90 DAYS OF DEVOPS

WITH SHUBHAM LONDHE

About the Author

Hi, I am **Shubham Londhe**, **YouTuber - TrainWithShubham**, and an experienced Software Engineer and Technical Trainer, passionate about DevOps and development, whose vision is to transform everyone's career into IT, irrespective of their background.

I have Trained more than **5000+ students** to get their **dream jobs in IT**.

TWS Website

For DevOps and Tutorials

TWS YouTube

For DevOps and Tutorials

PART 1

The Fundamentals

Days 1-3

Previously on...

Before we step into the world of DevOps transformation at NexusCorp, you need to grasp the core principles that will be your guiding light. In Part I, we explore the essentials of DevOps—what it is, why it matters, and how it evolved.

Day 1-3: What is DevOps?

History and Evolution

Understand where DevOps comes from and how it has evolved over time.

Exercise: Research a case study about a company that failed due to the lack of DevOps principles. Write down your observations.

DevOps Culture

What makes DevOps culture special? How does it differ from traditional IT culture?

Exercise: List the DevOps cultural principles that you feel would have helped prevent NexusCorp's initial problems.

The DevOps Lifecycle

Get familiar with the cycle of DevOps from coding to deployment and monitoring.

Mini-Project: Write a blog post summarizing your understanding of what DevOps is, incorporating what you've learned about its history, culture, and lifecycle. Share it with a colleague or friend and ask for feedback.

Questions to Open Up Minds:

1. What was the most shocking aspect of NexusCorp's initial state to you?
2. Why do you think DevOps could be a solution to NexusCorp's problems?
3. How can DevOps be customized to fit different organizational needs?

RESOURCES AND GLOSSARY

- Start with a DevOps Roadmap: YouTube Video
- How Netflix Became A Master of DevOps? An Exclusive Case Study: Read More

PART 2

Unlock the Terminal

Day 4-12

Transition into Linux

Transition into Linux

→ Before we move further into the DevOps transformation at NexusCorp, it's crucial to lay the groundwork with a solid understanding of Linux. If DevOps is the engine that drives continuous integration and delivery, Linux is the fuel that powers that engine. Or, it powers just about everything.

→ Linux is ubiquitous. It's the backbone of vast cloud infrastructures and the cornerstone of countless data centers. Government organizations rely on Linux for its robust security features. Space agencies use Linux to interface with complex systems on Earth and beyond. Even in cybersecurity, Linux stands as a titan for its versatility and robustness. Financial institutions, healthcare systems, telecom industries—the list goes on, and I could keep naming sectors that leverage Linux for its unparalleled capabilities.

Preparing for Your Linux Journey

→ Now that we've established the critical role that Linux plays in the world of DevOps and in the IT industry at large, it's time to prepare for a deeper dive. You may be wondering, why dedicate an entire section to Linux in a book about DevOps? The answer is simple: Mastery over Linux can significantly elevate your DevOps skills. Linux offers you the flexibility, control, and power that are essential for managing complex infrastructures, automating tasks, and much more.

What to Expect

→ This next section offers you a 7-day learning plan designed to take you from a complete beginner to a confident Linux user. We've broken down the essential skills into manageable, bite-sized chunks. By the end of this, you'll not only understand the Linux operating system but also be prepared to take on more advanced DevOps topics.

Hands-On Learning

→ This book is committed to a learning-by-doing approach. Each day's lesson comes with practical exercises and mini-projects that encourage you to get your hands dirty. It's one thing to read about Linux commands; it's another to actually execute them.

DAY 4

Introduction to Linux and Setting Up

- Brief history of Linux and its variants (distros).
- Installing Linux on your system through a Virtual Machine or dual boot.

→ Why: Before you start, you need to understand what you're getting into and have a system to practice on.

Navigating the Filesystem

- cd, ls, pwd and other commands to traverse directories.
- File types and structure.

→ Why: The Linux filesystem is the foundation of your Linux journey. Understanding it is like knowing how to read a map.

DAY 4

Introduction to Linux and Setting Up

- Brief history of Linux and its variants (distros).
- Installing Linux on your system through a Virtual Machine or dual boot.

→ Why: Before you start, you need to understand what you're getting into and have a system to practice on.

Navigating the Filesystem

- cd, ls, pwd and other commands to traverse directories.
- File types and structure.

→ Why: The Linux filesystem is the foundation of your Linux journey. Understanding it is like knowing how to read a map.

Brief History of Linux and its Variants (Distros)

→ **Depth:** Learn about the evolution of Linux, starting from UNIX. Discuss key milestones such as the creation of GNU, the Linux Kernel by Linus Torvalds, and the rise of distros like Ubuntu, Fedora, and CentOS.

→ **Breadth:** Understand the ecosystem of Linux distros, categorized by use-case: desktop (Ubuntu, Mint), server (CentOS, Debian), and specialized distros (Kali Linux for cybersecurity, Raspbian for Raspberry Pi).

Installing Linux on your system

→ **Depth:** Go through a step-by-step guide for installing Linux through a Virtual Machine using software like VirtualBox or VMware. Alternatively, discuss how to set up a dual boot with Windows.

→ **Breadth:** Cover the pros and cons of using a VM vs dual boot, and briefly discuss cloud-based Linux solutions.

Practical Exercise

❖ Install a Linux distro on your system either through a VM or by setting up a dual boot.

❖ Write down the steps you took during the installation and any challenges you faced.

Navigating the Filesystem

→ **Depth:** Understand directory structure, how to list files (ls), change directories (cd), and display the present working directory (pwd).

→ **Breadth:** Differentiate between absolute and relative paths and understand special directories like ~ (home), . (current), and .. (parent).

Practical Exercise

❖ Navigate to your home directory and create a new directory called Day1.

❖ Inside Day1, create a text file named intro.txt and write "My first day learning Linux" in it.

❖ Use commands to find out which directory you're currently in and list the contents of the directory.

RESOURCES AND GLOSSARY

- Cheat Sheet: cheat.sh for Linux commands.
- In-depth Reading: The Linux Kernel.
- Handwritten Commands: Kunal Kushwaha notes.

- Distro: A variant of Linux, bundled with software packages tailored for specific use cases.
- Absolute Path: The full path to a directory or file.
- Relative Path: The path to a directory or file relative to the current directory.

DAY 5

File Operations

- touch, mkdir, cp, mv, rm.
- Manipulating files and directories.

→ Why: File operations are your basic verbs in the Linux language; you need to know them to do almost anything.

Basic Text Manipulation

- cat, echo, nano, vi.
- Reading and editing files.

→ Why: Whether it's config files or scripts, text manipulation is a daily task on Linux.

File Operations: touch, mkdir, cp, mv, rm

→ **Depth:** Delve into each command, explaining its function, flags, and uses. For instance, cp can copy files but also directories with the -r flag. Learn how to use mv not just to move files but to rename them.

→ **Breadth:** Discuss the common patterns where these commands come together in everyday use. For example, creating a directory (mkdir) and immediately copying some files (cp) into it.

Practical Exercise

- Create a directory called Day2Files. Inside, generate three empty files named file1.txt, file2.txt, and file3.txt using touch.
- Copy file1.txt to a new directory you create named Day2FilesBackup.
- Rename file3.txt to file3_renamed.txt.
- Delete file2.txt.

Basic Text Manipulation: cat, echo, nano, vi

⟶ **Depth:** Go over the basics of each text manipulation tool. Show how cat can not just display a file but concatenate multiple. Explain how echo can be used to append text to files or print variables.

⟶ **Breadth:** Introduce nano and vi, two text editors, mentioning that vi is generally available on all systems by default, whereas nano is more beginner-friendly.

Practical Exercise

- Use echo to add the text "Linux is great" to a file named opinion.txt.
- Use cat to display the contents of opinion.txt.
- Open opinion.txt with nano and vi, make some changes and save them.

RESOURCES AND GLOSSARY

- Vim Cheat Sheet: For those wanting to master vi, this Vim Cheat Sheet is an invaluable resource.

- Flags: Additional options that can be added to commands to alter their behavior.
- Concatenate: The action of linking things together in a chain or series, often used in the context of files and strings in Linux.
- Text Editor: Software used for editing plain text files.

DAY 6

Permissions and Ownership

- Understanding User Roles, Groups, and Permissions.
- chmod, chown.
- Special Permissions: SUID, SGID, and Sticky Bit

→ Why: Security and permissions are critical for managing a stable and secure system, an essential skill for DevOps.

Understanding User Roles, Groups, and Permissions

→ **Depth:** Introduce the concept of users, groups, and the Linux permission model. Learn about the three types of permissions: read (r), write (w), and execute (x), and how they map to files and directories.

→ **Breadth:** Discuss the significance of the root user and how permissions can affect system stability and security. Understand permission groups: user, group, and others.

chmod: Changing File and Directory Permissions

→ **Depth:** Explain how to use chmod to change permissions in both symbolic (e.g., u+x) and octal formats (e.g., 755). Discuss the implications of setting permissions too loosely or too strictly.

→ **Breadth:** Provide examples that show real-world scenarios where chmod would be applied.

Practical Exercise

- Create a file called private.txt. Remove all permissions for group and others.
- Create a directory called shared_folder. Give read and execute permissions to the group and others.
- Change permissions of private.txt to be fully open, then restrict it again to be read-only for the user.

chown: Changing File and Directory Ownership

⟶ **Depth:** Dive into how chown can change the user and group ownership of files and directories. Explain flags like -R for recursive ownership change.

⟶ **Breadth:** Discuss how chown relates to system administration tasks and the precautions to take while using it.

Practical Exercise

❖ Create a file named ownership.txt. Use chown to change its ownership to another user on your system (you may need superuser privileges for this).

❖ Create a directory named ownership_folder. Change its group ownership.

Special Permissions: SUID, SGID, and Sticky Bit

⟶ **Depth:** Understand what each special permission does. SUID changes the user identity upon the execution of a file, SGID changes the group identity, and the Sticky Bit protects the deletion of files.

⟶ **Breadth:** Explain where and when to use these special permissions. Discuss their implications in shared environments and why they can be both useful and potentially dangerous.

Practical Exercise

❖ Create a directory called sticky_folder. Set the Sticky Bit on this folder and verify the permissions.

❖ Create a script called special_script.sh. Set it with SUID and execute the script as a different user. Observe the behavior.

❖ Create a directory and set it with SGID. Create a file in that directory and observe the group ownership.

RESOURCES AND GLOSSARY

- chmod Calculator: chmod Calculator to understand how different permission settings affect files and directories.
- chmod Command Guide: This command guide offers explanations and examples to help you get more comfortable with chmod.

