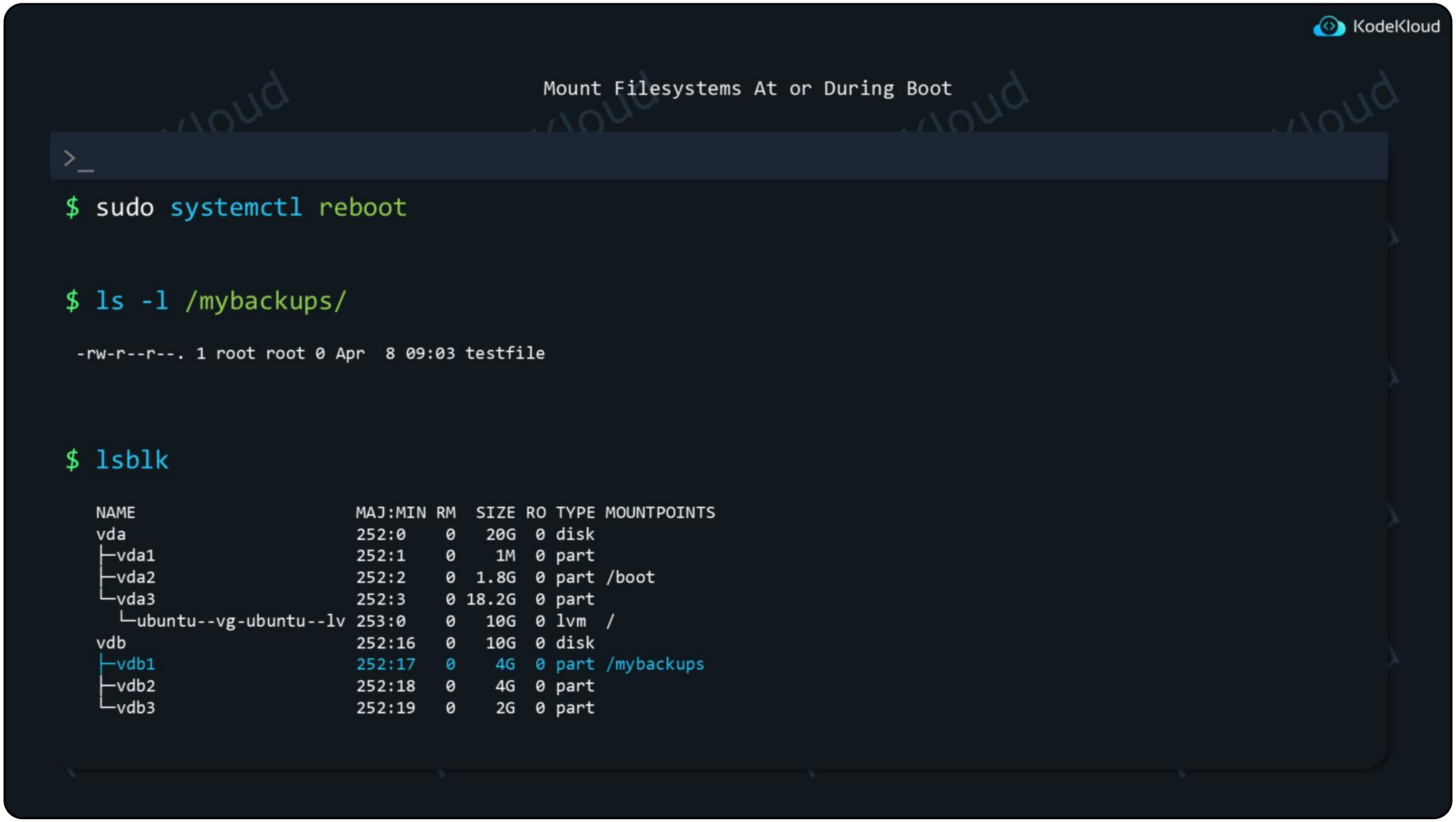
ls /mybackups/

we'll see it's empty, as expected.

Also,

```
lsblk
```

will show nothing is mounted in /mybackups/.



Mount Filesystems At or During Boot

```
>_

$ sudo systemctl reboot


$ ls -l /mybackups/

 -rw-r--r--. 1 root root 0 Apr  8 09:03 testfile



$ lsblk

   NAME                        MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
   vda                         252:0    0   20G  0 disk
   ├─vda1                      252:1    0    1M  0 part
   ├─vda2                      252:2    0  1.8G  0 part /boot
   └─vda3                      252:3    0 18.2G  0 part
     └─ubuntu--vg-ubuntu--lv   253:0    0   10G  0 lvm  /
   vdb                         252:16   0   10G  0 disk
   ├─vdb1                      252:17   0    4G  0 part /mybackups
   ├─vdb2                      252:18   0    4G  0 part
   └─vdb3                      252:19   0    2G  0 part
```

But if we reboot:

sudo systemctl reboot

and then SSH back in, and enter this command:

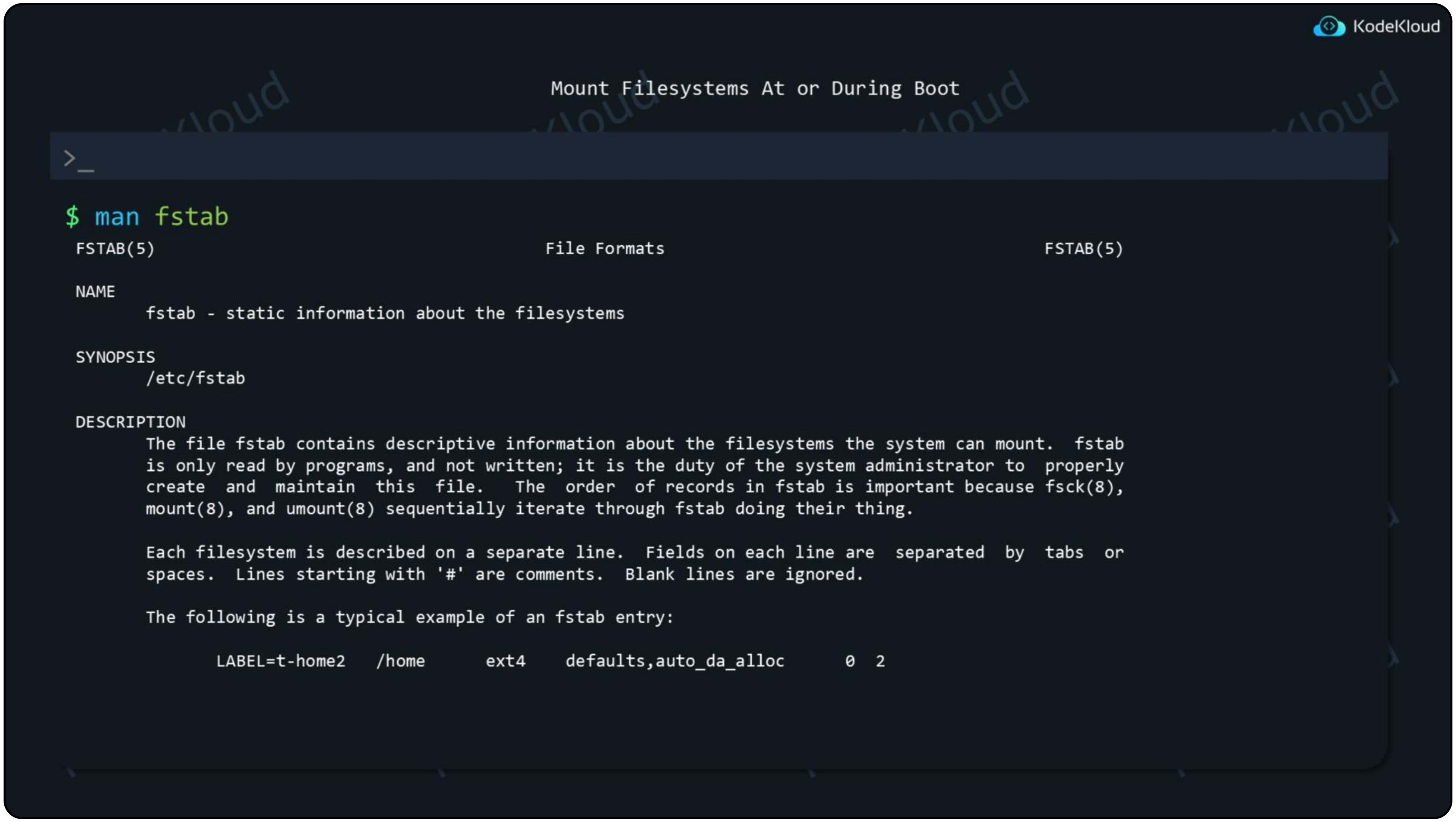
```
ls -l /mybackups/
```

we'll see the file we created on our XFS filesystem at the beginning of this lesson.

Also,

```
lsblk
```

will confirm that everything worked as expected. Our filesystem was automatically mounted to the /mybackups directory when our operating system booted up.



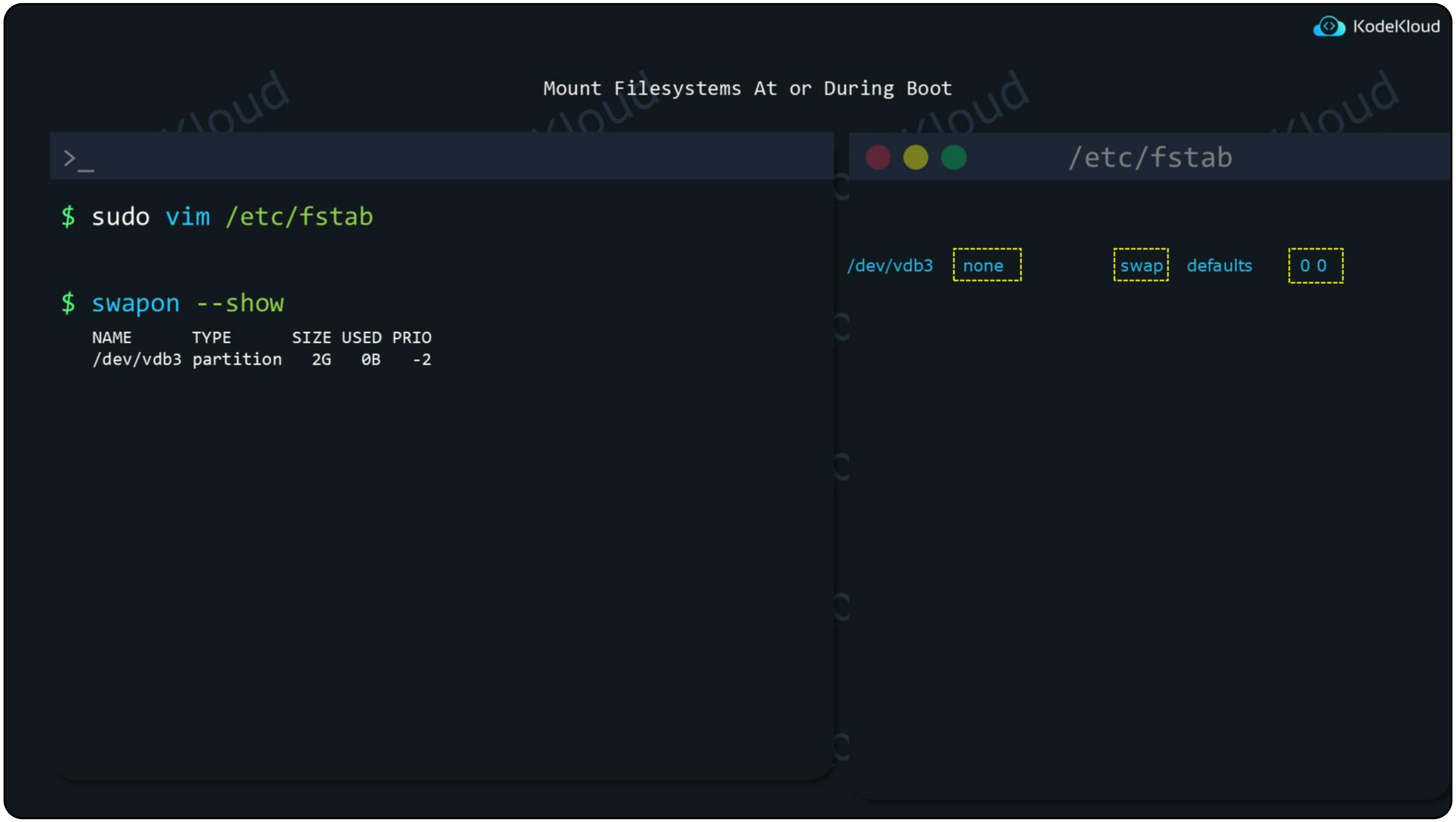
Mount Filesystems At or During Boot

```
$ man fstab
FSTAB(5)                              File Formats                              FSTAB(5)

NAME
       fstab - static information about the filesystems

SYNOPSIS
       /etc/fstab

DESCRIPTION
       The file fstab contains descriptive information about the filesystems the system can mount.  fstab
       is only read by programs, and not written; it is the duty of the system administrator to  properly
       create  and  maintain  this  file.   The  order  of records in fstab is important because fsck(8),
       mount(8), and umount(8) sequentially iterate through fstab doing their thing.

       Each filesystem is described on a separate line.  Fields on each line are  separated  by  tabs  or
       spaces.  Lines starting with '#' are comments.  Blank lines are ignored.

       The following is a typical example of an fstab entry:

              LABEL=t-home2   /home      ext4    defaults,auto_da_alloc      0  2
```

We can usually figure out what we need to write in /etc/fstab just by looking at existing entries. But if you ever forget what each field represents, remember that the

man fstab

command can be helpful.



Mount Filesystems At or During Boot

```
>_
$ sudo vim /etc/fstab

$ swapon --show
  NAME      TYPE      SIZE USED PRIO
  /dev/vdb3 partition   2G   0B   -2
```

/etc/fstab

```
/dev/vdb3  none          swap  defaults   0 0
```

In a previous lesson we created a swap partition at /dev/vdb3. It would be useful if this would be mounted automatically at boot time too. So, let's edit fstab again:

sudo vim /etc/fstab

We'll add this line:

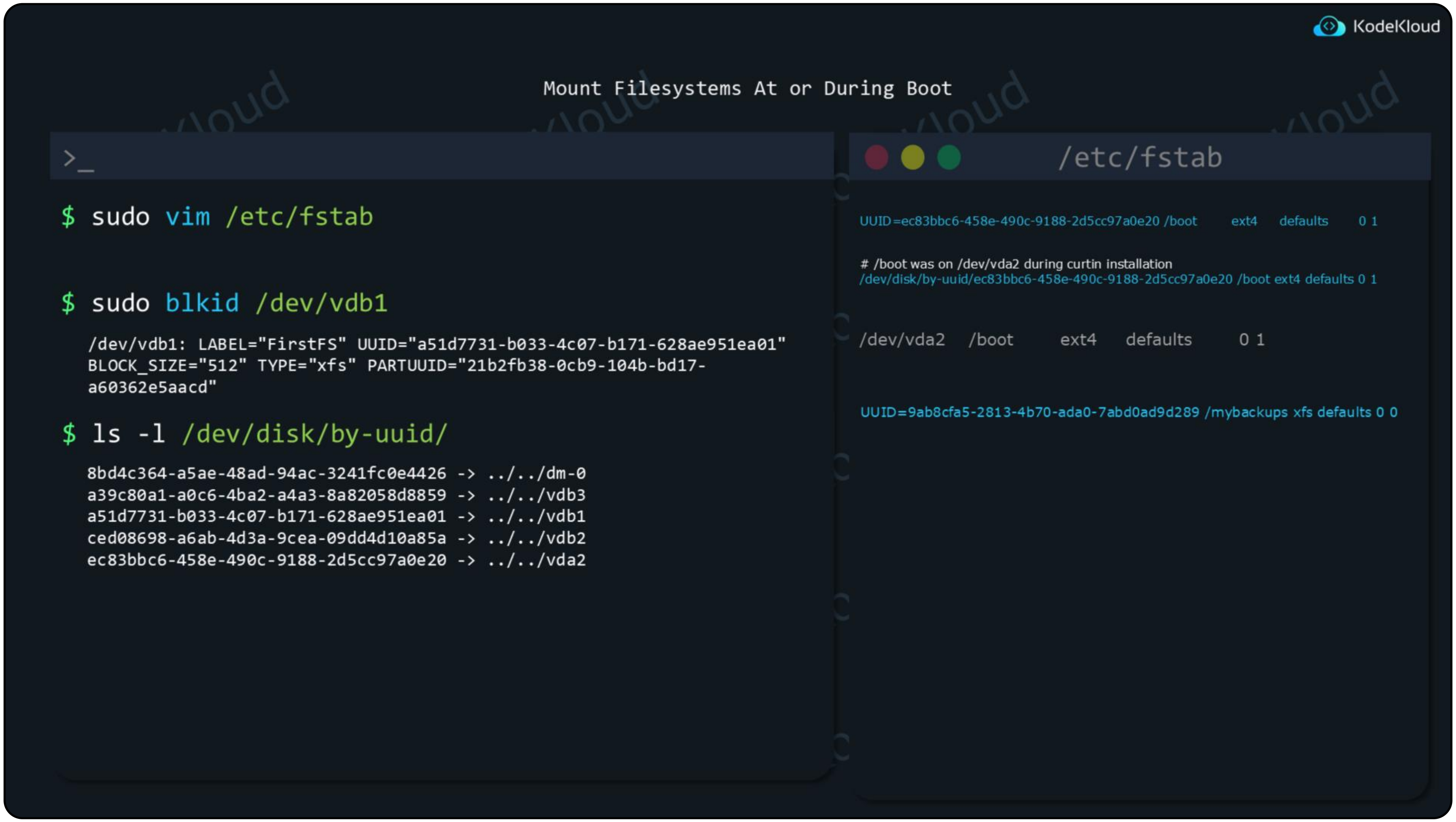
```
/dev/vdb3 none swap defaults 0 0
```

We see this line is similar to the line we created for our xfs filesystem. There are only two notable differences in the second and third field. The second field should normally contain the mount point, the directory a filesystem should be mounted in. But we specify none here, since swap is not meant to be mounted in any directory. In the third field, the filesystem type, we specify this is swap. And, finally, we use "0 0" in the last two fields as swap is not meant to be backed up or scanned for errors.

If we would reboot and then use this command:

```
swapon --show
```

We would see that our swap partition was automatically mounted at boot time.



Mount Filesystems At or During Boot

```
$ sudo vim /etc/fstab


$ sudo blkid /dev/vdb1

  /dev/vdb1: LABEL="FirstFS" UUID="a51d7731-b033-4c07-b171-628ae951ea01"
  BLOCK_SIZE="512" TYPE="xfs" PARTUUID="21b2fb38-0cb9-104b-bd17-
  a60362e5aacd"

$ ls -l /dev/disk/by-uuid/

  8bd4c364-a5ae-48ad-94ac-3241fc0e4426 -> ../../dm-0
  a39c80a1-a0c6-4ba2-a4a3-8a82058d8859 -> ../../vdb3
  a51d7731-b033-4c07-b171-628ae951ea01 -> ../../vdb1
  ced08698-a6ab-4d3a-9cea-09dd4d10a85a -> ../../vdb2
  ec83bbc6-458e-490c-9188-2d5cc97a0e20 -> ../../vda2
```

/etc/fstab

```
UUID=ec83bbc6-458e-490c-9188-2d5cc97a0e20 /boot     ext4    defaults     0 1

# /boot was on /dev/vda2 during curtin installation
/dev/disk/by-uuid/ec83bbc6-458e-490c-9188-2d5cc97a0e20 /boot ext4 defaults 0 1


/dev/vda2    /boot      ext4    defaults      0 1


UUID=9ab8cfa5-2813-4b70-ada0-7abd0ad9d289 /mybackups xfs defaults 0 0
```

In the /etc/fstab file we can also notice lines like these:

UUID=ec83bbc6-458e-490c-9188-2d5cc97a0e20   /boot   xfs   defaults   0 0

Or

```
/dev/disk/by-uuid/ec83bbc6-458e-490c-9188-2d5cc97a0e20 /boot ext4 defaults 0 1
```

Note the comment above the /dev/disk/ line.

These lines can be equivalent to something like /dev/vda2, mounted in the /boot/ directory. So the lines above could have also be written this way:

```
/dev/vda2    /boot     ext4     defaults       0 1
```

So why was a UUID, universally unique identifier, used instead of a block device file like /dev/vda2? Imagine a real server with 2 SSDs connected to some SATA ports on the motherboard. First SSD could be found at /dev/sda. Second SSD at /dev/sdb. But if we remove the SATA cables and connect them in reverse order on the motherboard, first SSD could be found at /dev/sdb instead of /dev/sda, the next time we boot. Linux assigns these /dev/vda, or /dev/vdb device names in the order it sees they're connected to the motherboard.

Mounting the wrong thing in the wrong place because the device name has changed could lead to bad results. Therefore, UUIDs can be used instead. These remain the same, even if we connect our storage devices in a different order on the motherboard.

To check the UUID of a block device, we can use this command:

```
sudo blkid /dev/vdb1
```

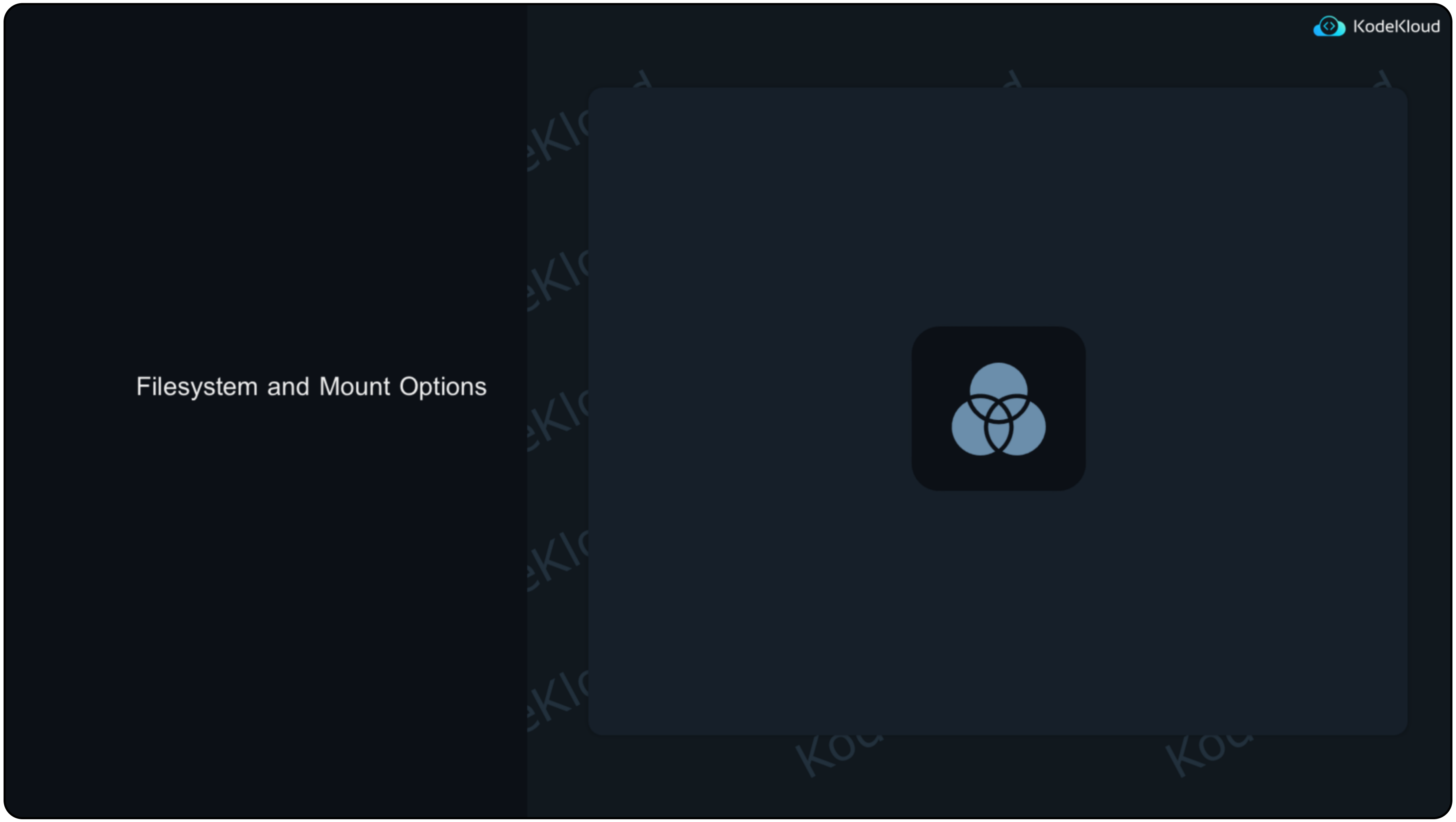
Also, if we want to use a device name in the form of /dev/disk/by-uuid/, we can just explore this directory with an ls -l command:

```
ls -l /dev/disk/by-uuid/
```

And we'd see the unique identifiers and what block devices they actually point to. Next we could pick what we need from here, and use the /dev/disk/by-uuid/ format, followed by the unique identifier as the first field in one of our fstab lines.

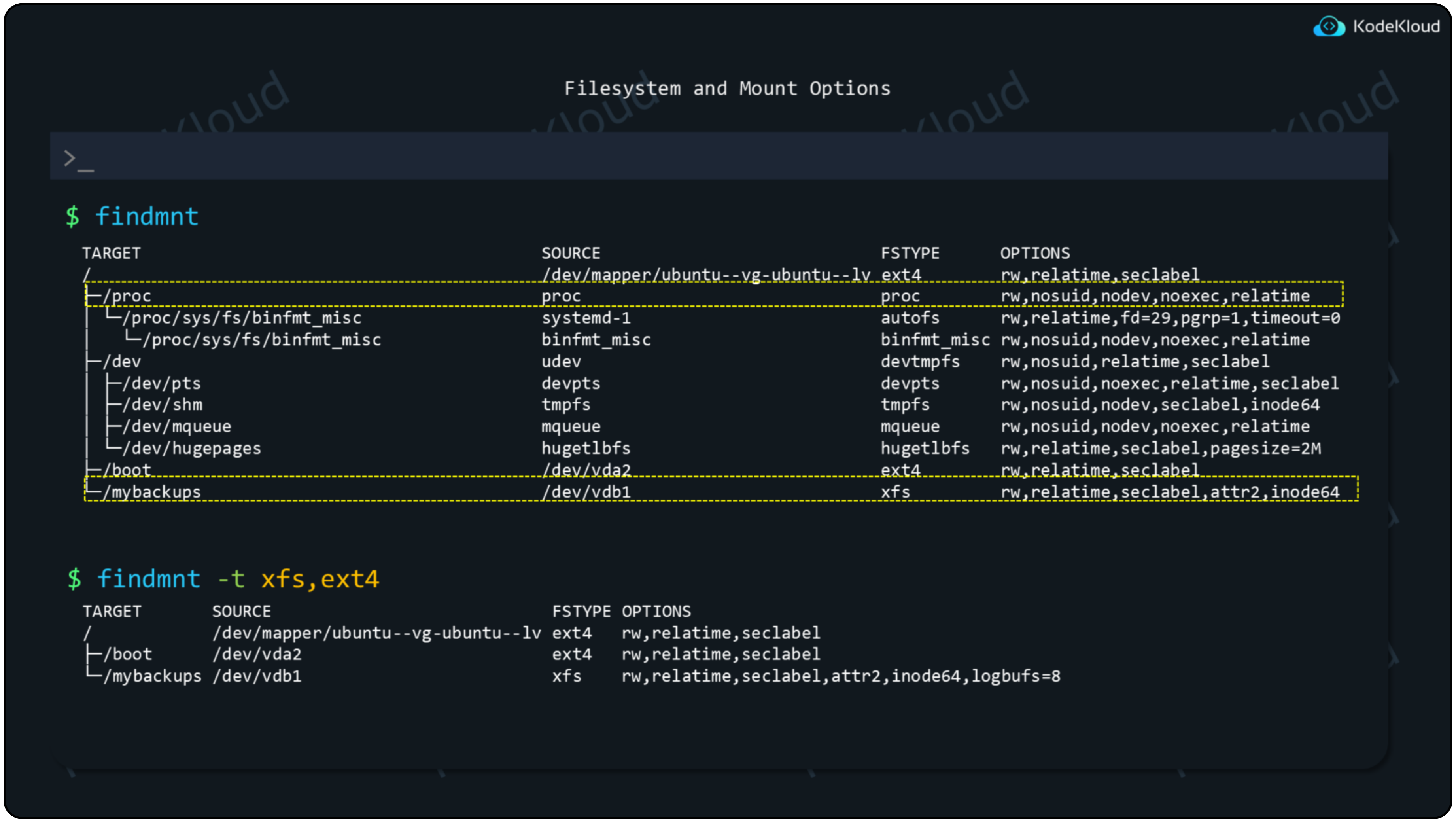
KodeKloud

Visit www.kodekloud.com to discover more.



