Day-Two

Notes

Warmup

```
- everyone of us have to share one thing that surprised us and one question.
```

Virtualization

```
- Virtualization in computer science refers to the process of creating a
virtual (rather than actual) version of something, including virtual
hardware platforms, storage devices, and computer network resources. It's a
core technology behind cloud computing, enabling more efficient utilization
of physical resources. Here’s an overview of different types of
virtualization:
```

1. Hardware Virtualization (Server Virtualization):

- **Definition**: Involves creating virtual machines (VMs) that act like real computers with their operating systems (OS) and applications.
- **Usage**: Allows multiple OS to run on a single physical machine, improving resource use and reducing the number of physical servers required.
- **Key Software**: Hypervisors (e.g., VMware, Hyper-V, KVM, Xen) manage these VMs.
- **Benefits**: Reduced hardware costs, simplified IT management, increased flexibility and disaster recovery capabilities.

2. Operating System Virtualization:

- **Definition**: The OS is virtualized, allowing different applications to run in isolated environments called containers.
- **Usage**: Lightweight containers (e.g., Docker) are popular in microservices architectures, where multiple services need to run in isolation but share the same OS.
- **Benefits**: Efficient resource sharing, fast deployment, and high portability.

3. Network Virtualization:

- **Definition**: Abstracts networking hardware and functionality into a software-based administrative entity.
- **Usage**: Virtual LANs (VLANs) and software-defined networking (SDN) are examples. This helps in dynamically managing and routing traffic more efficiently.
- **Benefits**: Improved network performance, scalability, and flexibility.

4. Storage Virtualization:

- **Definition**: The abstraction of physical storage resources to appear as a single storage device to the user or applications.
- **Usage**: Implemented in Storage Area Networks (SANs) to pool storage from multiple devices.
- **Benefits**: Increased storage utilization, better backup, and disaster recovery management.

5. Desktop Virtualization:

- **Definition**: Separates the desktop environment from the physical machine, allowing users to access their desktops remotely from any device.
- **Usage**: Virtual desktop infrastructure (VDI) like Citrix or VMware Horizon.
- **Benefits**: Centralized management of desktops, improved security, and reduced hardware costs for individual users.

6. Application Virtualization:

- **Definition**: Applications are encapsulated in a way that they can run independently of the underlying operating system.
- **Usage**: Allows applications to be run on different operating systems without compatibility issues. For instance, using Wine to run Windows applications on Linux.
- **Benefits**: Increased flexibility and the ability to run legacy applications on modern systems.

7. Virtualization in Cloud Computing:

- **Definition**: Cloud computing leverages virtualization to deliver scalable, on-demand resources over the internet.

- **Usage**: IaaS (Infrastructure as a Service) platforms like AWS, Google Cloud, and Microsoft Azure use virtualization to provide computing resources.
- **Benefits**: Elasticity, pay-as-you-go pricing models, and rapid deployment of services.

Benefits of Virtualization:

- **Cost Efficiency**: Reduced need for physical infrastructure.
- **Scalability**: Easily scale computing resources up or down.
- **Resource Optimization**: Maximizes resource utilization by allowing multiple systems to run on the same physical machine.
- **Disaster Recovery**: Easier to replicate and recover systems.
- **Isolation**: Virtual environments are isolated, making them more secure and less prone to system crashes.

Virtualization has become integral to modern IT infrastructure, especially in cloud computing, data centers, and development environments where efficiency, flexibility, and resource management are key.

From a developer's perspective, virtualization offers several key benefits:

1. **Environment
2. Isolation**: Developers can run multiple projects with different dependencies in isolated VMs or containers, preventing conflicts between tools and libraries.
3. **Cross-Platform Testing**: It allows easy testing of applications across different OS (e.g., Linux, Windows) without needing multiple physical machines.
4. **Reproducibility**: Developers can replicate production environments for consistent testing and debugging, reducing "it works on my machine" issues.
5. **Snapshots & Rollbacks**: Virtualization allows you to take snapshots and revert to previous states, which is invaluable for testing and quick recovery after errors.
6. **Collaboration**: Pre-configured VMs or container images can be shared with team members, ensuring everyone works in the same environment.
7. **Resource Efficiency**: Virtual machines allow flexible scaling of resources (RAM, CPU) as project needs change, reducing hardware costs.
8. **Security**: Isolated environments protect the main system from potentially harmful code during testing or experimentation.
9. **Faster Setup**: Virtual environments allow for quick setup and teardown, making it easy to start new projects or run different configurations without manual setup.

In summary, virtualization enhances development by providing isolated, scalable, and flexible environments, improving testing, collaboration, and resource management.

