Week 1 Learnings

By: Suyash Nepal

Made with <3, with **slidev**

https://sli.dev

Day 01: Linux Setup and Basic Terminal Commands

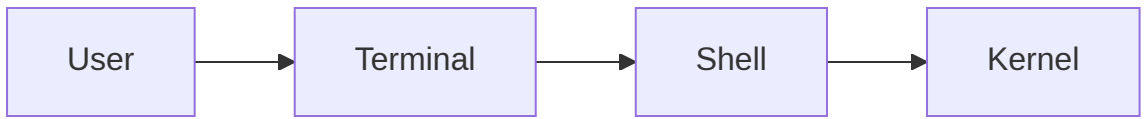
Topics Learned:

1. `UNIX` / `macOS` Terminal
2. Terminal vs. GUI
3. Command Structure Basics
4. Navigation and File Operation Commands

UNIX vs. macOS Terminal

Terminal is a *wrapper program* that runs the `shell`. In simple terms, it can be viewed of as a text-based bridge to the computer's operating system.

Viz.



Terminal vs. GUI

A Graphical User Interface is aimed at improving Human Computer Interaction by implementing visual cues (buttons, forms, designs, etc.) such that the end-user can interact with the GUI Program using Keyboard, Mouse and various other input sources.

A Terminal, on the other hand, processes textual commands via the `shell` it runs, and displays the resulting output to the *terminal screen*. Terminal commands are super lightweight, and allow for executing complex, repetitive tasks to *power users*.

Command Structure Basics

Linux/Unix follows a strict structure for its commands: `command [-options] [arguments]`

For instance, the **command** (e.g. `ls`, `cd`, `rm` tells the system *what* to do), the **options/flags** (prefixed with `-` or `--`) modify how the command behaves, and the **arguments** are the target objects (files, directories) the command acts upon.

Examples:

```
cp -r /source /destination
mv -i file1.txt file2.txt
rm -rf /tmp/folder
mkdir -p /a/b/c
```

Day 02: Navigation and File Commands

Navigation and File Operations are managed through command line utilities. Key commands that allow the user to navigate:

- `ls` for listing contents of a directory,
- `cd` for navigating through directories,
- `pwd` for viewing current directory,
- `find` for searching for contents in the system/directory
- `tree` (not a standard utility) for viewing directory tree structure
  - (e.g. `tree -L 2` shows the directory tree at `max depth` of `2`)

Some Navigation Shortcuts

- `/` refers to the root directory of the entire file system
- `.` refers to current directory
- `..` refers to the parent directory
- `~` refers to user's home directory
- `-` switches to the previously visited directory (e.g. `cd -`)

File Operation Commands

- `touch` creates a new empty file or updates timestamp of existing file
- `mkdir` creates a new directory
- `cp` copies files or directories (with the `-r` flag)
- `mv` moves a file/directory to a new location (supplied as argument)
- `cat` displays contents of the specified file
- `less` allows for a navigable view of the contents of a file *(per page content displayed)*
- `ln` creates *symbolic* or *hard links* between files

Unix Directory Structure

`Home Folder` : In `*nix` systems (Unix or Linux), the home directory is typically at `/home/username/` . If a system contains multiple users, the `/home/` directory contains Home Folders per user *(E.g.* `/home/user1` *and* `/home/user2` *)*

`/etc` : folder contains system-wide global configuration files for various processes and commands. *(E.g.* `/etc/zsh/` *contains configurations for the* `z-shell` *)*

`/var` : Contains variable data files i.e. **files that are expected to change in size, frequency, or content** during normal system operation. The fact that `/var` contains *dynamic data* allows rest of the system (e.g. `/usr`) to be mounted as read-only.

`/usr` : Contains read-only, shareable data, user programs, libraries and documentation. Comprises the largest portion of a system's files. It contains user's data, rather than the essential system boot files, thus the name `usr` .

macOS Specific Folders

-

`/Users` : Equivalent to `/home` in Linux Systems, wherein it contains the personal home folders for every user account.

`/Applications` : Contains installed software, pre-installed Apple software, utilities and other third-party apps.