- Root User: The superuser with the highest level of privileges.
- User: A Linux account with its own set of permissions.
- Group: A collection of users that share permissions.
- SUID: Set User ID upon execution.
- SGID: Set Group ID upon execution.
- Sticky Bit: Protects the deletion of files in a directory.
- Octal Format: A numeral system for setting permissions in Linux.
- Symbolic Format: A letter-based system for setting permissions in Linux.
- Read (r): Permission to read a file or directory.
- Write (w): Permission to modify a file or directory.
- Execute (x): Permission to execute a file or access a directory.

DAY 7

Process Management

❖ ps, top, kill, nice, and other commands for process monitoring and control

⟶ Why: Understanding how to manage processes is crucial for system optimization and troubleshooting.

Understanding Processes and Threads

⟶ **Depth:** Introduce what a process is and how it differs from a program. Discuss threads and how they relate to processes.

⟶ **Breadth:** Explain the process lifecycle and states (running, sleeping, terminated, etc.). Understand how processes can spawn child processes.

ps: Viewing Active Processes

⟶ **Depth:** Dive deep into the ps command. Explain its various options like -e, -f, and -u that provide extensive information about all the running processes.

⟶ **Breadth:** Illustrate how ps can be used in real-world troubleshooting scenarios.

Practical Exercise

❖ *se ps -e* to list all processes and identify their Process IDs (PIDs).

❖ Use *ps -u* [username] to see all processes started by a particular user.

top: Real-time Process Monitoring

⟶ **Depth:** Describe what top displays, including CPU usage, memory, and other process-level statistics.

⟶ **Breadth:** Explain how top is useful for system administrators and DevOps engineers for real-time monitoring.

Practical Exercise

❖ Open top and identify the top 3 CPU-consuming processes.

❖ Use the h key in top to display a list of commands you can use within top.

kill and nice: Managing Process Execution

→ **Depth:** Understand the kill command and its signals (-9, -15, etc.). Learn how nice can adjust a task's priority.

→ **Breadth:** Discuss scenarios where killing a process is necessary and how changing process priority can optimize system performance.

Practical Exercise

❖ Start a long-running command and find its PID. Use kill to terminate it.

❖ Use nice to start a new process with a lower priority, then observe its CPU utilization compared to other tasks.

RESOURCES AND GLOSSARY

- Linux Performance Monitoring: Monitor Linux Performance on TecMint.
- Red Hat Monitoring and Automation: Red Hat Enterprise Linux 6 Deployment Guide enterprise-level monitoring and automation.

- Process: An instance of a running program.
- Thread: The smallest unit of a CPU's utilization, subordinate to a process.
- PID: Process ID, a unique identifier for each process.

DAY 8

Package Management and Software Installation

- apt, yum
- Installing, updating, and removing software
- Update vs. Upgrade

→ Why: Managing software is a routine part of system administration, and understanding it rounds off your basic Linux toolkit.

Introduction to Package Managers: apt and yum

→ **Depth:** Explain what a package manager is and its role in a Linux system. Introduce apt (Debian-based) and yum (RPM-based) as two of the most commonly used package managers.

→ **Breadth:** Discuss why package managers are useful for not just installing software, but also for handling dependencies and updates.

Practical Exercise

- Use apt or yum to install a new package.
- List all installed packages using apt list --installed or yum list installed.

apt vs yum

→ **Depth:** Describe the differences between apt and yum, focusing on syntax, configuration files, and the underlying package formats (deb vs rpm).

→ **Breadth:** Offer scenarios where one might be preferable over the other, depending on the Linux distribution and specific requirements.

Practical Exercise

- Update package lists using apt update or yum update.
- Upgrade a specific package using apt upgrade [package_name] or yum upgrade [package_name].

Update vs. Upgrade

⟶ **Depth:** Differentiate between 'update' and 'upgrade'. An 'update' refreshes the package lists, while an 'upgrade' actually installs new versions of packages.

⟶ **Breadth:** Discuss the importance of keeping a system up-to-date and the potential risks involved in upgrading packages.

Practical Exercise

- Perform an 'update' and then an 'upgrade' on your system.
- Check the logs to verify the updates and upgrades.

RESOURCES AND GLOSSARY

- Package Managers: Tecmint's Guide on Linux Package Managers.

- Package Manager: A utility to manage software packages on Linux.
- apt: Advanced Package Tool, used in Debian and Ubuntu.
- yum: Yellowdog Updater, Modified, used in Red Hat and CentOS.
- Dependencies: Additional packages required by a software package.
- Update: Refreshing the package list.
- Upgrade: Installing newer versions of installed packages.

DAY 9

Networking Basics

- ping, ifconfig, netstat
- Basic network troubleshooting and configuration

→ Why: Networking is fundamental to the interconnected world of DevOps.

Text Processing Tools

- grep, awk, sed

→ Why: These tools are extremely powerful for text manipulation, a common task in Linux.

Networking Basics: ping, ifconfig, netstat

→ **Depth:** Introduce the role of networking in Linux and DevOps. Explain what each command (ping, ifconfig, netstat) does, and how they can be used for basic troubleshooting and network configuration.

→ **Breadth:** Discuss how network knowledge is essential for understanding the architecture of large systems, facilitating better communication and data flow between servers.

Practical Exercise

- Use ping to check connectivity to a domain.
- Use ifconfig to display network configurations.
- Use netstat to list all network connections.

Text Processing Tools: grep, awk, sed

→ **Depth:** Dive into text processing tools like grep, awk, and sed. Explain what each tool is used for and give examples of how they can make text manipulation much more efficient.

→ **Breadth:** Discuss the flexibility these tools offer, and how they are often used together to perform complex text processing tasks efficiently.

Practical Exercise

- Use grep to search for a specific string in a text file.
- Use awk to process a CSV file and extract specific fields.
- Use sed to find and replace text in a file.

RESOURCES AND GLOSSARY

- Linux Networking: Linux Networking Commands.
- Understanding sed and awk: What is sed & awk?

- ping: A command-line utility used to test the reachability of a host.
- ifconfig: A command-line utility for displaying and configuring network
- interfaces.
- netstat: A command-line utility that prints network connections, routing
- tables, and other network-related information.
- grep: A text-processing utility for searching lines in text.
- awk: A text-processing utility for scanning and parsing each line in a text
- file based on a particular pattern.
- sed: A stream editor for filtering and transforming text.

DAY 10

Basic Shell Scripting

- Writing your first Bash script
- Variables, loops, and conditionals

→ Why: Automation is a big part of Linux and DevOps, and basic scripting helps you automate tasks.

Task Scheduling

cron and at for automated task scheduling

→ Why: Automation includes not just scripting but also scheduling tasks to run unattended.

Basic Shell Scripting: Writing Your First Bash Script

→ **Depth:** Start with the importance of scripting in automation and introduce Bash as a commonly used shell for scripting in Linux. Discuss syntax, variables, loops, and conditionals.

→ **Breadth:** Briefly touch upon other shell scripting languages and their use cases, but focus on Bash for its ubiquity and ease of use for beginners.

Practical Exercise

- Write a basic Bash script that prints "Hello, World!".
- Modify that script to use a variable for the greeting.
- Write a Bash script that loops through numbers 1 to 10 and prints them.
- Create a Bash script that checks if a number is even or odd using conditionals.

Task Scheduling: cron and at

→ **Depth:** Discuss the need for task scheduling in a typical DevOps workflow. Go through the basics of cron and at commands for task scheduling.

→ **Breadth:** Highlight how scheduling tasks can help in automating backups, system maintenance, and other repetitive tasks.

Practical Exercise

❖ Schedule a Bash script to run every minute using cron.

❖ Use at to run a script at a specific time in the future.

❖ Create a cron job that runs a script every day at a specific time.

❖ Experiment with scheduling different types of tasks, like sending emails or running system checks.

RESOURCES

- A Beginner Guide To Cron Jobs.
- Crontab.guru.
- Video Tutorial: Linux Shell Scripting Playlist on YouTube.
- JSON Processing with jq: jq is a command-line JSON processor. It is used to parse, extract, transform, and generate JSON data. For more details, visit jq's official site.
- Bash Cheat Sheet: For quick references and tips, visit devhints.
- Quick Bash Guide: For a rapid tour of Bash scripting, check Learn X in Y minutes.

Supplementary Exercises and Challenges

Fun Project: Pomodoro Timer for Terminal

❖ Why: A Pomodoro timer can help you manage your time more efficiently, which is a crucial skill in any work environment, including DevOps. This project will also give you practical experience with Bash scripting.

❖ Details: Create a Pomodoro timer right within your terminal to keep you focused and efficient.

Resource

Pomodoro Timer for Terminal on dev.to

Challenge: BashBlaze - 7 Days of Bash Scripting

❖ Why: his 7-day challenge can significantly enhance your Bash scripting skills. Each day introduces a new challenge, making the learning curve smooth yet comprehensive.

❖ Details: The challenge covers a range of topics from basic to advanced, ensuring that you get a well-rounded understanding of Bash scripting.

Resource

BashBlaze on GitHub

OPTIONAL DAY 11 | OPTIONAL DAY FOR FURTHER LEARNING

Filesystem Hierarchy and Disk Management

- Understanding the Linux Filesystem Hierarchy (/etc, /var, /usr, etc.)
- Disk Partitioning.
- Filesystem Management (df, du, fdisk, mkfs)

→ Why: A deep understanding of the filesystem hierarchy and disk management is crucial for anyone responsible for maintaining and optimizing a Linux system, especially in a DevOps context.

Exercise

- Navigate through different directories and identify their purposes.
- Practice disk partitioning on a virtual machine.
- Monitor disk usage with df and du.

OPTIONAL DAY 12 | OPTIONAL DAY FOR FURTHER LEARNING

Logs and Monitoring

- System Logs (/var/log)
- syslog and rsyslog.
- Monitoring Tools (htop, iostat, vmstat)

→ Why: Monitoring and logs are critical for diagnosing issues, optimizing performance, and maintaining the security of your systems. These skills are especially important for system administrators and DevOps professionals.

Exercise

- Browse through the logs in /var/log and interpret common entries.
- Configure rsyslog to handle custom logging.
- Use monitoring tools to identify system bottlenecks.

PART 3

The Journey Is Far from Over

Get Your Hands Dirty!
As the saying goes, "The best way to learn is by doing." Don't hesitate to dive into complex tasks or tackle challenging problems.

What's Next?

Up next, we'll delve into the Version Control System, another cornerstone in the DevOps landscape. But before that, make sure you've got a good handle on Linux. After all, a well-constructed building needs a solid foundation.