$/ %/ "#"

-In Linux, the `$` and `%` symbols represent the shell prompt, indicating the user's privilege level when interacting with the command line:

`$` — Regular User Prompt

- **Meaning**: The `$` symbol at the end of the shell prompt indicates that you are operating as a **regular (non-root) user**. This user has limited permissions and can only perform tasks allowed by the system's permission settings.
- **Example**:

```
user@hostname:~$
```

`%` — csh/tcsh Shell Prompt for Regular Users

- **Meaning**: In **csh** or **tcsh** (other types of shells), `%` is used as the prompt symbol for regular users instead of `$`.
- **Example**:

```
user@hostname:~%
```

`#` — Root User Prompt

- **Meaning**: The `#` symbol is shown when you are logged in as the **root user** (or using elevated privileges). Root has unrestricted access to the system, including the ability to modify system files and install software.
- **Example**:

```
root@hostname:~#
```

In general, `$` and `%` denote **limited privileges**, while `#` represents **administrator (root) access**.

Prompts

```
Here’s a list of the most commonly used commands in Linux Bash, categorized by their function:
```

1. File and Directory Operations

- `ls`: List the contents of a directory.
- `cd`: Change the current directory.
- `pwd`: Print the current working directory.
- `mkdir`: Create a new directory.
- `rmdir`: Remove an empty directory.
- `rm`: Remove files or directories (`rm -r` for recursive removal).
- `cp`: Copy files or directories.
- `mv`: Move or rename files or directories.
- `touch`: Create an empty file or update a file's timestamp.
- `find`: Search for files or directories in a directory hierarchy.

2. File Viewing and Editing

- `cat`: Concatenate and display file contents.
- `more`: View file contents one screen at a time.
- `less`: View file contents with backward/forward navigation.
- `nano`: Simple text editor in the terminal.
- `vi` or `vim`: Advanced text editor.
- `head`: Display the first lines of a file.
- `tail`: Display the last lines of a file (`tail -f` to follow file updates).

3. File Permissions and Ownership

- `chmod`: Change file permissions.
- `chown`: Change file ownership.

- `chgrp`: Change group ownership of a file or directory.

4. Process Management

- `ps`: Display currently running processes.
- `top`: Display real-time system resource usage and processes.
- `kill`: Terminate a process by PID.
- `killall`: Kill processes by name.
- `bg`: Resume a job in the background.
- `fg`: Bring a background job to the foreground.
- `jobs`: List active background jobs.
- `nohup`: Run a command immune to hangups (keep running after logout).

5. File Compression/Decompression

- `tar`: Archive and extract tar files (`tar -xvf`, `tar -cvf`).
- `gzip`: Compress files.
- `gunzip`: Decompress `.gz` files.
- `zip`: Compress files into a zip archive.
- `unzip`: Extract files from a zip archive.

6. Disk and System Information

- `df`: Display disk space usage.
- `du`: Estimate file space usage.
- `free`: Display memory usage.
- `uname`: Show system information (`uname -a` for detailed info).
- `uptime`: Show how long the system has been running.
- `whoami`: Display the current logged-in username.

7. Networking

- `ping`: Test network connectivity to a host.
- `ifconfig` or `ip`: Display or configure network interfaces.
- `netstat`: Show network connections, routing tables, and interface statistics.
- `wget`: Download files from the web.

- `curl`: Transfer data from or to a server (supports multiple protocols).
- `ssh`: Secure shell login to a remote machine.
- `scp`: Secure copy (transfer files between hosts).
- `ftp`: Transfer files using File Transfer Protocol.

8. Package Management (varies by distribution)

- **Debian/Ubuntu**:
  - `apt-get`: Install, upgrade, or remove packages.
  - `apt`: More modern package manager command.
  - `dpkg`: Low-level package manager.
- **RedHat/CentOS/Fedora**:
  - `yum` or `dnf`: Install, update, or remove packages.
  - `rpm`: Low-level package manager.
- **Arch Linux**:
  - `pacman`: Package manager for Arch-based systems.

9. User and Group Management

- `useradd`: Add a new user.
- `usermod`: Modify an existing user.
- `passwd`: Change user password.
- `groupadd`: Add a new group.
- `groups`: Display groups a user belongs to.

10. Input/Output Redirection and Pipes

- `>`: Redirect output to a file (overwrites).
- `>>`: Append output to a file.
- `<`: Redirect input from a file.
- `|`: Pipe output of one command as input to another.
- `tee`: Read from standard input and write to standard output and files.

11. Permissions and Ownership

- `chmod`: Change file permissions (e.g., `chmod 755 file`).

- `chown`: Change file owner and group.
- `chgrp`: Change group ownership.

12. Archiving and Compression

- `tar`: Archive files (`tar -cvf archive.tar files`, `tar -xvf archive.tar`).
- `gzip`: Compress files (`gzip file`).
- `gunzip`: Decompress files (`gunzip file.gz`).
- `zip/unzip`: Compress and decompress `.zip` files.