Day 03: Processes & the `sudo` group

Topics to learn

1. What is a process?
2. Foreground and Background Processes
3. System Monitoring
4. Package management using `brew`

What is a process?

Process is an active, executing instance of a program, representing the basic unit of work in an Operating System. The OS manages processes via a Process Control Block (PCB) to handle execution, scheduling, and resources.

A Process consists of the *executable code* (text section), *program counter*, *stack*, and *data section* (global variables).

State:

Processes move through various states, including ->

- **new** (creation),
- **ready** (waiting for CPU),
- **running** (executing instructions),
- **waiting/blocked** (waiting for I/O) and
- **terminated** (finished).

PCB

The Operating System maintains a PCB for each process, storing information like Process ID (PID), process state, program counter, and CPU registers

In Linux Systems, the `ps` command can be used to display all the processes currently running:

- `ps` can be thought of as a snapshot that gives the static list of what is running at the exact moment `Enter` is hit.
- `ps aux` : Which gives every process in the system in BSD syntax
- `ps -eF` or `ps -ef` : Which gives the same output as `ps aux` but in full-format listing

Foreground vs. Background processes

Foreground Process:

runs interactively, occupying the terminal until completion

- Properties:
  - Require user interaction (input/output)
  - The shell is blocked until the foreground process finishes or is moved to background (`Ctrl-Z` for suspension)
  - Can be terminated using `Ctrl-C` which sends a `SIGINT` signal

Background Process:

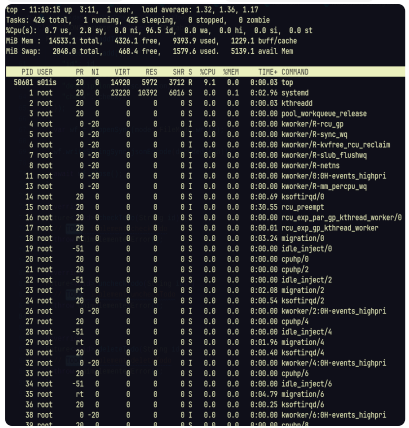
runs independently, allowing users to execute other commands simultaneously

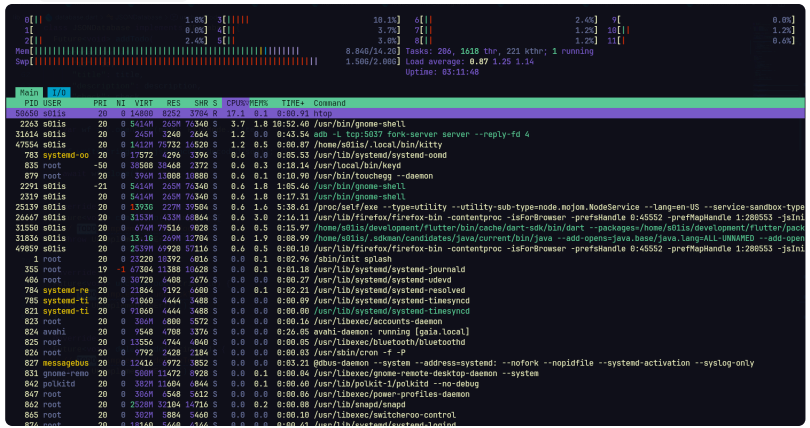
- Properties:
  - Run without direct user interaction, freeing the command prompt immediately (append the command with an ampersand `&` to make it a background process, e.g. `sleep 100 &`)
  - Useful for long-running tasks like large file transfers that don't require active monitoring
  - The output may still appear in the terminal until and unless it's redirected or piped (`>` for redirection, `|` for piping) (e.g. `(sleep 10 && echo 'hello') &` prints `'hello'` in the `stdout` after `10` seconds)