The Behind-the-Scenes Magic of Making Apps

In the vastest world of software development, DevOps engineers wear a unique crown. While we are not typically the ones writing application code, our role is paramount in ensuring that the software delivery pipeline is seamless, efficient, and agile. Understanding how developers work and the intricacies of the Software Development Lifecycle (SDLC) is extremely important for any seasoned DevOps engineer.

1. The Software Development Life Cycle (SDLC)

At its core, SDLC is a structured process employed to develop software. It consists of phases:

- **Requirement Gathering:** This is the phase where stakeholders communicate their needs and developers assess the feasibility.
- **Design:** Based on the requirements, architects and developers create system and software designs.
- **Implementation (Coding):** Developers translate designs into code.
- **Testing:** The software undergoes rigorous testing to ensure it meets specified requirements and is free from defects.
- **Deployment:** If testing is successful, the software is deployed to production.
- **Maintenance:** Post-deployment, software needs regular updates and fixes. As a DevOps engineer, you'll work closely with teams across all these phases, particularly from implementation to deployment.

2. Version Control Systems (VCS)

When multiple developers collaborate on a project, keeping track of changes becomes vital. Enter Version Control Systems like Git. These tools allow developers to track modifications, branch out to develop features in isolation, and merge changes back. As a DevOps engineer, you'll automate many of these processes and ensure that code changes seamlessly move through development, testing, and production environments.

3. Collaborative Culture

Beyond tools and practices, it's essential to understand that DevOps is also about fostering a culture. It's about collaboration between development, operations, and even business teams. Regular communication, feedback loops, and shared responsibilities are foundational elements. As a bridge between teams, a DevOps engineer often plays a role in nurturing this culture.

DAY 13

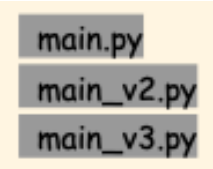
Introduction to Version Control System

- Explanation.
- Types of VCS.

→ Why: Version control systems are a vital part of modern software development. They allow teams to manage changes to code over time, enabling collaboration and rollback capabilities.

Introduction to Version Control System: Explanation

→ Early software development was chaotic. Developers maintained different versions of code by copying and renaming files:



→ Like above
While this worked for solo developers, it quickly broke down for teams. Developers would overwrite each other's changes, leading to lost work and bugs.

→ The first version of control systems emerged in the 1970s and 1980s. These centralized systems stored all code in a central repository. Developers would pull the latest version, make changes locally, and push those changes back.

→ While this brought some order, issues remained. If the central repository went down, no work could be done. And changes from multiple developers could conflict when pushing.

→ The rise of distributed version control in the mid-2000s was a game changer. In these systems, each developer has a full copy of the repository history. Changes are made on local branches before being merged into the central repo.
If the central repo fails, any local copy can restore it.

Today's version control provides:

- A complete history of all changes
- Branching and merging capabilities
- Traceability to identify bugs and fix issues
- Rollback options to restore previous versions

While early approaches were chaotic and clumsy, modern version control systems enable teams to collaborate effectively and build software swiftly and reliably. The evolution of version control mirrors the evolution of software development itself - from a solo endeavor to a team sport.

→ In short, the story of version control is the story of code itself - constantly evolving and improving to enable human creativity at scale.

Types of VCS:

→ **Local Version Control Systems (LVCS):** These are simple databases that maintain changes in files on a local system. They're essentially the ancestors of modern VCS, best suited for solo projects.

→ **Centralized Version Control Systems (CVCS):** Platforms like SVN fall under this category. Here, a single server stores all the versions of a file, while various clients fetch them from there. This structure enables collaboration but can become a single point of failure.

→ **Distributed Version Control Systems (DVCS):** Git, Mercurial, and the like belong to this category. Instead of a single centralized repository, every developer's working copy of the codebase is also a repository that can contain the full history of all changes. This enhances collaboration, provides redundancy, and allows for offline work.

Benefits:

1. **Collaboration:** Multiple developers can work on a project without stepping on each other's toes.
2. **History:** You can travel back in time, review changes, and understand the evolution of a project.
3. **Backup:** Every copy in DVCS is a full-fledged repository, reducing risks associated with failures.
4. **Branching and Merging:** Facilitates parallel development and seamless integration.
5. **Accountability:** With a VCS, you know who made what change and when.

Practical Exercise

❖ **Blog Writing:** Share your newfound knowledge with the world! Draft a blog post elucidating the differences between centralized and distributed version control. Discuss their pros, cons, and ideal use cases.

❖ **Analyze a Project:** Choose any open-source project on platforms like GitHub or GitLab. Explore its commit history, branches, and merges. Make a note of patterns you observe and any interesting or complex merges you come across.

❖ What issues did early version control systems have that distributed systems aimed to solve?

❖ What advantages does branching provide in version control systems?

❖ What benefits does having a complete history of changes offer during software development?

RESOURCES

- Version control concepts and best practices.

BEFORE DAY 14 DEBUNKING MYTHS & SOME KEY POINTS

- Debunking Myths & Some Key Points
- Myth 1: Git and GitHub are the same thing.
- Myth 2: Git is only for developers.
- Myth 3: Using GitHub means my code is open source.
- Difference between Git and GitHub:
- Why is this distinction important?

Myth 1: Git and GitHub are the same thing.

- Fact: Git is a version control system, a tool that tracks changes in source code during software development. GitHub, on the other hand, is a cloud-based platform built around the Git tool. GitHub provides a space for developers to host and share their code, collaborate on projects, and streamline version control.

Myth 2: Git is only for developers.

- Fact: While Git is primarily used by developers, its applications extend beyond coding. Writers, designers, and professionals in various domains use Git for version control of documents, images, and other files. It's a versatile tool for anyone looking to track changes and collaborate.

Myth 3: Using GitHub means my code is open source.

- Fact: GitHub does champion open-source collaboration, but it also allows private repositories where your code can remain hidden from the public eye.

Difference between Git and GitHub:

- **Nature:** Git is a tool, while GitHub is a service.
- **Functionality:** Git provides the mechanism to track changes in the source code, whereas GitHub provides a hosting platform to manage and report on these changes.
- **Access:** You can use Git on your local machine without an internet connection. In contrast, GitHub (unless you're using the Enterprise version) requires an internet connection to push to or pull from repositories.
- **Independence:** Git can exist without GitHub, but GitHub relies on Git for its primary functionality.

Why is this distinction important?

⟶ Understanding the difference between Git and GitHub is crucial for several reasons. Firstly, it helps you select the right tools for your project. If all you need is version control, Git suffices. But if you're looking for collaboration, integrated issue tracking, and a platform for showcasing your projects, GitHub comes into play.

⟶ Additionally, many other platforms, like GitLab and Bitbucket, also utilize Git for version control but provide different features and interfaces. Being clear about what Git is (and isn't) allows you to make informed choices about the platforms you choose to work with.

DAY 14

Setting Up and Basic Commands

- Configuring Git.
- Basic Commands.

→ Why: Version control systems are a vital part of modern software development. They allow teams to manage changes to code over time, enabling collaboration and rollback capabilities.

Configuring Git:

→ Before diving into commands, ensure you've set up Git with your personal configurations. This helps in identifying who is making changes, especially when collaborating.

```
git config --global user.name "Your Name"
git config --global user.email "youremail@example.com"
```

Basic Commands:

→ 1. **git init:** Initializes a new Git repository and starts tracking an existing directory.

```
mkdir new_project
cd new_project
git init
```

→ 2. **git clone:** Clone (or download) a repository from an existing URL.

```
git clone https://github.com/example/repo.git
```

→ 3. **git add:** Adds changes in the working directory to the staging area.

```
echo "Hello Git" > hello.txt
git add hello.txt
```

→ 4. **git commit:** Captures the state of your repository's tracked files and prepares them to push to a remote repository.

```
git commit -m "Added hello.txt"
```

→ **git status:** Displays the status of changes as untracked, modified, or staged.

```
git status
```

Practical Exercise

1. Initialize and Commit:

- Create a new directory named my_git_project and navigate into it.
- Initialize it as a Git repository.
- Create a file named readme.md.
- Write a brief introduction about yourself in this file.
- Add and commit this file to your repository.

2. Clone and Explore:

- Clone a public repository from GitHub that you find interesting.
- Explore its contents, check the commit history, and examine a few commit messages.

3. Staging Area Workflow:

- Create a new file notes.txt in my_git_project.
- Add a few lines about what you learned from the YouTube video and save it.
- Check the status of your Git repository.
- Stage the notes.txt file and then check the status again.
- Commit the file with an appropriate message.

BONUS

After doing the above exercises, write a brief blog post or journal entry detailing your experience. Share what you've learned, what was challenging, and what you're looking forward to learning next in Git.

RESOURCES AND GLOSSARY

- Download Git: git-scm.com/downloads
- Quick Class on Git & GitHub: YouTube Video

DAY 15

Remote Repositories, Branching, and Branching Strategies

- Remote Repositories.
- Branching.

Remote Repositories:

→ Remote repositories allow you to collaborate with others and keep a backup of your local repository on a remote server. Platforms like GitHub, GitLab, and Bitbucket offer hosting for remote repositories.

Basic Commands:

→ **git remote:** Used to view remote repositories connected to your local repository.

→ **git push:** Pushes changes from the local repository to a remote repository.

→ **git pull:** Fetches changes from a remote repository and merges them into the local repository.

→ **git fetch:** Retrieves changes from a remote repository but doesn't merge them.

Branching:

→ Branching in Git is a powerful feature that allows developers to work on different features, fixes, or experiments concurrently without affecting the main codebase.

Basic Commands:

→ **git branch:** Lists all branches in the repository. The active branch is indicated with an asterisk (*).

→ git checkout: Switches to a different branch. Combined with -b, it can create a new branch and switch to it.

→ **git merge:** Merges changes from one branch into another.

Practical Exercise

1. Pushing to a Remote:

- Fork a public repository on GitHub.
- Clone your fork to your local machine.
- Make a change in a file, add, and commit it.
- Push the change to your fork on GitHub.

2. Branch Creation and Switching:

- In the previously cloned repository, create a new branch named feature-x.
- Switch between main (or master) and feature-x multiple times to get a feel for it.

3. Simulating Collaboration:

- On the feature-x branch, make some changes to a file, add, and commit.
- Switch to the main branch and make changes to the same file, then add and commit.
- Try merging feature-x into main. Handle any merge conflicts that arise.

4. Experiment with Different Branching Strategies:

- Research a branching strategy from the GitKraken guide.
- Implement the chosen strategy in a new repository, simulating a workflow with multiple features and releases.

