**S. No. Content pg.no**

**1 sed 121**

**Filters**

**grep vs egrep**

**in egrep u can search for more than one strings at same time**

**ex: egrep “key1|key2|key4”**

**| is a pipe**

**grep -c**

**Text manipulation**

**Scripting**

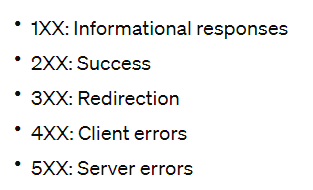
Interview Question?

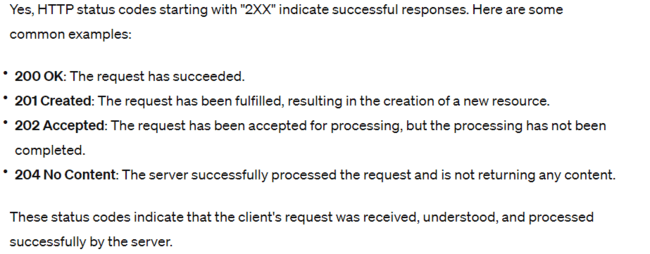
1. Write a script that finds all 0 size files in /etc directory and list it.

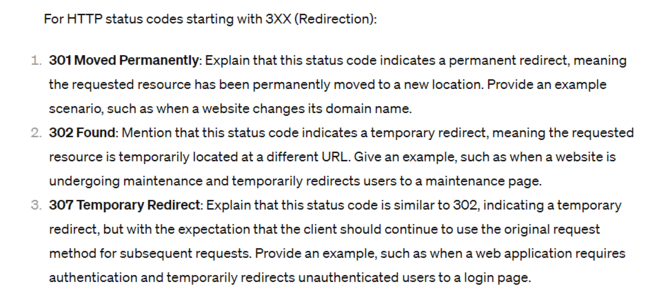
**OS**

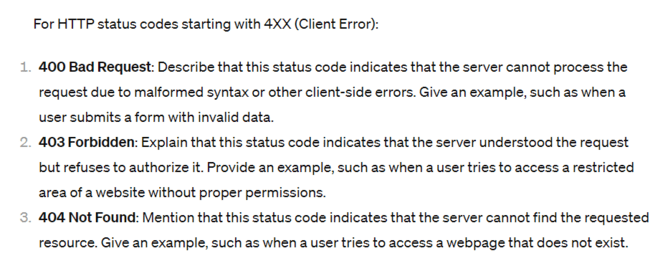
* An operating system(OS) is the software that manages computer hardware and software resources and provides common services for applications installed on a computer.
* For hardware functions such as input, output, and memory allocation, the operating system acts as an intermediary between application software and computer hardware.

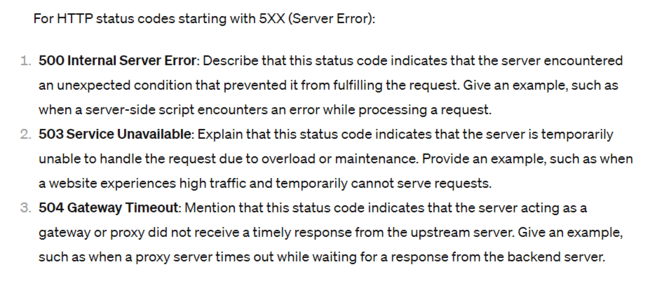
**HTTP Status Code:**

****

****

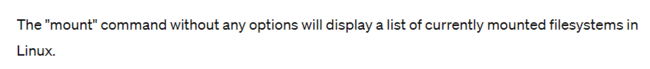
****

****

****

**File Systems**

****

****

* Each disk must be formatted with a file system.
* Some file systems are optimized for large files, others for network files, etc.
* Linux and Windows use different systems:
  + Common local file systems:
    - Windows: NTFS
      * "New Technology File System." It's a proprietary journaling file system developed by Microsoft for their Windows NT operating system family. NTFS offers features like file permissions, encryption, compression, and support for large file sizes and volumes.
    - OS/2 & MAC: HFS and HFS+ HPFS, APFS

### [OS/2](https://en.wikipedia.org/wiki/OS/2)

* The name stands for "**Operating System/2**", because it was introduced as part of the same generation change release as IBM's "Personal System/2 (PS/2)".
* OS/2 is a series of computer operating systems, initially created by Microsoft and IBM under the leadership of IBM software designer Ed Iacobucci.
* **HFS** stands for Hierarchical File System and is the filesystem used by the Mac Plus and all later Macintosh models.
* MacOS 8.1 and newer support a filesystem called HFS+ that's similar to HFS but is extended in various areas.
* **HPFS**(High Performance File System) The file system introduced with OS/2 Version 1.2 that handles large disks (2TB volumes; 2GB files) and long file names (256 bytes). It coexists with the existing FAT system.
* Apple File System **(APFS),** the default file system for Mac computers using macOS 10.13 or later, features strong encryption, space sharing, snapshots, fast directory sizing and improved file system fundamentals.
* APFS was introduced in macOS High Sierra (10.13) and is optimized for modern storage systems like solid-state drives (SSDs) and flash storage.
  + - Unix/Linux: xfs(extended file system),btrfs(B-tree file system), ext3, ext4,
      * In CentOS 7, the default filesystem is usually Ext4 (Fourth Extended Filesystem). Ext4 is a widely used filesystem in Linux distributions due to its stability, performance, and backward compatibility with its predecessors (Ext2 and Ext3). However, CentOS 7 also supports other filesystems like XFS and Btrfs, which can be chosen during installation or set up post-installation.
      * Btrfs is a modern file system with features like
        + snapshots(create point-in-time read only copies of the file system.),
        + checksums(ensure data integrity by verifying the integrity of stored data blocks.), and
        + built-in RAID support(provides a flexible and integrated solution for data protection and fault tolerance, allowing users to configure and manage RAID (Redundant Array of Independent Disks) arrays directly within the file system.). It's gaining popularity for its advanced capabilities and flexibility.
      * Btrfs = modern file system, ext4 = default file system
    - CD/DVD: ISO 9660
    - VMware: VMFS
  + Network file systems:
    - Linux: NFS
      * User - Client
      * NFS allows a user on a client computer to access files over a network as if those files were located on the client's own hard drive.
    - Windows: CIFS (SMB)
      * CIFS (Common Internet File System) is an updated version of the Server Message Block (SMB) protocol, which is used for sharing files, printers, and other resources over a network.
    - Web:
      * FTP (File Transfer Protocol)
      * Client - Server
      * It's a standard network protocol used for transferring files between a client and a server on a computer network.

**File System Hierarchy**

* All Linux systems have a Directory Structure that starts at the root directory.
* The root directory is represented by a forward slash i.e /
* /bin: The /bin Directory contains the essential user command binaries (also called binaries) fundamental to the system’s operation.
  + required for basic system functioning, such as ls, cp, mv, cat, grep, and others.
* /sbin: /sbin directory contains essential system administration binaries.
  + primarily intended for use by the system administrator (root) or for administrative tasks.
  + tasks such as system configuration, management, and maintenance such as fdisk, ifconfig, shutdown, and reboot..
* /lib: binaries found in /bin and /sbin use libraries located in /lib.
  + essential shared libraries (also known as dynamic-link libraries) that are required for the proper functioning of programs and applications on the system.
  + Shared libraries contain code that can be used by multiple programs simultaneously, helping to reduce duplication of code and conserve memory.
* /opt: the purpose of /opt is to store any optional software in Linux.
* /boot: The /boot directory contains all files that are needed to boot the computer.
* /etc: all of the machine / software specific configuration files should be present in this directory.
* /home: this directory is used by linux users say (eg. Tharun) to store personal or project data.

This directory also contains home directory of users created in linux.

* /media: The /media directory serves as a mount point for removable devices eg. CD ROM, Pen Drive etc.
* /tmp: Applications or users should use /tmp directory for storing Temporary Data.
* /USR: Unix System Resources, this contains shareable and mostly read only data.

Eg. /usr/bin contains most commands that we have used..

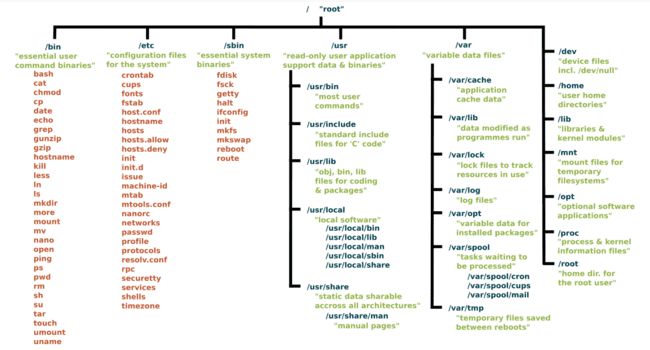
* /var: (Variable Data) this folder is used to store Files whose SIZE is unpredictable eg.

Log Files

SPOOL Files (eg. The print commands received)

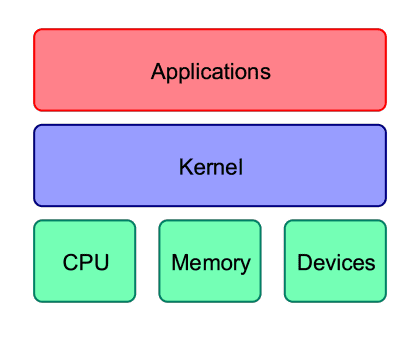
Cache Files.

* Note: /var/log directory serves as a central point to contain all log files.



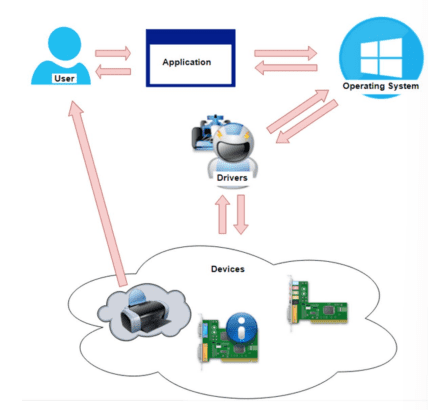
**Kernel**

* The kernel is the core of a computer’s operating system, with complete control over everything in the system.
  + Kernel is the third program loaded on start-up after BIOS and BootLoader
  + The kernel is responsible for low-level tasks such as disk/network management, task management, and memory management.
  + When a process makes a request to the operating system kernel for a service, it is called a system call.
  + for input/output operation, process management, accessing hardware resources etc.

****

**Device Drivers**

* Applications must use a driver to access a device.
  + A device driver is a software program that controls a particular type of hardware device that is attached to a computer.
  + For example, if you get a new printer, you will usually have to install a driver for that printer.
  + Drivers often have configuration/monitoring tools that load in the notification area (on the task bar just to the left of the clock).
  + Other drivers are generally invisible to the user, but they can be managed using Windows Device Manager.

****

**Services**

* A system service is software loaded into memory that works in the background for a specific process.
  + Some services load at startup and some load as required or manually.
    - **System Service:** A service needed to run the OS
    - **Network Service:** A service for the network devices and dependent components
    - **User Service:** Services that user operations depend on
  + There is a service control process to watch over all services.
  + Some services depend on other services to work.

**Class Topics**

**Why to create aws account as a root user option instead of IAM user?**

* The “root user” in the context of AWS typically refers to the initial AWS account holder who has full administrative access to all AWS services and resources within the account.
* This account is created when you sign up for AWS.
* The “root user” has elevated privileges, similar to the superuser or administrator on a traditional operating system.
* It’s important to note that AWS strongly recommends not using the root user for everyday tasks or long-term access.
* Instead, it’s advised to create an use IAM (Identity and Access Management) users with appropriate permissions for improved security and accountability.
* Reasons why using IAM users is preferred over the root user:

1. **Principle Of Least Privilege:**
   * IAM allows you to apply the principle of least privilege, meaning that users are given only the permissions they need to perform their tasks.
   * The root user has unrestricted access to all AWS resources, which can pose a security risk if compromised.
2. **Security Best Practices:**
   * IAM users have finely tuned permissions based on the tasks they need to perform.
3. **Access Key Rotation:**
   * IAM users can have access kays, and AWS recommends regular rotation of access keys for security purposes.
   * The root user also has access keys, but using IAM allows for more controlled access key management.
4. **Activity Logging and Monitoring:**
   * Each IAM user has a unique identity, and their actions are tracked in AWS CloudTrail.
5. **Enhanced Security Features:**
   * IAM supports features like multi-factor authentication (MFA), password policies, and other security features that can be configured for individual users.
6. **Granular Permissions:**
   * IAM allows you to create policies with granular permissions, specifying exactly what actions users are allowed or denied on specific AWS resources.
   * This level of granularity is not achievable with the root user.

**What is VirtualBox?**

* VirtualBox is a free and open-source virtualization software.
* Allows you to run multiple operating systems on a single physical machine.
* A Type 1 hypervisor, also known as a bare-metal hypervisor, is a virtualization technology that runs directly on the physical hardware of a host system.
  + It does not require a separate underlying operating system to function.
  + They are commonly used to create and manage virtual machines (VMs) on servers, allowing organizations to consolidate multiple workloads onto a single physical server while maintaining isolation and flexibility.
  + They offer efficient resource utilization, better performance, and lower overhead compared to Type 2 hypervisors because they eliminate the need for an additional operating system layer.
  + Examples of Type 1 hypervisors
    - VMware vSphere/ESXi,
    - Microsoft Hyper-V Server
    - Xen.
* Developed by Oracle, VirtualBox is a type-2 hypervisor,
* A Type 2 hypervisor, also called a hosted hypervisor, is virtualization software that runs on top of an existing operating system, allowing users to create and manage virtual machines within their desktop or laptop environment.
* Virtualization is a technology that allows multiple virtual instances of operating systems, servers, storage devices, or networks to run on a single physical hardware platform.

1. Cross-Platform Support

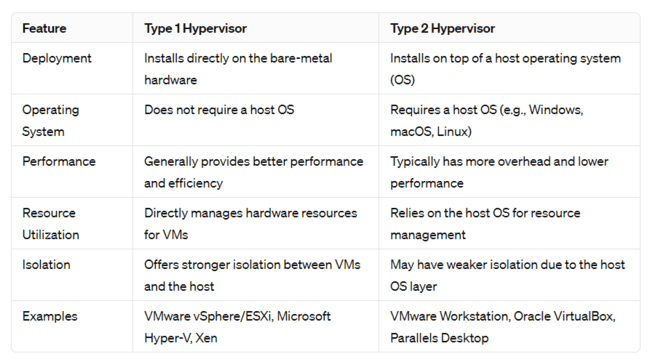
* VB is compatible with various host operating systems(windows, macOS, Linux and Solaris).
* It allows you to run VMs on different host platforms seamlessly.

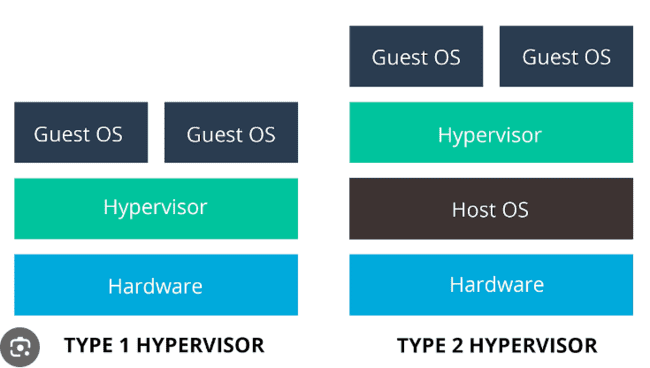
1. Snapshot and Cloning
   * VB allows you to take snapshots of virtual machines at various states.
2. Virtual Networking

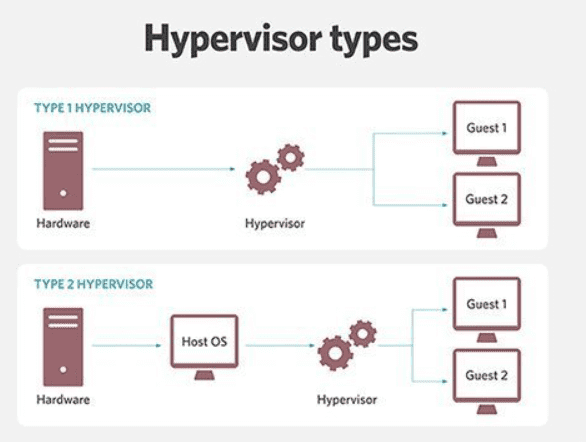
* You can configure network interfaces, set up internal networks, and connect VMs to the host machine or other VMs.

1. USB Device Support

* VB supports USB passthrough, allowing you to connect USB devices to virtual machines.







**Installing VirtualBox**

* Visit the official VirtualBox website: <https://www.virtualbox.org/>.
* Clicks downloads
* Under “VirtualBox 6.1.x platform packages,” choose the installer for windows hosts.
* Download the installer executable (e.g., VirtualBox-6.1.x-xxxxx-Win.exe).
* Leave the default settings and continue to install VirtualBox.

**Steps to create a virtual machine on VirtualBox for centos7**

* First we need to download a ISO image that serves as a way to install the operating system on the virtual machine.
* Go to the CentOS official website: https://www.centos.org/
* Navigate to the "Download" section. Look for the CentOS 7 release.
* Find the appropriate architecture and download the ISO image. For CentOS 7, the architecture is typically available in 64-bit (x86\_64).
* **Need for iso image:**
  + An ISO image serves as a way to install the operating system on the virtual machine.
  + ISO image is a file that contains a complete representation of a CD or DVD, including the file system and the files and directories of operating system.
  + This has a iso file extension and file type as Disc Image File.
  + The ISO image typically contains the installation files for an operating system (OS), such as Linux distributions (e.g., CentOS, Ubuntu) or other OS versions (e.g., Windows).
  + This allows you to install the chosen OS on the virtual machine.
  + An ISO image is a bootable medium, meaning that it can be used to start the installation process.
  + When you attach the ISO image to the VM, VirtualBox emulates a virtual CD/DVD drive, allowing the VM to boot from the ISO image as if it were a physical CD/DVD.
  + You can upload the ISO image to a server or cloud storage and use it to install the OS on a VM.
* **Setting VB**
  + Click on “New” option after opening oracle VM VirtualBox Manager.
  + “Create Virtual Machine” dialogue box will be opened.
  + Select the dropdown “Name and Operating System”.
    - At the name field provide a name for VM ex., Linux.
    - At the folder box provide the path for the VB Virtual machine
      * Ex: C:\Users\Lenovo\Downloads\VirtualBox VMs
    - At the ISO image box select path for the iso image.
      * Ex: C:\Users\Lenovo\OneDrive\Desktop\CentOS-7-x86\_64-Everything-1804.iso
    - At the type box select as Linux.
    - Version as Red Hat (64-bit)
  + Now select the “Hardware” dropdown.
    - Hardware must be allocated to VM.
    - Base Memory must be added this means RAM not disk memory.
      * Default of 2GB will be assigned
      * But can be increased and decreased based on the RAM available in the system.
      * To check the available RAM got to “Task Manager”
      * Under second icon i.e., “Performance” go to “Memory” and check the “Available” option and select less than that to allocate memory.
      * Now for Processors: allocate 1CPU.
  + There will be no changes under “Hard Disk”.
  + You can click the “Skip Unattended Installation” checkbox under “Name and Operating System” dropdown.
  + Now setting up VM for Linux in VB is done.
* **Installing CentOS Linux distribution on VM**
  + In the Oracle VM VirtualBox Manager select the Linux virtual machine and click on “Start” button.
  + Now the installation for the CentOS on the virtual machine starts running.
  + Follow the commands and click “install CentOS 7” then installation starts.
  + Note: cursor will be locked inside virtual Box, to get cursor out back to windows use Right CTRL KEY.
  + After the CentOS is installed we see a dialogue box saying “WELCOME TO CENTOS 7.” and asks to select language and click continue.
  + Under “SOFTWARE” section click “SOFTWARE SELECTION”.
    - If you have installed “minimal version” all the software options are not available only “minimal install” option will be there and just select that.
    - If you have installed “everything version” under base environment select “Minimal Install” and select Add-Ons as “Development Tools” and “System Administration Tools”.
    - Then click on done on top left corner.
  + Open “INSTALLATION DESTINATION” don’t make any changes and click done simply.
  + Then continue to click “Begin Installation”.
  + Now click on “ROOT PASSWORD” and create a password for root user then click on done.
  + CentOS Linux distribution is ready to use on Virtual Machine.

**Creating a EC2(Elastic Compute Cloud) instance on AWS Console**

****



* Search for EC2 on the search bar of AWS Console after sign in.
* Select EC2 which mentions “Virtual Services in the Cloud”.
* On the left panel scroll down to select a “Key Pairs option”.
* What is the need to add a “Key Pair” to a ec2 Instance.
  + SSH Access: Key pairs are primarily used for secure shell (SSH) access to your EC2 instance.
  + What is SSH(Secure Shell) access?
    - is a cryptographic network protocol used to establish a secure communication between two computers typically a client and a server over a potentially unsecured network.



* + - provides a secure way to access and manage remote devices, servers, and systems.



* + - Used for remote administration, file transfers, and tunneling network connections.



* + - SSH protocol itself is an open standard and is available to everyone.
    - Created by Finnish computer scientist and now it’s a collaborative effort of many developers.
    - Ssh replaced insecure protocols like Telnet and unencrypted FTP with more security and encryption.
    - Key features:
      * Encryption: encrypts the data transmitted between the client and the server, providing a secure way to communicate over an insecure network such as the internet.
      * Authentication: SSH uses a combination of cryptographic keys (asymmetric key pairs) to authenticate the client and server. This is commonly done through the use of public and private key pairs. The client possesses the private key, while the server has the corresponding public key. This method enhances security compared to traditional password-based authentication.



* + - * + What is asymmetric key pairs?

Asymmetric key pairs, also known as public-key cryptography, involve the use of two different keys a public key and a private key.

Public key – to encrypt message



Private key – to decrypt message



Information encrypted with one key can only be decrypted by the other key in the pair using some of the mathematical algorithm like RSA (Rivest-Shamir-Adleman) / DSA (Digital Signature Algorithm).

For ex, if Bob wants to send a secure message to Alice, he would use Alice’s public key (this key involves a part of a pattern from Alice’s private key) to encrypt the message (this step involves mathematical algorithm).

Alice generates a key pair, consisting of a public key and a private key. These keys are mathematically linked but computationally infeasible to derive one from the other.

Once encrypted, only Alice, who possesses the corresponding private key, can decrypt and read the original message.

Asymmetric key pairs are widely used in various security protocols, including HTTPS for secure web communication and secure email communication.

* + - * SSH is a platform-independent and widely supported on various operating system, including Linux, Unix, macOS, and Windows.
      * SSH operates on the client-server model. When a user wants to connect to a remote server or device using SSH, they typically use an SSH client.
      * The server, which has the SSH server software running, authenticates the client and grants access if the authentication is successful.
      * Using ssh on local system:
        + Open cmd or powershell

Cmd vs powershell

Cmd executed using cmd’s scripting language. commands and scripting capabilities are relatively basic compared to ps.

Cmd relies on traditional DOS commands and executables.

Cmd scripting capabilities are limited, doesn’t support advanced automation or scripting features found in modern shells.

More sophisticated scripting language designed for system administration and automation based on .NET Framework. Supports advance automation, task scheduling and remote management.

Ps is object-oriented meaning deals with objects rather than plain text. Commands output objects, making it easier to manipulate and filter data.

* + - * + ssh username@hostname
        + replace username with your username on the remote server and hostname with the IP address or domain name of the server.
        + After entering the command, you’ll be prompted to enter your password for the specified user on the remote server.
        + Instead of entering a password every time, you can set up SSH key based authentication, which is more secure and convenient.
        + Generate an SSH key pair on your local machine:



* + - * + ssh-keygen -t rsa -b 2048
        + ssh-keygen: this is the command-line utility for generating SSH key pairs.



* + - * + -t rsa : Specifies the type of key to create, in this case, RSA.
        + -b 2048: Specifies the number of bits in the key. In this example, it’s 2048 bits, which is a common and secure choice for RSA keys.
        + After entering the command ssh-keygen
        + Asks to select a file to save the key by default it suggests a file if ok with that press enter.
        + Then asks enter passphrase(password for private key while accessing means adding ext) (empty for if don’t want passphrase simply press enter):
        + This is asking if you want to secure your private key with a passphrase. A passphrase adds an extra layer of security. If you enter a passphrase, you’ll need to provide it every time you use the private key.
        + Then a pair of public and private keys are downloaded to selected file public key = <filename>.pub private key = <filename>



* + - * + Private key file must be kept secure on your local machine. It should not be shared or exposed to others. It is used to prove your identity when connecting to a remote server.



* + - * + Public key (.pub) file is shared with the remote server. It can be added to the ‘~/.ssh/authorized\_keys’ file on the server.



* + - * + The public key is used by the server to verify that the person connecting has the corresponding private key.
        + Copy the public key to the remote server:
        + ssh-copy-id username@hostname.

When you run this command, it will prompt you for your password on the remote server.

After providing the password, the ssh-copy-id command will append your public key to the ~/.ssh/authorized\_keys file on the remote server.

This allows you to authenticate to the remote server using your private key without having to enter a password.

* + - * + To connect to a remote server using SSH with your private key:
        + ssh -i path/to/downloads username@remote\_server\_ip
        + replace ‘path/to/downloads’ with the actual path to your private key file, ‘username’ with your remote server username, and ‘remote\_server\_ip’ with the IP address or hostname of the remote server.
        + If the server is configured to accept your public key, and you provide the correct private key, you should be able to connect without entering a password.
        + Once connected, you can execute commands on the remote server as if you were using a local terminal.
        + To exit the SSH session, simply type ‘exit’ and press Enter.
      * To establish an SSH connection, a user needs the following:



* + - * + **Hostname or IP address of the remote server:** This is the address of the system they want to connect to.



* + - * + **SSH username:** The username used to log into the remote system.



* + - * + **Authentication method:** This involves providing the appropriate credentials, such as a password or, more securely, using a private key associated with the user’s public key stored on the server.



* Instance Recovery: if you lose access to your EC2 instance due to a lost private key, you may have to terminate the instance and launch a new one.
* Now click on “Create Key Pair” option on top right corner.
* Select any name for the key pair.
* Under “Key pair type” select any of the public key encrypting algorithm either RSA or ED25519 mostly RSA is taken.
* Next under “Private key file format” there are two options like .pem and .ppk.
  + .pem vs .ppk
  + .pem and .ppk files can store private keys for SSH, they have different formats and are associated with different applications.



* + **Format:**
    - **.pem (Privacy Enhanced Mail):** This format is a widely used container format that can include various types of data, not just key pairs. In the context of SSH, a .pem file typically contains a private key or a certificate. The file is encoded in base64 with delimiters such as “-----BEGIN CERTIFICATE-----” and “-----END CERTIFICATE-----“.



* + - **.ppk (PuTTY Private Key):** This format is specific to the PuTTY SSH client on Windows. It is a proprietary format used by PuTTY to store private keys. Unlike .pem, .ppk files have their own format and do not use the base64 encoding with delimiters.

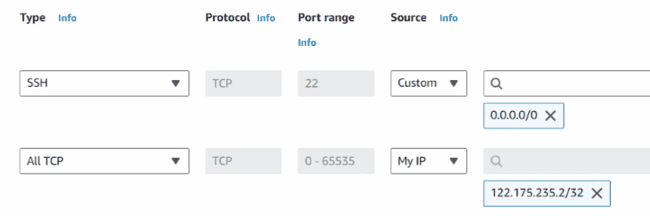


* + **Applications:**
    - **.pem:** it is a more generic format and can be used with various applications that support PEM encoding, including OpenSSH and OpenSSL.



* + - **.ppk:** It is specific to PuTTY, a popular SSH client for Windows. PuTTY does not directly support .pem files for private keys.



* + **Conversion:** 
    - You can convert a .pem file to a .ppk file using PuTTYgen, a key generator tool that comes with PuTTY. PuTTYgen can import a .pem private key and then save it as a .ppk file.
    - Conversely, you can convert a .ppk file to a .pem file using PuTTYgen as well. PuTTYgen allows you to export a private key in various formats, including .pem.
* Therefore, select .pem while using OpenSSH connection(Windows PowerShell or linux) through local machine and .ppk file while using PuTTY client on windows.
* Click on create key pair then a green pop up shows up saying “Successfully created key pair” and the respective key file will also be downloaded and must be kept safe.
* Now click on “AMI Catalog” that appears on bottom left of the scroll bar.
  + In EC2 (Amazon Elastic Compute Cloud), and Amazon Machine Image (AMI) serves as a template for the virtual server, known as an instance.
  + AMIs are pre-configured with specific operating systems (e.g., Amazon Linux, Ubuntu, Windows) by selecting an AMI, you are specifying the base environment for you EC2 instance.
  + AMI determines the initial configuration, software stack, and environment of your virtual server.
* In this case we select CentOS 7(x86\_64) free tire select and continue then click on Launch instance with AMI.
  + What does x86\_64 mean?
    - 64-bit architecture based on the x86 instruction set.
    - **x86:** Originally, the x86 architecture referred to a series of instruction set architectures based on the Intel8086 microprocessor. Over time, it has become standard architecture used by many processors, including those from Intel and AMD.
    - **64:** Indicates that the architecture is 64-bit. This means that the processor is capable of handling. 64-bit memory addresses, allowing for larger amounts of RAM to be addressed compared to 32-bit architectures.
* Provide the name and instance type(free tier) , browse for the key pair from the device and click on launch instance.
* Select the instance tab and refresh to see if allocated.
* Now add security groups for the instance under security tab.
* Click the link and select “Edit inbound rules” .
  + Why inbound rules?
  + Acts as a virtual firewall that controls the traffic to and from your instances.
  + In bound rules within the security group determine what incoming network traffic is allowed or denied to reach your instance.
  + **Access Control:** inbound rules allow you to control which IP addresses or ranges are permitted to connect to your EC2 instance.
  + **SSH Access:** If you plan to connect to your CentOS Linux instance using Secure Shell (SSH), you’ll need to add an inbound rule to allow incoming traffic on port 22 (the default SSH port). This rule enables you to establish a secure connection to your instance for remote management.
  + **Application Services:** If your CentOS instance is running specific services or applications (e.g., a web server, database server), you’ll need to define inbound rules to allow traffic on the corresponding ports. For example, if your web server is running on port 80 for HTTP, you’ll need to add an inbound rule to permit traffic on port 80.
  + **Default Deny Principle:** security groups follow the principle of default deny, meaning that all inbound traffic is denied by default unless explicitly allowed. Therefore, if you want to receive any inbound traffic on your instance, you must create and configure inbound rules to permit that traffic.
* Select a “Add rule” option
* Now under type select SSH and protocol as TCP and port range as 22 (default SSH port) and source as MyIP then click on save rules.
* 
* SSH uses the TCP (Transmission Control Protocol) for communication. Selecting TCP ensures that the rule is applied to the correct transport protocol.
* Now your EC2 instance is ready to use.

**Connecting to EC2 Instance Using PuTTY**

* From the above steps your online instance for Linux using centOS is ready.
* Copy Public IPv4 DNS under networking tab on the ec2 instance created.
* Now open PuTTY configuration click on “Session” and copy the public ipv4 DNS address under “Host Name (or IP address)” bar.
* Now under “Connection>SSH>Auth>Credentials” browse (under “Private key file for authentication” bar) the private key downloaded while creating ec2 instance to authenticate with public key stored on the ec2 instance.\
* The click on open and accept to connect.
* Now give the login as centos default username by ec2.
* Then you will be authenticated with the private key if successful you start working on linux commands remotely using ec2 Linux instance.
* You can terminate the instance under “instance sate” under ec2 in aws console.

**Connecting to EC2 Instance using scp(secure copy) for File Transfer:**

* you can use various methods. Here, I'll outline a common approach using the scp (secure copy) command, which is part of the SSH (Secure Shell) suite. This method assumes you have SSH access to your EC2 instance.

1. Install an SSH client on your Windows machine:

* + You can use a tool like PuTTY or use the Windows Subsystem for Linux (WSL) if you have it installed. If you are using WSL, you can directly use the scp command from the Linux terminal.

2. Locate the file you want to transfer:

* + Make sure you know the full path of the file on your Windows machine.

3. Open a terminal or command prompt:

* + If you are using PuTTY, open a PuTTY terminal. If you are using WSL, open a WSL terminal or command prompt.

4. Use scp to copy the file:

* + The basic syntax for using scp is:

**scp -i path/to/your/key.pem path/to/local/file username@ec2-instance-ip-or-dns:/path/on/ec2**



* -i: Specifies the path to your private key file.
* path/to/local/file: Specifies the path to the file on your local machine.
* username: The username you use to connect to the EC2 instance (e.g., ec2-user for Amazon Linux instances).
* ec2-instance-ip-or-dns: The public IP address or DNS of your EC2 instance.
* /path/on/ec2: The destination path on your EC2 instance.
* For example:

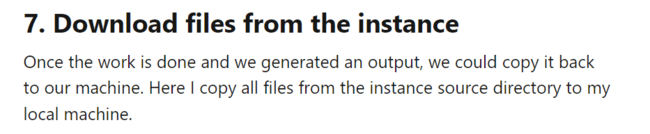
scp -i path/to/your/key.pem path/to/local/file.txt ec2-user@your-ec2-instance-ip:/home/ec2-user/

5. Authenticate with your private key:

* If prompted, provide the path to your private key when connecting.

6. Check the destination folder on the EC2 instance:

* After the transfer is complete, log in to your EC2 instance using SSH and navigate to the specified destination path to confirm that the file has been transferred.





**ssh -i path/to/your/key.pem ec2-user@your-ec2-instance-ip**

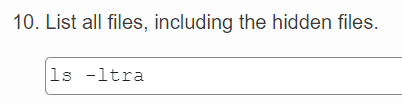
**cd /home/ec2-user/**

* Replace placeholders such as path/to/your/key.pem, path/to/local/file.txt, ec2-user, and your-ec2-instance-ip with your actual values.
* Remember to adjust security group settings to allow SSH (port 22) traffic from your local machine to the EC2 instance if you encounter connection issues.





**-t** flag shows files created based on timestamp from present to past.



**Connecting to Remote server or Machine using SSH command**

* Open a Terminal:
  + Open a terminal or command prompt on your local machine. This depends on the operating system you are using (e.g., Terminal on Linux or macOS, Command Prompt on Windows, or PowerShell).
* Check for SSH availability:
  + On windows you may need to install an SSH client. You can use the OpenSSH client that comes with windows 10 or install third-party alternatives like PuTTY ow WinSCP.
* Identify Remote Server Details:
  + To find the IP address or hostname of the remote server, you can use the following command on Windows in the Command Prompt:



* + **ping remote-server**



* + Replace ‘remote-server’ with the actual hostname or domain name of the remote server.



* + alternatively, if you know the IP address, you can use the ‘nslookup’ command.
  + **nslookup remote-server**



* + Replace ‘remote-server’ with the actual hostname or IP address of the remote server.
* OpenSSH command:
  + Use the following command syntax to connect via SSH.

**ssh username@remote-server**

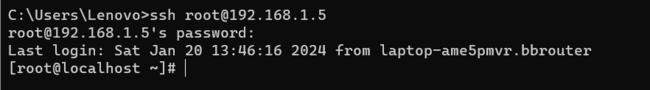


* + Replace ‘username’ with your username on the remote server.
  + Replace ‘remote-server’ with the IP address or hostname of the remote server.
* Enter Password or Use key authentication:
  + If you’re using password-based authentication, enter your password when prompted.

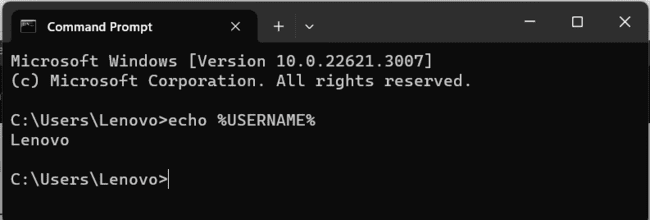


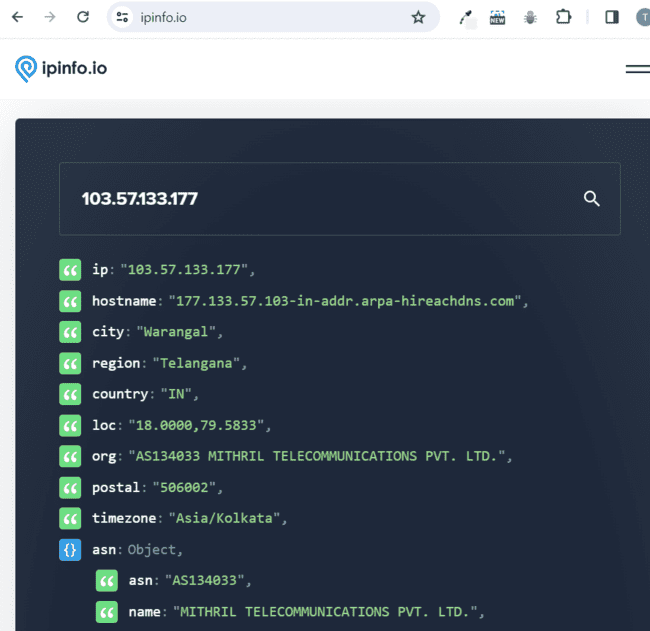
* + If you’re using key-based authentication (ssh -i <pem-key-file> <user>@<DNS>), ensure your public key is added to the remote server’s ‘**~/.ssh/authorized\_keys’** file. You won’t be prompted for a password if using keys.



****

* Now you’re successfully connected to remote machine and can run the commands needed.
* To close the connection press ‘exit’ or ‘logout’ command and press enter.
* **ssh -i <pem-key-file> <user>@DNS**
  + ‘-i <pem-key-file>’ : use of an identity (private key) file specified by the ‘-i’. <pem-key-file> specifies the path to the private key file (in PEM format) that will be used for authentication. This key file is associated with the corresponding public key stored on the remote server.
  + <user>: replace this with your username on the remote server.
  + @ : separates the username form the DNS (Domain Name System) or IP address of the remote server.
  + <DNS> : Replace this with the DNS name or IP address of the remote server.
* When you use ‘ssh username@remote-server’ without the ‘-i’ option, SSH will attempt to use the default private key files (‘~/.ssh/id\_rsa’, ‘~/.ssh/id\_dsa’, etc.) for authentication. If you have a custom key or want to use a specific one, the ‘-i’ option allows you to explicitly specify the key.
* “C:/Users/Lenovo/.ssh in your case where private keys are stored.
* To check my username and ip on my command prompt to help others connect

****

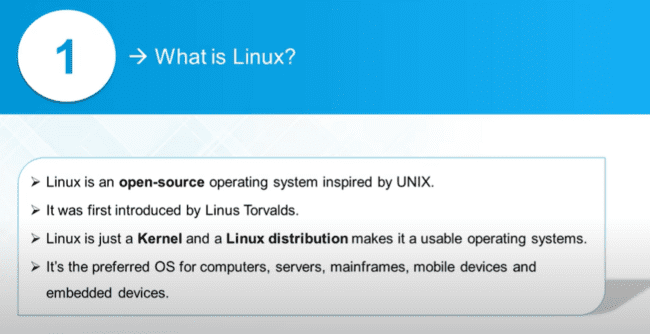
****

* Why my ip address is same everytime i disconnect and reconnect to my wifi?
  + When you connect to a Wi-Fi network, your device device typically sends a DHCP request to the router or DHCP server. The server then assigns an IP address to your device for the duration of the connection. If your device disconnects and reconnects to the same network within a certain period (before the DHCP lease expires), the DHCP server may assign the same IP address to your device to maintain consistency.

**Cloning VM**

* Easily experiment with changes without affecting the original VM.
* Creates backups for quick restoration in case of issues or data loss.

**Linux**

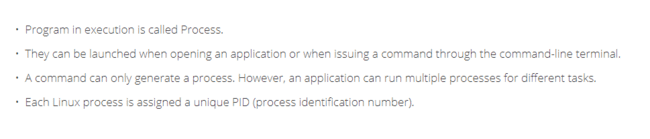
****

**Why Linux?**



* More than 30+ years and ruling industry adaptable means you can install/run/configure Linux on device of any size from a small Air pod to a Super Computer.
* **Open source / Community Support :** Linux kernel development involves contributions from a global community and collaboration of open source and quick and reliable support for Linux-related issues.
* **Server Management:** Many web servers, such as those hosting websites or applications, run on Linux-based operating systems like Ubuntu Server or CentOS.
* **Developer Environment:** Linux is the primary choice for developers using tools like Git, Docker, and programming languages such as Python and Ruby.
* **Networking Technologies:** Networking devices like routers often use Linux-based operating systems for their reliability and flexibility.
* And many more like Command-Line Efficiency, Security and Stability in running financial transactions.

**Process**

****



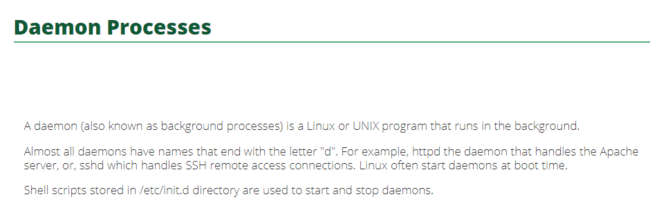
**Zombie process**

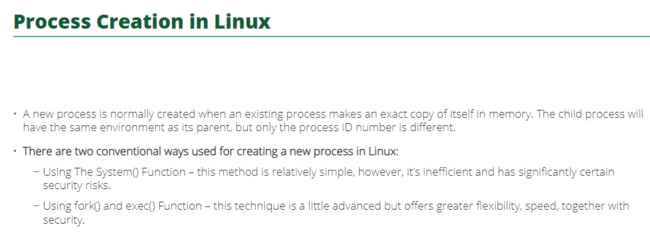


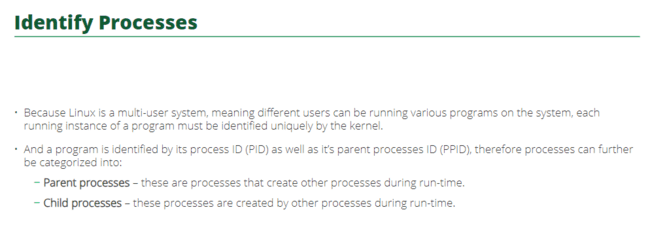
* A zombie process, also known as a defunct process, is a process that has completed execution but still has an entry in the process table.
* It's a child process that has finished executing its task, but its parent process has not yet called the system function to retrieve its exit status.
* The zombie processes may linger until the system is rebooted or the parent process is terminated.
* Zombie process is a terminated process that has not been properly cleaned up, leading to resource wastage and potential system instability if left unchecked.

**Orphan process**

* An orphan process is a process whose parent process has terminated or finished execution before the child process completes.
* When a parent process creates a child process, it typically waits for the child process to complete its execution and retrieves its exit status.
* However, if the parent process terminates prematurely, the child process may continue running as an orphan.
* Orphan processes are adopted by the init process (usually PID 1), which serves as the parent of all orphan processes on the system.
* Init periodically checks for orphaned processes and reaps them, ensuring that they do not consume system resources indefinitely.
* Orphan process is a process whose parent process has terminated, leaving it without a parent. It is eventually adopted by the init process and cleaned up to prevent resource exhaustion.

****

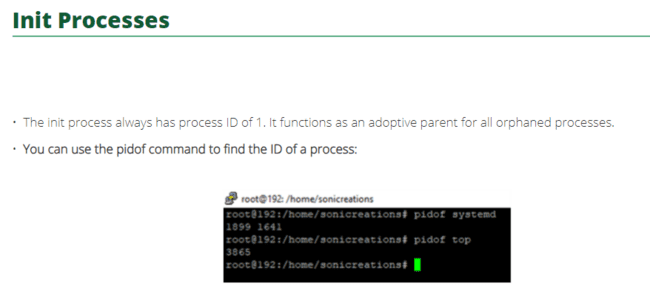
****

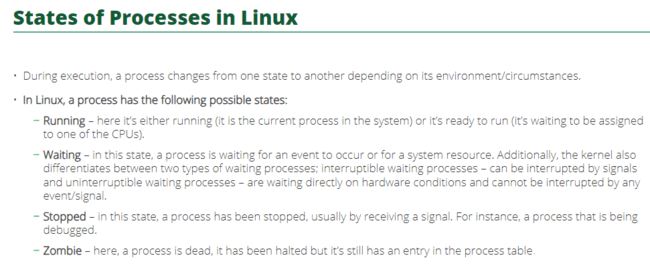
****

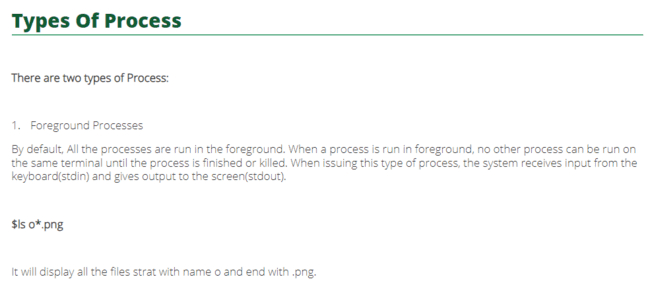
**pidof**

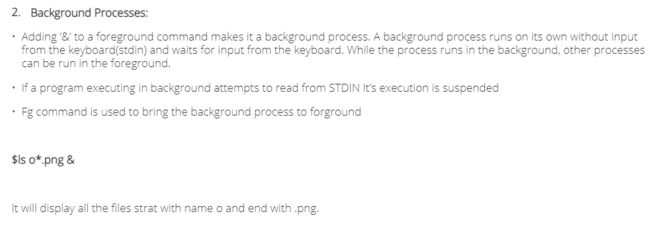
* The pidof command in Linux is used to find the process ID (PID) of a running program.
* It takes the name of a program as an argument and prints out the PID(s) of the process(es) with that name.
* If multiple instances of the program are running, pidof will print out all their PIDs, each

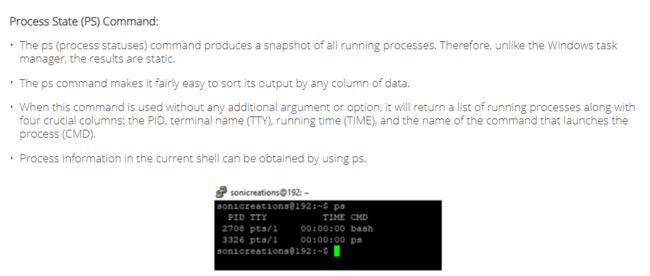


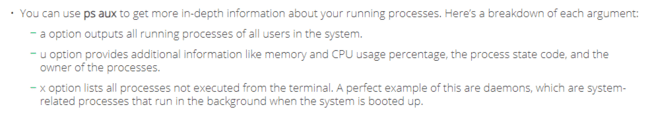
****

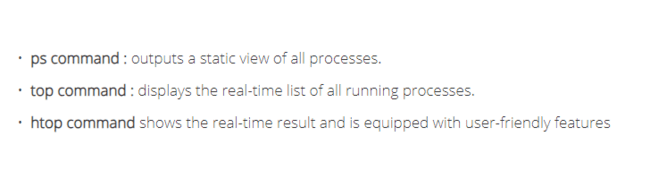
****

****

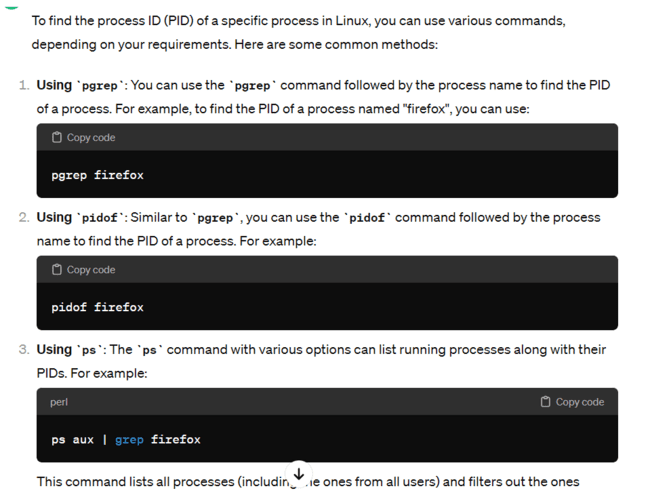
****

****

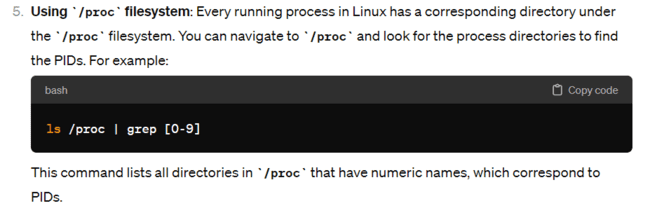
****

****

**ps aux**

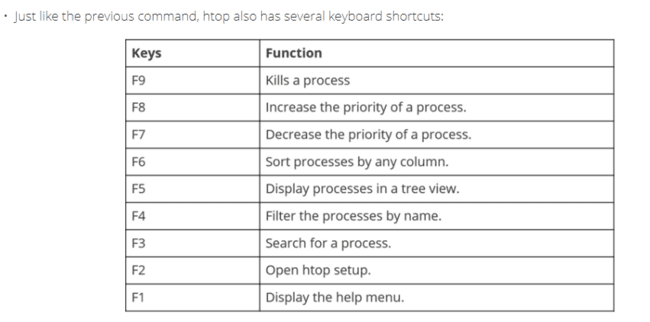
****

**/proc**

****

**ps**

to check if a process/application is running or not.

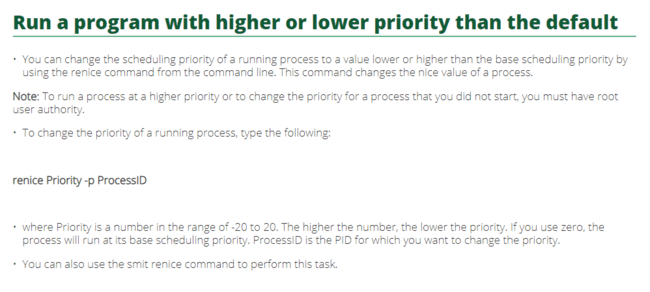
****

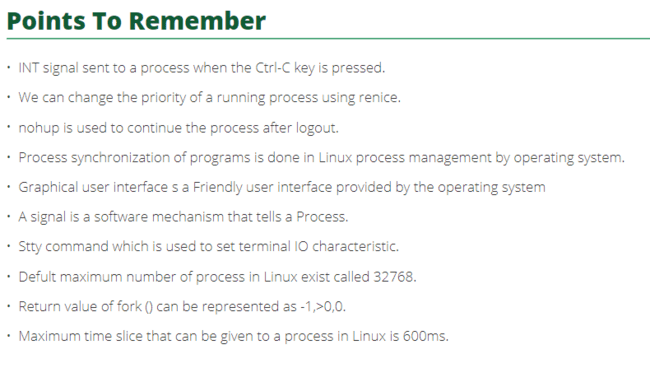
Three ways to send signal to a process:

* Using kill command
* Send signal to a process from another process
* Send signal to a process from keyboard

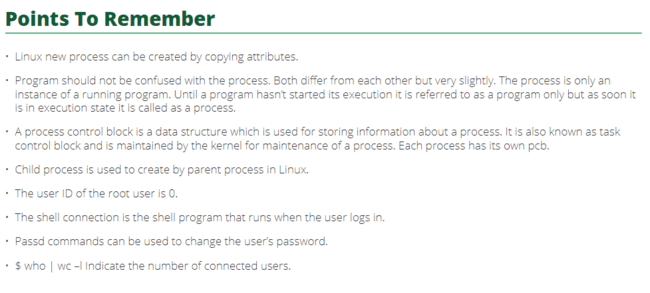
How to terminate/stop a running process?

kill command



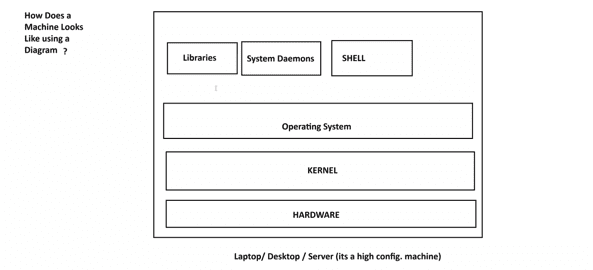


* In a multitasking system, multiple processes may be competing for CPU time.
* To fairly allocate CPU resources among these processes and ensure that each process gets a chance to execute, the operating system divides the available CPU time into small intervals called time slices.



****

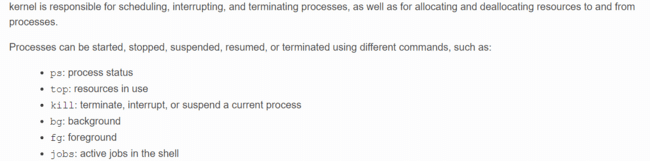
**Components of Linux:**

****

**Kernel**

Kernel is the core component of an operating system that acts as an intermediary between the hardware and software.

Manage system resources, provides essential services to software applications, and facilitates communication between hardware devices and the user-level applications.



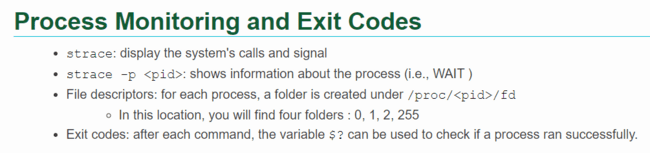
Functions of kernel:

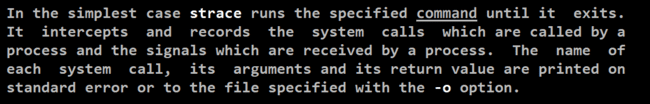
1. **Process Management:** Processes are instances of executing programs. It allocates resources, schedules tasks, and ensures proper isolation between processes.
2. **Memory Management:** Allocating and deallocating memory space for processes, and handling virtual memory addressing.
3. **Device Drivers:** It interacts with hardware devices through device drivers, enabling the operating system to communicate with peripherals such as printers, disk drivers, and network interfaces.
4. **File System Management:** File operations, including reading, writing, and organizing data on storage devices. It interprets file systems and provides an interface for user-level applications to access files.
5. **System Calls:** The kernel provides a set of system calls, which are interfaces that allow user-level applications to request services from the OS, such as file operations, process creation, and communication between processes.