Filesystem and Mount Options

Now, let's look at how to adjust filesystem and mount options.



Filesystem and Mount Options

```
$ findmnt
TARGET                                      SOURCE                            FSTYPE      OPTIONS
/                                           /dev/mapper/ubuntu--vg-ubuntu--lv ext4        rw,relatime,seclabel
├─/proc                                     proc                              proc        rw,nosuid,nodev,noexec,relatime
│ └─/proc/sys/fs/binfmt_misc                systemd-1                         autofs      rw,relatime,fd=29,pgrp=1,timeout=0
│   └─/proc/sys/fs/binfmt_misc              binfmt_misc                       binfmt_misc rw,nosuid,nodev,noexec,relatime
├─/dev                                      udev                              devtmpfs    rw,nosuid,relatime,seclabel
│ ├─/dev/pts                                devpts                            devpts      rw,nosuid,noexec,relatime,seclabel
│ ├─/dev/shm                                tmpfs                             tmpfs       rw,nosuid,nodev,seclabel,inode64
│ ├─/dev/mqueue                             mqueue                            mqueue      rw,nosuid,nodev,noexec,relatime
│ └─/dev/hugepages                          hugetlbfs                         hugetlbfs   rw,relatime,seclabel,pagesize=2M
├─/boot                                     /dev/vda2                         ext4        rw,relatime,seclabel
└─/mybackups                                /dev/vdb1                         xfs         rw,relatime,seclabel,attr2,inode64
```

```
$ findmnt -t xfs,ext4
TARGET       SOURCE                            FSTYPE OPTIONS
/            /dev/mapper/ubuntu--vg-ubuntu--lv ext4   rw,relatime,seclabel
├─/boot      /dev/vda2                         ext4   rw,relatime,seclabel
└─/mybackups /dev/vdb1                         xfs    rw,relatime,seclabel,attr2,inode64,logbufs=8
```

We saw how the

lsblk

command gives us a nice overview of what is mounted where. It's short and to the point. But, sometimes, we'll also want the details; details like: what filesystem is mounted here? And what are the mount options for this filesystem? Of course, we'll first need to understand what mount options are. But we'll get to that in a few seconds.

First, let's look at another command that shows everything that is mounted on this system:

```
findmnt
```

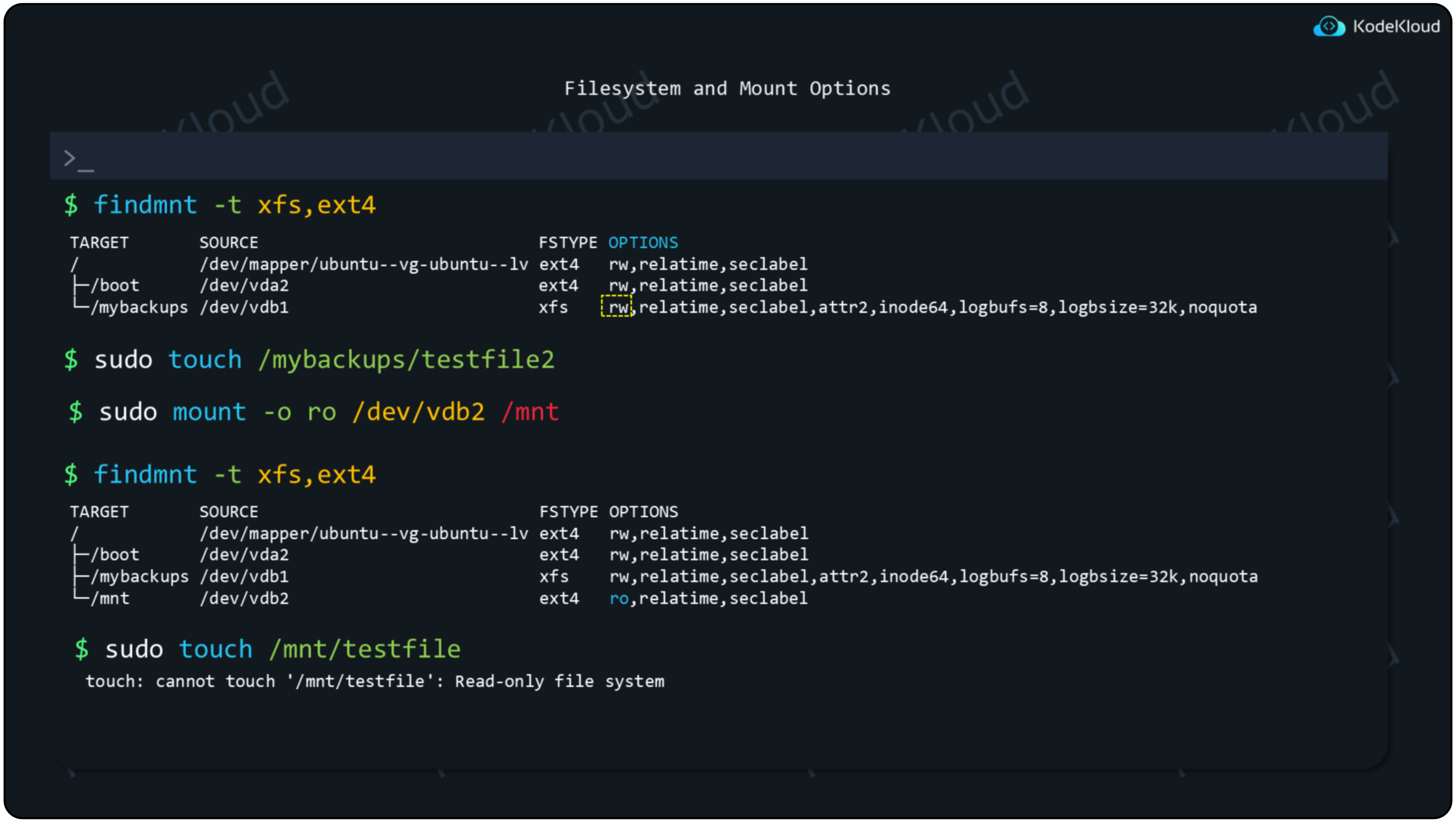
We can spot our /dev/vdb1 partition with the xfs filesystem, mounted in the /mybackups directory.

This command will output many lines. Usually, double of what we can see here. We've truncated output in this example so that it can fit our screen. It can look messy and hard to read, especially if we don't need all this information. findmnt shows all mount points, including some virtual stuff. For example, proc is a virtual filesystem, mounted in the /proc directory. This filesystem does not exist on some storage device somewhere, only in the computer's Random Access Memory. In our situation, we don't need to see all this virtual stuff. So, we can tell findmnt to only show us "real" filesystems. For example, we can say: show us only xfs and ext4 filesystems that are mounted. To do so, we use the "-t" type command line option:

```
findmnt -t xfs,ext4
```

Now that looks much cleaner and is easy to read.

An important note to make is that findmnt only shows us filesystems that are currently mounted. Those that exist on some partition somewhere, but are not yet mounted, won't show up in this output.



Filesystem and Mount Options

```
$ findmnt -t xfs,ext4

TARGET        SOURCE                            FSTYPE  OPTIONS
/             /dev/mapper/ubuntu--vg-ubuntu--lv ext4    rw,relatime,seclabel
├─/boot       /dev/vda2                         ext4    rw,relatime,seclabel
└─/mybackups  /dev/vdb1                         xfs     rw,relatime,seclabel,attr2,inode64,logbufs=8,logbsize=32k,noquota

$ sudo touch /mybackups/testfile2

$ sudo mount -o ro /dev/vdb2 /mnt

$ findmnt -t xfs,ext4

TARGET        SOURCE                            FSTYPE  OPTIONS
/             /dev/mapper/ubuntu--vg-ubuntu--lv ext4    rw,relatime,seclabel
├─/boot       /dev/vda2                         ext4    rw,relatime,seclabel
├─/mybackups  /dev/vdb1                         xfs     rw,relatime,seclabel,attr2,inode64,logbufs=8,logbsize=32k,noquota
└─/mnt        /dev/vdb2                         ext4    ro,relatime,seclabel

$ sudo touch /mnt/testfile
 touch: cannot touch '/mnt/testfile': Read-only file system
```

Notice the OPTIONS column here. This is where findmnt shows us the mount options. What are mount options? It will be easier to understand if we look at the effects of these. For example, we see /dev/vdb1 is mounted with an option called rw. That means we can read and write to this filesystem. We see it's true since we can create a new file on it:

sudo touch /mybackups/testfile2

But what if we would use an option called ro instead of rw? That would make the filesystem read-only.

We can mount a filesystem with specific options with the -o switch passed to the mount command. For example:

```
sudo mount -o ro /dev/vdb2 /mnt
```

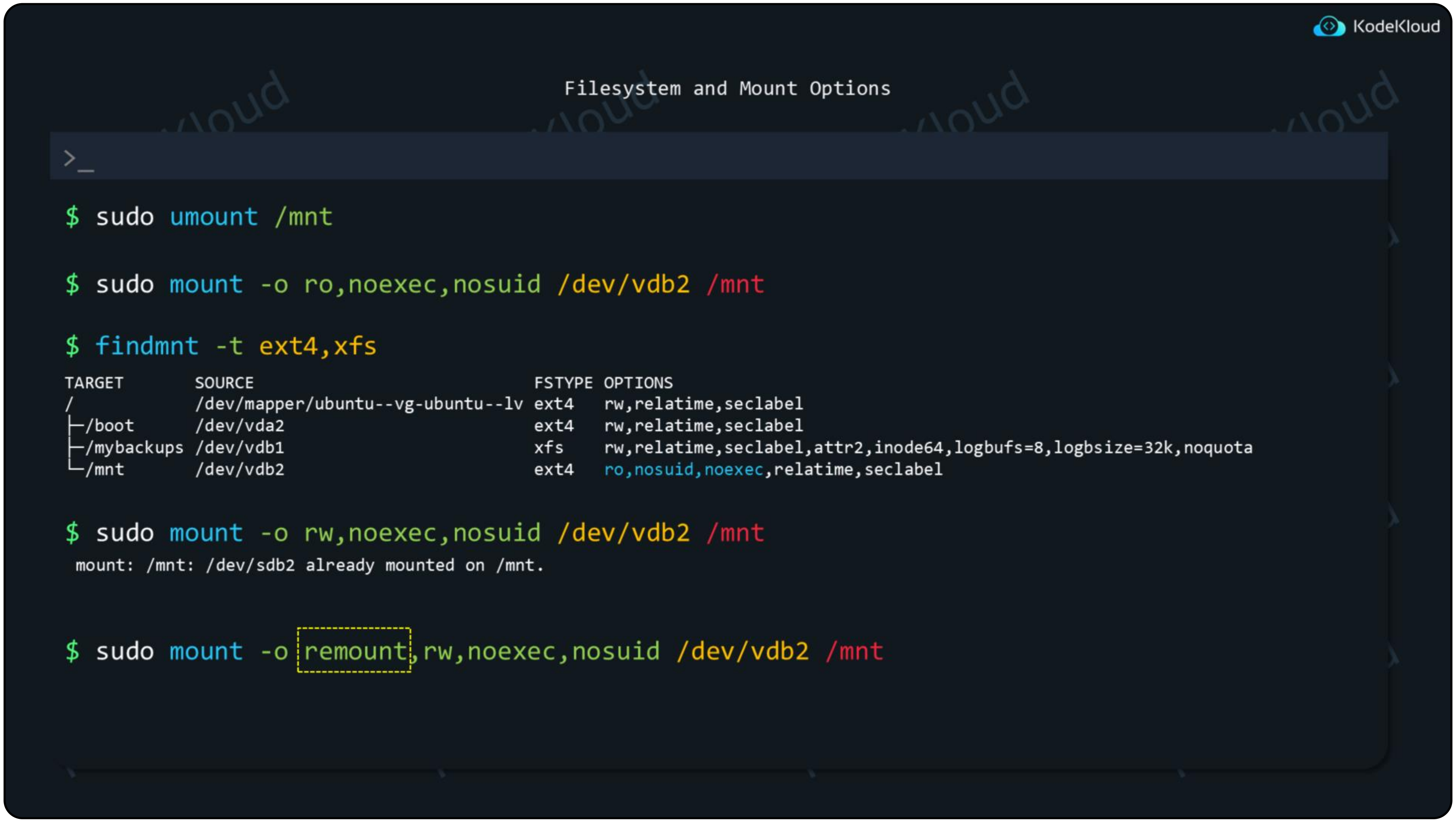
Let's check out our mounted filesystems again:

```
findmnt -t xfs,ext4
```

We see our mount option was applied. Now we can also test it. Let's try to create a file:

```
sudo touch /mnt/testfile
```

As expected, this action is denied since our filesystem is mounted in read-only mode.



```
$ sudo umount /mnt

$ sudo mount -o ro,noexec,nosuid /dev/vdb2 /mnt

$ findmnt -t ext4,xfs
TARGET        SOURCE                            FSTYPE  OPTIONS
/             /dev/mapper/ubuntu--vg-ubuntu--lv ext4    rw,relatime,seclabel
├─/boot       /dev/vda2                         ext4    rw,relatime,seclabel
├─/mybackups  /dev/vdb1                         xfs     rw,relatime,seclabel,attr2,inode64,logbufs=8,logbsize=32k,noquota
└─/mnt        /dev/vdb2                         ext4    ro,nosuid,noexec,relatime,seclabel

$ sudo mount -o rw,noexec,nosuid /dev/vdb2 /mnt
 mount: /mnt: /dev/sdb2 already mounted on /mnt.


$ sudo mount -o remount,rw,noexec,nosuid /dev/vdb2 /mnt
```

So, we can draw the general conclusion that mount options, in a way, tell the filesystem how to behave, what rules it should follow, while it is mounted.

Let's unmount this:

sudo umount /mnt

And see how we can use multiple mount options, like ro, noexec and nosuid. noexec makes it impossible to launch a program stored on this filesystem. nosuid disables the SUID permission that can allow programs to run with root privileges, without needing the sudo command. Sometimes options like noexec and nosuid are used to improve security on filesystems that should not have programs on them. For example, these are used on Android phones, on the storage area that should contain photos, videos, and similar data. Now even if we plug the phone into an infected computer and a virus would try to replicate on this filesystem, it would have no effect. The virus would be able to write infected files. But they would be saved on a noexec and nosuid mount point. This way, they cannot execute and do any harm. They're just a bunch of files sitting there, doing nothing.

To mount our filesystem with these options, we run this command:

```
sudo mount -o ro,noexec,nosuid /dev/vdb2 /mnt
```

These options should be separated only by commas, without any spaces between them.

Once again, we can confirm our options were applied:

```
findmnt -t ext4,xfs
```

Now let's say we want to write to this filesystem. We should change the ro mount option to rw.

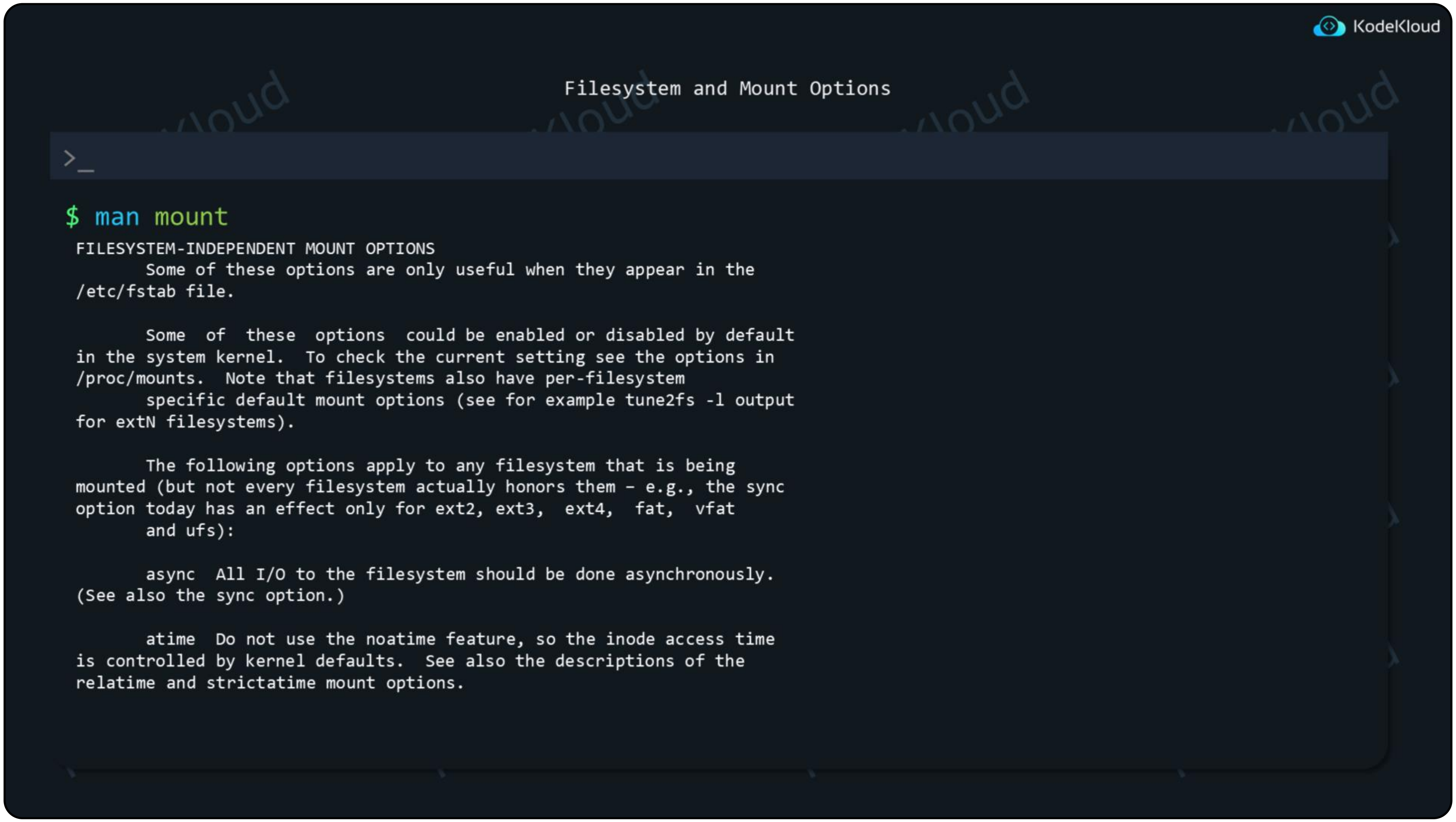
If we try this:

```
sudo mount -o rw,noexec,nosuid /dev/vdb2 /mnt
```

It wouldn't work. mount complains this is already mounted. But we can tell it to remount the filesystem with the new options, with a command like this:

```
sudo mount -o remount,rw,noexec,nosuid /dev/vdb2 /mnt
```

Basically, we just threw "remount" in there as yet another mount option.

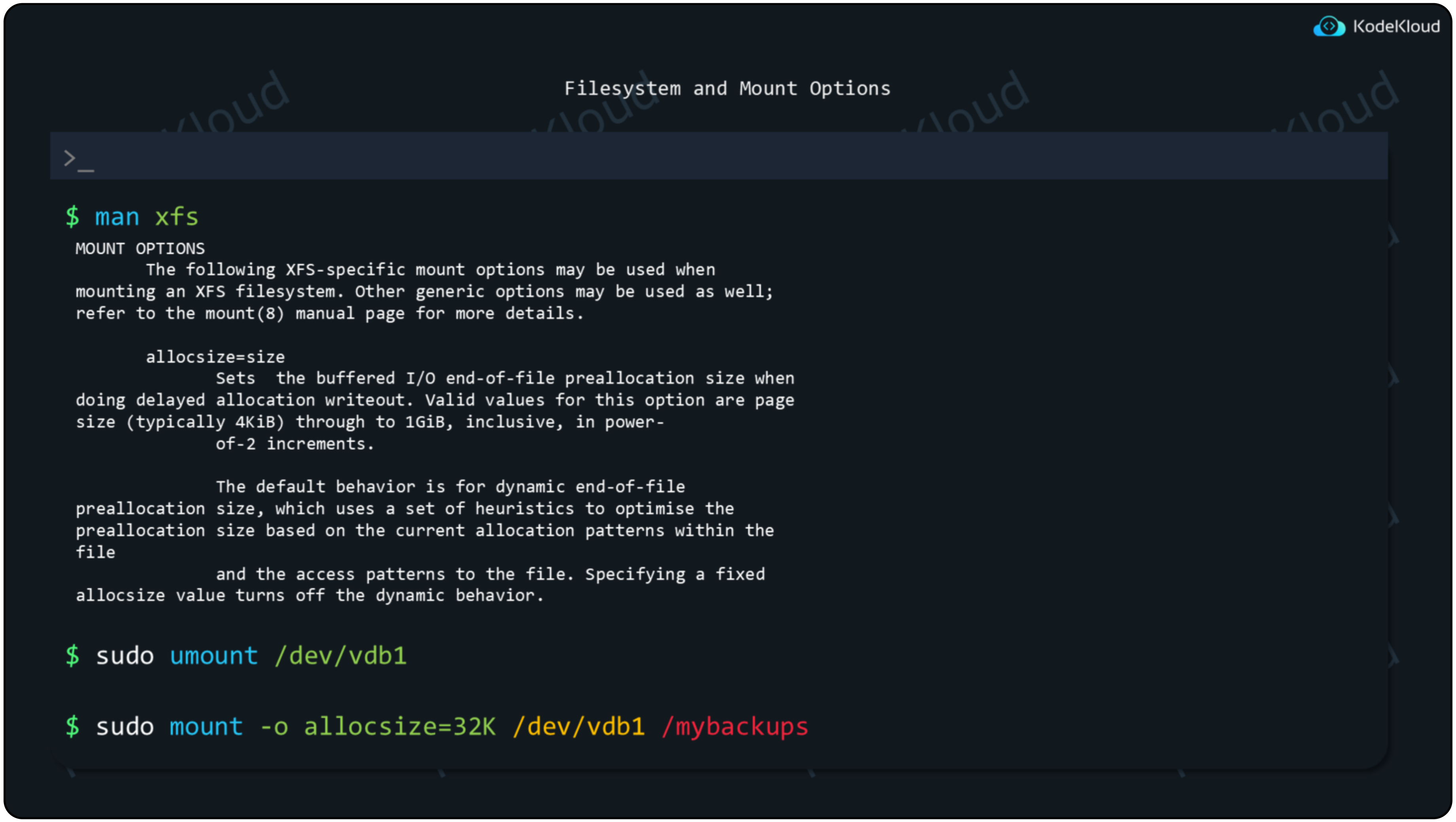

Filesystem and Mount Options

```
$ man mount
FILESYSTEM-INDEPENDENT MOUNT OPTIONS
       Some of these options are only useful when they appear in the
/etc/fstab file.

       Some  of  these  options  could be enabled or disabled by default
in the system kernel.  To check the current setting see the options in
/proc/mounts.  Note that filesystems also have per-filesystem
       specific default mount options (see for example tune2fs -l output
for extN filesystems).

       The following options apply to any filesystem that is being
mounted (but not every filesystem actually honors them - e.g., the sync
option today has an effect only for ext2, ext3,  ext4,  fat,  vfat
       and ufs):

       async  All I/O to the filesystem should be done asynchronously.
(See also the sync option.)

       atime  Do not use the noatime feature, so the inode access time
is controlled by kernel defaults.  See also the descriptions of the
relatime and strictatime mount options.
```

The options we used up to this point are called Filesystem-Independent. That's because they can be used on almost any filesystem.

Most of these mount options are described in the manual of the "mount" command:

man mount



KodeKloud

Filesystem and Mount Options

```
$ man xfs
 MOUNT OPTIONS
        The following XFS-specific mount options may be used when
 mounting an XFS filesystem. Other generic options may be used as well;
 refer to the mount(8) manual page for more details.

        allocsize=size
               Sets  the buffered I/O end-of-file preallocation size when
 doing delayed allocation writeout. Valid values for this option are page
 size (typically 4KiB) through to 1GiB, inclusive, in power-
               of-2 increments.

               The default behavior is for dynamic end-of-file
 preallocation size, which uses a set of heuristics to optimise the
 preallocation size based on the current allocation patterns within the
 file
               and the access patterns to the file. Specifying a fixed
 allocsize value turns off the dynamic behavior.

$ sudo umount /dev/vdb1

$ sudo mount -o allocsize=32K /dev/vdb1 /mybackups
```

But other options are filesystem-specific. Some options only work on xfs, others only on ext4, and so on. To see filesystem-specific mount options, we need to consult the manual of that filesystem. For example, in

man xfs

We can scroll down and reach the MOUNT OPTIONS section, where we can see these described:

For example, let's say we want to use the allocsize option described here. We'll first unmount /dev/vdb1 since it's already mounted in our scenario.