BONUS

Organize a group activity with friends or classmates. Each member should clone the same repository, create a feature branch, make changes, and then attempt to merge them into the main codebase. Discuss merge conflicts and how to resolve them.

RESOURCES

- Git Branching Strategies: GitKraken's Guide to Best Practices.

DAY 16

Merging and Handling Conflicts

- Merging.
- Handling Conflicts.
- Why is Understanding Merging and Conflict Resolution Essential?

Merging:

⟶ In Git, merging is the process of integrating changes from one branch into another. It's a primary way of combining the separate lines of development. When you're done with a feature or fix in a branch, you'll merge it into your main branch to roll out the change.

Basic Commands:

⟶ **git merge [branch]:** Merges the specified branch into the current branch.

Handling Conflicts:

⟶ Sometimes, when you attempt to merge, Git can't automatically combine the changes because both branches have edited the same line of a file differently. This results in a merge conflict. Thankfully, Git doesn't just leave you stranded. It marks the problematic area in the file and asks you to resolve it manually.

Commands:

1. Open the conflicted file and look for the conflict markers (<<<<<<<, =======, and >>>>>>>).
2. Decide if you want to keep only your branch's changes, the other branch's changes, or a combination of both.
3. Once resolved, remove the conflict markers and save the file.
4. Run git add [filename] to mark the conflict as resolved.
5. Complete the merge with git commit.

Why is Understanding Merging and Conflict Resolution Essential?:

→ Merge conflicts are an inevitable part of collaborative coding. Addressing them ensures smooth collaboration and a clean codebase. Without resolving these conflicts, the code remains broken and can't function. By mastering conflict resolution, you ensure your projects remain on track and your team can work efficiently.

Practical Exercise

1. Simulate a Merge Conflict:

- Clone or use an existing repository.
- Create a new branch.
- In both the new branch and the main branch, make different changes to the same line of a file.
- Attempt to merge the new branch into the main branch and observe the conflict.
- Resolve the conflict.

2. Collaborative Merge Challenge:

- Team up with a friend or a colleague.
- Both of you should fork the same repository.
- Make different changes to the same part of a file in your respective forks.
- One of you creates a pull request to the other's repository. Try to merge and resolve the conflict that arises.

ADVANCED CHALLENGE:

Implement a feature in a branch and then intentionally create a few conflicts with the main branch. Merge the feature branch into the main branch and practice resolving each conflict.

RESOURCES

- Resolving a merge conflict using the command line: GitHub Guide

DAY 17

Advanced Rebasing, Git Log, Stash, Reset, and More

- Advanced Rebasing.
- Git Log.
- Stash.
- Reset.
- Cherry-Picking.
- Other Useful Concepts

Advanced Rebasing:

⟶ Rebasing is the process of moving or combining a sequence of commits to a new base commit. It's a way to integrate changes from one branch into another, without creating a merge commit.

Basic Commands:

⟶ **git rebase [branch]:** Apply any change from the current branch onto [branch].

Git Log:

⟶ Git log shows a list of commits in a repository. It's a way to see the history of your repo, allowing you to understand the changes and navigate through them.

Commands:

⟶ **git log:** Show commit logs.
⟶ **git log --oneline:** Show commit logs in a concise format.

Stash:

⟶ Stashing takes the changes of the working directory and saves them for later, allowing you to switch branches without committing your changes.

Commands:

⟶ **git stash save "message":** Save changes with a descriptive message.
⟶ **git stash list:** List stashed changes.
⟶ **git stash applies:** Apply the latest stashed changes.

Reset:

→ Git reset is used to undo changes in your working directory that haven't been committed yet.

Basic Commands:

→ git reset [commit]: Move the current branch tip to [commit].
→ git reset --hard [commit]: Move the current branch tip to [commit] and match the working directory.

Cherry-Picking:

→ Cherry-picking in Git means to choose a commit from one branch and apply it onto another.

Commands:

→ **git cherry-pick [commit]:** Apply changes introduced by the named commit.

OTHER USEFUL CONCEPTS:

.gitconfig: It's a configuration file where you can set Git parameters for user info, aliases, and more.
.git folder: A hidden folder in your repository where Git tracks the changes and history.

Practical Exercise

1. Rebase Playground:

- Create a new feature branch and make a few commits.
- Go to the main branch and make a few commits.
- Now, rebase the feature branch onto the main branch.

2. Stashing Challenge:

- Commit three changes sequentially.
- Use git reset to go back to the first commit, simulating the need to undo the last two changes.

3. Reset Game:

- Commit three changes sequentially.
- Use git reset to go back to the first commit, simulating the need to undo the last two changes.

4. Cherry-Picking Exercise:

- Commit a change in one branch.
- Use git cherry-pick to apply that change to another branch.

INTERVIEW QUESTIONS AND PRACTICE:

1. What is the difference between git merge and git rebase? How would you undo the last commit?
2. Describe a scenario where you might use git stash.
3. What is the difference between git reset and git revert?

DAY 18

Engage, Reflect & Share: Best Practices and Collaboration in Git

- Objective.
- Tasks: Reflect on Your Git Journey.
- Tasks: Design Your Git Cheat Sheet.
- Tasks: Learning in Public: Share on LinkedIn.
- Tasks: Engage with the Community.
- Why This Exercise?

Objective:

→ By the end of this session, learners will have a better understanding of their own unique journey with Git and will have shared their insights, tools, and best practices with a wider audience, fostering community learning and collaboration.

TASK 1: REFLECT ON YOUR GIT JOURNEY

- Think about how you started with Git, the challenges you faced, and how you overcame them.
- List down the best practices you've adopted that you think set you apart.

TASK 2: DESIGN YOUR GIT CHEAT SHEET:

- Collate the most valuable commands, tricks, and insights you've gained throughout the Git sessions. It can be handwritten notes from your learning, a digital compilation, or a fancy infographic. Use tools like Canva, Piktochart, or even simple word processors.

TASK 3: LEARNING IN PUBLIC: SHARE ON LINKEDIN

❖ Post your Git cheat sheet on LinkedIn.

❖ Caption it with insights from your reflection: What unique best practices did you adopt? How has your Git journey been? What did you learn about collaboration and version control that you didn't know before?

❖ Use hashtags like #GitJourney, #LearningInPublic, #GitBestPractices to increase visibility.

❖ Tag any influencers, educators, or mentors who inspired you during this learning journey.

TASK 4: ENGAGE WITH THE COMMUNITY

❖ Spend some time browsing the aforementioned hashtags.

❖ Engage with posts from other learners - like, share, and comment. Collaboration is key!

❖ Offer help, answer questions, or simply appreciate the efforts of fellow learners. Build your network and learn from others' experiences.

Why This Exercise?

- Reinforces your understanding.
- Builds your personal brand in the tech community.
- Promotes the culture of 'learning in public,' which can be a powerful tool for growth and networking.

PART 4

The Unsung Hero of the Digital World - Networking

Ever paused to marvel at the marvels of the Internet? Have you ever typed something into your browser and wondered, "What exactly happens behind the scenes for this page to load in a fraction of a second?"

What's Next?

We'll dive deep (quite literally, to the ocean floors) and fly high (through the satellite-studded skies) to explore how data packets navigate this vast expanse.

SELF LEARNING | CONNECT THESE DOTS

Let's connect those dots and light up the digital universe together!

→ Instead of me delivering a monologue about networking, let's make this interactive. Because, hey, we all learn better when we're actively involved, right? This approach will not only fuel your curiosity but also ensure that you build a solid foundation.

DAY 19 | Networking Roadmap for DevOps and Cloud Professionals

1. Introduction to Networking

- What is Networking?
- Understanding Data Packets

2. Foundational Concepts

- IP Addressing and Subnetting
- Ports and Protocols
- OSI and TCP/IP Models

3. Network DeviceS and Infrastructure

- Routers, Switches, and Hubs
- Ethernet vs. Wi-Fi
- Network Topologies

4. Domain Name System (DNS)

- How DNS Works?
- Importance of DNS in Web Traffic.

5. Dynamic Host Configuration Protocol (DHCP)

- Role of DHCP in IP Assignment.
- DHCP Lease Process.

SELF LEARNING | CONNECT THESE DOTS

6.Network Protocols

- Transmission Control Protocol (TCP) vs. User Datagram Protocol (UDP).
- Understanding ICMP (Ping).

7.Network Address Translation (NAT)

- Purpose of NAT in Private and Public Networks.

8.Firewalls and Network Security

- Understanding Firewalls Network.
- Security Best Practices.

9.Virtual LAN (VLAN)

- Why and How VLANs are Used.
- VLAN Tagging and Trunking.

10.Virtual Private Networks (VPN).

- Understanding VPNs and Their Uses.
- VPN Tunnels and Protocols.

11.Load Balancing

- Basics of Load Balancing.
- Importance in Traffic Management.

12.Secure Socket Layer (SSL)/Transport Layer Security (TLS)

- Understanding SSL/TLS Handshake.
- Importance of Encryption in Web Traffic.

SELF LEARNING | CONNECT THESE DOTS

13. Network Monitoring and Troubleshooting

- Basics of Network Monitoring.
- Common Tools and Techniques.

14.Quality of Service (QoS)

- Understanding QoS.
- Implementing QoS for Traffic Management.

15.Software-Defined Networking (SDN)

- Introduction to SDN
- Benefits and Use Cases.

16.Network Automation

- The Need for Automation in Networking.
- Basics of Infrastructure as Code (IaC).

17.Internet Protocols

- Understanding IPv4 vs. IPv6
- IP Address Allocation and Management

18.Network Security and Best Practices

- Intrusion Detection and Prevention
- Systems (IDS/IPS) Securing Network Devices and Traffic.

19.Content Delivery Networks (CDN)

- Understanding CDNs and Their Role in Web Traffic.
- Benefits of Using CDNs.

Now, here's your challenge:

→ For each set of questions, take the initiative to research, explore, and document your findings. Create a reference guide for yourself. This way, you aren't just passively consuming content; you're actively building your resource, tailored to your understanding.

RESOURCES

- GitHub Repository: What Happens When... An insightful deep dive into what happens when specific networking actions are taken.
- GitHub CodeBits: Top 100 Networking Interview Questions & Answers.
- YouTube Video: TWS Computer Networking Masterclass.

PART 5

Building Logic with Python: Why It Matters for DevOps

As we stand on the brink of this Python journey, it's essential to understand why mastering Python, especially building logic with it, is invaluable for a DevOps professional.

What's Next?