Linux – Linux Kernel,

Windows – Windows NT kernel

macOS – XNU kernel





**Operating System**

An operating system is a software program that acts as an intermediary between computer hardware and the computer user.

It provides a platform for other software applications to run and facilitates communication between the hardware components and the user-level software.

Process management, memory management, file system management device drivers’ same definitions of kernel key one here is User Interface.

**User Interface:** Operating systems provide a user interface, which can be command-line-based (text-based) or graphical (GUI). The user interface allows users to interact with the computer and run applications.

**Are the operating system and kernel the same?**

No an operating system (OS) and a kernel are not the same, but they are closely related components with a computer system.

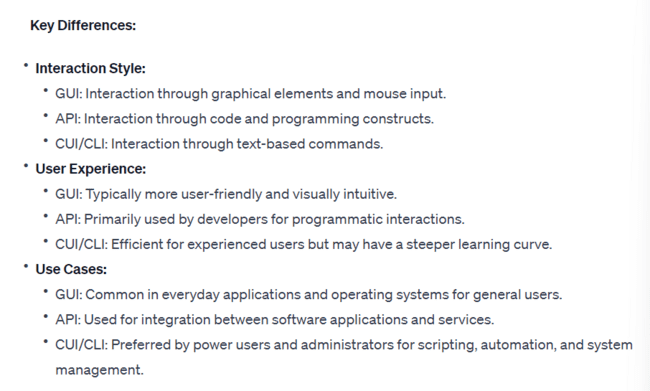
OS includes user interface, system libraries, system utilities, Api’s , GUI, CLI etc.

The kernel is a crucial part of the operating system, but it alone does not constitute the entire operating system.

The OS includes additional components that work together to provide a complete computing environment.

OS to Kernel: OS process management whereas Kernel process scheduling, OS memory management whereas Kernel memory allocation.

**GUI vs API vs CLI vs CUI**



**System Libraries**

* are collections or pre-written code that provides essential functionalities for software applications.
* These libraries contain commonly used functions and routines, allowing developers to save time by using existing code rather than writing everything from scratch.
* Ex’s : input/output operations, mathematical calculations and handling data structures.

Different ways to access a Linux server remotely from a windows?

Using some tools and terminal like:

* Putty
* Git bash
* cmd

**What is a daemon?**

* is a process or a service in a computer operating system that keeps running in background without direct user interaction, typically initiated during system startup or in response to specific event or conditions



* Ex : print spooler.



* Daemons perform various tasks, such as managing system services, handling scheduled activities, or responding to external requests.
* They operate independently of user sessions and play a crucial role in the continuous and automated functioning of a system.
* SSHD 🡨 Linux example only activates when someone tries to connect to the machine.
* httpd and chronyd are other examples.

How to check if a service is running or not?

*systemctl status service\_name*

How to start/stop any service?

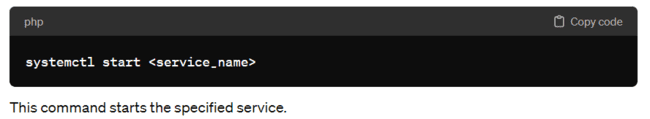
*systemctl start service\_name*

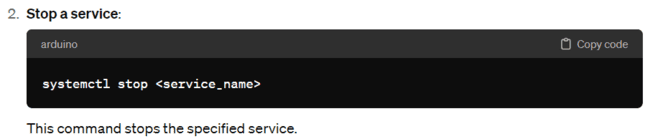
*systemctl stop service\_name*

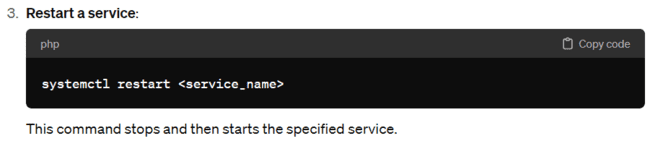
**systemctl**

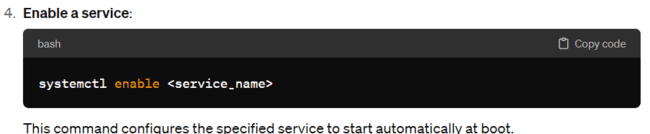
systemctl = system control

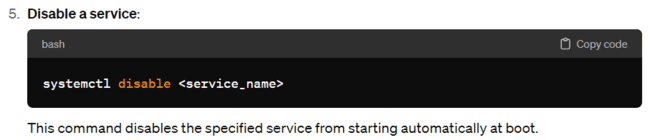
**systemctl** is a command-line utility in Linux used to manage system services (daemons) and control the systemd system and service manager.

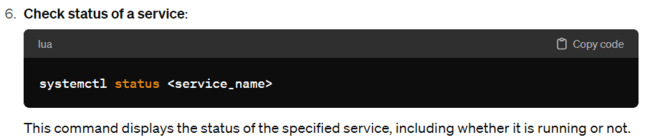


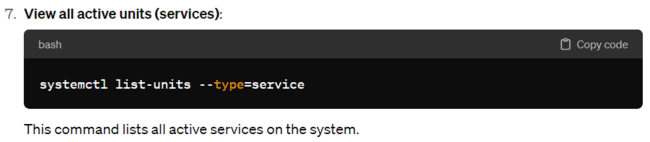












**Shell**

* is a command-line interface (CLI) that allows users to interact with an operating system or execute commands.
* In the context of a command-line interface, the shell is a program that interprets user commands and communicates with the operating system kernel to execute those commands.
* It acts as an intermediary between the user and the kernel, facilitating user input and providing a means to control the system.
* In simpler terms, a shell is a way for users to give instructions to a computer by typing commands.
* It can be a text-based interface (command-line shell) or a visual interface with buttons and menus (graphical shell).
* Examples of command-line shells include Bash (Bourne Again Shell) on Linux and Command Prompt on Windows, while graphical shells include desktop environments like GNOME and KDE and Linux or the Windows desktop on Microsoft systems.
* Browser is category and chrome is an example

Shell is a category and bash/zsh/fish shell is an example.

**Environment Variables:**

in Linux are variables that are part of the environment in which processes run. They contain information about the system environment, user preferences, and system configuration. These variables are accessible to all processes that run in that environment, and they can be used to influence the behavior of programs or provide information about the system.

Here are some common ways to work with environment variables in Linux:

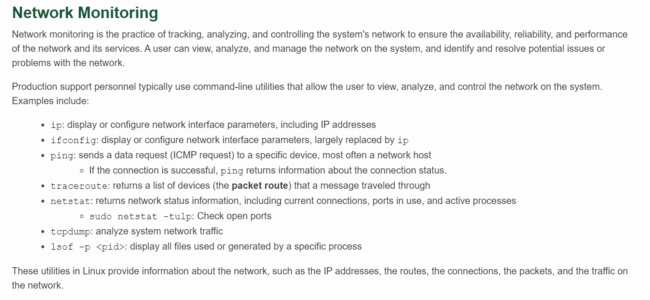
* **Viewing Environment Variables:**
  + To view all environment variables, you can use the env command or printenv command.
  + env
  + printenv
  + To view the value of a specific variable, you can use echo with the variable name preceded by a dollar sign ($).
  + echo $HOME
* **Setting Environment Variables:**
  + To set an environment variable for the current session, you can use the export command.
  + export MY\_VARIABLE="some\_value"
  + To set a variable that persists across sessions, you can add the export statement to your shell profile file (e.g., ~/.bashrc, ~/.bash\_profile, or ~/.profile).
* **Unsetting Environment Variables:**
  + To unset (remove) an environment variable, you can use the unset command.
  + unset MY\_VARIABLE
* **System-wide Environment Variables:**
  + System-wide environment variables are often set in configuration files. For example, system-wide variables can be set in /etc/environment or /etc/profile.
* **Special Environment Variables:**
  + There are some special environment variables commonly used, such as PATH (specifies directories to search for executable files) and HOME (specifies the user's home directory).
* **Temporary Environment Variables (without export):**
  + You can set a variable for a single command without exporting it to the environment.
  + MY\_VARIABLE="temporary\_value" command\_to\_execute

Remember that changes made to environment variables in a particular shell session will only affect processes started from that session. If you want changes to persist across sessions, you'll need to set them in appropriate shell profile files.

**Configuration Files:**

* are plain text files that store settings and preferences for various software applications and the user environment.
* They allow users to customize the behavior of their system, shell, and applications according to their preferences.
* Examples:
  + ~/.bashrc: Configuration file for the Bash shell.
  + ~/.bash\_profile: Executed when a user logs in.
* Location: Configuration files are typically located in the user’s home directory (‘~’) and start with a dot, making them hidden by default.
* Editing: Users can modify these files using a text editor to tailor system and application settings.

**Network**



ICMP (Internet Control Message Protocol)

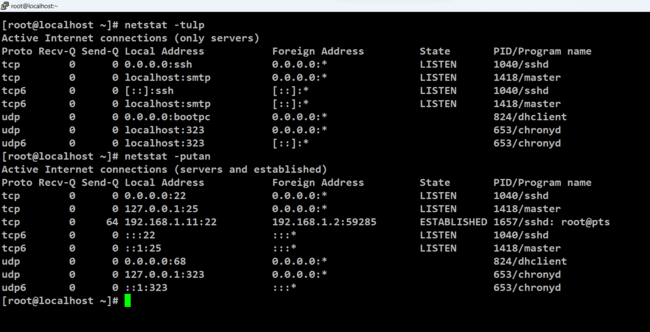
Which command is used to get info about ports?

netstat (network statistics)

difference between netstat -tulp and netstat -putan?

netstat -tulp only displays active servers/

netstat -putan show all the active servers and established connections.

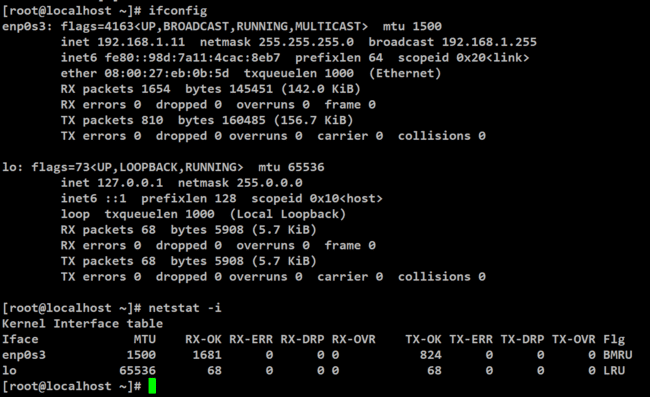


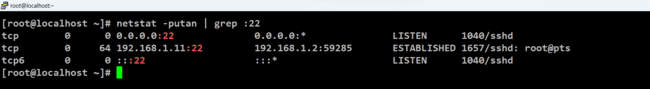
how to check open port on linux?

netstat -putan | grep :22 (this is for port 22 similarly can be used for other ports)

how to check network interfaces in linux?

ifconfig and netstat -i command





**System**



**Enabling network connectivity on Linux**

* Check Network Interface Status:
  + Open a terminal and use the following command to check the status of network interfaces:
    - Ifconfig – view ip address
      * ifconfig stands for “interface configuration” and this command allows you to view and configure network interfaces, including assigning IP addresses, enabling or disabling interfaces, setting the broadcast address, and more.
* Alternatively, you can use the ‘ip’ command:
* ip addr – view ip address
* identify the network interface (form above output) you want to enable, such as ‘eth0’ or enp0s3.
* Activate the network Interface:
  + ifconfig eth0 up

(or)

* + ip link set etho up
* Obtain an IP Address:
  + Using DHCP (Dynamic Host Configuration Protocol):
    - DHCP is a network protocol that automatically assigns IP addresses to devices on a network.
    - Run the following command to request an IP address from the DHCP server for the specified network interface (replace ‘eth0’ with your interface):
      * dhclient eth0
* The DHCP client (dhclient) will send a request to the DHCP server on the network, and if successful, it will receive an IP address, subnet mask, gateway, and DNS server information.
* Check if the IP address has been assigned successfully by running the following command:
  + ip addr show eth0
* The ouput should display the assigned IP address along with other network-related information.
* Using a Static IP Address:
  + If your network requires a static (fixed) IP address, you need to manually configure it. Replace ‘xxx.xxx.xxx.xxx’ with the desired IP address, ‘yyy.yyy.yyy.yyy’ with the subnet mask, and ‘zzz.zzz.zzz.zzz’ with the gateway:
    - Running the following command to set the IP address, subnet mask, and gateway (replace ‘eth0’ with your interface):

ifconfig eth0 xxx.xxx.xxx.xxx netmask yyy.yyy.yyy.yyy

route add default gw zzz.zzz.zzz.zzz eth0

or using the ‘ip’ command:

ip addr add xxx.xxx.xxx.xxx/yyy.yyy.yyy.yyy dev eth0

ip route add default via zzz.zzz.zzz.zzz dev eth0

* Verify the changes by checking the network configuration:

ip addr show eth0

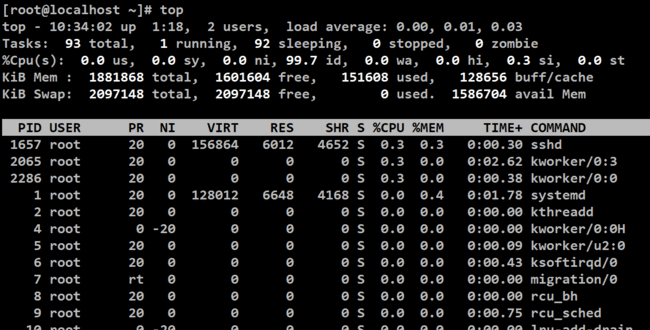
* The output should display the configured static IP address along with other network-related information.
* Verify Connection:
  + ping -c 4 google.com
* Configure DNS:
  + nano /etc/resolv.conf
  + Add lines like:
    - nameserver 8.8.8.8
    - nameserver 8.8.4.4
* Save the file.
* Check Network Status:
  + nmcli device show (or)
  + ip link
* Restart Networking Service (Optional):
  + On some distributions:
    - service networking restart (or)
    - systemctl restart network
      * ‘systemctl’: is a command-line utility that interacts with systemd, which is a system and service manager for Linux.
      * ‘restart’: this is an argument specifying the action to be taken by ‘systemctl’. In this case, it indicates that the specified service or unit should be restarted.
      * ‘network’: this is the name of the service or unit that is being managed by ‘systemctl’. In this context, it refers to the network-related services.
    - Restarting the network services can be useful when making changes to network configurations or resolving issues related to network connectivity.
    - During the restart process, the current state of the network services is terminated, and then they are started again, applying any changes made to the configuration.

**/ vs /root**

* / this represents root (it is Parent Location of ALL)
* /root here root is a FOLDER inside / (parent location)
* whoami > root – here root is the name of Logged in USER.
* Note: All users in Linux are assigned a Home Directory. For user ‘root’ the home directory is /root.

**top**

to check CPU usage of a process



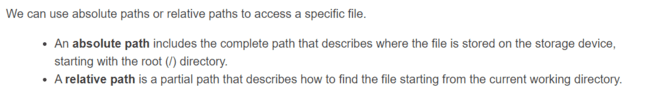
**Absolute Path:**

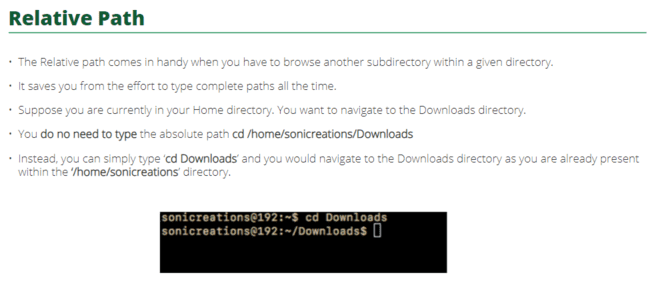
* An absolute path provides the complete path from the root directory (denoted by “/”) and includes all the directories in the hierarchy leading to the target file or directory.
* Example: ‘**/home/user/documents/file.txt**’

****

**Relative Path:**

* A relative path specifies the location of a file or directory relative to the current working directory.
* It doesn’t start from the root directory but rather assumes a starting point (current directory).
* Relative paths are typically shorter and more convenient when navigating with the current directory or its subdirectories.
* Examples:
  + If the current directory is ‘**/home/user/**’, a relative path to ‘**file.txt**’ in the ‘**documents**’ directory would be: ‘**doucuments/file.txt**’.
  + If the current directory is ‘**/var/log/**’, a relative path to ‘**/etc/config.conf**’ would be: ‘**../../etc/config.conf**’.





**Steps for a system to start when clicked on power button:**

1. **Power Supply:**
   * When the power button is pressed, it sends a signal to the power supply unit (PSU) to start providing power to the system.
2. **Motherboard:**
   * Power is then supplied to the motherboard, which is the central circuit board that connects various components.
3. **BIOS/UEFI:**
   * The Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) is activated. This firmware is responsible for initializing and configuring basic hardware components.
4. **POST (Power-On Self-Test):**
   * The BIOS/UEFI performs a Power-On Self-Test (POST) to check the integrity of critical hardware components such as the CPU, RAM, storage devices, and other connected peripherals.
5. **Bootloader:**
   * If the POST is successful, the BIOS/UEFI hands over control to the bootloader. Common Linux bootloaders include GRUB (GRand Unified Bootloader) and LILO (LInux LOader). The bootloader is a small program that loads the operating system kernel into memory.
6. **Kernel Initialization:**
   * Once the bootloader loads the Linux kernel into the memory, the kernel takes control. It initializes hardware, mounts the root filesystem, and sets up essential data structures.
7. **Init Process:**
   * The kernel starts the init process, which is the first user-space process. The init process is responsible for initializing the system and launching other processes. In modern Linux systems, init has been replaced by systems like system.
8. **Operating System:**
   * The operating system is loaded into RAM from the storage device. The OS takes control of the system and begins initializing drivers and services.
9. **User Interface:**
   * Finally, the user interface (e.g., desktop environment) is presented, and the system is ready for user interaction.

* **Note:** All Files are case sensitive in Linux
* Everything in Linux is FILE ONLY (including directory is also a file)

Please Note:

* In Linux File Extensions are of NO use or they don’t matter.
* It is file content which describes the file and not extension.
  + - Powered by jim carry.

Note:

* Whenever we terminate or close any application all application SEND a Numeric Code or Integer to the Operating System.
* That Integer is called EXIT CODE.
* An EXIT CODE 0 means that the application has ended / terminated or closed normally.
* If Exit code is other than 0 then the application ended/terminated or closed abnormally.

In CentOS, the default package manager is yum. However, starting from CentOS 8, yum has been replaced with dnf (short for Dandified YUM) as the default package manager.

**$?**

is a special variable that holds the exit status of the last executed command. The exit status is a numerical value returned by a command to indicate whether it succeeded or encountered an error during execution.

* If the exit status is 0, it usually means that the command executed successfully.
* If the exit status is non-zero, it typically indicates an error or failure during command execution.
* it can be used to control the flow of the script, such as exiting a script or retrying a command if it failed.

**command**

**status=$?**

**echo "The exit status of the command is $status"**

* In this example, after executing the command, the exit status is captured in the variable status, and then it's printed to the terminal.
* Here's a more practical example:

**ls non-existent-directory**

**status=$?**

**if [ $status -eq 0 ]; then**

**echo "Directory exists."**

**else**

**echo "Directory does not exist or there was an error."**

**fi**

* In this example, the ls command attempts to list the contents of a directory that doesn't exist.
* The exit status is checked using $?, and based on that, a message is printed to the terminal.
* This mechanism is commonly used in shell scripts to check whether a command was successful and take appropriate actions based on the exit status.
* It provides a way for scripts to handle errors and make decisions based on the success or failure of commands in the script.

**Variables**

* In Linux, variables used in shell scripts are stored in the memory of the running shell process.
* When a script is executed, it creates a new shell process, and any variables defined within that script are stored in the memory allocated to that process.
* When a script finishes execution, the variables defined within that script are no longer accessible, and the memory allocated to the process is released.
* If you want to persistently store variables or make them accessible to other processes, you can use environment variables or store them in configuration files.
* Variables in shell scripting are placeholders for data, and when referenced with the $ symbol, they are replaced with their actual values during execution.



Two types:

1. **System Variables**
2. **User defined Variables**

variables are used to store and manage data within shell scripts or interactive shell sessions. They provide a way to store information, such as strings or numbers, for later use in commands or scripts. Here are some key points about variables in Linux:

* **Variable Naming Rules:**
* Variable names can consist of letters, numbers, and underscores.
* The first character of a variable name must be a letter or an underscore.
* Variable names are case-sensitive.
* By convention, variable names are often written in uppercase letters.
* **Variable Assignment:**
* Variables are assigned values using the = operator with no spaces around it.
* No spaces are allowed around the equal sign in variable assignments.
* **variable\_name="Hello, World!"**
* Accessing Variable Values:
* To access the value of a variable, prefix the variable name with the $ symbol.
* **echo $variable\_name**
* This would output:
* Hello, World!
* **Quotes in Variable Assignment:**
* Single quotes and double quotes have different effects on variable values.
* Single-quoted strings preserve the literal values of each character.
* In the case of single\_quoted\_var, the entire string within single quotes is treated literally.
* The $variable\_name inside the single quotes is not interpreted as the variable; it is treated as part of the string.
* Double-quoted strings allow variable substitution.
* **single\_quoted\_var='This is $variable\_name'**
* **double\_quoted\_var="This is $variable\_name"**
* In this case, echo $single\_quoted\_var would output:
* **This is $variable\_name**
* And echo $double\_quoted\_var would output:
* **This is Hello, World!**
* **Read-Only Variables:**
* You can make a variable read-only using the readonly keyword.
* A read-only variable cannot be reassigned.
* readonly readonly\_var="This variable is read-only"
* **Unsetting Variables:**
* The unset command is used to unset or delete a variable.
* unset variable\_name
* **Special Variables:**
* There are several special variables in Linux, such as $HOME, $USER, $PATH, etc., which store information about the system and the environment.
* echo $HOME
* This would output the home directory of the current user.
* **Important System variables in Linux:**
* **$HOME:** Home directory of the current user.
* **$USER and $LOGNAME:** Username of the current user.
* **$PATH:** Colon-separated list of directories where executable files are located.
  + $PATH Variable : this variable contains system path which are scanned to look for commands executed by us.
  + echo $PATH
  + /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
* **$PWD:** Current working directory.
* **$SHELL**: Default shell for the user.
* **$TERM:** Terminal type.
* **$LANG and $LC\_\*:** Locale and language settings.
* **$PS1:** Primary prompt string for the shell.
  + - this variable represents your prompt
  + If you want to customize your command prompt using the PS1 variable, you can do so by setting it in your shell configuration file. For example, in Bash, you can modify your ~/.bashrc file:
  + **export PS1='\u@\h:\w\$ '**
  + This sets the prompt to display the username (\u), the hostname (\h), and the current working directory (\w). The $ at the end represents the command prompt.
  + After modifying the ~/.bashrc file, you can either start a new terminal session, or you can run source ~/.bashrc to apply the changes immediately.
  + To unset PS1 do below option:
  + If you set PS1 in your ~/.bashrc file, open the file using a text editor and either comment out the line or remove it. For example:
  + **# export PS1='\u@\h:\w\$ '**
* **$PS2:** Secondary prompt string for the shell.
* **$OLDPWD:** Previous working directory.
* **$IFS (Internal Field Separator):** Delimiter used by the shell for word splitting.
* **$RANDOM:** Random integer between 0 and 32767.
* **$UID and $EUID:** User ID and effective user ID of the current user.
* **$BASH\_VERSION**: Version number of the Bash shell.

Let us create some user defined variables

planet=”earth”

echo we live on $planet

we live on earth

**File Globbing**

also known as wildcard expansion, is a feature in Linux shells that allows you to specify a pattern to match filenames.

The shell then expands the pattern into a list of matching filenames before executing a command.

Here are some common file globbing characters:

* **(Asterisk):**
* Matches any sequence of characters (including none).
* For example, \*.txt matches all files ending with ".txt".
* ls \*.txt
* **? (Question Mark):**
  + Matches any single character.
  + For example, file?.txt matches files like "file1.txt", "fileA.txt", etc.
  + ls file?.txt
* **[ ] (Square Brackets):**
  + Matches any single character within the specified range or set.
  + For example, [aeiou] matches any vowel, and [0-9] matches any digit.
  + ls file[0-9].txt
* **[! ] (Exclamation Mark within Square Brackets):**
  + Matches any single character NOT in the specified range or set.
  + For example, [!aeiou] matches any consonant.
  + ls file[!aeiou].txt
* **{ } (Curly Braces):**
  + Allows the generation of alternative patterns.
  + For example, {jpg,png,gif} matches files ending with ".jpg", ".png", or ".gif".
  + ls \*.{jpg,png,gif}
* **\*\* (Double Asterisk):**
  + Matches directories and subdirectories recursively.
  + Used with tools like find.
  + find . -name "\*.txt"

**NGINX Installation:**

**NGINX commands:**

#Install NGINX

yum install epel-release

yum install nginx

#Update the host system and install NGINX

yum update

#Check the version installed:

nginx -v

#Access the server from a Windows Browser:

ip addr show

#Note the ip address under 'inet' entry

#open browser and enter ip address

# If the output doesn't show

# Check the incoming network is allowed using below commands

systemctl status nginx

systemctl start nginx

#Update the firewall using below commands to allow http as well

firewall-emd --state

sudo firewall-cmd --add-service=http --permanent

firewall-cmd --reload

**ClamAv**

#update your system

yum update

#install ClamAV

yum install clamav clamav-update

#configure Freshclam (ClamAV Updater)

sed -i 's/^Example/#Example/' /etc/freshclaam.conf

freshclam

**LMD**

#installi REquired Dependencies

yum install epel-release

yum install perl wget net-tools tar unzip

#Download and Install LMD

cd /usr/src

wget https://www.rfxn.com/downloads/maldetect-current.tar.gz

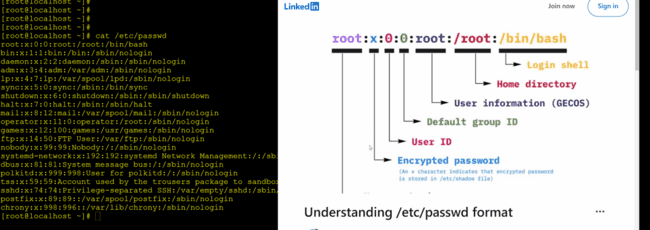
tar -zxvf maldetect-current.tar.gz

cd maldetect-\*

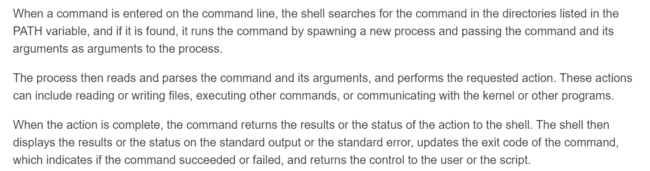
bash install.sh

freshclam

maldet --scan-all /path



**Commands**

****

**echo**

* In Linux, ‘echo’ is a command used to print text or variables to the terminal.
* echo is a binary file.

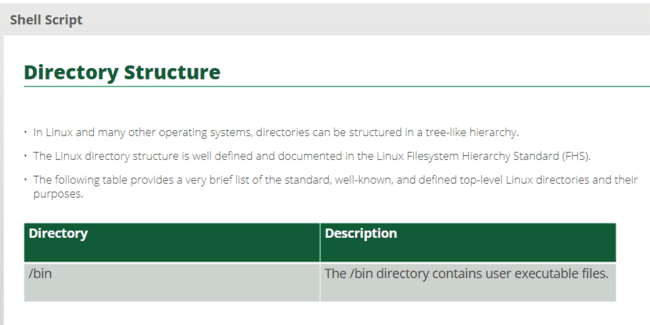
**shell**

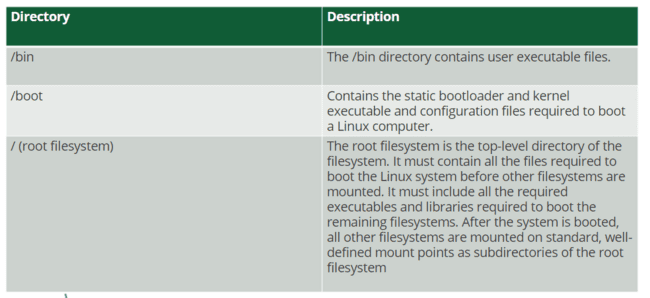
To find out the total number of available shells on your system, you can check the contents of the "/etc/shells" file.

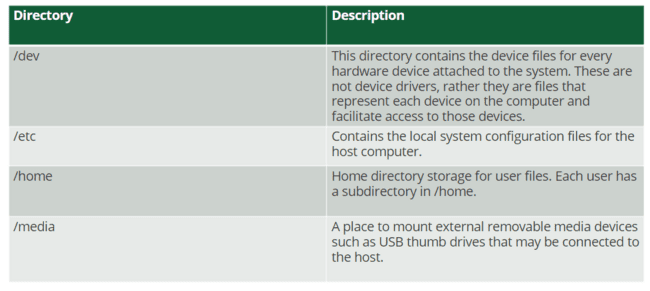
cat /etc/shells

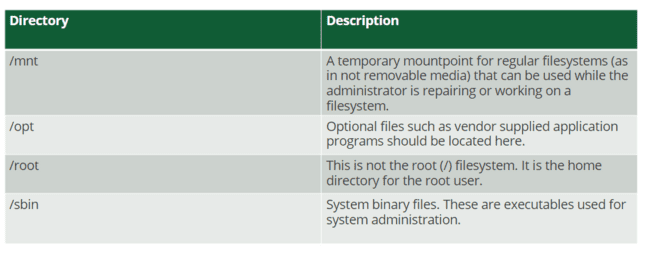
This will display a list of shells available on your system. Each line in the output represents a valid shell.

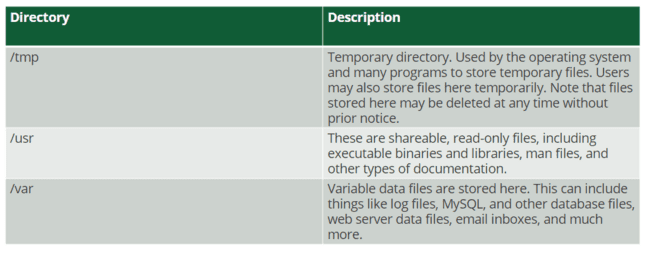
**File structure**

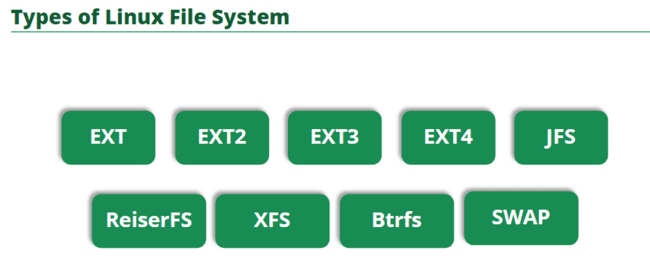


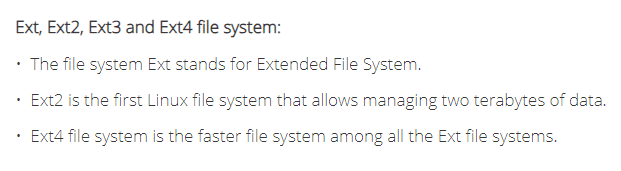


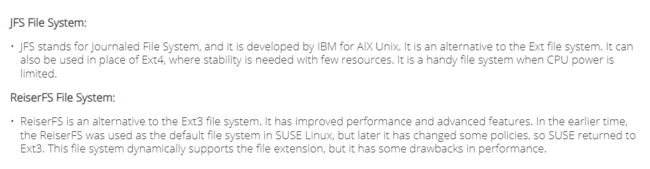


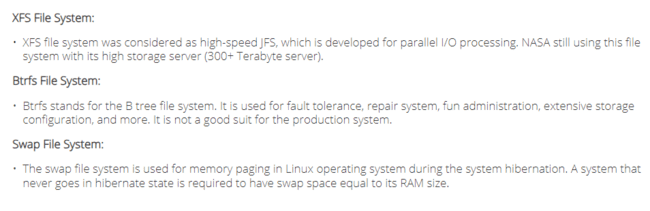


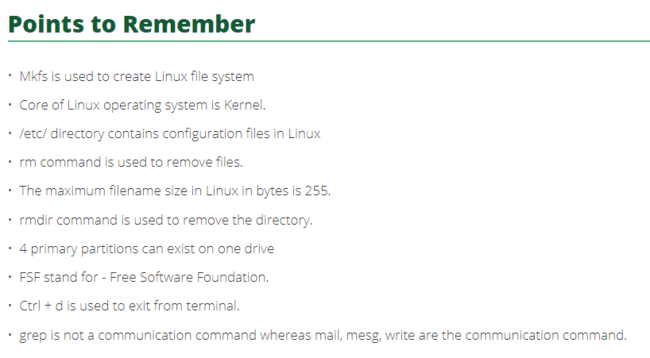


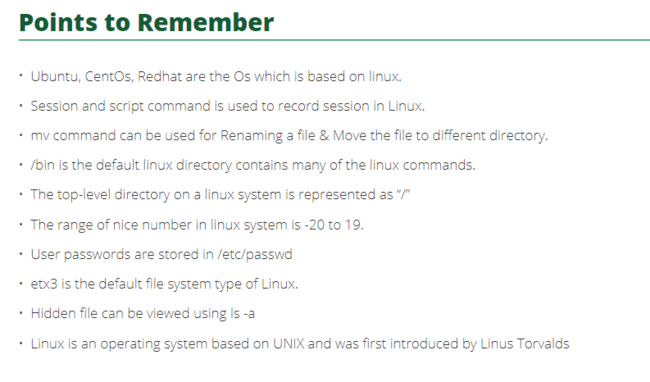


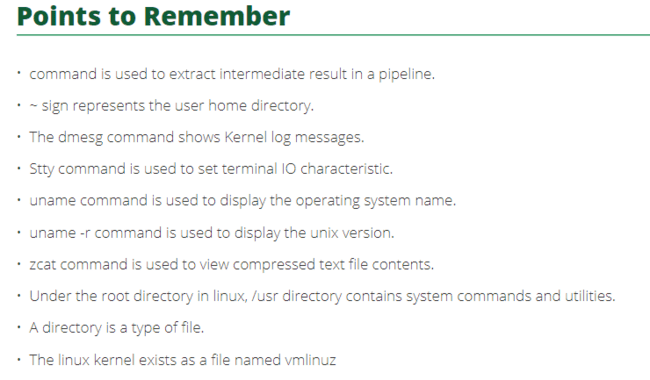












Mounting vs unmounting filesystem

**$SHELL**

* $SHELL is an environment variable in Linux that stores the path to the default shell for the user.
* This variable is usually set in the user’s configuration files during login.
* Ex: $SHELL -c ‘echo Hello’
* -c option is used to pass a command to the shell.
* So the above command starts a new shell and runs the ‘echo Hello’ command within that shell and the output will be Hello.

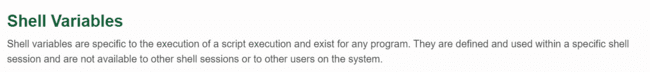
Reasons for adopting bash shell as the default shell for Linux

* Because of its additional functionalities over original shell (sh)
* Command aliasing (ex: alias c=clear)
* Command completion
* File completion by using tab keys
* Command history
* Easily you can execute previous commands using up and down arrow keys without typing entire command.

**scope**



****

****

**echo $SHELL**

* When you run ‘**echo $SHELL**’ in a Linux terminal, echo prints the value stored in the ‘$SHELL’ variable, which is the path to the default shell for the current user.
* Output: of the command will be the path to the user’s default shell, such as ‘**/bin/bash**’ for the Bash shell or ‘**/bin/zsh**’ for the Zsh shell.
* Default shell information of the user can be useful for configuration purposes or when troubleshooting issues related to the user’s shell environment.

**ip a**

* ‘ip’ is a versatile command-line utility for configuring and displaying network interfaces on Linux systems. It is part of the iproute2 package.
* ‘a’ option is short for ‘address’ indicating that we want to display information about network addresses, which includes IP addresses.
* ‘ip a’ without any additional parameters typically displays information about all active network interfaces on the system, including details such as the interface name (‘eth0’, ‘lo’, etc.), IP addresses assigned to each interface, MAC addresses, and other configuration details.
  + ‘ip a show dev eth0’: show details only for the ‘eth0’ interface.
  + ‘ip a show up’: show details only for interfaces that are currently up.

**IP Address (Internet Protocol Address) (Logical Address):**

* An IP address is a unique numerical label assigned to each device connected to a computer network that uses the Internet Protocol for communication.
* It serves as an identifier for a device on a network and enables communication between devices over the Internet or local networks.

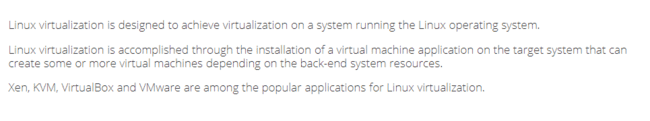
**MAC Address (Media Access Control Address) (Physical Address):**

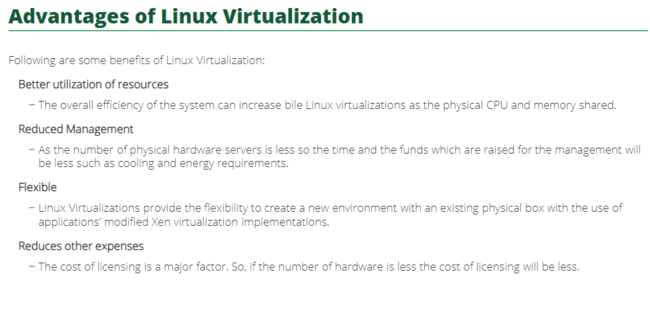
* A MAC address is a globally unique identifier assigned to a network interface controller (NIC) by the manufacturer.
* It serves as a permanent hardware identifier for a device and is used for communication on the physical network segment, allowing devices to identify and communicate with each other within the same local network.

**vi /etc/sysconfig/network-scripts/ifcfg-enp0s3**

* **‘vi’:**
  + Is a text editor commonly available on Unix-like operating systems, including Linux. It stands for “visual editor”
* **‘/etc/sysconfig/network-scripts/ifcfg-enp0s3’:**
  + This is the path to the configuration file for the network interface name ‘enp0s3’.
  + The file is located in the ‘/etc/sysconfig/network-scripts/’ directory, which is a common location for network configuration files on Linux distributions that use the Red Hat or CentOS network configuration conventions.
* ‘vi’ command is being used to open the specified configuration file (‘ifcfg-enp0s3’) in the ‘/etc/sysconfig/network-scripts/’ directory for editing.
* If you were to edit network configuration parameters, you might see lines like:
  + TYPE=Ethernet
  + BOOTPROTO=dhcp
  + NAME=enp0s3
  + DEVICE=enp0s3
  + ONBOOT=yes
* These lines define configuration parameters for the ‘enp0s3’ network interface, indicating that it uses DHCP to obtain an IP address, is enabled (‘ONBOOT=yes’), etc.
* So when we do the above step by default enp0s3 ip address is public address so we need to change to our IP address for security so that no one can access.
* To do this go to the virtual machine (VM) in virtual box and select settings option to particular VM you would like to change the ip address.
* NAT vs Bridged Adapter
  + **NAT** (Network Address Translation) : is a networking technique that allows multiple devices within a local network to share a single public IP address for internet access.
  + The NAT-enabled router assigns private IP addresses to devices on the local network. Outgoing traffic from devices is translated to the router’s public IP address before reaching the internet.
  + Incoming responses are translated back to the corresponding private IP addresses.
  + **Bridged Adapter:** allows a virtual machine to appear as a separate device on the physical network, obtaining its own IP address.
  + The virtual machine’s network interface is bridged with a physical network adapter on the host machine.
  + The virtual machine behaves like any other device on the physical network, with its own unique IP address.
* In settings > Network > by default NAT will be there select > Bridged Adapter in dropdown then name will be selected to the wifi connected to the laptop.
* Now go to VM and type ‘systemctl restart network’ to restart network service.
* Now check updated ip address using ‘ip a’ command.

**Virtualization**





**pwd :** print working directory

* It prints the absolute path of the current working directory of the file system in to the terminal.
* Ex: pwd
* /home/yourusername/documents

**mkdir :** to create new directories (folders).

* mkdir [options] directory\_name
  + ‘options’: Optional flags that modify the behaviour of the command.
  + ‘directory\_name’: The name of the directory you want to create.
* Examples:
  + Create a Directory:
    - mkdir my\_directory
    - This command creates a directory name “my\_directory” in the current working directory.
  + Create Nested Directories:
    - mkdir -p my\_parent\_directory/my\_child\_directory
    - The ‘-p’ option allows you to create nested directories.
    - In this example, it creates both “my\_parent\_directory” and “my\_child\_directory” if they don’t exist.
  + Create Multiple directores:
    - mkdir dir1 dir2 dir3

**ls**

list the files and directories in a directory.

ls [options] [directory]

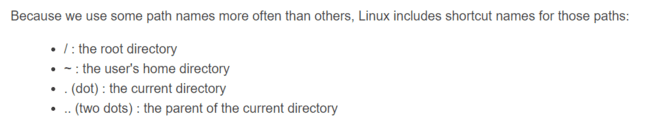
* ‘options’ : Optional flags that modify the behavior of the command.
* ‘directory’: The path to the directory you want to list. If not specified, it defaults to the current working directory.
* List files and Directories in the Current Directory:
  + **ls**
  + this command lists the files and directories in the current working directory.
* List Files with Detailed Information:
  + **ls -l**
  + The ‘-l’ option provides a detailed listing, showing additional information such as permissions, owner, group, file size, modification time, and more.
  + ls -l your\_file\_or\_directory
  + -rw-r--r-- 1 user user 1234 Feb 3 10:00 your\_file\_or\_directory
* List All Files (including Hidden Files):
  + **ls -a**
  + The ‘-a’ option includes hidden files (those starting with a dot) in the listing.
* List Files with Human-Readable File Sizes:
  + **ls -h**
  + The ‘-h’ option makes the file sizes more human-readable, using units like KB, MB, GB, etc.
* List Files with Detailed Information and Human-Readable File Sizes:
  + **ls -lh**
  + This combines the ‘-l’ and ‘-h’ options to provide a detailed lisitng with human-readable file size.
  + If you want to check the permissions in a more human-readable format, you can use the stat command:
  + stat your\_file\_or\_directory
* List Files Recursively (include Subdirectories):
  + **ls -R**
  + The ‘-R’ option lists files and directories recursively, including those in subdirectories.
* To list contents in reverse order:
  + **ls -r**
  + The ‘-r’ option lists files and directories in reverse order.
* To list the entries in comma separated format:
  + **ls -m**
  + The ‘-m’ option is used to fill width with a comma separated list of entries.
* To list contents based on time:
  + **ls -t**
  + The ‘-t’ option lists files and directories by modification time, newest first
* To list entries in lines:
  + **ls -x**
  + The ‘-x’ option is used to list files and directories by lines instead of by columns
* **ls -1**
  + The ‘-1’ option is used to list one file per line.
  + The ls command is a standard command in Linux and is typically found in the **/bin directory.**
  + The /bin directory contains essential binary files that are required for the system to function properly, and these binaries are often part of the system's basic functionality.
  + You can find the ls command by checking the /bin directory. You can use the which command to determine the location of the ls binary:
  + **which ls**
  + This command will output the path to the ls command, which is likely to be something like /bin/ls.
  + Alternatively, you can use the type command:
  + **type ls**
  + This will provide information about the type of command ls is, and it will show you the path to the executable.

**rmdir:** used to remove empty directories.

rmdir [options] directory

* ‘options’: Optional flags that modify the behavior of the command.
* ‘directory’: The name of the directory you want to remove.
* **rmdir my\_empty\_directory**
  + This command will remove the directory named “my\_empty\_directory” if it is empty.
  + If the directory contains any files or subdirectories, the ‘rmdir’ command will not work, and you would need to use the ‘rm’ command with the ‘-r’ option to remove directories and their contents recursively.
  + rm -r my\_non\_empty\_directory
* **rmdir dir1 dir2 dir3**
  + if you want to remove multiple empty directories, you can provide multiple directory names as arguments to the ‘rmdir’ command.

**cd**

****

change the current working directory

Used for navigating file system

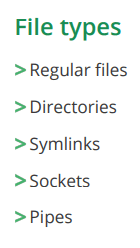
* **cd [directory\_path]**
  + ‘directory\_path’: The path to the directory you want to change to. If not specified, it defaults to the user’s home directory.
* Change to a Specific Directory:
  + **cd /path/to/directory**
  + Replace ‘/path/to/directory’ with the actual path of the directory you want to navigate to.
* Move Up One Directory Level:
  + **cd ..**
* Move Up Two Directory Levels:
  + **cd ../..**
  + Use ‘..’ to move up one level, and you can concatenate it to move up multiple levels.
* Navigate to a Subdirectory Relative to the Current Directory:
  + **cd subdirectory**
* Use Tab Completion:
  + You can use tab completion to quickly navigate through directories. For example, if you type **‘cd /u’** and press Tab, it will autocomplete to the **‘/usr’** directory (assuming it exists).
* Use Environment Variables:
  + cd $HOME
  + You can use environment variables, such as **‘$HOME’**, to quickly navigate to specific directories.

**rm:** remove or delete files and directories.

rm [options] file1 file2 …

* Remove a file:
  + **rm filename**
  + This command removes a single file named “filename”.
* Remove Multiple Files:
  + **rm file1 file2 file3**
  + You can specify multiple files separated by space to remove them.
* Remove a Directory and Its Contents Recursively:
  + **rm -r directory\_name**
  + The ‘-r’ option is used to remove directories and their contents recursively.
* Forcefully Remove Files without Confirmation:
  + **rm -f file1 file2**
  + The ‘-f’ option suppresses confirmation prompts and forcefully removes the specified files.
* Interactive Mode (Prompt Before Removal):
  + **rm -i file1 file2**
* Remove directories and Their Contents Quietly:
  + **rm -rf directory\_name**
  + the combination of ‘-r’ and ‘-f’ options removes directories and their contents without prompting or displaying error messages.

**file type**

****

**whoami**

* displays the username of the current user.

**who am i**

* This command tells us information about our current session.

**who**

command in Linux is used to display information about users who are currently logged into the system.

When you run the who command without any options, it provides a list of logged-in users along with details such as username, terminal, login time, and originating IP address (if applicable).

* Here's an example:
  + who
  + The output may look something like this:
  + username1 tty1 2022-01-01 10:00
  + username2 pts/0 2022-01-01 11:30 (192.168.1.2)
* In this example:
  + **username1** is logged in on the local terminal (tty1) since 10:00 AM.
  + **username2** is logged in on a pseudo-terminal (pts/0) since 11:30 AM, and the IP address 192.168.1.2 indicates a remote login.
* The **who** command has various options to display additional information or filter the output. For instance:
  + **who -u**: Displays additional information, including the idle time and process ID.
  + **who -q**: Shows only the number of logged-in users.

**id**

this command gives u ur current userid

It provides details such as the user's UID (user ID), GID (group ID), supplementary group IDs, and the associated username and groups.

* Here's the basic syntax:
* **id [username]**
  + If you run id without specifying a username, it will display information about the current user:
* **id**
  + If you provide a specific username, it will show information for that user:
* **id username**
  + Example output for the current user:
  + **uid=1000(username) gid=1000(username) groups=1000(username),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare)**
  + In this example:
  + uid is the user ID.
  + gid is the primary group ID.
  + groups list the supplementary group IDs to which the user belongs.
  + If you specify a username, you'll get similar information for that user.

The id command is useful for quickly checking the identity and group memberships of a user on a Linux system.

**inode**

ls -li

inode is an “index node”. It serves as a unique identifier for a specific piece of metadata on a given filesystem.

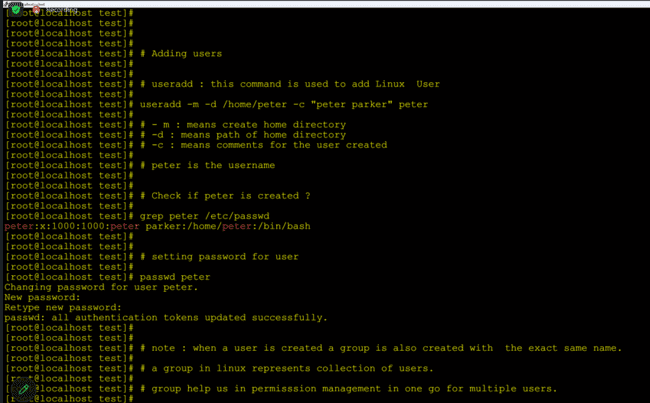
**useradd**

* **sudo useradd new\_username**
  + replace ‘new\_username’ with the desired username for the new user.
* **sudo passwd new\_username**
  + you will be prompted to enter and confirm the password.
* **sudo usermod -aG sudo new\_username**
  + To grant administrative privileges to the new user, you can add them to the **‘sudo’** group using the above command.
  + This allows the user to execute commands with **‘sudo’.**
  + After creating the new user, you can switch to that user using the **‘su’** command or log in directly with the new username and password.

Useradd command in Linux is used to create a new user account. When you create a new user using useradd, it adds the necessary entries to the system files, such as /etc/passwd, /etc/shadow, and others, to define the user's login credentials and attributes.

* Here is the basic syntax of the useradd command:
* sudo useradd [options] username
* Common options include:
  + **-c comment**: Specifies a comment or additional information about the user.
  + **-g group:** Sets the initial login group for the user (default is to create a group with the same name as the user).
  + -**G groups**: Specifies additional groups to which the user should belong.
  + -**m**: Creates the user's home directory if it doesn't exist.
  + **-s shell**: Sets the user's login shell (default is /bin/bash).
  + **-u uid**: Specifies the numeric user ID for the new user.
* Here is an example of creating a new user named "john" with the default settings:
  + sudo useradd john
  + To set a password for the new user, use the passwd command:
  + sudo passwd john
  + This command will prompt you to enter and confirm the password for the new user.
  + If you want to add the user to additional groups, you can use the -G option:
  + sudo useradd -G group1,group2 john
* Remember that, depending on your system, you may need to use sudo to execute these commands with administrative privileges.

After creating a new user, you can use commands like passwd and usermod to further manage the user's account, including changing the password or modifying group memberships.



**usermod**

* usermod is a command-line utility used to modify user account properties.
* It allows administrators to make changes to existing user accounts without having to delete and recreate them.
* The usermod command is particularly useful for tasks such as changing a user's username, primary group, home directory, shell, or user ID (UID).
* **Change Username (-l):**
  + usermod -l newusername oldusername
  + This command changes the username from oldusername to newusername.
* **Change Primary Group (-g):**
  + usermod -g newprimarygroup username
  + This command changes the user's primary group to newprimarygroup.
* **Change Home Directory (-d):**
  + usermod -d /new/home/directory username
  + This command changes the user's home directory to /new/home/directory.
* **Change Shell (-s):**
  + usermod -s /path/to/shell username
  + This command changes the user's login shell to /path/to/shell.
* **Change User ID (-u):**
  + usermod -u newuid username
  + This command changes the user's UID to newuid.
* **Add Supplementary Groups (-aG):**
  + usermod -aG groupname username
  + This command adds groupname to the list of supplementary groups for the user.
* **Lock or Unlock User Account (-L/-U):**
  + usermod -L username
  + This command locks the user account, preventing login.
  + usermod -U username
  + This command unlocks the user account, allowing login.

**~**

* The tilde (‘~’) is a shorthand notation used to represent the home directory of the currently logged-in user.
* cd ~john : this changes the current working directory to the home directory of the user “john”.

**touch**

used to create empty files

update the access and modification timestamps of existing files.

* Create a New Empty File:
  + **touch filename.txt**
  + This command creates an empty file named “filename.txt” in the current working directory. If the file already exists, it updates its timestamps.
* Update Timestamps of Existing Files:
  + **touch existingfile.txt**
  + This command updates the timestamps of the file “existingfile.txt”
* Create Multiple Files:
  + **touch file1.txt file2.txt file3.txt**
* Create Files in a Specific Directory:
  + **touch /path/to/directory/newfile.txt**
* Use with Wildcards:
  + **touch prefix\_{1..5}.txt**
  + You can use wildcards to create multiple files with a similar pattern.
  + This command creates files with names like “prefix\_1.txt”, “prefix\_2.txt” and so on.
* Using the ‘-t’ option:
  + You can use the ‘-t’ option with the **‘touch’** command to specify a specific timestamp in the format ‘**[[CC]YY]MMDDhhmm[.ss]’.**
  + **touch -t 202201011200.00 filename.txt**
  + this sets the timestamps of “filename.txt” to January 1, 2022, 12:00PM.
* Using a reference file:
  + **touch -r reference\_file.txt filename.txt**
  + You can use the **‘-r’** option to set the timestamps of a file based on another file’s timestamps.
  + This sets the timestamps of “filename.txt” to be the same as those of “reference\_file.txt”.

**/boot**

Is a specific directory that contains essential files for the initial stages of the boot process.

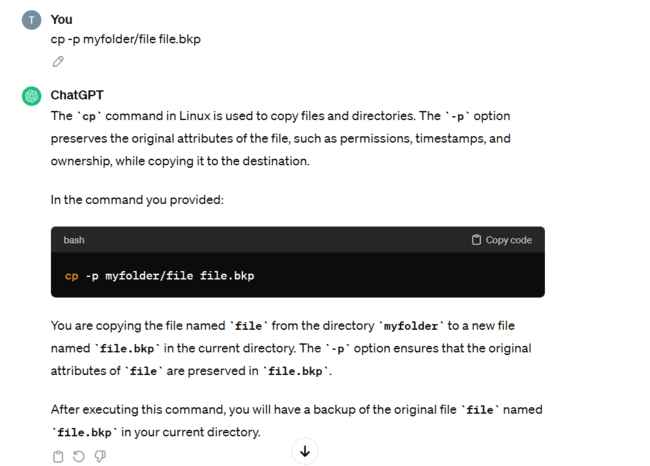
* Bootloader Configuration Files
* Kernel Images
* Initial RAM Disk (initramfs) : (initial RAM filesystem) is an initial root filesystem that is loaded into memory during the early stages of the boot process.