13. Searching

- `grep`: Search text using patterns.
- `find`: Search for files in a directory hierarchy.
- `locate`: Find files by name (uses a database).
- `which`: Locate a command's executable.

14. Permissions

- `chmod`: Modify file permissions.
- `chown`: Change file ownership.
- `chgrp`: Change group ownership.

```
These commands cover the basics for file handling, system monitoring,
networking, and package management, which are essential for daily tasks in a
Linux environment.
```

Git Version Control

Git is a distributed version control system (VCS) designed to track changes in source code and other files during software development. It allows multiple developers to collaborate on a project, manage changes efficiently, and maintain a history of all modifications.

Key Concepts of Git and Version Control:

1. **Version Control**:

- **Definition**: Version control systems track changes to files over time, allowing you to revert to earlier versions, collaborate with others, and maintain a full history of your project.
- **Purpose**: Helps developers manage code versions, avoid conflicts when working together, and ensure that changes can be reviewed, merged, or rolled back.

2. **Why Use Git (or Version Control)?**
   - **Collaboration**: Multiple developers can work on the same project simultaneously without overwriting each other's work.
   - **History**: Git stores a history of changes, so you can always look back, understand who made what changes, and when.
   - **Backup**: Changes are saved locally and can be pushed to remote repositories (e.g., GitHub, GitLab), providing a backup.
   - **Branching**: Developers can create branches to work on features or fixes independently, then merge them back into the main codebase when ready.
   - **Reverting**: If something goes wrong, Git allows you to roll back to a previous version of the project.

Core Components of Git:

1. **Repository (Repo)**:
   - A directory containing all the files and history of a project. It tracks all changes to the project files.
2. **Commit**:
   - A snapshot of the project at a specific point in time. Commits represent changes made to the files, along with a message describing what was done.
3. **Branch**:
   - A branch is a separate line of development. The main branch is typically called `master` or `main`. Developers can create new branches to work on features independently.
4. **Merge**:
   - Merging combines changes from one branch into another, typically merging feature branches back into the main branch after review.
5. **Clone**:
   - A copy of an existing repository from a remote location (e.g., GitHub) to a local machine.
6. **Pull**:
   - Fetches changes from a remote repository and merges them into your local copy.
7. **Push**:

- Uploads local changes to a remote repository, allowing others to access and collaborate.

8. **Staging Area**:
   - A space where changes are placed before committing. You can stage specific files or parts of files before creating a commit.
9. **Diff**:
   - Shows the differences between versions of files or commits. This is useful for reviewing changes before merging or committing.

Common Git Commands:

- `git init`: Initialize a new Git repository.
- `git clone <url>`: Clone a remote repository to your local machine.
- `git add <file>`: Add file(s) to the staging area for the next commit.
- `git commit -m "message"`: Commit staged changes with a descriptive message.
- `git push`: Push committed changes to a remote repository.
- `git pull`: Fetch and merge changes from a remote repository into your local copy.
- `git status`: Show the status of the working directory and staged changes.
- `git log`: Show the history of commits.
- `git branch`: List, create, or delete branches.
- `git checkout <branch>`: Switch to another branch.

Popular Git Hosting Services:

- **GitHub**: The most popular platform for hosting Git repositories, often used for open-source projects.
- **GitLab**: A Git-based platform offering additional DevOps tools like CI/CD pipelines.
- **Bitbucket**: Another platform for Git repositories, commonly used in enterprise settings.

Benefits of Git and Version Control:

- **Collaboration**: Simplifies team collaboration and prevents file overwrites.
- **Traceability**: Provides a clear history of changes for auditing and review.
- **Backup**: Keeps code backed up in multiple locations.
- **Branching & Experimentation**: Enables safe experimentation with new features on separate branches without affecting the main codebase.

- **Open Source & Widely Used**: Git is free, open-source, and the most widely used VCS in the industry.

  Git is an essential tool for software developers to manage code effectively, especially in team environments.

Git Configuration Levels

1. **System (`--system`)**: Applies to every user on the system and is stored in the system Git configuration file (usually in `/etc/gitconfig`).
2. **Global (`--global`)**: Applies to a specific user across all repositories. The settings are stored in the user's home directory (usually in `~/.gitconfig`).
3. **Local**: Specific to the current repository only. The settings are stored in the repository's `.git/config` file.