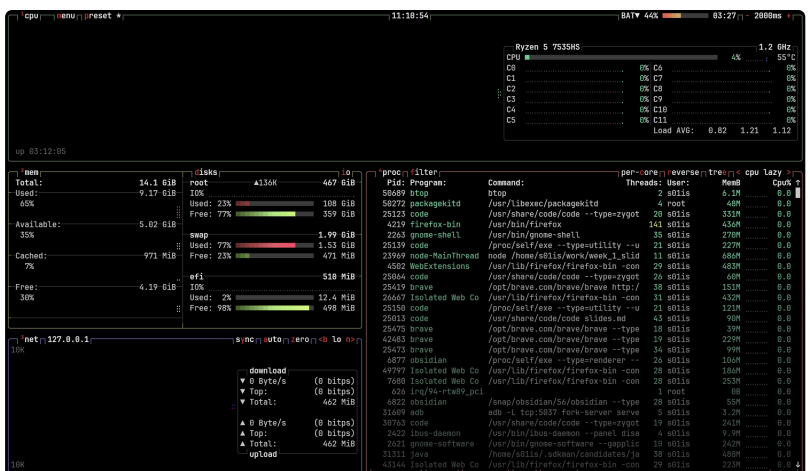
System Monitoring

- Tools like `htop`, `top` or the more advanced `btop`, can be used to monitor the system in Linux `tty`s.
- System Monitoring is the process is viewing the System's Resources, I/O Operations, Disk Usage, and the likes in a real-time fashion.
- While `top` (Table Of Processes) comes pre-installed in most vanilla Linux Distributions (Ubuntu/Fedora/Arch/Gentoo etc.) as a part of core utilities, advanced tools like `htop` and `btop` require user installation (using distro's package managers (e.g. `apt` for Debian/Ubuntu and derivatives, `pacman` / `yay` for Arch and derivatives, `dnf` / `zypper` for Fedora, SUSE and derivatives etc.), or using alternatives package managers like `homebrew` or `nix`).
  - `htop` and `btop` are widely used and loved for their colored outputs and neat, segregated views

`top` vs. `htop` vs. `btop`







Package Management using `brew`

> Package management is the process of, quite literally, managing packages. Packages are CLI utilities, GUI programs, SDKs, development libraries (DLLs), or any other "packaged" set of tools that, in tandem, build a Program (CLI/GUI).

- There are a suite of tools that can be used for managing packages, some are distribution-dependent, while others are distribution-agnostic.
- Package managers like `apt`, `aptitude` etc. are prominent in the Debian/Ubuntu space of things. While other managers like `pacman`, `dnf`, `zypper` etc. are prominent in Arch, Fedora, and SUSE distributions respectively.
- Distribution-agnostic package managers like `snap`, `flatpak`, `nix`, `homebrew`, and even `pip` (which is for installing Python tools/libraries) can be used in, theoretically, any distribution.
- **While** `brew` was primarily designed for `macOS` systems, it's now widely used in Linux as well. It allows a user to install, update, and manage software from the command line without manual downloads.

Core Commands in `brew`:

- `brew search [text]`: Finds packages matching the given query
- `brew install [package]`: Downloads and sets up the package
- `brew list`: Shows every package currently installed using `brew` in the system
- `brew uninstall [package]`: Removes the software and its dependencies cleanly
- `brew update`: Updates Homebrew itself and the list of installed packages (called the `formulae`)

Formulae and Casks in `brew`

- `Formulae`: Ruby-based scripts that define how to download, configure, and compile software from source
  - E.g. `brew install wget`, `brew install python`
- `Casks`: Ruby-based scripts that install binary packages, usually GUI applications, **without** compiling them from source (Mac-specific, apparently)
  - E.g. `brew install --cask firefox`, `brew install --cask visual-studio-code`

Other Commands

- `brew doctor` : A diagnostic tool (similar to `flutter doctor`)
- `brew upgrade` : Installs newer versions of out-of-date packages
- `brew cleanup` : Deletes old versions, cache, and saves disk space, basically