Learning Python will arm you with a tool that can
Simplify Complex Tasks:
Enhance Problem Solving:
Bridge Communication Gaps:
Future-proof Your Career:

DAY 20

Building Logic with Python:Why It Matters for DevOps

- Introduction.
- Simplify Complex Tasks:
- Enhance Problem Solving:
- Bridge Communication Gaps:
- Future-proof Your Career:

Introduction:

→ Firstly, what is logic in the context of programming? It's the ability to think systematically, anticipate outcomes, and design sequences that produce desired results. When you write a program, you're creating a series of steps, a logical flow, for the computer to follow.

→ Now, transpose that concept to the vast world of DevOps. Here, every day is about creating, managing, and optimizing workflows—be it deploying applications, automating infrastructure, or ensuring CI/CD pipelines run flawlessly. At its heart, DevOps is about efficient and effective workflows, and what is a workflow if not a logical sequence?

Simplify Complex Tasks:

→ Automation is at the heart of DevOps, and Python is a powerful ally. By building robust logic with Python, you can automate intricate tasks, saving time and eliminating human errors.

Enhance Problem Solving:

→ Python sharpens your problem-solving skills. As you grapple with programming challenges, you cultivate a mindset that seeks solutions, a skill vital in the ever-evolving landscape of DevOps.

Bridge Communication Gaps:

→ With a coding language under your belt, you can communicate more effectively with developers, making collaborations smoother and more productive.

Future-proof Your Career:

→ The tech world is rapidly integrating development and operations. As tools and platforms evolve, the line between a developer and a system admin blurs. Python stands as a bridge between these domains, making you versatile and more adaptable to change.

NOTE

```
The best learning happens when you're curious. So
always question, experiment, and dive deep into the
logic of things. It's time to make your mark in the
DevOps world!
```

DAY 21

Introduction to Python and Tasks

- Introduction to Python.
- Tasks for the Day.

Introduction:

→ Python is an interpreted, high-level, general-purpose programming language. Due to its simplicity and readability, it's become one of the most popular languages for a wide range of tasks, including DevOps.

Practical Exercise - Tasks for the day

1. Install Python:

- Depending on your OS, download and install Python. You can find the official releases at Python's website.

2. Subtask:

- After installation, open your terminal or command prompt and type python --version to check the installed version.
- Study Data Types: Python has several built-in data types like integers, float (decimal), string (text), list, tuple, etc.
- Spend some time getting familiar with them. The official Python documentation is a good starting point.

3. First Python Script:

- Create a new text file with the extension .py (e.g., hello_devops.py).
- Inside this file, type the following code: print("Hello, DevOps!").
- Save the file and run it using the terminal or command prompt by navigating to the directory where you saved it and typing python hello_devops.py.

4. Exploration:

- Spend some time exploring the Python interactive shell.
- Simply type python in your terminal or command prompt, and you'll enter a mode where you can type Python commands directly.
- Try doing some basic arithmetic or define simple variables. To exit, you can type exit().

Practical Exercise - Tasks for the day

5. Documentation Dive:

- Take some time to navigate through Python's official documentation.
- Familiarizing yourself with it now will be invaluable later on.
- The "Beginner's Guide" is a great place to start.

NOTE

With these tasks, you'll have a solid foundational knowledge of setting up Python and understanding its basic data types.

DAY 22

Python Fundamentals and Tasks

- Python Fundamentals.
- Tasks for the Day.

Introduction:

→ Python, while known for its simplicity, has powerful features that allow for complex operations and logic. This chapter will cover basic programming constructs that are essential for any DevOps engineer.

Practical Exercise - Tasks for the day

1. Understanding Variables:

- In Python, a variable is used to store information that can be referenced and manipulated.
- There's no need to declare a variable type; Python does this automatically.
- **Subtask:** Create variables of different types: name = "DevOps", age = 30, salary = 1000.50, and then print each one using the print() function.

2. Control Structures:

- These are the building blocks of any program. Begin with if, else, and elif.
- **Subtask:** Write a script that checks if a number is positive, negative, or zero.

3. Loops:

- Learn about the two main types of loops in Python: for and while.
- **Subtask:** Write a script that prints numbers from 1 to 10 using both types of loops.

4. Functions:

- Functions allow for code reuse. Explore the def keyword to create a function.
- **Subtask:** Create a function named greet that takes a name as a parameter and prints "Hello, [name]!".

5. Python's Built-in Functions:

- Python comes with a battery of built-in functions ready to use.
- **Subtask:** Use the len() function to find the length of your name and the type() function to check the data type of various variables.

Practical Exercise - Tasks for the day

6. Lists and Tuples:

- These are Python's built-in data structures for storing collections of items.
- **Subtask**: Create a list of five DevOps tools you've heard of, and try adding, removing, and accessing items in the list.
- Do the same with a tuple and note the differences.

7. Python's Interactive Help:

- Python's help() function is useful for exploring modules, functions, and methods.
- **Subtask**: In the Python shell, type help(str) to see the methods and attributes available for string objects.

8. Exploring Errors:

- Mistakes happen. When you encounter an error, Python will throw an exception.
- **Subtask**: Deliberately make a syntax error in your code (like missing a closing parenthesis).
- Note the type of error and the message Python gives you.
- This will help you in troubleshooting in the future.

NOTE

By the end of these tasks, you'll have a strong foundation in the basic constructs of Python. Remember, practice is key. The more you code, the more fluent you'll become.

DAY 23

Python Libraries, Cloud Integration, and Data Parsing

- Introduction.
- Tasks for the Day.

Introduction:

→ Python offers an extensive ecosystem of libraries that can simplify complex tasks. Coupled with the cloud and data parsing abilities, Python becomes an indispensable tool for the modern DevOps professional.

Practical Exercise - Tasks for the day

1. Exploring Python Libraries:

- Python's vast standard library and third-party libraries are one of its biggest strengths.
- **Subtask:** Visit the YouTube resource provided and watch the introductory videos on Python libraries.
- **Subtask:** Use the pip command to install the requests library: pip install requests.

2. Python in the Cloud:

- Many cloud providers offer SDKs (Software Development Kits) in Python, allowing you to manage and automate cloud resources.
- **Subtask:** Skim through the Hashnode blog you shared to familiarize yourself with the integration of Python with cloud technologies.

3. Working with JSON Data:

- JSON (JavaScript Object Notation) is a lightweight data-interchange format that's easy to read and write.
- **Subtask:** Write a simple Python script to read a JSON file and print its content. You can create a mock JSON file for this, like:

```
{
  "name": "John",
  "age": 30,
  "city": "New York"
}
```

- **Subtask:** Extend the script to modify the data and write it back to the JSON file.

Practical Exercise - Tasks for the day

4. Other Data Formats:

❖ Apart from JSON, Python can handle several other data formats, such as XML, CSV, etc.

❖ **Subtask:** Explore Python's csv module and write a script to read a mock CSV file, then print its content.

5. Exploring More Libraries:

❖ As mentioned in the Hashnode resource, there are several essential libraries for DevOps and Cloud.

❖ **Subtask:** Pick one library from the blog post, such as boto3 (for AWS) or docker, and install it using pip. Skim through its documentation to understand its basic functionalities.

6. Final Reflection:

❖ After diving deep into these topics, reflect on your learning journey.

❖ **Subtask:** List down three key takeaways from today's tasks. It can be something you found interesting, something challenging, or any new idea you might have encountered.

RESOURCES

- Python Libraries YouTube Series.
- Hashnode Blog: Essential Libraries for DevOps and Cloud.

NOTE

With the completion of these tasks, you will have ventured into the powerful realm of Python libraries, integrated cloud services, and data handling capabilities in Python. This marks the final day of our guided learning.

PART 6

Containerization and Automation with Docker and Jenkins

In the realm of DevOps, learning Docker and automation is vital. Docker streamlines application deployment, while automation ensures efficiency and reliability, enabling faster delivery and improved collaboration in the fast-paced world of software development and IT operations.

DAY 24

Containerization and Automation with Docker and Jenkins

- Docker: Simplifying Containerization.
- What is a Container?
- Introduction to Docker in Layman's Language.
- What is an Image?

Docker: Simplifying Containerization

→ In the world of DevOps, if there's one buzzword that has echoed across corridors and conference rooms, it's Docker. And today, we demystify it.

What is a Container?

→ Imagine you're trying to move your house. Instead of packing each item individually and hoping they all fit well together in the truck, wouldn't it be easier to pack an entire room into a giant box and transport that? When you reach your new home, you just place these room-sized boxes where they belong. Your house is set, without the hassles of figuring out which item goes where.

→ That's essentially what a container does for software. Instead of packaging just the software, it wraps up the entire environment—code, runtime, system tools, system libraries, and settings—into a single 'box' called a container. This ensures that the software will run consistently, regardless of where the container is run.

Introduction to Docker in Layman's Language

→ In our house-moving analogy, if containers are the room-sized boxes, then Docker is the company that provides these boxes, helps you pack, and ensures the transport goes smoothly. In the tech world, Docker is a platform that makes creating, deploying, and running these containers super easy.

→ Now, while Docker is arguably the most popular 'company' providing this container service, it isn't the only one. There are other 'movers' in town. For instance, Podman is another tool that allows you to manage containers. Just like Docker, Podman lets you create, deploy, and run applications in containers.

What is an Image?

→ Let's stretch our analogy a tad more. Before packing up your room into a container, you take a photo of the room. This photo captures everything—the arrangement, the items, the color of the walls. If ever in the future, you want to recreate this exact room, you just look at the photo and set it up.

→ In Docker's world, this photo is what we call an 'image'. It's a lightweight, stand-alone, executable software package that encapsulates everything needed to run a piece of software, including the code, runtime, system tools, and libraries. An image becomes the blueprint for containers. Once you have an image, you can create as many containers from it as you need.

Practical Exercise - Tasks for the day

1. Your Own Words:

❖ Write a short paragraph or a series of bullet points explaining the concepts of Docker, Containers, and Images as if you're teaching a friend or colleague.

❖ This will help solidify your understanding and ensure you've grasped the basics.

2. Real-world Analogy Creation:

❖ The house-moving analogy was one perspective.

❖ Can you think of another real-world analogy that describes the relationship between Docker, Containers, and Images?

❖ Share your analogy with a peer or on a forum and get feedback.

3. Blog Post:

❖ Start a blog, if you haven't already. Write a post titled "Docker Demystified: My First Impressions". Cover the concepts you've learned today, and don't forget to include:

- Your personal understanding of each concept.
- The analogies that helped you grasp them.
- Any questions or curiosities you still have.

DAY 25

Docker Installation and some key points

- Docker Installation.
- Note on Using Linux (EC2 Instance).
- Post-Installation Steps for Linux.
- Why Add Your User to the Docker Group?
- Tasks.

