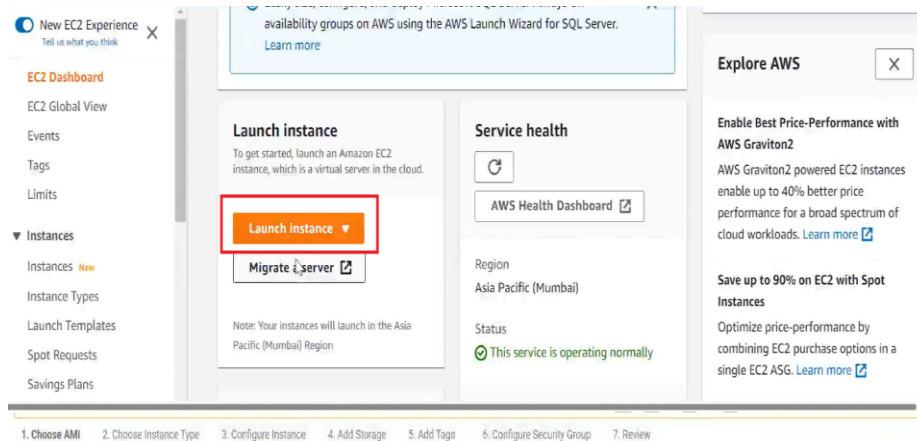


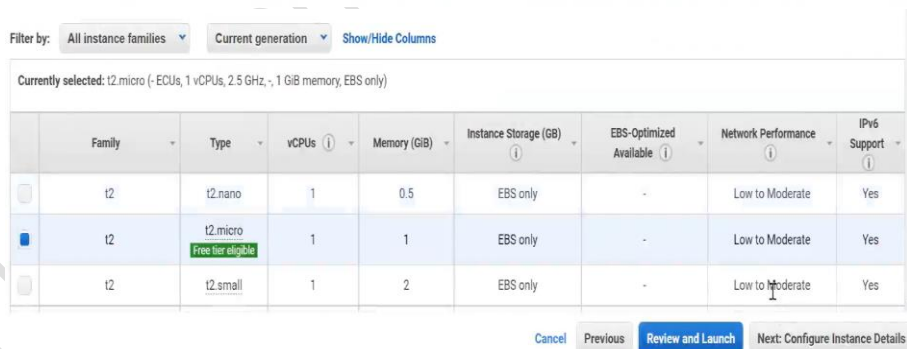
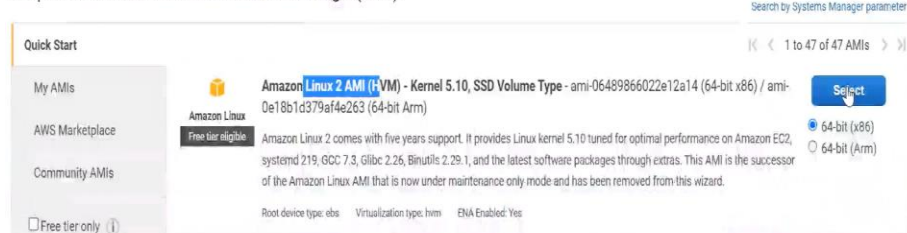


## Shell and Shell Scripting Session No.3.1

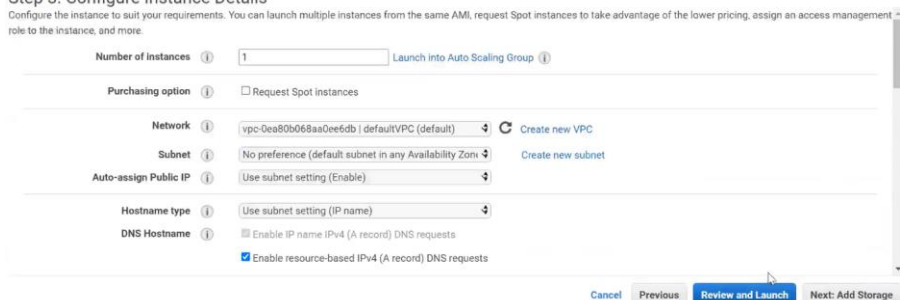
- For doing today's practical we need a Linux OS, and we launch one Linux OS instance on AWS cloud.