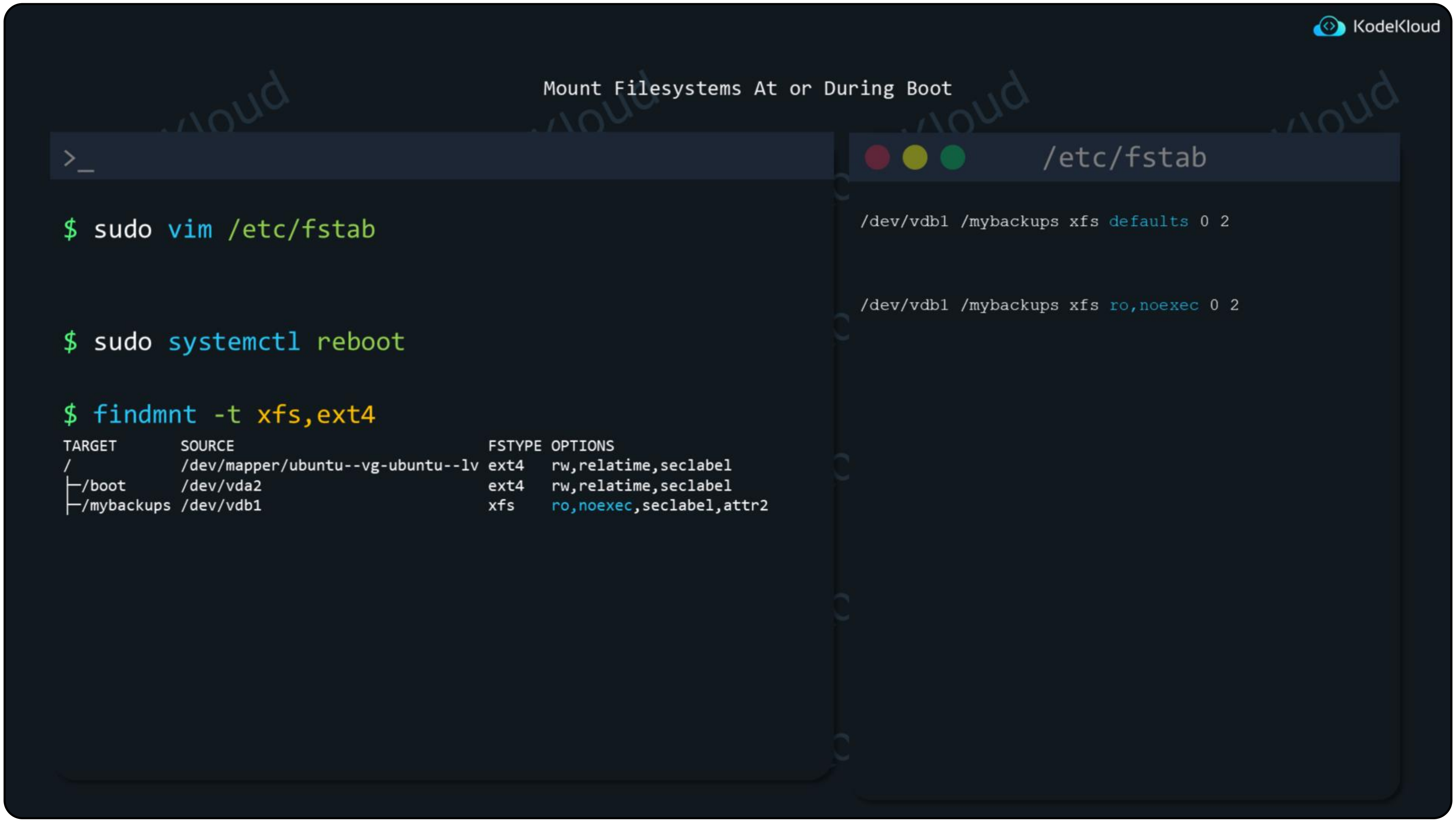
```
sudo umount /dev/vdb1
```

It's not recommended to use mount -o remount, in this case. With filesystem-independent options it's ok. With filesystem-dependent options, these might not be applied if the filesystem is already mounted.

Here's how we would use the allocsize option and set it to 32 kilobytes:

```
sudo mount -o allocsize=32K /dev/vdb1 /mybackups
```

To see filesystem-specific mount options for ext4, use the "man ext4" command.



Mount Filesystems At or During Boot

```
$ sudo vim /etc/fstab

$ sudo systemctl reboot

$ findmnt -t xfs,ext4
TARGET       SOURCE                             FSTYPE OPTIONS
/            /dev/mapper/ubuntu--vg-ubuntu--lv ext4   rw,relatime,seclabel
├─/boot      /dev/vda2                          ext4   rw,relatime,seclabel
├─/mybackups /dev/vdb1                          xfs    ro,noexec,seclabel,attr2
```

/etc/fstab

```
/dev/vdb1 /mybackups xfs defaults 0 2

/dev/vdb1 /mybackups xfs ro,noexec 0 2
```

Up to this point, we mounted our filesystems manually. But let's remember that these can also be mounted automatically at boot time. How do we apply mount options there?

Let's open up /etc/fstab:

sudo vim /etc/fstab

Remember how in a previous lesson we said we'll use the default mount options on this line?

```
/dev/vdb1 /mybackups xfs defaults 0 2
```

If we'd want to use custom options, such as ro and noexec we explored earlier, we can rewrite the line like this:

```
/dev/vdb1 /mybackups xfs ro,noexec 0 2
```

If we save the file and reboot:

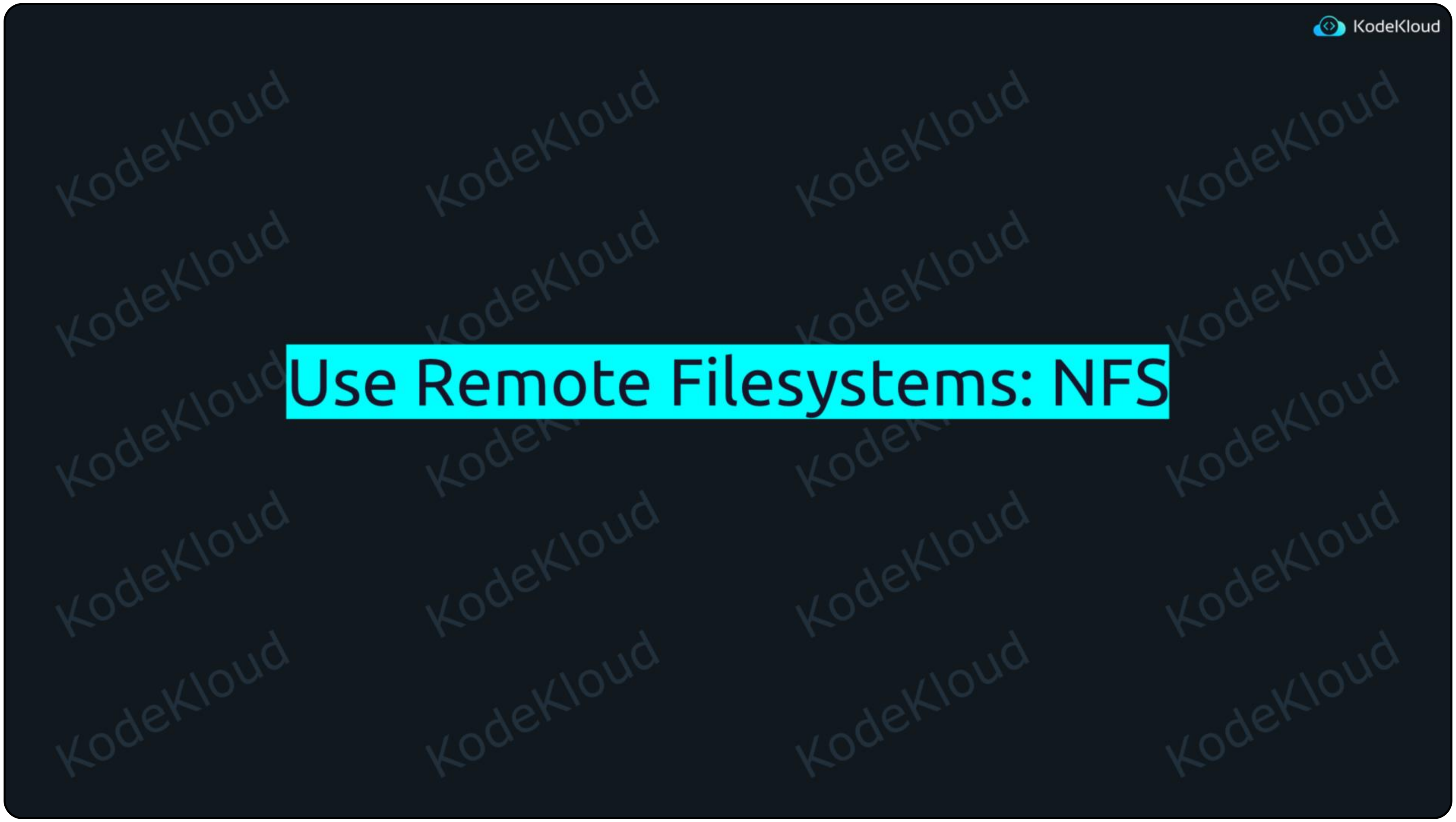
```
sudo systemctl reboot
```

we can see our mount options were automatically applied at boot time:

```
findmnt -t xfs,ext4
```

KodeKloud

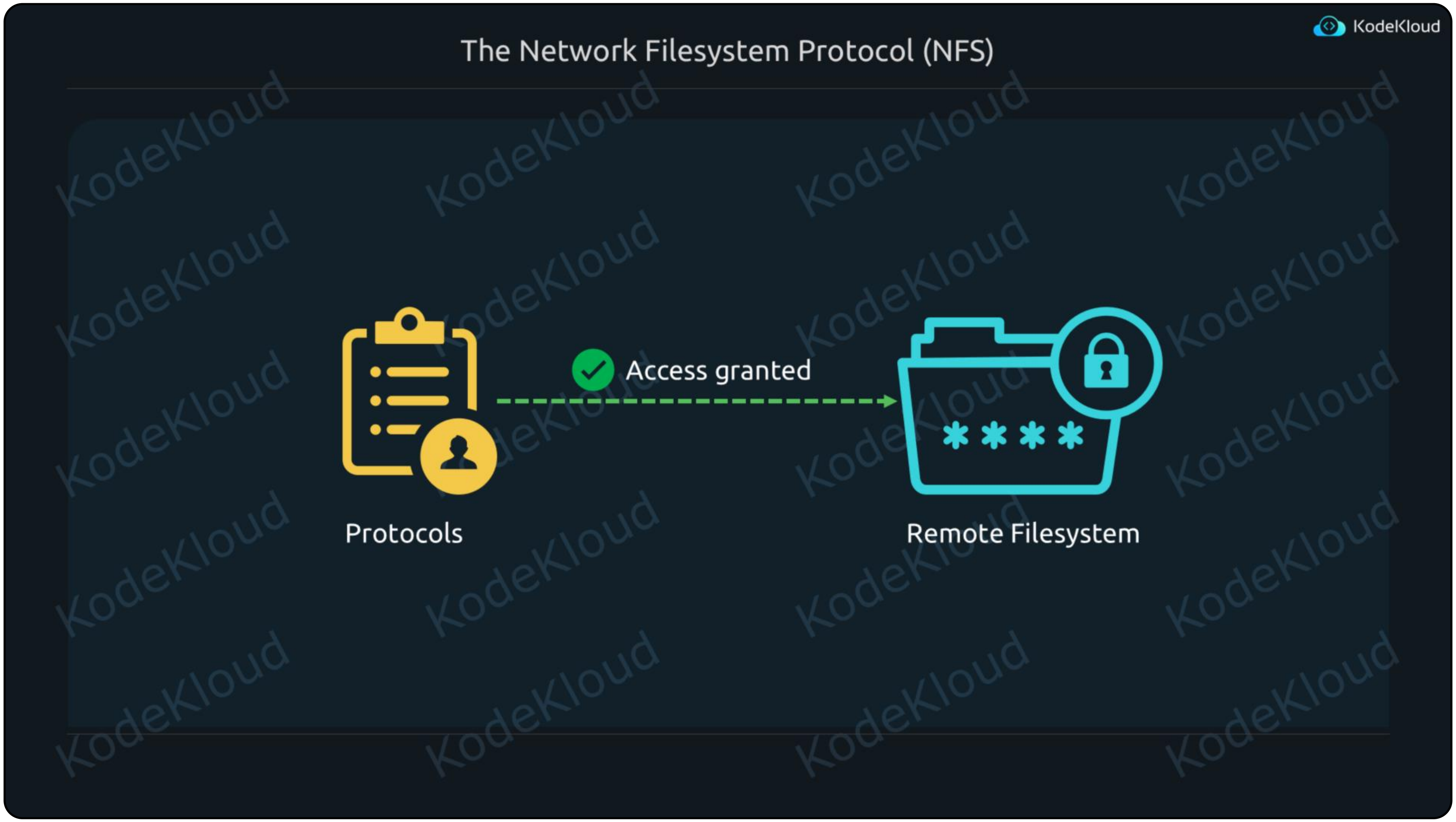
Visit www.kodekloud.com to discover more.



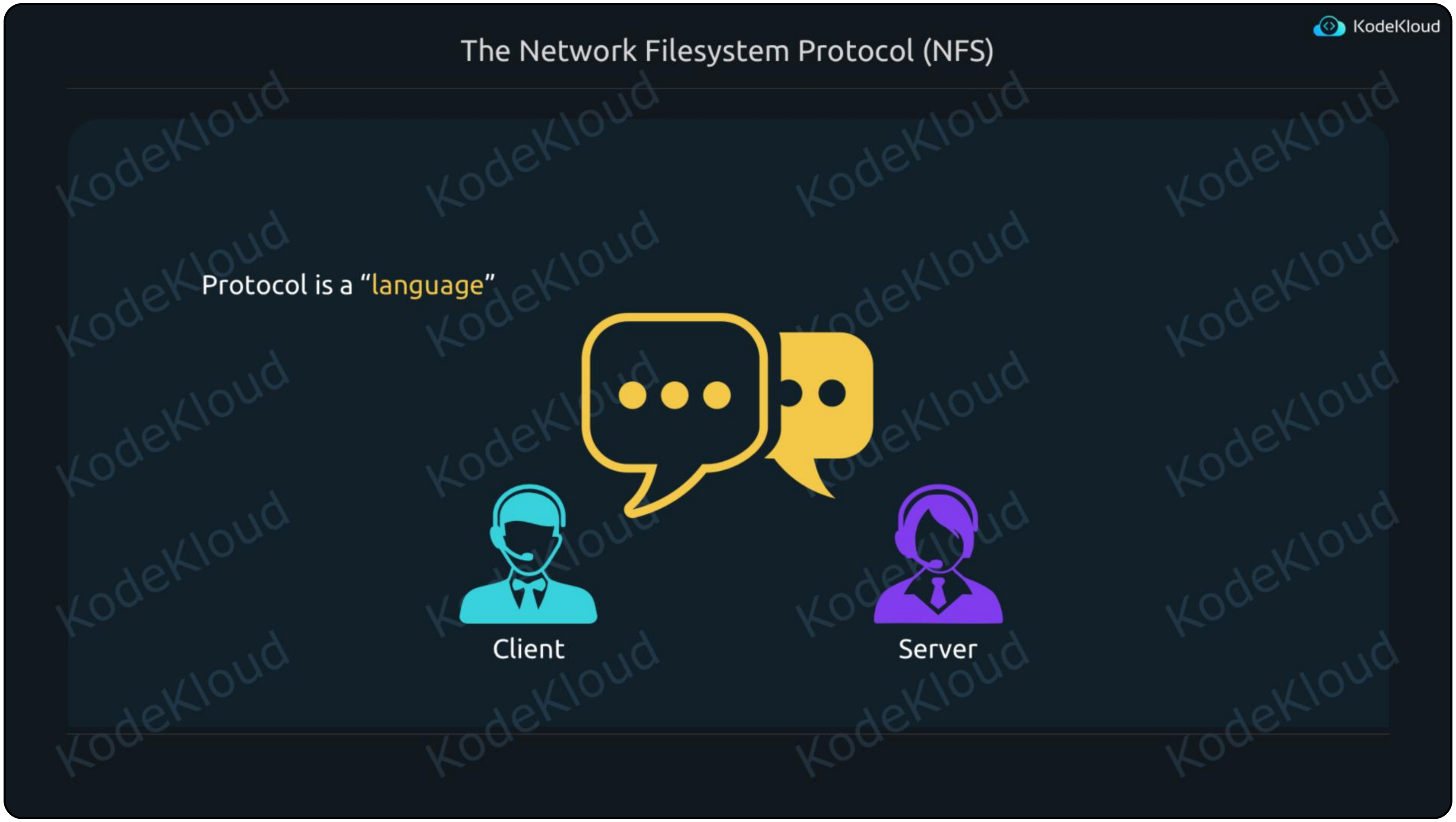
Use Remote Filesystems: NFS

In our previous lessons we learned how to work with local filesystems and local block storage devices. Basically, data that exists on the same system we're logged into. But, sometimes, we'll need data that resides on a different system.

In this lesson, we'll learn how to use remote filesystems. And in one of our next lessons, how to use remote block devices. Let's go through these scenarios in order.


The Network Filesystem Protocol (NFS)

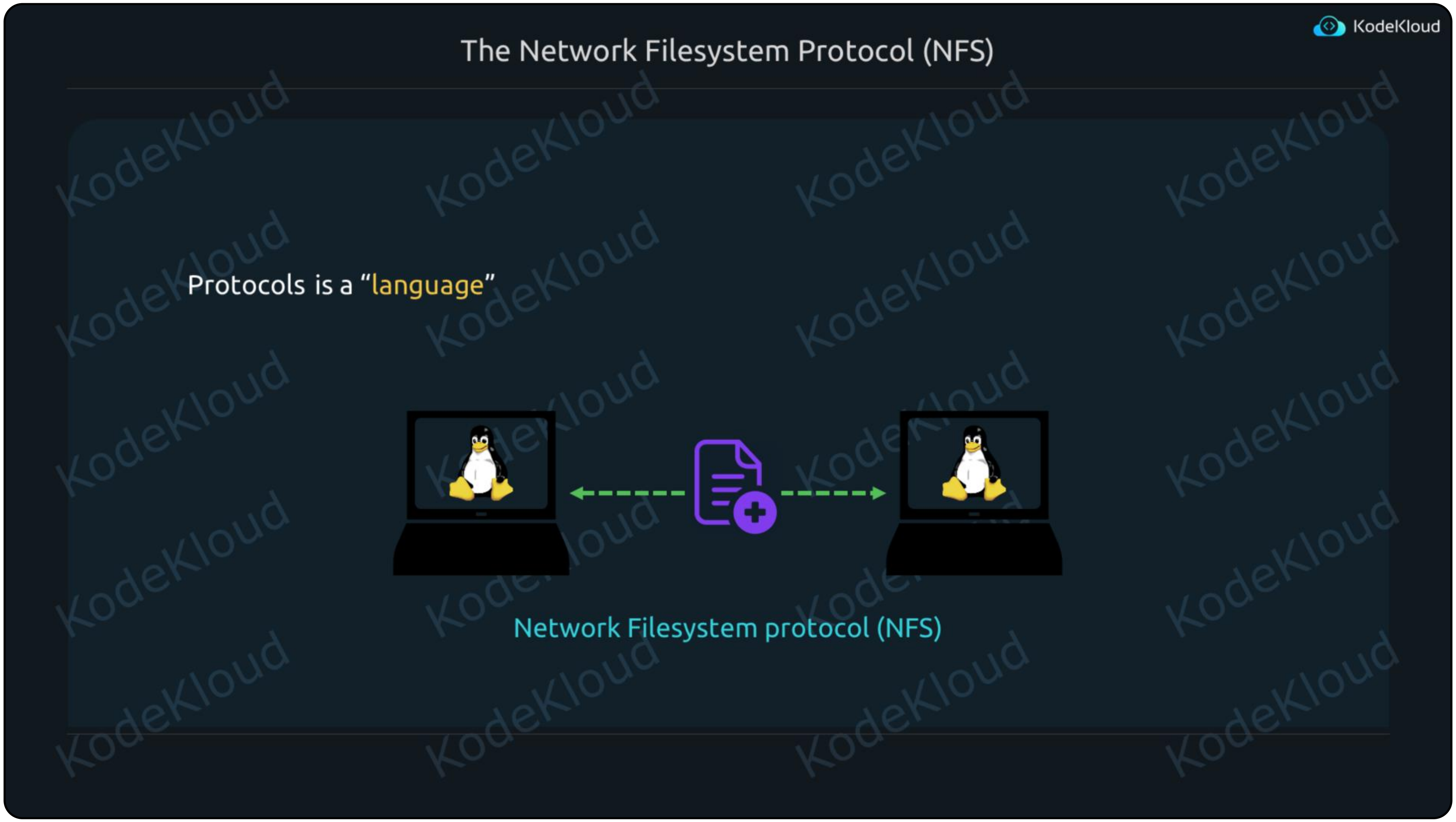
There are multiple protocols that allow us to access remote filesystems.



The Network Filesystem Protocol (NFS)

Protocol is a “language”

A protocol is, in a way, a "language" that a client and server use to communicate. They need to "speak" the same language so that they can understand each other, negotiate a connection, and transfer data.

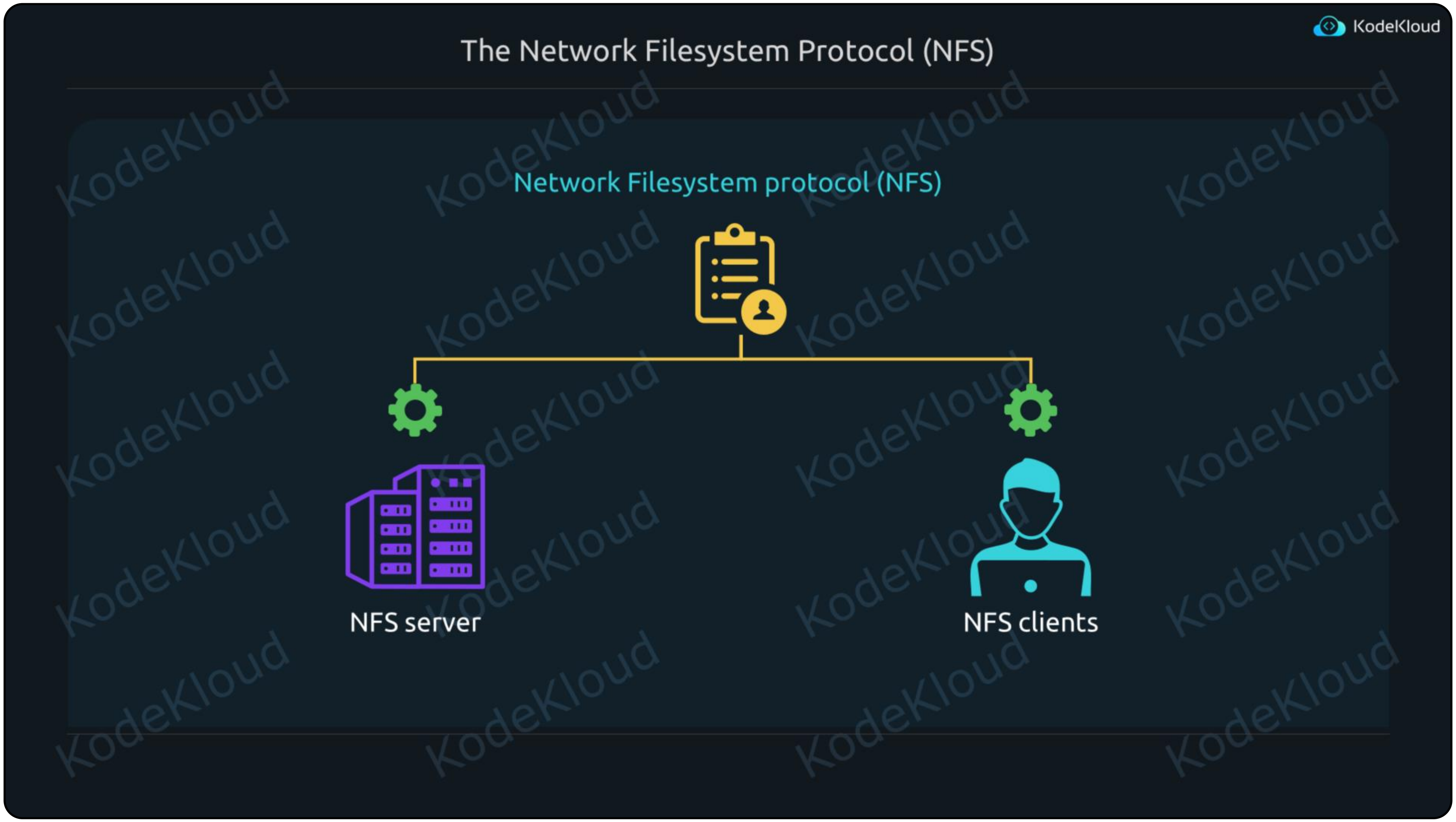

The Network Filesystem Protocol (NFS)

Protocols is a “language”

Linux supports many protocols for the purpose of sharing filesystems. But for sharing data between two Linux computers, the Network Filesystem protocol is most often used. This is also abbreviated as NFS.

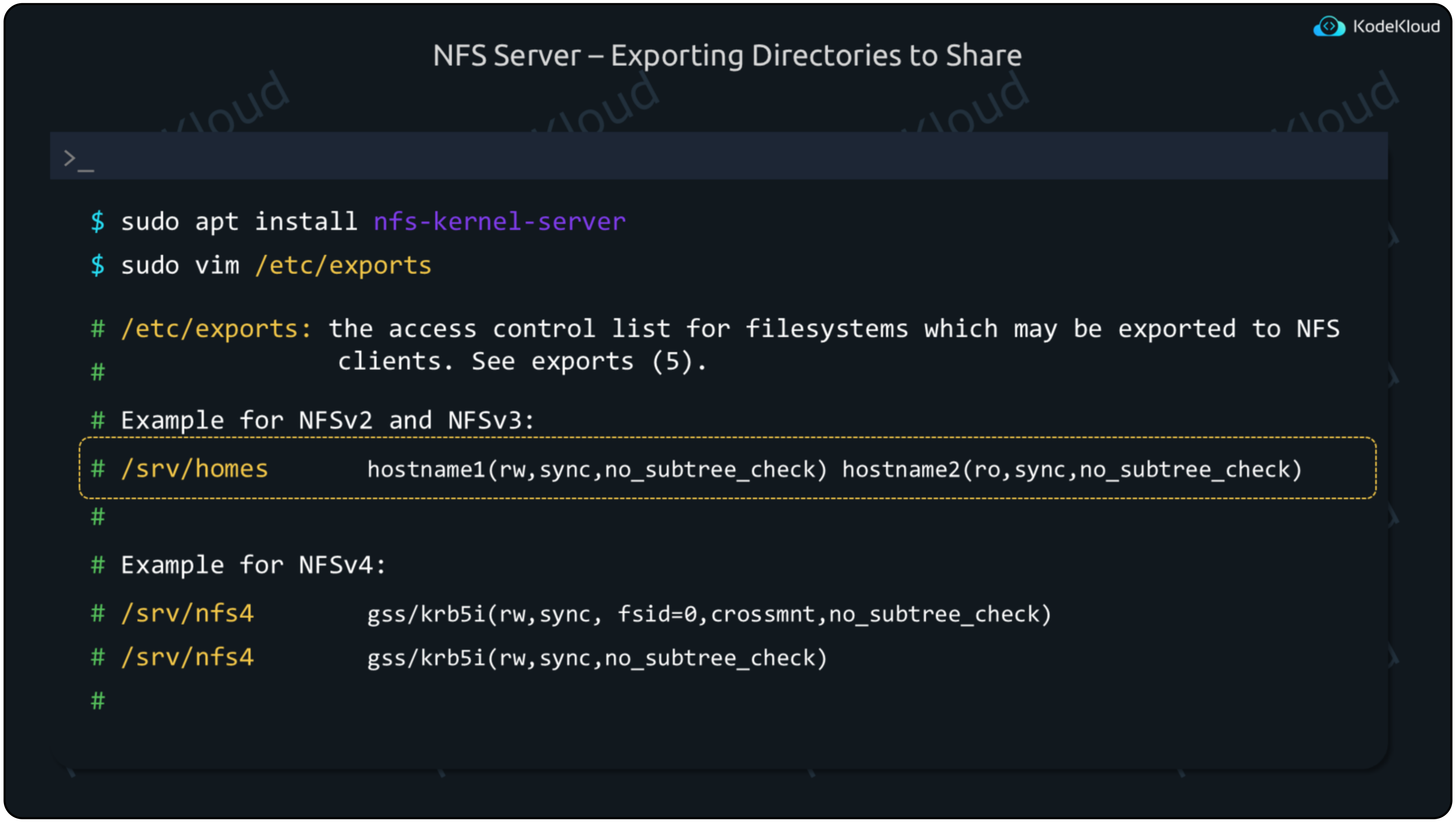
Let's see how we can use NFS on Linux.

The Network Filesystem Protocol (NFS)



There are two parts to using NFS: setting up the NFS server, and then setting up the NFS clients. On the server, we configure everything to allow it to share a filesystem with the world. And on the client-sides, we configure everything to allow them to mount the remote filesystem from that server.

Let's jump right into that first part, setting up the NFS server. Then we'll explore how to use NFS on the client side as well.