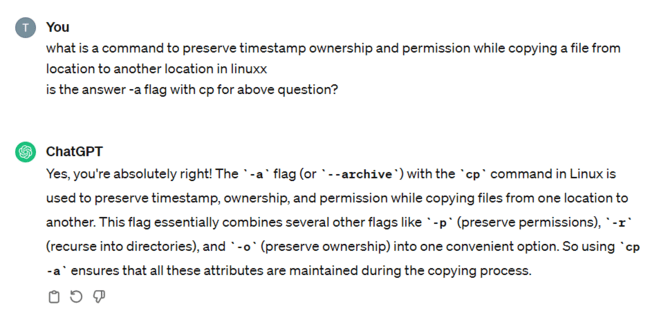
**cp**

copy files or directories from one location to another.

**cp [options] source destination**

* Copy a File:
  + **cp file.txt /path/to/destination/**
* Copy Multiple Files:
  + **cp file1.txt file2.txt /path/to/destination/**
* Copy a Directory and Its Contents:
  + **cp -r /path/to/source\_directory/ /path/to/destination/**
  + The ‘-r’ option is used to copy directories recursively.
* Preserve File Attributes (Permissions, Ownership, Timestamps):
  + **cp -a /path/to/source/ /path/to/destination/**
  + The ‘-a’ option preserves the file attributes, including permissions, ownership, and timestamps
* Interactive Copy (Prompt Before Overwriting):
  + **cp -i file.txt /path/to/destination/**
  + The ‘-i’ option prompts for confirmation before overwriting an existing file.
* Verbose Mode (Display Detailed Information):
  + **cp -v file.txt /path/to/destination/**
  + The ‘-v’ option enables verbose mode, displaying each file as it is copied.
* Force Overwrite (Without Prompting):
  + **cp -f file.txt /path/to/destination/**
  + The ‘-f’ option forces the copy without prompting, potentially overwriting existing files.





**mv**

Move or rename files and directories

**mv [options] source destination**

* **Options:** Optional. There are several options you can use with ‘**mv**’.
  + **‘-i**’ : Prompt before overwriting files.
  + **‘-u**’ : Move only when the source file is newer than the destination file or when the destination file is missing.
  + **‘-b**’ : Create a backup of each existing destination file.
  + **‘-v’** : Be verbose, print the names of the files as they are moved.
* Move a file to another directory:
  + **mv file.txt /path/to/destination/**
* Rename a file:
  + **mv oldfile.txt newfile.txt**
* Move a directory and its contents:
  + **mv directory1 /path/to/destination/**
* Rename a directory:
  + **mv old\_directory new\_directory**
* Move and prompt before overwriting:
  + **mv -i file.txt /path/to/destination**

**man**

to display the manual pages for other commands.

The manual pages provide detailed information about various commands, including their usage, options and examples.

**man [command]**

Examples

man ls

man mv

**whatis**

is used to display a brief one-line description of a specified command.

It provides a concise summary of the command’s purpose.

* **whatis [command]**
* Example:
* whatis ls

**whereis**

is used to locate the binary, source code, and manual page files for a specified command.

It provides information about the locations of various components associated with a command.

* **whereis [option] command**
* Options:
  + ‘-b’ : Search for the binary executable.
  + ‘-s’ : Search for the source code.
  + ‘-m’ : Search for the manual page.
  + By default, ‘whereis’ searches for all components.
* Examples:
* Search for the binary executable of a command:
  + **whereis ls**
  + This command will display the path to the binary executable of the ‘ls’ command.
* Search for the source code of a command:
  + **whereis -s ls**
  + This command will display the path to the source code files related to the ‘ls’ command.
* Search for the manual page of a command:
  + **whereis -m ls**
  + This command will display the path to the manual page associated with the ‘ls’ command.
* Search for all components of a command:
  + **whereis grep**
  + This command will show the locations of the binary executable, source code, and manual page for the ‘grep’ command.

**file**

to determine the type of a file.

Examines the file and provides information about its type, such as whether it’s a text file, binary file, image file, etc.

The ‘file’ command uses a combination of heuristics and magic number matching to identify the file type.

**file [options] filename**

* Example : file example.txt
* ‘-i’ : outputs MIME type along with the file type.
  + The MIME type (Multipurpose Internet Mail Extensions type) is a standardized way of indicating the nature and format of a document, file, or data.
* ‘-b’: provides a brief output, useful for scripting.
* ‘-z’: Causes the output to be in a machine-readable format.

**Note:** ‘file’ command depends on the accuracy of its magic number database, which is a collection of patterns used to identify file types. If a file has an unusual or non-standard format, ‘file’ might not be able to accurately determine its type.

**SCP**

If you have SSH access to the Linux guest, you can use the ‘scp’ command to securely copy files over the network.

1. Open a command prompt on the windows host.
2. Use the **‘scp’** command to copy the file to the Linux guest. Replace **‘<username>’**, **‘<hostname>’, ‘<path\_on\_guest’**, and **‘<local\_file>’** with appropriate values.

**scp <local\_file> <username>@<hostname>:<path\_on\_guest>**

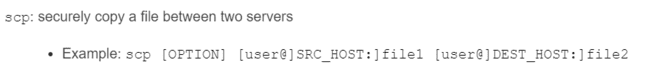
Example:

scp myfile.txt [user@192.168.1.100:/home/user/documents/](mailto:user@192.168.1.100:/home/user/documents/)

1. Enter the password when prompted.



Note: This method requires SSH to be installed and configured on the Linux guest, and the Windows host should have an SCP client (like WinSCP) or an SSH client with SCP support (like PuTTY).



**head**

to display the beginning of a file or the first few lines of a file.

It is particularly useful when you want to quickly view the contents of a file without displaying the entire content.

**head [options] [filename]**

‘n N’ or ‘--lines=N’: Display the first N lines of the file.

‘-c N’ or ‘--bytes=N’: Display the first Nbytes of the file.

‘-q’ or ‘--quiet’: --silent’: Never print headers, even when processing multiple files.

‘-v’ or ‘—verbose’: Always print headers, even when processing multiple files.

Examples:

* Display the first 10 lines of a file:
  + head filename (or)
  + head -n 10 filename
* Display the first 20bytes of a file:
  + head -c 20 filename
* Display the first 5 lines of multiple files, with headers:
  + head -n 5 -v file.txt file2.txt
* Display the first 15 lines of a file without headers:
  + head -n 15 -q filename

**tail**

to display the end of a file or the last few lines of a file.

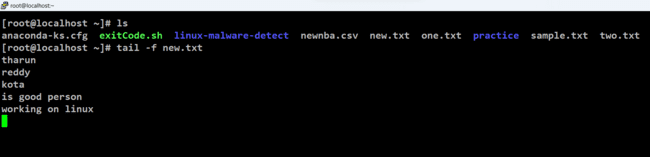
It is useful for checking the recent additions or changes to a log file or any other text file.

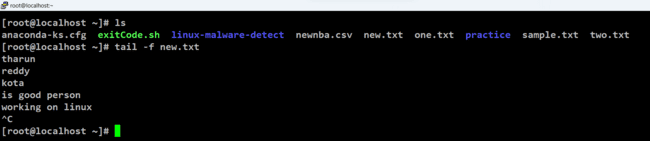
**tail [options] [filename]**

‘-n N’ or ‘--lines=N’: Display the last N lines of the file.

‘-c N’ or ‘--bytes=N’: Display the last N bytes of the file.

‘-f’ or ‘--follow’: Output appended data as the file grows (useful for watching log files in real-time).





‘-q’ or ‘--quiet’, ‘--silent’: Never print headers, even when processing multiple files.

‘-v’ or ‘--verbose’: Always print headers, even when processing multiple files.

Examples:

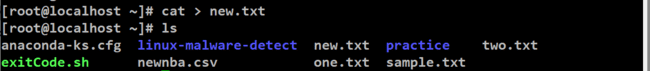
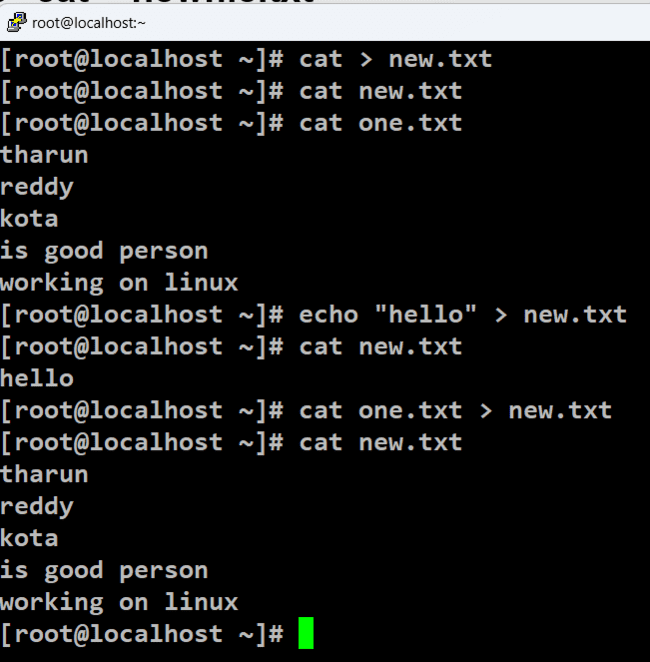
* Display the last 10 lines of a file
  + tail filename (or)
  + tail -n 10 filename
* Display the last 20 bytes of a file:
  + tail -c 20 filename
* Display the last 5 lines of multiple files, with headers:
  + tail -n 5 -v file1.txt file2.txt
* Display the last 15 lines of a file without headers:
  + tail -n 15 -q filename
* Follow and output appended data to a file (useful for watching log files in real-time):
  + tail -f filename
  + this enables the “follow” mode.
  + ‘tail’ monitors the specified file for new lines that are added to it in real-time, and it continuously displays the appended data as the file grows.
  + This is particularly useful when you want to observe changes in log files or any other files that are actively being written to, such as system logs or application logs.
  + When using the ‘-f’ option, ‘tail’ does not exit after reaching the end of the file but rather continues to run, updating its output whenever new lines are added to the file.
  + In this mode, ‘tail’ will display the last few lines of the file, and then it will wait for new lines to be appended to the file. As new lines are added, they will be displayed in real-time. To stop ‘tail’ when in follow mode, you can usually use the keyboard shortcut ‘Ctrl+C’ to interrupt the command.
  + This feature is especially helpful for monitoring log files and diagnosing issues as they occur. It allows you to keep track of the ongoing activity in a file without having to repeatedly run the ‘tail’ command manually.

**cat**

to combine two files.

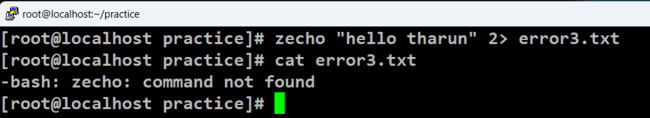
to concatenate and display the content of files.

**cat [options] [file(s)]**

* Display the Content of a File
  + **cat filename**
  + This command will display the entire content of the specified file (‘filename’) in the terminal.
* Concatenate Multiple Files:
  + **cat file1 file2**
  + This command will concatenate the contents of ‘file1’ and ‘file2’ and display the combined output in the terminal.
  + This will not append anything from file1 to file2.
* Create a New File or Overwrite Existing File:
  + **cat > newfile.txt**
  + This allows you to create a new file named ‘newfile.txt’.
  + You can then type or paste content into the terminal, and when you press ‘Ctrl+D’, the input will be saved to the file.
  + 
  + 
* Append Content to an Existing File:
  + **cat >> existingfile.txt**
  + Similar to the previous example, this allows you to append content to an existing file (‘existingfile.txt’).
  + Press ‘Ctrl+D’ to save the changes.
* Display Line Numbers:
  + **cat -n filename**
  + This option (‘-n’) displays line numbers along with the content of the file.
  + It is helpful when you want to see the line numbers in the output.
* Display All Non-Printable Characters:
  + **cat -v filename**
  + This option (‘-v’) displays all non-printable characters, making it useful for viewing special characters or control characters in the file.
* Display Tabs as ^I:
  + **cat -T filename**
  + this option (‘-T’) displays tabs as ‘^I’ in the output, making it easier to identify tab characters.

How to redirect an error of a command into a file?

To redirect an error we need to use 2>



To redirect both error and output, 2>&1

**more**

used to display the content of a file or the output of a command in a paginated form.

It allows you to view text one screen at a time, making it easier to read and navigate through large files.

**more [options] [file]**

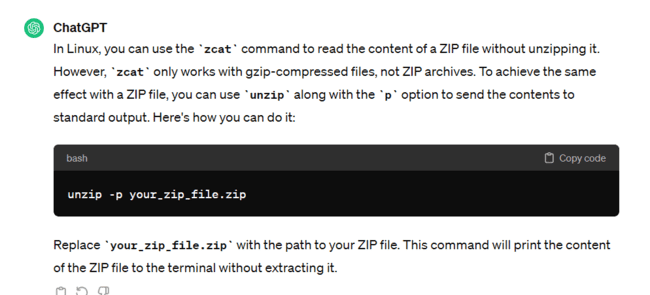
* Display the Contents of a File:
  + **more filename**
  + This command will display the content of the specified file (‘filename’) one screen at a time. Use the Spacebar to advance to the next screen, and press ‘q’ to exit.
* Navigate Through File with Arrow Keys:
  + **more -f filename**
  + this option (‘-f’) enables navigation using arrow keys. You can use the Up and Down arrow keys to scroll through the file.
* Display Line Numbers:
  + **more -n filename**
  + This option (‘-n’) displays line numbers along with the content of the file. It can be useful when you want to reference specific lines.
* Search for Text in the File:
  + **more +/pattern filename**
  + This command will open the file (‘filename’) and display it, starting from the first occurrence of the specified pattern.
* View Compressed Files:
  + zcat filename.gz | more
  + You can use ‘more’ in combination with other commands to view the contents of compressed files.
  + In this example, ‘zcat’ is used to decompress a gzip file.
* Pipe Output of a Command:
  + ls -l | more
  + You can use ‘more’ to paginate the output of a command.
  + In this example, the ‘ls -l’ command lists the contents of a directory, and ‘more’ paginates the output.
* Quit After Displaying One Screen:
  + **more -l filename**
  + this option (‘-l’) makes ‘more’ quit after displaying one screenful of text.
  + It’s useful for quickly checking the beginning of a file.

**Compress**

If you want to transfer a large file (10GB) remotely, what would be the first thig you will do?

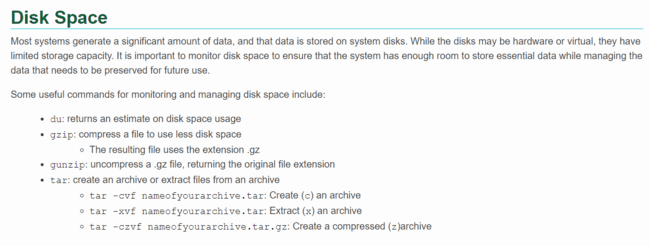
tar->gzip->gunzip

what is the difference between tar, gzip and gunzip?

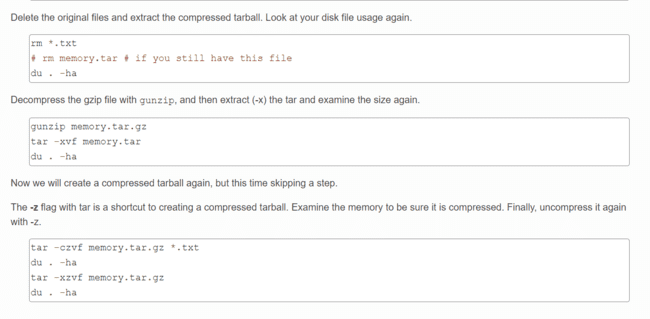


**Space**

**Memory**

****

****

****

**less**

is similar to ‘more’, but it provides more advanced features and greater flexibility.

It allows you to view and navigate through the content of files or command outputs one screenful at a time, just like ‘more’.

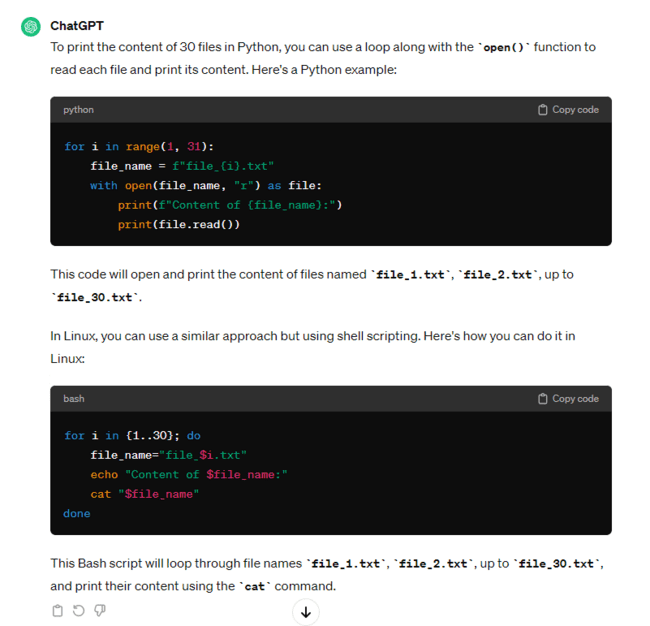
Additionally, ‘less’ supports features like searching, scrolling, and backward navigation.

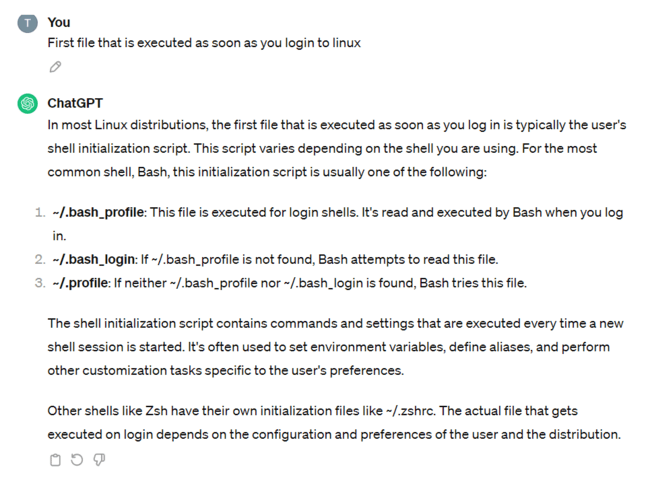
**less [options] [file]**

* Display the Contents of a File:
  + **less filename**
  + This command will display the content of the specified file (‘**filename’**) one screen at a time. You can use the Spacebar to advance to the next screen, and press ‘q’ to exit.
* Navigate Through File with Arrow Keys:
  + **less -N filename**
  + This option (‘-N’) displays the numbers along with the content of the file.
  + You can use the up and down arrow keys to scroll through the file.
* Search for Text in the File:
  + **less +/pattern filename**
  + This command opens the file (‘filename’) and displays it, starting from the first occurrence of the specified pattern. You can also use the ‘/’ key followed by the search term to search interactively within the file.
* View Compressed Files:
  + **less filename.gz**
  + ‘less’ can directly view compressed files, such as gzip-compressed files. It decompresses the content on-the-fly for viewing.
* View Multiple Files:
  + **less file1 file2**
  + You can view the content of multiple files sequentially by providing their names as arguments to ‘less’.
* Display Help:
  + **less --help**
  + This option (‘--help’) provides information about the command-line options and usage of ‘less’.
* Forward and Backward Navigation:
  + Use the ‘Spacebar’ to move forward one screen.
  + Use the ‘b’ key to move backward one screen.
  + Use the ‘G’ key to go to the end of the file.
  + Use the ‘g’ key to go to the beginning of the file.
* Exit Less:
  + Press ‘q’ to exit ‘less’.

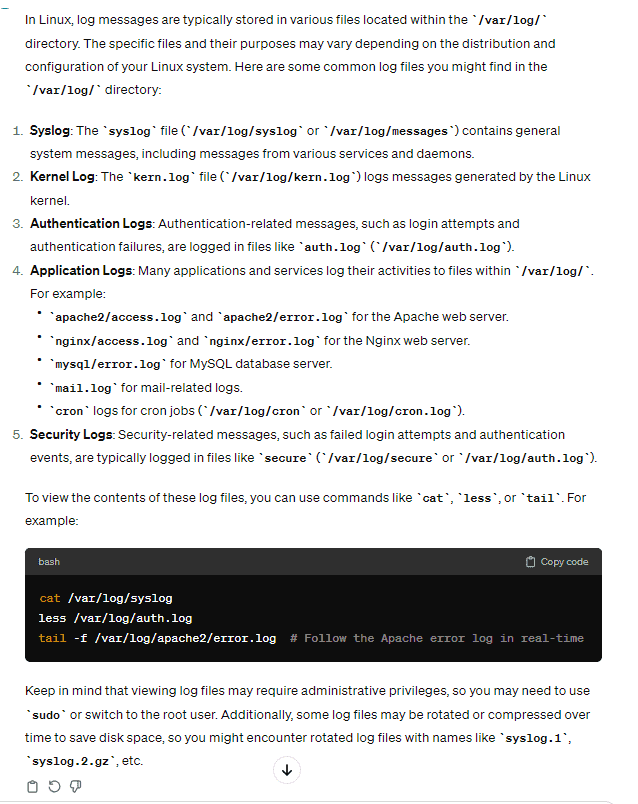
**Note:** ‘less’ offers more interactive features compared to ‘more’, making it a preferred choice for viewing large files or logs.

Remember that ‘less’ is highly customizable, and you can explore additional options by referring to its manual (‘man less’).





**/var/log/**

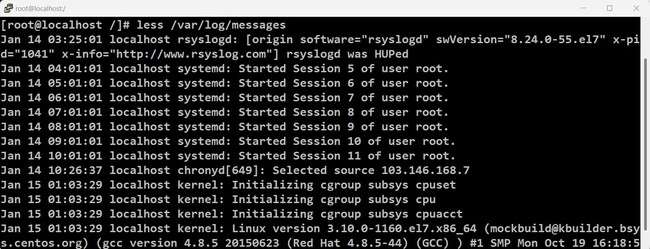
****

**System Log**

provide a record of events and activities on the system, which are crucial for monitoring, troubleshooting, and analyzing the system’s health and performance.

The logs are typically stored in the ‘**/var/log**’ directory.

* **Syslog (‘/var/log/syslog’ or ‘/var/log/messages’):**
  + Contains general system messages, including kernel messages, daemons, and system-related events.



* **Kernel Log (‘/var/log/kern.log’):**
  + Records kernel-specific messages, including hardware and driver-related issues.
* **Authentication Logs (/var/log/auth.log or /var/log/secure):**
  + Logs authentication-related events, such as login attempts, authentication failures, and system access information.
* **Boot Log (/var/log/boot.log):**
  + Captures messages related to the system boot process.
* **Package Manager Logs (/var/log/dpkg.log or /var/log/yum.log):**
  + Records package installation, removal, and upgrade activities.
* **Systemd Journal (/var/log/journal/):**
  + Contains logs managed by the systemd journal, providing a centralized and structured log storage.
* **User Login Information (/var/log/wtmp):**
  + Keeps a record of user logins and logouts.
* **Last Log (/var/log/lastlog):**
  + Displays the last login information for all users.
* **Cron Jobs (/var/log/cron or /var/log/cron.log):**
  + Logs activities related to scheduled tasks and cron jobs.
* **Apache Web Server Logs (/var/log/apache2/ or /var/log/httpd/):**
  + Contains access logs (access.log) and error logs (error.log) for the Apache web server.
* **Nginx Web Server Logs (/var/log/nginx/):**
  + Includes access logs (access.log) and error logs (error.log) for the Nginx web server.
* **System Resource Usage (/var/log/:**
  + Various logs related to system resource usage, including top, dmesg, and others.

**-e**

* **‘echo’ command:**
  + The ‘echo’ command in linux is used to display a message on the terminal. The ‘-e’ option enables the interpretation of backslash escapes in the string.
  + Example:
  + echo -e “Hello\nWorld”
  + In this example, the ‘-e’ option allows the interpretation of the ‘\n’ escape sequence, resulting in the output:

Hello

World

* echo -e Hello \n World

Hello n World

* **‘grep’ command:**
  + The ‘grep’ command is used for searching text patterns in files.
  + The ‘-e’ option is used to specify the pattern to be searched.
  + Example:
  + grep -e “pattern” filename
  + In this example, the ‘-e’ option is used to specify the search pattern “pattern” in the file named ‘filename’.

**>**

is used for output redirection.

It is commonly used to redirect the standard output of a command to a file.

* Redirect Output to a File:
* **command > output.txt**
* This command runs the specified command and redirects its standard output to a file named output.txt. If the file already exists, it will be overwritten. If the file does not exist, it will be created.
* Example:
* e**cho "Hello, World!" > greeting.txt**
* This command writes the text "Hello, World!" to a file named greeting.txt. If greeting.txt already exists, its content will be replaced with the new text.
* Append Output to a File:
* If you want to append the output to an existing file rather than overwriting it, you can use >>:
* **command >> output.txt**
* This appends the standard output of the command to the end of the file.
* Example:
* **echo "Additional text" >> greeting.txt**
* This appends the text "Additional text" to the end of the greeting.txt file.
* These redirection operators (>, >>) are useful for capturing the output of commands and saving it to files.

**tac**

to reverse the order of lines in a file. It stands for "reverse cat," as it is essentially the reverse of the cat command, which displays the contents of a file.

**tac [options] file**

* file: Specifies the name of the file you want to reverse.
* options: You can use various options with the tac command. However, the basic usage typically involves just specifying the file name.
* For example, if you have a file called example.txt with the following content:
* Line 1
  + Line 2
  + Line 3
  + Running the tac command on this file:
* tac example.txt
* Will output:
* Line 3
* Line 2
* Line 1
* This command is useful when you want to quickly view the contents of a file in reverse order, or if you need to process the file in a reverse order using other commands in a command pipeline.

**strings**

is used to extract printable character sequences from a binary file. It is particularly useful for examining binary files, such as executables or libraries, to discover human-readable information embedded in them.

**strings [options] filename**

* filename: Specifies the name of the file you want to analyze.
* For example, if you have an executable file named example.exe, you can use the strings command to extract printable character sequences from it:
* strings example.exe
* This command will output all the readable strings present in the example.exe file.
* -n <number>: Specifies the minimum number of characters a sequence must have to be considered a string. For example, strings -n 6 example.exe will only display strings with six or more characters.
* -a: Causes strings to show all strings, including very short ones. By default, strings ignore strings shorter than four characters.

**alias**

is a way to create a custom shorthand for a longer command or series of commands. It allows you to define your own commands or modify existing ones without actually changing the original command.

To create an alias in Linux, you can use the alias command. The basic syntax is as follows:

**alias alias\_name='command'**

* Here, alias\_name is the name you want to give to your alias, and command is the actual command or series of commands you want to associate with the alias.
* For example, let's say you want to create an alias called ll for the ls -l command to list files with detailed information:
* **alias ll='ls -l'**
* Now, whenever you type ll in the terminal, it will be equivalent to typing ls -l.
* To make your aliases permanent and available every time you open a new terminal session, you can add them to your shell configuration file. The file name depends on the shell you are using:
* For Bash, you can add aliases to the ~/.bashrc file.
* For Zsh, you can add aliases to the ~/.zshrc file.
* For Fish, you can add aliases to the ~/.config/fish/config.fish file.
* Here's an example of adding the ll alias to the Bash configuration:
* echo "alias ll='ls -l'" >> ~/.bashrc
* source ~/.bashrc
* The source ~/.bashrc command is used to apply the changes immediately without restarting the terminal.
* You can create aliases for more complex commands or combine multiple commands into a single alias.
* Aliases are a convenient way to customize your command-line experience and save time by creating shortcuts for frequently used commands.



**unalias**

removes an alias that you have previously defined. The basic syntax is:

**unalias alias\_name**

* Here, alias\_name is the name of the alias you want to remove.
* For example, if you have previously created an alias called ll for the ls -l command and you want to remove it, you can use:
* **unalias ll**
* After running this command, the ll alias will no longer be available in your current shell session. If you added aliases to your shell configuration file (e.g., ~/.bashrc for Bash), you may need to restart your shell or run source on the configuration file to apply the changes:
* **source ~/.bashrc**
* Keep in mind that the removal of an alias is only effective for the duration of the current shell session. If you want to permanently remove an alias, you should edit your shell configuration file to remove the alias definition from there.
* It's also worth noting that you can use the alias command without any arguments to display a list of currently defined aliases in your shell session. This can be helpful to see which aliases are currently active and confirm whether the alias you want to remove exists.

**Shell Expansion**

refers to the process by which the shell interprets certain characters and replaces them with their expanded values before executing a command.

* Wildcard Expansion (\*, ?, [ ]):
* \*: Matches any sequence of characters.
* ?: Matches any single character.
* [ ]: Matches any character within the specified range or set.
* Example:
* ls \*.txt # Lists all files with a .txt extension
* rm file?.txt # Removes files like file1.txt, file2.txt, etc.
* Tilde Expansion (~):
  + ~: Represents the home directory of the current user.
  + ~username: Represents the home directory of the specified user.
  + Example:
  + cd ~ # Changes to the home directory of the current user
  + cd ~username # Changes to the home directory of the specified user
* Variable Expansion ($variable):
  + $variable: Expands to the value of the specified shell variable.
  + Example:
  + my\_var="Hello"
  + echo $my\_var # Prints the value of the variable my\_var
* Command Substitution ($(command) or `command`):
  + $(command): Executes the command and replaces it with the command's output.
  + `command`: An older syntax for command substitution.
  + Example:
  + current\_date=$(date)
  + echo "Current date is: $current\_date"
* Brace Expansion ({a,b,c}):
  + {a,b,c}: Expands to multiple comma-separated values.
  + Example:
  + echo file{1,2,3}.txt # Expands to file1.txt file2.txt file3.txt

**shred**

is used to permanently delete a file which is unable to recover.

shred -u filename

shred --remove filename

--remove will overwrite and then delete the file

This implies that when you use the "rm" command (or any other file deletion command that simply removes the file from the file system's directory), the file is not necessarily completely erased from the storage device.

Instead, only the reference to the file is removed, making the space it occupied available for new data to be written over it.

Using some of the recovery tools we can recover the files deleted by rm using reference in the storage device.

**set -x**

command is used to enable debugging in shell scripts. When this command is used, the shell will print each command and its arguments to the standard error output (stderr) before executing them.

This can be helpful for understanding the flow of a script and identifying issues during execution.

* Here's an example:

**#!/bin/bash**

**set -x**

**echo "This is a debug message."**

**variable="Hello, World!"**

**echo $variable**

**set +x # Disable debugging from this point onward**

**echo "Script execution without debugging."**

* In this script:
* The shebang line (#!/bin/bash) at the beginning of the script specifies that the script should be interpreted and executed using the Bash shell.
* On the other hand, /bin/sh is typically a symbolic link or a binary executable for a POSIX-compatible shell.
* while /bin/bash explicitly specifies the Bash shell, /bin/sh is more generic and could refer to any shell that follows the POSIX standard. In many modern systems, they are effectively the same or provide very similar functionality.
* set -x is used to enable debugging. After this line, each command and its arguments will be printed to stderr before execution.
* set +x is used to disable debugging. It stops the printing of commands.
* When you run the script, you'll see output similar to the following:

**+ echo 'This is a debug message.'**

**This is a debug message.**

**+ variable='Hello, World!'**

**+ echo 'Hello, World!'**

**Hello, World!**

**+ set +x**

**Script execution without debugging.**

* As you can see, each line starting with + indicates the command and its arguments before it is executed. This output is helpful for troubleshooting and understanding the sequence of commands in a script.
* Using set -x is especially useful when you want to trace the execution of your script and identify errors or unexpected behavior. Just make sure to use set +x to disable debugging when you no longer need it, as leaving it enabled can clutter the output.

**;**

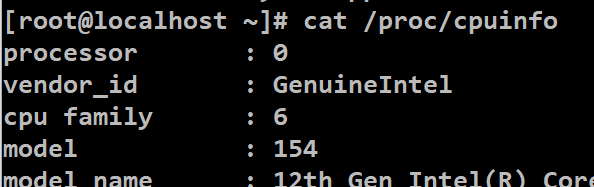
is used as a command separator in the shell. It allows you to execute multiple commands on a single command line, sequentially.

The commands are executed in the order they are written, regardless of the success or failure of previous commands.

**lscpu**

to check the system architecture info.

Can also use “dmidecode”



**command1 ; command2 ; command3**

* For example:
* **echo "Hello, "; echo "world!"**
* In this example, two echo commands are separated by a semicolon, and they are executed one after the other.
* You can also use the semicolon to put multiple commands on the same line in a script or the command line. For instance:
* **#!/bin/bash**

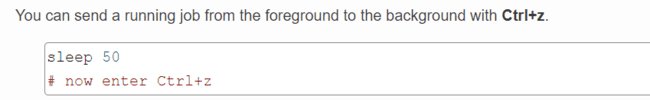
**echo "Command 1"; echo "Command 2"; echo "Command 3"**

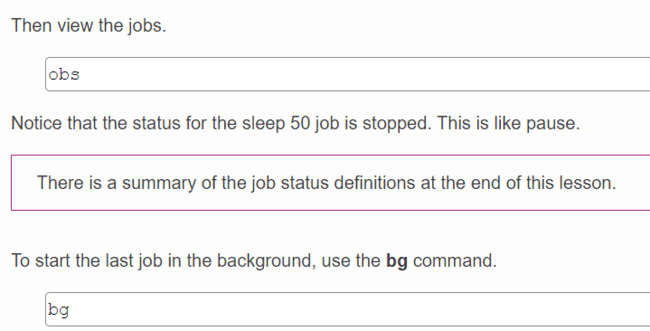
* Each command is separated by a semicolon, and they will be executed sequentially.
* It's important to note that the semicolon is different from the double ampersand (&&) and double pipe (||) operators, which are used for conditional execution based on the success or failure of previous commands. The semicolon simply separates commands and executes them sequentially.

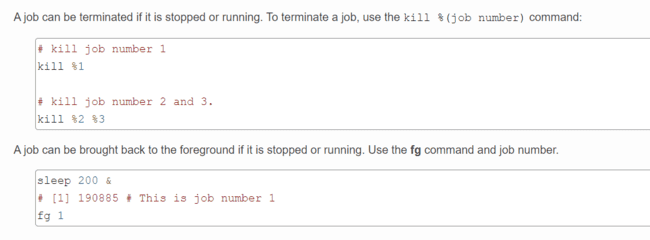
**&**

**Background Execution:**

* When you append an ampersand to the end of a command, it runs the command in the background, allowing you to continue using the shell without waiting for the command to finish. For example:
* **command &**
* This is useful for running commands that may take some time to complete, and you don't want to tie up your terminal while waiting.
* **sleep 10 &** # Sleep for 10 seconds in the background
* Keep in mind that the command's output will still be visible in the terminal unless you redirect it to a file or use other means.



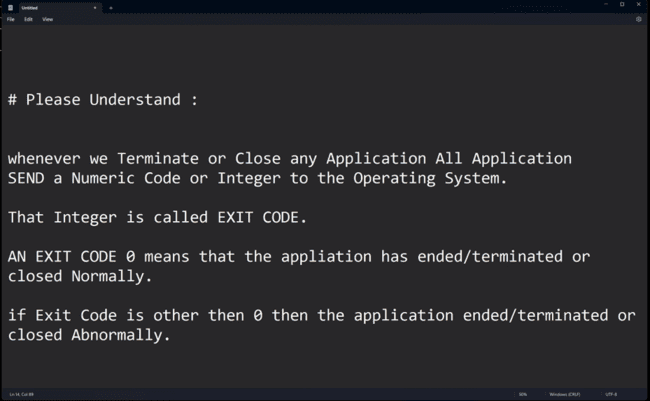




**Control Operators:**

are special characters used in shell commands to control the flow of execution, combine multiple commands, and handle input/output redirection.

* **Semicolon (;):**
* Purpose: Separates multiple commands on a single command line.
* Example:
* command1 ; command2
* **Ampersand (&):**
* Purpose: Runs a command in the background, allowing the shell to continue executing other commands without waiting for the background command to finish.
* Example:
* command &
* **Double Ampersand (&&):**
* Purpose: Executes the second command only if the first command succeeds (returns a zero-exit status).
* Example:
* command1 && command2
* **Double Pipe (||):**
* Purpose: Executes the second command only if the first command fails (returns a non-zero exit status).
* Example:
* command1 || command2
* **Pipeline (|):**
* Purpose: Connects the output of one command to the input of another, allowing you to create a sequence of commands (pipeline).
* Example:
* command1 | command2
* **Redirect Output (>):**
* Purpose: Redirects the output of a command to a file, overwriting the file if it already exists.
* Example:
* command > output.txt
* **Append Output (>>):**
* Purpose: Redirects the output of a command to a file, appending the output to the file if it already exists.
* Example:
* command >> output.txt
* **Redirect Input (<):**
* Purpose: Redirects input from a file to a command.
* Example:
* command < input.txt
* **Command Substitution ($(...) or `...`):**
* Purpose: Replaces the command substitution expression with the output of the enclosed command.
* Example:
* variable=$(command)
* These control operators allow you to build complex command sequences, control the flow of execution, and manage input/output in the Linux shell. Understanding how to use these operators is essential for efficient command-line usage and scripting.



**# (pound symbol)**

character is used for comments in shell scripts and configuration files. Anything following a # on a line is treated as a comment and is ignored by the shell.

Comments are used to add explanatory notes or documentation within the script or configuration file to make the code more readable.

* Here are a couple of examples:
* Shell Script:

**#!/bin/bash**

**# This is a comment**

**echo "Hello, World!" # This is another comment**

* In the above example, both lines starting with # are comments, and they do not affect the execution of the script.

**\**

backslash (\) is an escape character. It is used to escape the interpretation of special characters by the shell. When a backslash precedes a character, it changes the character's meaning, preventing it from being treated as a special character.

* **Escape Characters:**
* \n: Represents a newline character.
* \t: Represents a tab character.
* \\: Represents a literal backslash.
* **echo "Line 1\nLine 2"**
* This would output:
* Line 1
* Line 2
* **Quoting Special Characters:**
  + Used to quote special characters to treat them as literal characters.
  + **echo "This is a single quote: \'"**
  + This would output:
  + This is a single quote: '
* **Continuation Lines:**
  + Used to split a command across multiple lines for readability.
  + echo "This is a long \
  + line continued on the next \
  + line."
  + This would output:
  + This is a long line continued on the next line.
* **Special Characters in File Names:**
  + Used to handle special characters in file names.
  + cat file\ with\ spaces.txt
  + This is necessary when dealing with file names containing spaces or special characters.
* **Escaping Special Characters in Regular Expressions:**
  + Used to escape special characters in regular expressions.
  + grep "word\." file.txt
  + This searches for the literal string "word." in the file.
  + The backslash is a versatile character in the Linux shell, and its usage depends on the context in which it is employed. It is crucial for handling special characters, spaces, and newlines within command strings.

backticks (``) are used for command substitution. Command substitution allows the output of a command to replace the command itself.

This is particularly useful when you want to capture the output of a command and use it as part of another command or assign it to a variable.

* Here is a basic example:
* **current\_date=`date`**

**echo "Today's date is $current\_date"**

* + In this example, the output of the date command is captured and stored in the current\_date variable. The variable is then used in the subsequent echo command.
  + This syntax using backticks for command substitution is equivalent to using the modern $(...) syntax. The example can also be written as:
* **current\_date=$(date)**

**echo "Today's date is $current\_date"**

* + Both backticks and the $(...) syntax achieve the same result, but the latter is recommended for its readability and ease of nesting commands.
  + Here's an example of nesting commands using backticks:
* **result=`echo $(ls)`**

**echo "List of files: $result"**

* + In this case, the inner $(ls) command is used for listing files in the current directory, and its output is captured by the outer backticks. The result is then echoed as part of a larger string

**!!**

the double exclamation mark (!!) is a special history expansion feature in the Bash shell. It is used to refer to the last command that was executed. When you use !! in a command, it gets replaced by the entire last command line.

Here are a few examples:

* **Repeating the Last Command:**
  + !! # Repeats the last command
  + If your last command was, for example, ls -l, then !! would be replaced by ls -l, and the command would be executed again.
* **Using !! in a New Command:**
  + echo "This is a new command" && !!
  + This will execute the echo command and then repeat the last command (whatever it was) using !!.
* **Sudo with !!:**
  + !!
  + If you forgot to use sudo with the last command that required elevated privileges, you can use sudo !! to repeat the last command with superuser privileges.
* **Modifying and Repeating the Last Command:**
* !!:s/old/new/
* This syntax allows you to substitute a part of the last command with a new value. For example:
* echo "Hello, world!"
* !!:s/world/John/
* After the second line, the command becomes echo "Hello, John!" and is executed.
* The !! is a convenient way to refer to the last command without having to retype it. It can be helpful for quickly repeating or modifying commands, especially in the context of command history in the Bash shell.

**history**

is used to display the command history of the current shell session. It shows a list of previously executed commands along with their line numbers. The command history is stored in a file (usually ~/.bash\_history for the Bash shell), and the history command accesses and displays this file.

Here are some common uses of the history command:

* **Displaying Command History:**
  + **history**
  + This command will print the entire command history for the current shell session, with line numbers.
* **Displaying Specific Number of Commands:**
  + **history 10**
  + This will display the last 10 commands from the history.
* **Repeating a Specific Command from History:**
  + **!n**
  + Replace n with the line number of the command you want to repeat. For example, !5 would repeat the command on line 5 of the history.
* **Repeating the Last Command:**
  + **!!**
  + This repeats the last command executed.
* **Searching History:**
  + **history | grep keyword**
  + Replace keyword with the term you are searching for. This will show commands containing the specified keyword.
* **Executing a Command by Partial Match:**
  + **!string**
  + Replace string with a portion of the command you want to repeat. It will execute the most recent command that matches the given string.
* **Executing a Command by Matching the Beginning:**
  + **!^**
  + This repeats the first argument of the previous command.
* **Executing a Command by Matching the Last Argument:**
  + **!$**
  + This repeats the last argument of the previous command.
* **Executing a Command by Matching a Specific Argument:**
  + **!:n**
  + Replace n with the argument position (starting from 1) of the previous command.
  + The history command provides a powerful way to review and reuse previously executed commands, making it easier to navigate and work with the command history in the terminal.
  + You can use Ctrl+R to search for previous commands.

**Yum**

Yellowdog Updater Modified

yum is a package management utility for RPM (Red Hat Package Manager) based systems, such as CentOS and Fedora.

It simplifies the process of installing, updating, and removing software packages on Linux systems.

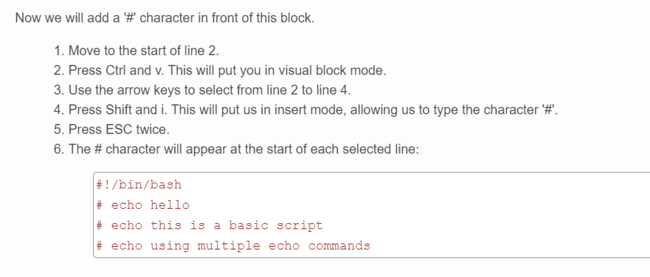
* **Install a Package:**
  + sudo yum install package\_name
  + Replace package\_name with the name of the package you want to install.
* **Update all Packages:**
  + sudo yum update
  + This command updates all installed packages to their latest versions.
* **Search for a Package:**
  + yum search package\_name
  + Use this command to search for packages. Replace package\_name with the name or keywords related to the package.
* **List Installed Packages:**
  + yum list installed
  + Shows a list of all installed packages.
* **Remove a Package:**
  + sudo yum remove package\_name
  + Replace package\_name with the name of the package you want to remove.
* **Clean Package Cache:**
  + sudo yum clean packages
  + Removes any cached packages from the system. This can free up disk space.
  + Caching is used to improve performance and reduce latency by storing copies of frequently accessed data in a location that is closer to the requester than the original source of the data.
* **List Available Updates:**
  + sudo yum list updates
  + Shows a list of packages that have updates available.
* **Check for Package Information:**
  + yum info package\_name
  + Displays detailed information about a specific package.
* **Enable a Repository:**
  + sudo yum-config-manager --enable repository\_name
  + Replace repository\_name with the name of the repository you want to enable.
* **Disable a Repository:**
  + sudo yum-config-manager --disable repository\_name
  + Replace repository\_name with the name of the repository you want to disable.

**vi**

is a text editor commonly found on Unix and Linux systems.

It is a powerful and widely used editor with a steep learning curve. Here are some basic commands to get started with vi:

* **Opening a File:**
  + vi filename
  + Replace filename with the name of the file you want to edit. If the file doesn't exist, vi will create it.
* **Switching to Insert Mode:**
  + Press i to enter insert mode. In insert mode, you can start typing and editing text.
* **Switching to Command Mode:**
  + Press Esc to exit insert mode and return to command mode.
* **Saving Changes:**
  + In command mode, type :w and press Enter to save changes to the file.
* **Saving and Quitting:**
  + In command mode, type :wq and press Enter to save changes and quit vi.
* **Quitting without Saving:**
  + In command mode, type :q! and press Enter to quit vi without saving changes.
* **Moving the Cursor:**
  + Use arrow keys to move the cursor.
  + h - Move left
  + j - Move down
  + k - Move up
  + l - Move right
* **Deleting Text:**
  + x - Delete the character under the cursor.
  + dd - Delete the current line.
* **Copying and Pasting:**
  + yy - Yank (copy) the current line.
  + p - Paste the text after the cursor.
* **Undo and Redo:**
  + u - Undo the last change.
  + Ctrl + r - Redo the undone change.
* **Search and Replace:**
  + In command mode, type / followed by the text to search. Press Enter.
  + To replace, type :s/old\_text/new\_text/g and press Enter.
* Remember that vi has different modes: command mode (for navigation and commands), insert mode (for typing and editing), and visual mode (for selecting text).



**nano**

is a simple and easy-to-use text editor available in Linux. It is often included by default in many Linux distributions and is suitable for users who prefer a straightforward text editor with basic features. nano is especially popular among beginners due to its user-friendly interface.

Here are some basic commands and features of nano:

* **Opening a File:**
  + nano filename
  + This command opens the specified file (filename) in the nano text editor.
* **Basic Navigation:**
  + Use arrow keys to move the cursor.
  + Page Up and Page Down keys scroll the text.
* **Editing:**
  + Simply start typing to insert text.
  + Use the Delete key to remove characters.
  + Cut, copy, and paste operations are available through keyboard shortcuts (Ctrl + K for cut, Ctrl + U for copy, and Ctrl + Shift + U for paste).
* **Saving Changes:**
  + Press Ctrl + O to write the changes to the file.
  + Confirm the file name and press Enter.
* **Exiting:**
  + Press Ctrl + X to exit nano.
* **Search and Replace:**
  + Press Ctrl + W to search for text.
  + Press Ctrl + \ to search and replace text.
* **Line Numbers:**
  + Press Ctrl + C to display the current line and column number.
* **Help:**
  + Press Ctrl + G to access the help menu, which provides a list of available commands.
  + nano provides a user-friendly environment and is suitable for quick edits, configuration file changes, or simple text document creation. It may lack some advanced features found in more powerful text editors like vim or emacs, but its simplicity makes it accessible to users who are new to the command line or prefer a straightforward editing experience.

**Pipes**

a pipe (|) is a mechanism that allows the output of one command to be used as the input for another command. It facilitates the chaining of commands, enabling the creation of powerful and complex command-line operations.

A pipe Takes stdout (output) from the previous command and sends it as stdin(input) to the next command.

* **Example 1: Basic Pipe Usage**
  + ls -l | less
  + Explanation: The ls -l command lists detailed information about files and directories. The output is then passed through a pipe (|) to the less command, which allows you to scroll through the output one page at a time.
* **Example 2: Using Multiple Pipes**
  + ps aux | grep "chrome" | awk '{print $2}'
  + Explanation: The ps aux command lists information about all processes. The output is then piped to grep to filter lines containing "chrome." Finally, the output is piped to awk to print the second column (process IDs).
* **Example 3: Combining Commands with Pipes**
  + echo "Hello" | tr '[:lower:]' '[:upper:]'
  + Explanation: The echo command outputs "Hello," which is then piped to tr (translate) to convert the text to uppercase using character classes.
* **Example 4: Counting Lines with wc**
  + cat file.txt | wc -l
  + Explanation: The cat command outputs the contents of file.txt, and the output is piped to wc -l to count the number of lines in the file.
* **Example 5: Redirecting Output to a File**
  + ls -l | grep "txt" > text\_files.txt
  + Explanation: The ls -l command lists files, and the output is piped to grep to filter for "txt" files. The filtered output is then redirected (>) to a file named text\_files.txt.
* **Example 6: Using sort with Pipes**
  + cat names.txt | sort
  + Explanation: The cat command outputs the contents of names.txt, and the output is piped to sort to alphabetically sort the names.
* **Example 7: Piping into a Command Substitution**
  + echo $(ls | grep "file")
  + Explanation: The ls command lists files, and the output is piped to grep to filter for lines containing "file." The result is then used as a command substitution within the echo command.
* **Example 8: Chaining Commands with ;**
  + ls -l ; pwd
  + Explanation: The semicolon (;) allows for the sequential execution of multiple commands. In this example, ls -l is executed, and then pwd is executed.
* **Example 9: Using tee with Pipes**
  + ls -l | tee output.txt
  + Explanation: The ls -l command lists files, and the output is both displayed on the terminal and written to a file named output.txt using tee.
* **Example 10: Piping find Output to xargs**
  + find . -type f -name "\*.txt" | xargs rm
  + Explanation: The find command searches for files matching the specified criteria, and the output is piped to xargs, which is used to remove (rm) each file found.

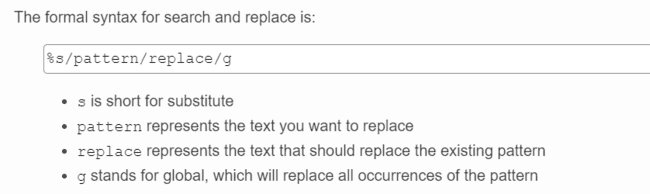
**\*\*\*Filters**

are commands that process and manipulate text or data that is passed through them via standard input (stdin).

They are an integral part of the Unix philosophy, which encourages small, modular tools that can be combined to perform complex tasks.

Here, we'll cover various filter commands, concepts, and examples:

* **cat - Concatenate and Display File Content:**
  + cat file.txt
  + Explanation: Displays the contents of file.txt.
* **grep - Search Text Patterns:**
  + grep "pattern" file.txt
  + Explanation: Searches for lines containing the specified "pattern" in file.txt.
* **sed - Stream Editor:**
  + echo "Hello, World!" | sed 's/Hello/Hi/'
  + Explanation: Uses the sed command to replace "Hello" with "Hi" in the input stream.



* **awk - Text Processing and Pattern Matching:**
  + cat data.txt | awk '{print $1}'
  + Explanation: Uses awk to print the first column of data in data.txt.
* **sort - Sort Lines of Text:**
  + cat names.txt | sort
  + Explanation: Alphabetically sorts the lines in names.txt.
* **uniq - Remove Duplicate Lines:**
  + cat data.txt | sort | uniq
  + Explanation: Sorts data and removes duplicate lines.
* **cut - Extract Sections from Each Line:**
  + cut -d"," -f2 file.csv
  + Explanation: Extracts the second field (column) from a CSV file.
* **tr - Translate Characters:**
  + echo "hello" | tr '[:lower:]' '[:upper:]'
  + Explanation: Translates lowercase letters to uppercase.
* **head - Display First Lines of a File:**
  + head -n 5 file.txt
  + Explanation: Displays the first 5 lines of file.txt.
* **tail - Display Last Lines of a File:**
  + tail -n 10 logfile.txt
  + Explanation: Displays the last 10 lines of logfile.txt.
* **tee - Redirect Output to Multiple Files:**
  + ls -l | tee output.txt
  + Explanation: Displays the output of ls -l on the terminal and writes it to output.txt.
* **wc - Count Words, Lines, and Characters:**
  + wc -l file.txt
  + Explanation: Counts the number of lines in file.txt.
* **tr - Delete Characters:**
  + echo "Hello, World!" | tr -d ' '
  + Explanation: Deletes spaces from the input stream.
* **paste - Merge Lines of Files:**
  + paste file1.txt file2.txt
  + Explanation: Merges lines from file1.txt and file2.txt side by side.
* **grep with Inverted Match:**
  + grep -v "pattern" file.txt
  + Explanation: Displays lines that do not match the specified "pattern."
* **awk with Conditional Statements:**
  + cat data.txt | awk '$3 > 50 {print $1}'
  + Explanation: Uses awk to print the first column for lines where the third column is greater than 50.

**tee**

is used to read from standard input and write to standard output and files simultaneously. It is particularly useful when you want to redirect the output of a command to both the terminal (standard output) and one or more files.

Here's the basic syntax of the tee command:

* **command | tee [OPTION]... [FILE]...**
  + command: The command whose output you want to capture.
  + **tee:** The command itself.
  + **[OPTION]...:** Optional flags that modify the behavior of tee.
  + **[FILE]...:** Optional file names where the output will be written.
* Example Usage:
* **Basic Usage:**
  + echo "Hello, World!" | tee output.txt
  + Explanation: The echo command prints "Hello, World!" to the standard output, and tee captures that output and writes it to both the terminal and a file named output.txt.
* **Appending to a File:**
  + echo "Additional text" | tee -a output.txt
  + Explanation: The -a option tells tee to append to the file (output.txt) rather than overwriting it. The new text is appended to the existing content.
* **Writing to Multiple Files:**
  + ls -l | tee file1.txt file2.txt
  + Explanation: The ls -l command lists files in long format, and tee writes the output both to the terminal and to two files (file1.txt and file2.txt).
* **Reading from a Pipeline:**
  + cat file.txt | grep "pattern" | tee filtered\_output.txt
  + Explanation: The cat command reads the content of file.txt, grep filters lines containing the specified "pattern," and tee captures and writes the filtered output to filtered\_output.txt.
* **Important Options:**
  + -a or --append: Append to the given files rather than overwriting them.
  + -i or --ignore-interrupts: Ignore interrupt signals (e.g., Ctrl+C).
  + --help: Display help information.
  + The tee command is useful in scenarios where you want to capture and preserve the output of a command in both the terminal and one or more files. It provides flexibility in handling command output while avoiding the need for multiple commands or complex redirections.