### Step 1: Choose an Amazon Machine Image (AMI)



### Step 3: Configure Instance Details



## [Shell And Shell Scripting]

Volume Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Throughput (MB/s)	Delete on Termination	Encryption
	Root	/dev/xvda	8	General Purpose SSD (gp2)	100 / 3000	N/A	<input checked="" type="checkbox"/>	Not Encrypted

Free tier eligible customers can get up to 30 GB of EBS General Purpose (SSD) or Magnetic storage. [Learn more](#) about free usage tier eligibility and usage restrictions.

Cancel Previous **Review and Launch** Next: Add Tags

### Step 5: Add Tags

A tag consists of a case-sensitive key-value pair. For example, you could define a tag with key = Name and value = Webserver. A copy of a tag can be applied to volumes, instances or both. Tags will be applied to all instances and volumes. [Learn more](#) about tagging your Amazon EC2 resources.

Key (128 characters maximum)	Value (256 characters maximum)	Instances	Volumes	Network Interfaces
Name	shellOSlinux	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Add another tag (Up to 50 tags maximum)

### Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group: ☒ Create a new security group ☐ Select an existing security group

Security group name: launch-wizard-70

Description: launch-wizard-70 created 2022-09-03T14:17:40.433+05:30

Type	Protocol	Port Range	Source	Description
SSH	TCP	22	Custom	0.0.0.0/0

Add Rule

Select an existing key pair or create a new key pair

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance. Amazon EC2 supports ED25519 and RSA key pair types.

Note: The selected key pair will be added to the set of keys authorized for this instance. [Learn more](#) about [removing existing key pairs from a public AMI](#).

Choose an existing key pair

Select a key pair

aws\_training\_2022\_key | RSA

☐ I acknowledge that I have access to the corresponding private key file, and that without this file, I won't be able to log into my instance.

Cancel Launch Instances

- And after all the above steps, you have your instance ready.
- When you have operating systems like Linux, or Mac, or appliances like Cisco routers or Cisco switches, we always work on the command line interface. It's a very rare case when we use GUI otherwise most of the time command line is the only way we interact with OS.

- But if we talk about Linux OS, whenever we type any command there is one program behind the scene that takes our command and return the output. This program is known as SHELL.

The screenshot shows a terminal window with the following text:

```
EC2
_ | _ | _
_ | ( _ | _ / Amazon Linux 2 AMI
_ | \ _ | _ |

https://aws.amazon.com/amazon-linux-2/
3 package(s) needed for security, out of 8 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-6-112 ~]$
[ec2-user@ip-172-31-6-112 ~]$
[ec2-user@ip-172-31-6-112 ~]$ date
Sat Sep  3 08:54:28 UTC 2022
[ec2-user@ip-172-31-6-112 ~]$
```

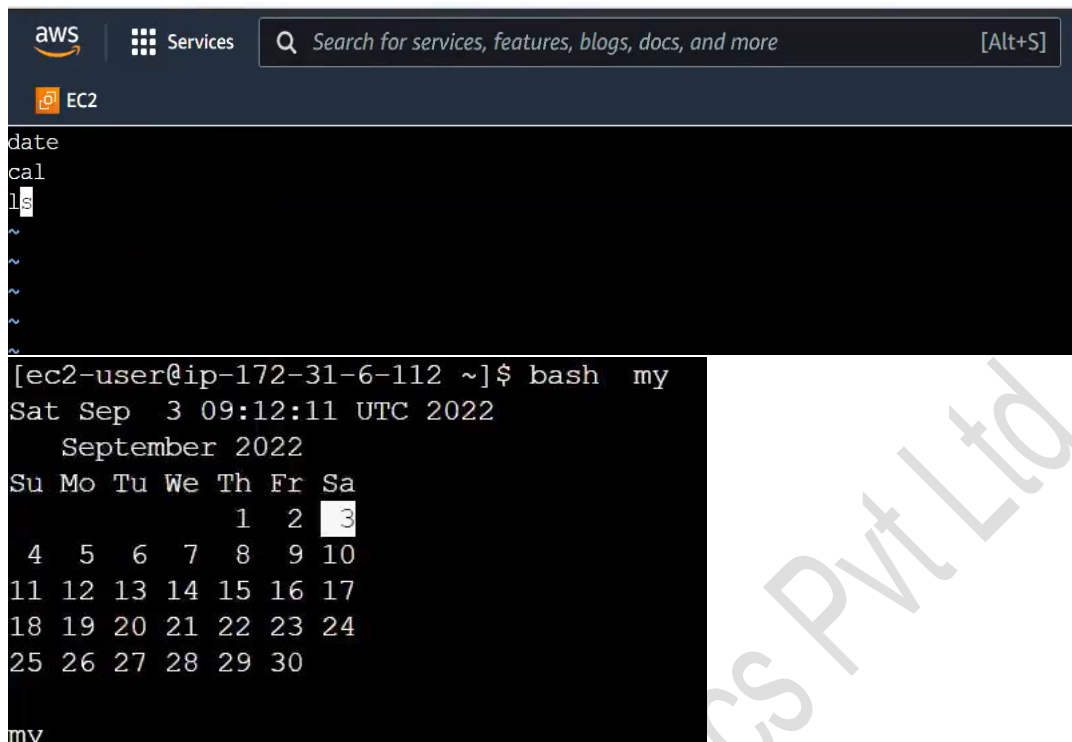
Handwritten annotations in yellow and orange include:

- A box around the prompt `[ec2-user@ip-172-31-6-112 ~]$` with an arrow pointing to the word "Shell".
- A box around the command `date` with an arrow pointing to the word "Program".
- A diagram on the right showing a box labeled "Shell" with an arrow pointing up to "H.B" and an arrow pointing down to "O.S".

- There are many types of shells available in the market like sh, ksh, csh, fish, etc. However, the standard shell available in the market that is mostly used by everyone is the BASH shell.
- Shell is an interface that takes command, it's not the one who gives the command. It can be used for telling it what to do on that command.
- As we can tell if the date command runs successfully then do something.

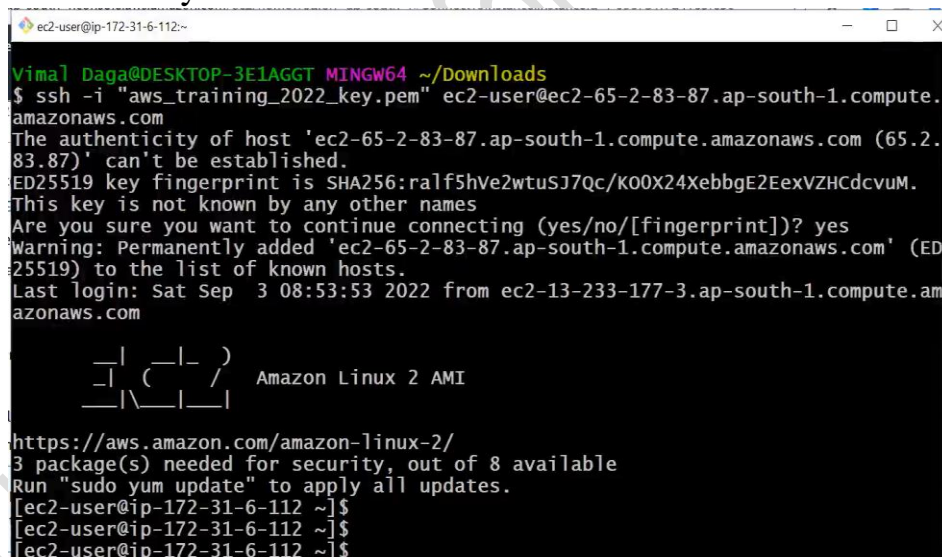
```
[ec2-user@ip-172-31-6-112 ~]$ if date
> then
> echo "i m done"
> else
> echo "error"
> fi
Sat Sep  3 09:00:57 UTC 2022
i m done
[ec2-user@ip-172-31-6-112 ~]$
```

- Every device has its commands that are different from each other but when the case is of controlling them or manipulating those commands we have the same interface which is a shell and the shell that is mostly applicable to all means the standard one is "SHELL"
- Suppose we have run some command again and again, so instead of running it manually, we can create a file and write all these commands there in a file in order and run that file. This file is known as Script.



```
aws Services Search for services, features, blogs, docs, and more [Alt+S]
EC2
date
cal
ls
~
~
~
~
~
[ec2-user@ip-172-31-6-112 ~]$ bash my
Sat Sep  3 09:12:11 UTC 2022
    September 2022
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30
my
```