NFS Server – Exporting Directories to Share

```
$ sudo apt install nfs-kernel-server
$ sudo vim /etc/exports

# /etc/exports: the access control list for filesystems which may be exported to NFS
#               clients. See exports (5).

# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync, fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4        gss/krb5i(rw,sync,no_subtree_check)
#
```

On the server side, the one we're sharing data from, we'll first need to install the "nfs-kernel-server" package. So we'll enter this command:

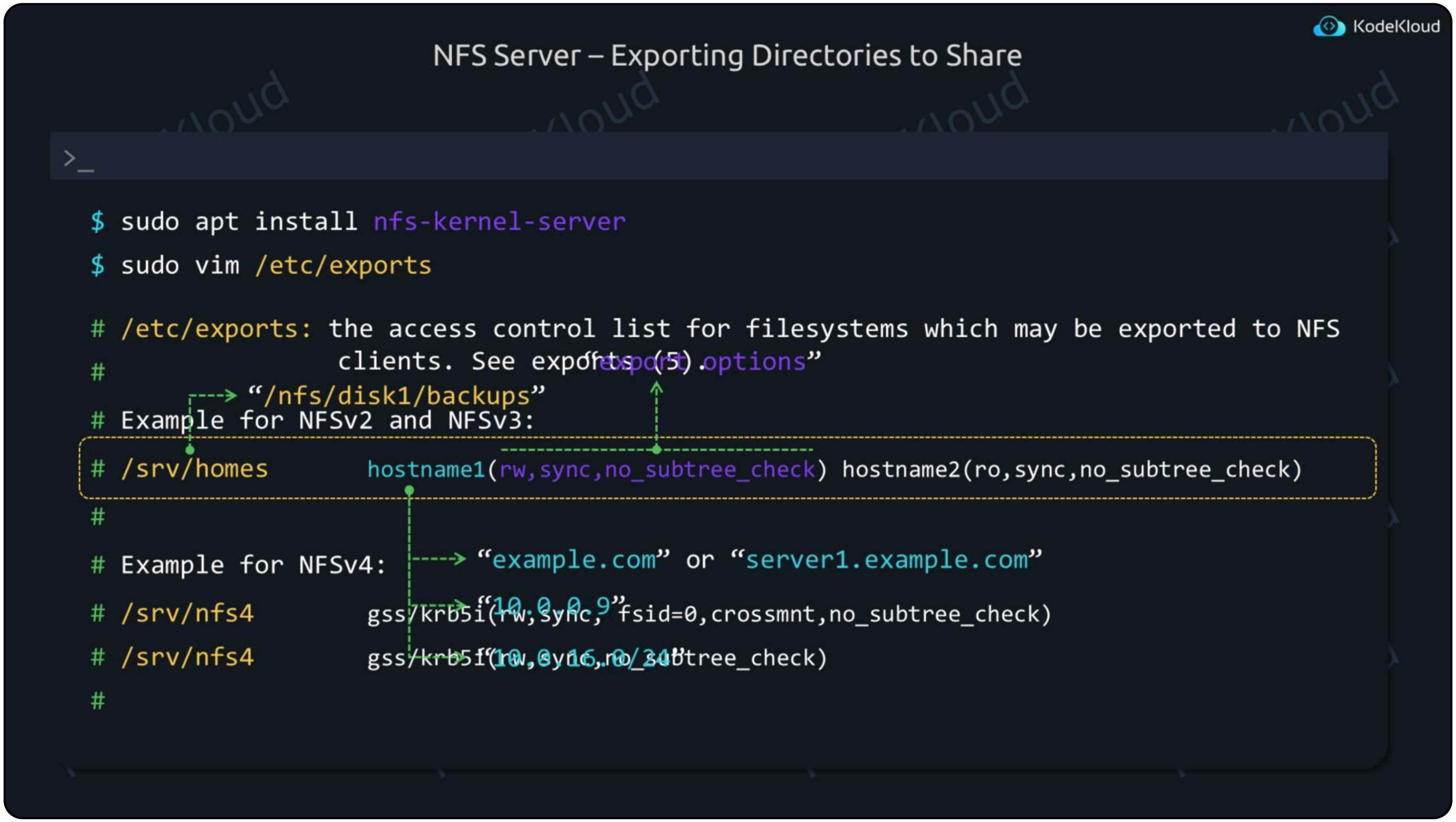
sudo apt install nfs-kernel-server

Next, we need to tell our NFS server what filesystems, or directories we want to share. And this is fairly straightforward. All we need to do is edit the "/etc/exports" file.

If we open it up for editing, with:

```
sudo vim /etc/exports
```

we'll see a few commented lines which are very helpful.



NFS Server – Exporting Directories to Share

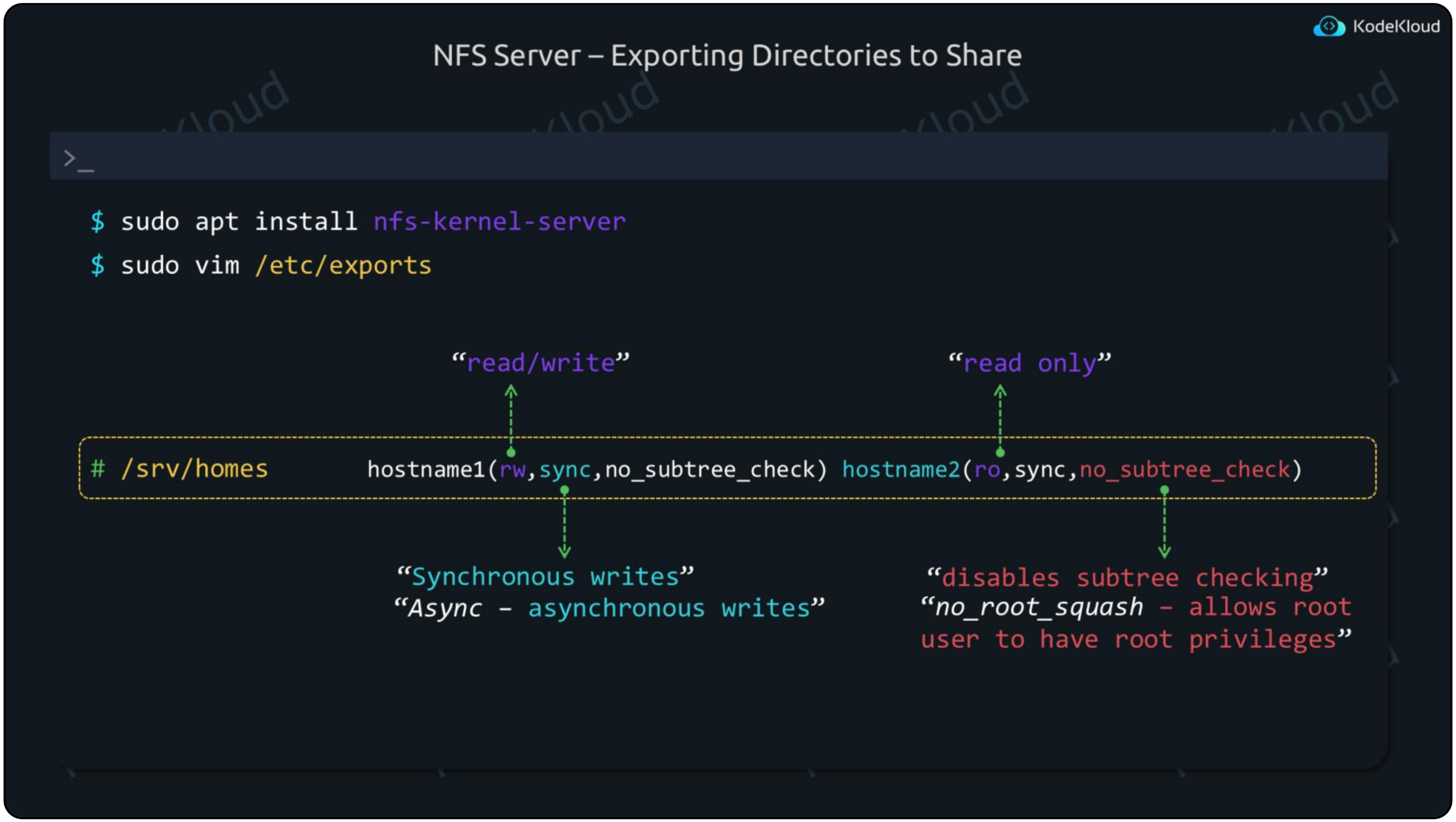
```
$ sudo apt install nfs-kernel-server
$ sudo vim /etc/exports

# /etc/exports: the access control list for filesystems which may be exported to NFS
#                 clients. See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4        gss/krb5i(rw,sync,no_subtree_check)
#
```

These show us a few examples, teaching us how to define our shares. From the highlighted line, we can see the basic structure:

First, we type the path to the directory we want to share. It can be something like "/srv/homes", or "/nfs/disk1/backups", or whatever we might want to use here. Usually, we'll want to point at the path where a filesystem is mounted. But we can also point to a subdirectory belonging to that filesystem.

Next, we need to choose who, or what can access this share. In other words, we specify which NFS clients should be allowed to use this remote filesystem. In this example, we can see "hostname1". That's the hostname of a computer on our network. These hostnames can potentially take more complex forms as well, such as "example.com" or "server1.example.com". Instead of hostnames, we can also use IP addresses here. So we could replace "hostname1" in this highlighted line, with something like "10.0.0.9", or whatever the IP address of the computer accessing this share would be. And there's a third option as well. If we want to allow an entire range of IPs to access this share, we can use the CIDR notation, which we learned about in our networking lessons. For example, if we want to make this share accessible to any computer with an IP address between 10.0.16.1 all the way to 10.0.16.255, we can type "10.0.16.0/24" here.

Finally, between parentheses, we include the export options. In this case, these are: "rw, sync, and no_subtree_check".



NFS Server – Exporting Directories to Share

```
$ sudo apt install nfs-kernel-server
$ sudo vim /etc/exports
```

```
# /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
```

"read/write"
"read only"
"Synchronous writes"
"Async – asynchronous writes"
"disables subtree checking"
"no_root_squash – allows root user to have root privileges"

What's interesting in this highlighted line is that we see we can enumerate multiple hosts that should be able to access this share. And we can do it on the same line. In this case, we can notice the "/srv/homes" directory is also shared with a second host, called "hostname2". And this includes slightly different export options. Notably, this uses the "ro" option instead of "rw". Now let's explore what these export options mean.

The important options to know about are rw, ro, sync, async, no_subtree_check, and no_root_squash.

rw stands for read/write. This allows clients mounting this share remotely, to both read, and write to this filesystem, or subdirectory.

ro stands for read-only. This allows clients to read, but not write.

sync is for synchronous writes of data. That is, NFS ensures that data written by a client is actually saved on the storage device, before reporting that the operation is successful.

async is for asynchronous writes. A client can issue multiple write requests. And NFS can report that the writes are completed, even before they are actually saved on the storage device. async allows clients to do things faster, but it does not guarantee that all changes are actually stored. So we can end up in a situation where NFS reports to the client that the write is complete, even though it isn't actually committed to the storage device yet. Then, if the server reboots unexpectedly, that write operation could be lost. sync is slower, but it guarantees that when a write is reported as being complete, it is actually stored on the device.

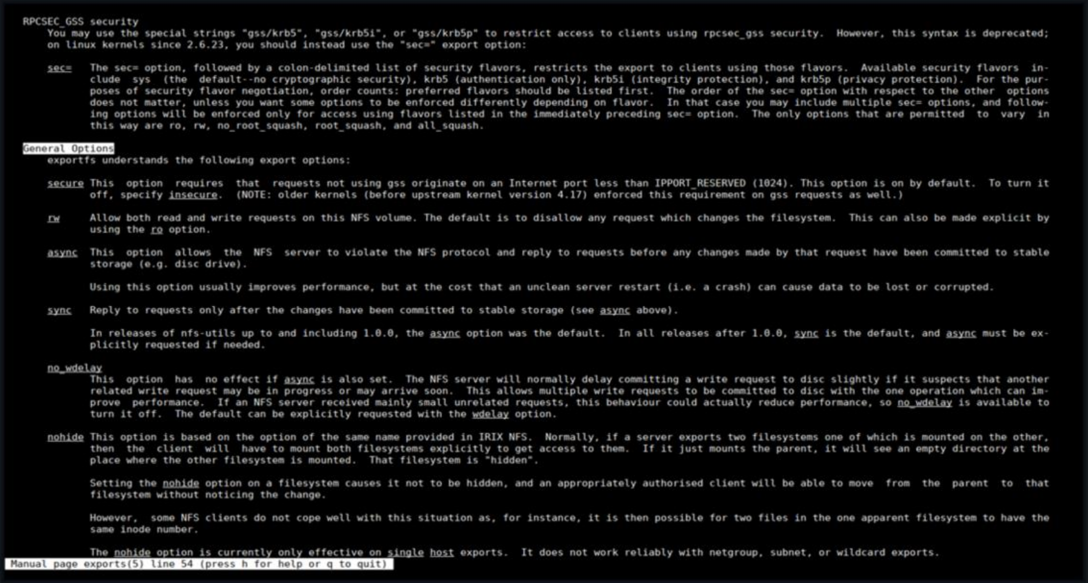
no_subtree_check disables subtree checking. Normally, with NFS, we'd want to export an entire filesystem. For example, say we have a backup disk on the NFS server. And we mount it in "/nfs/disk1". Then we'll export "/nfs/disk1" as an NFS share. But we can also export a subdirectory here, like "/nfs/disk1/backups/databases". If subtree checking would be active, the NFS server would always check if a requested file resides in this specific subdirectory that was exported. Which can cause a few issues when files are renamed, or moved to another subdirectory. Subtree checking is basically intended as an extra security check. But it has some drawbacks and a few performance implications. That's why, by default, "no_subtree_check" will be assigned to all NFS exports, even if we don't specify this option ourselves.

no_root_squash allows a root user on the NFS client to also have root privileges on a remote NFS share they mount. By default, NFS squashes root privileges. That is, if we are root on the client, and we mount an NFS share, then we won't be able to read and write as root on that remote filesystem. We'll be squashed, or downgraded, to regular user privileges instead. All read and write requests as "root" on the client will be mapped to a user called "nobody" on the server. This is a mild security measure that ensures root users on NFS clients cannot do anything they want on remote filesystems. If we want to deactivate this behavior, then we can use the "no_root_squash" option. This way, root on the client, will also be treated as the root user on the remote filesystem.

NFS Server – Exporting Directories to Share

```
>_
```

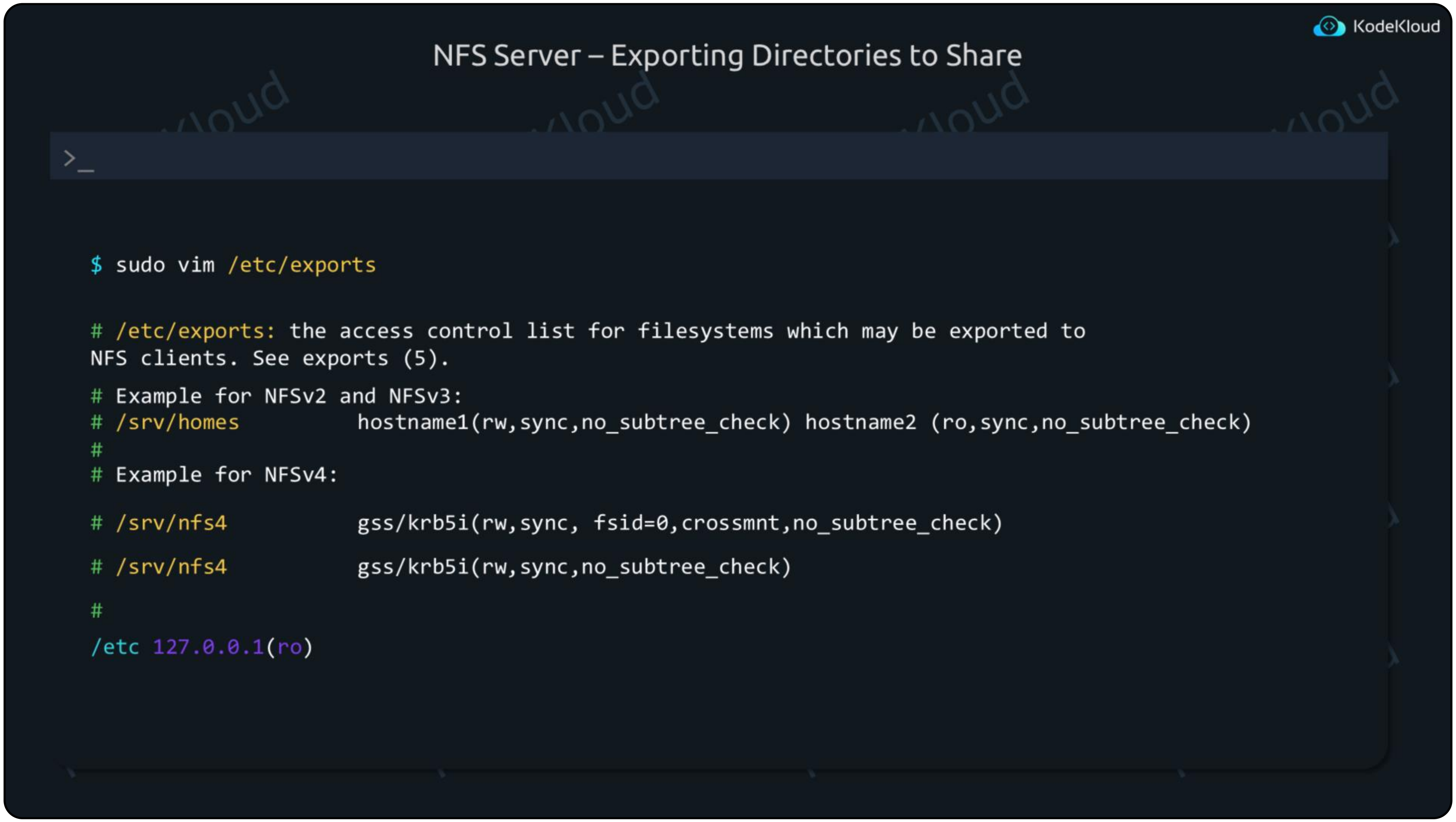
To see info about the /etc/exports file, use this command: man exports



```
RPCSEC_GSS security
    You may use the special strings "gss/krb5", "gss/krb5i", or "gss/krb5p" to restrict access to clients using rpcsec_gss security.  However, this syntax is deprecated;
    on linux kernels since 2.6.23, you should instead use the "sec=" export option:

    sec=   The sec= option, followed by a colon-delimited list of security flavors, restricts the export to clients using those flavors.  Available security flavors  in-
           clude  sys  (the  default--no cryptographic security), krb5 (authentication only), krb5i (integrity protection), and krb5p (privacy protection).  For the pur-
           poses of security flavor negotiation, order counts: preferred flavors should be listed first.  The order of the sec= option with respect to the other  options
           does not matter, unless you want some options to be enforced differently depending on flavor.  In that case you may include multiple sec= options, and follow-
           ing options will be enforced only for access using flavors listed in the immediately preceding sec= option.  The only options that are permitted  to  vary  in
           this way are ro, rw, no_root_squash, root_squash, and all_squash.

General Options
    exportfs understands the following export options:

    secure This  option  requires  that  requests not using gss originate on an Internet port less than IPPORT_RESERVED (1024). This option is on by default.  To turn it
           off, specify insecure.  (NOTE: older kernels (before upstream kernel version 4.17) enforced this requirement on gss requests as well.)

    rw     Allow both read and write requests on this NFS volume. The default is to disallow any request which changes the filesystem.  This can also be made explicit by
           using the ro option.

    async  This  option  allows  the  NFS  server to violate the NFS protocol and reply to requests before any changes made by that request have been committed to stable
           storage (e.g. disc drive).

           Using this option usually improves performance, but at the cost that an unclean server restart (i.e. a crash) can cause data to be lost or corrupted.

    sync   Reply to requests only after the changes have been committed to stable storage (see async above).

           In releases of nfs-utils up to and including 1.0.0, the async option was the default.  In all releases after 1.0.0, sync is the default, and async must be ex-
           plicitly requested if needed.

    no_wdelay
           This  option  has  no effect if async is also set.  The NFS server will normally delay committing a write request to disc slightly if it suspects that another
           related write request may be in progress or may arrive soon.  This allows multiple write requests to be committed to disc with the one operation which can im-
           prove  performance.  If an NFS server received mainly small unrelated requests, this behaviour could actually reduce performance, so no_wdelay is available to
           turn it off.  The default can be explicitly requested with the wdelay option.

    nohide This option is based on the option of the same name provided in IRIX NFS.  Normally, if a server exports two filesystems one of which is mounted on the other,
           then  the  client  will  have to mount both filesystems explicitly to get access to them.  If it just mounts the parent, it will see an empty directory at the
           place where the other filesystem is mounted.  That filesystem is "hidden".

           Setting the nohide option on a filesystem causes it not to be hidden, and an appropriately authorised client will be able to move  from  the  parent  to  that
           filesystem without noticing the change.

           However,  some NFS clients do not cope well with this situation as, for instance, it is then possible for two files in the one apparent filesystem to have the
           same inode number.

           The nohide option is currently only effective on single host exports.  It does not work reliably with netgroup, subnet, or wildcard exports.
 Manual page exports(5) line 54 (press h for help or q to quit)
```

To see information about the /etc/exports file, including other mount options and syntax specifications, we can use this command:

man exports



NFS Server – Exporting Directories to Share

```
$ sudo vim /etc/exports

# /etc/exports: the access control list for filesystems which may be exported to
NFS clients. See exports (5).
# Example for NFSv2 and NFSv3:
# /srv/homes          hostname1(rw,sync,no_subtree_check) hostname2 (ro,sync,no_subtree_check)
#
# Example for NFSv4:

# /srv/nfs4           gss/krb5i(rw,sync, fsid=0,crossmnt,no_subtree_check)

# /srv/nfs4           gss/krb5i(rw,sync,no_subtree_check)

#

/etc 127.0.0.1(ro)
```

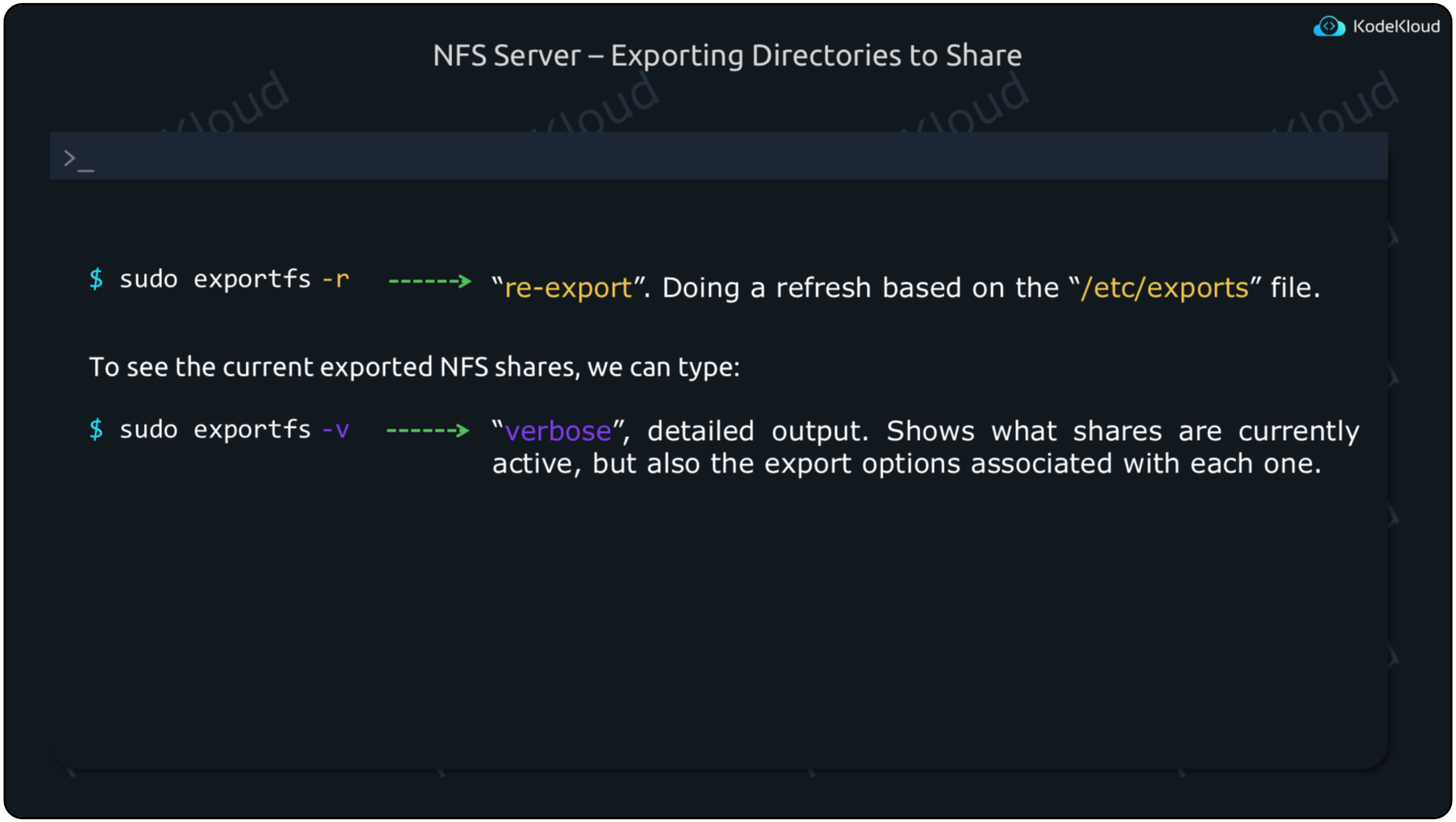
Now that we understand how to define an NFS share on our server, let's say we want to add our own line. What would we write if we wanted to share the "/etc" directory with a computer with the IP address: "127.0.0.1"? And if we also wanted that computer to only be able to read files from this share, but deny write access to it.

First, we'd open up the exports file with:

```
sudo vim /etc/exports
```

Then we'd write a line like this, at the end:

```
/etc 127.0.0.1(ro)
```



NFS Server – Exporting Directories to Share

```
>_

$ sudo exportfs -r   ------>
```

"re-export". Doing a refresh based on the "/etc/exports" file.

To see the current exported NFS shares, we can type:

```
$ sudo exportfs -v   ------>
```

"verbose", detailed output. Shows what shares are currently active, but also the export options associated with each one.