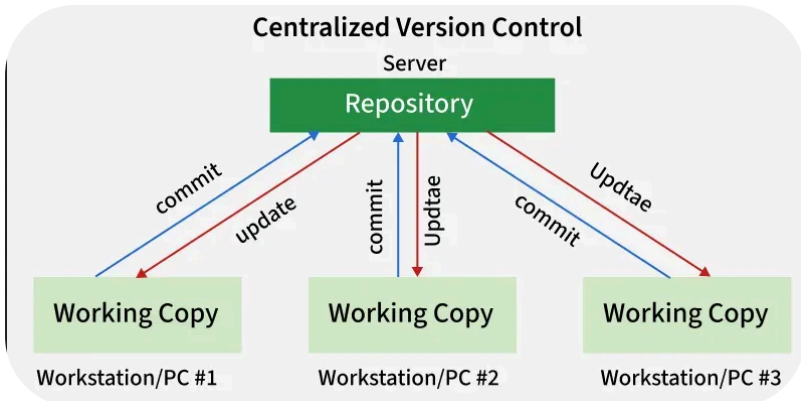
Day 04: `git` Fundamentals

Topics to learn

1. What is Version Control?
2. What is `git`?
3. Working directory in `git`
4. Staging area in `git`
5. Repository
6. Commit, Branch, Merge

What is Version Control?

Version Control is a system that *records* changes to files over time, allowing individuals or teams to *track history*, *revert to previous versions*, and *collaborate without code conflicts*.



What is `git`?

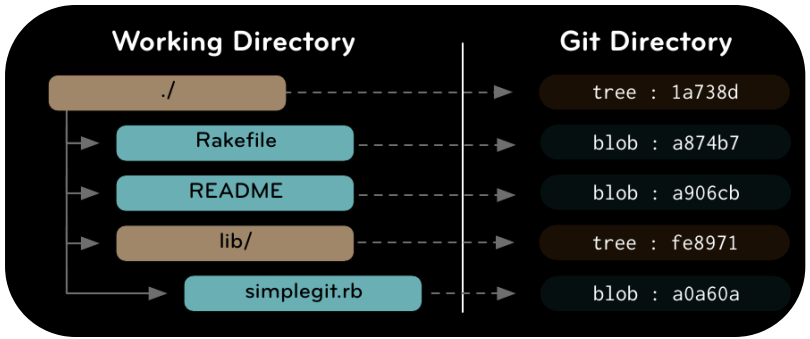
The following answer is heavily based on a beautiful book: *Git Internals* by *Scott Chacon*

- "`git` is a *stupid content tracker*."
- `git` tracks content: `files` and `directories`
- At its heart, `git` is a collection of tools that implement:
  - A tree history storage (will explore this soon) and
  - directory content management system

> **If you were asked to implement your own SCM, how would you represent the directory tree?**
> `git` does the following: *Each git repository is a Directed Acyclic Graph and each Updated File is stored as a new reference in the same.*

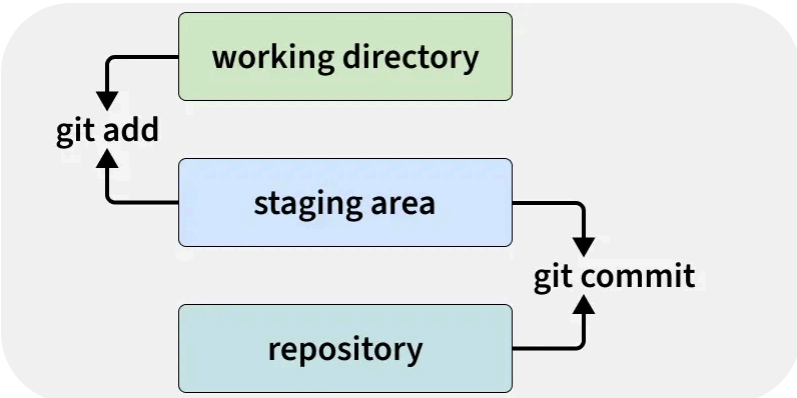
> Four types of git `objects` exist: **Blob (SHA-1 Hash of Files), Tree (Directories), Commit (Objects that represent change), Tag (unique representation of each commit object)** *If a file is Unchanged between commits (which act as parent node for blobs and trees), the same* `blob` *object is the pointee of both commit sub-trees.*

Working Directory in `git`



- The working directory is the folder on your hard drive where you see and work with the actual, editable files of your project
- Directories in `git` correspond to `trees`
  - A `tree` is a simple list of `trees` and `blobs` that the tree contains, along with the names and modes of those `trees` and `blobs`.