- But here our plan is not just to learn Bash shell but to learn scripting and scripting not only involves commands but it also includes logic and conditions and many more things.
- Connect to your instance via ssh from the base OS.



```
ec2-user@ip-172-31-6-112:~
Vimal Daga@DESKTOP-3E1AGGT MINGW64 ~/Downloads
$ ssh -i "aws_training_2022_key.pem" ec2-user@ec2-65-2-83-87.ap-south-1.compute.amazonaws.com
The authenticity of host 'ec2-65-2-83-87.ap-south-1.compute.amazonaws.com (65.2.83.87)' can't be established.
ED25519 key fingerprint is SHA256:raIf5hve2wtuSj7Qc/K00X24XebbgE2EexVZHCdcvUM.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ec2-65-2-83-87.ap-south-1.compute.amazonaws.com' (ED25519) to the list of known hosts.
Last login: Sat Sep  3 08:53:53 2022 from ec2-13-233-177-3.ap-south-1.compute.amazonaws.com

 _ _ | _ _ | _ _ )
 _ | ( _ _ | _ _ /  Amazon Linux 2 AMI
 _ | \ _ _ | _ _ |

https://aws.amazon.com/amazon-linux-2/
3 package(s) needed for security, out of 8 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-31-6-112 ~]$
[ec2-user@ip-172-31-6-112 ~]$
[ec2-user@ip-172-31-6-112 ~]$
```

- And login to root user from ec2-user.

```
root@ip-172-31-6-112:~  
[ec2-user@ip-172-31-6-112 ~]$  
[ec2-user@ip-172-31-6-112 ~]$  
[ec2-user@ip-172-31-6-112 ~]$ sudo su - root  
[root@ip-172-31-6-112 ~]#  
[root@ip-172-31-6-112 ~]# whoami  
root  
[root@ip-172-31-6-112 ~]#
```

- Scripting is like a programming language means whatever we do in a programming language, we can also do it here.
- Here also we have the concept of a variable. We can set variables on a shell and also remove it.

```
root@ip-172-31-6-112:~  
[root@ip-172-31-6-112 ~]#  
[root@ip-172-31-6-112 ~]# echo hi  
hi  
[root@ip-172-31-6-112 ~]# echo "hi"  
hi  
[root@ip-172-31-6-112 ~]# x=5  
[root@ip-172-31-6-112 ~]# echo x  
x  
[root@ip-172-31-6-112 ~]#  
[root@ip-172-31-6-112 ~]# echo $x  
5  
[root@ip-172-31-6-112 ~]# x=7  
[root@ip-172-31-6-112 ~]# echo $x  
7  
[root@ip-172-31-6-112 ~]# unset x  
[root@ip-172-31-6-112 ~]# echo $x  
[root@ip-172-31-6-112 ~]#
```

- For variables we use the '\$' symbol. Otherwise, echo will print x as a string not as a variable.
- And for using the variable between strings, we use the '\$' symbol again.

```
[root@ip-172-31-6-112 ~]# x=7  
[root@ip-172-31-6-112 ~]# echo "i m vimal hi $x you r u"  
i m vimal hi 7 you r u
```

- But if the requirement is like you have some suffix or prefix with that variable like, with 7 we have 'th'. So for telling shell that for 7 x is the variable and 'th' is the string to add after it, we use { }.

```
[root@ip-172-31-6-112 ~]# echo "i m vimal hi $x th you r u"  
i m vimal hi 7 th you r u  
[root@ip-172-31-6-112 ~]# echo "i m vimal hi $xth you r u"  
i m vimal hi 7th you r u  
[root@ip-172-31-6-112 ~]# echo "i m vimal hi ${x}th you r u"  
i m vimal hi 7th you r u  
[root@ip-172-31-6-112 ~]#
```

- And if you want to store the output of the command in a variable rather than storing the string then you can use below two ways.



```
[root@ip-172-31-6-112 ~]# y=$(date)
[root@ip-172-31-6-112 ~]# echo $y
Sat Sep 3 09:27:38 UTC 2022
[root@ip-172-31-6-112 ~]# z=`date`
[root@ip-172-31-6-112 ~]# echo $z
Sat Sep 3 09:28:27 UTC 2022
```

- Rather than giving input to a variable this way, we can use the 'read' command. It will take input on a live prompt and store it in a variable.

```
[root@ip-172-31-6-112 ~]# read p
jack
[root@ip-172-31-6-112 ~]# echo $p
jack
[root@ip-172-31-6-112 ~]#
```

- There are also more options in the read command.

```
[root@ip-172-31-6-112 ~]# read -p "Enter ur name : " n
Enter ur name : vimal
[root@ip-172-31-6-112 ~]# echo $n
vimal
[root@ip-172-31-6-112 ~]# read -p "Enter ur name : " -n 2 myname
Enter ur name : vi[root@ip-172-31-6-112 ~]#
[root@ip-172-31-6-112 ~]#
[root@ip-172-31-6-112 ~]# echo $myname
vi
```