After modifying the exports file, we must also inform our NFS server about these changes. So it can actually start sharing what we defined here. To apply these changes, we use this command:

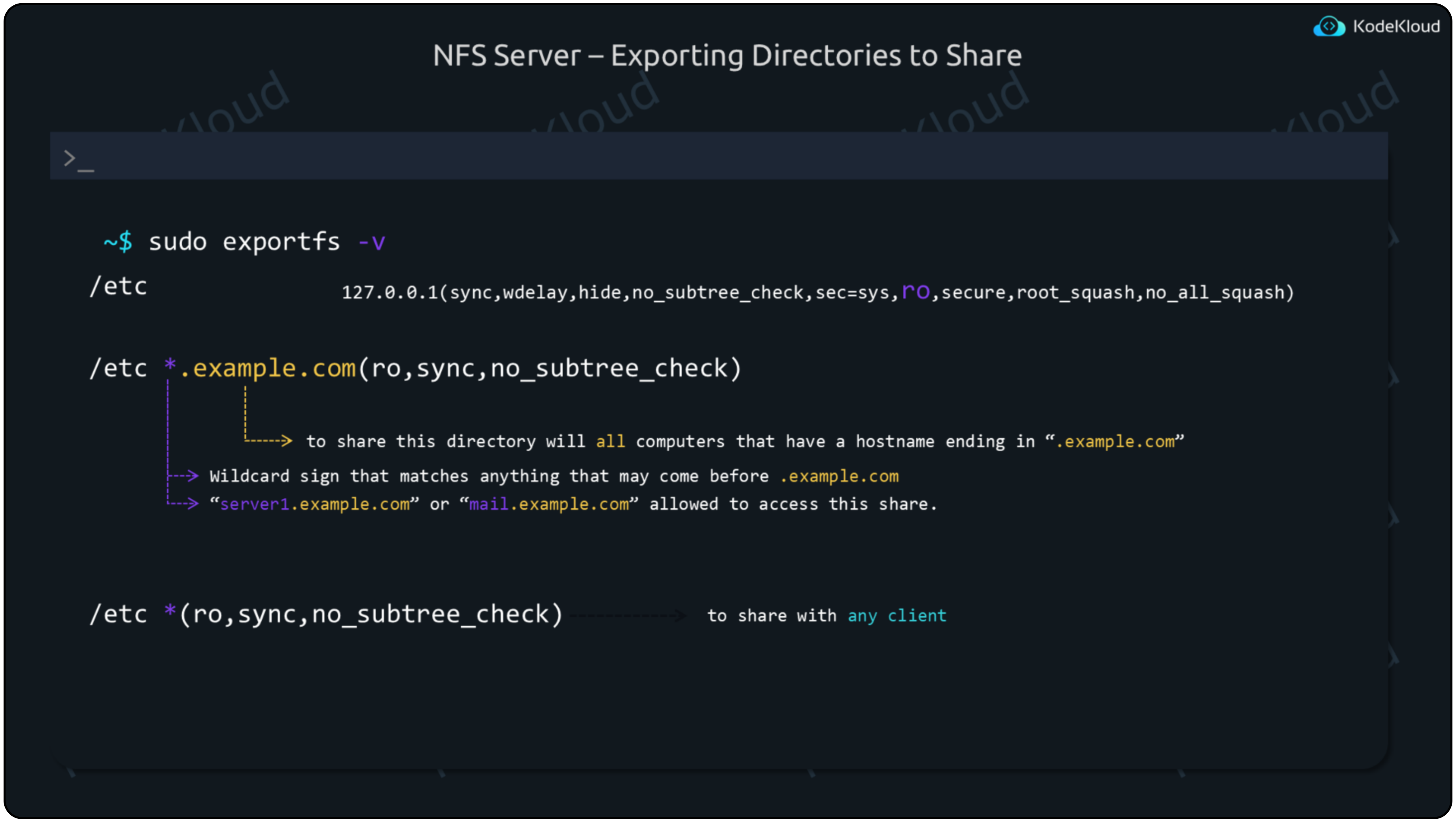
sudo exportfs -r

The -r option stands for re-export. You can think of it as doing a refresh, based on the "/etc/exports" file. It will share what is defined in that file, but also unshare whatever has been removed.

To see the current exported NFS shares, we can type:

```
sudo exportfs -v
```

-v stands for verbose, that is, detailed output. This will show us what shares are currently active, but also the export options associated with each one.


KodeKloud

NFS Server – Exporting Directories to Share

```
~$ sudo exportfs -v
/etc                127.0.0.1(sync,wdelay,hide,no_subtree_check,sec=sys,ro,secure,root_squash,no_all_squash)

/etc *.example.com(ro,sync,no_subtree_check)
         |
         └-----> to share this directory will all computers that have a hostname ending in ".example.com"
     ├--> Wildcard sign that matches anything that may come before .example.com
     └--> "server1.example.com" or "mail.example.com" allowed to access this share.


/etc *(ro,sync,no_subtree_check) ----------> to share with any client
```

Notice that in our case, we only used "ro" as our option. But there are many more options active for this share. That's because the NFS server will assign some options by default, even if they are not explicitly defined by the administrators in the /etc/exports file.

To wrap up this section, here are a few extra tips:

We can also use wildcards in the hostname field, in our "/etc/exports" file. For example, to share this directory with all computers that have a hostname ending in ".example.com" we could add a line like this:

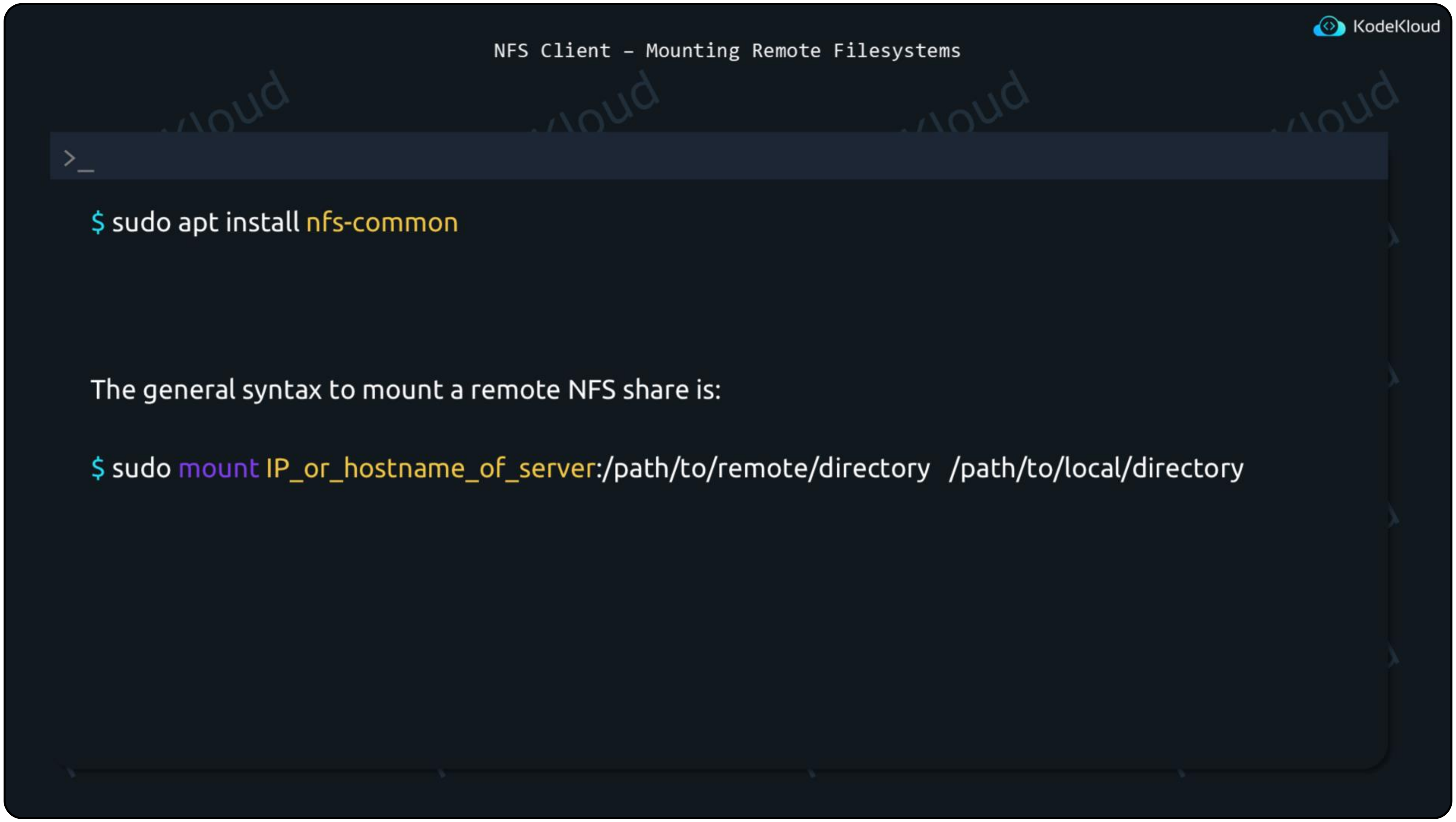
```
/etc *.example.com(ro,sync,no_subtree_check)
```

The asterisk "*" is the wildcard sign that matches anything that may come before .example.com. So clients with hostnames like "server1.example.com" or "mail.example.com" will be allowed to access this share.

And to share with any client, we can simply use the asterisk sign alone:

```
/etc *(ro,sync,no_subtree_check)
```

It's important to remember to not add any extra spaces between the asterisk and the mount options. Same rule applies when you use a hostname, or an IP address. The field where we define the clients allowed to access this share should be glued together with the open parenthesis that includes the options.



NFS Client – Mounting Remote Filesystems

```
$ sudo apt install nfs-common
```

The general syntax to mount a remote NFS share is:

```
$ sudo mount IP_or_hostname_of_server:/path/to/remote/directory  /path/to/local/directory
```

Now let's move on to the client side. Things are much simpler here.

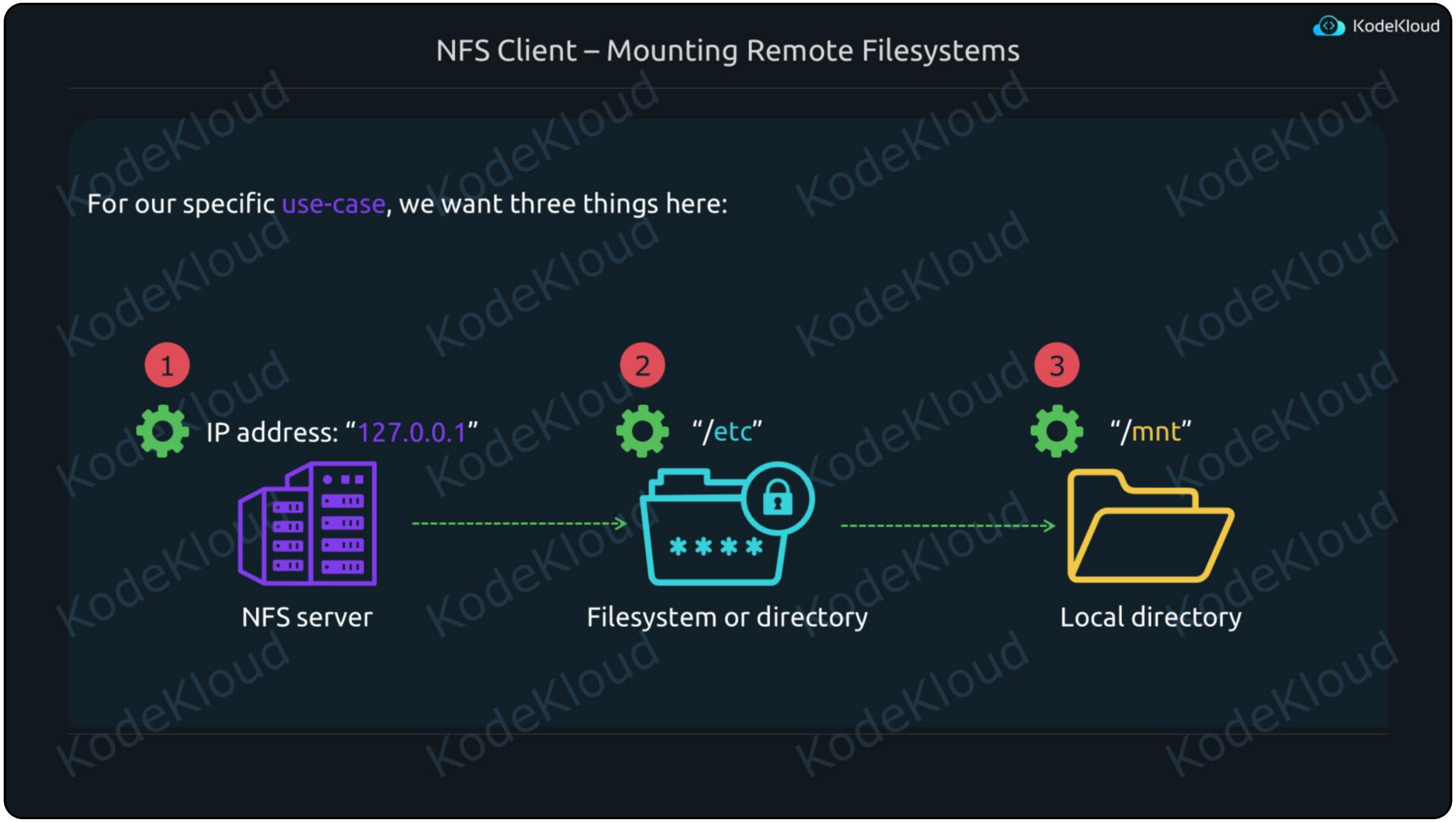
First, we need to ensure we have the utilities necessary to mount NFS shares. These utilities are included in the "nfs-common" package, which we can install with:

sudo apt install nfs-common

Now that we already shared the /etc directory on our server, how would we mount it from a remote client? Well, we already learned about the mount command in previous lessons. All we need to do is adapt it to this use case.

The general syntax to mount a remote NFS share is:

```
sudo mount IP_or_hostname_of_server:/path/to/remote/directory /path/to/local/directory
```

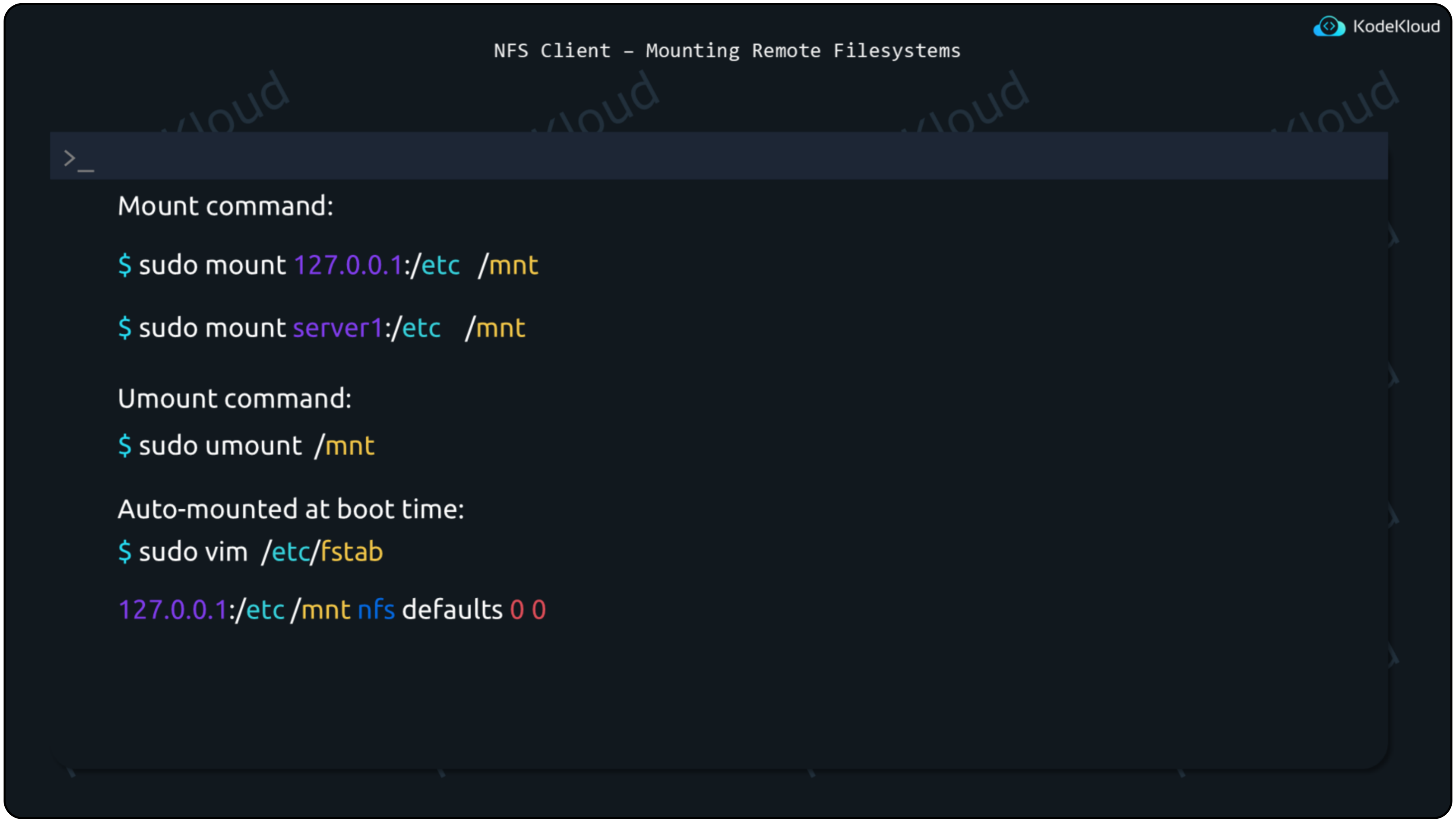


NFS Client – Mounting Remote Filesystems

For our specific use-case, we want three things here:

For our specific use-case, we want three things here:

1. Mount something from an NFS server with IP address "**127.0.0.1**".
2. Mount the remote "**/etc**" filesystem, or directory, from that server.
3. And, finally, mount this into our local directory, "**/mnt**".



NFS Client – Mounting Remote Filesystems

Mount command:

```
$ sudo mount 127.0.0.1:/etc  /mnt

$ sudo mount server1:/etc   /mnt
```

Umount command:

```
$ sudo umount  /mnt
```

Auto-mounted at boot time:

```
$ sudo vim  /etc/fstab

127.0.0.1:/etc /mnt nfs defaults 0 0
```

Which means our command should be:

sudo mount 127.0.0.1:/etc /mnt

Basically, all we did was add an extra part to our regular mount command, where we specified the location of our remote server, followed by the ":" colon symbol.

If we'd want to use a hostname, like "server1" instead, we would type something like:

```
sudo mount server1:/etc /mnt
```

The rest of the commands are similar to what we've learned. For example, to unmount this NFS share, all we need to do is type:

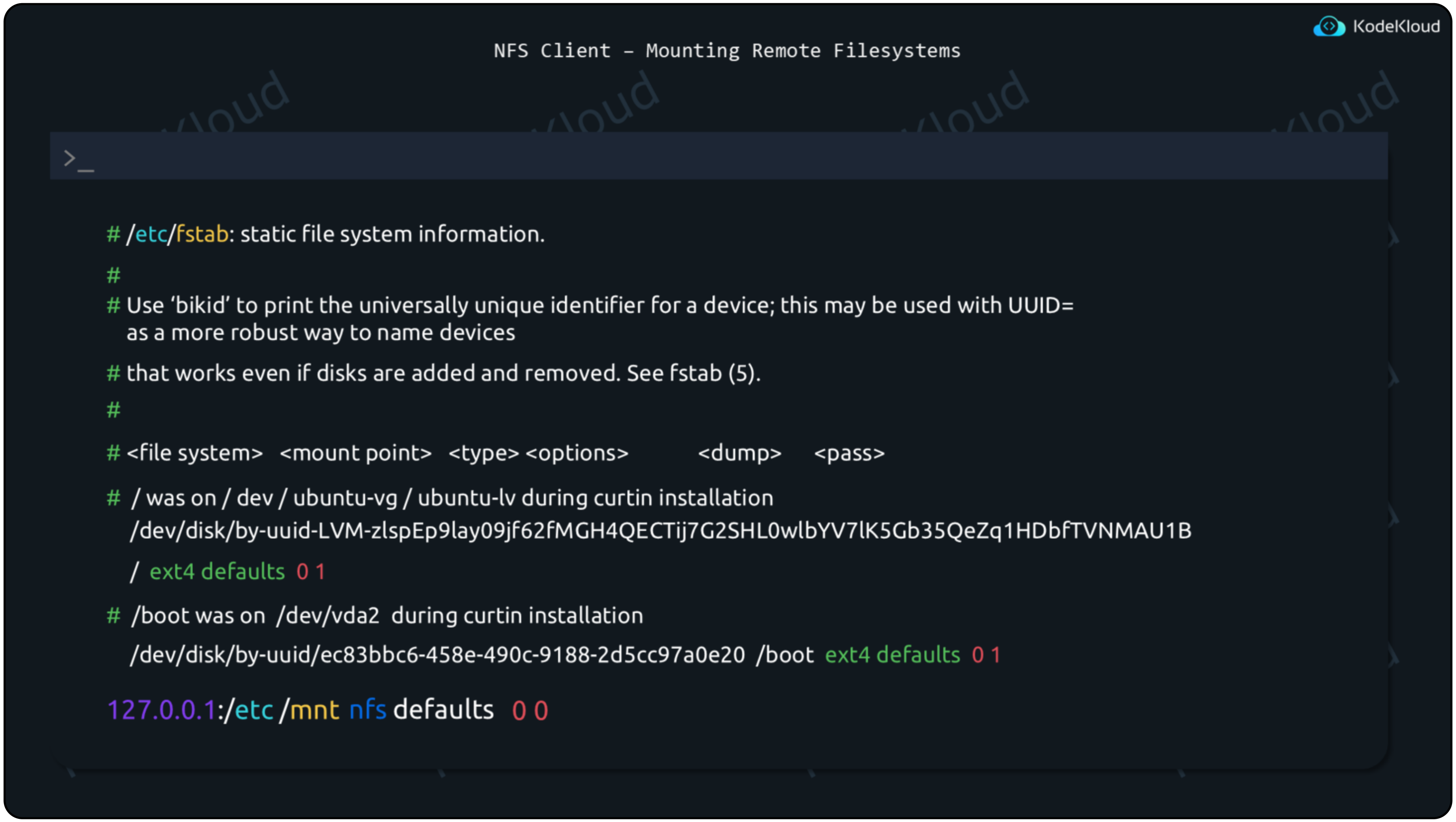
```
sudo umount /mnt
```

Finally, if we'd want this NFS share to be auto-mounted at boot time, we'd go through familiar steps. First, we'd open the /etc/fstab file for editing:

```
sudo vim /etc/fstab
```

Then we'd add a line like this:

```
127.0.0.1:/etc /mnt nfs defaults 0 0
```



NFS Client – Mounting Remote Filesystems

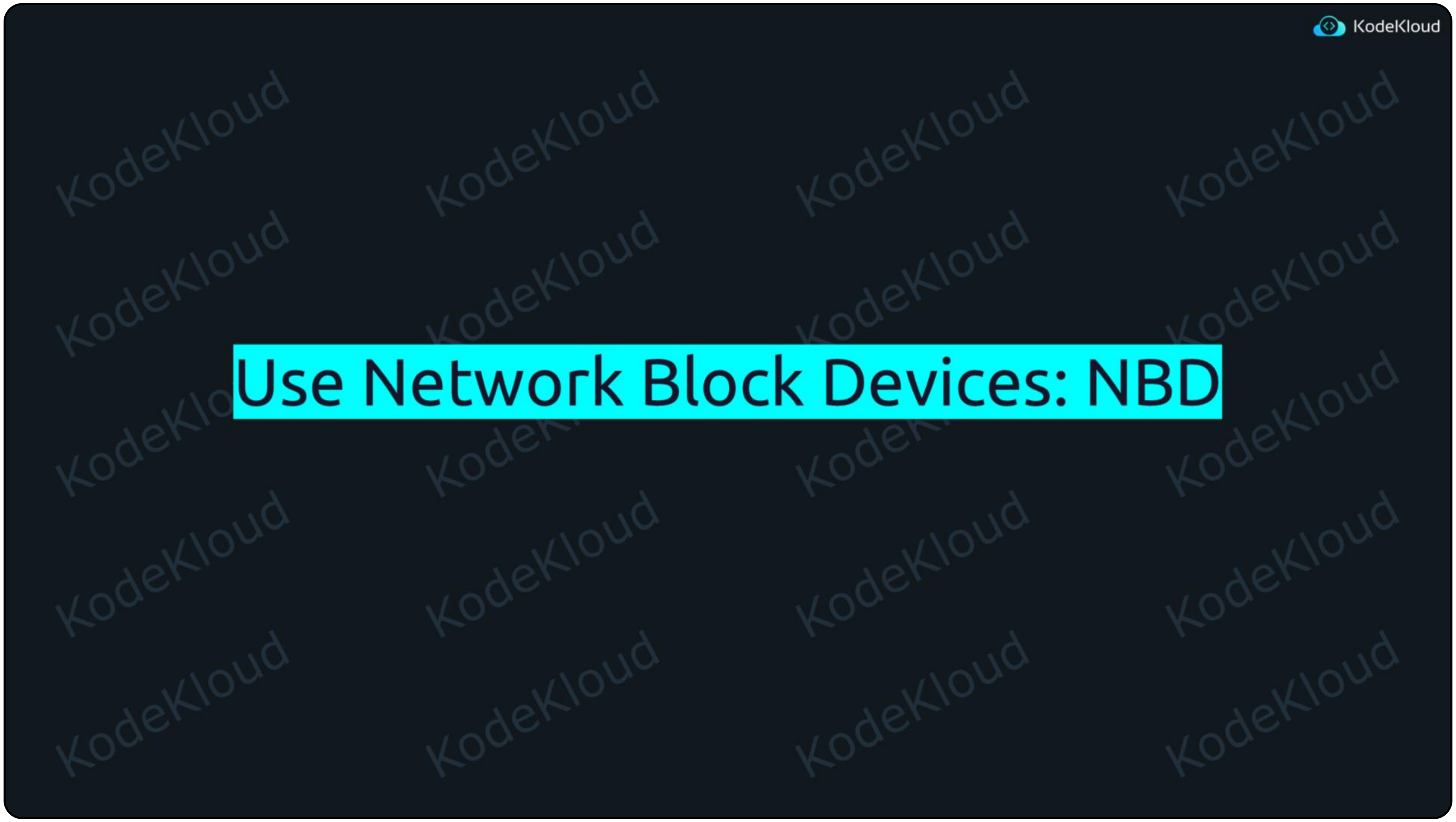
```
# /etc/fstab: static file system information.
#
# Use 'bikid' to print the universally unique identifier for a device; this may be used with UUID=
  as a more robust way to name devices
# that works even if disks are added and removed. See fstab (5).
#
# <file system>  <mount point>  <type> <options>        <dump>    <pass>
#  / was on / dev / ubuntu-vg / ubuntu-lv during curtin installation
  /dev/disk/by-uuid-LVM-zlspEp9lay09jf62fMGH4QECTij7G2SHL0wlbYV7lK5Gb35QeZq1HDbfTVNMAU1B
  /  ext4 defaults  0 1
#  /boot was on  /dev/vda2  during curtin installation
  /dev/disk/by-uuid/ec83bbc6-458e-490c-9188-2d5cc97a0e20  /boot  ext4 defaults  0 1
127.0.0.1:/etc /mnt nfs defaults   0 0
```

The only differences compared to what we did in fstab in earlier lessons, is that the source of this mount specifies an IP address and a remote directory. And in the field where we normally specify "ext4" or "xfs" as the filesystem type, we entered "nfs".

The rest is similar. Where we specify mount options, we can write "defaults" to let NFS autoconfigure its mount options. Of course, if you need specific mount options you can just enumerate them here just like we did in our previous lessons.