**grep**

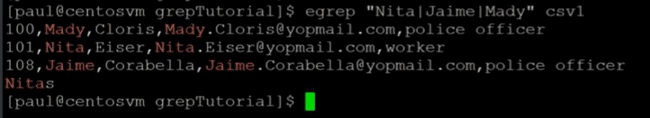
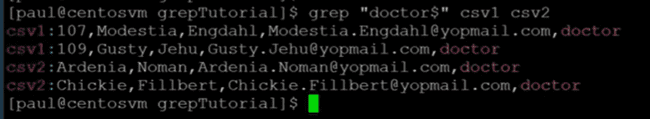
is a powerful command-line utility in Linux used for searching text patterns within files.

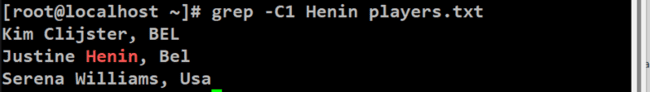
It stands for "Global Regular Expression Print" and is a versatile tool for text pattern matching and extraction.

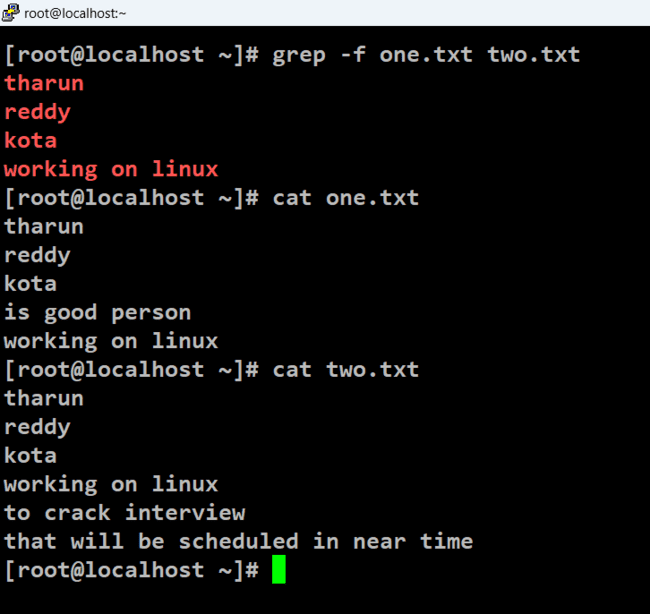
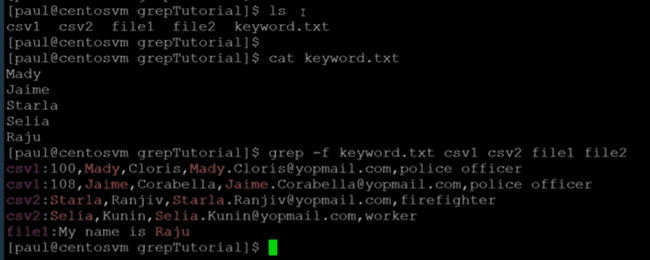
It checks line by line and print lines matching given pattern.

For example we can use grep command to efficiently analyze log files.

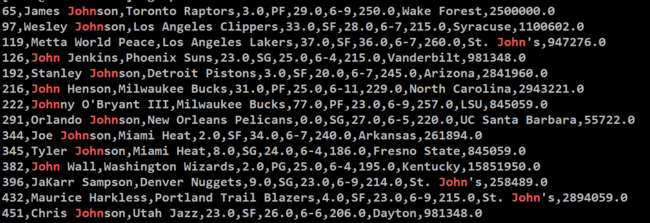
**Basic Syntax:**

* + **grep [OPTIONS] PATTERN [FILE...]**
  + OPTIONS: Additional flags that modify the behavior of grep.
  + PATTERN: The text pattern or regular expression to search for.
  + FILE...: Optional file names. If not provided, grep reads from standard input.
  + Example Usage:
* **Search for a Pattern in a File:**
  + grep "pattern" file.txt
  + Explanation: Searches for the specified "pattern" in file.txt and prints matching lines.
* **Case-Insensitive Search:**
  + grep -i "pattern" file.txt
  + Explanation: Performs a case-insensitive search for the specified "pattern" in file.txt.
* **Display Line Numbers:**
  + grep -n "pattern" file.txt
  + Explanation: Displays line numbers along with the matching lines.
* **Count the Number of Matches:**
  + grep -c "pattern" file.txt
  + Explanation: Counts and prints the number of lines containing the specified "pattern."
* **Recursive Search in Directories:**
  + grep -r "pattern" /path/to/directory
  + Explanation: Recursively searches for the specified "pattern" in all files within the specified directory.
* **Invert Match (Exclude Lines):**
  + grep -v "pattern" file.txt
  + Explanation: Displays lines that do not match the specified "pattern."
* **To search multiple keywords in a file**
  + **grep -e “keyword1” -e “keyword2” file**
  + Explanation: displays all the lines those match the patterns keyword1 and keyword2 in file
  + We can use egrep command for the multiple keywords search
  + **egrep “keyword1|keyword2” file**
  + so instead or writing -e for every keyword to be searched we can simply use egrep command.
  + 
* **Display Only Matching Part:**
  + grep -o "pattern" file.txt
  + Explanation: Displays only the part of each line that matches the specified "pattern."
* **Use Regular Expressions:**
  + grep -E "[0-9]+" file.txt
  + Explanation: Uses extended regular expressions (ERE) to search for lines containing one or more digits in file.txt.
* To print the matching line which start with given keyword
  + grep “^keyword” file
  + ex: grep “^hello” file.txt
  + this means print all the lines starting with keyword hello in the file file.txt
* If you just wanna search but don’t want to print on terminal
  + grep -q “keyword” file (-q quietly)
  + useful in scripting purposes want to search a pattern for further action but don’t want to print.
  + To check if the above command has run successfully we can use exit status of the above command with echo $? Output is 0 means successful.
  + If no successful pattern is found the command returns non zero number.
  + If the command itself is wrong ex: wrong directory name no such file or directory error will be displayed.
  + -q is only for the pattern matching not for command error.
  + To suppress command error use -s.
* If you want to suppress error message
  + grep -s “keyword” file
* To print the matching line which end with given keyword
  + grep “keyword$” file
  + ex: “bye$” file.txt
  + This means print all the lines in the file file.txt ending with the pattern bye.
  + 
* **Display Lines Before and After Match:**
  + grep -B 2 -A 2 "pattern" file.txt
  + Explanation: Displays two lines before and after each line containing the specified "pattern."
* **Display Lines Before and After**

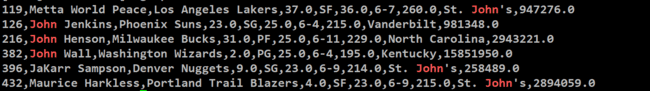


* **Count Occurrences of Each Match:**
  + grep -o -c "pattern" file.txt
  + Explanation: Counts and prints the occurrences of each match separately.
* To only print file names which matches given keyword
  + grep -l “keyword” file1 file2….
* To get the keywords/pattern from a file and match with a another file
  + grep -f keyword.txt file
  + 
  + This is useful when we need to search pattern for multiple strings.
  + In that case we write all those patterns in a file and use -f to match the patterns with the other files easily.
  + Instead of writing multiple patterns in the command we can keep them in a file and match the pattern with desired file we want to search in.
  + 
* Without giving the complete patter of the text we need to find grep finds it.
  + For ex: if Tharun needs to be searched in a file called file.txt
  + You can do this using sub pattern of Tharun like Tha
    - grep “Tha” file.txt
  + this also gives the lines having pattern Tharun.
  + To avoid this and find exact match of the given pattern we need to use -w flag.
    - grep -w “Tha” file.txt
  + this only checks for the pattern Tha not for all the matching patterns to Tha like Tharun.







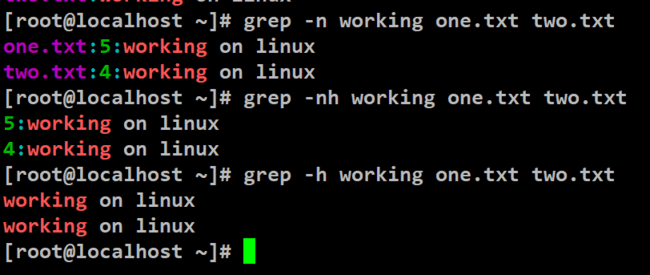


When we do grep on multiple files to match a pattern we get file names also for every matched line.

To remove those file names at starting of the output and print on the lines then we can use suppress flag -h.

Ex: How to suppress file names while search a given keyword in multiple files?

**grep -h “keyword” file1 file2 …**

****

But it’s hard to identify the file from which the particular belongs to if we use -h.

There can be use cases to remove names of the files so

we use -h.

* **Important Options:**
  + -i or --ignore-case: Perform case-insensitive matching.
  + -n or --line-number: Display line numbers with output lines.
  + -r or -R or --recursive: Read all files under each directory recursively.
  + -v or --invert-match: Invert the sense of matching to select non-matching lines.
  + -o or --only-matching: Print only the matched (non-empty) parts of matching lines.
  + -B or --before-context: Print N lines of leading context before matching lines.
  + -A or --after-context: Print N lines of trailing context after matching lines.
  + -c or --count: Suppress normal output and display a count of matching lines.

**wc**

command in Linux is used to count lines, words, and characters in a file or standard input. It provides a simple way to analyze and display the basic statistics of a text file.

* **Basic Syntax:**
  + wc [OPTIONS] [FILE...]
  + OPTIONS: Additional flags that modify the behavior of wc.
  + FILE...: Optional file names. If not provided, wc reads from standard input.
* **Example Usage:**
  + Count Lines in a File:
  + wc -l file.txt
  + Explanation: Displays the number of lines in file.txt.
* **Count Words in a File:**
  + wc -w file.txt
  + Explanation: Displays the number of words in file.txt.
* **Count Characters in a File:**
  + wc -c file.txt
  + Explanation: Displays the number of characters in file.txt.
* **Count Lines, Words, and Characters in a File:**
  + wc file.txt
  + Explanation: Displays the line, word, and character counts in file.txt.
* **Count Lines in Multiple Files:**
  + wc -l file1.txt file2.txt
  + Explanation: Displays the number of lines in both file1.txt and file2.txt.
* **Count Lines in a Directory Recursively:**
  + wc -l -r /path/to/directory
  + Explanation: Recursively counts lines in all files within the specified directory.
* **Count Lines from Standard Input:**
  + echo -e "Line 1\nLine 2\nLine 3" | wc -l
  + Explanation: Uses echo to produce three lines of text, and wc -l counts the lines from standard input.
* **Important Options:**
  + -l or --lines: Count lines.
  + -w or --words: Count words.
  + -c or --bytes: Count bytes.
  + -m or --chars: Count characters (equivalent to -c).
  + -L or --max-line-length: Display the length of the longest line.

**cut**

cut command is used to filter columns

command in Linux is used to extract sections from each line of a file or from standard input. It is particularly useful for working with structured text data, such as delimited files (e.g., CSV files) or fixed-width formatted data. Here's the basic syntax and some examples of using the cut command:

**Basic Syntax:**

cut OPTION... [FILE]...

OPTION...: Additional flags that modify the behavior of cut.

FILE...: Optional file names. If not provided, cut reads from standard input.

* **Example Usage:**
* **Cut by Delimiter (e.g., Comma in CSV):**
  + cut -d',' -f2 file.csv
  + Explanation: Uses a comma (-d',') as the delimiter and extracts the second field (-f2) from each line in file.csv.
* **Cut by Tab as Delimiter:**
  + cut -f2 -d$'\t' file.csv
  + Explanation: Uses a tab (-d$'\t') as the delimiter and extracts the second field (-f2) from each line in file.tsv.
* **Cut Fixed Width Fields:**
  + cut -c1-5 file.txt
  + Explanation: Extracts characters 1 to 5 (-c1-5) from each line in file.txt.
* **Cut by Character Positions:**
  + cut -c1,3,5 file.txt
  + Explanation: Extracts characters at positions 1, 3, and 5 (-c1,3,5) from each line in file.txt.
* **Cut by Range of Characters:**
  + cut -c1-3,7-9 file.txt
  + Explanation: Extracts characters 1 to 3 and 7 to 9 (-c1-3,7-9) from each line in file.txt.
* **Cut by Delimiter with Range:**
  + echo "John,Doe,30" | cut -d',' -f1-2
  + Explanation: Uses a comma (-d',') as the delimiter and extracts fields 1 to 2 (-f1-2) from the input string.
* **Cut by Comma and Space Delimiters:**
  + cut -d', ' -f2 file.txt
  + Explanation: Uses a comma followed by a space (-d', ') as the delimiter and extracts the second field (-f2) from each line in file.txt.
* **Important Options:**
  + -f FIELDS: Select only these fields.
  + -d DELIMITER: Use DELIMITER as the field delimiter.
  + -c CHARACTERS: Select only these characters.
  + --complement: Complement the set of selected bytes, characters, or fields.

**diff**

is used to find the difference between to files

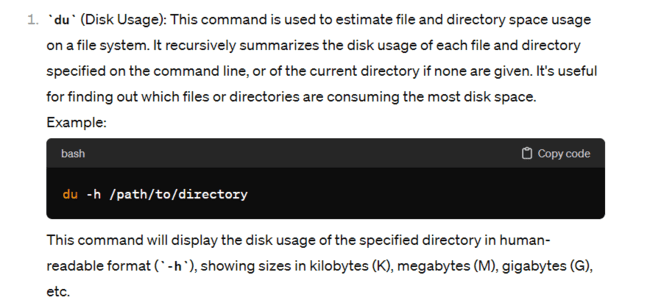
diff <file1> <file2>

**tr**

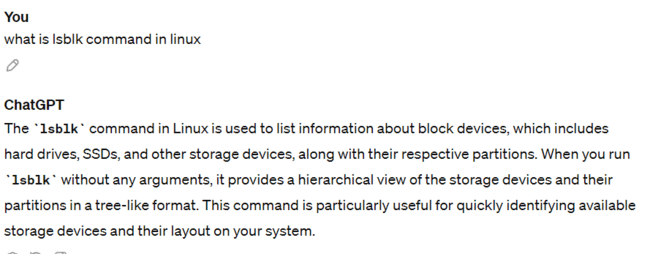
command in Linux is a versatile utility used for translating or deleting characters. It is particularly useful for transforming or deleting characters in a stream of text. The basic syntax is as follows:

* **Basic Syntax:**
  + **tr [OPTION] SET1 [SET2]**
  + OPTION: Optional flags that modify the behavior of tr.
  + SET1: The set of characters to be replaced.
  + SET2: The set of replacement characters.
  + Example Usage:
* **Translate Lowercase to Uppercase:**
  + echo "hello" | tr '[:lower:]' '[:upper:]'
  + Explanation: Translates lowercase characters to uppercase. The character classes [:lower:] and [:upper:] represent all lowercase and uppercase letters, respectively.
* **Delete Spaces:**
  + echo "Hello, World!" | tr -d ' '
  + Explanation: Deletes (removes) spaces from the input string.
* **Replace Characters:**
  + echo "abc123" | tr '123' 'XYZ'
  + Explanation: Replaces '1' with 'X', '2' with 'Y', and '3' with 'Z'.
* **Delete Newlines:**
  + echo -e "Line 1\nLine 2" | tr -d '\n'
  + Explanation: Deletes newlines from the input, effectively joining the lines.
* **Translate Tabs to Spaces:**
  + echo -e "Tab\tSeparated" | tr '\t' ' '
  + Explanation: Translates tabs to spaces in the input.
* **Delete Specific Characters:**
  + echo "Hello, World!" | tr -d 'aeiou'
  + Explanation: Deletes the vowels 'a', 'e', 'i', 'o', and 'u' from the input string.
* **Squeeze Duplicate Characters:**
  + echo "Hello World!" | tr -s ' '
  + Explanation: Squeezes consecutive spaces into a single space.
* **Important Options:**
  + -c or --complement: Use the complement of SET1.
  + -d or --delete: Delete characters in SET1, do not translate.
  + -s or --squeeze-repeats: Replace each input sequence of a repeated character that is listed in SET1 with a single occurrence.

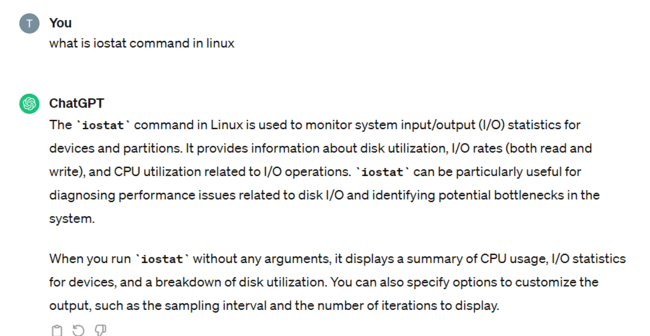
**du**

****

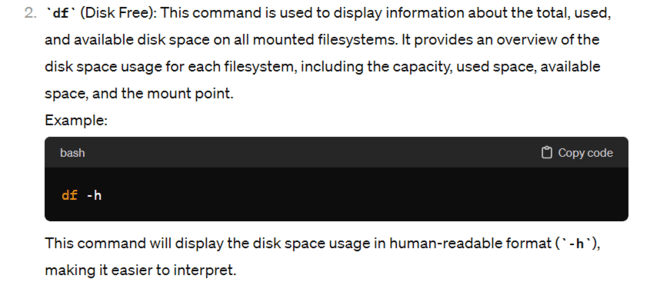
**lsblk**

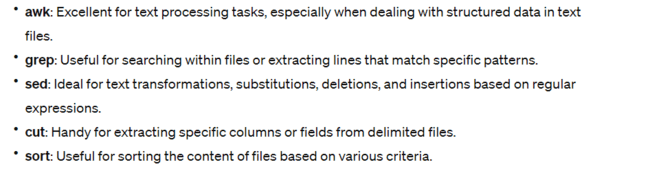
****

**iostat**

****

**df**

****

****

**sort command**

in Linux is used for sorting lines of text files or standard input. It arranges lines in ascending or descending order based on the contents of the lines.

* **Basic Syntax:**
  + **sort [OPTION]... [FILE]...**
  + OPTION: Additional flags that modify the behavior of sort.
  + FILE...: Optional file names. If not provided, sort reads from standard input.
* Example Usage:
* **Sort Lines in Ascending Order:**
  + sort file.txt
  + Explanation: Sorts the lines in file.txt in ascending order (lexicographically).
* **Sort Lines in Descending Order:**
  + sort -r file.txt
  + Explanation: Sorts the lines in file.txt in descending order.
* **Sort Numerically:**
  + sort -n numbers.txt
  + Explanation: Sorts the lines in numbers.txt in numerical order.
* **Sort and Remove Duplicate Lines:**
  + sort -u file.txt
  + Explanation: Sorts the lines in file.txt and removes duplicate lines.
* **Sort CSV File by a Specific Column:**
  + sort -t',' -k2,2 -n data.csv
  + Explanation: Sorts the lines in data.csv using ',' as the field separator (-t',') and sorts numerically (-n) based on the second column (-k2,2).
    - In the sort command -k option, the argument specifies the key definition for sorting. The argument to -k has the form start[,end], where start and end are field numbers separated by a comma. The start field is the key used for sorting, and the sorting considers characters starting from the start field up to the end field.
    - In your specific command:
    - **sort -t',' -k2,2 -n data.csv**
    - -t',': Specifies that the field separator is a comma.
    - -k2,2: Specifies that the sorting key is the second field (2). The ,2 part indicates that the sorting should be based on characters in the second field only. This means that the sorting considers the content of the second field and not beyond it.
    - -n: Indicates a numeric sort, treating the values in the specified key as numbers rather than strings.
    - data.csv: The file to be sorted.
    - So, the 2 after -k2 indicates that you want to sort based on the content of the second field only, considering characters in that field for the sorting operation. This is useful when you have a CSV (Comma-Separated Values) file, and you want to sort the lines based on the numerical values in the second column.
* **Sort and Display Line Prefixes:**
  + sort -k1,1 -b file.txt
  + Explanation: Sorts the lines in file.txt based on the first column, considering leading blanks as insignificant (-k1,1 -b).
* **Sort Human-Readable Sizes:**
  + du -h | sort -h
  + Explanation: Uses du -h to display human-readable sizes and sorts them in ascending order (sort -h).
* **Sort Randomly:**
  + echo -e "apple\norange\nbanana" | sort -R
  + Explanation: Sorts the lines in the input randomly (sort -R).

If sort is getting input from preceding command and the column to be sorted is the last column then sort can be applied simply like (sort -t, -k2 -n).

Else if sort is getting input from preceding command and the column to be sorted is the starting or the middle column then sort must be carefully and specifically applied to that column only like (sort -t, -k2,2 -n).

* **Important Options:**
  + -r or --reverse: Reverse the order of the sort.
  + -n or --numeric-sort: Compare according to string numerical value.
  + -u or --unique: Output only the first of an equal run.
  + -t CHAR or --field-separator=CHAR: Use CHAR as the field separator.
  + -k KEYDEF or --key=KEYDEF: Sort by a key defined by KEYDEF.

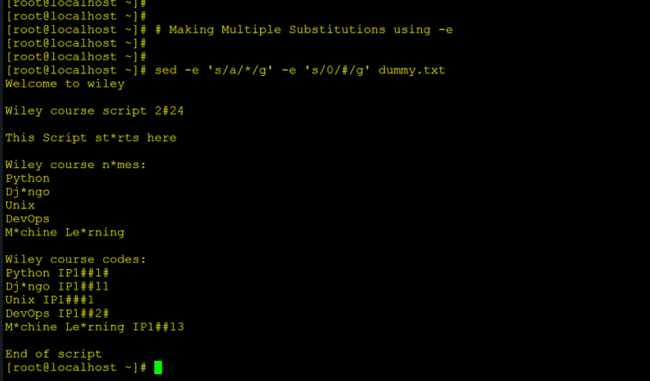
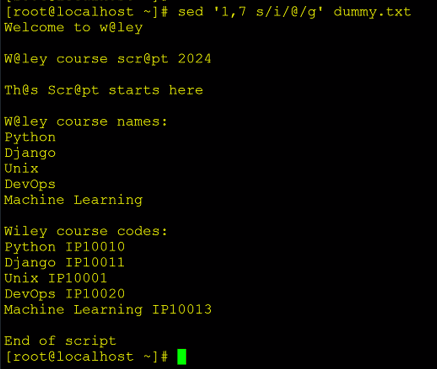
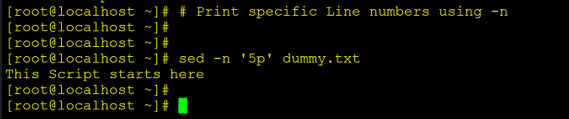
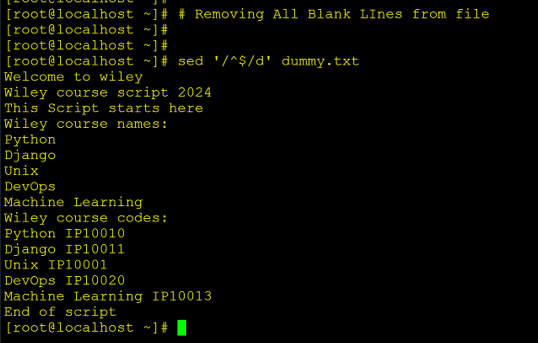
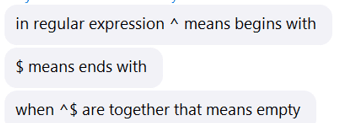
**uniq command**

in Linux is used to remove or filter adjacent duplicate lines from a sorted file or from input data. It's commonly used in conjunction with the sort command. Here's the basic syntax and some examples of using the uniq command:

* **Basic Syntax:**
  + uniq [OPTION]... [INPUT [OUTPUT]]
  + OPTION: Optional flags that modify the behavior of uniq.
  + INPUT: The file or input source containing sorted lines. If not specified, uniq reads from standard input.
  + OUTPUT: The file where the unique lines will be written. If not specified, uniq writes to standard output.
* Example Usage:
* **Remove Adjacent Duplicates from Sorted File:**
  + sort file.txt | uniq
  + Explanation: Sorts the lines in file.txt and removes adjacent duplicate lines.
* **Count and Remove Adjacent Duplicates:**
  + sort file.txt | uniq -c
  + Explanation: Counts the number of occurrences of each line and removes adjacent duplicates. The count is displayed before each line.
* **Count Only, Keep Adjacent Duplicates:**
  + sort file.txt | uniq -d
  + Explanation: Counts and displays only the lines that have duplicates (occurred more than once) in the sorted input.
* **Count Only, Keep Unique Lines:**
  + sort file.txt | uniq -u
  + Explanation: Counts and displays only the lines that are unique (occurred only once) in the sorted input.
* **Specify Number of Characters to Compare:**
  + sort file.txt | uniq -w 3
  + Explanation: Compares only the first 3 characters of each line to determine duplicates.
* **Ignore a Specific Number of Characters:**
  + sort file.txt | uniq -s 2
  + Explanation: Ignores the first 2 characters of each line when comparing for duplicates.
* **Important Options:**
  + -c or --count: Prefix lines with the number of occurrences.
  + -d or --repeated: Only print duplicate lines.
  + -u or --unique: Only print unique lines.
  + -i or --ignore-case: Ignore case differences when comparing lines.
  + -w N or --check-chars=N: Compare at most N characters in lines.
  + -s N or --skip-chars=N: Skip the first N characters before comparing lines.

**sed**

short for "stream editor," is a powerful text processing tool in Linux. It's commonly used for performing basic text transformations on an input stream (a file or input from a pipeline). sed operates by reading input line by line and applying specified text transformations.

* **Basic Syntax:**
  + sed [OPTIONS] 'script' [input-file...]
  + OPTIONS: Optional flags that modify the behavior of sed.
  + script: The set of sed commands enclosed in single quotes.
  + input-file: The file(s) to process. If not specified, sed reads from standard input.
* **Example Usage:**
* **Replace Text in a File:**
  + sed 's/old-text/new-text/' input.txt
  + Explanation: Replaces the first occurrence of "old-text" with "new-text" in each line of input.txt.
* **Replace All Occurrences in a Line:**
  + sed 's/old-text/new-text/g' input.txt
  + Explanation: Replaces all occurrences of "old-text" with "new-text" in each line of input.txt.
* **Replace Only on Specific Lines:**
  + sed '2s/old-text/new-text/' input.txt
  + Explanation: Replaces the first occurrence of "old-text" with "new-text" only in the second line of input.txt.
* **Delete Lines Matching a Pattern:**
  + sed '/pattern/d' input.txt
  + Explanation: Deletes all lines containing the specified "pattern" from input.txt.
* **Print Only Matching Lines:**
  + sed -n '/pattern/p' input.txt
  + Explanation: Prints only the lines containing the specified "pattern" from input.txt.
* **Substitute and Save Changes to Original File:**
  + sed -i 's/old-text/new-text/' input.txt
  + Explanation: Substitutes "old-text" with "new-text" in input.txt and saves the changes to the original file.
* **Print Specific Range of Lines:**
  + sed -n '2,5p' input.txt
  + Explanation: Prints lines 2 to 5 from input.txt.
* **Append Text After Each Line:**
  + sed 'a\Additional text' input.txt
  + Explanation: Appends "Additional text" after each line in input.txt.
* **Replace Only Lines Matching a Pattern:**
  + sed '/pattern/s/old-text/new-text/' input.txt
  + Explanation: Replaces "old-text" with "new-text" only in lines containing the specified "pattern" in input.txt.
* **Making Multiple Substitutions using -e**
* 
* 
* 
* 
* 
* **Important Options:**
  + -n or --quiet or --silent: Suppress automatic printing of pattern space.
  + -e SCRIPT or --expression=SCRIPT: Add the script to the commands to be executed.
  + -i[SUFFIX] or --in-place[=SUFFIX]: Edit files in place (creates backup with the specified suffix).
  + -r or --regexp-extended: Use extended regular expressions.

refer SED Commands.pdf in wiley edge folder for more practical commands

**awk**

is a powerful text processing tool in Linux, and it has numerous applications.

Syntax:

awk options ‘pattern {action}’ file\_name

echo “Hello” | awk options ‘pattern {action}’

options:

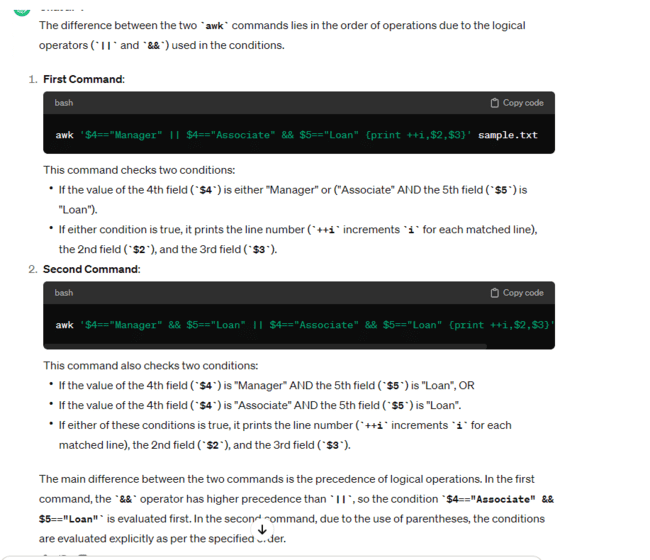
-F field separator

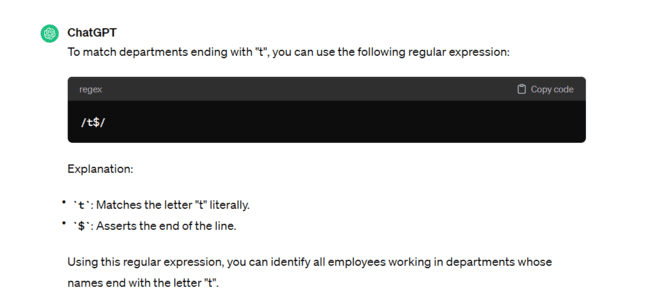
-v var=value

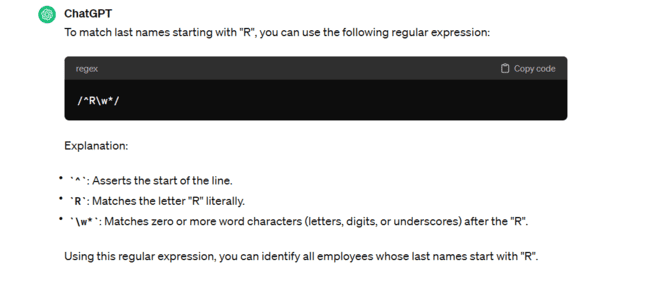
-f file

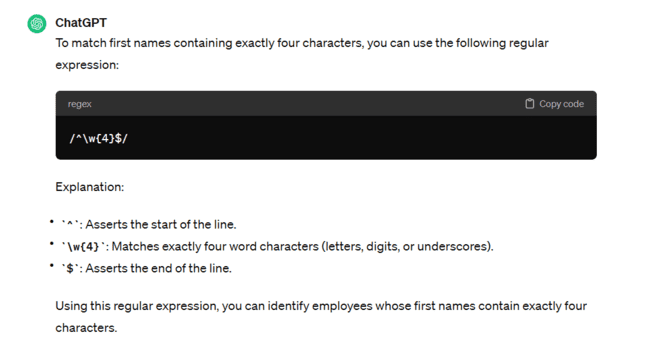
Terms used in AWK

* NR – No.of record/row
* NF - No.of fields
* $0 - Print everything
* $1, $2 - Field no.
* **Printing Specific Columns:**
  + Print the first and third columns of a file:
  + awk '{ print $1, $3 }' filename
* **Pattern Matching:**
* Print lines containing the word "error":
* awk '/error/' filename
* **Conditional Statements:**
  + Print lines where the second column is greater than 50:
  + awk '$2 > 50 { print }' filename
* **Calculations:**
  + Calculate and print the average of the second column:
  + awk '{ sum += $2 } END { print "Average: " sum/NR }' filename
* **Changing Field Separator:**
  + Use a comma as the field separator:
  + awk -F',' '{ print $1 }' filename.csv
* **Displaying Line Numbers:**
  + Print line numbers along with the lines:
  + awk '{ print NR, $0 }' filename
* **Deleting Lines Based on a Condition:**
  + Delete lines where the second column is less than 10:
  + awk '$2 >= 10' filename
* **Finding and Replacing:**
  + Find and replace "old" with "new" in a file:
  + awk '{ gsub(/old/, "new"); print }' filename
* **Summing Values in a Column:**
  + Calculate and print the sum of the third column:
  + awk '{ sum += $3 } END { print "Sum: " sum }' filename
* **Formatting Output:**
  + Format the output with headers:
  + awk 'BEGIN { printf "%-10s %-10s\n", "Name", "Age" } { printf "%-10s %-10s\n", $1, $2 }' filename
* **Processing Multiple Files:**
  + Process multiple files and print their names:
  + awk '{ print FILENAME, $0 }' file1.txt file2.txt
* **Counting Occurrences:**
  + count the occurrences of a specific word:
  + awk '/pattern/ { count++ } END { print "Count: " count }' filename











**to print last column?**

awk ‘{print $NF}’ sample.txt

NF is a pre-defined variable it stores the no.of fields so NF has the count of total fields and prints the final field.

**To search a word using awk?**

Steps (mentioning delimiter using -F if present is default step)

Step 1 : awk ‘{}’ sample.txt #define structure

Step 2 : awk ‘{print $0}’ sample.txt #select required fields

Step 3 : awk ‘/Paul/{print $0}’ sample.txt #search the /Paul/ keyword

**Print the line number where Paul is present?**

Step 1 : awk ‘{}’ sample.txt #define structure

Step 2 : awk ‘{print $0}’ sample.txt #select required data

Step 3 : awk ‘{print NR, $0}’ sample.txt #adding row numbers

Step 4 : awk ‘/Paul/{print NR, $0}’ sample.txt #select /Paul/ keyword

Here NR or row number is pre-defined variable which stores record/row number.

**Print a particular row?**

Step 1 : awk ‘{}’ sample.txt #define structure

Step 2 : awk ‘NR==6 {}’ sample.txt #mention the condition

Step 3 : awk ‘NR==6 {print $0}’ sample.txt #mention data to be printed

after condition is met.

So the overall syntax is that

* in the curly braces we provide the action to be done.
* Before the curly braces and inside single quote we mention the pattern or the condition.
* If the pattern or condition matches the action is taken.

**Print range of lines from 3 to 6?**

awk ‘NR==3,NR==6 {print NR, $0}’ sample.txt

**print row number of empty lines?**

awk ‘NF==0 {print NR}’ sample.txt

NF==0 because empty line doesn’t contain any fields.

**Search lines with multiple keywords?**

awk ‘/Raju|Sham|Alex/ {print $0}’ sample.txt

use | to add multiple search keywords.

**Ignore case while searching?**

BEGIN is used to tell to do something before searching for a pattern ex: ignoring the case.

awk ‘BEGIN{IGNORECASE=1} /raju/ {print $0}’ sample.txt

**How to search for a character in particular column?**

awk ‘$2 ~ /a/ {print $0}’ sample.txt

$2 ~ /a/

Here we are saying check for character a in all the names of column 2.

**Print the data whose salary is greater that 50k?**

awk -F, ‘$NF>50000 {print $0}’ emp.csv

**how to handle multiple delimiter file?**

awk -F[,:-] ‘{print $2}’ multi\_del.txt

In the above example it has , and - and : as the delimiters.

**Output only active/inactive for httpd service?**

systemctl status httpd | awk ‘NR==3 {print $2}’

for firewalld service?

systemctl status firewalld.service | awk ‘NR==3 {print $2}’

**Print the data of the file excluding row 1?**

ls -lt | awk ‘NR>1 {print $NF}’

**Read logs in range of time?**

less /var/log/messages | awk ‘ $3>=“01:05:55” && $3<=“01:05:56” ’

**Built in functions:**

**Replace a word**

gsub = global substitute

awk ‘{gsub(“Tharun”,“Tharun Reddy”); print $0}’ sample.txt

**length of line/field**

using length() function

awk ‘{print $2, length($2)}’ sample.txt

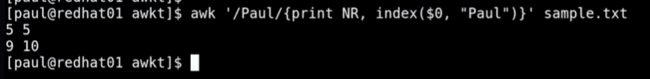
awk ‘{print length($0)}’ sample.txt

**index position of a word in a given line**

using index() function

awk ‘/Paul/{print NR, $0}’ sample.txt

awk ‘/Paul/{print NR, index($0, “Paul”)}’ sample.txt



Paul is at 5th character position in line 5.

Paul is at 10th character position in line 9.

Print character in lower or upper case?

awk ‘{print tolower($5)}’ sample.txt

awk ‘{print toupper($5)}’ sample.txt

**AWK Scripting Concepts**

awk

‘BEGIN{start\_action}

pattern/condition {action}

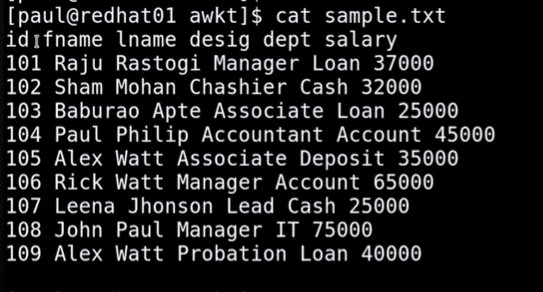
END{stop\_action}’

file\_name

* What are BEGIN and END in the above syntax?
* Here pattern/condition {action} works directly on the file
* Before and after reading or scanning a file we need to do some action for that purpose we use two blocks called BEGIN{} and END{}.
* We can observe that everything is between single quotes.

Basic ex:

Data:



Print description of above data first and at the end print the data?

Step 1: awk ‘BEGIN{} {} END{}’ sample.txt

Step 2: awk ‘BEGIN{} {print $0} END{}’ sample.txt

Before reading the file ({print $0}) we first want to give the description using BEGIN{}

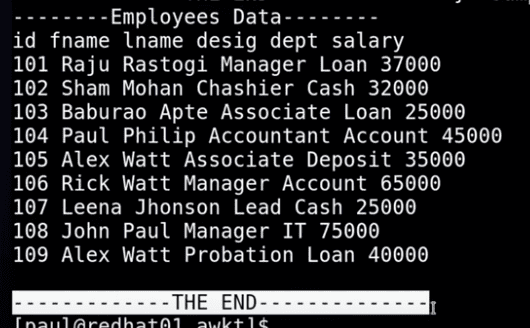
Step 3:

awk ‘BEGIN{print “-----Employees Data -----} {print $0} END{}’ sample.txt

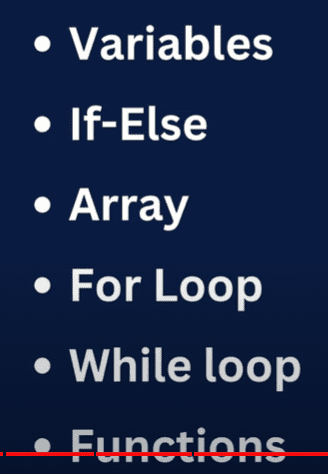
Step 4:

Now we will the action to be done at the end after reading the file

awk ‘BEGIN{print “-----Employees Data -----"} {print $0} END{print “---------------------THE END----------------"}’ sample.txt







How to find total/sum of salary?

Step1: awk ‘’ sample.txt

Step 2: awk ‘BEGIN{} {} END{}’ sample.txt

Step 3:

Before starting the scan we will first create a variable as sum=0

awk ‘BEGIN{sum=0} {} END{}’ sample.txt

Step 4:

Now we can use the variable in the main task i.e, in the action

awk ‘BEGIN{sum=0} {print sum} END{}’ sample.txt

now the action print sum will be checked for every line in the file.

Step 5:

Suppose if we want to print the action only once.

awk ‘BEGIN{sum=0} {} END{ print sum }’ sample.txt

Above small change must be done.

So it must be placed in the end task.

Step 6:

To specify the output clearly.

awk ‘BEGIN{sum=0} {} END{print “Sum of salary: ” sum}’ sample.txt

step 7:

as we know action part goes through all the rows.

So now we keep the summation part in the action part and let summation happen on the last column.

awk ‘BEGIN{sum=0} {sum=sum+$NF} END{print “Sum of salary : ” sum}’ sample.txt

actually we don’t need to mention sum=0 in the BEGIN{} even then the query works.

awk ‘{sum=sum+$NF} END{print “Sum of salary : ” sum}’ sample.txt

**to count total no.of employees?**

awk ‘{count++} END{print “Total employees are: “ count}’ sample.txt

but in the output we can observe it is 11 which is not correct it includes the header of the file and blank line at the end.

To avoid this we need to make below changes

awk ‘NR>1 {if($NF>0)count++} END{print “Total employees are: “ count}’ sample.txt

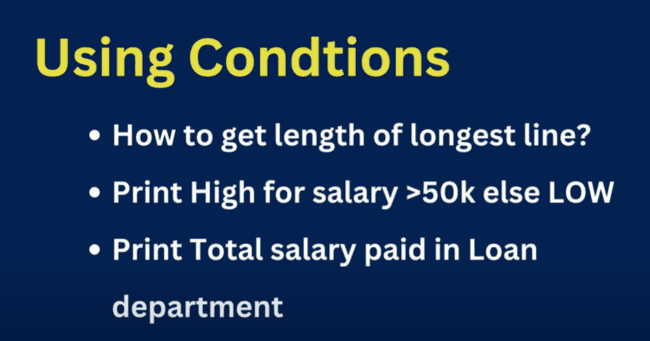
**How to ignore headers/first row to count no.of users?**

**How to print no.of lines?**

**How to find average salary?**

awk 'NR>1 {if($NF>0)count++; sum+=$NF} END{print "Average salary is : "sum/count}' sample.txt





Length of the longest line?

awk ‘{if(length($0)>max)max=length($0)} END{print “Length of longest line is “ max}’ sample.txt

print high for salary >50k else LOW?

awk ‘{if($NF>50000)$7=”High”;else $7=”Low”; print $0}’ sample.txt

$7 is used to create new column at the end and print result in that column.

Print Total salary paid in Loan department?

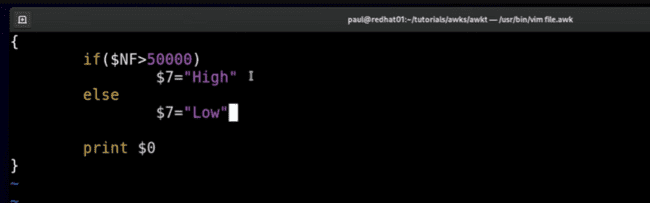
awk ‘{if($5==”Loan”)sum+=$NF} END{print “Total salary in loan dept “ sum}’ sample.txt



Let’s take the above example to run the awk code in a file.

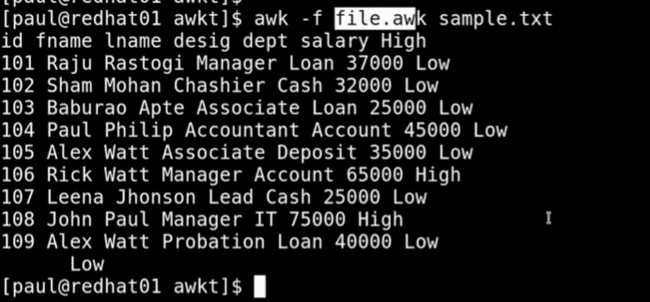
Let’s take file name as file.awk extension is not mandatory .awk helps identify it as a awk command file.

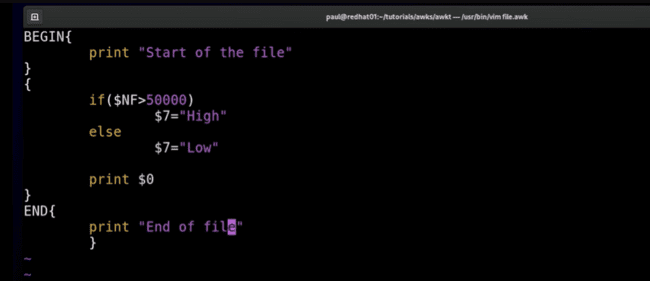
Just paste the action part condition in the file file.awk

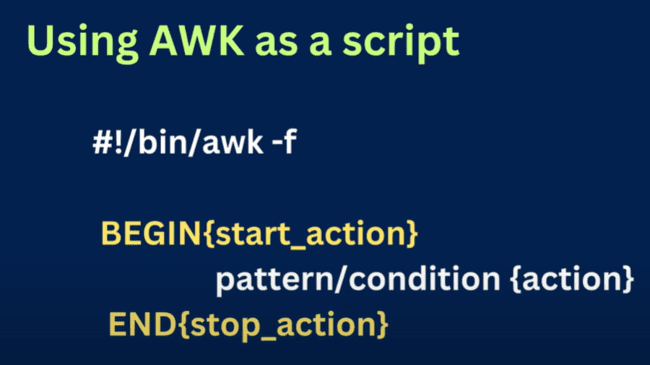


The difference is that when we use the command in normal awk we need to give semicolon.

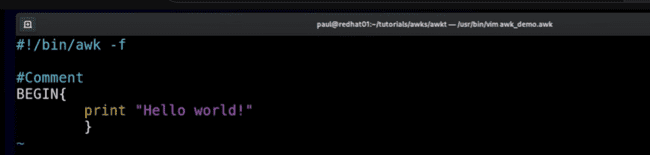
Where as in awk file when we write in the newline that is only the separation no need to give semicolon.

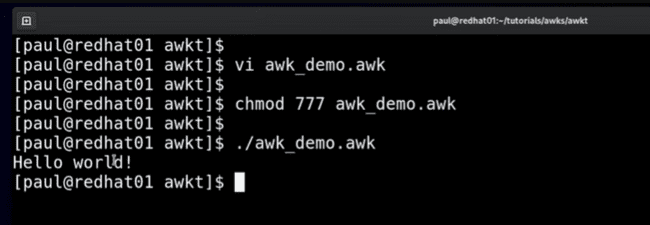


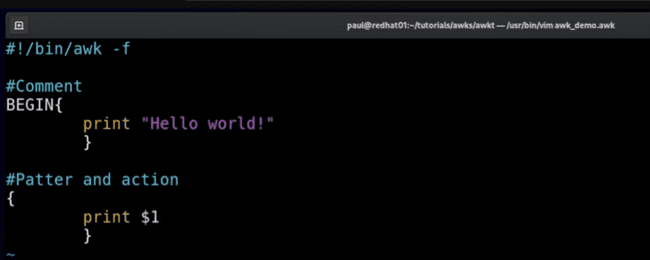


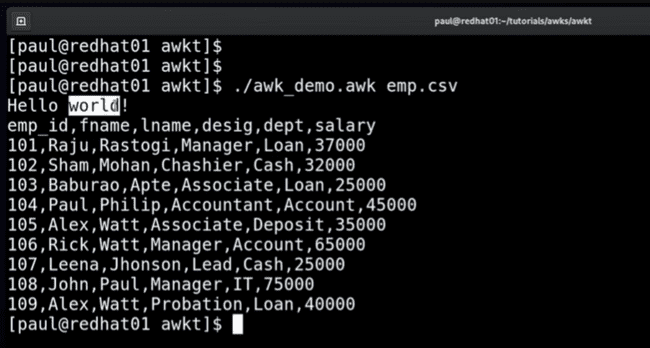


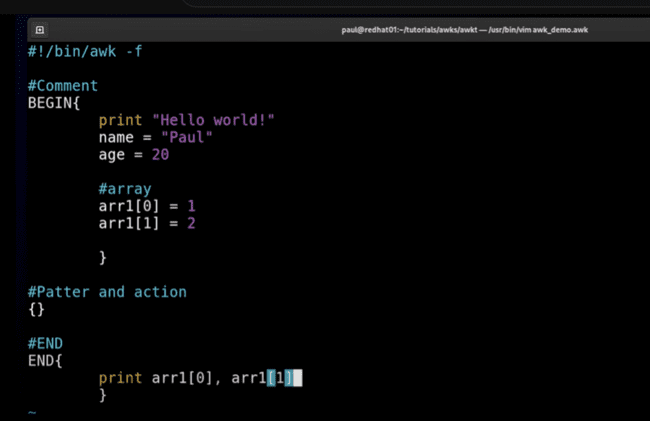


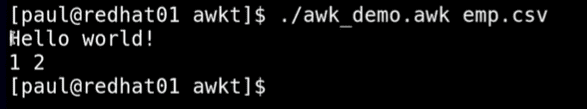


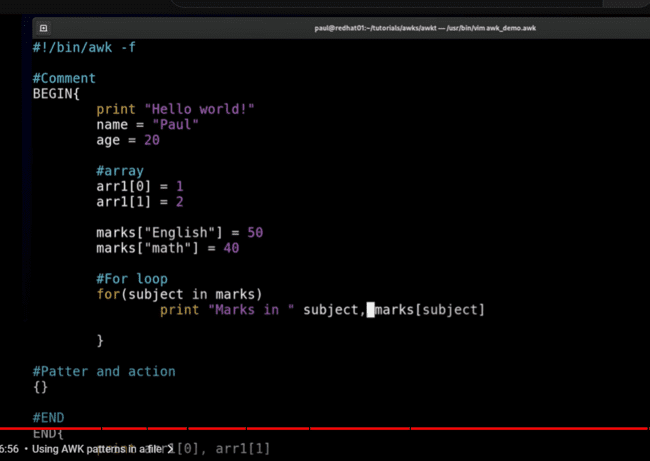


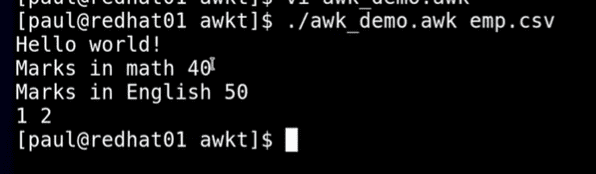


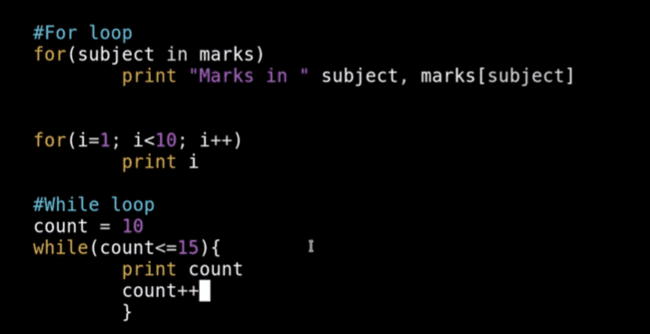












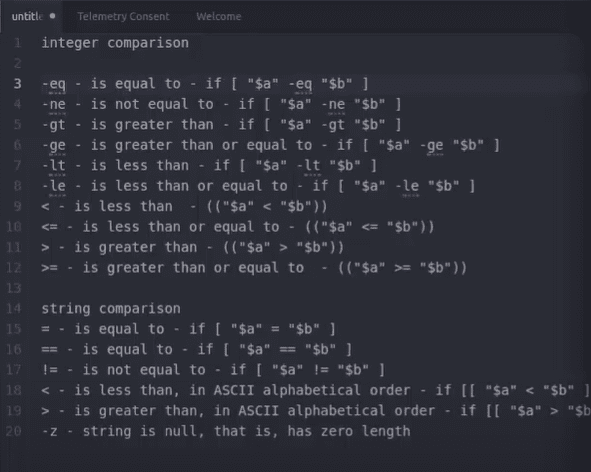
**Shell Scripting**

* Shell scripting is an important part of process automation. (here process automation refers to any task which can be automated via programming).
* Scripting helps you write sequence of commands in a file and then execute them.
* This saves your time because you don’t have to write commands again and again.
* What is a Bash Script?
  + A Bash script Executes Line by Line.
  + File Extension of BashScript is .sh
  + A Bash Script starts with SHEBANG.
  + Shebang is a combination of BASH i.e # and BAND i.e ! > #!
  + This is an example of shebang followed by bash shell path.
  + #!/bin/bash

**input**

there are several ways to read input from the user.

* **read Command:**
  + The read command allows you to prompt the user for input and store the result in a variable.
  + echo "Enter your name:"
  + read name
  + echo "Hello, $name!"
  + The user's input will be stored in the variable name.
* **Command-Line Arguments:**
  + You can also read input from command-line arguments passed to the script.
  + # Assuming the script is named script.sh
  + # Usage: ./script.sh arg1 arg2
  + arg1=$1
  + arg2=$2
  + echo "Argument 1: $arg1"
  + echo "Argument 2: $arg2"
  + Here, $1 and $2 represent the first and second command-line arguments.
  + The $ symbol is used to reference the value of the variable.
* **Using read with Prompt:**
  + You can combine the read command with a prompt to make it more user-friendly.
  + read -p "Enter your age: " age
  + echo "Your age is $age"
  + The -p option allows you to provide a prompt on the same line as the input.
* **Reading Passwords Securely:**
  + To read passwords without displaying them on the screen, you can use the read -s option.
  + read -s -p "Enter your password: " password
  + echo "Password entered."
  + The -s option makes the input silent.



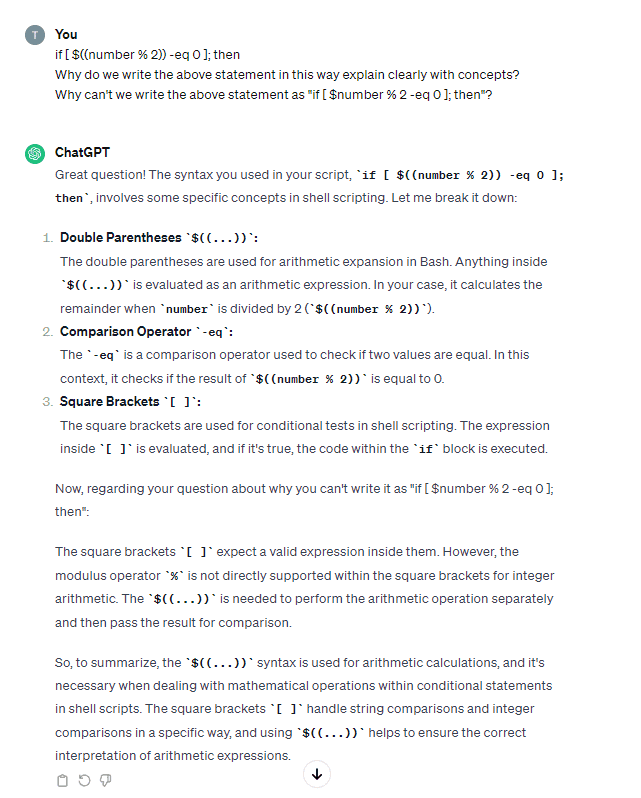
* From the line 9 to 12 we can observe that when <, <=, >, >= are used with integers then the expression is enclosed in double parenthesis – (( “$a” < “$b” )).
* From the line 18 and 19 we can observe that when < , > are used with strings then the expression must be enclosed between double square brackets – [[ “$a” < “$b” ]].