Docker Installation:

⟶ Docker is a platform-independent tool, which means it can be run on various operating systems. However, the installation process varies slightly depending on the OS. Let's walk through the primary installation methods for the most popular platforms.

1. Docker Desktop for Windows:

- For Windows 10 64-bit (Pro, Enterprise, and Education editions), you can use Docker Desktop. It integrates nicely with Windows and provides a GUI for easy management.
- Visit the Docker Hub to download Docker Desktop for Windows.
- After installation, you can access Docker both from the Command Prompt and Docker Desktop GUI.

2. Docker Desktop for Mac:

- Similar to Windows, Docker offers a friendly desktop version for MacOS. Visit the Docker Hub to download Docker Desktop for Mac.
- Post installation, you can manage Docker containers directly from the terminal or via the GUI.

3. Docker Desktop for Linux:

⟶ Technically, Docker doesn't have a "Docker Desktop" version for Linux because Linux is Docker's native environment. Instead, you install Docker CE (Community Edition).

Note on Using Linux (EC2 Instance):

⟶ Using Docker on a native Linux environment, such as an AWS EC2 instance, is arguably the most efficient way to experience Docker. It avoids the overhead of virtualization that you might encounter on Windows or Mac.

Post-Installation Steps for Linux:

- After installing Docker on Linux, you might not want to precede every Docker command with sudo. To avoid this:
- Add your user to the docker group: sudo usermod -aG docker $USER
- This command adds the current user to the Docker group, granting permission to run Docker commands without sudo.

NOTE

Remember to log out and log back in so that your group membership is re-evaluated or type newgrp docker for the changes to take effect immediately.

Why Add Your User to the Docker Group?

⟶ By default, Docker runs as the root user, requiring users to prefix every Docker command with sudo. By adding your user to the docker group, you grant regular users permission to run Docker commands, enhancing ease of use. However, be cautious: this also means that any processes (including those in containers) that break out of Docker will have root privileges on the host machine.

Practical Exercise

1. Verify Docker Installation:

❖ Run the following command in your terminal or command prompt:

-> *docker --version*

❖ This command will display the installed version of Docker. Make sure the output aligns with the version you installed.

Practical Exercise

2. Test Docker with Hello World:

❖ Docker provides a simple "Hello World" container, which, when run, sends a Hello World message to your screen. This is a great way to confirm that Docker can download and run containers.

❖ Run the following command:

-> *docker run hello-world*

❖ Upon execution, Docker will attempt to find the "hello-world" image locally. If it's not present (which it won't be the first time you run this), Docker will fetch the image from Docker Hub and then run it. You should see a message indicating that Docker is working correctly.

3. List Docker Images:

❖ After running the Hello World container, the image is now saved on your system. You can view the images stored locally using:

-> *docker images*

❖ This will display a list of images, and you should see the hello-world image listed.

❖ The more you interact with Docker commands, the more comfortable and efficient you'll become. Don't hesitate to explore further, read the Docker documentation, or try running other containers from the Docker Hub.

DAY 26

Delving into Docker Images and Containers

- Docker Images: The Blueprints of Containers.
- Running and Interacting with Containers.
- Common Commands and Their Nuances.

Docker Images: The Blueprints of Containers

→ Think of Docker images as the blueprints of your containers. They contain the instructions to initiate a specific environment.

Practical Exercise

1. Verify Docker Installation:

- Pull a Basic Image: To fetch an image from Docker Hub, use the pull command.

  -> *docker pull ubuntu:latest*

- This command pulls the latest Ubuntu image.
- List Available Images: Once the pull is completed, check the images present on your system.

  -> *docker images*

- This will display all images you've pulled or built. You should see the ubuntu image you just pulled.

Running and Interacting with Containers:

→ Containers are instances of Docker images. When you want to run a program, you initiate a container from an image.

Practical Exercise

1. Verify Docker Installation:

❖ Run and Interact with an Ubuntu Container:

-> *docker run -it ubuntu /bin/bash*

❖ This command initializes a container from the Ubuntu image and lets you interact with it using the bash shell. The -it flags allow interactive processes (like a bash shell) to run continuously.

❖ Inside the Container: Now, you're inside the container's environment. Try these commands:

-> *ls* - Lists files and directories.

-> *pwd* - Prints the working directory.

-> *echo "Hello from inside the container!"* - Outputs the provided string.

Common Commands and Their Nuances:

-> *docker ps*: Lists running containers. To see all containers, including stopped ones, use -> *docker ps -a*.
-> *docker stop [CONTAINER_ID]*: Stops a running container.
-> *docker start [CONTAINER_ID]*: Starts a stopped container.
-> *docker rm [CONTAINER_ID]*: Removes a container. Ensure it's stopped before doing this.
-> *docker rmi [IMAGE_ID]*: Removes an image. Ensure no containers are using
the image.

Practical Exercise

1. Verify Docker Installation:

❖ Run Multiple Containers: Try pulling and running other images like nginx, alpine, or httpd. Remember to use docker ps to manage your running containers.

❖ Interact with Multiple Containers: Run two different containers and switch between their interactive shells.

❖ Clean-Up Exercise: Pull a few images, run containers from them, stop those containers, and then remove both containers and images. This will familiarize you with the lifecycle of images and containers.

❖ Environment Variables: Start a container with an environment variable and print it. Use *-e* flag. For example:

-> *docker run -e MY_VARIABLE="Hello Docker" ubuntu echo $MY_VARIABLE*

❖ Remember, these exercises will help you understand the practical side of Docker. The more scenarios you experience, the better equipped you'll be in real-world applications.

DAY 27

Into Docker Networking

- Introduction.
- Docker Networking Basics.
- Networking with standalone containers | Docker Docs.
- Tasks.

Introduction:

→ Docker, at its core, is about creating isolated environments for your applications. But often, these isolated containers need to talk to each other, to the host machine, or to the external world. Today, we dive into how Docker handles networking and data storage.vice versa

Docker Networking Basics:

→ Every Docker container is assigned its IP address and can communicate with other containers using that IP. Docker provides different network modes, each serving a specific purpose:

1. **bridge:** The default network type. Creates a virtual bridge between the host and containers. Containers can communicate with each other and access the internet, but are isolated from the host network.
2. **host:** Containers share the host network stack and see the host's network interfaces directly. They are not isolated and port bindings are published directly on the host.
3. **overlay:** Used for distributed container networks across multiple Docker hosts. Allows containers on different hosts to communicate directly.
4. **macvlan:** Assigns each container a unique MAC address, allowing them to appear as physical devices on the host network.
5. **ipvlan:** Provides precise control over IP addresses and VLAN tags for containers.

NOTE

The bridge network is suitable for most use cases where some network isolation is needed. The host network is useful when you want to expose container ports directly on the host.

Docker networking works by manipulating iptables rules to route traffic to containers. Containers have their own network namespaces that provide isolation.

Common Commands and Their Nuances:

-> *docker network create*
-> *docker network connect*
-> *docker network disconnect*
-> *docker network rm*

When using Docker Compose, services within the same Compose file are automatically connected to a default network, allowing them to communicate. You can also define custom networks that only some services are connected to.

Networking with standalone containers | Docker Docs

- What is the default Docker network used for container communication?
- Besides ip addresses, what other names can be used for container communication?
- How do you determine a container's IP address on a Docker net U have to search for the above questions...

Docker Networking | project

TASKS

- Inspect Default Network: Use the following command to inspect the default bridge network.
  -> docker network inspect bridge
- Run Containers in Different Network Modes: Start containers in bridge, host, and none network modes and observe the differences.

Common Commands and Their Nuances:

-> *docker network ls*: Lists all networks.
-> *docker network create --driver [TYPE] [NAME]*: Creates a custom network of a specified type.
-> *docker network rm [NAME]*: Removes a specified network.

Practical Exercise

1. Communication Between Containers:

- Run two containers inside the same custom bridge network.
- Install ping or curl utilities and test communication using container names.

2. Isolation:

- Run two containers on different networks and attempt communication.
- Observe the isolation provided by Docker's network modes.

3. Exploring Docker Network Drivers:

- Apart from bridge and host, explore other network drivers like overlay and macvlan.
- Research their use cases.

NOTE

By understanding Docker's networking capabilities, you're equipped to set up complex, interconnected applications with multiple containers that can securely and efficiently communicate with each other.

DAY 28

Docker Data Management - Volumes and Bind Mounts

- Introduction.
- Understanding Docker's Storage Mechanics.
- Tasks.

Introduction:

→ For containers to be truly portable, they shouldn't store data within them. This is where Docker volumes come in. They provide the ability to store and manage data outside of the container's lifecycle.

→ Handling data and ensuring its persistence across container lifecycles is a fundamental aspect of containerization. Docker offers various mechanisms, primarily Volumes and Bind Mounts, to manage data. Understanding these is essential, especially when deploying applications that require data persistence like databases.

Understanding Docker's Storage Mechanics:

→ By default, the data in a container is ephemeral; it gets deleted when the container is removed. This is where volumes and bind mounts come into play, allowing data to persist beyond a container's life.

→ **Volumes:** Managed by Docker, volumes are the preferred mechanism for persisting data. They are stored outside of the container filesystem, ensuring data isn't lost when the container is deleted.

→ **Bind Mounts:** This relies on the directory structure of the host machine. A directory (or file) on the host system is mounted into a container. It's suitable for specific scenarios, such as developing applications where you need to reflect changes immediately inside the container.

Practical Exercise

1. Create a Volume:

❖ -> *docker volume create myvolume*

2. Inspect the Volume:

❖ -> *docker volume inspect myvolume*

3. Run a Container with Volume Attached:

❖ -> *docker run -d --name=mycontainer -v myvolume:/app nginx*

4. Bind Mount a Host Directory:

❖ -> *docker run -d --name=mybindcontainer -v*

-> */path/on/host:/path/in/container nginx*

❖ Task: Database Persiste ❖

1. Run a MySQL Container without Volumes:

❖ -> *docker run --name mysqltest -e MYSQL_ROOT_PASSWORD=my-secret-pw -d*

-> *mysql:latest*

❖ Add some data into the database ❖

❖ Delete and Recreate the Container:

Observe that the data is lost.

Run a MySQL Container with a Volume:

-> *docker run --name mysqlvol -e MYSQL_ROOT_PASSWORD=my-secret-pw -v*

-> *mysql_data:/var/lib/mysql -d mysql:latest*

❖ Again, add some data into the database.

❖ Delete and Recreate the Container:

❖ Now, the data should persist, showcasing the importance of volumes in stateful applications.