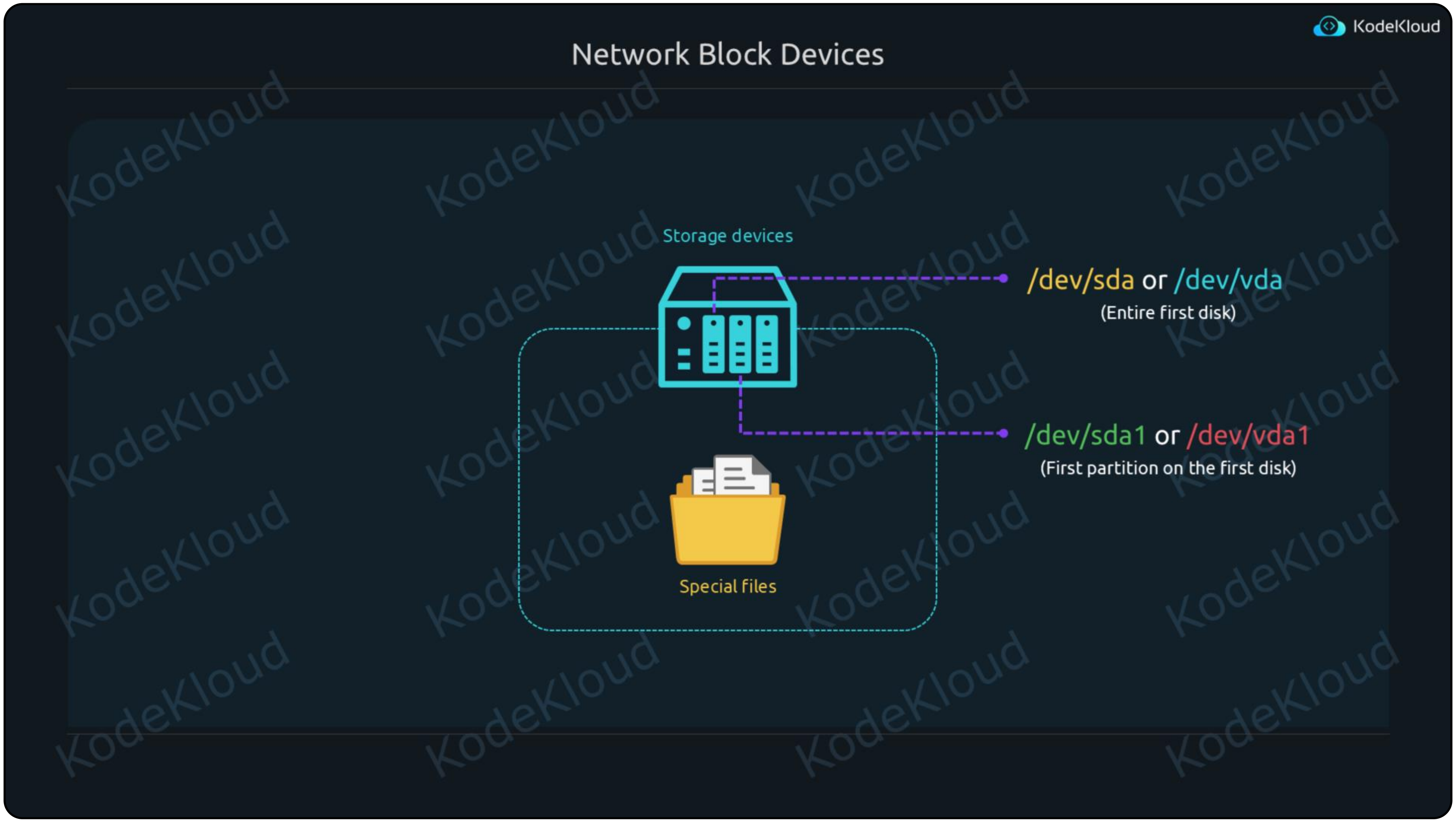
Multiple values separated by commas. Finally, since this directory is not local, we don't need to check the filesystem for errors. So our last two fields should be "0 0".

This covers the essentials of creating NFS servers, and mounting remote filesystems on NFS clients. Let's jump into our next lesson.



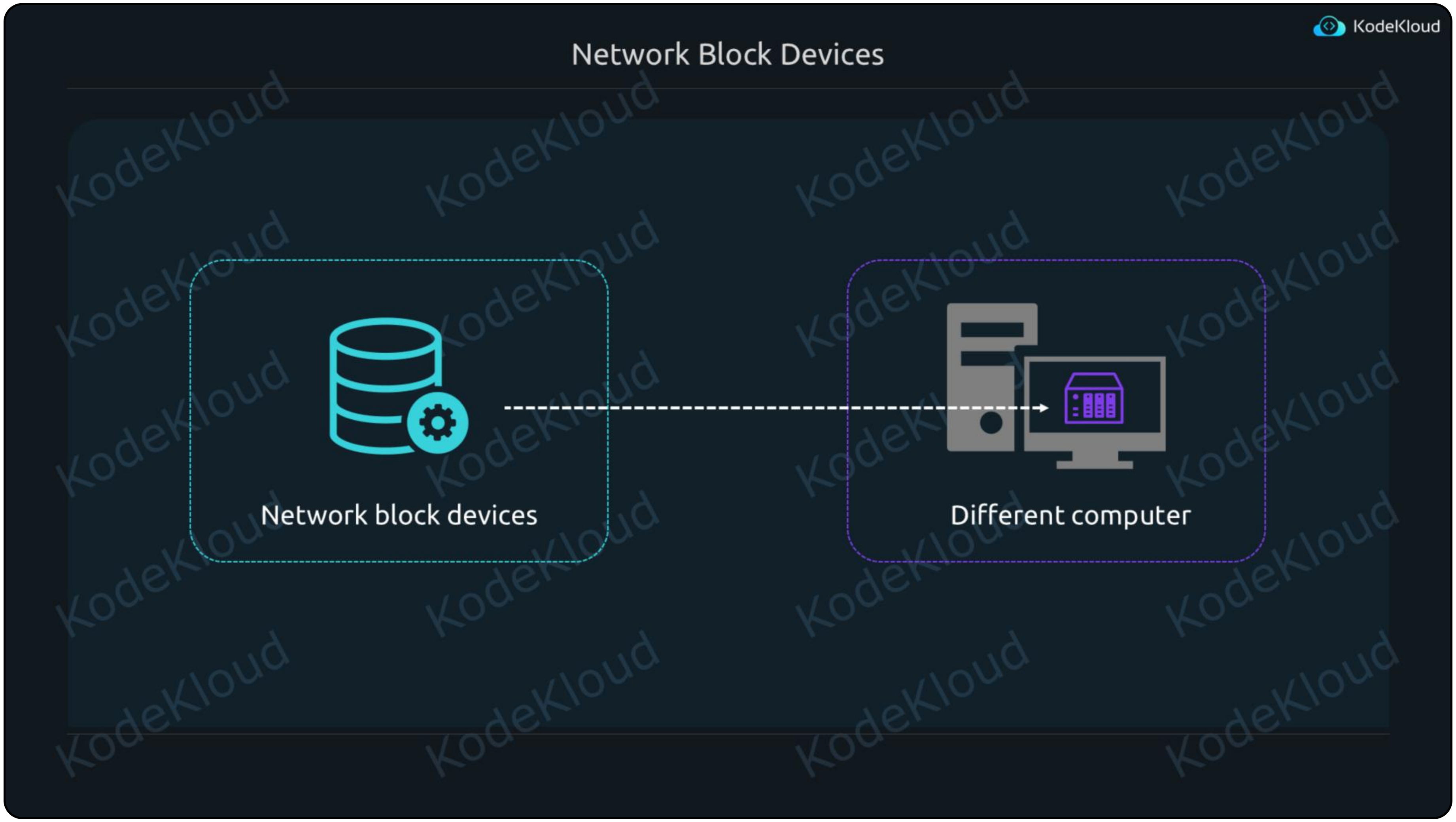
Use Network Block Devices: NBD

In this lesson we'll explore how to mount block devices from remote servers.



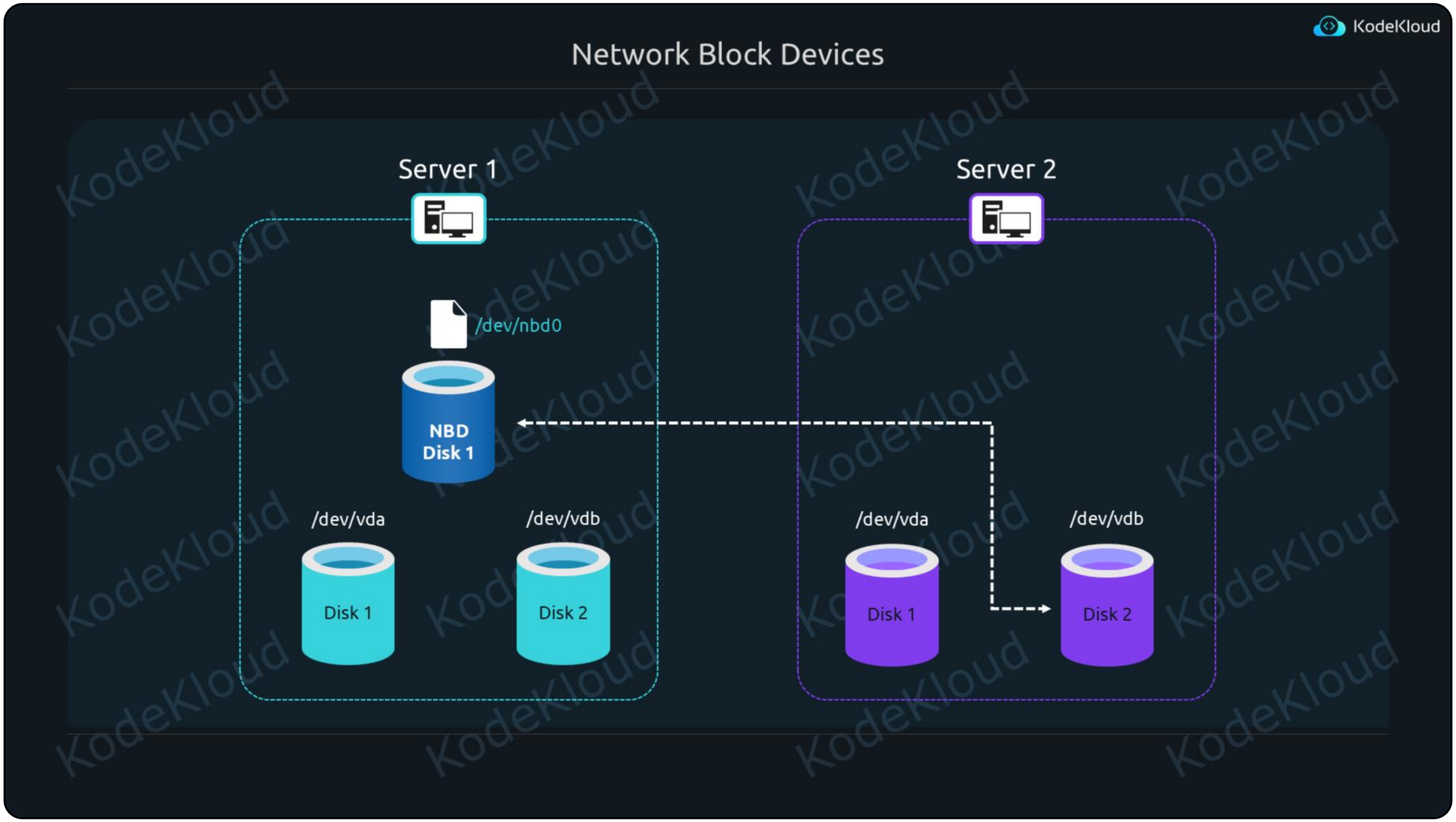
Network Block Devices

In previous lessons we learned about special files that can reference our storage devices. For example, /dev/sda, or /dev/vda points to our entire first storage device on the system. And /dev/sda1, or /dev/vda1 points to the first partition on the first disk. These are block special files that reference block devices.



Network Block Devices

Network block devices do a similar thing. But instead of pointing to storage devices plugged into our system, they point to storage devices on an entirely different computer.

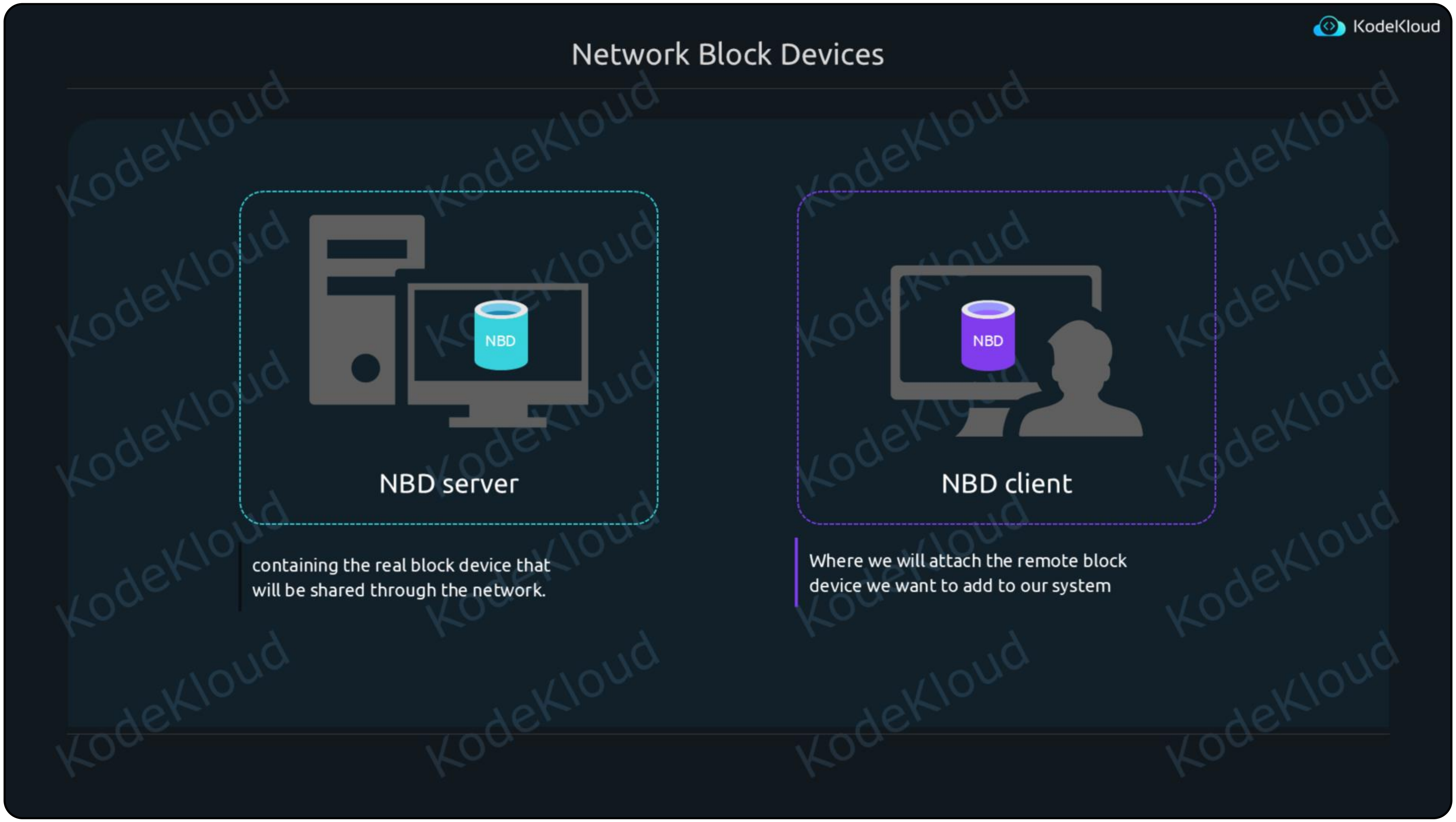


For example, imagine this setup:

We have one server with two storage devices referenced by the /dev/vda and /dev/vdb block special files. Then we have another server with two storage devices, referenced by similar files. With Network Block Devices, we can essentially add a third disk to our first server.

NBD utilities create a special file called /dev/nbd0. As far as applications are concerned, /dev/nbd0 looks and behaves just like any other block device. But whenever something reads or writes to this device, the requests are sent to the real block device on Server 2. So any write request to /dev/nbd0 is redirected to /dev/vdb on Server 2.

But things will be even easier to understand if we look at this in practice. So let's see how we can actually use Network Block Devices, also called NBD when abbreviated.



Network Block Devices

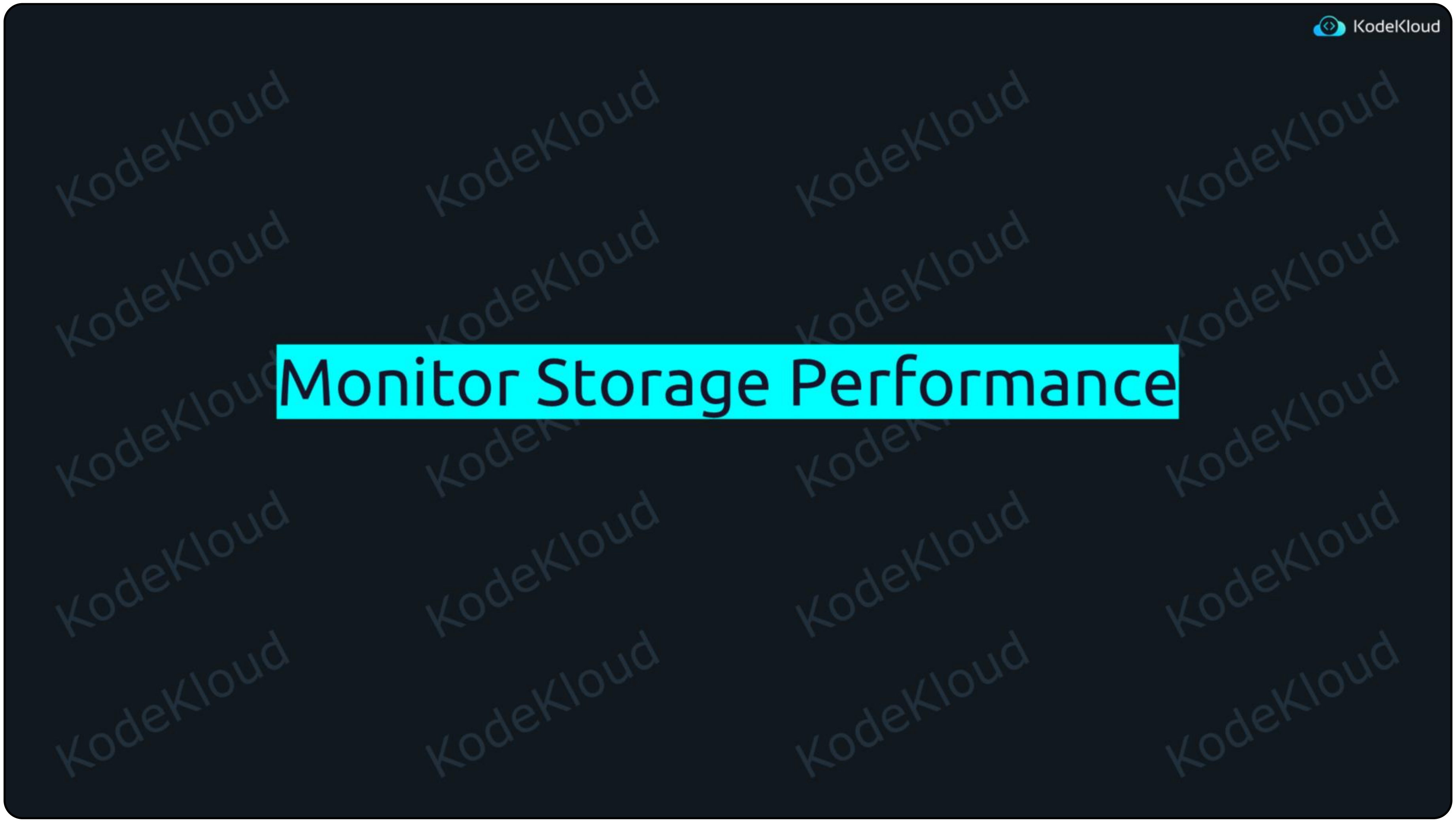
containing the real block device that will be shared through the network.

Where we will attach the remote block device we want to add to our system

When we deal with NBD, we have two locations that need to be configured:

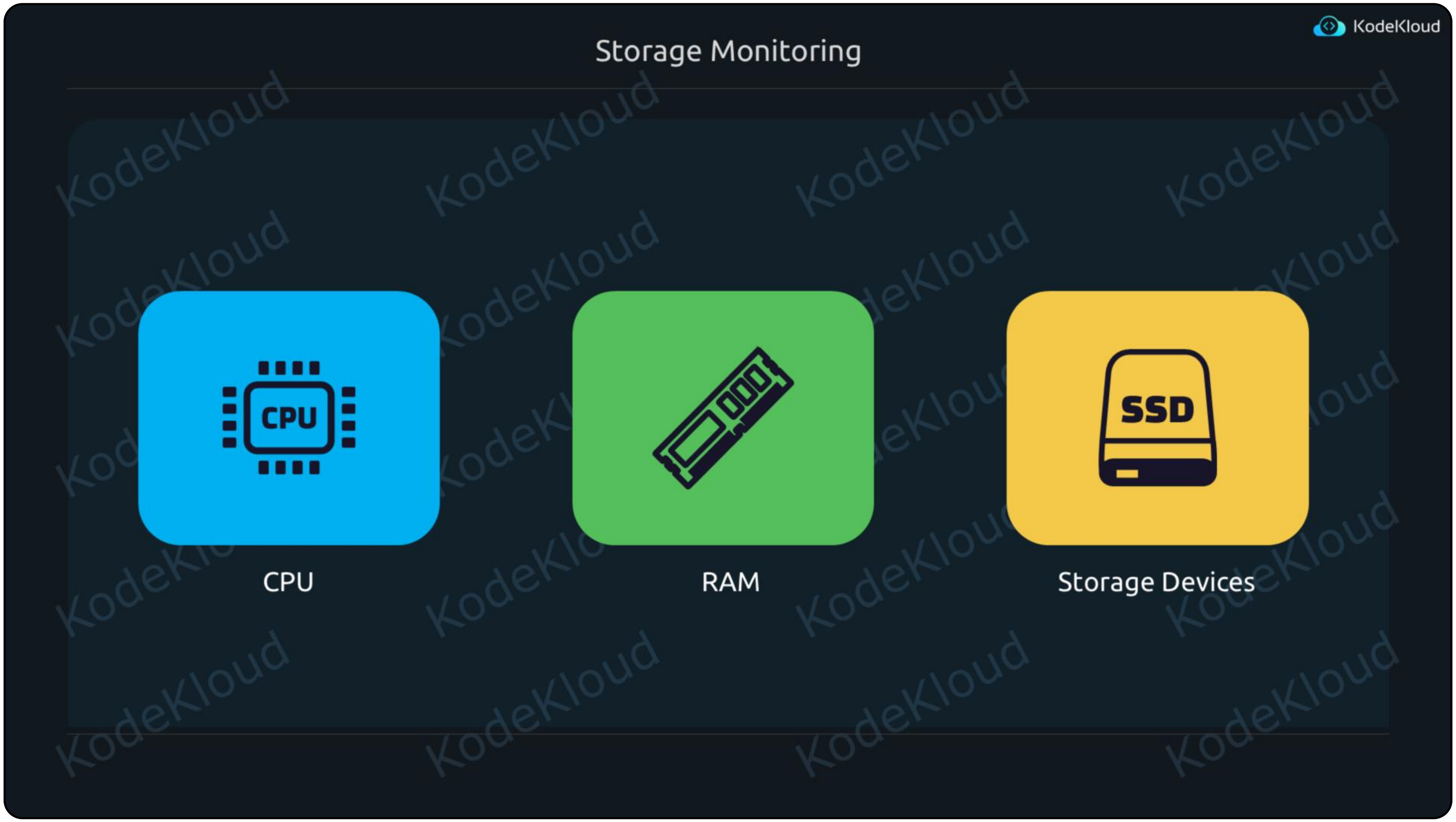
The NBD server, containing the real block device that will be shared through the network.
The NBD client, where we will attach the remote remote block device we want to add to our system.

Visit www.kodekloud.com to discover more.



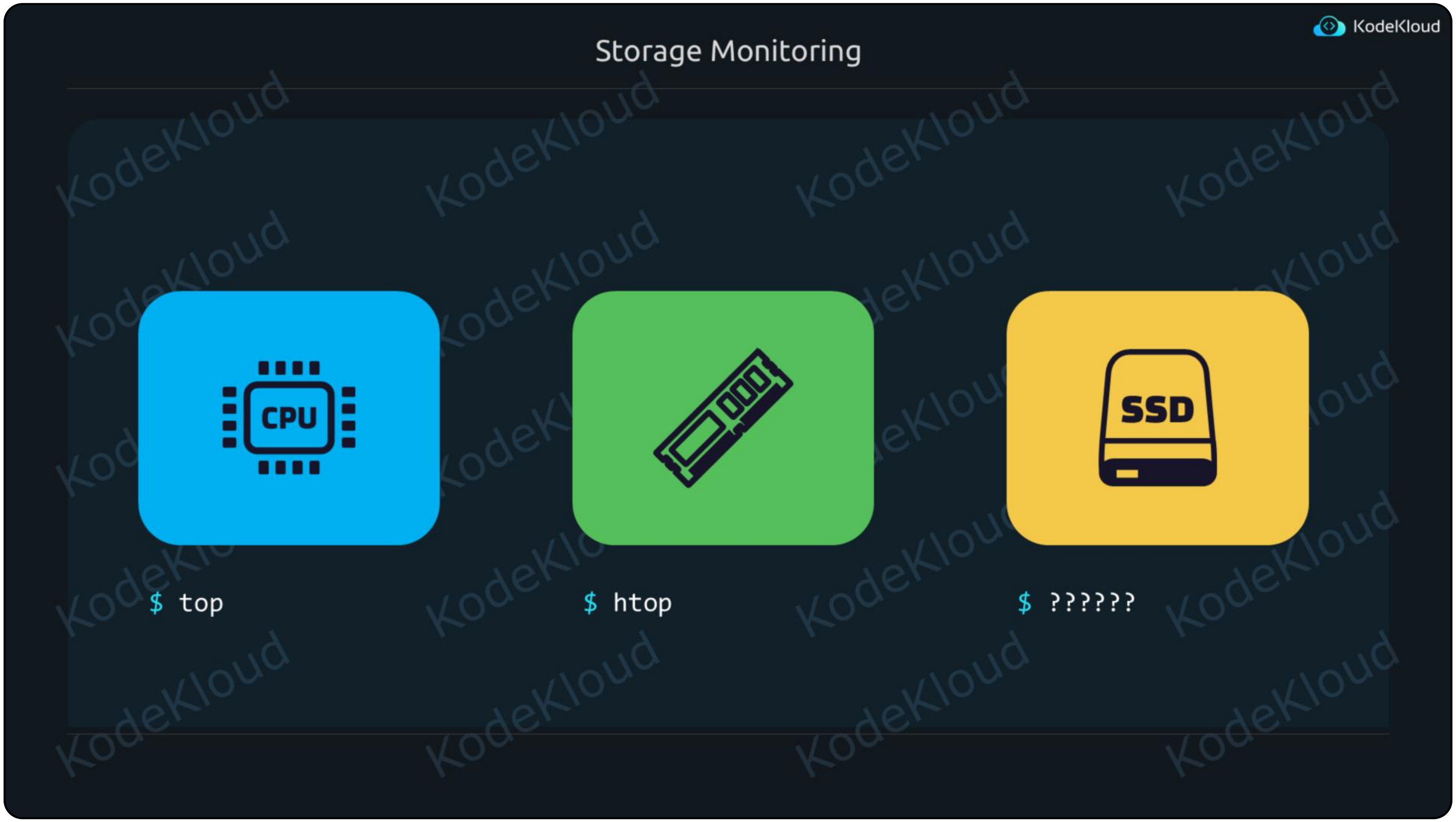
Monitor Storage Performance

In this lesson we'll look at how we can monitor the performance of our storage devices.



Storage Monitoring

Just like CPU and RAM, storage devices have their limits. If a process is overusing the CPU, keeping it at 100%, the system will slow down considerably. And the same can happen if a storage device is overused.