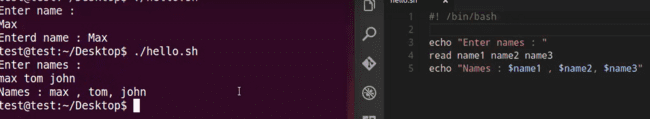
**Conditional operators**

In Linux shell scripting, conditional operators are used to make decisions based on the success or failure of commands or conditions. Here are some common conditional operators:

* **if statement:**
  + The if statement is used to conditionally execute a block of code based on a specified condition.
  + if [ condition ]; then
  + # Code to be executed if the condition is true
  + fi
* **elif statement:**
  + The elif statement allows you to check multiple conditions sequentially after an initial if statement.
  + if [ condition1 ]; then
  + # Code to be executed if condition1 is true
  + elif [ condition2 ]; then
  + # Code to be executed if condition2 is true
  + else
  + # Code to be executed if none of the conditions are true
  + fi
* **else statement:**
  + The else statement is used in conjunction with if or elif to specify code to be executed when none of the preceding conditions are true.
  + if [ condition ]; then
  + # Code to be executed if the condition is true
  + else
  + # Code to be executed if the condition is false
  + fi
* **Logical Operators:**
  + **Logical operators like -a (AND), -o (OR), and ! (NOT) can be used to combine multiple conditions.**
  + if [ condition1 -a condition2 ]; then
  + # Code to be executed if both condition1 and condition2 are true
  + fi
* **Comparison Operators:**
  + **Comparison operators such as -eq (equal), -ne (not equal), -lt (less than), -le (less than or equal), -gt (greater than), and -ge (greater than or equal) are used to compare values.**
  + if [ "$num1" -eq "$num2" ]; then
  + # Code to be executed if num1 is equal to num2
  + fi
* These operators are commonly used in shell scripts to create decision-making structures based on various conditions.







**Program 1**

#!/bin/bash

echo “Hello, World!”

Certainly! Below is a simple "Hello, World!" script written in Bash, a common shell scripting language in Linux:

#!/bin/bash

echo "Hello, World!"

**Explanation:**

#!/bin/bash: This line is called the shebang. It tells the system to use the Bash shell to interpret and execute the script.

echo "Hello, World!": The echo command is used to print the specified text to the standard output (usually the terminal). In this case, it prints the string "Hello, World!".

Save the above script in a file, for example, hello\_world.sh. Make the script executable using the following command:

**chmod +x hello\_world.sh**

Now, you can run the script using:

**./hello\_world.sh**

This will execute the script and print "Hello, World!" to the terminal.

**Linux shell scripting,** you can perform arithmetic operations using various methods. Here are some common approaches:

* **Using expr command:**
  + The expr command evaluates expressions and performs arithmetic operations. Here's an example:

#!/bin/bash

# Arithmetic operations using expr

a=5

b=3

sum=$(expr $a + $b)

echo "Sum: $sum"

difference=$(expr $a - $b)

echo "Difference: $difference"

product=$(expr $a \\* $b)

echo "Product: $product"

quotient=$(expr $a / $b)

echo "Quotient: $quotient"

remainder=$(expr $a % $b)

echo "Remainder: $remainder"

* In this script, the backslash () before the asterisk (\*) is used to prevent it from being interpreted as a wildcard by the shell.
* **Using double parentheses (( )):**
* You can perform arithmetic operations within double parentheses:

#!/bin/bash

# Arithmetic operations using double parentheses

a=5

b=3

sum=$((a + b))

echo "Sum: $sum"

difference=$((a - b))

echo "Difference: $difference"

product=$((a \* b))

echo "Product: $product"

quotient=$((a / b))

echo "Quotient: $quotient"

remainder=$((a % b))

echo "Remainder: $remainder"

* **Using let command:**
* The let command allows arithmetic operations:

#!/bin/bash

# Arithmetic operations using let

a=5

b=3

let "sum = a + b"

echo "Sum: $sum"

let "difference = a - b"

echo "Difference: $difference"

let "product = a \* b"

echo "Product: $product"

let "quotient = a / b"

echo "Quotient: $quotient"

let "remainder = a % b"

echo "Remainder: $remainder"

* **Using bc command for floating-point arithmetic:**
* If you need to perform floating-point arithmetic, you can use the bc command:

#!/bin/bash

# Floating-point arithmetic using bc

a=5.5

b=3.2

sum=$(echo "$a + $b" | bc)

echo "Sum: $sum"

product=$(echo "$a \* $b" | bc)

echo "Product: $product"

**File Permissions:**

**-rw-r--r--. 1 root root 63 Jan 21 23:52 hell\_world.sh**

* **File Permissions (-rw-r--r--.):**
  + The first field represents file permissions.
  + In this case, it's -rw-r--r--..
  + The leading - indicates that it is a regular file.
  + The next three characters (rw-) indicate read, write, and no execute permissions for the file owner.
  + The next three characters (r--) indicate read-only permissions for the group.
  + The final three characters (r--) indicate read-only permissions for others.
  + The dot (.) at the end may indicate additional SELinux information.
* **Number of Hard Links (1):**
  + The second field is the number of hard links to the file.
  + In this case, it's 1, which means there is only one hard link pointing to the file.
* **Owner (root):**
  + The third and fourth fields represent the owner and group of the file.
  + In this case, both are set to root.
  + root is the owner, and root is also the group.
* **File Size (63):**
  + The fifth field is the size of the file in bytes.
  + In this case, it's 63 bytes.
* **Modification Time (Jan 21 23:52):**
  + The next fields represent the last modification time of the file.
  + In this case, it's Jan 21 23:52.
* **File Name (hell\_world.sh):**
  + The final field is the name of the file.
  + In this case, it's hell\_world.sh.

Which permission allow a user to run an executable file (script)?

A: we need to provide executable(x) permission to the user.

**chmod**

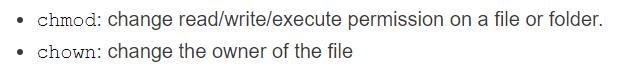
**getfacl <filename>**

used to see file permissions clearly for a particular file

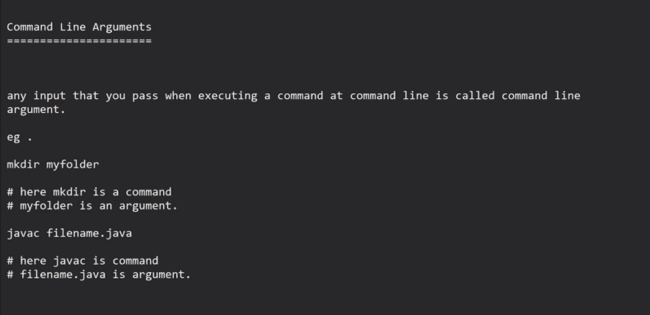
* To remove execute permission for the owner of the file or directory:
  + **chmod u-x your\_file\_or\_directory**
* If you want to remove execute permission for the group:
  + **chmod g-x your\_file\_or\_directory**
* And to remove execute permission for others:
  + **chmod o-x your\_file\_or\_directory**
* If you want to remove execute permission for everyone (owner, group, and others) at once:
  + **chmod a-x your\_file\_or\_directory**
* If you want to remove execute permissions recursively for a directory and its contents, you can use the -R option:
  + **chmod -R a-x your\_directory**

is a command in Unix and Unix-like operating systems (including Linux) that is used to change the permissions of a file or directory. The name "chmod" stands for "change mode." File permissions determine who can access a file and what actions they can perform on it.

* The basic syntax of the **chmod command** is as follows:
* chmod [options] mode file
* Here, options are optional parameters, mode specifies the permissions you want to set, and file is the name of the file or directory whose permissions you want to change.
* **Modes:**
  + **Numeric Mode:** The mode can be represented numerically using a three-digit octal number. Each digit corresponds to the permission set for a particular user category (owner, group, and others).
  + The first digit represents the owner's permissions.
  + The second digit represents the group's permissions.
  + The third digit represents others' (everyone else) permissions.
  + Each digit is a combination of read (4), write (2), and execute (1) permissions. For example:
  + chmod 755 file
  + In this case, the owner has read (4) + write (2) + execute (1) permissions (7), and the group and others have read (4) + execute (1) permissions (5).
* **Symbolic Mode:** The mode can also be represented symbolically using letters (u, g, o, a for user, group, other, and all, respectively) and operators (+ for add, - for subtract, and = for set). For example:
  + chmod u+rwx,g+rx,o+r file
  + This command adds read, write, and execute permissions for the owner, read and execute permissions for the group, and read permission for others.
* **Options:**
  + -R: Recursively changes permissions for the specified directory and its subdirectories.
  + -v: Verbose mode. Displays information about the changes made.
  + -c: Like -v, but only displays information when a change is made.



**Command Line Arguments**

****

**User Management in Linux:**

involves creating, modifying, and deleting user accounts, as well as managing user attributes and access rights. Key commands and configuration files are used for these tasks. Here are the essential aspects of user management in Linux:

* **Adding Users:**
  + To add a new user, you can use the useradd command:
  + sudo useradd username
  + This creates a new user account named "username." You may also need to set the user's password using the passwd command:
  + sudo passwd username
* **Modifying User Attributes:**
  + Changing Password:
  + sudo passwd username
  + Changing Username:
  + sudo usermod -l newusername oldusername
  + Adding User to a Group:
  + sudo usermod -aG groupname username
* **Deleting Users:**
  + To delete a user account, use the userdel command:
  + sudo userdel username
  + This command removes the user but doesn't remove the home directory. If you want to remove the home directory as well, use the -r option:
  + sudo userdel -r username
* **Listing Users:**
  + To list all users on the system, you can use the cat command on the /etc/passwd file or use the getent command:
  + cat /etc/passwd
  + getent passwd
* **Granting Sudo Privileges:**
  + To grant a user sudo privileges, add the user to the sudo group or edit the /etc/sudoers file:
  + sudo usermod -aG sudo username
* **Configuring User Defaults:**
  + User defaults are specified in the /etc/login.defs file. You can modify this file to set default values for user accounts.
* **User Home Directory:**
  + By default, the home directory for a user is /home/username. The useradd command creates this directory automatically.
* **Password Policies:**
  + Password policies are defined in the /etc/security/pwquality.conf and /etc/security/pwcheck.conf files. These files control password strength requirements.
* **Locking/Unlocking User Accounts:**
* To lock or unlock a user account, you can use the passwd command with the -l (lock) or -u (unlock) option:
* sudo passwd -l username # Lock
* sudo passwd -u username # Unlock
* **SSH Key Authentication:**
  + llow or deny SSH access using keys, manage the ~/.ssh/authorized\_keys file in the user's home directory.

User management in Linux involves a combination of commands and configuration file edits. Always exercise caution, especially when making changes that can affect system security and user access.

**uname**

command in Linux is used to display system information. It provides information about the operating system and the system hardware. When you run the uname command without any options, it typically prints the kernel name, network node hostname, kernel release, kernel version, and machine hardware.

* Here's the basic syntax:
* **uname [options]**
  + Commonly used options include:
  + -a: Displays all available information.
* **uname -a**
  + -s: Prints the kernel name.
* **uname -s**
  + -n: Shows the network node hostname.
* **uname -n**
  + -r: Displays the kernel release.
* **uname -r**
  + -v: Prints the kernel version.
* **uname -v**
  + -m or -i: Shows the machine hardware.
* **uname -m**
  + The output of the uname -a command, for example, might look something like this:
  + Linux your\_hostname 5.4.0-91-generic #114-Ubuntu SMP Thu Oct 22 09:09:44 UTC 2020 x86\_64 x86\_64 x86\_64 GNU/Linux
* **In this output:**
  + Linux is the kernel name.
  + your\_hostname is the network node hostname.
  + 5.4.0-91-generic is the kernel release.
  + #114-Ubuntu SMP Thu Oct 22 09:09:44 UTC 2020 is additional version information.
  + x86\_64 indicates the machine hardware.

The uname command is useful for obtaining basic information about the system from the command line.

**/etc/passwd**

file is a crucial system file in Linux and Unix-like operating systems. It stores essential information about user accounts on the system. Each line in the /etc/passwd file represents a user account and contains several fields separated by colons (:). The fields typically include:

* **Username:** The login name for the user.
* **Password**: The user's encrypted password (historically stored here, but now usually in /etc/shadow for security reasons).
* **User ID (UID):** A unique numerical identifier for the user.
* **Group ID (GID):** The primary group identifier for the user.
* **User Info:** Additional information about the user (often the full name or a description).
* **Home Directory:** The user's home directory.
* **Login Shell:** The default shell for the user when logging in.
* Here is an example entry from the /etc/passwd file:
  + **john:x:1001:1001:John Doe:/home/john:/bin/bash**
* In this example:
  + Username: john
  + Password: (The "x" indicates that the encrypted password is stored in the /etc/shadow file.)
  + UID: 1001
  + GID: 1001
  + User Info: John Doe
  + Home Directory: /home/john
  + Login Shell: /bin/bash
* It's important to note that modern Linux systems use the /etc/shadow file to store encrypted passwords and other security-related information to enhance system security.
* To view the content of the /etc/passwd file, you can use various commands such as cat, less, or more. For example:
  + cat /etc/passwd

Be cautious when modifying the /etc/passwd file manually, as incorrect changes can lead to login issues. User management commands like useradd, usermod, and userdel are recommended for adding, modifying, and deleting user accounts.

**/etc/shadow**

file in Linux is a critical system file that stores the encrypted passwords of user accounts. Unlike the older /etc/passwd file, which used to store encrypted passwords but has been replaced for security reasons, /etc/shadow is more secure and only accessible by privileged users.

* Each line in the /etc/shadow file represents a user account and contains several fields separated by colons (:). The fields typically include:
* **Username:** The login name for the user.
* **Password**: The user's encrypted password.
* **Last Password Change:** The number of days since the password was last changed.
* **Minimum Password Age:** The minimum number of days the user must keep a password before changing it.
* **Maximum Password Age:** The maximum number of days a password is valid.
* **Password Warning Period:** The number of days before password expiration that a warning is given.
* **Password Inactivity Period:** The number of days after the password has expired until the account is locked.
* **Account Expiry Date:** The date when the account will be locked.

Here is an example entry from the /etc/shadow file:

**john:$6$D6L1ZgSb$cyx.ku6KmC8WE.DlzvgCNHQGbVmhnPLFEqFTM1en44nrESsosNlr8kZQjFJ.Et/VJ2dDXJ.VJGsd4yLlZoNLl.:18440:0:99999:7:::**

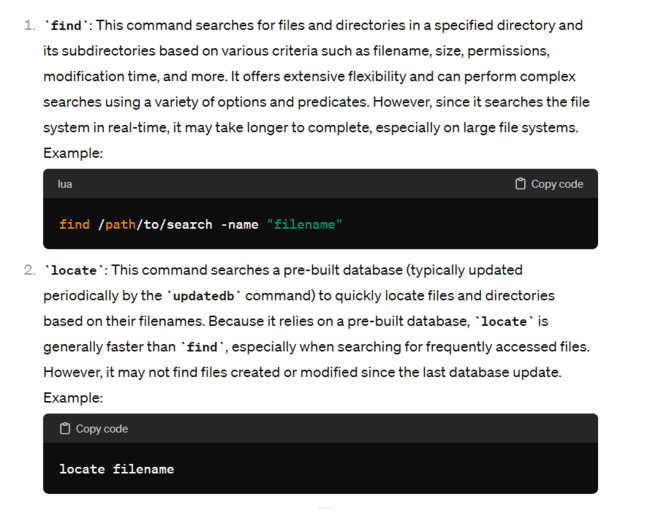
In this example:

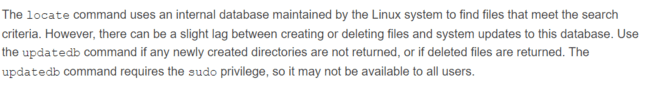
* **Username:** john
* **Encrypted Password:** $6$D6L1ZgSb$cyx.ku6KmC8WE.DlzvgCNHQGbVmhnPLFEqFTM1en44nrESsosNlr8kZQjFJ.Et/VJ2dDXJ.VJGsd4yLlZoNLl. (encrypted using SHA-512)
* **Last Password Change:** 18440 days since January 1, 1970 (Unix epoch).
* **Minimum Password Age**: 0 (no minimum).
* **Maximum Password Age:** 99999 (no maximum).
* **Password Warning Period**: 7 days before password expiration.
* **Password Inactivity Period**: 7 days after password expiration.
* **Account Expiry Date**: ::: (no expiry).

The use of /etc/shadow enhances system security by restricting access to the password information, making it readable only by users with elevated privileges. Normal users cannot directly view or modify this file.

**find**

**locate**

****

****

**su**

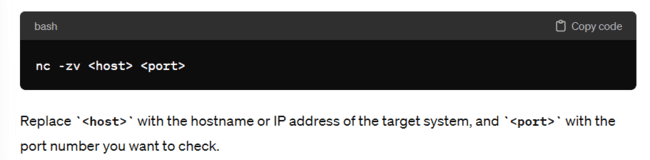
command in Linux is used to switch to another user account in a shell session. It stands for "switch user" or "substitute user." By default, if you run su without any arguments, it will attempt to switch to the root user (superuser). If you provide a username as an argument, it will attempt to switch to that specific user.

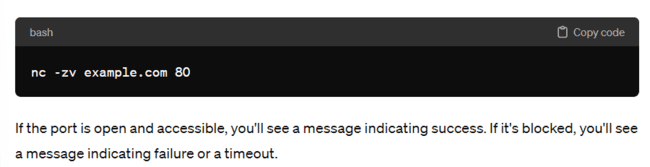
* Here is the basic syntax:
* **su [options] [username]**
  + If you run su without specifying a username, it will switch to the root user:
* **su**
  + To switch to a specific user, you can provide the username as an argument:
* **su username**
  + You will be prompted to enter the password for the target user.
* Options:
  + **-c command:** Specify a command to run as the new user.
  + **-l or --login:** Simulate a full login by creating a new shell session with the target user's environment.
  + **-s shell:** Use a specific shell instead of the default shell for the target user.
* Example:
  + **su -l username**
  + This command simulates a full login for the specified user, setting up the environment variables as if the user had logged in directly.
  + After switching to another user, you can use the exit command to return to your original user or close the terminal window.

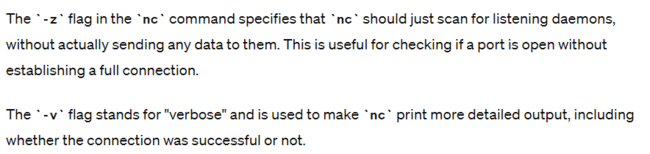
It's important to note that su requires the password of the target user unless you are switching to the root user, in which case it requires the root password. Additionally, on some systems, you might need to be a member of the sudo group or have the appropriate permissions to use su. On modern systems, it's common to use sudo to execute commands with elevated privileges instead of relying on su.











**sudo**

(superuser do) command in Linux is used to execute commands with elevated privileges. It allows a permitted user to execute a command as the superuser (or another user), as specified by the security policy configured in the /etc/sudoers file.

* Here is the basic syntax of the sudo command:
  + sudo [options] command [arguments]
  + **options:** Additional flags or settings.
  + **command:** The command to execute with elevated privileges.
  + **arguments:** The arguments to pass to the command.
  + For example, to update the system package list using the apt package manager:
* **sudo apt update**
  + The sudo command typically requires the user to enter their own password to confirm their identity before executing the command with elevated privileges. This adds a layer of security by ensuring that only authorized users can perform administrative tasks.
* Examples of sudo Usage:
* **Install Software:**
  + sudo apt install packageName
* **Edit System Configuration Files:**
  + sudo nano /etc/configfile.conf
* **Restart Services:**
  + sudo systemctl restart serviceName
* **Remove Files:**
  + sudo rm /path/to/file
* **Switch to the Root User Shell:**
  + sudo -i
* **Configuring sudo:**
  + The configuration for sudo is stored in the /etc/sudoers file. It is recommended to use the visudo command to edit this file to prevent syntax errors:
* **sudo visudo**
  + This opens the sudoers file in the system's default text editor. Only users listed in the sudoers file and members of the sudo group are typically allowed to use sudo. The configuration file allows administrators to define which users can run which commands with elevated privileges and under what conditions.
* Using sudo is a safer practice than using the root account directly, as it limits potential damage caused by accidental or malicious commands.

**/etc/sudoers**

file in Linux contains configuration information for the sudo command. It defines the rules and permissions regarding which users or groups are allowed to execute commands as superusers (root) or other users, and under what conditions.

* To edit the /etc/sudoers file, you should use the visudo command, which opens the file in a safe manner and performs syntax checking to prevent errors that could lock you out of your system.
  + **sudo visudo**
  + This command opens the sudoers file in the default text editor defined in your system. Commonly, this is set to use the nano or vi text editors.
  + The typical structure of an entry in the sudoers file is as follows:
  + user\_or\_group host=(runas\_user:runas\_group) command(s)
  + **user\_or\_group:** Specifies the user or group to which the rule applies.
  + **host:** Defines the host or hosts on which the rule is effective.
  + **runas\_user:** Specifies the user as whom the command should be run.
  + **runas\_group:** Specifies the group as whom the command should be run.
  + **command(s):** Specifies the command or commands allowed by the rule.
* Here is a simple example:
  + **john ALL=(ALL:ALL) ALL**
* In this example:
  + **john** is the username.
  + **ALL** means any host is allowed.
  + **(ALL:ALL)** specifies that the user can run commands as any user and any group.
  + **ALL** at the end allows running any command.
* Remember that incorrect changes to the sudoers file can result in system access issues. Always use the visudo command to edit the file and verify the syntax before saving.

It's also important to note that some systems may have additional configurations or use sudo configuration files in the /etc/sudoers.d/ directory to modularize configurations and make management more manageable.

**userdel**

To delete a user in Linux, you can use the userdel command. Deleting a user typically involves removing the user's entry from the /etc/passwd and /etc/shadow files. Optionally, you may also want to remove the user's home directory and mail spool.

* Here's the basic syntax for the userdel command:
* **sudo userdel [options] username**
  + Here are a few common options:
  + **-r:** Remove the user's home directory and its contents.
  + Here's an example:
* **sudo userdel -r username**
  + Replace username with the actual username of the account you want to delete.
  + If you don't use the -r option, the user's home directory and mail spool will not be removed. It's important to be cautious when using the -r option, especially if the user has important data in their home directory.
  + Additionally, if the user is a member of any additional groups, you may want to remove the user from those groups using the gpasswd command or usermod -G before deleting the user.
* Example:
* **sudo gpasswd -d username groupname**
  + Replace username with the actual username and groupname with the name of the group.

Always be careful when deleting user accounts, especially if they have important data associated with their home directory. Double-check that you have the correct username and that you won't accidentally delete critical files or affect other users.

**kill**

command in Linux is used to send signals to processes. It is commonly used to terminate or signal processes to perform certain actions. The basic syntax of the kill command is as follows:

* **kill [options] <PID>**
* Here, PID is the process ID of the target process. You can find the process ID using commands like ps or pgrep.
* Commonly Used Signals:
  + SIGTERM (15): Default signal, asking the process to terminate gracefully.
  + **kill -15 <PID>**
* SIGKILL (9): Forcibly kills the process without giving it a chance to clean up.
  + **kill -9 <PID>**
* SIGHUP (1): Hang up. It's often used to restart daemons or reload configuration files.
  + **kill -1 <PID>**
* Examples:
* **Terminate a Process Gracefully:**
  + kill -15 <PID>
  + Replace <PID> with the actual process ID.
* **Forcibly Terminate a Process:**
  + kill -9 <PID>
* **Restart a Daemon:**
  + kill -1 <PID>
  + This is often used to ask a daemon to re-read its configuration files.
* **Kill All Processes Matching a Name:**
  + pkill process\_name
  + This sends the default SIGTERM signal. Use -9 for SIGKILL.

Remember, sending a SIGKILL (kill -9) should be used cautiously, as it doesn't allow the process to perform cleanup actions before termination. It forcefully terminates the process, and data loss or corruption might occur.

**Cron Job**

Cron is a time-based job scheduler in Unix-like operating systems. It allows you to schedule tasks (cron jobs) to run periodically at fixed times, dates, or intervals. Here's a brief overview of how to use cron jobs in Linux:

* **Viewing and Editing Your Cron Jobs:**
  + To view and edit your existing cron jobs, you can use the following commands:
* **View your existing cron jobs:**
  + crontab -l
* **Edit your cron jobs:**
  + crontab -e
  + This opens the default text editor (usually vi or nano) for editing your cron jobs.
* **Syntax of a Cron Job:**
  + A cron job has the following syntax:
  + minute hour day month day\_of\_week command\_to\_execute
  + minute: 0-59
  + hour: 0-23
  + day: 1-31
  + month: 1-12
  + day\_of\_week: 0-6 (Sunday to Saturday)
  + command\_to\_execute: The command or script to run.
* **Examples of Common Cron Expressions:**
* **Every day at 3:30 AM:**
  + 30 3 \* \* \* command\_to\_execute
* **Every Monday at 2:00 AM:**
  + 0 2 \* \* 1 command\_to\_execute
* **Every hour:**
  + 0 \* \* \* \* command\_to\_execute
* **Every 15 minutes:**
  + \*/15 \* \* \* \* command\_to\_execute
* **Useful Tips:**
* **Redirecting Output:**
  + Redirect the output to a file to capture any errors or logs:
  + 0 2 \* \* \* command\_to\_execute >> /path/to/logfile.log 2>&1
  + 2>&1: This part redirects the standard error (stderr) to the same location as the standard output. In Unix-like systems, file descriptor 1 (&1) represents stdout, and file descriptor 2 (&2) represents stderr. So, 2>&1 essentially means "redirect stderr to the same location as stdout."
* **Editing Existing Cron Jobs:**
  + When you use crontab -e, it opens the existing crontab file for editing. Make your changes, save, and exit to update your cron jobs.
* **Removing All Cron Jobs:**
  + To remove all your existing cron jobs:
  + crontab -r
* **Special Strings:**
  + Cron jobs also support special strings for certain fields:
  + **@reboot:** Run once at startup.
  + **@daily** or **@midnight:** Run once a day at midnight.
  + **@weekly:** Run once a week at midnight on Sunday.
  + **@monthly:** Run once a month at midnight on the first day of the month.
  + **@yearly** or **@annually:** Run once a year at midnight on January 1.
* For example:
  + @daily command\_to\_execute
  + This will run command\_to\_execute once a day at midnight.
* Remember that cron jobs run with the environment of the user who owns the cron tab, so make sure the necessary environment variables are set if needed for your scripts or commands.

If your cron job didn’t work, how would you check?

1. Check system time, (system time zone to cronjob time zone similarity.)
2. Crontab entry,
3. Check /var/log/messages

**bin**

In Linux and other Unix-like operating systems, the "bin" directory is a fundamental directory within the file system hierarchy. "Bin" stands for "binary," and this directory typically contains executable binaries (compiled programs or scripts) that are essential for the system's functioning or for users to perform basic tasks.

* Here are some key points about the "bin" directory:
* **Location:** The "bin" directory is usually found at the top level of the file system, often in the root directory ("/bin"). Other common locations include "/usr/bin" and "/sbin."
* **System Binaries:** The "/bin" directory contains essential system binaries and commands that are required for basic system operations, even in single-user mode.
* **User Binaries:** The "/usr/bin" directory contains user-related binaries and applications. This is where most of the executable programs installed by the system or the user are located.
* **Superuser Binaries:** The "/sbin" directory contains system binaries that are typically intended for use by the superuser (root) for system administration tasks. These binaries may require elevated privileges.
* Here's a brief breakdown:
  + **/bin:** Essential system binaries.
  + **/usr/bin:** User binaries and applications.
  + **/sbin:** System binaries for system administration (superuser).
* Examples of binaries you might find in these directories include basic shell commands (like ls, cp, mv), system utilities, and administration tools.

**Docker**

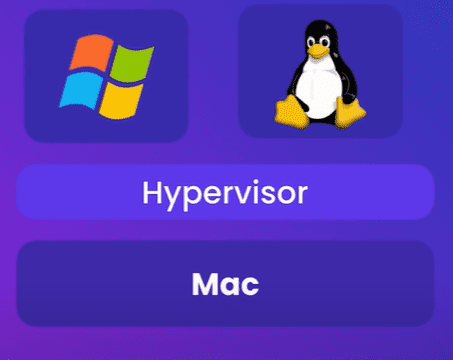
* is a platform for building, running and shipping applications.
* Reasons why application that runs on your system doesn’t run on another system:
  + One or more files missing
  + Software version mismatch
    - If your machine runs version node 14 but target machine runs node version 13.
  + Different configuration settings
* With docker we can easily package our application with everything it need and run it on any machine with docker.
* Docker helps us in consistently build, run and ship applications.

**Virtual machine vs Container**

**Virtual machine:**

****

Ex: of an virtualization



* Hypervisor is software we use to create and manage virtual machines.



What is the benefit of VMs?

For us software developers we can run applications in isolation inside a virtual machine (VM)

On the same physical machine we can have two different virtual machines vm1 and vm2 each running completely different application.

Problems of virtualization:

* Each VM needs a full-blown OS.
  + Full copy of an os, need to be licensed, patched and monitored.
* Slow to start
  + Entire os has to be loaded just like starting your computer.
* Resource intensive
  + Because each VM takes out the slice of actual physical hardware resources like CPU, Memory and disk space.
  + Suppose if you have 8GB of memory that memory has to be divided between different VMs.

**Containers:**

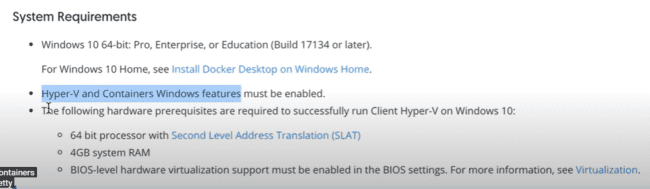
* Allow running multiple apps in isolation.
* Are lightweight
  + They don’t need a full os.
* Use OS of the host.
  + All containers on a single machine share the os of the host that means we need to license, patch and monitor a single OS.
* Start quickly
  + Because the os has already started on the host a container can start up pretty quickly usually in a second sometime less.
* Need less hardware resources.

**Docker Architecture**

* Docker uses client server architecture.
* Client component talks to server component using a REST API.
* Server also called docker engine sits on the background and takes care of building and running docker containers.
* But technically a container is just a process like other processes running on your computer but it’s a special kind of process.
* Containers doesn’t share full-blown os instead all containers on a host share the os of the host now more accurately all these containers share the kernel of the host.
* A kernel is the core of an os it like the engine of a car.
* Kernel is the part that manage all applications and hardware resources like memory and cpu.
* Every os has its own kernel or engine and these kernels have different api’s that’s why we cannot run a windows application on linux because under the hood this application needs to talk to the kernel of the underlying os.
* That means on a linux machine we can only run linux containers because these containers need linux on a windows machine however we can run both windows and linux containers because windows 10 is now shipped with a custom built linux kernel.
* This is in addition to the windows kernel that’s always been in windows it’s not a replacement.
* So with this Linux kernel now we can run Linux applications natively on windows so on windows we can run both Linux and windows containers.
* Our windows containers share the windows kernel and our linux containers share the linux kernel.
* Mac os has its own kernel which is different from linux and windows kernels and this kernel does not have native support for continuous applications.
* So docker on mac uses a lightweight linux virtual machine to run linux containers.

**Installing Docker**

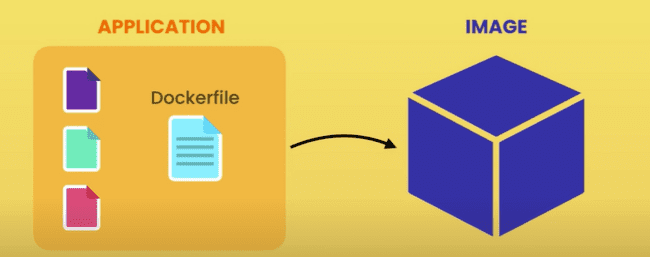




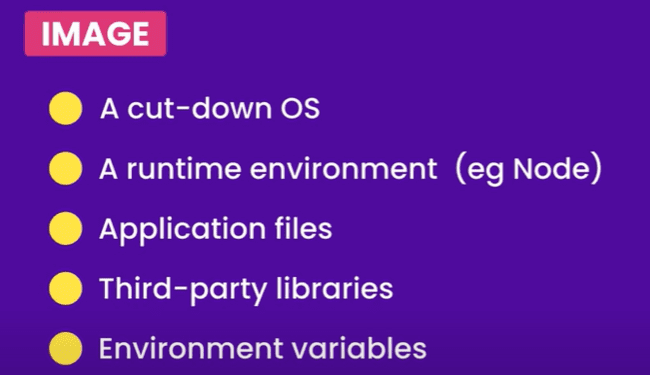


**Docker Workflow**

* We take an application it doesn’t matter what kind of application it is or how it’s built we take that application and dockerize it.
* Which means we make a small change so that it can be run by docker.
* We just add a docker file to it, a docker file is a plain text file that includes instructions that docker uses to package up this application into an image.

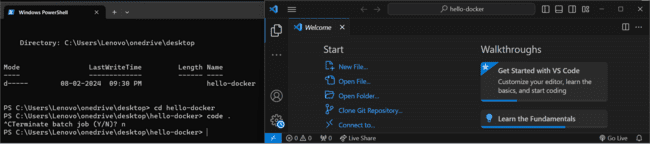


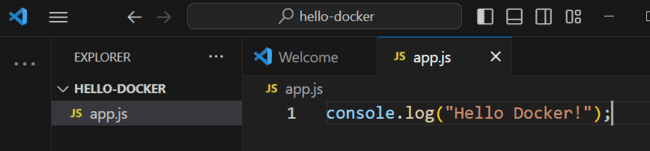
* This image contains everything our application needs to run.
* Typically a cut down os, a runtime environment like node or python, it also contains application files, third-party libraries, environment variables.



* We create a docker file and give it to docker for packaging our application into an image.
* Once we have an image we tell docker to start a container using that image.
* So a container is just a process but it’s a special kind of process because it has it’s own file system which is provided by the image.
* So our application gets loaded inside a container or a process and this is how we run our application locally on our development machine.
* So instead of directly launching the application and running it inside a typical process we tell docker to run it inside a container an isolated environment.
* Once we have this image we can push it to a docker registry like docker hub.
* Docker hub to docker is like github to git it’s a storage for docker images that anyone can use.
* So once our application image is on docker hub then we can put it on any machine running docker.
* This machine has the same image we have on our development machine which contains a specific version of our application with everything it needs.
* so we can start our application the same way we started on our development machine we just tell docker to start a container using this image.
* so with docker we no longer need to maintain long complex release documents that have to be precisely followed’
* All the instructions for building an image of an application are written in a docker file with that we can package up our application into an image and run it virtually anywhere.

**Docker in Action**

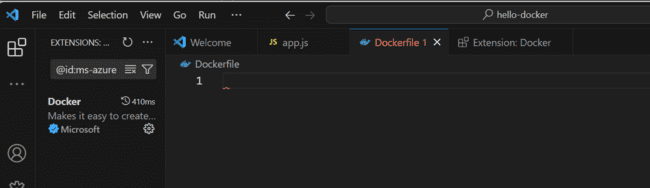




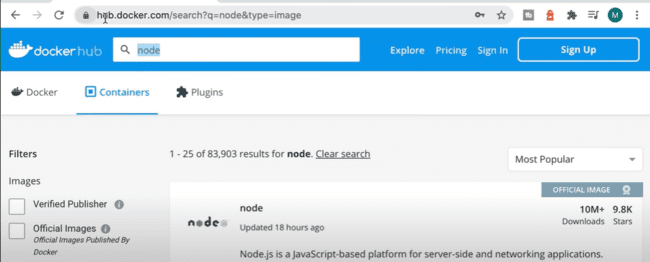
* To just run a javascript hello world program without docker we need below all steps to be followed:



* What if we need to run a complex application imagine the no.of instructions we need to follow in such case.
* This is where docker comes to rescue.
* We can write these instructions inside a docker file and let docker package up our application.
* Back to vs code we’re going to add another file to this project called docker file so capital d and all the other letters are lowercase and this file doesn’t have any extensions.
* Now vs code asks if we want to install the recommended extensions for docker we can go ahead with that.



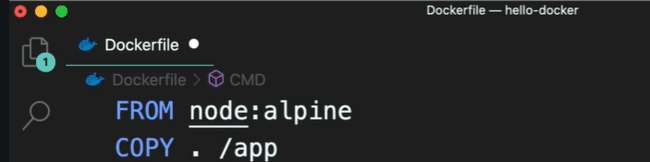
* Back to this docker file here we write instructions for packaging our application.
* So typically we start from a base image this base image has a bunch of files.
* We’re going to take those files and add additional files to it this is kind of like inheritance in programming.
* So what is a base image?
* We can start from a linux image and then install node on top of it or we can start from a node image this image is already built on top of linux.
* How do I know these names?
* These images are officially published on docker hub.



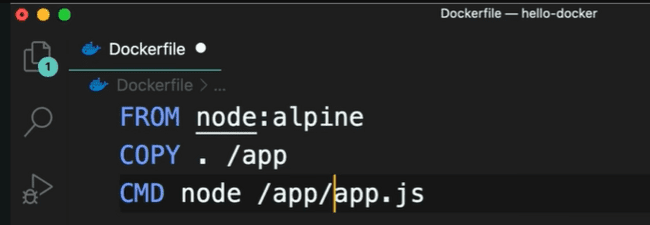
* So docker hub is a registry for docker images.
* Now back to our docker file.
* We start from a node image.
* Now if you look at docker hub you will see that there are multiple node images these node images are built on top of different distributions of linux.
* So linux has different distributions or different flavors used for different purposes.
* Now here we can specify a tag using a colon to specify which linux distribution we want to use.
* For this we are going to use alpine which is a very small linux distribution so the size of the image that we’re going to download and build on top of is going to be very small.



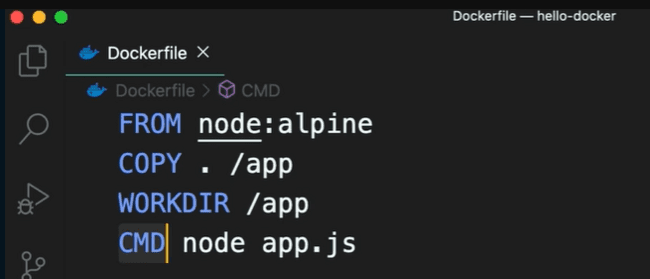
* So we start form that image then we need to copy our application or program files for that we use the copy instruction or copy command.
* We’re going to copy all the files in the current directory into the app directory into that image.
* So that image has a file system and in that file system we’re going to create a directory called app.



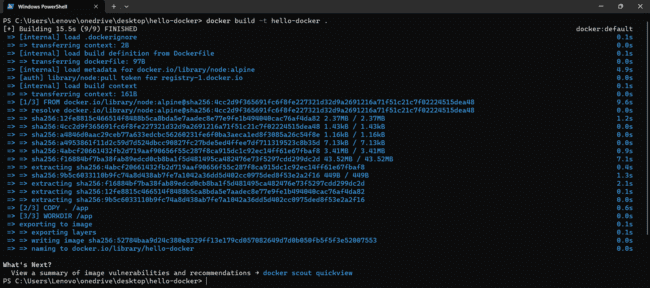
* Now finally we’re going to use the command instruction to execute a command.
* What command we execute here ? node app.js
* But this file is inside the app directory so we have to prefix it with the directory name.



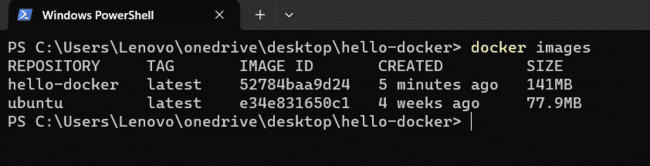
* Alternatively, here we could set the current working directory workdir to /app and then we don’t need to prefix this with the directory name.
* So when we use this instruction all the following instructions assume that we’re currently inside the app directory.
* These instructions clearly document our deployment process.



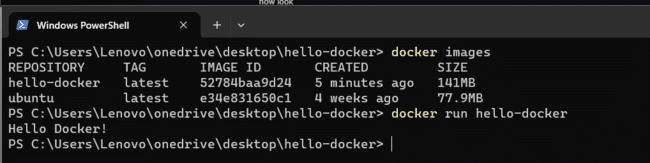
* Now we go to the terminal and tell docker to package up our application.
* We say **docker build**.
* We need to give our image a tag, a tag to identify.
* So -t here we specify a name like “hello docker” and then we need to specify where docker can find a docker file.
* So we’re currently inside hello docker directory and our docker file is right here so we use a period to reference the current directory.



* But back in vs code there is nothing here.
* Because the image is not stored here and in fact an image is not a single file.
* How docker stores this image is very complex and we don’t have to worry about it.
* Back to the terminal to see all the images on this computer we type “docker images”.

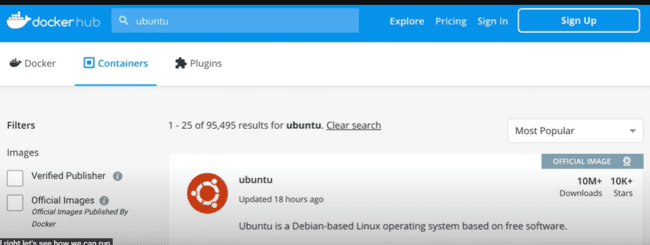


* On this machine we have a repository called hello-docker.
* In this repository we have an image with this tag latest.
* So docker added this by default.
* Basically we use these tags for versioning our images.
* So each image can contain a different version of our application.
* Each image also has a unique identifier.
* Here we can see when the image was created and the size of this image so because we used node from linux alpine we ended up with 141 megabytes of data in this image.
* So this image contains alpine linux node and our application files and the total size is 141 megabytes.
* Now if we used a different node image that was based on a different distribution of linux we would end up with a larger image and when deploying that image we would have to transfer that image from one computer to another.
* So that’s why we use node alpine because this a very small image.
* So we have built this image now we can run this image on any computer running docker.
* So on this machine we can say “docker run” and then type the image name “hello-docker”.
* And it doesn’t matter which directory you’re in because this image contains all the files for running our application.

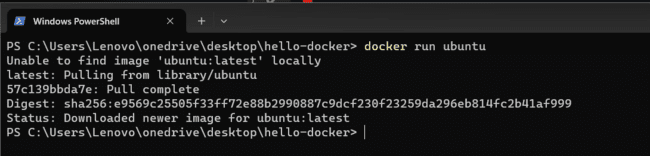


* We can see the message on the terminal.
* Now you can go ahead and publish this image to docker hub so anyone can use this image.
* Then I can go on another machine like a test or a production machine and pull and run this image.
* So we can take any application and dockerize it by adding a docker file to it.
* This docker file contains instructions for packaging an application into an image once we have an image we can run it virtually anywhere on any machine with docker.

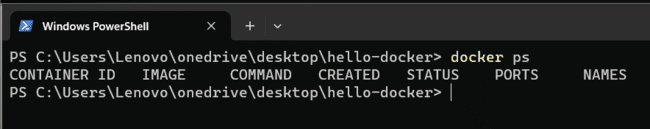
**Running Linux**

****

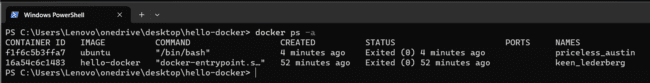
* Let’s see how we can run ubuntu on this machine.
* We go to hub.docker.com and search for ubuntu over there you can see the official ubuntu image that’s been downloaded more than 10 million times.
* So for each image you can see the command to pull that image onto your machine.
* Here we instead of “docker pull ubuntu” we are going to use a shortcut called “docker run ubuntu”.
* Now if we have this image locally docker is going to start a container with this image otherwise it is going to pull this image behind the scene and then start a container.



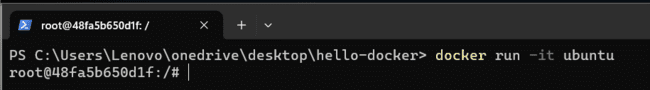
* Docker started a container but because we didn’t interact with this container the container stopped.
* Lets prove this:
* Now when we use the command “docker ps” we can see the running processes or the running containers.



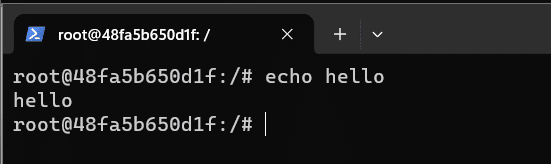
* Look we don’t have any containers running here but if we type “docker ps -a” we can see the stopped containers as well.



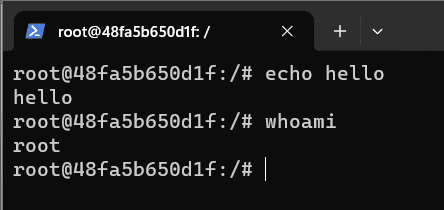
* To delete images use command “docker rmi <image\_id>” to forcefully delete images even containers linked to it use “docker rmi -f <image\_id>”.
* To delete container use command “docker rm <container\_id>”.
* So we have two stopped containers first one is using the ubuntu image this is the one that we just started.
* The second one is hello-docker that we started earlier.
* So start a container and interact with it we have to type “docker run -it (short for interactive) ubuntu”.
* We’re going to start a container in the interactive mode and in this container we’re going to load the ubuntu image which we have locally.



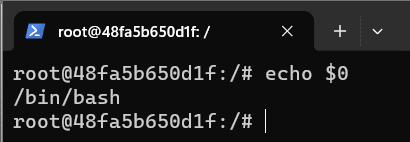
* Now, what we have here is called the shell.
* A shell is a program that takes our commands and passes them to the operating system for execution.
* What we have here is called the shell prompt.
* The first part “root” represents the currently logged in user.
* So by default I’m logged in as the root user which has the highest privileges.
* And then after the @ sign we have the name of the machine(48fa5b650d1f).
* So this container has this id which is automatically generated by docker and in this case it’s like the name of a machine.
* And after the colon you can see forward slash that represents where we are in the file system a forward slash represents the root directory that is the highest directory in the file system.
* Then we have a pound and this means I have the highest privileges because I’ve logged in as the root user.
* If I logged in as a normal user instead of a pound we would see a dollar sign.
* So in this shell we can execute a bunch of commands for ex we can say “echo hello” and this prints hello on the terminal.



* We can also say “whoami” this show the current user.



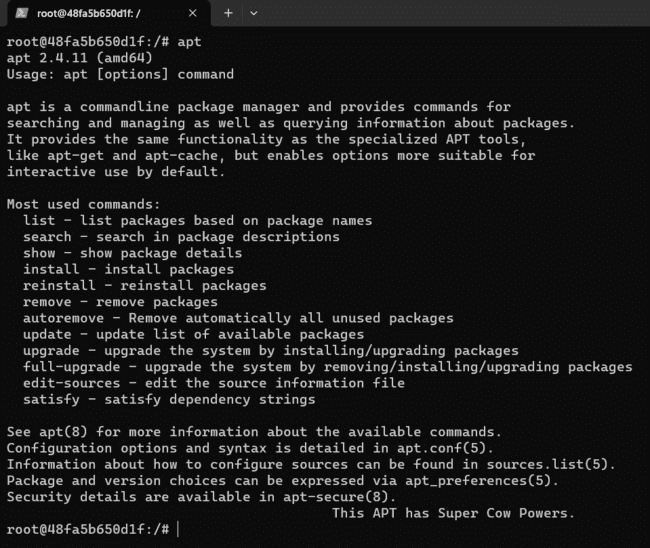
* This shell program takes these commands and passes them to the kernel for execution.
* If we type echo dollar sign 0 we can see the location of this shell program.

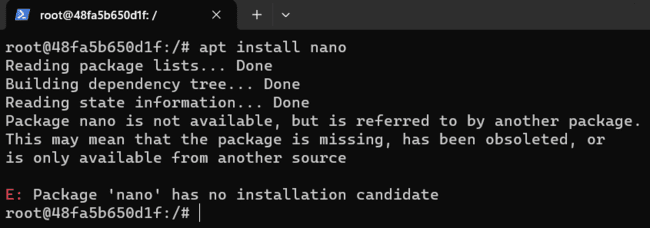


* /bin/bash so bin is a folder or a directory and inside this directory we have a program called bash which is short for born again shell.
* So apparently steve bourne is the first person who created a shell program bash or born-again shell is a reference to steve bourne.
* Bash is an enhanced version of the original shell program.
* In linux we use a forward slash to separate files and directories but in windows we use a backslash so that’s one of the first differences.
* The other difference is that linux is a case sensitive os so if you type echo with capital e i.e, “Echo $0” it’s not going to work.
* Bash tells us echo command not found so lowercase and uppercase letters are different and it’s not limited to commands it’s applicable everywhere.
* If you want to reference a file or a directory or a user pretty much anything we should always spell it properly with the right uppercase and lowercase letters.

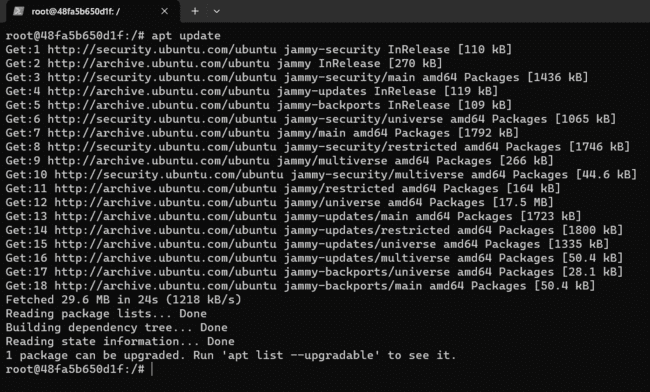
**Managing Packages**

* These days most operating systems and development platforms come with a package manager you’ve probably worked with tools like npm, yarn, pip and so on….
* Here in ubuntu we also have a package manager called apt which is short for advanced package tool.

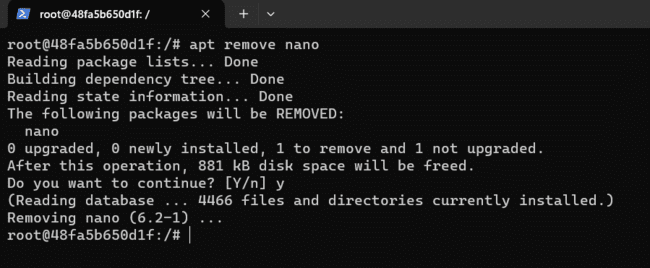




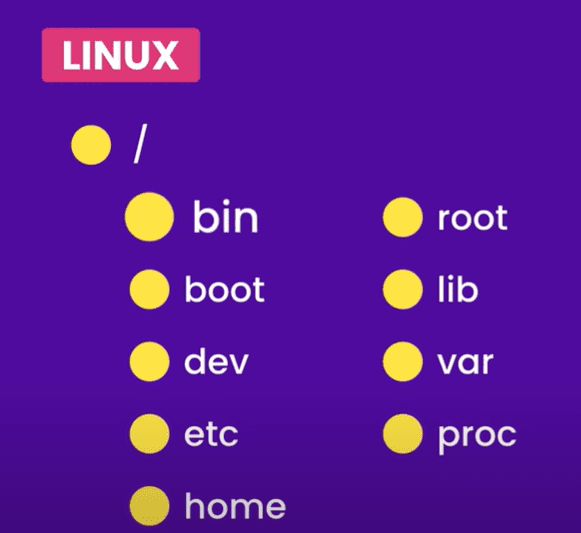
* “apt update” is used to update the apt database packages.
* This command is used because there is no nano package present in apt database we update the apt database using update command and now try to install the nano command after the package is installed in apt database.
* To check the list of installed package we can use “apt list” command if the package is installed we can see a installed text beside the package if not we need to install that package.







**Linux Filesystems**



bin = binaries or programs

boot = all the files related to booting

dev = devices

in linux everything is a file including devices, directories, network sockets, pipes and so on.

So the files that are needed to access devices are stored in this directory.

etc = there are different options usually known as editable text configuration.

So this is where we have configuration files.

home = home directories for users are stored.

So on a machine with multiple users each user is going to have a home directory here.

root = is the home directory of the root user.

Only root user can access this directory.

lib = for keeping library files like software library dependencies

var = which is short for variable.

this is where we have files that are updated frequently like log files, application data and so on..

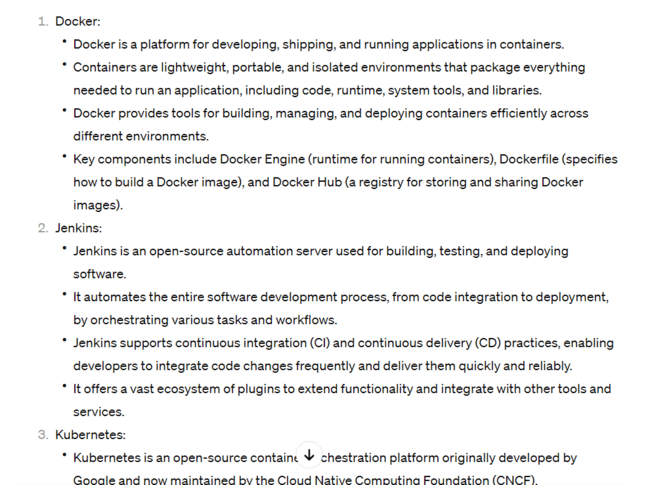
proc = which includes files that represent running processes

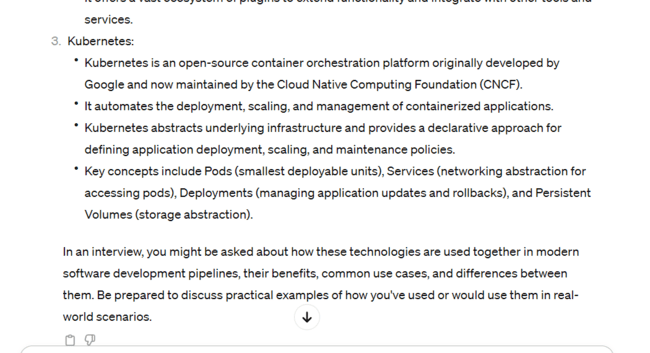
in linux everything is a file like processes, devices even directories are files.