KEY POINTS

1. **Volume Manipulation:** Create, inspect, and delete volumes. Practice attaching them to different containers and observe data persistence.
2. **Bind Mount Practice:** Make changes to files on your host system and observe how they reflect inside a container when using bind mounts.
3. **Multiple Mount Points:** Run a container with both a volume and a bind mount attached to different paths. Explore the nuances between them.

NOTE

Understanding Docker's data management mechanics ensures you can design systems that preserve critical data across container restarts, deletions, and migrations.

DAY 29

Docker Compose - Orchestrating Multi-Container Applications

- Introduction.
- Understanding Docker Compose.
- Tasks.
- Real-World Task: WordPress Deployment.

Introduction:

→ In many real-world scenarios, applications aren't just a single container but rather a set of interlinked containers that work together. For instance, a web application might consist of a web server, a database, and a cache. Managing such multi-container setups individually can be cumbersome. Docker Compose offers a solution by allowing you to define and run multi-container Docker applications.

Understanding Docker Compose:

→ **Docker Compose:** A tool for defining and running multi-container Docker applications. With Compose, you use a docker-compose.yml file to configure your app's services, networks, and volumes.

Practical Exercise

1. Install Docker Compose:

- Depending on your OS, follow Docker's official documentation to install Docker Compose.

2. Simple Docker Compose File:

- Create a docker-compose.yml file for a simple Python Flask application with a Redis database.

```
version: '3'
services:
  web:
    image: "flask:latest"
    ports:
      - "5000:5000"
  redis:
    image: "redis:latest"
```

Practical Exercise

3. Run the Compose File:

❖ -> *docker-compose up*

4. Stop the Compose Services:

❖ -> *docker-compose down*

❖ Real-World Task: WordPress Deployment ❖

❖ Deploy a WordPress site using Docker Compose, showcasing a real-world use case of multi-container orchestration. Your docker-compose.yml might look something like this:

```
version: '3'

services:
  db:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: somewordpress
      MYSQL_DATABASE: wordpress
      MYSQL_USER: wordpress
      MYSQL_PASSWORD: wordpress

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wordpress
      WORDPRESS_DB_PASSWORD: wordpress
volumes:
  db_data: {}
```

Practical Exercise

- **Docker Compose Commands:** Familiarize yourself with commands such as docker-compose start, docker-compose stop, docker-compose ps, and docker-compose logs.
- **Multi-Service Networking:** Explore how services in the same Docker Compose setup communicate with each other using service names as hostnames.
- **Environment Variables with Docker Compose:** Use environment variables in your docker-compose.yml file to make your setup more configurable.

NOTE

Docker Compose streamlines the management and orchestration of multi-container applications, making complex deployments as simple as writing a configuration file and running a command. As applications grow in complexity, Docker Compose becomes an invaluable tool in a developer's toolkit.

DAY 30

Building Custom Docker Images - Dockerfile Basics

- Introduction to Dockerfile.
- Tasks.
- Real-World Task: Customized Database Image.

Introduction:

→ **Dockerfile:** It's a textual script that contains all the commands a user would call to assemble an image. Using the docker build command, Docker reads these instructions and constructs a final image.

Topics Covered:

→
- **Understanding Base Images**: Starting point for building your image.
- **Dockerfile Directives**: FROM, RUN, CMD, ADD, COPY, EXPOSE, WORKDIR, ENV, and others.
- **Layering in Docker Images**: Every instruction in a Dockerfile creates a new layer in the image.

Tasks:

- 1. **Your First Dockerfile:** Create a basic Dockerfile to set up a simple Python Flask web application. Here's a basic example:

```
# Use an official Python image as a parent image
FROM python:3.8-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install Flask
RUN pip install flask

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Tasks:

2. Building the Image: Use the following command to build your Docker image:

❖ -> *docker build -t flask-app:latest .*

3. Running Your Custom Image: After building, run your custom Flask app image:

❖ -> *docker run -p 4000:80 flask-app:latest*

❖ Real-World Task: WordPress Deployment ❖

❖ Suppose your application requires specific data pre-loaded into a database. Instead of manually entering data each time a container starts, you can customize a MySQL image to have data pre-loaded.

❖ You'll need SQL scripts to set up the database schema and load data. Create a Dockerfile that starts from the official MySQL image, adds your scripts, and configures the container to run them on startup.

Hands-On Exercises:

❖ **Optimizing Image Size:** Explore using different base images, such as Alpine variants, to reduce the size of your final image. Compare the sizes using docker images.

❖ **Using .dockerignore:** Like .gitignore, .dockerignore helps you exclude files that aren't needed in the image, resulting in faster builds and smaller images.

❖ **Multi-stage Builds:** Create Dockerfiles that use multi-stage builds. This advanced technique helps in creating optimized images especially useful for compiled languages.

❖ **Tagging and Pushing Images:** Tag your custom images and push them to Docker Hub or another registry. Understand the importance of tagging in versioning and distribution.

DAY 31

Consolidation and Creation of a Docker Cheat Sheet

- Tasks.
- Bonus Task.

Tasks:

- For today's exercise, your mission is to create a comprehensive Docker cheat sheet. This is a summary of all the important Docker commands and concepts you've learned over the past days. This task serves multiple purposes:

1. Reinforcement:

- By revisiting all the Docker commands and concepts, you're reinforcing your memory and understanding.

2. Quick Reference:

- Once you've made your cheat sheet, you can refer back to it anytime you need to quickly remember a Docker command or concept.

3. Sharing & Feedback:

- By sharing your cheat sheet online, you can help others in their Docker journey and get feedback which might help you refine your understanding further.

NOTE

Also dont forgot to mention Best Practices, for example while creating multi stage dockerfile.

Bonus Tasks:

- If you're up for a challenge, try to make your cheat sheet interactive. For instance, you could create a web page where clicking on a Docker command displays its description and example.

PART 7

CI/CD - The Heartbeat of DevOps

Continuous Integration (CI) and Continuous Deployment/Delivery (CD) are software development practices that emphasize frequent commits to the main branch and automated testing and deployments.

DAY 32

CI/CD - The Heartbeat of DevOps

- Understanding CI/CD - The Foundation of Modern DevOps.
- Key Concepts in CI/CD.
- Benefits of CI/CD:
- CI/CD in the Real World:
- Stepping into Jenkins - The Automation Maestro
- Setting Up and Getting Started with Jenkins:
- Your First Dive into Jenkins:
- Jenkins in the CI/CD World:

Understanding CI/CD - The Foundation of Modern DevOps

→ Continuous Integration (CI) and Continuous Deployment/Delivery (CD) are software development practices that emphasize frequent commits to the main branch and automated testing and deployments.

Key Concepts in CI/CD:

→ **Continuous Integration:** Involves automatically testing each change done to your codebase, ensuring early detection of integration bugs.

→ **Continuous Delivery:** Ensures your codebase is always in a deployable state, making releases painless and quick.

→ **Continuous Deployment:** A step further than Continuous Delivery - every change that passes all stages of the production pipeline is released automatically. No human intervention!

CI/CD in the Real World:

⟶ Consider an example where a team is building a web application:

- Developers commit changes to the repository.
- A CI server automatically picks up the changes, runs unit and integration tests.
- If tests pass, the CD process deploys the application to a staging server.
- Further tests (UAT, load tests) are run.
- If everything looks good, the application is deployed to production.
- Monitoring tools observe the application in production to ensure everything runs smoothly.

Tasks/Exercises:

1. Identify Stages in Your Workflow: Reflect on a past project or an ongoing one. Can you identify stages that can be integrated into a CI/CD pipeline?
2. Tool Research: There are many tools in the CI/CD ecosystem. Make a list of tools you've heard of or used (like Jenkins, Travis CI, CircleCI) and write a brief note on each.

1. Stepping into Jenkins - The Automation Maestro

❖ What's Jenkins?

Jenkins, an open-source automation server, fits perfectly into the CI/CD ecosystem, enabling automated builds, tests, and deployments.

2. Setting Up and Getting Started with Jenkins:

❖ **Installation**: Jenkins is versatile, offering both a standalone application and integration into Java application servers.

❖ **Configuration**: On the first launch, Jenkins offers a setup wizard guiding users through plugin selection and admin user creation.

Your First Dive into Jenkins:

1. **Your First Jenkins Job:** Create a 'Freestyle project', add basic build steps, and watch Jenkins work its magic.
2. **Integration with SCM:** As you evolve, Jenkins seamlessly integrates with Source Code Management tools (like Git), pulling the latest code for build and deployment.

Jenkins in the CI/CD World:

→ **Pipelines:** Jenkins supports the creation of pipelines, providing an integrated suite of plugins to support building, deploying, and automating any project.

→ **Distributed Builds:** Jenkins' master-slave architecture allows the distribution of builds, running builds on different machines, and environments.

Tasks and Exercises:

❖ Dive into Jenkins: Install Jenkins and set up a simple job. Get familiar with the interface.

RESOURCES & READINGS

- The CI/CD Process in a Cloud-Native World
- What is cloud native continuous integration? |GitLab
- Tutorials overview official docs.

DAY 33

Building and Managing Jobs in Jenkins

- Understanding Jenkins Jobs.
- Managing and Monitoring Jobs.
- Tasks/Exercises.

Understanding Jenkins Jobs

→ A job in Jenkins represents a runnable task which is controlled and monitored by Jenkins. It's the cornerstone of Jenkins' functionality, enabling automation in CI/CD pipelines.

Types of Jobs in Jenkins:

- **Freestyle projects:** The simplest type of job; great for basic build and deploy tasks.
- **Pipeline projects:** More advanced, allowing complex workflows by defining stages and steps.
- **Matrix projects:** Useful to test on multiple platforms or configurations simultaneously.
- **Maven projects:** Specifically tailored for Maven (Java) projects.
- **External projects:** Jobs that manage tasks running outside of Jenkins.

Managing and Monitoring Jobs:

- **Build History:** Every time a job runs, Jenkins keeps a record. You can check the build history to see past runs, their status, and logs.
- **Workspace:** Each Jenkins job has a workspace directory where Jenkins runs the build. It's a directory on the Jenkins master/node machine.
- **Console Output:** For every build, Jenkins provides console output logs, which detail everything that happened during the build. Crucial for troubleshooting.

Tasks/Exercises:

1. Create Different Job Types:

❖ Set up a Freestyle job for a basic build.

❖ Create a Matrix job to simulate a multi-configuration test.

❖ Define a simple Pipeline job with at least two stages.

2. Create a Freestyle Pipeline to Print "Hello World!":

❖ Set up a Freestyle project in Jenkins.

❖ In the build step, select 'Execute shell' or 'Execute Windows batch command' based on your system.

❖ Enter the command to echo "Hello World!".

❖ Save and run the job. The console output should display "Hello World!".

3. Analyze Console Output:

❖ Trigger a few builds and examine the console output. Can you understand the flow of events?

RESOURCES

- EASIEST Jenkins CICD Pipeline Tutorial for DevOps Engineers // DevOps ...

DAY 34

Jenkins Integrations and Plugins

- Introduction to Jenkins Plugins.
- Essential Jenkins Plugins.
- Integrating Jenkins with Version Control Systems (VCS).

Introduction to Jenkins Plugins

→ Plugins are essential for Jenkins. They represent the tools that allow Jenkins to integrate with various software tools, transforming it from a simple automation server to a powerful CI/CD tool.

1. What are Jenkins Plugins?

→ They are essentially tools designed to increase Jenkins' functionality, allowing it to integrate with almost every part of the CI/CD toolchain.

2. Installing and Managing Plugins:

- Go to the Jenkins dashboard, select 'Manage Jenkins', then 'Manage Plugins'.
- There are four tabs: Updates, Available, Installed, and Advanced.
- You can browse available plugins, check for updates to currently installed ones, and see which plugins you've already added.

Essential Jenkins Plugins:

- Source Code Management (SCM) Plugins: e.g., Git, GitHub, Bitbucket.
- Build Tools Plugins: e.g., Maven, Gradle.
- Containerization & Deployment Plugins: e.g., Docker, Kubernetes.
- Notification Plugins: e.g., Email Extension, Slack Notification.
- UI/UX Plugins: e.g., Blue Ocean, Simple Theme Plugin.

Integrating Jenkins with Version Control Systems (VCS):

→ Most software projects today use VCS like Git. Jenkins plugins for VCS allow seamless integration.

Tasks/Exercises:

1. Plugin Exploration:

❖ Navigate to the 'Available' tab under 'Manage Plugins'. Browse and familiarize yourself with some of the popular plugins available.

2. Integrate Jenkins with Git:

❖ If you haven't done this yet, integrate Jenkins with a Git repository and ensure that Jenkins can pull the latest code.

3. Notification Setup:

❖ Install a notification plugin (like Slack Notification). Integrate it with a Slack workspace and set Jenkins to send a message upon successful builds.

RESOURCES

- Online Resource: Jenkins' official guide on managing plugins.

NOTE

Jenkins' strength is its adaptability, which is largely due to its vast plugin ecosystem. Whether it's source code management, building, testing, deploying, or even notifications, there's likely a plugin for every tool you need.

DAY 35

Integrating Jenkins with Docker

- Why Integrate Jenkins with Docker?
- Setting up Jenkins inside a Docker Container.
- Integrating Jenkins with Version Control Systems (VCS).
- Running Jenkins in Docker.
- Granting Docker Access.

Why Integrate Jenkins with Docker?

→ Jenkins and Docker are two powerful tools, and when integrated, they can automate the entire lifecycle of applications in a CI/CD pipeline. By combining Jenkins, a continuous integration tool, with Docker, a containerization tool, you can streamline builds, tests, and deployments in isolated environments.

Setting up Jenkins inside a Docker Container:

Running Jenkins in Docker:

1. You can run Jenkins itself inside a Docker container. This ensures that Jenkins always runs in the same environment.