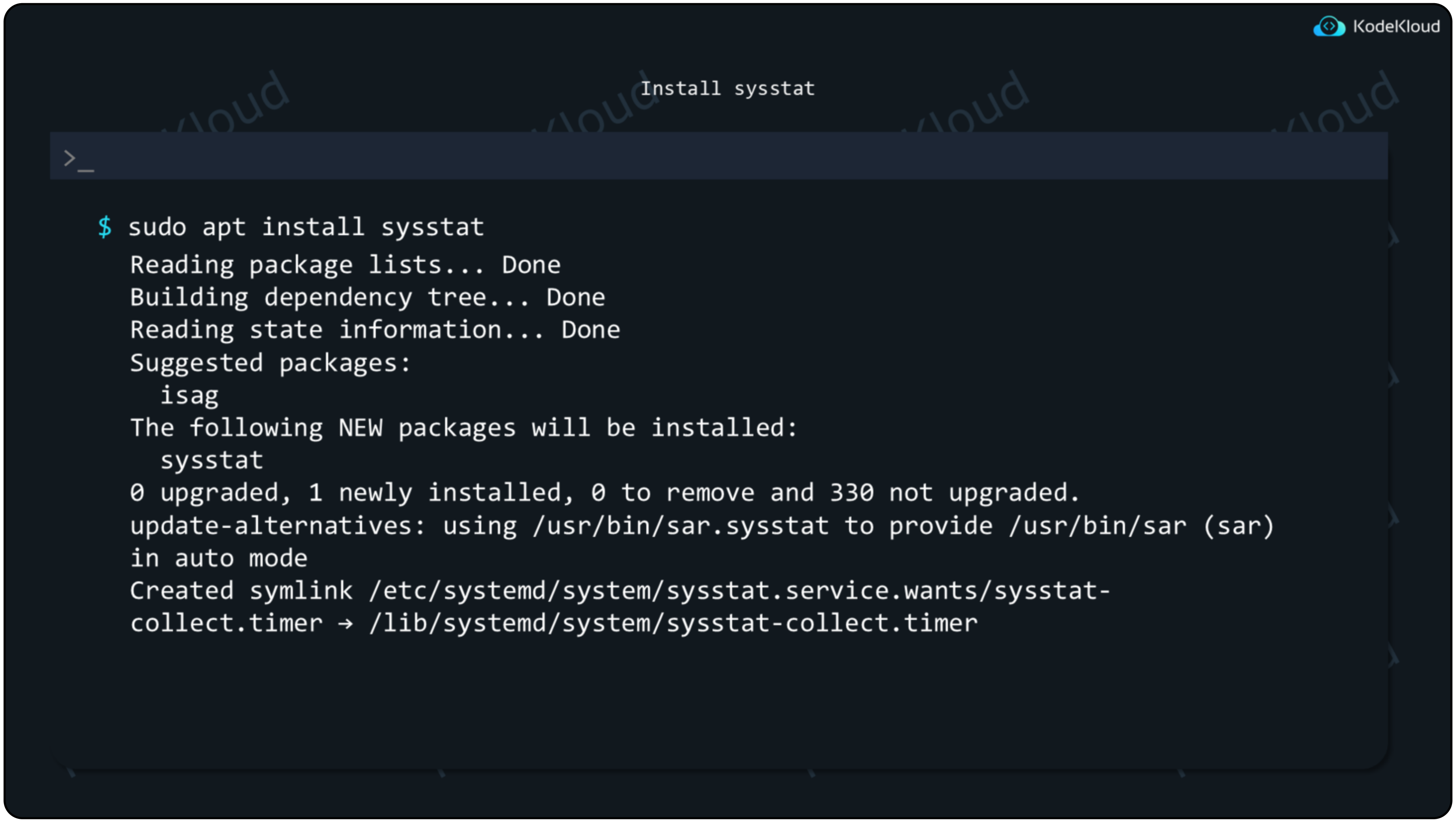
Storage Monitoring

```
$ top
```

```
$ htop
```

```
$ ??????
```

We have utilities like "top", and "htop", to look at how processes are using the CPU and our RAM. But what about storage devices? How can we see what's reading and writing to these?

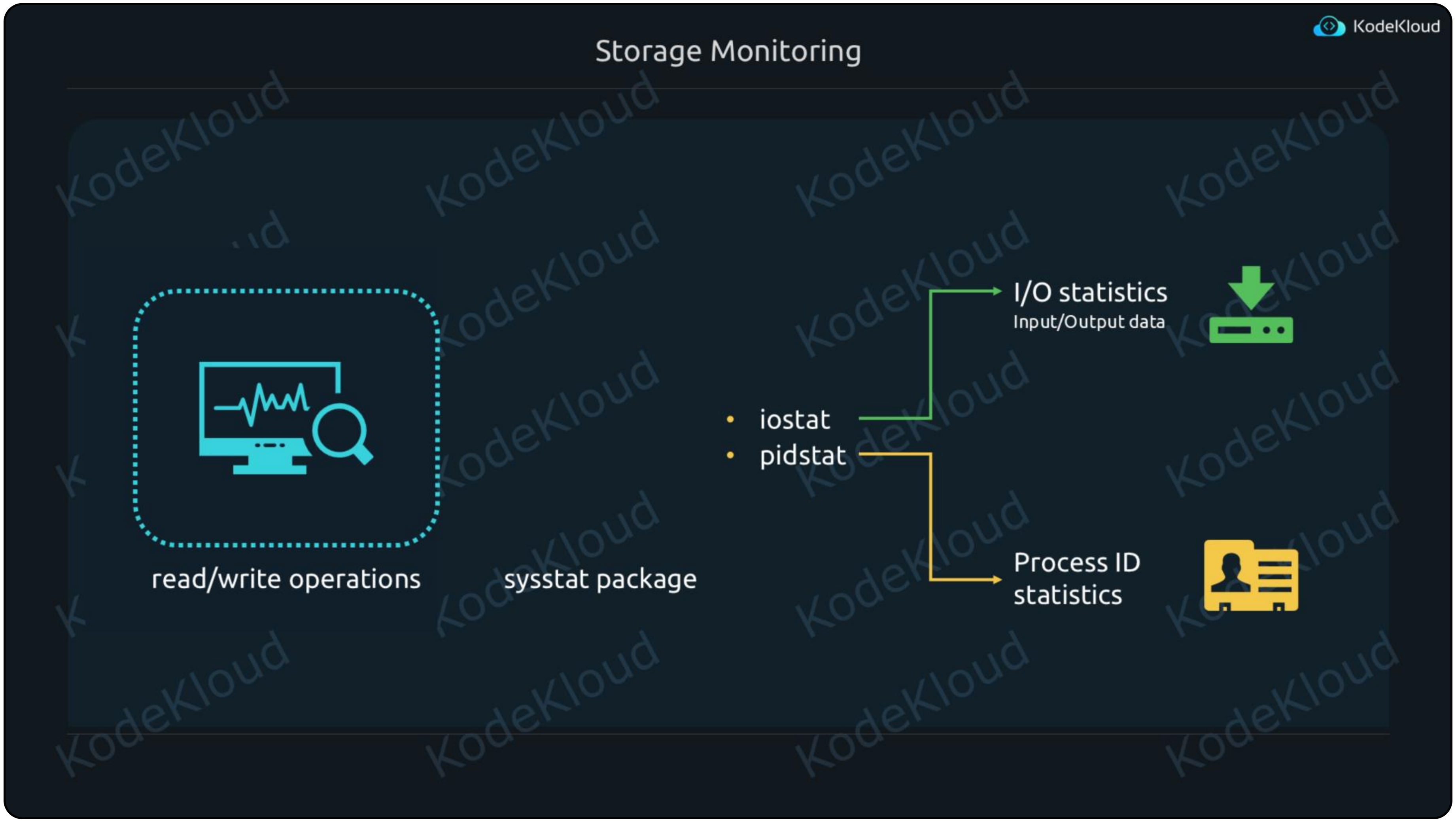

Install sysstat

```
$ sudo apt install sysstat
  Reading package lists... Done
  Building dependency tree... Done
  Reading state information... Done
  Suggested packages:
    isag
  The following NEW packages will be installed:
    sysstat
  0 upgraded, 1 newly installed, 0 to remove and 330 not upgraded.
  update-alternatives: using /usr/bin/sar.sysstat to provide /usr/bin/sar (sar)
  in auto mode
  Created symlink /etc/systemd/system/sysstat.service.wants/sysstat-
  collect.timer → /lib/systemd/system/sysstat-collect.timer
```

There are many tools that can monitor read/write operations. Let's look at a few of the simpler ones. There's a package called "sysstat" that contains a few such tools. We can install it with the usual command:

sudo apt install sysstat

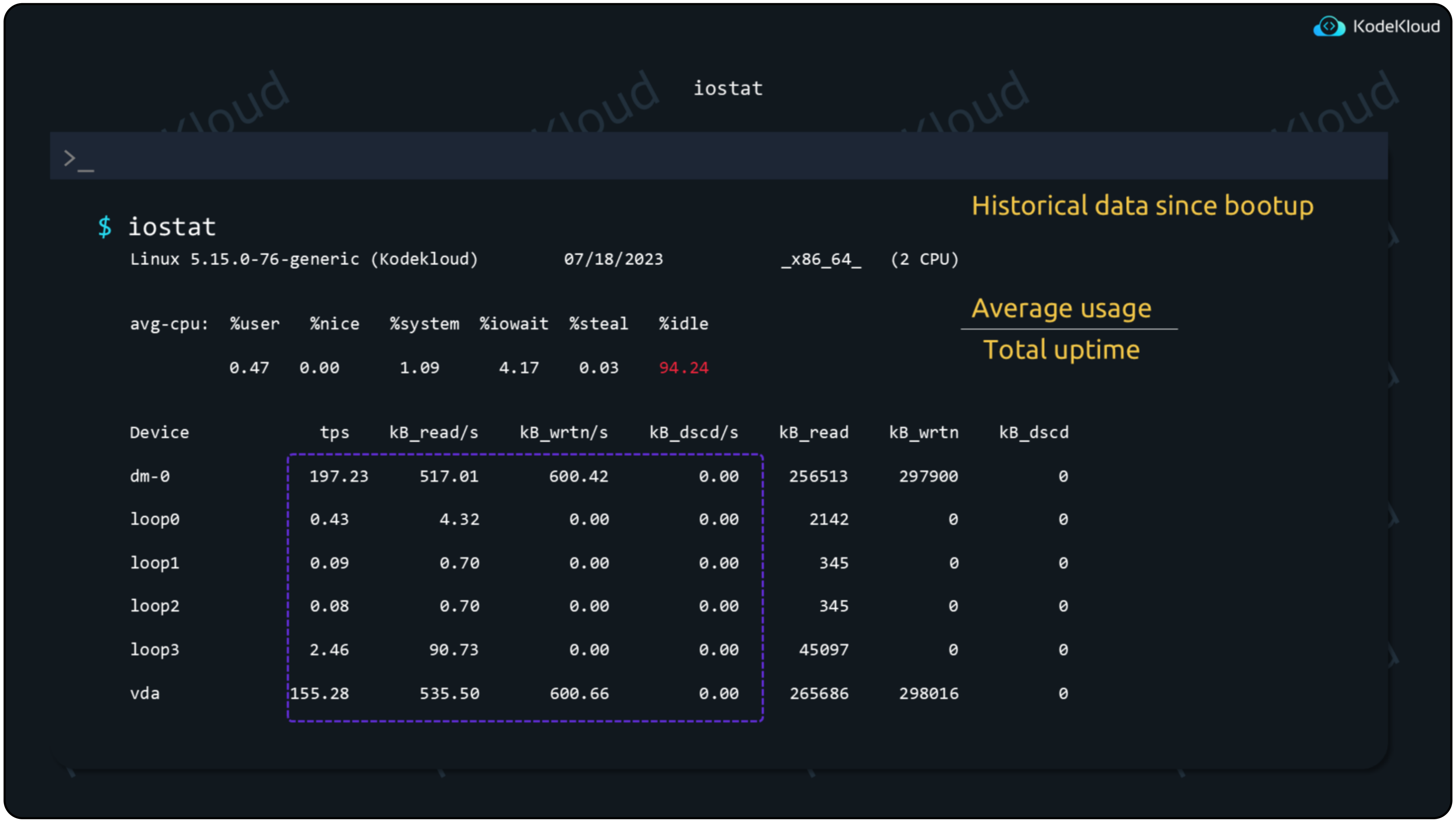
This contains a few other system statistic utilities, but the ones we're interested in are called "iostat" and "pidstat".



iostat's name comes from **I/O stat**istics. And I/O is short for Input/Output. Because we can *input* data to a storage device, when we're writing to it. And the storage device *outputs* data when we read from it.

pidstat is short for "**p**rocess **ID stat**istics". Each process on Linux has an unique IDentifier. So this tool will show us statistics for each process on our system, alongside their ID numbers.

We'll see why both "iostat" and "pidstat" are needed for our purposes.



```
$ iostat
Linux 5.15.0-76-generic (Kodekloud)        07/18/2023          _x86_64_   (2 CPU)

avg-cpu:  %user   %nice   %system  %iowait  %steal   %idle

           0.47    0.00      1.09     4.17     0.03    94.24

Device              tps    kB_read/s    kB_wrtn/s    kB_dscd/s    kB_read    kB_wrtn    kB_dscd

dm-0              197.23      517.01       600.42         0.00     256513     297900          0

loop0               0.43        4.32         0.00         0.00       2142          0          0

loop1               0.09        0.70         0.00         0.00        345          0          0

loop2               0.08        0.70         0.00         0.00        345          0          0

loop3               2.46       90.73         0.00         0.00      45097          0          0

vda               155.28      535.50       600.66         0.00     265686     298016          0
```

Let's start with the simplest way to see a summary of how storage devices are used. We can use the iostat command with no arguments. If we type:

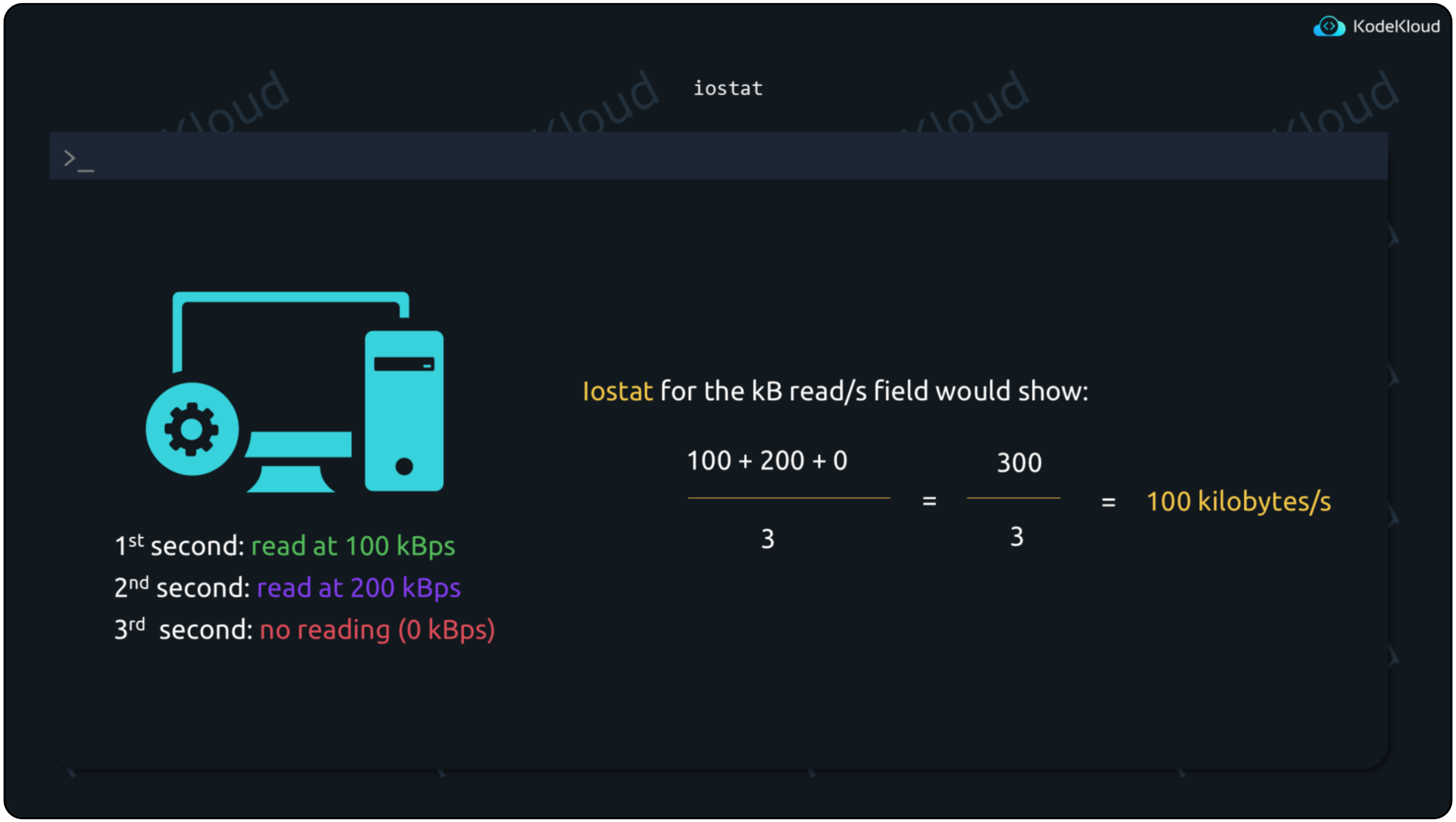
iostat

at the command line, we'll see output like this:

Now let's figure out what this output shows us.

First of all, this displays historical usage of our storage devices. That is, it tells us how the devices were used since the system was booted up, not necessarily how they're used currently. And this is a bit tricky to understand when looking at the per second fields here. But let's simplify.

Those fields show an average of usage, divided by the total time since the system was started.



iostat

```
>_
```

1st second: read at 100 kBps

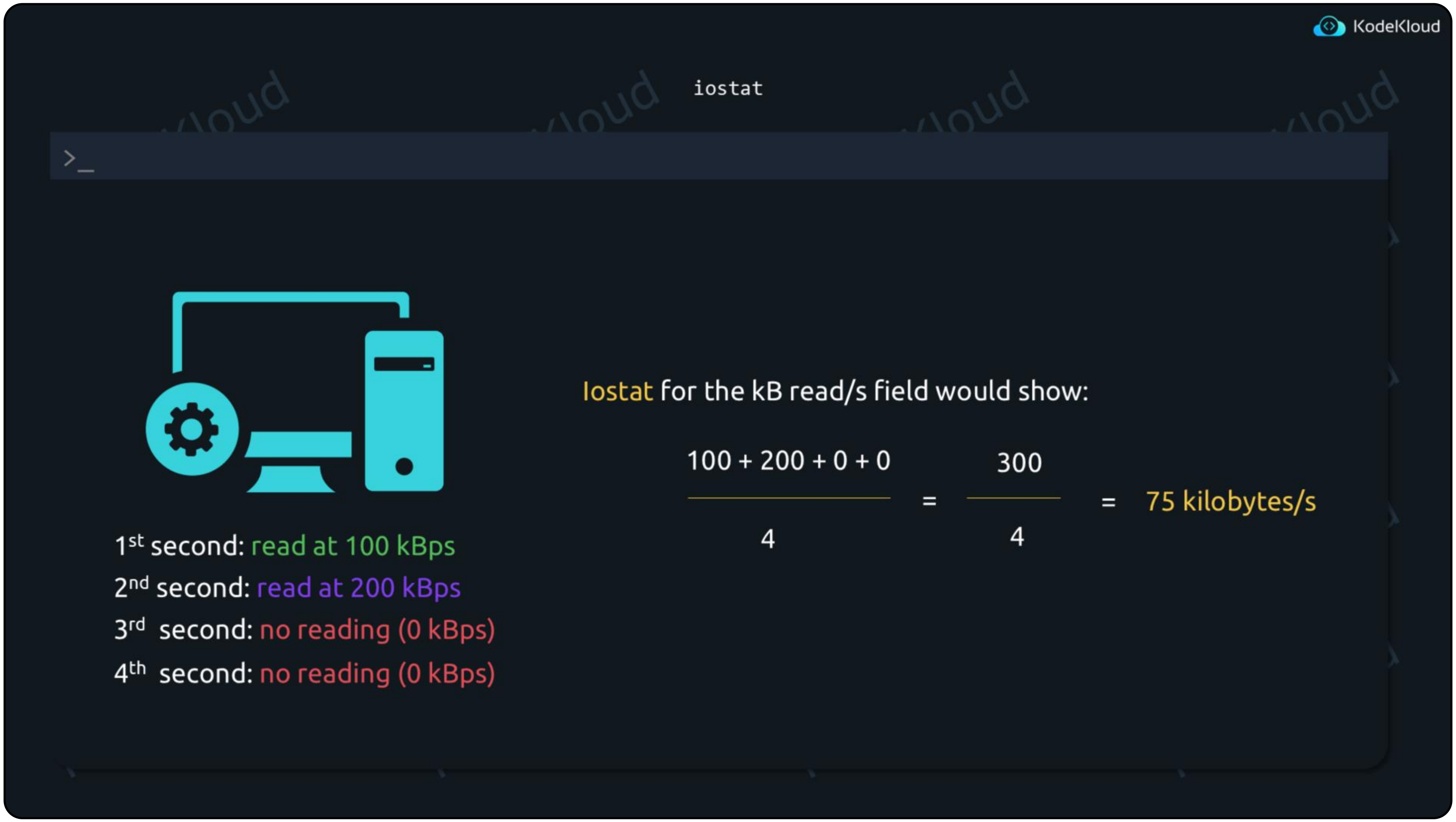
2nd second: read at 200 kBps

3rd second: no reading (0 kBps)

Iostat for the kB read/s field would show:

$$\frac{100 + 200 + 0}{3} = \frac{300}{3} = 100 \text{ kilobytes/s}$$

For example, let's say the system was booted 3 seconds ago. In the first second, something read from the disk at 100 kilobytes per second. In the next second, something read from the disk at 200 kilobytes per second. In the third second, nothing was read from the disk anymore, so usage was at zero kilobytes per second. If we add 100+200+0 and then divide by the total number of seconds since the system was booted, three seconds, we get an average usage of 300/3 = 100 kilobytes per second. So iostat might show us "100" in the "kB_read/s" field.



KodeKloud

iostat

```
>_
```

1st second: read at 100 kBps

2nd second: read at 200 kBps

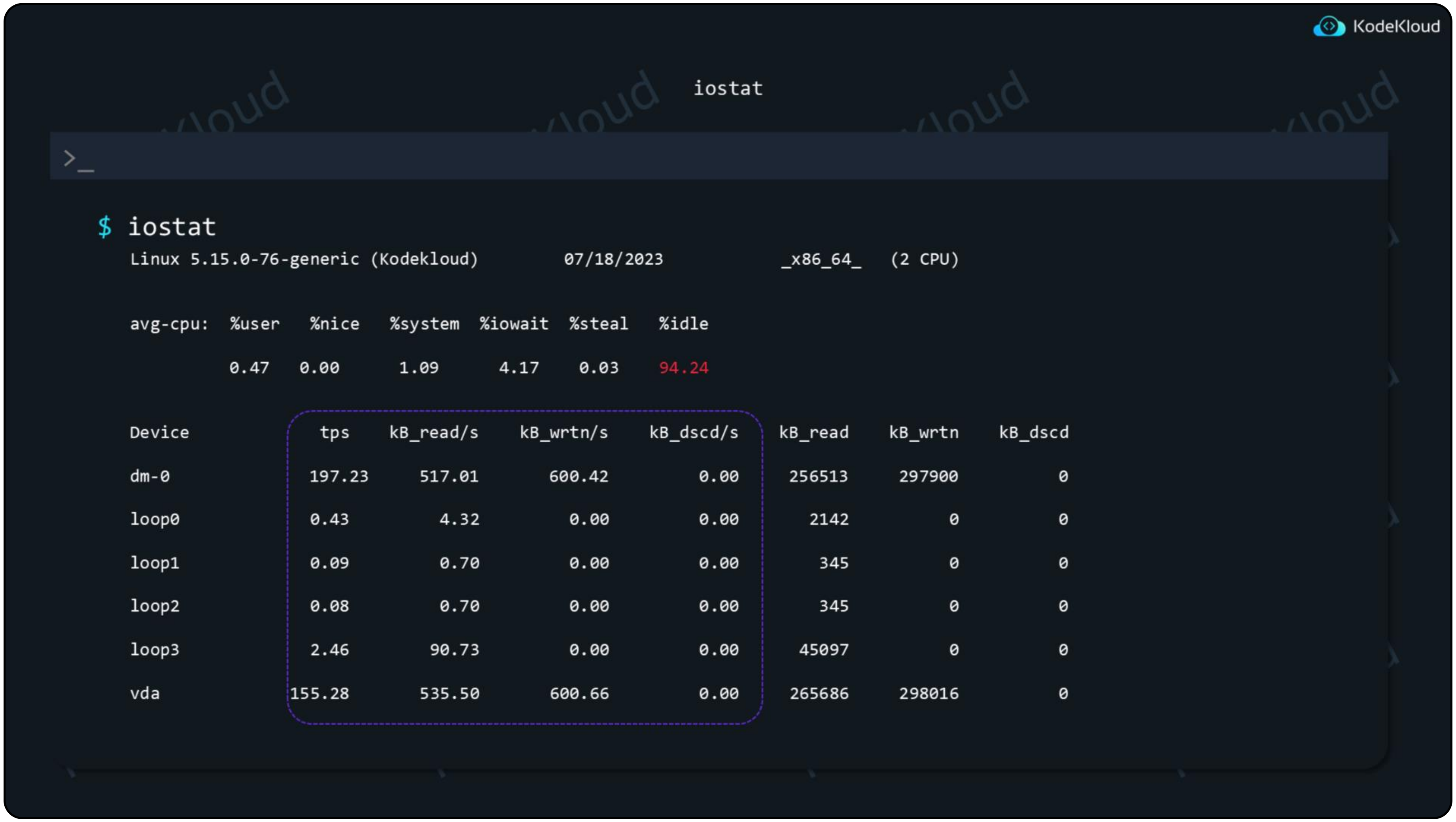
3rd second: no reading (0 kBps)

4th second: no reading (0 kBps)

Iostat for the kB read/s field would show:

$$\frac{100 + 200 + 0 + 0}{4} = \frac{300}{4} = 75 \text{ kilobytes/s}$$

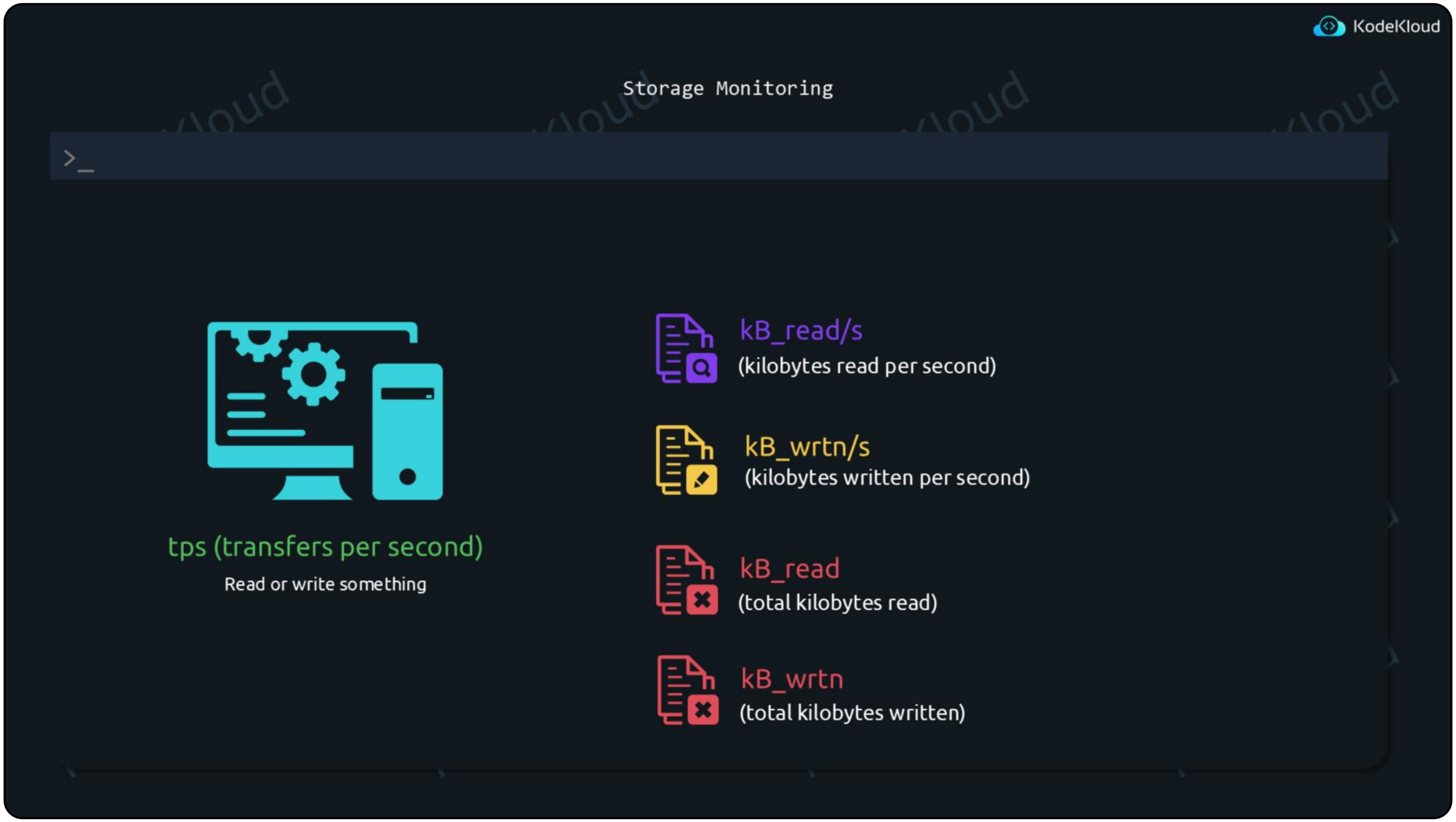
If another second passes, and nothing reads from the disk, iostat would show something different the next time we run it. Now we have 100+200+0+0 divided by 4 total seconds since the system is up. So we might see "75" in the "kB_read/s" field.



iostat

```
$ iostat
  Linux 5.15.0-76-generic (Kodekloud)        07/18/2023           _x86_64_   (2 CPU)

  avg-cpu:  %user   %nice   %system  %iowait  %steal   %idle

            0.47   0.00      1.09      4.17     0.03    94.24

  Device            tps     kB_read/s    kB_wrtn/s    kB_dscd/s   kB_read    kB_wrtn    kB_dscd

  dm-0             197.23      517.01       600.42         0.00    256513     297900          0

  loop0            0.43          4.32         0.00         0.00      2142          0          0

  loop1            0.09          0.70         0.00         0.00       345          0          0

  loop2            0.08          0.70         0.00         0.00       345          0          0

  loop3            2.46         90.73         0.00         0.00     45097          0          0

  vda            155.28        535.50       600.66         0.00    265686     298016          0
```

The fields which are calculated this way are "tps", "kB_read/s", "kB_wrtn/s", and "kB_dscd/s".