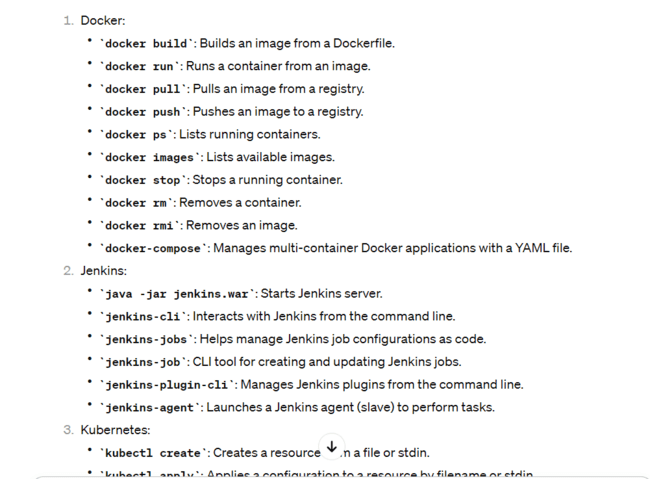
**Docker Compose**

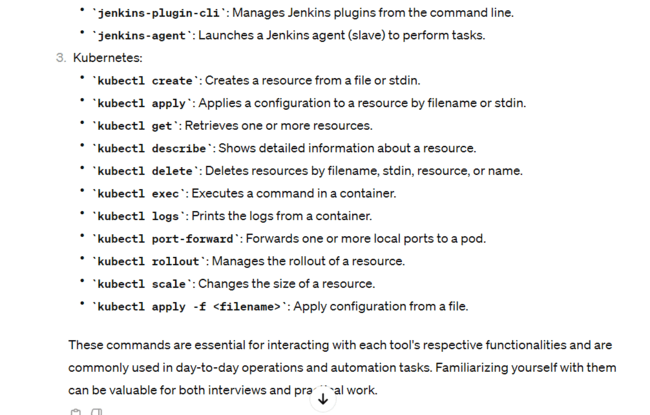
**Running Multi-container Apps**

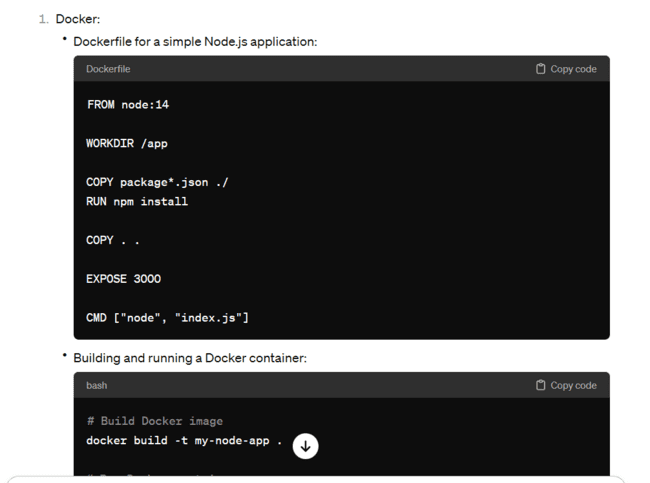
**Docker**

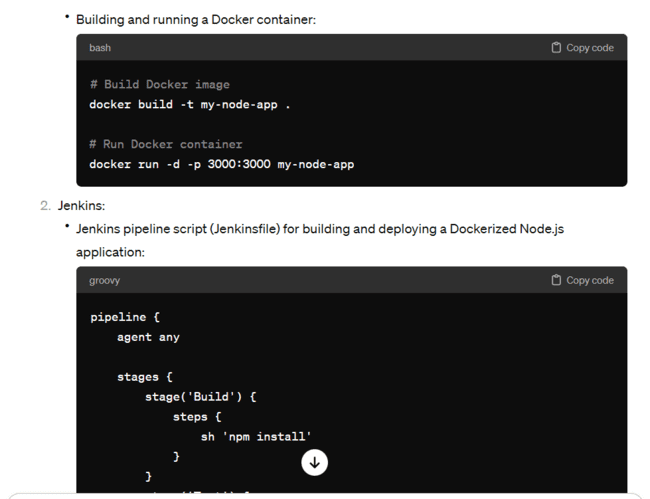


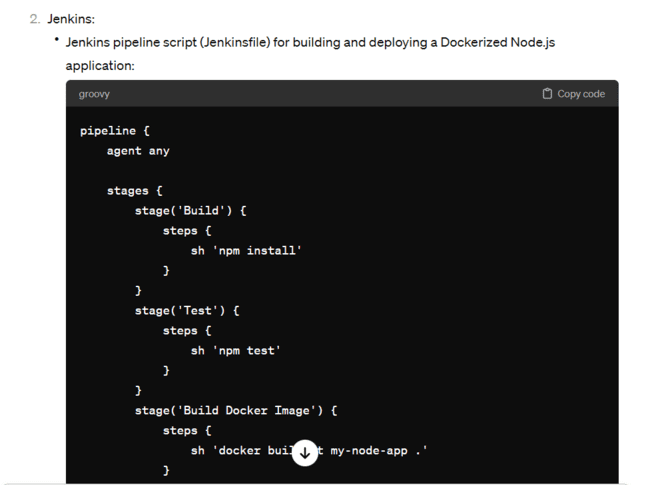


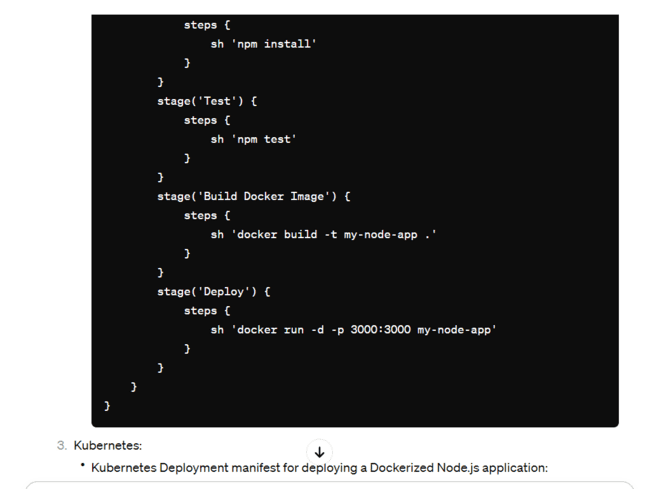


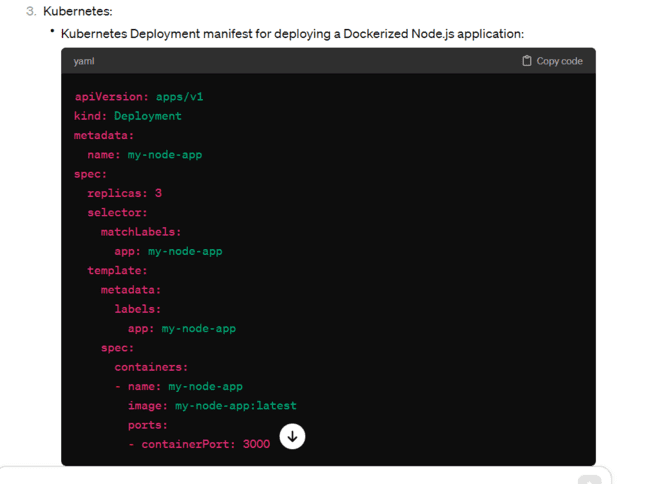


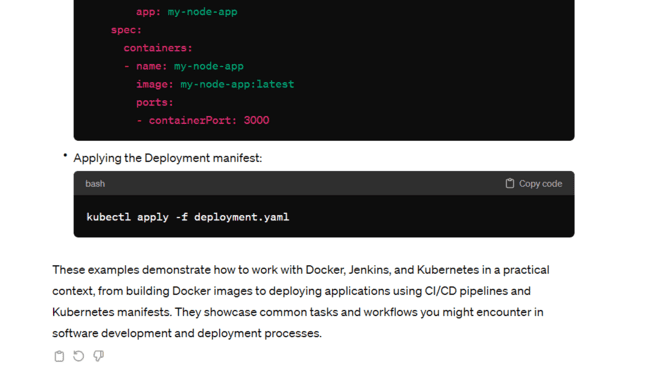








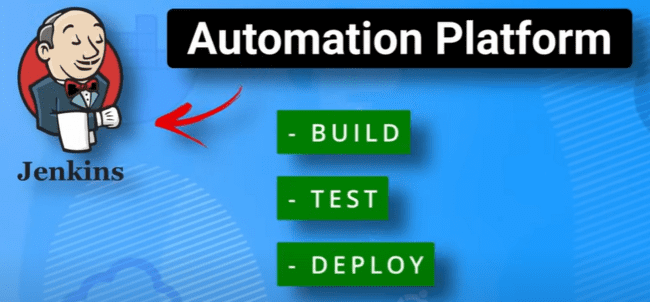




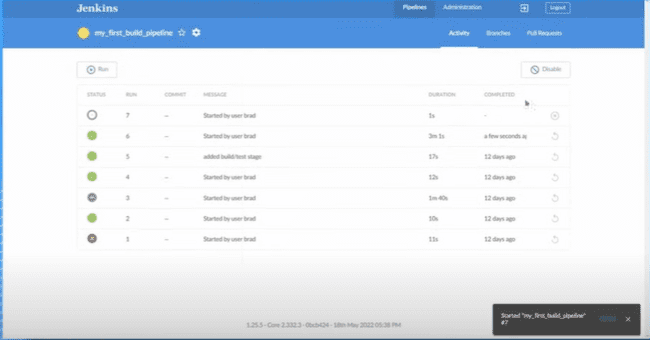
**Jenkins**

****

* Jenkins is an automation platform that allows you to build, test and deploy software using pipelines.



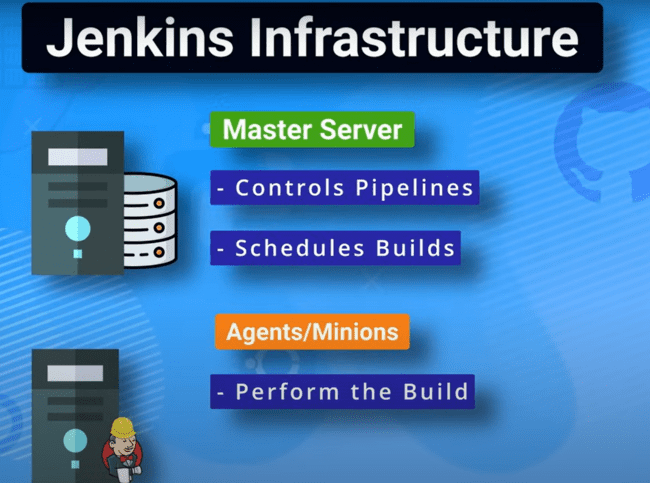
* But it’s not just limited to creating pipelines for code it can be used to automate any task.
* Jenkins provides a web gui where you can create jobs and customize all the functionality that you want.
* Such as source control management, pre and post build actions, as well as build triggers.



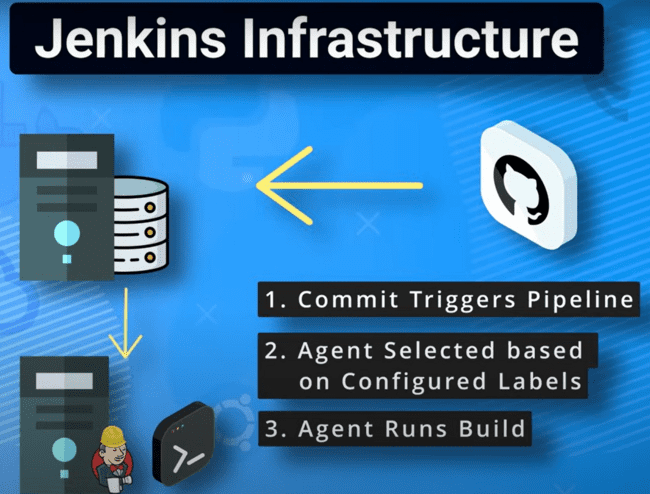
* This allows you to run tasks on demand by clicking a button or have them triggered automatically via web hooks.

**Jenkins Infrastructure**

* First you have the master server which controls the pipelines and schedules builds to agents.
* And then you have the agents which run the build in their workspace



* An example workflow may look like this:
* A developer commits some code to a git repository the Jenkins master becomes aware of this commit and triggers the appropriate pipeline and distributes the build to one of the agents to run.
* It selects the appropriate agent based on labels which is something that you can configure through the Jenkins ui.
* The agent then runs the build which is usually just a bunch of linux commands to build, test and distribute your code.



* There are two main categories of Jenkins agents:
* Permanent node Agents
  + You can think of these just as standalone linux or windows servers that are configured to run Jenkins jobs.
  + And these are just basic every day servers the only real setup of them is you need to have java installed and you need to make sure that ssh is set up.
  + As the master server usually makes connections over ssh.
  + You will also want to make sure that you have any type of build tools that you want to use on these servers installed as the agents are going to be the workhorses that run your actual builds.
  + 
* There is another type of agent known as **cloud agents.**
* Which are a much more popular choice in real world environments.
* Some examples of cloud agents are docker, Kubernetes and aws fleet manager.
* In these scenarios Jenkins can dynamically spin up agents based on the agent templates you configure.



* Two main build jobs that you’re going to run into with Jenkins.
* These are the freestyle build projects and pipelines.
* First is the freestyle build projects which is the simplest way to get started with Jenkins.
* Basically just think of them as shell scripts that will be run on a server that can be triggered by specific events like a developer making a commit to a github repository.
* Freestyle projects are easy because you can use the ui and plugins to manage just about anything in the build.
* Most freestyle projects will be set up to execute a shell script so as long as you are familiar with shell scripting or the linux command line you should be good.
* The next style is pipelines.
* Pipelines use Jenkins files written in the groovy syntax to specify what happens during the build.
* Pipelines are commonly broken into different stages as you can see this pipeline is broken into five different stages – the clone, build, test, package and deploy stages.



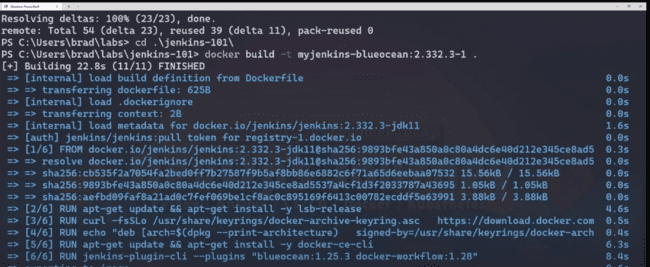
* These stages can differ depending on the project but this is a common workflow that I usually see.
* The clone stage is usually made up of pulling down the code from a git repository and setting up the local environment on the agent.
* The build stage takes the code and builds it which usually means generating some type of local artifact on the code like a jar file executable or container image.
* The test stage runs tests against the newly built code and in the package stage it gets packaged up so it is ready for deployment.
* In the deploy stage is when I usually send out artifacts to my registry so a good example would be sending out a newly built docker image to docker hub.
* Therefore, Jenkins is basically just a way to automate the work that developers don’t want to do so they can save their time and do more productive things.

**Working on Jenkins using docker container**

**GitHub link for code:** [**https://github.com/devopsjourney1/jenkins-101**](https://github.com/devopsjourney1/jenkins-101)



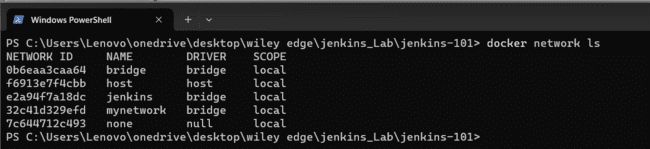
* Now building the image:
* Grabbing the docker image of Jenkins
* And then running apt update install and installing a couple of plugins.



* Now docker network create Jenkins



* To check and verify if the network is created

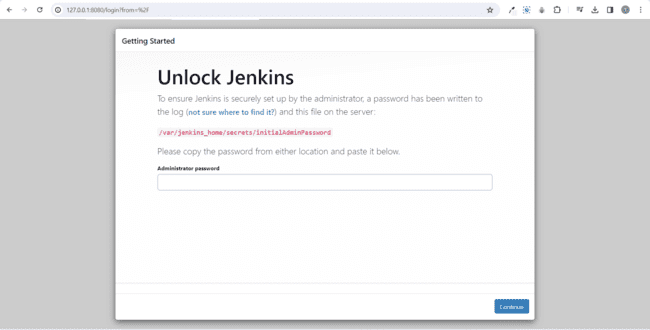


* You can see we have this Jenkins network already created.
* Now running the container command





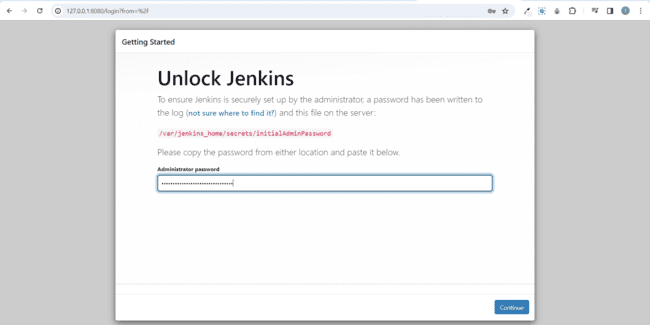
* Now we can see docker container is running and it’s listening on port 8080 so let’s go ahead pull up a browser and see if we can get to this website.

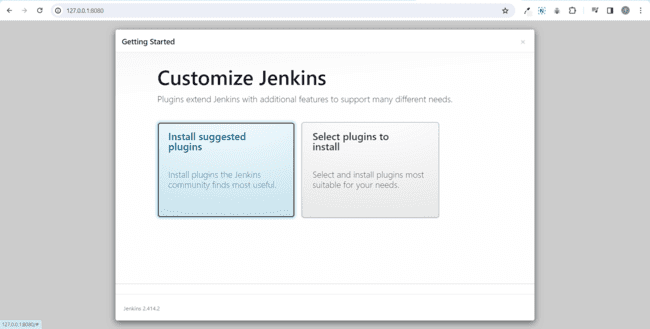


* Password is going to be at this (/var/Jenkins\_home/secrets/initialAdminPassword) location when you first install Jenkins.
* Since we are running this on a docker container we’re going to have to use the docker exact command to cat out this file.

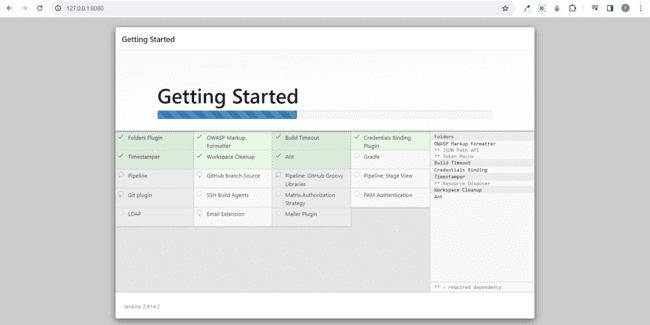


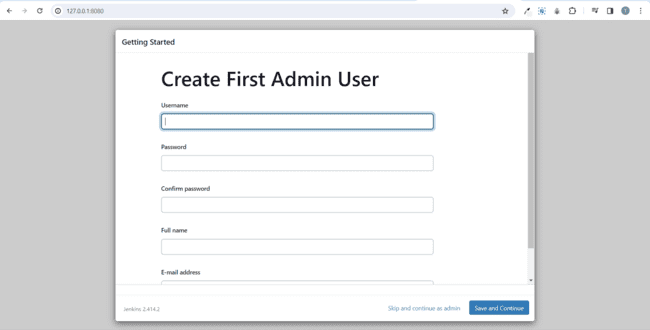
* Grad the password put it in



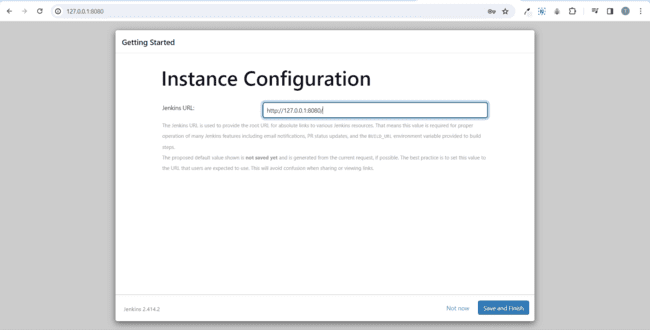


* And we’re in.

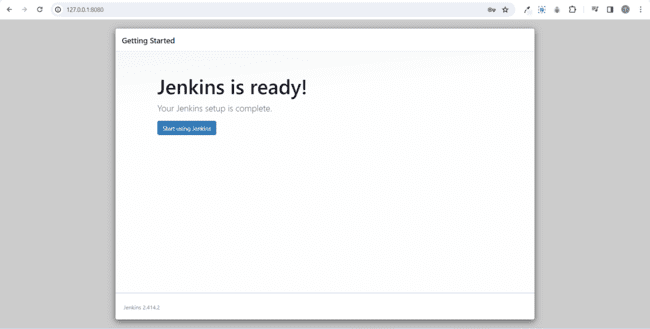




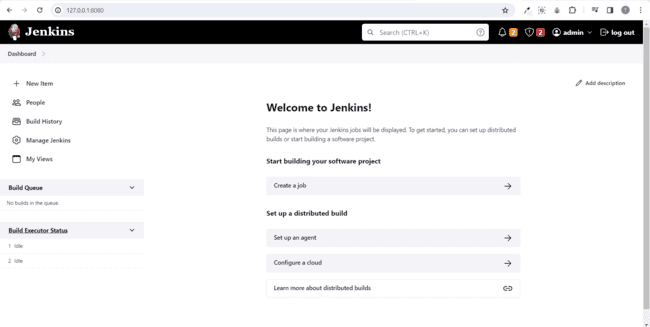
* Enter the details and create the account.



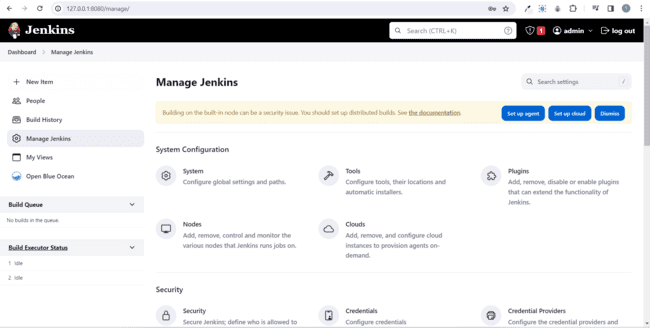
* Leave the instance configuration same.



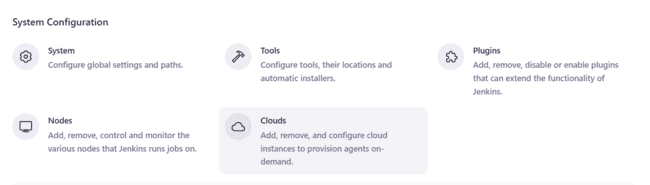
* Now everything is done click on start using jenkins.



* Click on manage Jenkins.

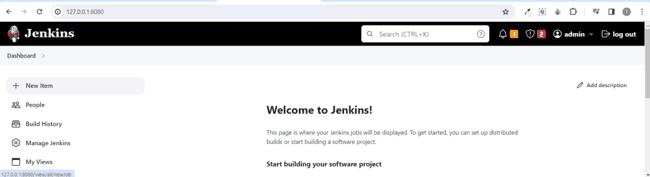


* These are all the options use for managing Jenkins.
* At the very top we can see there’s some notifications.
* It is saying that building on the built in node can be a security risk.
* This is letting us know that we don’t have any agents set up so if we were to make a Jenkins pipeline it would be running on the Jenkins master which is not recommended.

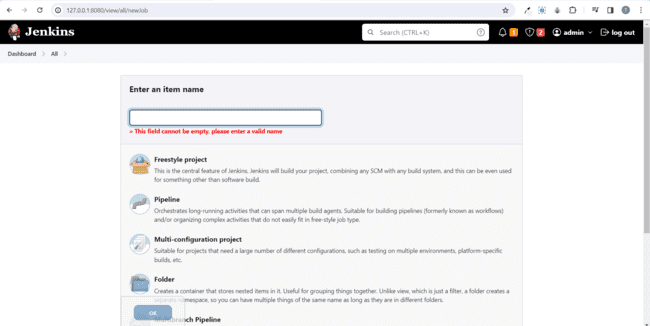


* **Configure system** is going to give you a lot of information about your Jenkins server and let you set some global parameters in regards to your Jenkins server.
* **Manage plugins** as a Jenkins administrator you’re going to be spending a lot of time in here because Jenkins is a beast when it comes to plugins and over time companies install plugins, developers install plugins and different plugins break each other when you upgrade Jenkins plugins break.
* so being a Jenkins administrator you’re going to dealing with plugins a lot.
* **Manage nodes and clouds** this is where you’re going to go ahead and set up your agents as well as clouds like Kubernetes, docker, aws anything to do with agents will be in here.
* The next section is the security section.
* All are mostly familiar but manage credentials is where you would store ssh keys or api tokens now you don’t have to use Jenkins as your credential manager you could use something like aws secrets if you want.
* But Jenkins does have a built-in credential manager so ahead and use that if you would like.
* Remaining sections all are mostly smiliar but the one Prepare for Shutdown you would use this in a situation where you need to take the Jenkins server offline to perform some sort of maintenance you wouldn’t want to just go ahead and reboot the Jenkins server or shut it down because you’d be interrupting current jobs that are running so what you’d want to do is set to prepare for shutdown and what that will do is make it so Jenkins doesn’t start any new jobs it just finishes the jobs that are currently running.

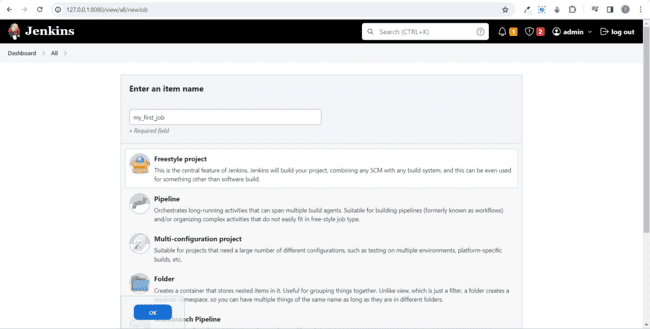
**Creating a Simple Freestyle Job**

****

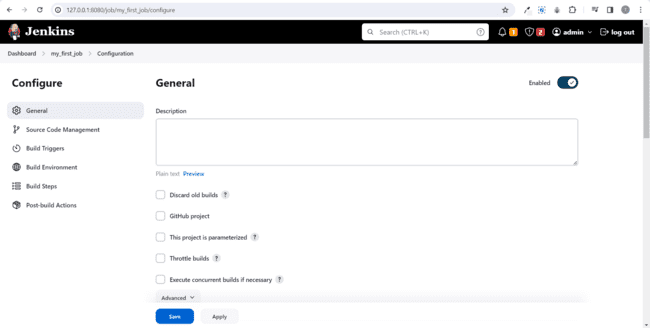
* Click on new item
* If you’re administrating Jenkins servers on the job doing any type of consulting you’re going to run into freestyle projects.

****

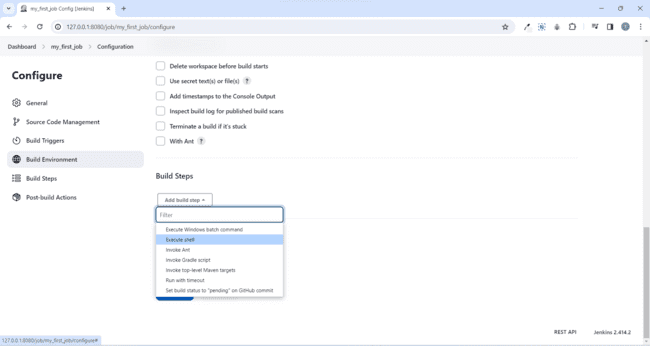
* We will have to give it a name now the first thing is do not create a name that has spaces in it.
* And the reason for this is because in the back-end Jenkins is actually going to be creating a folder based on this job.
* You can use underscores or dashes.

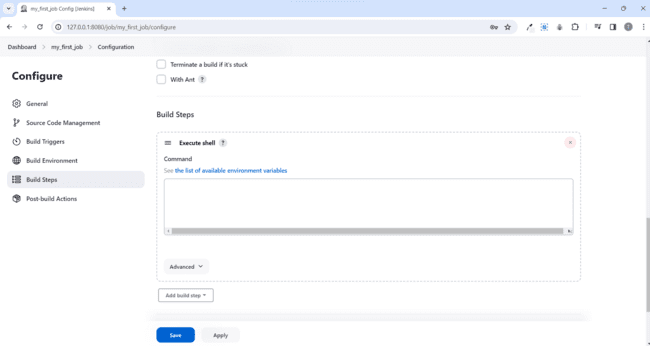
****

* Give it a name and don’t forget to select “Freestyle project” and click on ok.

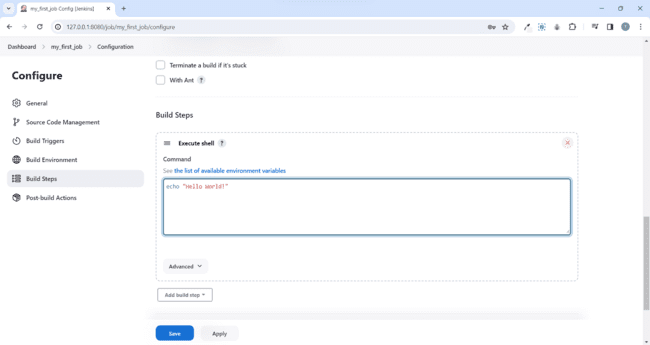
****

* Now we are in the project settings of our first build that we’re creating here
* In this you are almost always going to be using something like get and then you just put in your repository url.
* Jenkins is going to go ahead and pull that repository down into your workspace and use that code.
* The next one is your build triggers so your build triggers is what is triggering your job so popular options here is to have github send a web hook to your Jenkins server.
* This could be little more difficult if your Jenkins server is behind a firewall.
* You will need to either open ports or use some sort of proxy service to make sure that web hooks get to your Jenkins server.
* Another way to sort of cheat the system is to use this polling scm and this is going to have Jenkins reach out to github and just check periodically so that’s one way to get around if your Jenkins is behind a firewall and cannot receive web hooks.
* Another option is to just build periodically and this is just the same thing as setting up a cron job you just set a schedule and your Jenkins build will run based on that schedule.
* For build environments a popular option is to delete the workspace before the build starts this just makes sure that you have a clean environment before your build runs.
* And it deletes any artifacts that have been left over by Jenkins that were created during the last run of the job.
* Post build actions happen when your build is complete.
* Popular options here might be to send an email notification.
* Now go to build and you are going to use is “Execute shell”.

****

****

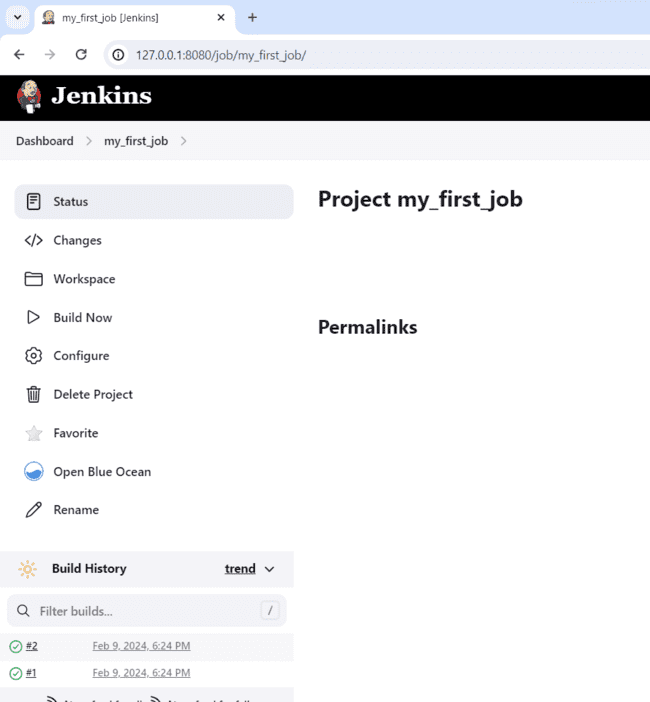
* Execute shell is just like a bash scripting so whatever we run here the Jenkins server is going to run within its workspace.
* Just write “echo ‘hello world’ “ and hit save.

****

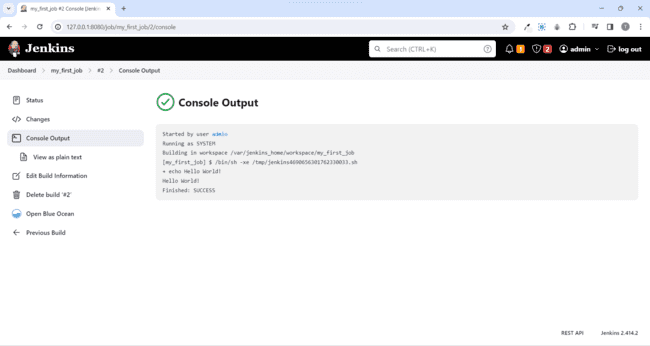
* After the save button click on build now.

****

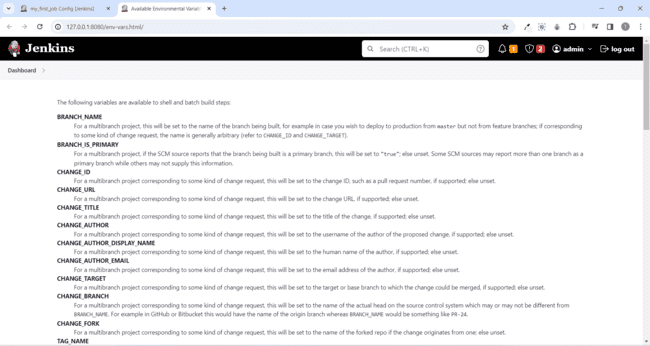
* And this is going to run our job.

****

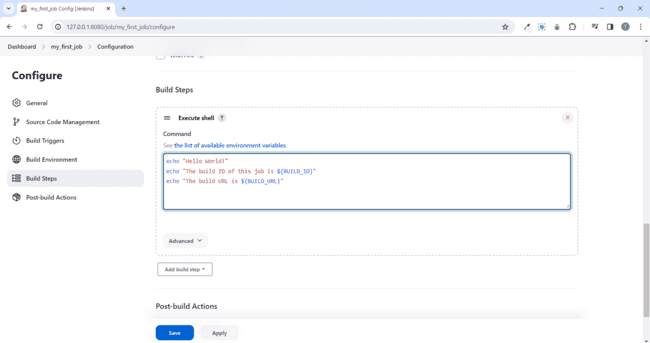
* now we can see the job has run successfully.
* So we can click into it and it echoes hello world!.

****

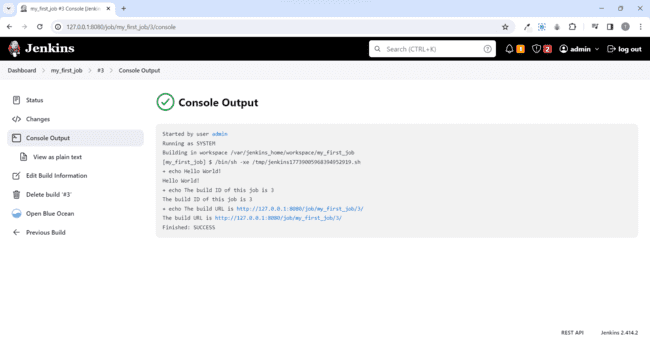
* Now go back to your project and click on configure.
* In the execute shell space click on “the list of available environment variables”.

****

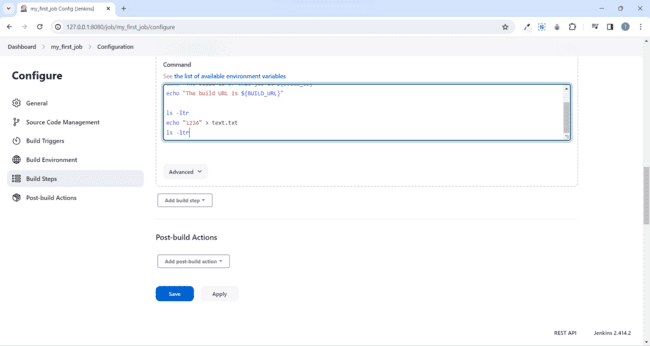
* Open it in a new tab you can see all the environment variables that are set for Jenkins that you can use through your build.
* You’re going to be using these variables quite a bit whenever you’re creating jobs.
* The most popular one is a BUILD\_ID and this gives you the current build id.
* This can be used when you’re versioning your docker images that you spit from my build you can specify the build id and that way you know which build actually built the image.
* Another one is the BUILD\_URL it gives the url for the build.
* Let us put these in our job.
* So to use Jenkins variable we can do it using echo.

****

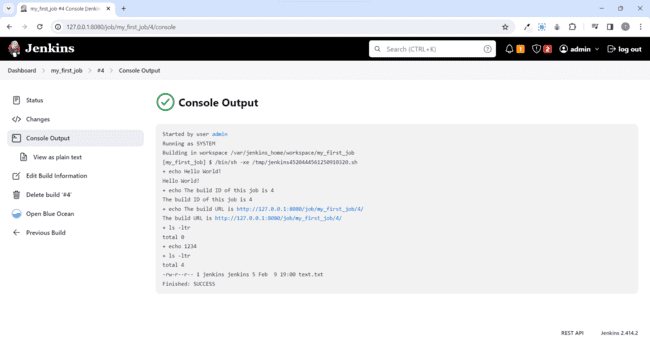
* now click on save and click on build now.

****

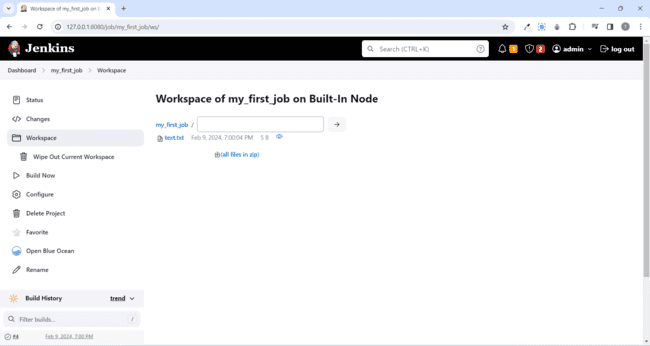
* We can see the output for echo, build\_id and build\_url.
* One thing you can notice is it’s spitting out everything twice this is because the + line is the actual command written and the next line is just the standard out of the console.
* Build url is useful to send slack notifications or knowing the developers that you have made a build.
* Now again go back to the project and select configure.
* If we add ls command this is just going to send the ls command to linux and ls -ltr this should show all the files within the current workspace.
* Now since we don’t have any files this probably won’t output anything.
* So let’s create a file using echo “1234” and we will send the output to a file called test.txt and again let’s do an ls after that.

****

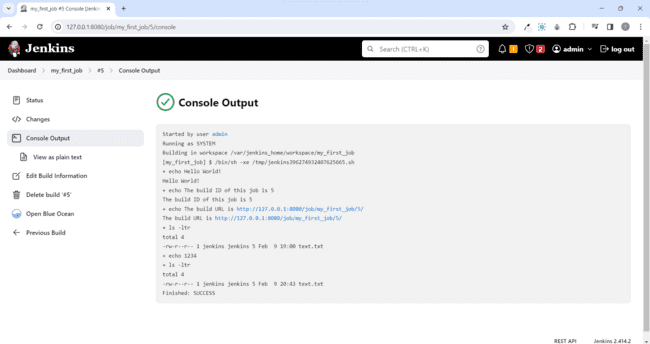
* Click on save and build now.

****

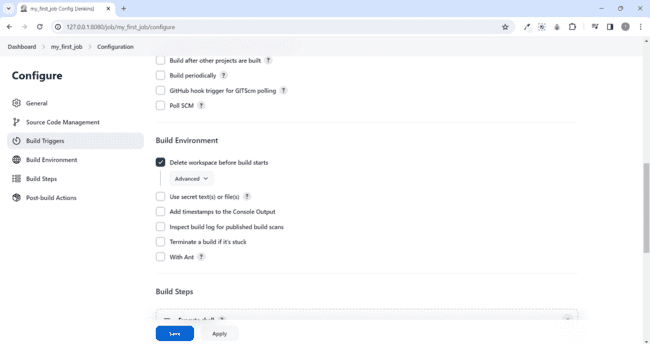
* Now you can go back to the project and click on workspace where you can find the file called text.txt.

****

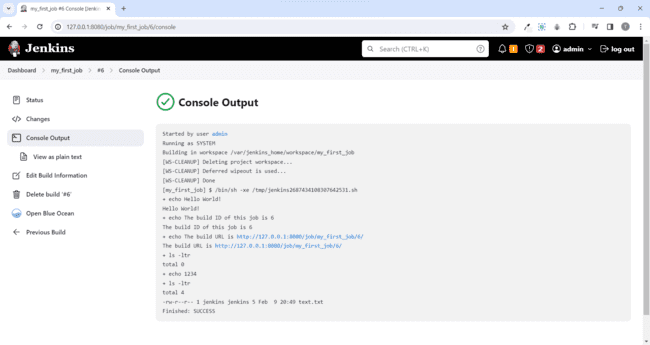
* Click the file to see the contents in it.
* Now if you run the build again and if you go to the console output this time you can see on the first ls -ltr that the file already exists.

****

* This means we are not clearing out our workspace and the files from the previous build are already in our directory.
* So if we go back to our project we can change that by going delete workspace before build starts.

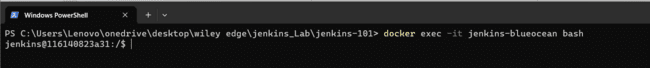
****

* Now click on save.
* Then click on build now again and your job is complete.
* Now on the console output you can see the first ls -ltr shows that there are zero files then we create the new file.

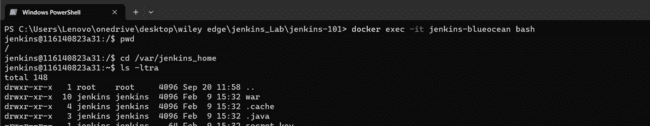
****

**Jenkins File System and workspace**

* Run a command to get into the actual docker container.



* Bash at the will give us a bash shell into the container.



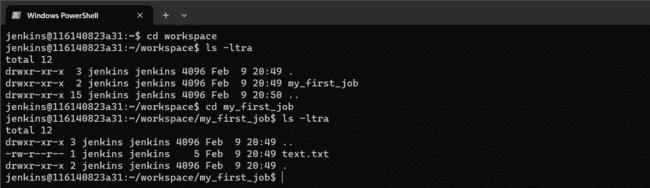
* In this list at the end we can see directory called workspace.



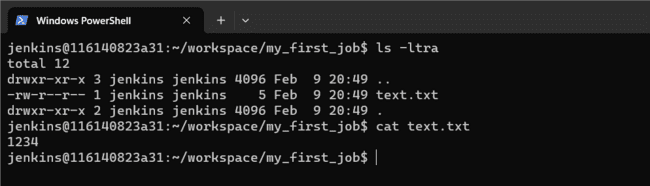
* You can now go into the workspace.



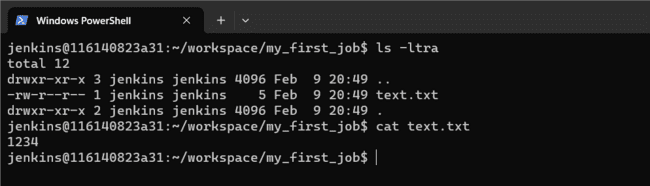
* You can see that we have a folder here called my\_first\_job.
* So this is the job that we have created.
* So whenever you create a build it’s going to have it’s own workspace under a folder of the job name that’s why you don’t want to create spaces.
* You can go in here and you will see all the files in here.



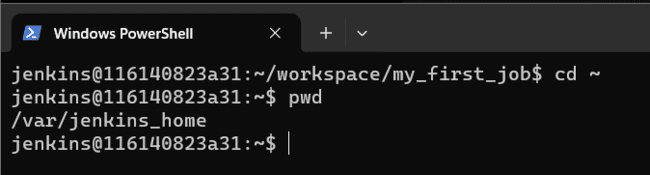
* You will see all the files in here so we can see our text.txt and then cat it out.



* You can see the output 1234 that we wrote in the execute shell.



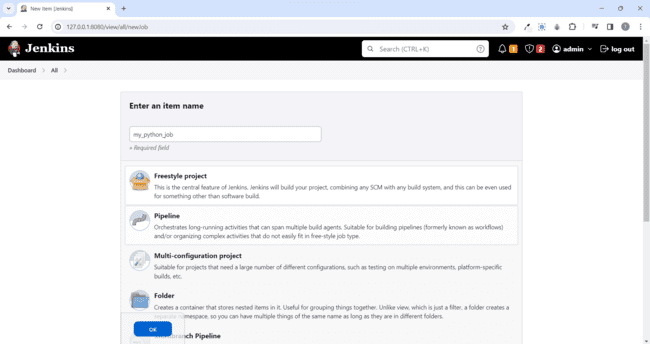
* So actually doing ssh into the Jenkins machine or doing a docker exec into the container and having a look at the file system is something you’re going to do quite a bit with Jenkins.
* Especially when you are troubleshooting builds and jobs sometimes it’s a lot easier to actually just go into the file system of the Jenkins server and have a look there.
* So a really great troubleshooting tool.
* Go back to the Jenkins home folder.



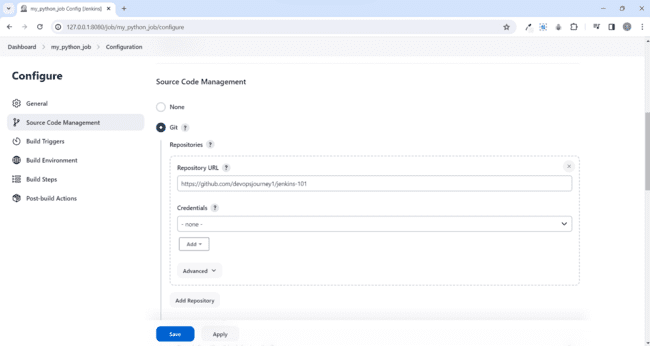
* You can have plugins folder, updates folder, some xml files for the Jenkins configuration as well as logs and various other folders in the Jenkins home folder.

**Running Python scripts with Jenkins**

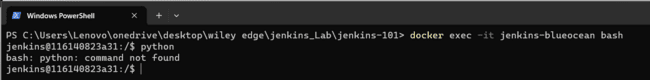
* For this one we’re going to clone down our repository and run a python script.
* Go to dashboard and click on new item to create a job and don’t forder to select the freestyle project and then click ok.



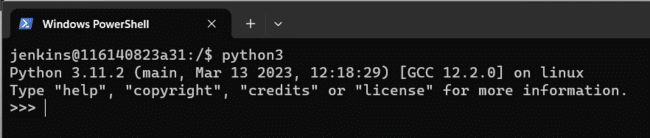
* Go to source code management click Git and put in github repository do the same if you’re following along there’s a python script in this repository that we’re going to run.
* And you can see there’s credentials here now this is a public repository so we don’t need any credentials for cloning it down.
* But if this was a private repository you would need to select your credentials or go ahead and add them so if you’re using a private repository go ahead and make sure you’re using credentials for that.



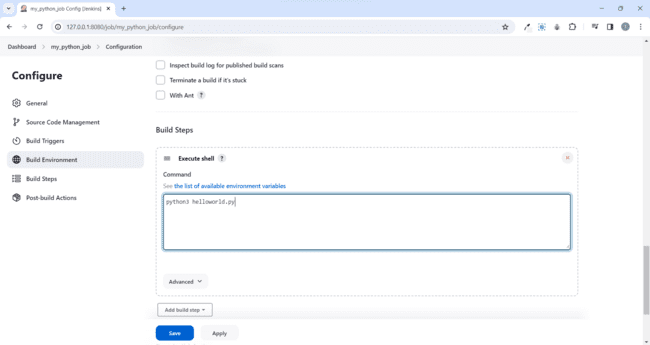
* Now head down to the bottom and we’re going to execute a shell.
* We will check which version of python is running.
* A quick troubleshoot to check if python is installed on the Jenkins server by typing python.



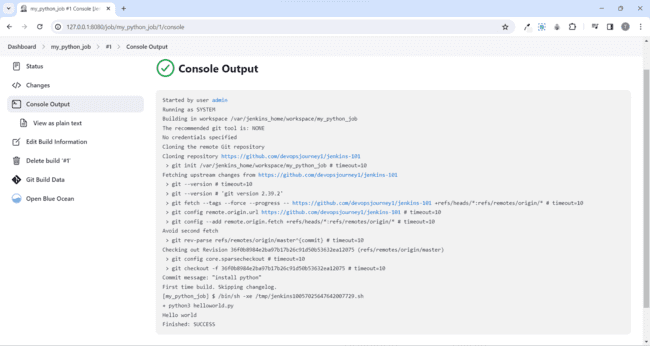
* We can see command not found.
* Let’s try python3.



* We can see python3 is installed.
* Let’s go to our job and make sure we use “python3” in the shell and then we want to run our script called “helloworld.py”.



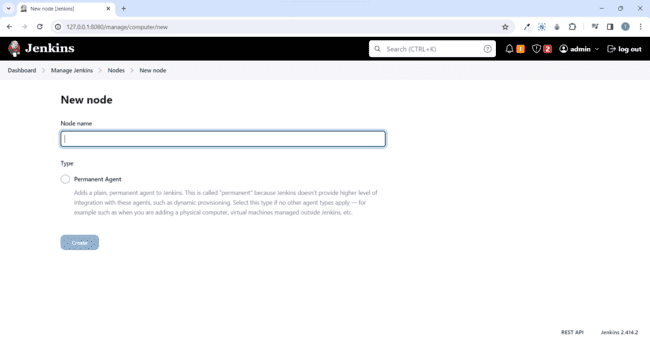
* Now save the job and build it.



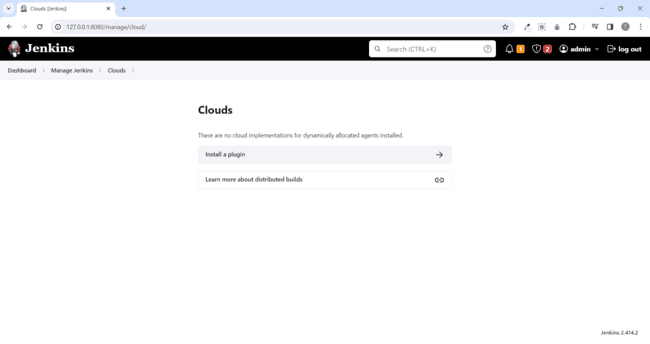
* It’s successful we can see it went out to public github repository it cloned everything down and then it ran python3 helloworld.py and we got the output of our script.
* This is actually quite powerful instead of us having to log on to a server to run a python script and make sure we have ssh to the server to make sure we can run the script we can just create a Jenkins job hit a button and we’ll know that it runs.
* We can also set up triggers for this job to be triggered like web hooks or when a repository is updated.
* As well as sort of like a cron schedule and we’ll also always get a history of when that script was run as well as full log output.
* This is just something that you can tie your python scripts too and then just sort of keep track of whenever they’re run.

**Setting up Docker Cloud Agents**

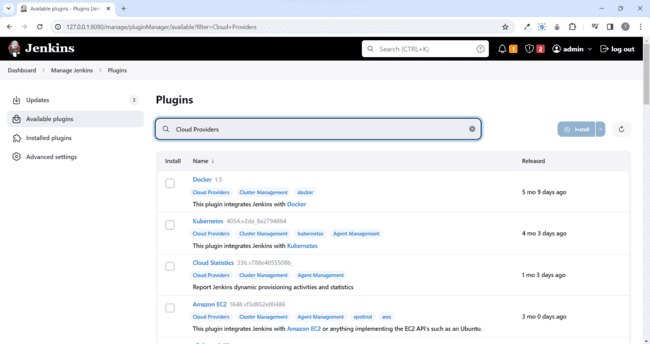
* Go back to the dashboard and then manage Jenkins > manage node and clouds and on the left-hand side there are two main options new node and configure clouds.



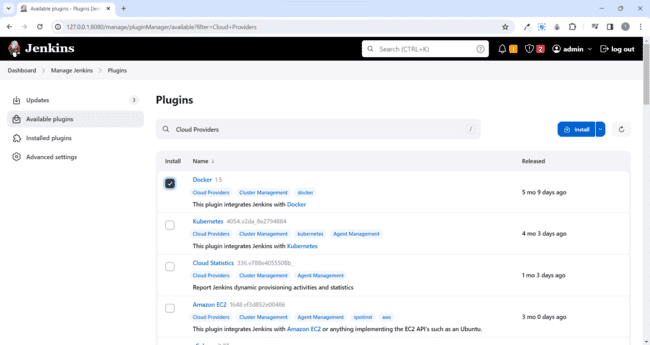
* If you go to new node this is how you configure a permanent agent.
* Basically this would just be any sort of like linux or windows server that you have that is sort of always available you set it up as a permanent agent and Jenkins just connects to it via ssh.
* And then distributes the jobs to it now this is sort of the older more deprecated ways of creating agents what people are usually doing now is configuring clouds.