Staging Area in `git`



In `git`, the staging area *(also called the `index`)* is an intermediate space where changes are gathered (added) before they are committed.

The staging area allows you to:

- Selectively *choose* changes to commit.
- *Break down large changes* into smaller, logical commits.
- *Review your work* before finalizing it in the repository.

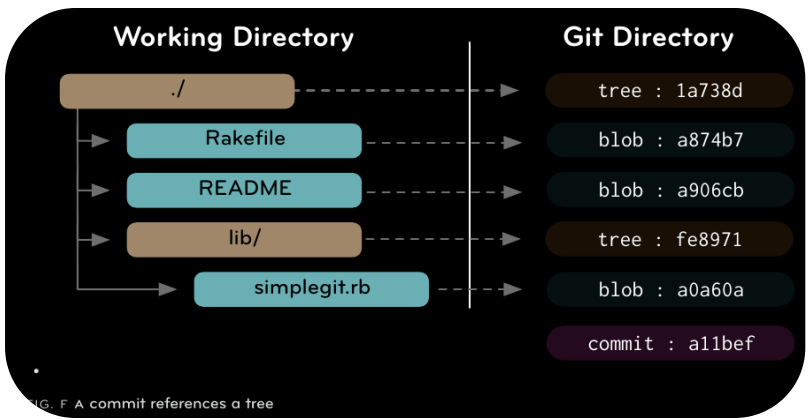
Repository in `git`

- a storage space that contains all the files for a project, along with their complete **revision history and metadata**
  - `.git/` folder located in the root directory of your project is where all the history, branches, and configuration files are stored.
    - **Deleting this folder** means deleting your **project's history**.

The `Commit` Object

> So, now that we can store arbitrary trees of content in Git, where does the '*history*' part of '*tree history storage system*' come in? The answer is the `commit` object.

The `commit` simply points to a *tree* and keeps an *author, committer, message* and any *parent commits* that directly precede it.



NG. F A commit references a tree

Branch in `git`

> The Branch is a lightweight and cheap way of **isolating** current content of the git tree. It allows for developing features in isolation, adhering to the idea of **Loose Coupling**.

- New branches can be created by:
  - `git branch <branch/name>`
- They can be navigated to by:
  - `git checkout <branch/name>` (after creating)
- The **following is equivalent to the above two commands**
  - `git checkout -B <branch/name>` (creates the new branch and checks it out)

Merge in `git`

Merging allows for multiple branches and/or commits to be merged into a single commit object

- From https://git-scm.com/docs/merge-strategies, *"The merge mechanism (`git merge` and `git pull` commands) allows the backend **merge strategies** to be chosen with `-s` option."*
  - `ort` strategy is the default merge strategy which uses a *3 way merge* algorithm

    > It takes three inputs: THIS, BASE, and OTHER. THIS is the current value in the user's tree. OTHER is the current value in the tree to be merged. BASE is a basis for comparison between THIS and OTHER, and is usually an ancestor of them.
    > The 3 way merge algo can be paraphrased as: *"keep my changes, but apply the changes OTHER made to my copy"*

Three-way merge strategy

Case	THIS	BASE	OTHER	RESULT
1.	A	A	A	A (Boring case)
2.	B	A	A	B (Take change from THIS)
3.	A	A	B	B (Take change from OTHER)
4.	A	B	A	A (AccidentalCleanMerge)
5.	A	B	C	**Conflict**

Practiced `git` commands

- `git init` : Initialize a `git` repo in the current directory
- `git status` : Show the working tree status
- `git add` : add the specified files to the staging area for future commit
- `git log` : view the history of commits, along with metadata such as authors, commit messages etc.
- `git diff` : view the line-wise difference (diff) between specified commit hashes. If no arguments are supplied, the output shows the changes that are relative to the index (the staging area)
- `git branch` : Allows for listing, creating and deleting branches.
  - `git branch <branch/name>` creates a new branch (flags like `-f` can be passed for force create)
- `git checkout` : traverse between branches and/or commit points (via their hashes)
- `git merge` : merge development histories together, typically used to merge branches. However, as per the manual pages, this command is used internally by `git pull` to merge remote to local as well.