KodeKloud
Storage Monitoring
>_
tps (transfers per second)
Read or write something
kB_read/s
(kilobytes read per second)
kB_wrtn/s
(kilobytes written per second)
kB_read
(total kilobytes read)
kB_wrtn
(total kilobytes written)

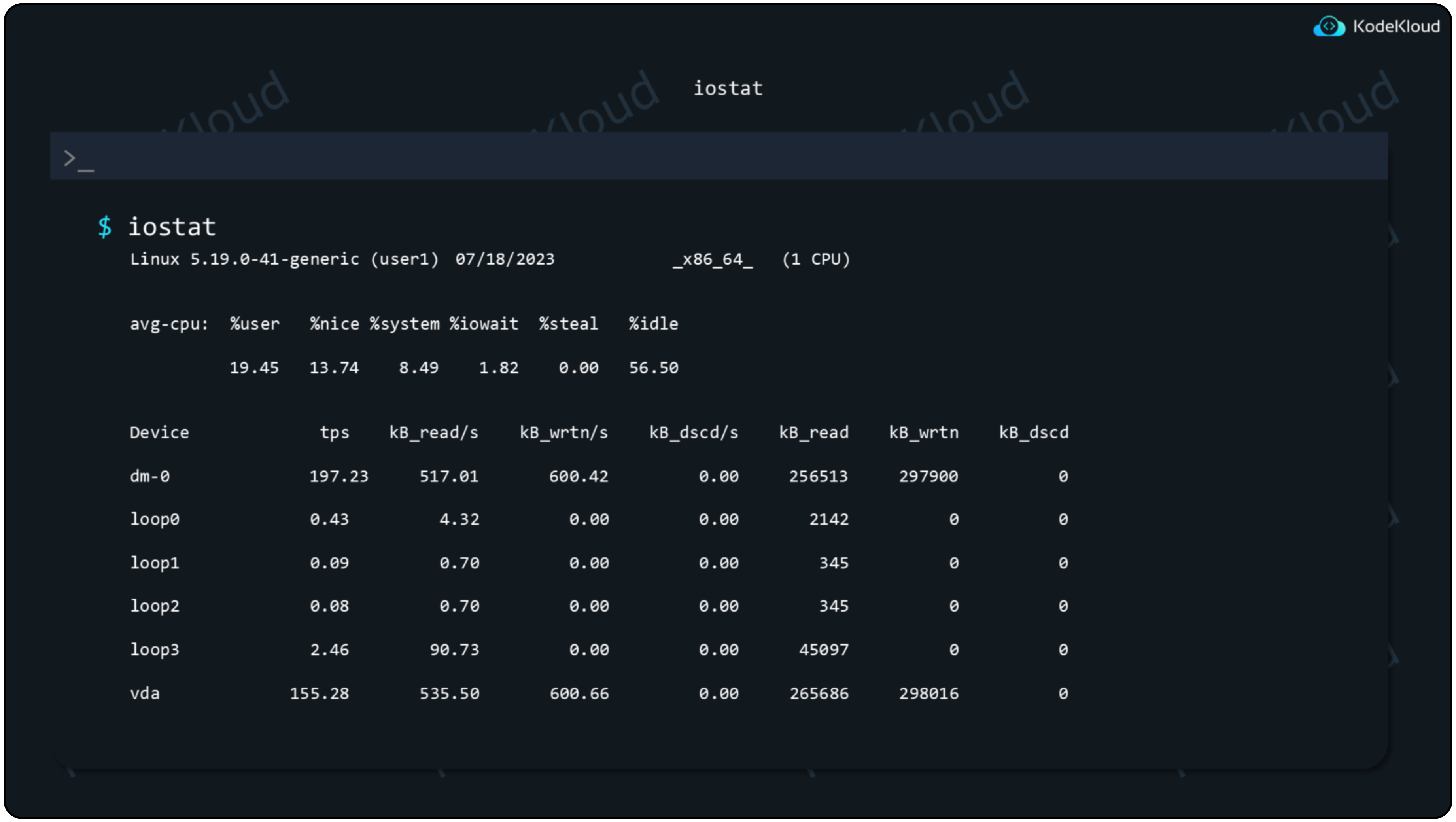
When we're looking at what's using the storage devices we're interested in the first three fields:

tps stands for transfers per second. We can think of it in terms of "How many times did the system tell this device to read or write something?".

kB_read/s refers to kilobytes read per second.

kB_wrtn/s shows kilobytes written per second.

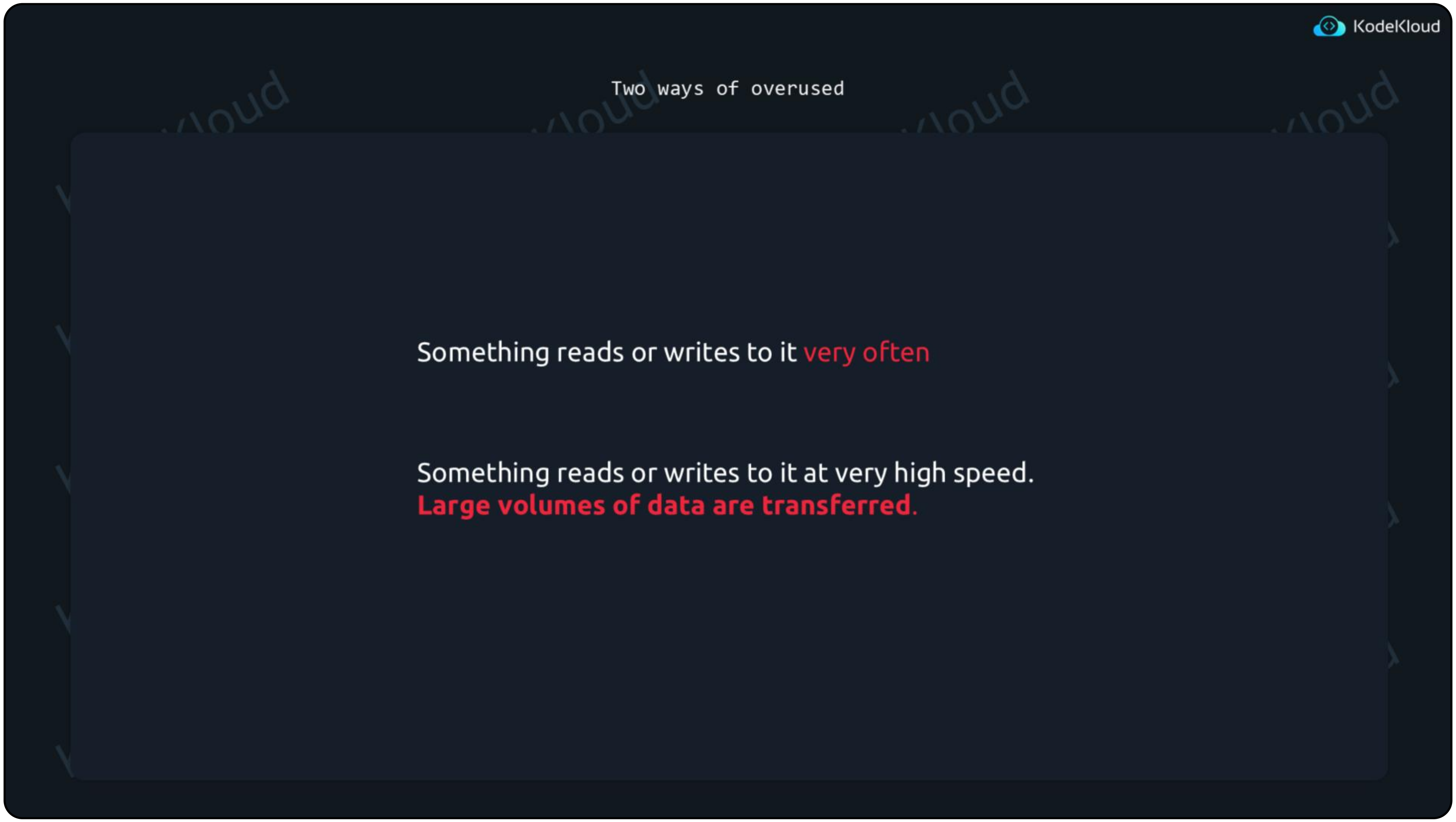
The fields "kB_read" and "kB_wrtn" are much more straightforward. They simply show the total number of kilobytes read or written since the system booted. They don't depend on time, so there's no average to calculate.



iostat

```
$ iostat
  Linux 5.19.0-41-generic (user1)  07/18/2023           _x86_64_   (1 CPU)

  avg-cpu:  %user   %nice %system %iowait  %steal   %idle

            19.45   13.74    8.49    1.82    0.00   56.50

  Device             tps    kB_read/s    kB_wrtn/s    kB_dscd/s    kB_read    kB_wrtn    kB_dscd

  dm-0            197.23       517.01       600.42         0.00     256513     297900          0

  loop0             0.43         4.32         0.00         0.00       2142          0          0

  loop1             0.09         0.70         0.00         0.00        345          0          0

  loop2             0.08         0.70         0.00         0.00        345          0          0

  loop3             2.46        90.73         0.00         0.00      45097          0          0

  vda             155.28       535.50       600.66         0.00     265686     298016          0
```

Something to note here is that what iostat shows us does not always reflect the exact amount of data per second written or read by a process. For example, a process can request to write 1 kilobyte every second. But iostat might show the device is used at thousands of kilobytes per second, during that time. That's because iostat tries to figure out these numbers by looking at what the device is reporting, not the process. And the device might work with larger blocks of data. Even if a process sends just 1kB, the device might report it updated a much larger block. So estimates can be inflated here, especially for small transfers. But they become more accurate for larger transfers.



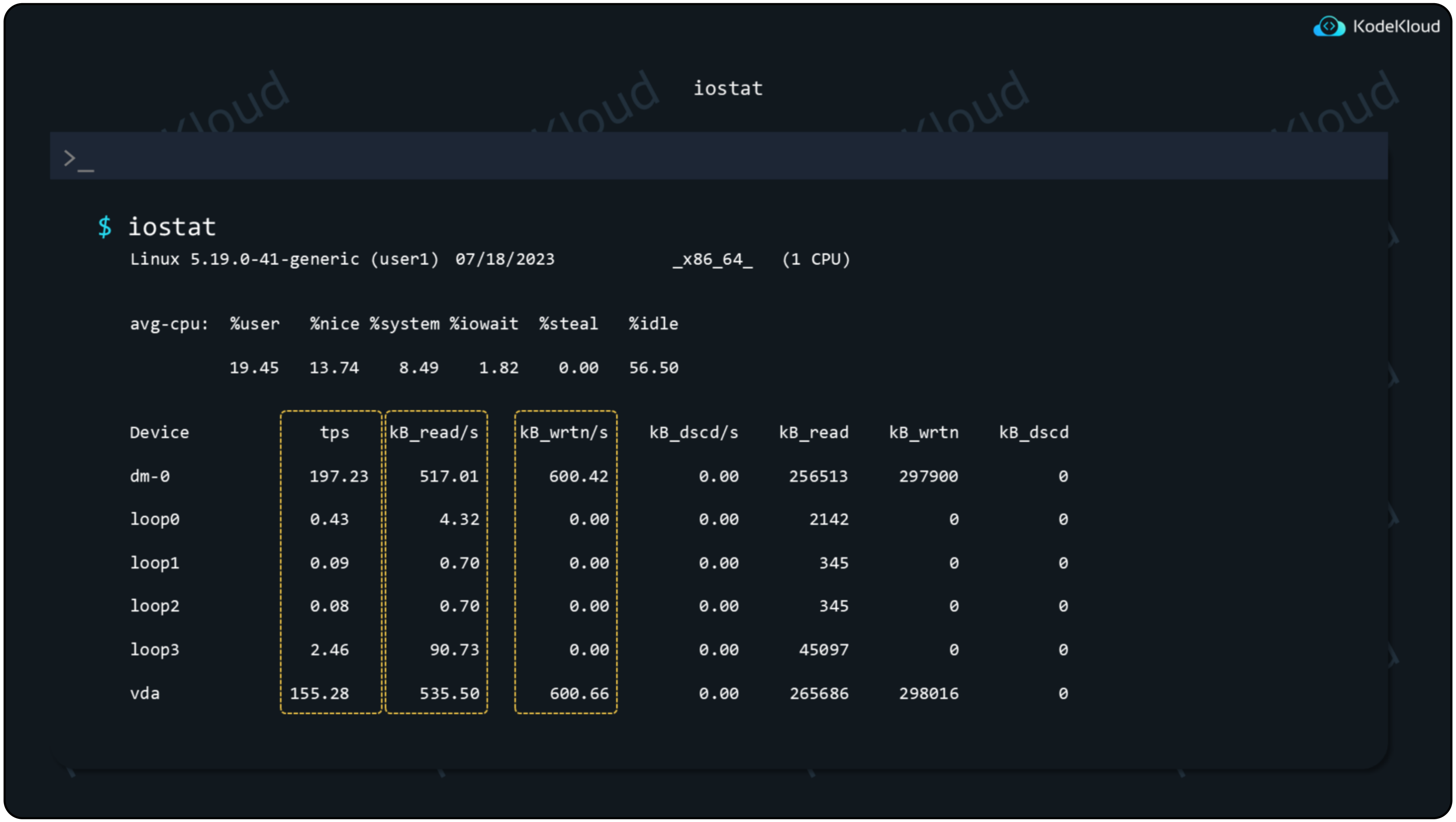
Two ways of overused

Something reads or writes to it very often

Something reads or writes to it at very high speed.
Large volumes of data are transferred.

A storage device can be overused, or stressed in two different ways:

Something reads or writes to it very often.
Or something reads or writes to it at very high speed. Large volumes of data are transferred.


iostat

```
$ iostat
Linux 5.19.0-41-generic (user1)  07/18/2023          _x86_64_   (1 CPU)

avg-cpu:  %user   %nice %system %iowait  %steal   %idle

          19.45   13.74    8.49    1.82    0.00   56.50

Device             tps    kB_read/s    kB_wrtn/s    kB_dscd/s    kB_read    kB_wrtn    kB_dscd
dm-0            197.23       517.01       600.42         0.00     256513     297900          0
loop0             0.43         4.32         0.00         0.00       2142          0          0
loop1             0.09         0.70         0.00         0.00        345          0          0
loop2             0.08         0.70         0.00         0.00        345          0          0
loop3             2.46        90.73         0.00         0.00      45097          0          0
vda             155.28       535.50       600.66         0.00     265686     298016          0
```

If something uses the devices often, we'll see a high tps number. And for large volumes of data, we'll see large numbers under the kB_read/s and kB_wrtn/s fields.

So even if something is writing very little data to the device, say 1 kilobyte at a time, if it does this hundreds of times per second, it's still stressful for the device. Potentially reaching its limits. Because each storage device can process a certain maximum amount of requests per second. If one process does a high number of transfers per second, there's not much left for other processes to use. So one process can starve many other processes by overusing a storage device.

The same goes for large transfers. If the device can only write 2 gigabytes of data per second, and a process is already writing at 1.98 gigabytes, there's very little left for other processes. So they will wait for a long, long time until they can finish writing their data. Which can lead to a very slow, unresponsive system.

And this is where "iostat" and "pidstat" can team up and help us investigate scenarios where processes overuse our storage devices. Let's put on our detective hats and see how by looking at some commands in action.

KodeKloud

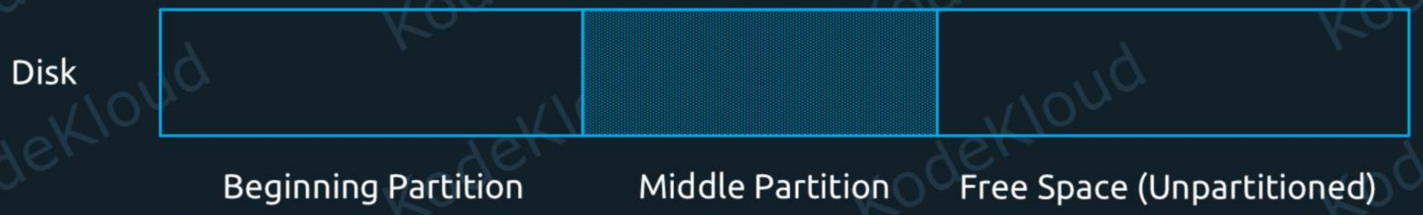
Visit www.kodekloud.com to discover more.

KodeKloud

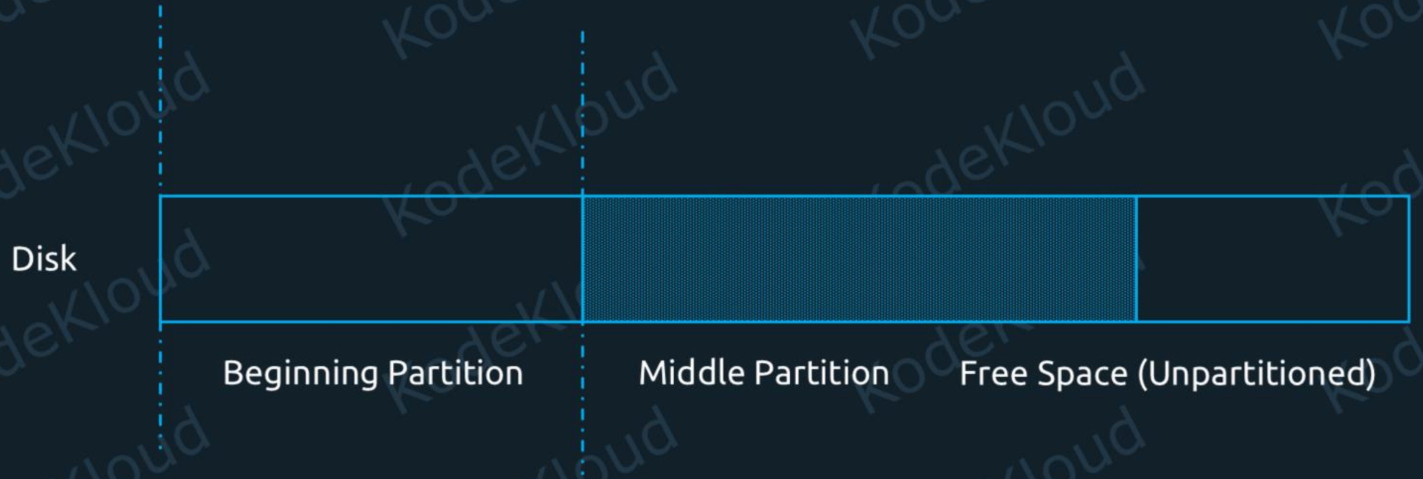
Logical Volume Manager

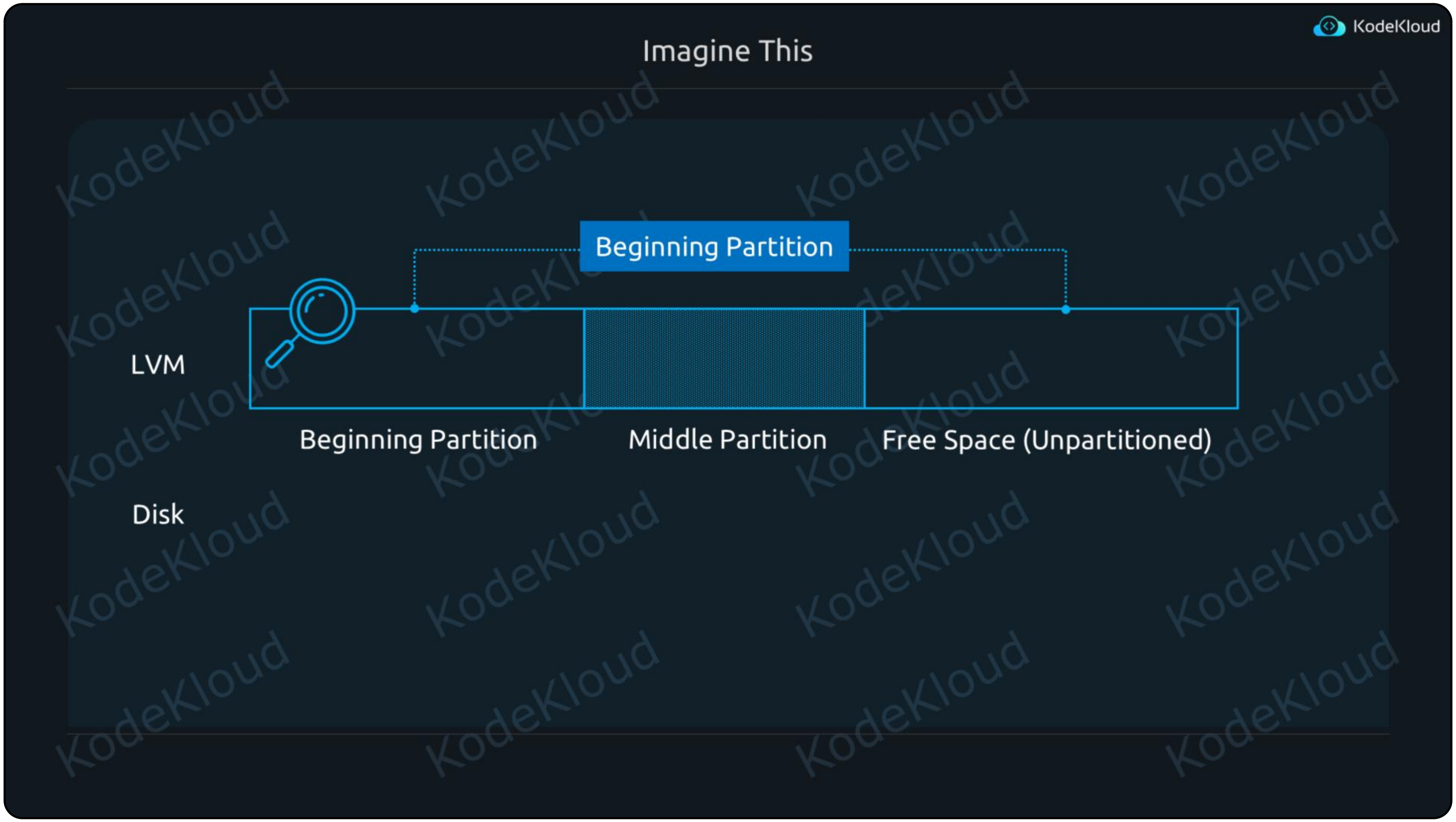
KodeKloud

Imagine This



Imagine This

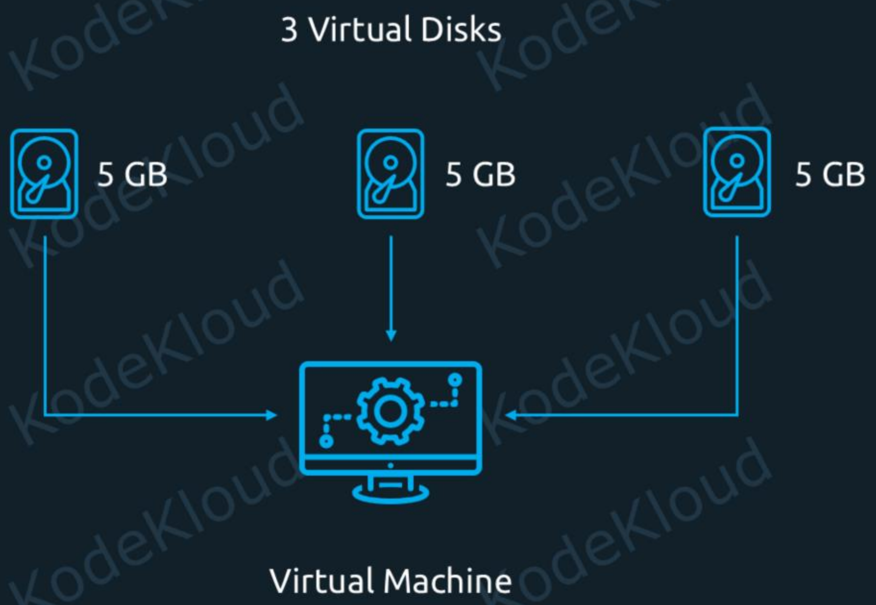




Imagine This

KodeKloud

Practical Exercise



KodeKloud

Visit www.kodekloud.com to discover more.