```
docker run --name jenkins-docker -p 8080:8080 -p 50000:50000 -v
/var/run/docker.sock:/var/run/docker.sock jenkins/jenkins:lts
```

But better way try to install it manually!

Granting Docker Access:

→ Jenkins inside a Docker container will need access to the Docker daemon to run commands. By mounting the Docker socket, you allow the Jenkins container to communicate with the Docker daemon.

Tasks/Exercises:

1. Dockerized Jenkins Job:

❖ Create a Jenkins project that clones a repository containing a Dockerfile and a docker-compose.yml.

❖ Add a build step that runs docker-compose up -d to start multiple containers defined in the compose file.

❖ As a post-build step or a cleanup step, run docker-compose down to stop and remove the containers.

2. Docker Build and Push:

❖ Create a Jenkins job that builds a Docker image from a Dockerfile in the SCM repository and then pushes it to DockerHub or another container registry. Make sure to use appropriate credentials for pushing.

3. Parameterized Docker Commands:

❖ Create a Jenkins job that accepts parameters like IMAGE_NAME and TAG. Use these parameters in Docker commands within the job to allow dynamic operations.

4. Cleanup Old Docker Images:

❖ Schedule a Jenkins job that runs periodically (e.g., once a week) to remove old Docker images from the system, freeing up space.

RESOURCES

- Live DevOps Project for Resume - Jenkins CICD with GitHub Integration

NOTE

Integrating Jenkins and Docker unlocks powerful CI/CD capabilities. Containerizing builds ensures consistency, and automating Docker operations with Jenkins ensures efficiency.

DAY 36

Jenkins Declarative Pipelines: Deep Dive

- Introduction to Jenkins Pipelines.
- Declarative vs. Scripted Pipelines.
- Advantages of Declarative Pipelines.
- Creating Your First Declarative Pipeline.
- Tasks/Exercises.

Introduction to Jenkins Pipelines:

→ Jenkins Pipelines provide an extensible set of tools for modeling simple-to-complex delivery pipelines as code. The "as code" nature of Pipelines means the pipeline, its stages, and the actions to take at each step are saved as a script. This script can then be managed in source control, promoting collaboration and reusability.

Declarative vs. Scripted Pipelines:

→ At the core, both Declarative and Scripted Pipelines are fundamentally based on the Groovy DSL, but there are key differences in their design and syntax.

1. Declarative Pipelines:

Structured and Simplified: Use a more rigid structure and offer a simpler, more straightforward syntax.
Jenkinsfile: Typically begins with a pipeline block.
Syntax: Each section of the pipeline is predefined. You can't write arbitrary Groovy code.

2. Scripted Pipelines:

Flexible and Complex: They are based on a subset of the Groovy scripting language. Hence, you can use most Groovy language features.
Jenkinsfile: Starts with a node block.
Syntax: More flexibility, but with greater complexity.

Advantages of Declarative Pipelines:

→ Readable Structure: Easier to read, making it better suited for beginners.
Built-in Error Checking: Contains a linter tool that can be used to verify the syntax.
Stage-based View: Better visualization in the Jenkins UI, with stages clearly defined.

RESOURCES

- Jenkins Declarative CI/CD Pipeline for DevOps Engineers // Live Project ...

Creating Your First Declarative Pipeline:

→ The essence of a Declarative Pipeline is the clear definition of stages and steps.

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo 'Building the application...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing the application...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying the application...'
            }
        }
    }
}
```

Tasks/Exercises:

1. Create a Basic Declarative Pipeline:

❖ Using the above example, set up a new Jenkins job. Define it as a Pipeline job and use the provided script as the Pipeline script.

2. Pipeline Visualization:

❖ After running the job, view the Blue Ocean visualization. Notice how each stage is visually represented.

3. Introduce Errors:

❖ Modify the Pipeline script to introduce errors intentionally, like a missing steps block. Use the built-in linter to validate the script and notice the feedback.

RESOURCES

- Declarative vs. Scripted: Pipeline
- The Apache Groovy programming language - Documentation

DAY 37

Scaling Jenkins: Master-Agent Architecture

- Jenkins Master (Server).
- Jenkins Agent.
- Why Master-Agent Architecture?
- Pre-requisites for Setting up an Agent.

Jenkins Master (Server):

→ The heart of the Jenkins setup. The master manages, schedules, and monitors the build system. It's the orchestrator, directing operations:

- **Primary Role:** Orchestrating the entire CI/CD workflow, from source code management to deployment.
- **Key Configurations:** All vital settings and configurations reside here.
- **User Interface:** Users interact with Jenkins through the web UI served by the master.

Jenkins Agent:

→ Agents are the executors. They do the heavy lifting:

- **Execution Role:** Performs the actual steps defined in the job. From cloning repositories, building the code, to tests and deployments.
- **Labelled Identity:** Each agent has a unique label to distinguish it, helping direct specific tasks or builds to particular agents.
- **Distribution:** By adding more agents, Jenkins can distribute the workload, running multiple jobs in parallel across different environments.

Why Master-Agent Architecture?

→ A single Jenkins master can manage a small team or project. But as teams, projects, or build complexities grow, distributing the load becomes vital. Agents assist in:

- **Scalability:** Handle more builds and jobs simultaneously.
- **Versatility:** Different agents can have different environments, tools, or configurations.
- **Efficiency:** Reduce the load on the master, ensuring it remains responsive.

Pre-requisites for Setting up an Agent

- Fresh Ubuntu 22.04 (or any other OS) installation.
- Ensure Java is installed, matching the version on the Jenkins master.
- Docker is installed if containerized builds are required.
- Ensure correct rights, permissions, and ownership for Jenkins users.

Tasks/Exercises:

1. Setting Up a Jenkins Agent:

❖ Create a new AWS EC2 instance.

❖ Install the necessary software (Java, Docker).

❖ From Jenkins master, navigate to "Manage Jenkins" -> "Manage Nodes and Clouds" -> "New Node".

❖ Name your node, select 'Permanent Agent', and specify the number of executors.

❖ Enter the SSH key details and ensure the master can communicate with the agent.

❖ Verify the agent's status under the "Nodes" section.

❖ Guide: Reference the provided article for a step-by-step walkthrough.

2. Run Jobs on the New Agent:

❖ Go back to your Jenkins jobs created on Day 26 and Day 27.

❖ Update the job configurations, specifying the new agent's label in the "Restrict where this project can be run" section.

❖ Trigger the jobs and monitor their execution on the new agent.

3. Agent Environment Validation:

❖ Create a new Jenkins job.

❖ Design the job to print the environment variables (env command) and installed software versions (like java -version and docker --version).

❖ Execute the job on the agent to verify its setup.

4. Parallel Execution:

❖ Design a Jenkins job that has parallel stages.

❖ Configure each stage to run on a different agent (including master). This will help you understand load distribution and parallel execution.

ENDING OF THE PART ONE

Engage with your community! Post updates, ask questions, and share your progress using the #90DaysOfDevOps hashtag.

TRAIN WITH SHUBHAM

And most importantly, take a moment to reflect on your learning journey after five intense weeks.

Meet Shubham Londhe

Shubham Londhe is a YouTuber, and an experienced Software Engineer and Technical Trainer, passionate about DevOps and development, whose vision is to transform everyone's career into IT, irrespective of their background.

Editor & Designer - Md Shoaib

Content Researcher - Prajwal Deshpande