- While creating a script file, it is good practice to give it an extension '.sh'
- Below is the simple script for creating a file on a live prompt.

```
root@ip-172-31-6-112:~
read -p "enter ur file name : " myfile
touch $myfile
~
~
~

[root@ip-172-31-6-112 ~]# vim my.sh
[root@ip-172-31-6-112 ~]# ls
a.txt b.txt c.txt my.sh
[root@ip-172-31-6-112 ~]# bash my.sh
enter ur file name : hi.txt
[root@ip-172-31-6-112 ~]# ls
a.txt b.txt c.txt hi.txt my.sh
[root@ip-172-31-6-112 ~]#
```

- Whenever we run any command, it gives us the status code. Exit code is the code that shows the status of your last command whether it's

successful or not. On success, it returns 0 and on fail it returns any other number.

```
[root@ip-172-31-6-112 ~]# date
Sat Sep  3 09:35:59 UTC 2022
[root@ip-172-31-6-112 ~]# echo  $?
0
[root@ip-172-31-6-112 ~]# date1
-bash: date1: command not found
[root@ip-172-31-6-112 ~]# echo  $?
127
```

- For testing a condition in a shell we have the `test` keyword. We can use the `test` keyword and pass any condition. It will check the condition and based on its exit code, we come to know whether the condition is true or not.

```
[root@ip-172-31-6-112 ~]# test "hi" == "hi" |
[root@ip-172-31-6-112 ~]# echo  $?
0
[root@ip-172-31-6-112 ~]# test "hi" == "hello"
[root@ip-172-31-6-112 ~]# echo  $?
1
[root@ip-172-31-6-112 ~]#
```

- And all these mathematical operators like ==, <=, >=, <, >, are working in this version of the shell but it doesn't need to work everywhere. So for these, we have another way. We use operators -eq, -ge, -le, -NE

```
[root@ip-172-31-6-112 ~]# test 5 -eq 6
[root@ip-172-31-6-112 ~]# echo  $?
1
[root@ip-172-31-6-112 ~]# test 5 -ne 6
[root@ip-172-31-6-112 ~]# echo  $?
0
[root@ip-172-31-6-112 ~]# test 5 -le 6
[root@ip-172-31-6-112 ~]# echo  $?
0
[root@ip-172-31-6-112 ~]# test 5 -ge 6
[root@ip-172-31-6-112 ~]# echo  $?
1
[root@ip-172-31-6-112 ~]#
```

- For running more than one command we use the `;` symbol between them. This will run all the commands concurrently and if one command

fails other command will not get affected.

```
[root@ip-172-31-6-112 ~]# date; cal; ls
Sat Sep  3 09:45:00 UTC 2022
    September 2022
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

a.txt  b.txt  c.txt  hi.txt  my.sh
```

- But if we want that If one command fails it stops the other command then we can use the '&&' symbol. If the first command fails then it will not run the second command. But if it's true then it will run other commands.

```
[root@ip-172-31-6-112 ~]# date1 && | cal
-bash: date1: command not found
[root@ip-172-31-6-112 ~]#
```

- And && will help in creating test cases also.

```
[root@ip-172-31-6-112 ~]# test $? -eq 0 && echo "done ok"
done ok
[root@ip-172-31-6-112 ~]# date1
-bash: date1: command not found
[root@ip-172-31-6-112 ~]# test $? -eq 0 && echo "done ok"
[root@ip-172-31-6-112 ~]#
```

- And '|' symbol works opposite to '&&' symbol. It's like the 'or' keyword that if the first one is True then it will not run another command and if the first one is false then it will run another command.

```
[root@ip-172-31-6-112 ~]# date || cal
Sat Sep  3 09:51:55 UTC 2022
[root@ip-172-31-6-112 ~]#
[root@ip-172-31-6-112 ~]# date1 || cal
-bash: date1: command not found
    September 2022
Su Mo Tu We Th Fr Sa
                1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

[root@ip-172-31-6-112 ~]#
```



- With the help of these symbols we can perform inline programming or syntax here.

```
[root@ip-172-31-6-112 ~]# test $? -eq 0 && echo "done ok" || echo "error not"
done ok
```