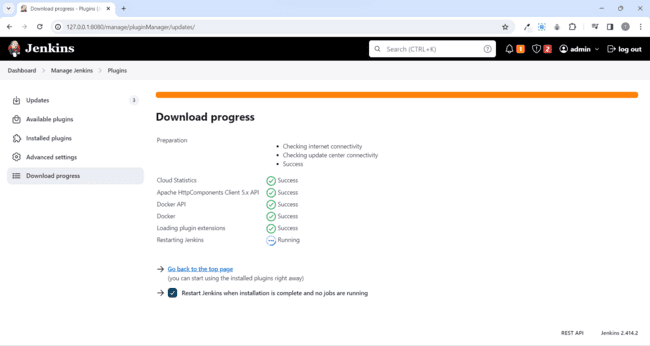


* This is how you set up agents using cloud platforms such as docker Kubernetes aws.
* Now that we are in here we can see that there no current cloud implementations available to us right now unless we go to the plugin manager (click on “Go to plugin manager”).
* And this just leads us to the plugin manager.

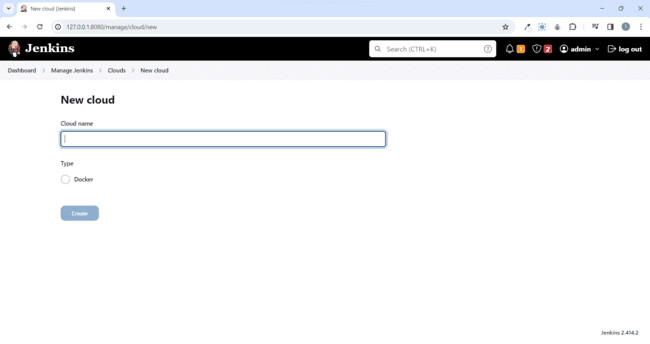


* We can see that it’s filtering on cloud providers here and this is basically just a list of all the cloud providers that you can use for your agents.
* Let’s go ahead and install docker but if you’re interested in Kubernetes or using aws you could use those.
* Click on docker and say download now and install.

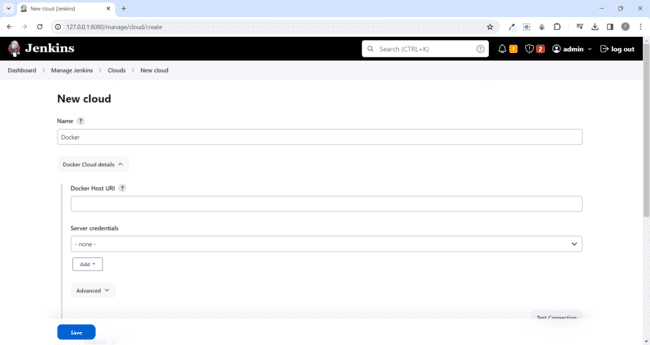




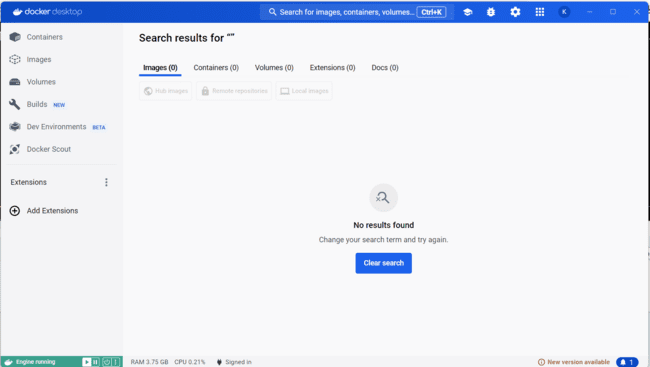
* After the Jenkins has restarted you will be asked to sign in again.
* Go back to dashboard > manage Jenkins > manage node and clouds > configure clouds > click on add a new cloud dropdown > we can see docker there.



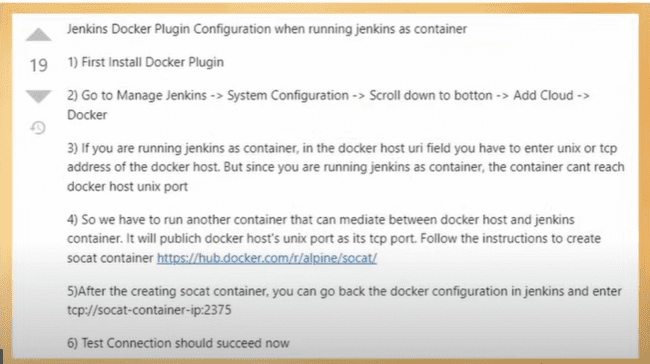
* Give a name and click on create.



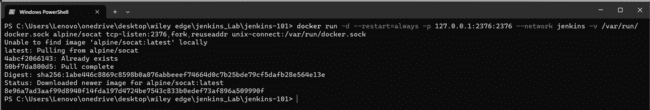
* We can see docker cloud details and we need to do couple of things to setup docker.
* The first thing is we need to set the url of the host that is running docker.
* So if you wanted to run docker within the Jenkins you could use like 127.0.0.1 and then the port for docker.
* But if you want to run it remotely then you’d put in the ip address of the remote server.
* Now run docker desktop on your local pc because we don’t want the docker to run within the Jenkins master.



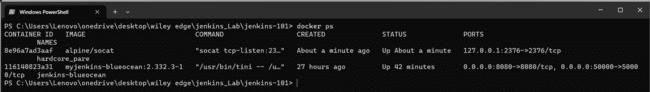
* We want it to run within our actual pc that’s running docker desktop.
* There’s a good stack overflow article that outlines how to do this.



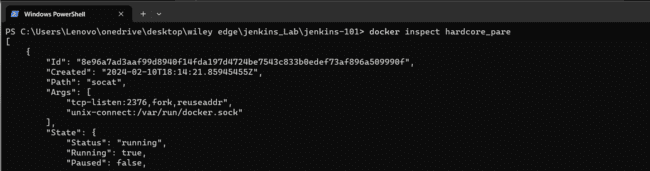
* So first launch a container running the alpine socat image.
* Write the command to start the container in the PowerShell that’s gonna help proxy the connection from our Jenkins master container over to our local host here.



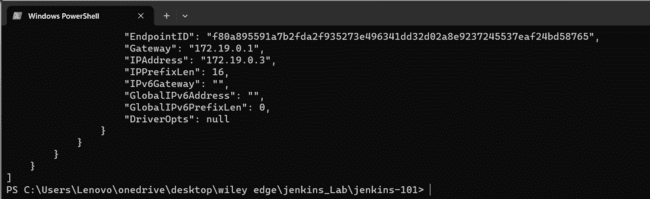
* Below you can see the container is up (alpine/socat).



* Now we need to inspect on the container using it’s name



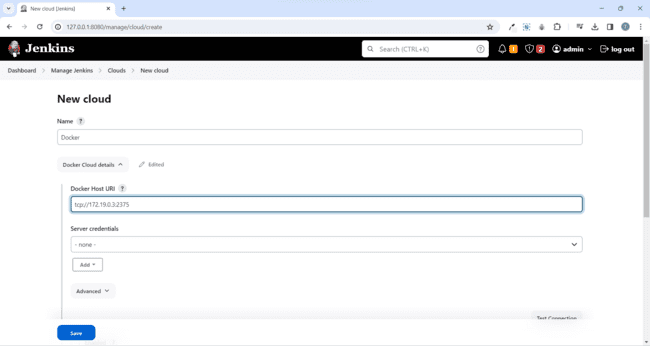
* Now we need to go to the bottom and take the ip address of the socat container under IPAddress row.



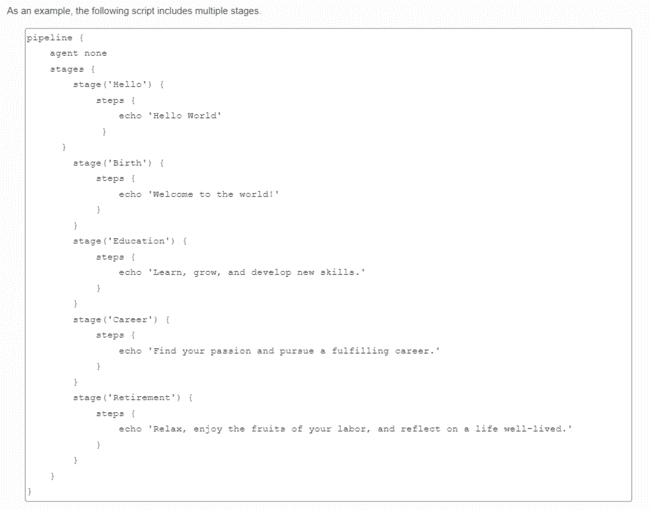


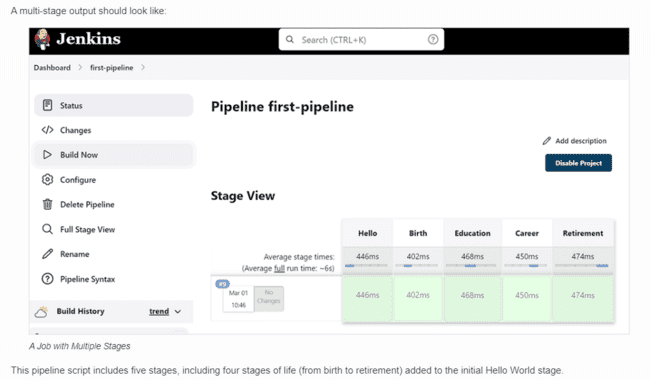


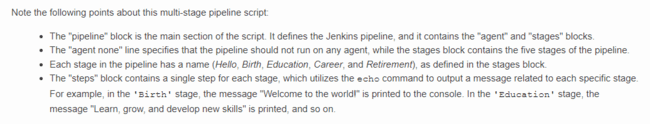
* So I want to forward my connections for docker to this ip address.
* Let’s go to Jenkins new cloud and under docker host url write tcp://172.19.0.3 and then it’s going to be port 2375.



* Then hit enabled option and then click on test connection to just make sure that everything goes through.
* And we can see it has connected properly as it gives version and api version left side.

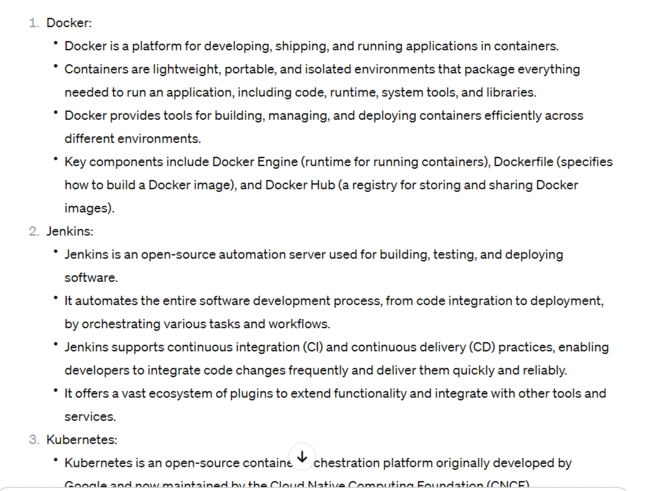






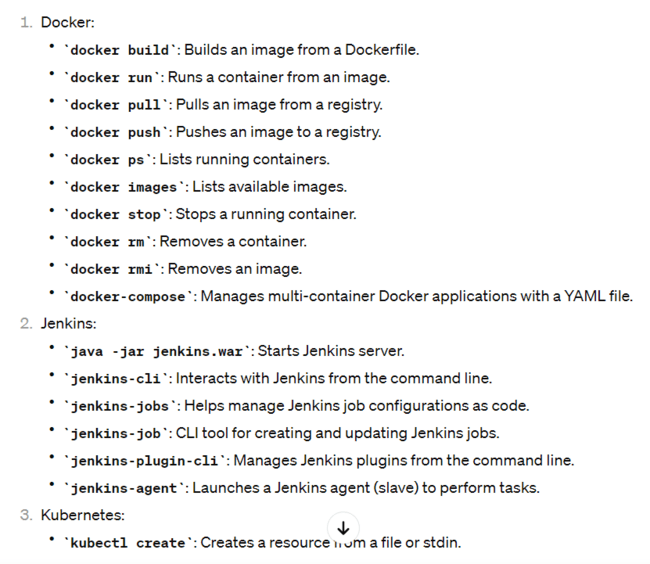
**Docker vs Jenkins vs Kubernetes**

**Definitions**



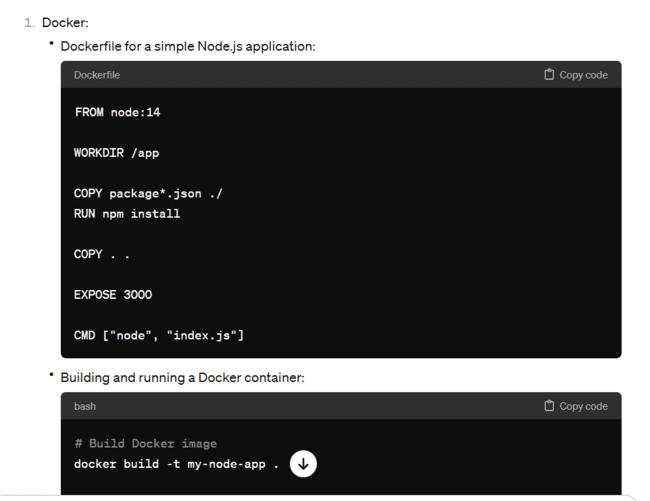


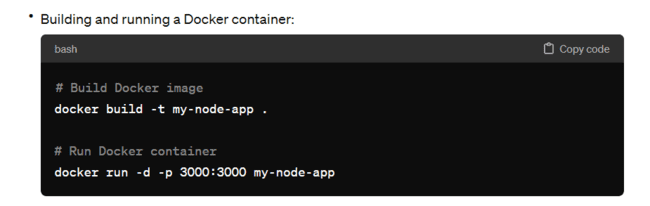
Basic commands

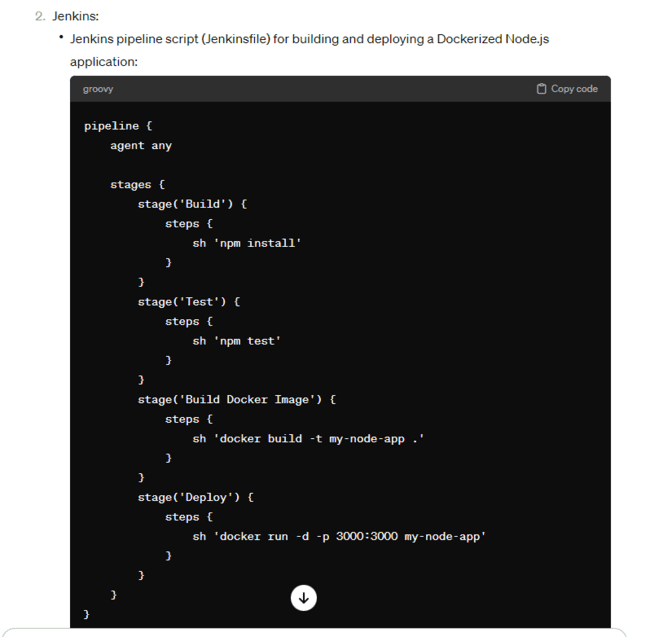


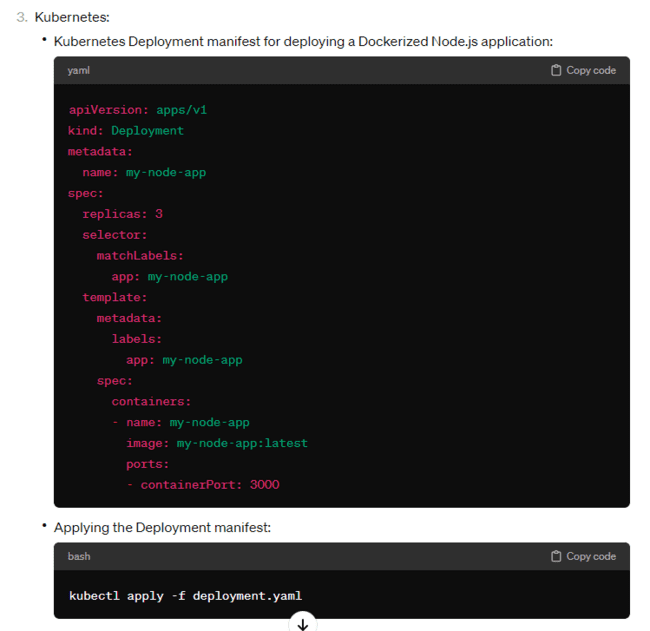


Example codes:





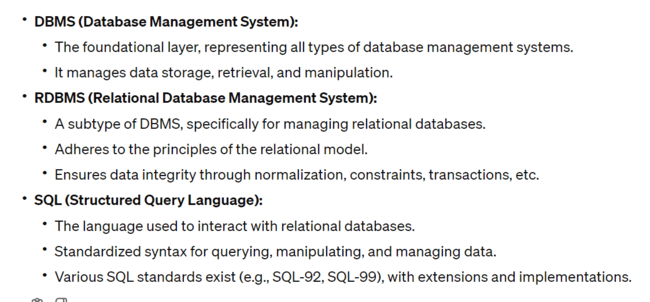




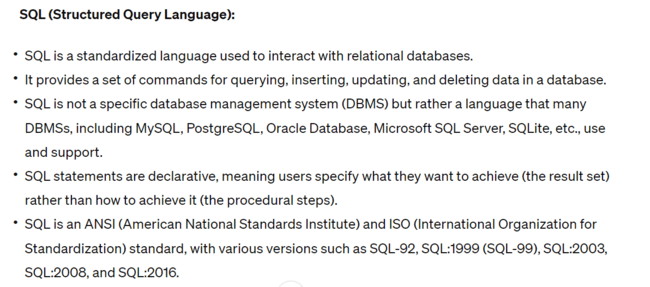
**DBMS**

term database can be used to refer to any electronic collection of data

Relational databases are highly structured and designed to minimize how much storage space the data requires and reducing data anomalies.



**SQL vs MySQL**



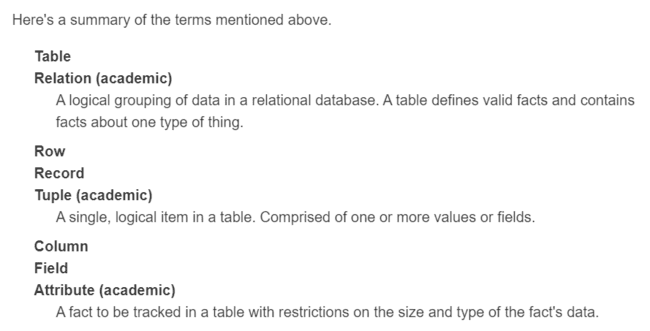
We often use the word persistent rather than permanent, because no data storage option is failproof.

**Database vs DBMS**

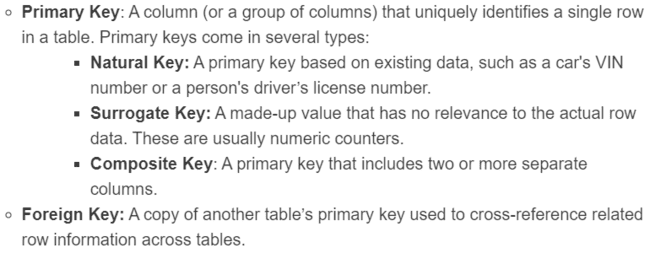
* A database is a structured representation of data that we can read from and write to.
  + **structured**—a database stores its data in files with a specific format that we conceptualize as tables, columns, and rows. The actual physical storage is not table-like. It's more complicated, and database storage formats are the result of many decades of research.
* A **DBMS** (database management system) is a software system that manages a database.
* All RDBMSs (**relational** database management systems) use SQL as the underlying language to create and manage data, but each RDBMS includes options that differ from each other, similar to how the same spoken language can have multiple dialects.
* Unstructured databases have the advantage of not requiring that the data be organized, making it useful for social media posts that can contain virtually any type of content, from text to pictures to videos.
  + Unstructured data is easy to create, save, and share, while structured data is easy to search and sort to find existing data.

**Relational Database Concepts**

* Relational databases organize data into one or more **tables** or **relations.**
* Rows are also known as **records** or **tuples**. The term record is common, while tuple is an academic term. A "database record" is a single row in a table in the database.
* Columns are also known as **field** or **attributes.** The word attribute is an academic term.

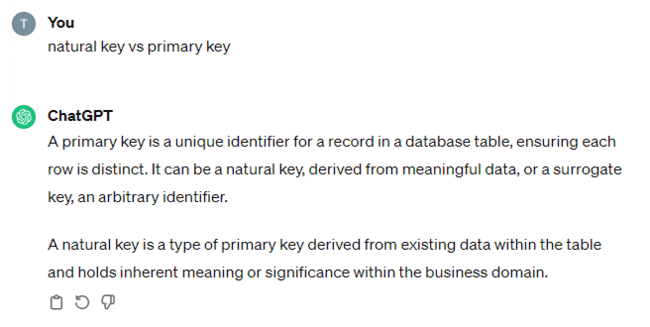


**Keys**

****

**Primary key**

* If the values in a column or set of columns can be used to uniquely identify a row, the columns are a candidate to be a **primary key**.
* The database system will reject records that use existing values as the primary key or that don't have any value at all for those fields.



**Natural key**

* One option for a primary key is to designate a single existing column, a **natural key**, as the primary key.
* For example, in a car database, we could potentially use the VIN as a primary key. Every car has one, and the VIN is different for every car. Likewise, in a retail store database, the UPC code already assigned to each product may work as a natural key in a product table.

**Compound key**

* Another option for a primary key is to use multiple columns to create a **compound key**.
* Compound keys with multiple columns, adding more and more columns to a primary key is complicated. Compound keys are hard to use in the long run.

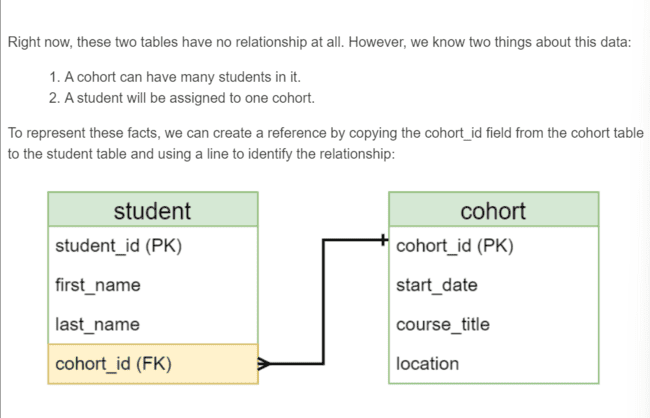
**Surrogate key**

* Database systems offer a solution: generate a unique value per row automatically and use it as the primary key. A generated unique value with no real-world meaning is called a **surrogate key**.
* It's just an arbitrary value that uniquely identifies the row.

**Relationships**

* the relational model described by Edgar F. Codd.
* Relational also means that data in one table can relate to data in another table. A row in one table may have a relationship with one or more rows in another table.
* Because a primary key uniquely identifies a single row of data, the best way to represent a relationship between tables in a database is to use a copy of one table's primary key as a column in the other table.





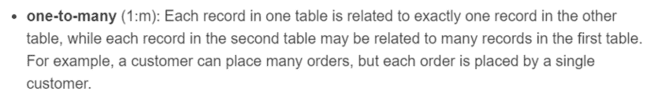
* Now we have a primary key cohort\_id in the cohort table and a second cohort\_id field in the student table.
* This means that any row in the student table can reference a specific cohort using the cohort's primary key.
* As the primary key, the cohort\_id must be unique to each cohort, so this serves to uniquely identify which cohort a student is enrolled in.

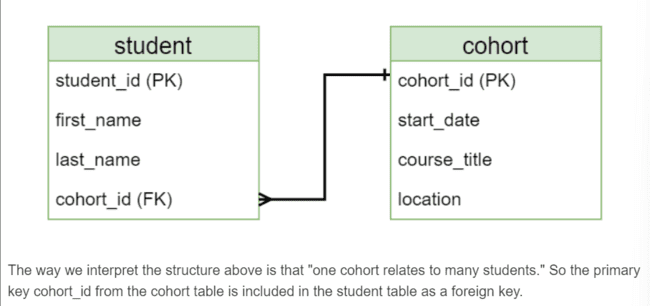
**Foreign key**

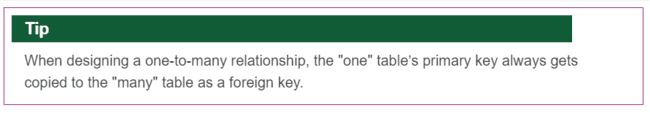
* When you add another table's primary key to create a relationship, the column becomes a **foreign key** in the second table.
* So cohort\_id in Cohort is a primary key and cohort\_id in student is a foreign key.
* Foreign keys are not unique within the table, which means that we can use the same cohort\_id for many different students, allowing us to assign multiple students to the same cohort.

**Cardinality**

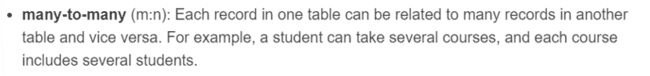
* How and where we add foreign key columns depends on the **cardinality** of the relationship.
* that is, the number of rows from one table that relate to a row in the other table.
* In a relational database, we're only concerned with three cardinalities:



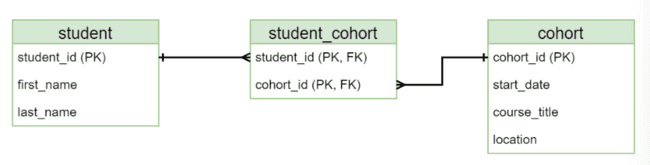








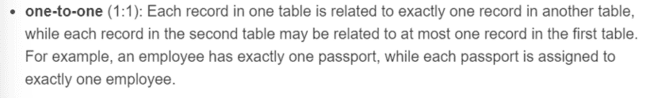
* When we have this type of relationship, we must introduce another table, called a **bridge table**, to the database.
* The bridge table’s job is to hold all of the unique key combinations from the tables that hold the actual data.
* As we saw above, if we put the cohort\_id foreign key in the student table, we can only assign one cohort per student. If we put student\_id in cohort, we can only assign one student per cohort.
* We want to support many combinations of student\_id and cohort\_id. This is what the bridge table does for us:



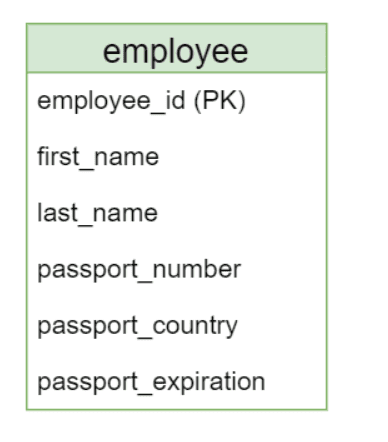
* The student\_cohort table is a bridge table that allows us to repeat student\_id as many times as we want AND repeat cohort\_id as many times as we want, allowing the database to assign multiple students to the same cohort as well as assign multiple cohorts to the same student.
* The bridge table normally includes both of the primary key fields used in the two original tables, so each of those fields is a foreign key.
* However, the fields themselves are used together as a primary key in the bridge table.

**Composite key**

* We use the term **composite key** to describe a primary key that includes two or more fields.
* In this case, the primary key is the combination of student\_id+cohort\_id.
* Because the combination student\_id+cohort\_id is allowed only once in the bridge table, this gives us the useful side effect of not being able to assign the same student to the same cohort multiple times.



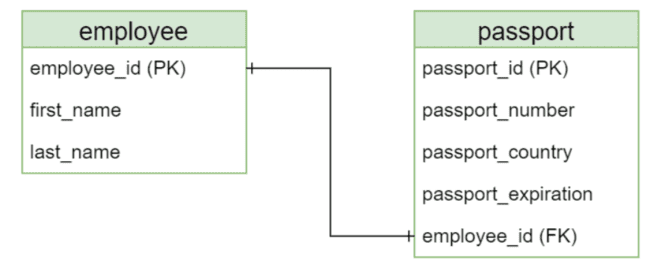
* One-to-one relationships are rare, however. Logically, if there is only one possible row in the first table that relates to only one possible row in a second table, we could simply put all of the data into one table.
* The most common reason to set up a one-to-one relationship is for optimization (performance) purposes.
* The more columns a database table has, the longer it takes to read the information out.
* A table with lots of columns (by lots, we mean 30+) will sometimes be split into two tables. There are a few reasons for doing this:
* Many of the columns in the table are rarely used.
  + Moving rarely-used columns to another table may make the storage and retrieval more efficient.
* There is a logical separation in the column information.
  + Database designers like the single responsibility principle as much as software developers do.
* Some data is more sensitive than other data, and moving the more sensitive data into a separate table helps make it more secure.



However, notice a few things here:

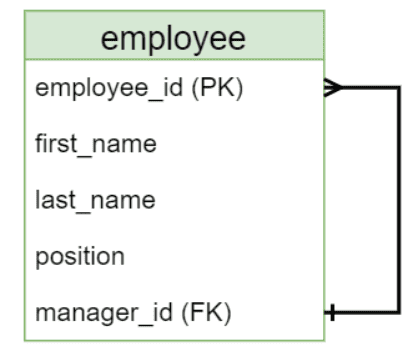
* The passport fields seem to represent a different concept than employee, in that the country and expiration date describe the passport, not the employee. We probably want to put this in a separate table just because employee and passport are separate concepts.
* These three fields will be blank for most employees, meaning that we are storing an empty (**null**) value in those fields for most records in the table. While this is possible, even null fields take up space and can slow down retrieval, especially when we don't care about the passport information for a specific report.
* Passport information is more sensitive than names, and putting this data in a separate table makes it more secure.

It makes sense, then to split this into two tables, where passport-specific information is in a separate table that uses the employee\_id field as a foreign key.

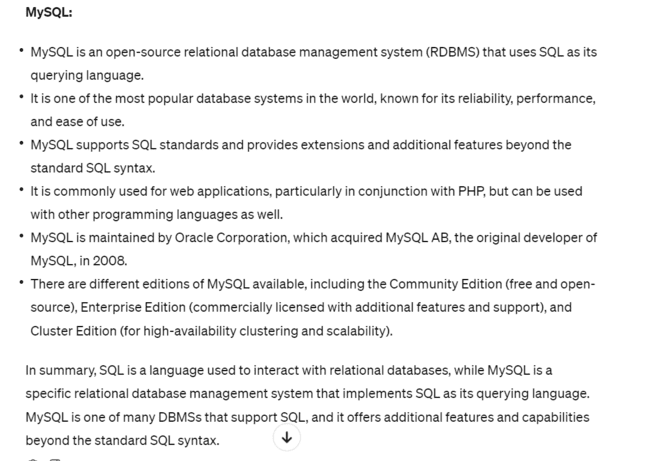


**Self-Referencing Relationships**

* Consider a table of employee data. If the company uses a typical hierarchical structure, then the manager of an employee is also an employee.
* We can represent this relationship by setting a column in the table as a foreign key to another column in the same table like so:



* One manager can be the manager of many other employees!



**TRANSACTIONS**

* A relational database allows the following actions:
* Read existing data
* Insert new data
* Update existing data
* Delete existing data
* Add or alter schema (tables and relationships)

A transaction is a set of one or more actions that represents a single, logical unit of work.

(or)

Is a set of SQL statements which execute as a single task. This means either complete successfully execute or doesn’t execute at all.

* If I purchase a concert ticket for an in-demand concert, the software system must first find an available ticket and then put it on hold until I can provide payment. That's a transaction.
* Characteristics of transaction is ACID.

**ACID Properties** (transaction characteristics)

**Atomicity**

* A transaction is atomic if it follows the “all or nothing” rule.
* If one action in the transaction fails, then the entire transaction fails. An atomic transaction never partially succeeds.

Ex:

Imagine a scenario where you write a new row of data to a table with 10 columns. On the 8th column write, a power failure occurs and your server shuts down. If the database supports atomicity, it will notice the unfinished transaction and restore the data to its pre-transaction state when it comes back online.

**Consistency**

A transaction is consistent if it can only move the database from one valid state to another valid state. A consistent database enforces constraints on the types and sizes of data that are allowed.

It also enforces primary and foreign key relationships. For primary keys, consistency means that the system will never allow a duplicate key in a table to occur and it will require that each record have a primary key value. For foreign keys, most DBMS systems by default will not allow you to **orphan** a row, where the value used as a foreign key does not correspond to a value in the primary key of the related table. As an example, consider a Customer and Order relationship where a Customer can have one or more Orders. If you were to try to delete only a Customer row without first deleting its Orders, then the Orders associated with that Customer would have a foreign key pointing to a record that no longer existed. A properly configured relational schema will prevent this from happening by either rejecting the delete transaction outright or by automatically deleting all the orders associated with the customer being deleted first. (This automation is called a **cascade delete**, and because it can lead to the deletion of millions of records without warning, it is usually not the preferred solution to resolving orphan keys.)

**Isolation**

* A transaction is isolated if its effects are not visible to other transactions until it is complete.
* This is often referred to as concurrency control.
* A large database application may have hundreds or thousands of users making changes to it at the same time, so if transactions are not isolated, this could cause inconsistent data.
* Imagine two users: John and Sally. John is updating data in the orders table. At the same time, Sally is reading data from the orders table, including records being edited by John.
* A DBMS has various levels of isolation it could apply.
* As a beginner you only need to know two:
* **Serializable –** Sally will not receive her data until John’s changes are committed. When John begins a transaction to change data, the data is **locked** until his transaction is complete.
* **Read Uncommitted –** Sally will get her data right away, including whatever changes John has made that haven't been committed yet. This is called a **dirty read** because it is possible that John’s transaction could fail and roll back.
* The default isolation in most DBMS systems is serializable.

**Durability**

* A transaction is durable if once it is committed (saved to the database), it will remain so, even in the event of catastrophic failures.
* Even if you kick the server's power cord out of the wall after a transaction, it will stay committed.
* This means a transaction is not fully committed until it is written to permanent storage, such as a storage drive.

**Log Files (transaction log, journal or audit trail)**

* a transaction log (sometimes referred to as a journal or audit trail) is a history of executed actions.
* The log file is physically separate from the actual database data. This is important to ensure a database remains consistent. For example, when you insert a new row into a table, a few things happen:

1. The DBMS validates the incoming command.
2. A record is added to the log file specifying what changes are about to be made.
3. The DBMS attempts to make the changes to the actual data in the table(s).
4. If successful, the log record is marked as committed.

* If a failure occurs between steps 2 and 4 above, like a server reboot, the DBMS will scan the log file for uncommitted transactions when it comes back online. If it finds them, it will examine the actions performed and undo them, effectively restoring the database to its former, consistent state.

**Backup Strategies**

it is important that the database administrator has backup and recovery options for both data and log files.

Because logs contain all transaction information, they provide point-in-time restoration information.

Full data backups tend to be very large and are only done periodically. Log backups tend to be much smaller.

As an example, we might perform a nightly data backup and a log backup every 10 minutes. If our data backup occurs at midnight and the server fails at 2:55 PM, we would restore the last data backup, then restore all the logged transactions until 2:50 PM. We would only lose changes between 2:50 PM and 2:55 PM. (Not great, but better than the alternative.)

To further reduce losses, we could use multiple database servers and execute transactions on each. If one server fails, another can take its place. This is called a database cluster.

**Normalization**

* Original designer of the relational model, E. F. Codd, proposed a set of rules that can be used to normalize a large dataset, and Codd showed that is possible for any data domain to be reduced to simple table relationships.
* **Normalization** is the process of breaking down complex relationships into simpler structures.
* A properly-normalized design improves performance and reduces the complexity of relationships by minimizing data duplication (redundancy).
* A database where all relations are reduced in this manner, following the process of normalization, is said to be normalized.

**Data Redundancy**

* Data redundancy is the act of storing the same piece of data multiple times in the database.
* Consider what happens if we were to include student data and cohort data in the same table.
* Aside from the inefficacy of storing many copies of the same data (disk space), having the data duplicated like this can lead to data anomalies (incorrect data).

**Update anomaly**

* Besides the time required to update that many different rows, what if you missed a row or mistyped the date (using 8 Jan instead of 6 Jan or 2024 instead of 2025) in at least one row.

**Insert anomaly**

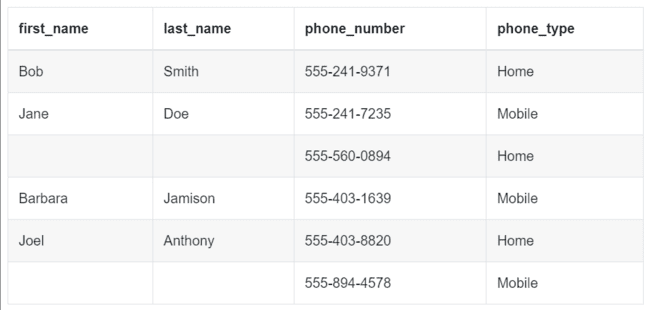
* Another issue that could come up might occur when setting up a new course. With this organization, how would we put the course data into our database if we haven't yet found students to take the course?

**Delete anomaly**

* If we just have the data above, canceling C140 would completely delete the data for Depti Bebum, who is enrolled in only that one course.

**Functional Dependencies**

* A **functional dependency**, as the name implies, is a dependency relationship.
* That is, "column A depends on column B", or "columns A, B, and E depend on columns C and D".
* In a well-designed table, all columns will depend on at least one column in the table. If there are columns that are independent of the others, they are candidates to be moved to a separate table.
* The process of normalizing a database involves applying a sequence of rules called **normal forms** to the database.
* Getting your database to the second and third normal form will eliminate most redundancy and insert, update, and delete anomalies.
* To work through forms one through three, we must start with a pile of denormalized information.

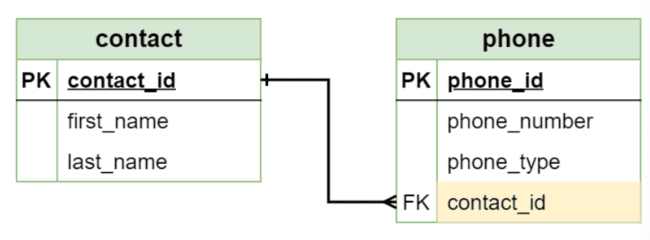


* The above table is what we call a **heap**.
* This means that it is unstructured and has no constraints that are required by any normal form.

**First Normal Form (1NF)**

To achieve 1NF, the table must satisfy the following conditions:

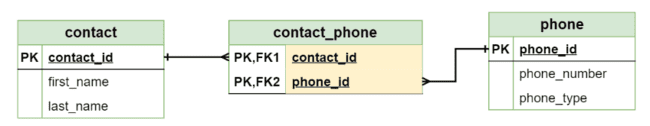
* There is no top-to-bottom ordering to the rows.
* There is no left-to-right ordering to the columns.
* Every row/column intersection (field) contains only one value.
* Every row can be uniquely identified.
* When we break up one large table into two or more separate tables, we have to create a relationship between the tables.
* We do this by first deciding which is the "one" side of the relationship and which is the "many" side of the relationship, and we then use the primary key of the "one" side as a foreign key of the "many" side.



* Whenever a new table is created, we have to apply the current normal form to the new table.

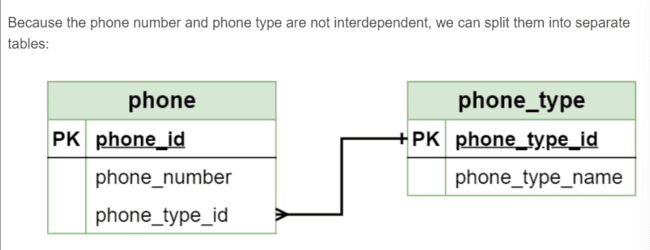
**Second Normal Form (2NF)**

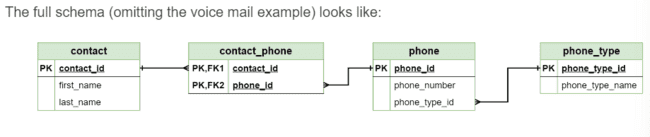
* In plain English, you must already be in 1NF and then all of the columns except the primary key need to be strictly related to each other.
* By definition, 2NF only applies to situations where a table has a **composite key**: a primary key that includes two or more fields.
* This comes up when we try to resolve the problem of two or more people sharing a phone number while also allowing one person to have multiple phone numbers: a classic many-to-many relationship.
* The normal way to resolve a many-to-many relationship is to create a third bridge table that connects the original two tables, using the primary keys from the original tables as the primary key of the new table.



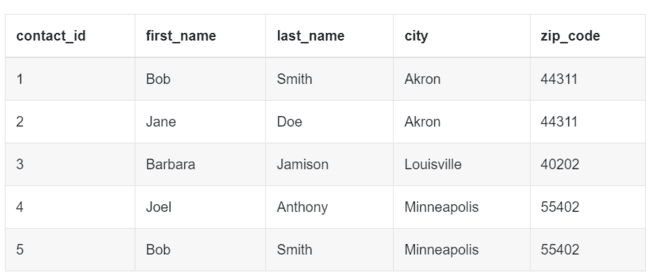
**Third Normal Form (3NF)**

* Alternatively, a table is in 3NF if and only if it is in 2NF and no non-primary-key column is functionally dependent on any non-key set of fields.
* In the abstract, it's this situation for a table with fields A, B, and a PK.
* If the value of A relies on PK and B relies on PK and A also relies on B, then you can say that A relies on PK through B. That is that A is transitively dependent on PK.

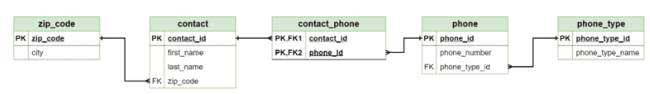








* In this design, we have a **transitive dependency**.
* Putting both city and zip code in the same table implies that both city and zip code depend on the primary key contact\_id, but in the real world, a zip code is assigned a city.
* So city relies on zip code which in turn relies on contact\_id.
* To put it simply, what if I were to change Bob Smith's city to Los Angeles but not update the zip code?
* We would then have an update anomaly in our table!
* We fix this by moving the zip\_code and city columns to their own table. Now to update the contact's location, we only have to update the zip code to keep the city in sync thanks to the new relationship:



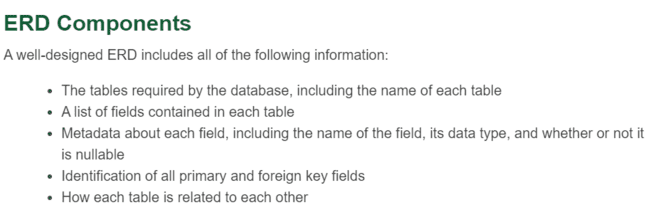
**Denormalization**

* When optimizing for extreme performance, there are times that you are less concerned with data protection and choose to move back towards 1NF. This is called denormalization.
* A large enterprise will often have databases that are normalized with good data protection for transaction processing and working with their applications.
* Then they also will have another set of denormalized databases (referred to as data warehouses) that are optimized for aggregating and reporting data (to easily get data from different isolated tables).
* however, that these denormalized data warehouses are essentially a snapshot of the data at a given point in time.
* No changes are made to this data, so there is no possibility of introducing anomalies in the data.

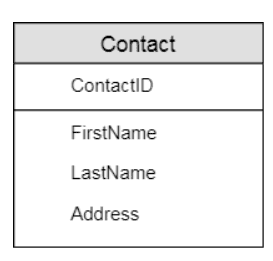
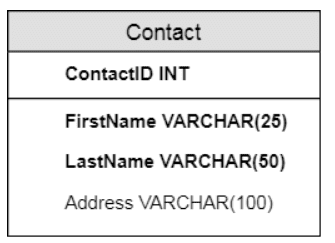
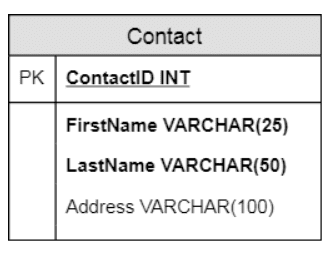
*There is a phrase that people say normalize until it hurts and then de-normalize until it works.*

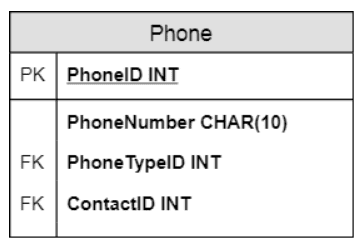
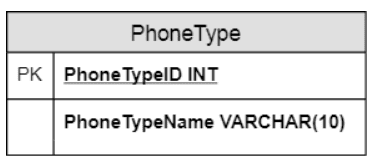
**Entity Relationship Diagram**

* a visual representation of the database structure.
  + It helps identify places where the proposed structure may not work.
  + Helps identify one or more tables aren’t related to other tables or some relationships don’t make sense when you try to map then to the normalized tables.
  + Identifying these problems in the design phase will help avoid problems that might otherwise appear in the SQL scripts used to define the database.
  + helps each team member quickly identify what fields are in which tables and how the tables are related.
  + It gives you a single, condensed representation of the structure, which helps you write SQL statements more quickly, especially when the SQL statement references two or more tables.
* An ERD is the equivalent of a floorplan or blueprint of the database.

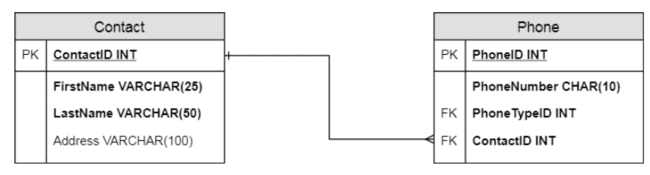


**Components**

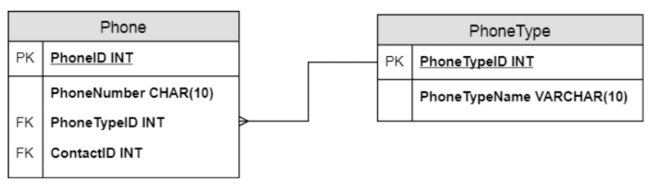
* Tables
  + The ERD normally includes the name of the table in the topmost section (like a title), the primary key field(s) in the next section down, and the remaining fields are added to the bottommost section.
  + 
* Fields
  + we need to know which fields are nullable and what data type each field will use.
  + we will format each required (non-nullable) field in bold and add the data type to the right of the field name.
  + 
* Keys
  + A final addition at the table level is to indicate the primary and foreign keys. This is often done by adding PK or FK (respectively) to the left of the appropriate field names in the table.
  + It is also common to underline the primary key field (or any other field whose value must be unique within the table), to help make it even more visible.
  + 
* Additional Tables



* Relationships

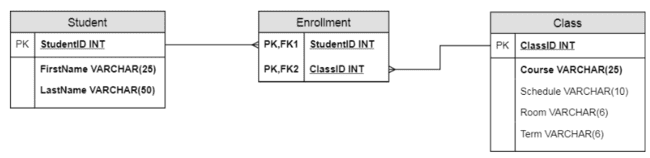


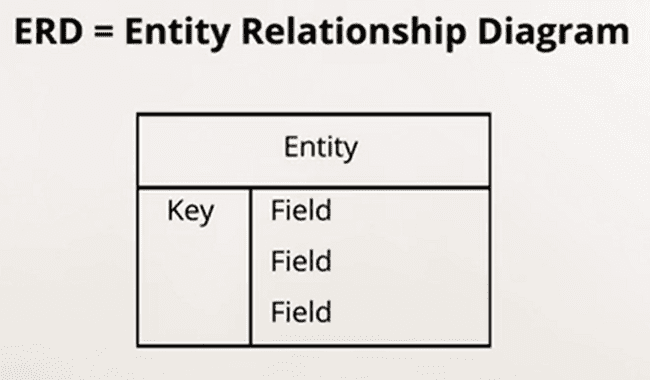
* crow's foot marking (three small lines replacing the arrowhead) on the many side of the relationship to indicate that any record in the Contact table can be associated with several records in the Phone table.
* In addition, there is a small vertical line on the Contact side to indicate that each record in the Contact table must have at least one related record in the Phone table.



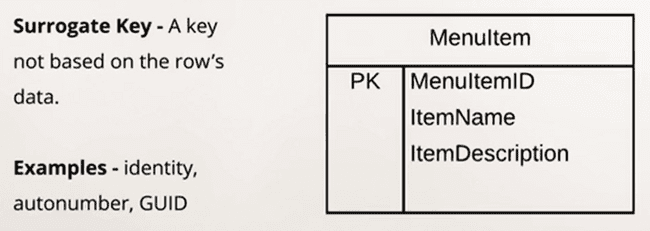
* each PhoneType can be associated with any number of phone numbers, but each phone number must have exactly one type, so the crow's foot appears on the Phone table (the many side of the relationship).
* Here, though, we exclude the vertical line on the PhoneType side. It's quite possible that when we set up the database, we might include phone types that may never be used (like fax numbers), so we don't want to force each PhoneType to be used if it isn't necessary.

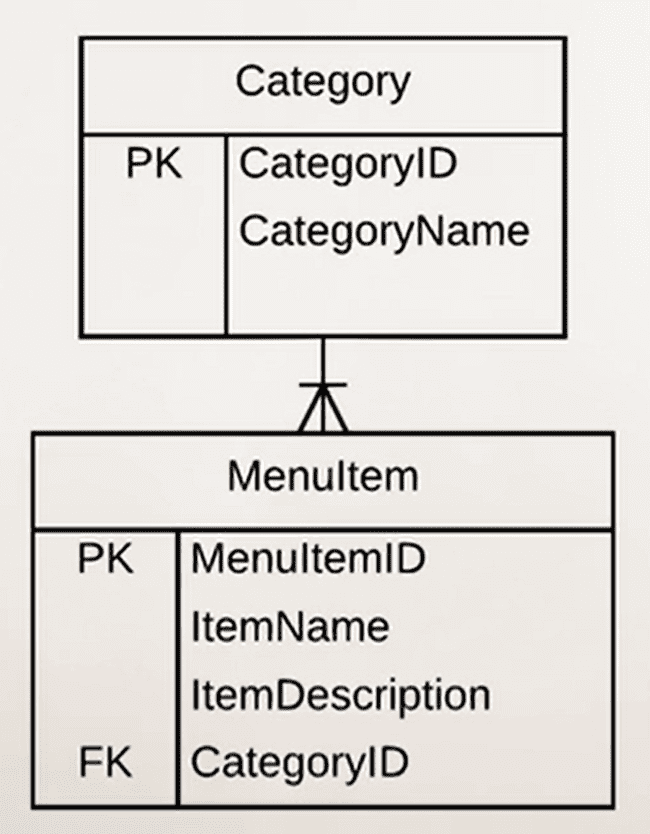
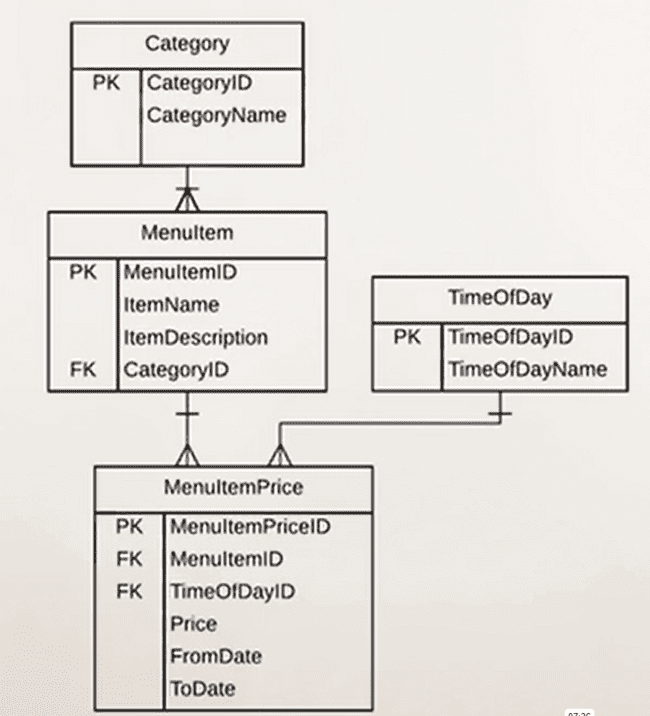
Any many-to-many relationships will be defined as two one-to-many relationships between three tables, using a bridge table to connect the original tables.

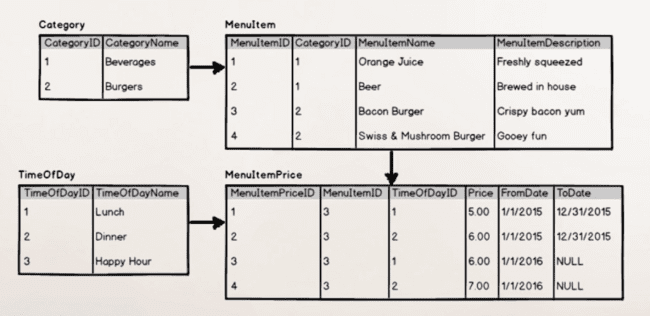




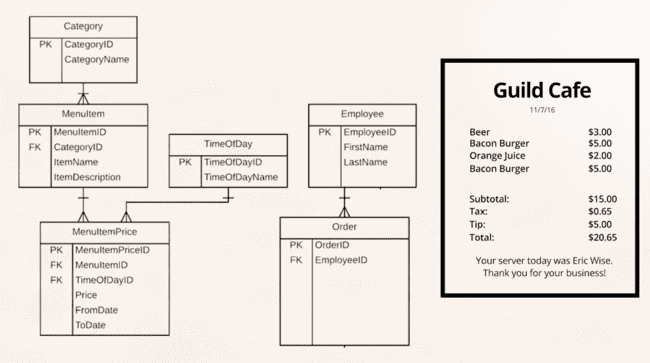
* When we have entities we start assigning the datatypes to the fields.