In Class Notes or Exercise

1. check git version

```
$       git --version
```

2. git stores configuration information in

```
$       ~/.gitconfig
```

3. each repository also has a local confiuration file named in

```
$       ~/.git/config
```

4. example of setting config

```
"Globally"
$ git config --global user.name "Your Name"
$ git config --global user.email "your.email@example.com"

"Locally"
$ git config user.name "Your Name"
$ git config user.email "your.email@example.com"
```

```
"List them"
$ git config --list

"check specific things"
git config user.name
git config user.email
```

git work flow

The **Git workflow** refers to the series of steps developers typically follow when using Git to manage a project. It defines how changes are made, committed, and integrated into the main codebase. Here's a general Git workflow, broken down into five key steps:

1. Cloning or Initializing a Repository

- **Clone a Remote Repository**: If you're working on an existing project, the first step is to clone the remote repository to your local machine.

  ```
  git clone <repository_url>
  ```

  This creates a local copy of the repository, including all files and the entire commit history.
- **Initialize a New Repository**: If you're starting a new project, you can initialize a new Git repository.

  ```
  git init
  ```

  This creates an empty Git repository in the current directory.

2. Creating and Switching Branches

- **Create a New Branch**: Before working on a feature or bug fix, create a new branch. This keeps your changes isolated from the main codebase (`master` or `main` branch).

  ```
  git checkout -b <branch_name>
  ```

  This creates a new branch and switches to it.
- **Switch Between Branches**: You can switch to any branch using:

```
git checkout <branch_name>
```

3. Making Changes and Staging Files

- **Modify Files**: Make changes to the files in your project (e.g., adding features or fixing bugs).
- **Check the Status**: Use `git status` to see what has been modified and what's currently staged.

```
git status
```

- **Stage Changes**: To add the modified files to the staging area (prepare them for a commit):

```
git add <file_name>        # Stage a specific file
git add .                  # Stage all changes in the current directory
```

4. Committing Changes

- **Commit Staged Changes**: After staging, commit the changes with a descriptive message.

```
git commit -m "Describe the changes made"
```

This creates a new snapshot of your project with the staged changes.

- **View Commit History**: You can check the commit history using:

```
git log
```

5. Pushing and Merging Changes

- **Push Changes to Remote Repository**: Once you're satisfied with the changes and the commits are made locally, push them to the remote repository (e.g., GitHub, GitLab).

```
git push origin <branch_name>
```

This sends the branch to the remote server so others can see it or merge it.

- **Create a Pull Request (PR)**: On platforms like GitHub or GitLab, you can open a pull request to merge the changes from your feature branch into the `main` branch. This allows for code review and discussion before merging.
- **Merge Branches**: If you're ready to integrate your branch with `main`, you can:
  - First, switch to the `main` branch:

    ```
    git checkout main
    ```

  - Then merge the feature branch:

    ```
    git merge <branch_name>
    ```

  - Resolve any conflicts if they arise during the merge process.
- **Pull Changes from Remote**: Regularly pull the latest changes from the remote repository to ensure your local repository is up to date:

  ```
  git pull
  ```

Example Workflow in Practice:

1. **Clone a Repository**:

   ```
   git clone https://github.com/your-repo/project.git
   cd project
   ```

2. **Create a Feature Branch**:

   ```
   git checkout -b new-feature
   ```

3. **Make Changes and Stage Files**:

   ```
   # Edit files
   git add .
   ```

4. **Commit the Changes**:

```
git commit -m "Add new feature implementation"
```

5. **Push to Remote and Open a Pull Request**:

```
git push origin new-feature
```

6. **Merge the Branch** (after approval):

```
git checkout main
git merge new-feature
```

7. **Delete the Feature Branch** (optional, after merging):

```
git branch -d new-feature
```

Summary of the Workflow:

1. **Clone or initialize** a repository.
2. **Create a new branch** to work on a feature.
3. **Stage and commit** your changes.
4. **Push** your changes to the remote repository.
5. **Open a pull request** and merge changes into the main branch.

This workflow ensures clean, organized development, facilitates collaboration, and keeps the project history intact.

```
1115  mkdir demo1
1116  ls
1117  cd demo1
1118  ls
1119  ls
1120  git init
1121  git config --global init.defaultBranch main
1122  config list
1123  git config list
1124  git branch
1125  git branch list
1126  touch chatper1.txt
1127  touch chatper2.txt chatper3.txt chatper4.txt chatper5.txt
1128  ls
1129  git add .
1130  git status
1131  git log
1132  git commit -m "add some chapters.txt"
1133  git log
1134  touch index.txt
1135  touch tableOfContent.txt
1136  ls
1137  git add .
1138  git statu
1139  git status
1140  git commit -m "add two more files"
1141  git log
1142  git status
1143  vim chatper5.txt
1144  git status
1145  git add chatper5.txt
1146  git commit -m "modified the chapter5.txt"
1147  git status
1148  git log
1149  ls
1150  cat .git/config
1151  history
```

```
yiminggao@MacBookAir demo1 % ls .git
COMMIT_EDITMSG  config          hooks           info            objects
HEAD            description     index           logs            refs
```

Java

-