- It is just like if and else condition.
- Test is just the keyword here and instead of test we can use [ ] brackets here that will work the same as the test keyword

```
[root@ip-172-31-6-112 ~]# date
Sat Sep  3 09:57:54 UTC 2022
[root@ip-172-31-6-112 ~]# [ $? -eq 0 ] && echo "done ok" || echo "error not"
done ok
[root@ip-172-31-6-112 ~]# date1
-bash: date1: command not found
[root@ip-172-31-6-112 ~]# [ $? -eq 0 ] && echo "done ok" || echo "error not"
error not
[root@ip-172-31-6-112 ~]#
```

- And if we want to check if the 'hi.txt' file exists or not then we have the -e option in the test command.

```
[root@ip-172-31-6-112 ~]# ls
a.txt b.txt c.txt hi.txt my.sh
[root@ip-172-31-6-112 ~]# test -e hi.txt
[root@ip-172-31-6-112 ~]# echo $?
0
[root@ip-172-31-6-112 ~]# test -e hi1.txt
[root@ip-172-31-6-112 ~]# echo $?
1
```

- So we can use this knowledge in our script and make it better.

```
root@ip-172-31-6-112:~
read -p "enter ur file name : " myfile

[ -e $myfile ] && echo "file already exists" || touch $myfile

~
~
```

- Similarly we have many more options in the test command that we can check with the command `man test`.

```
root@ip-172-31-6-112:~
echo -n "enter ur name : "
read myname

echo "your name is $myname"

~
~
~
```

- But in the above script if the input is none or the variable is unset, in any case, What we can do, we can set the default value of this variable in any

case if the input is none then we can return this value.

```
root@ip-172-31-6-112:~  
echo -n "enter ur name : "  
read myname  
  
unset myname  
  
#x=${myname:-vimal}  
x=${myname:=vimal}  
  
echo "your name is $x"  
~  
~  
~
```

- Similarly, like this, curly braces have much more capabilities. Below are more examples:

```
[root@ip-172-31-6-112 ~]# x="linux world"  
[root@ip-172-31-6-112 ~]# echo $x  
linux world  
[root@ip-172-31-6-112 ~]# echo ${x}  
linux world  
[root@ip-172-31-6-112 ~]# echo ${x^}  
Linux world  
[root@ip-172-31-6-112 ~]# echo ${x^^}  
LINUX WORLD  
[root@ip-172-31-6-112 ~]# y="VImaL"  
[root@ip-172-31-6-112 ~]# echo ${y}  
VImaL  
[root@ip-172-31-6-112 ~]# echo ${y,}  
vImaL  
[root@ip-172-31-6-112 ~]# echo ${y,,}  
vimal  
[root@ip-172-31-6-112 ~]# echo $x  
linux worldddd  
[root@ip-172-31-6-112 ~]# echo ${x%}  
linux worldddd  
[root@ip-172-31-6-112 ~]# echo ${x%d}  
linux worlddd  
[root@ip-172-31-6-112 ~]# echo ${x%dd}  
linux world  
[root@ip-172-31-6-112 ~]# echo ${x%dd}s  
linux worlds  
[root@ip-172-31-6-112 ~]# echo ${x%dd}sss  
linux worldsss  
[root@ip-172-31-6-112 ~]#
```

- Suppose there is url we stored in a variable and we want to change the extension of the file from there to html. There also we can use the ‘%’

symbol.

```
[root@ip-172-31-6-112 ~]# x="/one/two/three/hello.txt"
[root@ip-172-31-6-112 ~]# echo $x
/one/two/three/hello.txt
[root@ip-172-31-6-112 ~]# echo $x
/one/two/three/hello.txt
[root@ip-172-31-6-112 ~]# echo ${x}
/one/two/three/hello.txt
[root@ip-172-31-6-112 ~]# echo ${x%.txt}
/one/two/three/hello
[root@ip-172-31-6-112 ~]# echo ${x%.*}
/one/two/three/hello
[root@ip-172-31-6-112 ~]# echo ${x%.*}.html
/one/two/three/hello.html
[root@ip-172-31-6-112 ~]#
```

- Similarly, we can modify the URL also with the % symbol.

```
[root@ip-172-31-6-112 ~]# y="http://www.google.com:443/data/hello.html"
[root@ip-172-31-6-112 ~]# echo $y
http://www.google.com:443/data/hello.html
[root@ip-172-31-6-112 ~]# echo ${y}
http://www.google.com:443/data/hello.html
[root@ip-172-31-6-112 ~]# echo ${y%/*}
http://www.google.com:443/data
[root@ip-172-31-6-112 ~]# echo ${y%/*/*}
http://www.google.com:443
[root@ip-172-31-6-112 ~]# echo ${y%:/*/*}
http://www.google.com
```