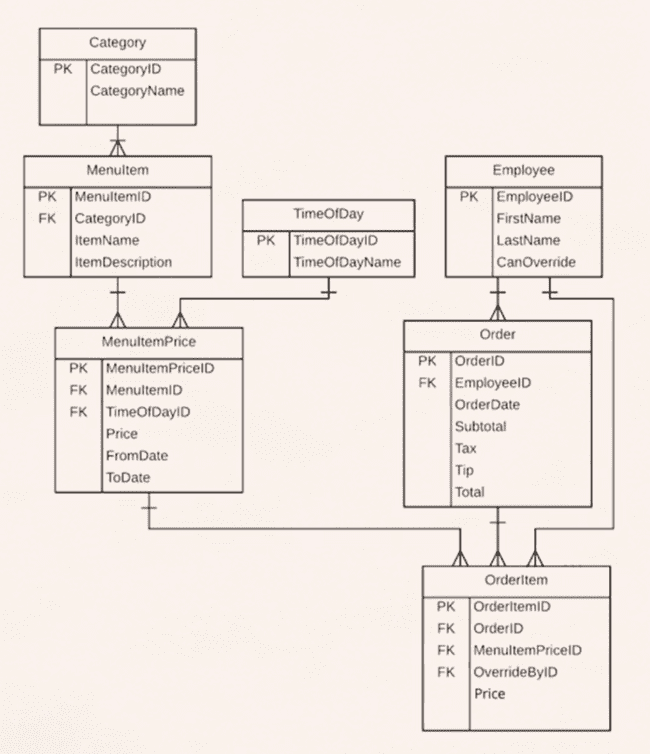




* Now updated the ERD to take the orders from the customers.





**Null Value**

* When we define a table, one of the key characteristics of any field is whether or not a value is required for that field in each record.
* We also use a variety of terms for that concept, including required (which means that there must be a value for every record), nullable (which means that a value is optional), and the official definition NOT NULL, which means that a value is required.
* Essentially, a null value is an empty value: it contains no value at all.
* *null* is not the same as zero. Zero is a value, while *null* is not.

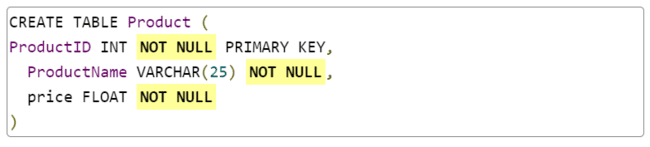
**Nullable Fields**

When we define a table for any RDBMS, we must determine whether each field in that table is nullable or not.

The one hard and fast rule about nullable is that any field used in a primary key is NOT nullable.

Entity integrity requires that we have a value for every primary key, and this is a default setting when we define a field as a primary key.

All other fields in a table are nullable by default. In other words, if we don't set a NOT NULL attribute on a field, SQL will allow users to leave those fields empty.





* Choosing to allow nulls in a table has consequences on the efficiency of the database.
* RDBMSs set aside storage space for each field in a record, based on the defined size of the field.
* For example, an INT field requires 4 bytes of space, while a DATETIME field requires 8 bytes of space.
* Note that this is **reserved** space, not used space, and the RDBMS will use that much storage regardless of the actual value stored in each of those fields, and even if nothing is stored there.
* However, if a table has a considerable number of nullable fields, it may be worth creating a separate table that includes only those fields, allowing users to create records with those fields only as necessary.

**Indexes**

* Most textbooks include an index of major topics, for example, but recipe books use indexes to help users find recipes based on ingredient or cuisine, while atlases use indexed to help users find specific maps based on location name.
* Relational databases use index values in much the same way and for the same reasons.

**Primary vs Secondary Storage**

**Primary Storage**

* primary storage (what we call memory or *RAM* in a computer system) and secondary storage (hard drives, flash drives, and solid-state drives, for example).
* Memory term = primary storage, storage term = secondary storage in this context.
* Memory is the space where a computer holds the instructions and data it is currently working with.
* Memory is significantly faster than other forms of storage, which makes it ideal for those things that the computer needs immediately.
* However, memory is one of the most expensive components of a computer system, so we try to limit how much is available to reduce those costs.
* For this reason, there is normally significantly less memory in any computer system than there is storage.
* Memory is also volatile, which means that as soon as the program using memory closes (or the computer is shut down for any reason), all data and instructions stored in memory are erased.

**Secondary Storage**

* Storage, especially hard drive storage, is relatively inexpensive these days. It is also nonvolatile, meaning that anything we store there will be available the next time we log into our computer or boot up an application, barring physical damage to the drive.
* However, it is very slow both in terms of writing/recording data and retrieving data that is stored on the disk, which makes it less than ideal if users need to retrieve data from a large database.
* To improve a database's efficiency, we need to take advantage of both technologies: data we are most likely to use should be in memory when the database opens, while data we aren't likely to use remains in storage until we need it.
* This is where indexing fields comes into play.

**Indexing Fields**

* When we index fields, we tell the database engine that we want the values in those fields to be loaded into memory(RAM) when the database opens.
* Just like the index entries in a book tell you what page to look at for a specific term, map, or recipe, an indexed field tells the database where to look for related data when the user wants to retrieve data using an indexed value.
* Unindexed data remains on the hard drive until it is retrieved by the user, and only the relevant data is retrieved at that time.
* A gut reaction to finding out that data stored in memory can be accessed more quickly than data in storage is to try to index **everything**, so that everything is always immediately available to the user.
* Remember, though, that most systems have a very limited amount of memory.
* If you try to add too much content to memory, that extra content goes into a **swap disk**, a portion of the hard drive that most systems use for memory overflow.
* Because that process puts the data back on the hard drive, you end up back where you started, and you gain no efficiency with that approach.
* Instead, an experienced database designer will look at how each field is used and index only those fields that users are most likely to need.
* The database engine will automatically load all values in an indexed field into memory when the database is opened, making it faster to find those values.
* This approach does not prevent users from searching by first name rather than by last name, but the search will be slower because the database will have to search through storage rather than through memory.

**Default Indexes**

* By default, an RDBMS will automatically index key fields, including primary and foreign keys.
* These fields are essential for retrieving data across tables, and indexing those fields helps make queries more efficient.
* *While relational database guidelines state that the order of the columns is not important in terms of data storage, column order is important in terms of data retrieval.*
* Good design practice puts primary key fields first in a table, which allows the database to find them quickly without having to search through all of the columns in the table.
* When a table's indexed fields are loaded into memory(RAM), the database sorts the values in ascending order -- alphabetically if the indexed values are strings or numerically if the values are numbers.
* This means that the values are in a predictable order, making searches much more effective.
* You can see this when you retrieve data from a table without specifying a sort order: regardless of the order in which the records are added to a table, the results will be sorted by the primary key values by default.
* Indexed fields are stored in a predictable order, to make it easier to retrieve data based on those indexes.
* Every time a new indexed value is added to a table, the database must reorganize the data in that table to match the expected order.
* Adding records using an auto-incremented key helps improve write efficiency because it guarantees that each new record will be added at the end of the table.

**Unique Index**

* a primary key index by definition verifies that each value used in that field is unique within the table.
* A foreign key index does not check for the unique quality, but it does automatically enforce referential integrity.
* This means that whenever we enter a value as a foreign key, the database engine will confirm that the value references the primary key of a related table, which helps improve data integrity across tables.

**Non-Unique Index**

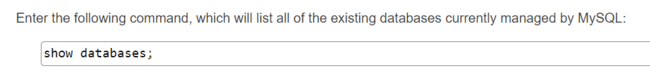
* An index on a last name field, however, should not be unique, mainly because many people have the same last name.
* If last name is frequently used for searches, the database designer can add a non-unique index to that field.

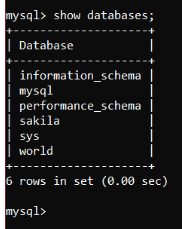
Data in indexed fields, including primary and foreign keys, is automatically loaded into memory when a database is opened, meaning that the database can quickly access those values to make searches more efficient.

In addition. database designers can identify additional fields to act as indexes, including especially fields that users are likely to access frequently.

**MySQL**

* MySQL is a **relational database management system (RDBMS)** that runs as a server providing multi-user access to multiple databases.





* MySQL Workbench is a database management application. It allows us to connect to a database server, view the server's databases, and execute queries.
* We can connect to any MySQL server by providing its hostname, port, and credentials. Once connected, we can:
  + Create databases and manage schema, including tables.
  + Manage security: users, roles, and permissions.
  + Backup and restore databases.
  + Execute SQL.
* **schema** is simply another term for database.

**SQL**

* SQL stands for Structured Query Language and it can be pronounced like "sequel" or using the letters S-Q-L.
* It is designed to work with any relational database management system (RDBMS), which includes MySQL Server, Microsoft SQL Server, and Oracle Database, among others.
* Because it works universally across RDBMSs, programming languages like Java, C#, and PHP support the use of SQL to retrieve data and manage data stored in an RDBMS.
* The term **query** technically refers to a statement that retrieves data from a database, but many developers also use the term "query" to refer to any SQL statement.
* GUIs like MySQL Workbench also typically use "query" in this way.

**Syntax**

* SQL follows a very rigid syntax, where each keyword and clause must be in a specific order.

**Execution flow**



**Select**

* The most common statement is a SELECT statement, which is used to retrieve data from one or more tables in a database, and it uses the following basic syntax:

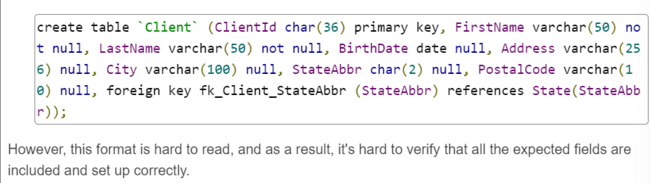
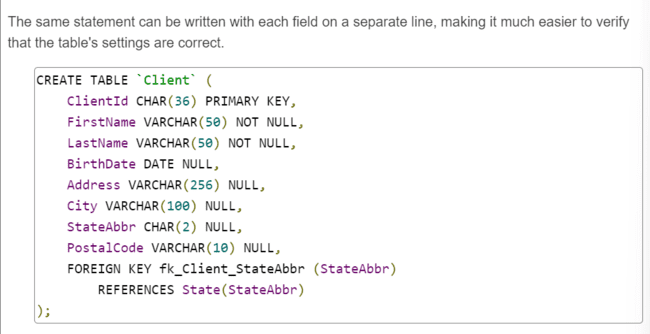
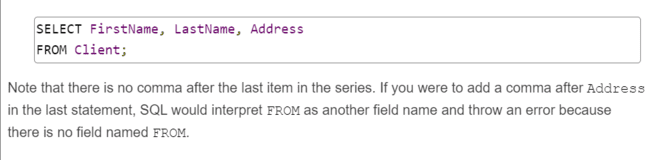


* This statement will retrieve the data from *field1, field2*, and *field3* whose values match the criteria statement in the WHERE clause in a table named *table1*. The results will be sorted by the values in *field1* and *field2*.
* Each of these clauses must be included in this order for the query to work correctly.

**Semi-Colon**

* MySQL requires the use of a semi-colon or /g at the end of each statement.

**Line Breaks and Indents**

* Neither line breaks nor indents are required within SQL statements, although we encourage you to use them to improve readability in the code.
* 
* 
* 
* SQL keywords are not case-sensitive so you can use SELECT, select, or even select interchangeably when keying out SQL statements.
* SQL uses commas to separate like objects in a series. For example, in the CREATE TABLE statement above, there is a comma after each field definition.
* 
* 
* 

**Data Definition Language (DDL)**

* Data Definition Language or **DDL**, is used to create, edit, and delete data structures.
* This includes the databases themselves, tables, columns, constraints, views, stored procedures, functions, and more.
* Data structures are not data. Instead, they define how data can be stored and accessed. Database data structures are also called **schema**.

**Data Manipulation Language (DML)**

* DML is used to read, create, edit, and delete data. Data includes the values stored in tables that live inside databases, and the table rows. The SELECT statement is DML. It reads data.

**Data Types**

* **Numeric types**
  + Whole numbers of various sizes; decimal numbers that are fast, suitable for scientific calculations, but not suitable for financial math; and decimal numbers that are suitable for financial math
* **Temporal types**
  + Years, dates, times, and dates combined with time
* **String types**
  + Fixed length character sequences, either ASCII or Unicode; variable length character sequences of all sizes
* **Large binary types**
  + Huge values represented as bytes or text that can grow to 4GB per field

**Floating Point**

FLOAT and DOUBLE are ISO standard floating point types. It is important to note that SQL floating point numbers have the same rounding and precision drawbacks as other programming languages and are not suitable for financial math.

**Fixed Point**

* DECIMAL is ISO standard. NUMERIC is an alias for DECIMAL.
* A DECIMAL is defined by precision, the total number of digits in the number on both sides of the decimal point, and scale, the number of digits to the right of the decimal point.
* Declaring a column DECIMAL(7,2) means it can store values in the pattern 00000.00 – 7 digits total, with 2 to the right of the decimal point.
* The decimal point is fixed, so 602341, while only having 6 digits, will not fit. 32452.686, however, will be reduced to fit. This will generally be done by truncation, but is dependent on the operating system.

**String Types**

* CHAR and VARCHAR are both declared with a single numeric value that defines the maximum characters allowed in a value.
* CHAR uses fixed storage. It is always allocated with its maximum size on disk. Shorter values are space-padded when they’re stored, and then spaces are removed when they're retrieved. The largest size allowed is 255 characters.
* VARCHAR uses variable storage. It only stores the characters in a value, up to the maximum size. The theoretical largest size of a VARCHAR is 65,535 characters.
* This is also the maximum number of bytes allowed in a table row. That means a VARCHAR can grow up to 65,535 characters minus the size of all other values in the row. So in practice, a VARCHAR will never grow to its largest maximum size.
  + Tables without variable-sized columns are accessed faster than those with a variable-sized column. Using all CHARs can increase performance.
  + If a column's values vary widely (say, street addresses or comments fields), using a VARCHAR can save significant space.
  + If the value is a fixed size, such as a Social Security number, a phone number, a state abbreviation, or other formalized format, always use CHAR.
* If our text is too big, we use TEXT. There are four flavors: TINYTEXT can store up to 255 characters depending on the character encoding (no improvement over VARCHAR), TEXT can store 65,535 characters, MEDIUMTEXT 16,777,215, and LONGTEXT 4,294,967,295 (which is 4GB).

**Date and Time Types**

* DATE stores year, month, and day with a range of '1000-01-01' to '9999-12-31'. Dates before CE 1000 cannot be stored and will need a different data representation. Quotes are used in date literals to distinguish the value from a math operation.
* TIME represents a time of day and an interval of time, in the format HHH:MM:SS.FFFFFF. F is a fractional second component. The range is -838:59:59.000000 to 838:59:59.000000. Fractional seconds are only valid inside of this range.
* DATETIME stores a date and time (time being only time of day here). It has the format ‘YYYY-MM-DD HH:MM:SS.FFFFFF’. Like DATE, it ranges from '1000-01-01 00:00:00' to '9999-12-31 23:59:59.999999'.

It's entirely possible to create a database schema without writing DDL. We can build a complete database with MySQL Workbench UI tools, where MySQL Workbench writes and executes the DDL for us

**Create Database**

Our first DDL statement is CREATE DATABASE.





**Create Table**

* Without a natural key, we need a surrogate primary key.
* A surrogate key doesn't have real-world meaning, so we let the database manage it with its auto-increment feature.
* Auto-incrementing keys are generated for us and are guaranteed to be unique. (They are not guaranteed to be contiguous.)

**Primary Key**

* Makes the column the primary key.
* Can only be included once per table in a column definition.
* If it's included, there's no need to specify NOT NULL; primary keys cannot be null.
* If a primary key covers multiple columns, it must be defined elsewhere.

**AUTO\_INCREMENT**

* Activates the SQL engine's auto-increment mechanism.
* Auto-increment columns must be numeric.
* If a number isn't provided for a row, the SQL engine generates a unique value.

**Null**

* Modifies a column so it can accept NULL values.
* If NULL and NOT NULL are omitted from a column definition, the default value is NULL.
* That means NULL is not strictly required, but is often included to indicate explicit intent.

**Not Null**

Modifies a column so it cannot accept NULL values.

**`Name`**

The identifier "Name" is a SQL keyword but it is not a **reserved** word. That means you can use it without quoting it, but we play it safe and add the quote literals.

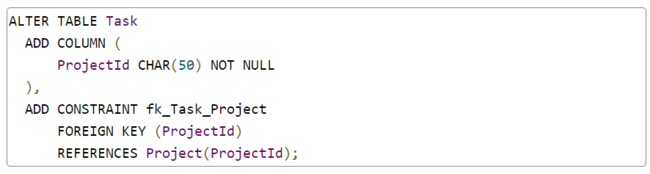
**DEFAULT <value>**

* The DEFAULT keyword modifies the IsActive column. If a value is not provided, IsActive will use the DEFAULT value 1.
* IsActive is a BOOL. In MySQL, BOOL is an alias for TINYINT(1), where the value 0 is false and everything else is true.
* Default values must be constant. They can't be based on a calculation except in one special case: the TIMESTAMP or DATETIME data type can base their default value on the function CURRENT\_TIMESTAMP.

**Alter**

To change an existing table, we use the ALTER TABLE statement. It can:

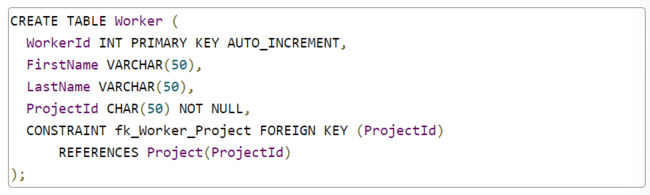
* Add or remove columns.
* Change an existing column definition: data type, nullability, or DEFAULT values.
* Add or remove constraints (a foreign key is a constraint).



* Constraints are database structures that limit (constrain) values or make values faster to retrieve.
* FOREIGN KEY is a referential constraint.
* It enforces that a value must exist in a source table before it can be added to a child.
* The DBMS will not allow you to create a foreign key constraint on a non-existent primary key in another table, to enforce the Consistency property of an ACID database.

**CREATE TABLE**

You can add a FOREIGN KEY in a CREATE statement. SQL is flexible. You don't have to wait for an ALTER statement.

****

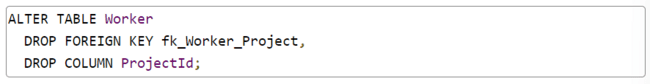
**DROP**

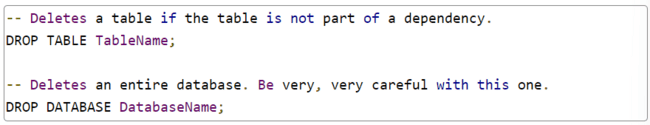
There are three big, over-arching DDL actions:

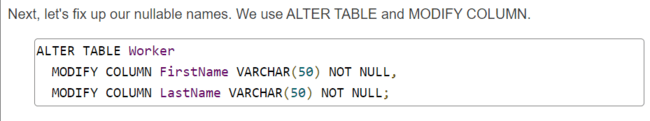
* CREATE builds schema from scratch.
* ALTER edits existing schema.
* DROP deletes schema.

We want to DROP the ProjectId column from Worker.

* Unfortunately, it would fail if we tried.
* Worker's ProjectId is part of a FOREIGN KEY constraint.
* The SQL engine sees the dependency and prevents the DROP.
* If we want to DROP ProjectId, we must first DROP the constraint.







**ALTER Tables with Existing Data**

* If a table contains data, it may not be possible to MODIFY a column.
* If you declare a column NOT NULL and your data contains nulls, the MODIFY will fail.
* Likewise, if you MODIFY a column to use a more restrictive data type, say, VARCHAR(10) versus VARCHAR(50), and your column contains values longer than 10 characters, your MODIFY will fail.

To force the modification, update the data to conform to restrictions and then MODIFY.

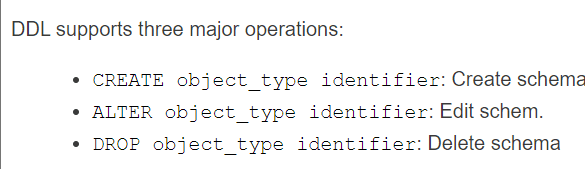
**Naming Constraints**

If you don't provide a name for a constraint, the SQL engine will generate a default.

**Many-to-Many Relationships**

We want each worker to be part of one or more projects and each project to be associated with one or more workers.

* The secret is a **bridge table**, also called an **associative entity.**
* A bridge table includes a foreign key from each table it bridges or associates.
* By including a key from two or more tables, the bridge table models a relationship between concepts versus a concept that stands on its own.



**Data Manipulation Language (DML)**

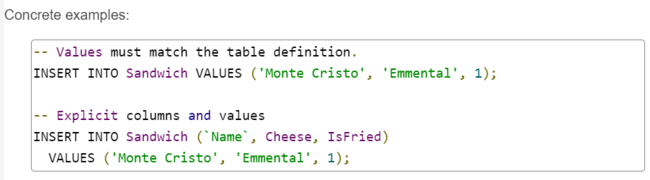
1. INSERT: Adds new data to a table
2. UPDATE: Edits existing data in a table
3. DELETE: Deletes data from a table

**INSERT**

The INSERT statement is used to insert new rows into a table.

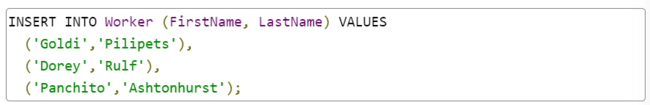


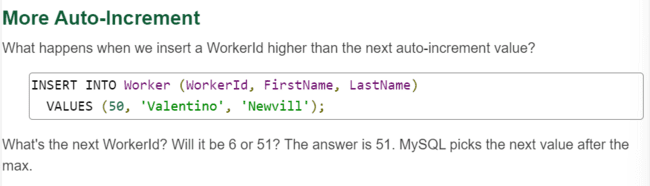
* The column list is optional.
* If columns are omitted, the value list must match columns in the table definition, exactly, in order.
* If the column list is included, the value list must match it.
* In either case, the number of columns and values must match and their order must match.
* Even though it's optional, it's a good practice to include the column list because tables change over time and your INSERT assumptions will grow stale.



* If you omit the value for an auto-incremented column, the database engine generates the value for you.
* If a column is defined *NOT NULL* and does not have a *DEFAULT*, you must provide a value in your insert statement.
* If a column is a non-null foreign key, you must provide a value that exists in its source table.
* For example, in ProjectWorker, both the ProjectId and WorkerId must be set to values that exist in the Project and Worker tables.
* If the values are omitted or don't exist in the related tables, the INSERT will fail.

**Inserting Multiple Rows**



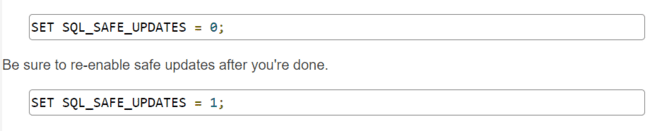


**UPDATE**

The UPDATE statement is used to change record values in a table.



* UPDATE TableName limits changes to the named table.
* One or more columns are assigned values, separated by commas, following the SET keyword.
* [Value] can be a value literal, another column, or even a query result.
* The WHERE [Condition] clause is identical to a SELECT's WHERE clause. It is a boolean expression, using AND, OR, or any boolean operators in any combination to limit records to be modified.
* If you want to update every row in a table, omit the WHERE clause.
* You can disable safe update configuration with a query.

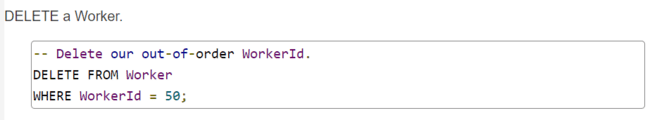


**DELETE**

The DELETE statement is used to delete rows from a table. Deletes are all or nothing. There is no partial-row delete option.



* DELETE FROM TableName limits row removal to the named table.
* Just like SELECT and UPDATE, the WHERE [Condition] clause evaluates to a boolean. If the result is true for a record, the record is deleted. If not, it is ignored.



**SQL Syntax**

SQL syntax is designed to be forgiving. Rules include:

* Queries are terminated with the semicolon ';' character.
* SQL is not whitespace sensitive. A query can have as much whitespace between keywords and identifiers as you like, including line breaks. Optimize for readability – it is poor style to jam a large query onto a single line. Use line breaks to separate a query's significant concepts and indent related items. This is the style we use in this course.
* By default, SQL is not case-sensitive. The keyword SELECT is the same as Select or select. You may use any style you like in your assignments as long as you are consistent. It is common practice to put SQL keywords (the words that tell the DBMS what you want it to do) in all caps, while database-specific words like table and column names are written in lower-case.
* Beyond syntax, SQL's string comparisons are also case-insensitive by default. The value 'Flower' is the same as 'flower'. Case sensitivity can be enabled via configuration.
* String/text values are surrounded by single quotes. If a string contains a literal single quote, escape it with a second single quote: e.g., 'I''m a string.'
* Any query or SQL command should end with a semicolon. A semicolon is optional in some DBMSs when you want to run a single query, but if you have a series of queries or commands in a single script, the semicolon tells the DBMS where one command ends and the next one begins. It's a good habit to add a semicolon at the end of each query, even if it isn't always necessary.

**SELECT Queries**

**Single Table SELECT**

The simplest query is a *SELECT* statement written for a single table.



**SELECT \***

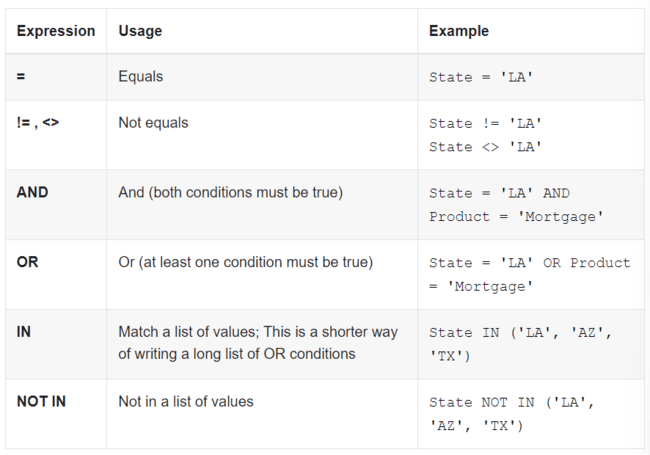
If you want all columns, you don't have to list them explicitly. SQL gives us a shortcut. Use the asterisk '\*' symbol:



**WHERE Clause**

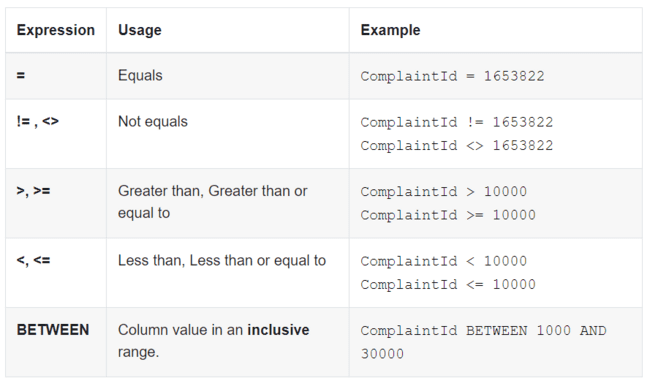
* The WHERE clause is a conditional expression, which means it resolves to a boolean, true or false.
* If the expression is true for a record, the record is included in the result. If not, it is excluded. We can build complex boolean expressions using AND, OR, and NOT (!).

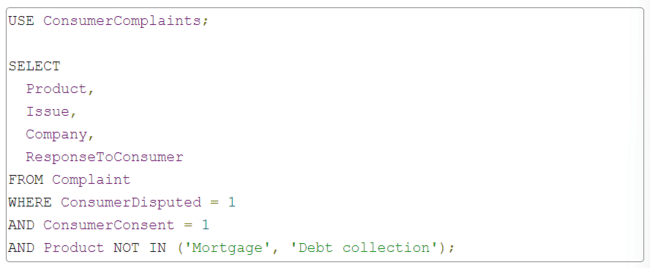
**WHERE Operators**

****

**Filtering Numbers**

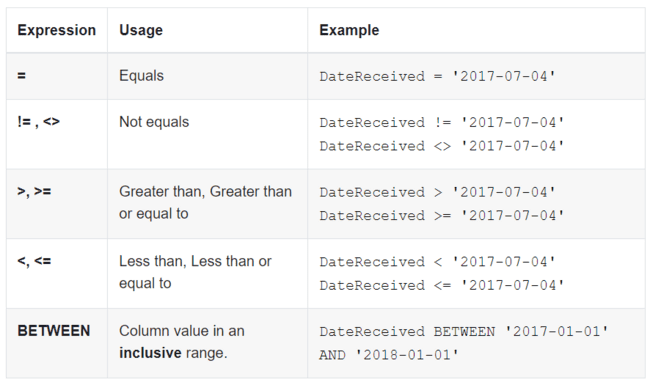
* Different columns can hold different types of data and the types may have different conditional operators.
* Numeric columns can be filtered using math comparison operators like < (less than) or >= (greater than or equal).
* There's also a keyword BETWEEN for value ranges.





**Filtering Dates**

* Relational databases store dates as a specific data type.
* The date type is small and fast, two attractive qualities for a data storage system.
* Dates can use many of the numeric operators; conceptually, we can think of a date as bigger or smaller than another date.



Date literals, e.g., '2017-07-04', are delimited with single ticks like strings.

They are not strings under the hood, however.

The SQL execution engine converts them to the date type if they have the proper format.

The format 'yyyy-MM-dd' is understood by MySQL and most other databases.

**Pattern Matching Text**

* SQL can match patterns in strings and text. The LIKE operator works from a string example. If a column value matches the example, the record is included.
* The example string may contain characters with special meaning, which differentiates LIKE from simple string comparisons. Special characters include:

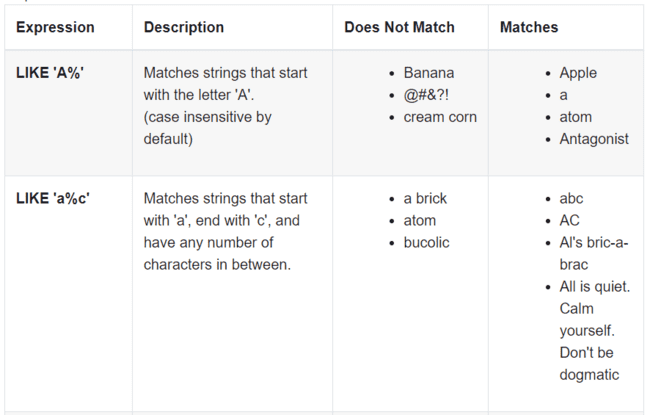
**% (percent)**

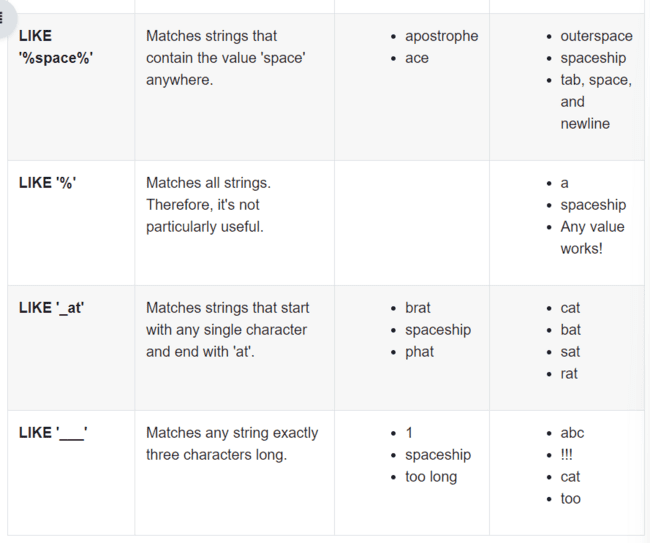
Matches any number of characters, including no characters.

**\_ (underscore)**

Matches any single character.

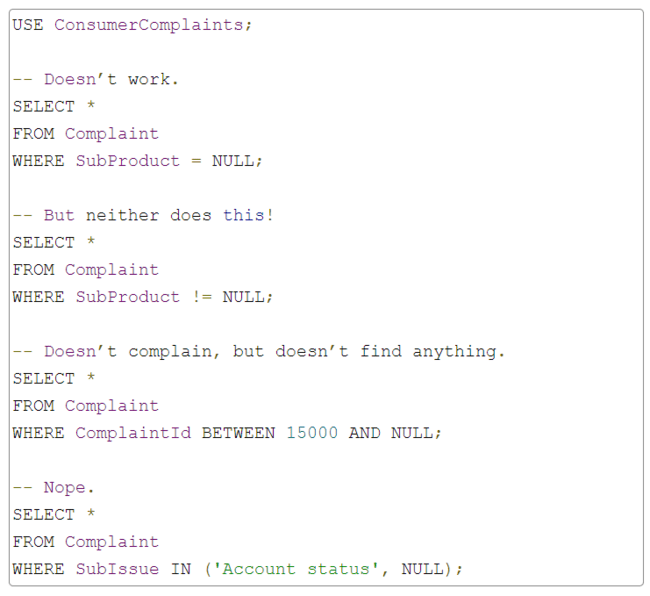
* If you want to match the literal value '%' or '\_', escape them with a backslash. For example: '\%' or '\\_'.



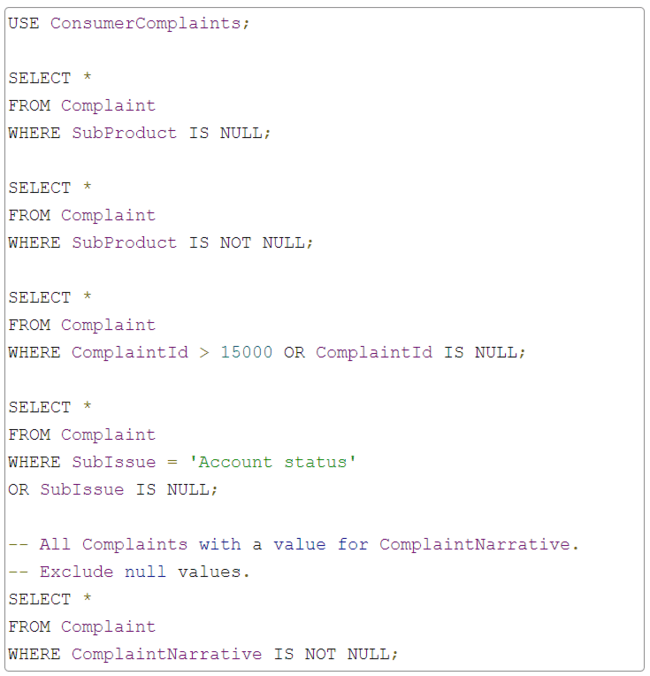


**NULL – the “Billion-Dollar Mistake”**

* The value NULL is special. It represents an unset value or missing information.
* Any table column can be configured to accept or reject NULL values, even columns that store numbers.
* Unfortunately, NULL is impervious to many operators in the WHERE clause.



* To find NULL values, we have to use the special operator IS.
* Then we can express that a value IS NULL or IS NOT NULL. Our queries can be rewritten:

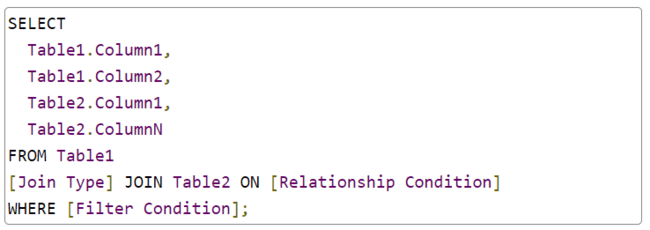


**Join Queries**

**JOIN**

* The JOIN clause is an optional clause in a SELECT statement.
* It expands a SELECT so it can retrieve results from more than one table and express relationships between rows.
* Rows from one table are joined to rows from another table and their values are combined in a single result.

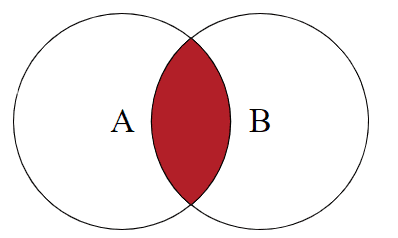
A JOIN clause follows the FROM clause and precedes the WHERE in a SELECT.

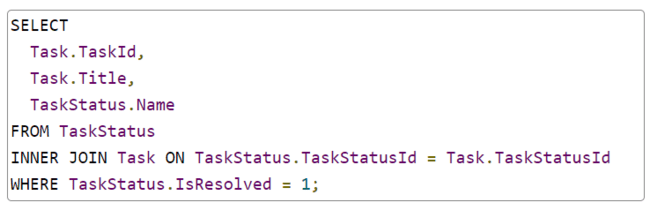


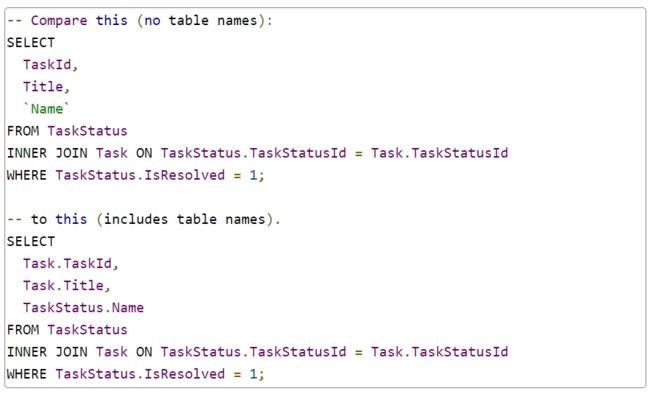
* JOIN [Table2] adds the table, Table2, to the query and makes its rows available for retrieval and filtering.
* ON [Relationship Condition] defines how rows in one table relate to rows in another.
* [Join Type] modifies the JOIN. It determines how unmatched rows are handled. Valid values include INNER, LEFT OUTER, RIGHT OUTER, FULL OUTER, and CROSS.

**INNER JOIN**

* An INNER JOIN returns a result only when rows from both tables match on their relationship condition.
* Visually, if we have tables A and B, the query results are the intersection of rows that satisfy the join condition.
* If a row from A doesn't match a row from B, it isn't included, and vice versa.







* The INNER keyword is also optional. If we omit it, the SQL engine assumes an INNER join. INNER JOIN is the default.

**OUTER JOIN**

**INNER JOIN Limitations**

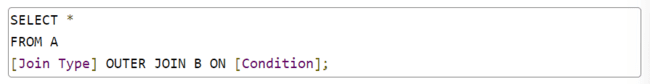
INNER JOIN returns a record for each row match between joined tables. What happens when a row exists in one table but doesn't match a row in the other?

OUTER JOINs are forgiving. They return a record even when rows don't match in joined tables. There are three flavors:

* LEFT OUTER JOIN
* RIGHT OUTER JOIN
* FULL OUTER JOIN

The left or right designation indicates where a table is mentioned in relation to the JOIN clause. If a table is mentioned before a JOIN, it is "left" of the JOIN. If it is mentioned after, it is "right" of the JOIN.

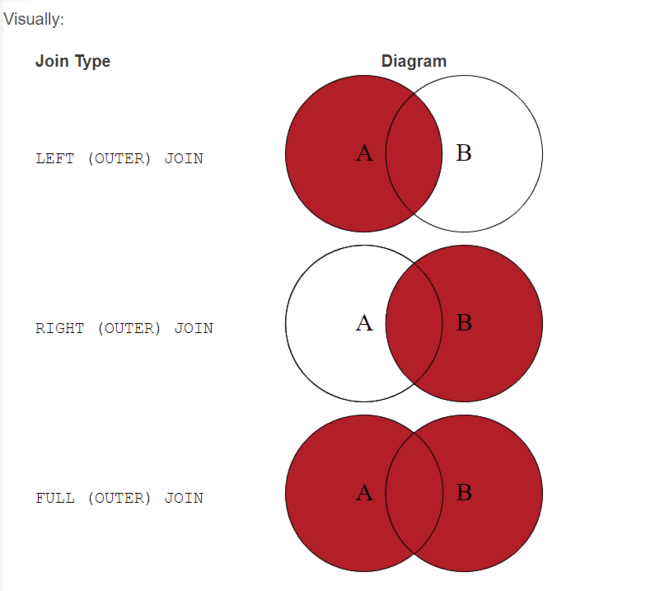
Consider this:

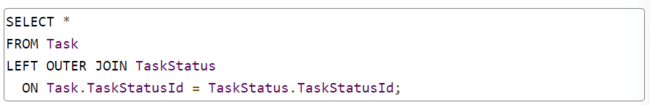


When the [Join Type] is LEFT, the results include "everything from table A and whatever matches from table B."

RIGHT results include "everything from table B and whatever matches from table A." FULL OUTER JOIN results are "everything from both tables regardless of match."

Just like INNER, the OUTER keyword is optional. The LEFT, RIGHT, or FULL keywords are not optional. (If you omit both LEFT and OUTER, the SQL engine would assume an INNER JOIN.)





**Self-JOIN**

**Aliases**

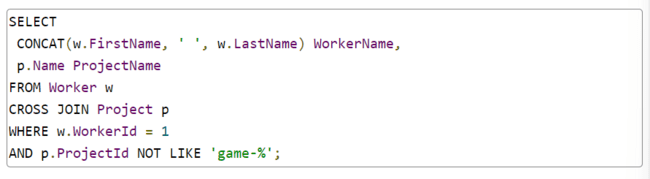
Self-referential relationships are unusual, but not that unusual. They're useful for homogeneous data organized in a hierarchy. Examples include:

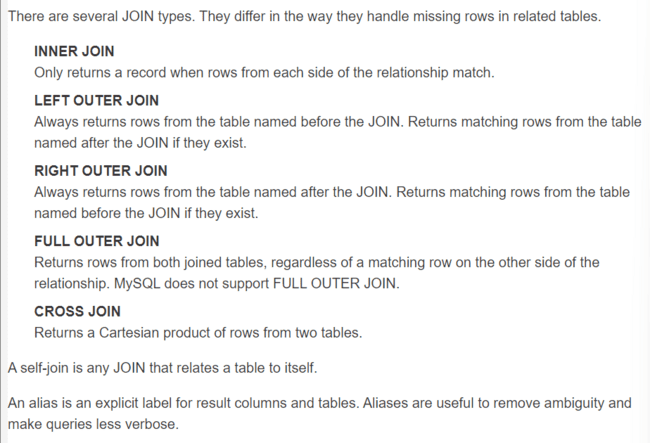
* File system folders – Each folder lives inside another, root folder excluded
* Comment threads – Comments may be a response to another comment, which in turn may be a response to a comment...
* Software UI menus – The File menu opens a list of menu options, select one and it opens a list of menu options, etc.

**Cross JOIN**

CROSS JOIN does not use an ON clause because it does not match on a condition.

Instead, CROSS JOIN creates a Cartesian product, with every possible combination of rows between the joined tables included in the results.





**Sorting**

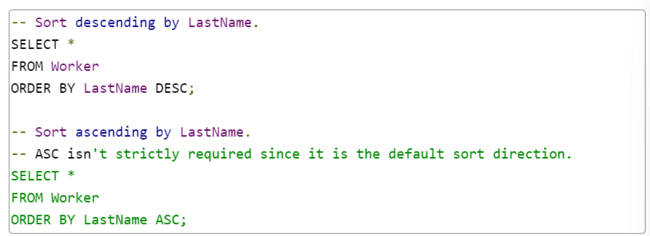
**ORDER BY**

* The ORDER BY clause is an optional extension to the SELECT statement.
* The ORDER BY keywords are followed by one or more columns. Results will be sorted by the columns' values.
* Optionally, an ORDER BY clause includes the sort direction, either ascending or descending order.
* The default direction is ascending.
* Results can be sorted by any column, not just columns retrieved in the SELECT value list.
* Sorting occurs before SELECT values are evaluated, so the SQL engine has access to all columns.

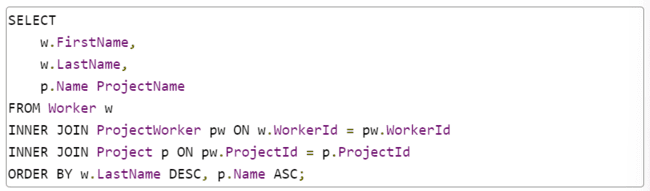
**Sort by a Single Column**



* The default sort direction is ascending. Our Workers are sorted from last name "Achromov" to "Zorzi." To reverse the direction,
* we must specify it explicitly. The keyword DESC sorts descending.
* ASC sorts ascending.

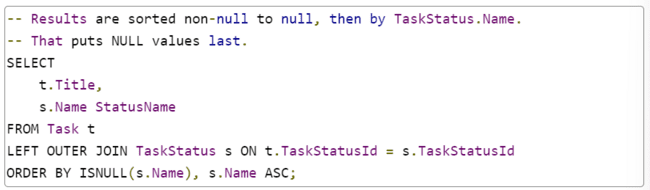


**Sort by Multiple Columns**

****

**Handling NULL**

* The MySQL engine had to make a choice.
* They chose to put NULL first. If you dislike NULLs first, you can force their order by adding an ORDER BY condition.
* The following query sorts NULLs last.



**LIMIT**

The LIMIT clause is an optional extension to the SELECT statement. It restricts (or limits) the records returned from a query.



LIMIT is the last syntax element in a SELECT statement.

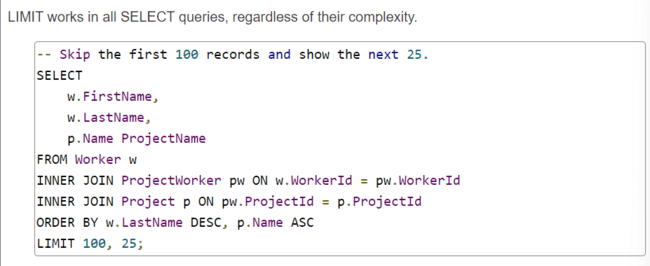


There is no offset (remember that programmers start counting from zero) and we grab 10 rows.

Offset by 10 rows and grab 10.







**Grouping**

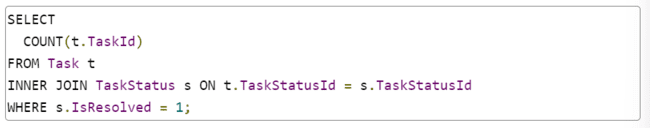
**Aggregates**

* There are 12 or more SQL aggregate functions, depending on the vendor. The most common and universally supported are:
* **COUNT**
  + Counts the number of non-NULL values in a set; works on any non-NULL value
* **SUM**
  + Sums values in a set; values must be numeric
* **AVG**
  + Calculates the average of values in a set; values must be numeric
* **MIN**
  + Determines the minimum value in a set; values must be comparable
* **MAX**
  + Determines the maximum value in a set; values must be comparable



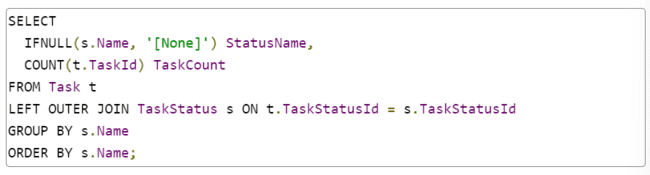
* Each of our five aggregate functions requires one argument: the source of values to be aggregated.
* It can be a field or any value expression.
* The \* argument in COUNT(\*) is special. It tells the SQL engine to count records, not values.

We can aggregate any value. The value can come from a joined table or from a result filtered with WHERE.



**GROUP BY**

* GROUP BY is an optional clause in a SELECT statement.
* It partitions a result into groups.
* GROUP BY can be used with aggregate functions to compute a value per group instead of computing across the entire result.
* GROUP BY is placed after WHERE, if it's present, and before ORDER BY.

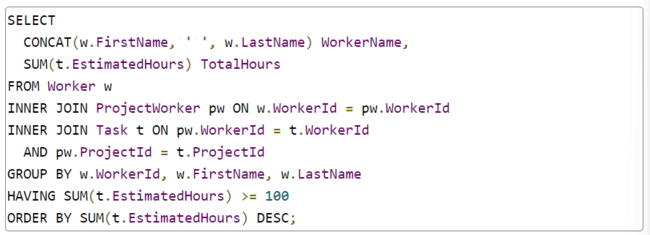


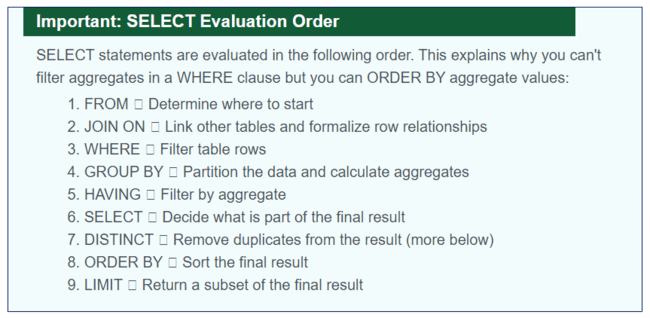
* There are a few nuances in the query:
* We use LEFT OUTER JOIN to get all Tasks. An INNER JOIN would eliminate NULL TaskStatusIds.
* We sort by s.Name because it's the value that drives IFNULL(s.Name, '[None]'). We could also sort by COUNT(s.TaskId).
* Because s.Name can be NULL, we provide a replacement value so it's easy to display the NULL status.
* Aliases provide meaningful names for aggregate values.

**HAVING**

HAVING is an optional clause in a SELECT statement, and it can only be included when a GROUP BY clause exists.

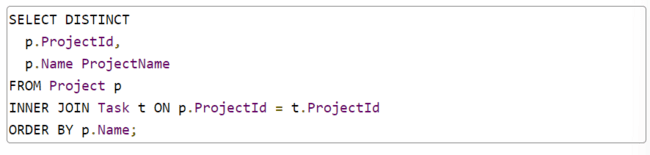
HAVING is followed by a boolean expression, just like a WHERE clause, but the expression includes comparisons against aggregate values.





**DISTINCT**

* DISTINCT is an optional keyword that may appear in the SELECT value list.
* If present, it removes duplicate records from the query result.



* Most uses of DISTINCT can be accomplished by grouping data with GROUP BY.
* In fact, MySQL uses GROUP BY optimizations to optimize DISTINCT queries.

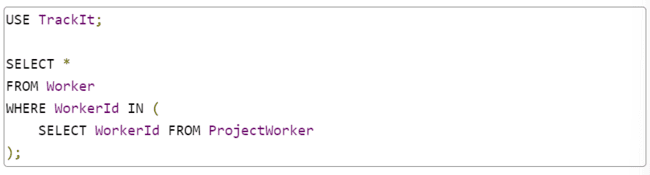
**Subqueries**

**Views**

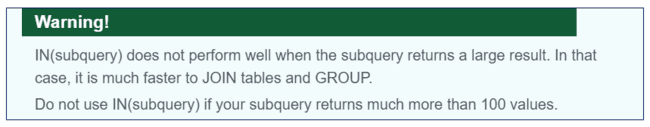
* A **subquery** is a syntactically correct, complete query that is embedded in another query to produce a value or tabular result set.
* Removed from its parent, a subquery is still valid, though it may use values from its parent to establish context.
* SQL also supports code reuse.
* A **view** is a named query that is stored in the database. It can be treated like a table.

**IN**

* Values in an IN operator can come from a query.
* Say we want to find all Workers who are assigned to a Project.
* We could grab all of the ProjectWorker.WorkerIds with a query and use the values in an IN clause.



* If a value occurs more than once in an IN, it is ignored.



**Table**

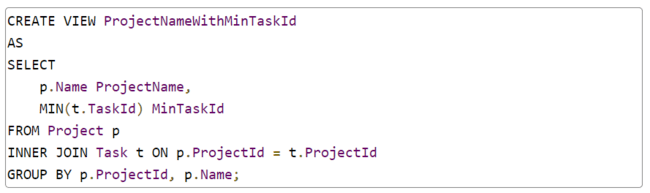
* Any table named in a query can be replaced by a subquery.
* We can build a secondary SELECT on top of a subquery or JOIN a subquery to a table (or another subquery!).

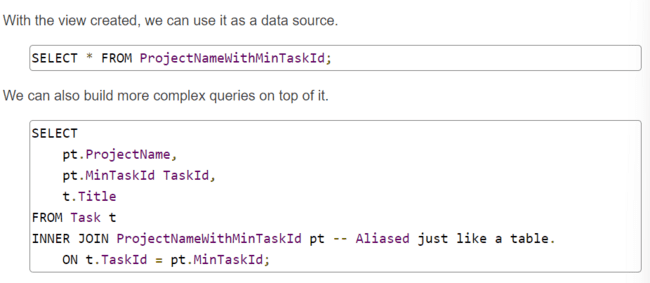
**Value**

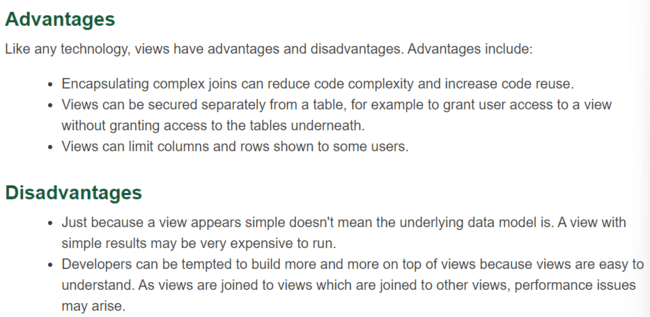
* Any field or calculated value can be replaced by a subquery.
* In effect, the subquery becomes the calculation.

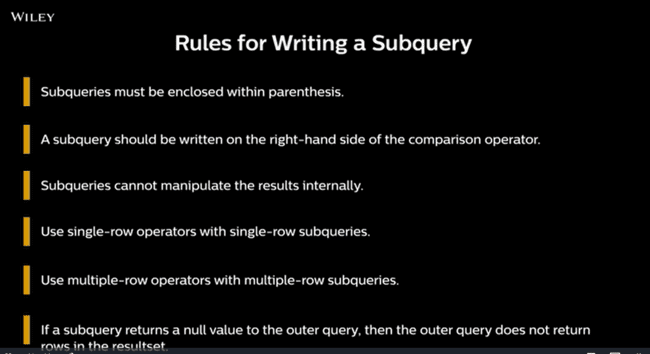
**Views**

* A view is a named query that's stored in a database.
* Once it's stored, other queries can build on it.
* A view can be treated like a table anywhere in a SELECT statement.
* We can also think of it as a named subquery.









* From point 3 it is confirm that we cannot use order by in the inside query.