Day 05: Flutter Basics & Setup

Covered:

- Flutter Installation (`v3.35.1`)
- Most Common of Flutter Commands
- Flutter Architecture (*widgets, state, etc.*)
- Dart Language Basics
- Flutter Project Structure
- Hot Reload vs Hot Restart

Flutter Installation

1. Get the Flutter SDK (`v3.35.1`)

```
wget https://storage.googleapis.com/flutter_infra_release/releases/stable/linux/flutter_linux_3.35.1-stable.tar.xz
```

2. Extract to an arbitrary path (since we were instructed to make the `path` the same among each other, we chose `$HOME/development/flutter/`)

```
mkdir -p $HOME/development/ # create
tar -xvf flutter_linux_3.35.1-stable.tar.xz && mv flutter/ $HOME/development/ # Move inside the above dir
```

3. Add the binary folder to `PATH` . In `~/.bashrc` or `~/.zshrc` OR, even better, in `~/.zshenv` , append:

```
export PATH=$PATH:$HOME/development/flutter/bin/
```

> Note that a shell restart is required for the binaries to be accessible

4. Install **Android Studio**, and follow the setup instructions
5. Finally, run `flutter doctor` , and resolve the issues

Most Common Flutter Commands

The following are the commands every flutter developer should be familiar with:

- `flutter doctor` - Check Flutter installation
- `flutter create <app_name>` - Create new Flutter project
- `flutter run` - Run the app
- `flutter pub get` - Get project dependencies
- `flutter pub add <package>` - Add a package
- `flutter clean` - Clean build files
- `flutter build` - Build the app
- `flutter test` - Run tests
- `flutter analyze` - Analyze code for issues
- `flutter devices` - List connected devices

For the Emulation Aficionados

If you develop using a virtual emulator, the following commands will be useful:

- `flutter emulators` - a list of available emulators
- `flutter emulator --launch <emulator_id>` - launch an emulator
- `flutter run -d <emulator_id>` - run the flutter project inside the emulator

Flutter Architecture and Components

Widgets

- The idea is that you build your UI out of widgets
- Widgets describe what their view should look like given their urrent configuration and state.
- When a widget's state changes, the widget rebuilds its description, which the framework `diffs` against the previous description in order to determine the **minimal changes needed** to transition between the old and new widget (EFFICIENCY!)

What is **State Management**?

- the process of organizing and controlling the data that determines the user interface (`UI`) of an application at any given moment.
- crucial for ensuring the `UI` stays consistent, responsive, and reflects the changes in data (e.g. user input, `API` responses) in a predictable manner.

Thinking Declaratively

> *While imperative programming focuses on **how** to achieve a result by providing explicit, step-by-step instructions,*

Declarative Programming focuses on *what* the end-result is, leaving the *implementation details* to the system or framework.

- Flutter is *declarative*, meaning that flutter builds its user interface to reflect the current state of your application.

- In Flutter, almost everything is a widget: even layout models are widgets
- You create a layout by composing widgets to build more complex widgets

But… what's a state?

`state` of an application is everything that exists in memory when the app is running. This includes, but isn't limited to, all the variables that the `Flutter` framework keeps about:

- the `UI`
- animation state
- textures
- fonts, etc.

> *the `state` is WHATEVER data you need in order to rebuild your UI at any moment in time*

Ephemeral State

The *ephemeral state*, also called *UI state* or *local state*, is the state you can neatly contain in a single widget.

While this is a vague way of putting things, the following might clarify:

- Current page in a `PageView`
- Current progress of a complex animation
- Current selected tab in a `BottomNavigationBar`

App State

- State that is NOT ephemeral (short-lived)
- State that you want to share across many parts of your application, AND that you keep between user sessions
- Also called *Shared State*

> *Examples of application state: User preferences, Login Info, Notifications in a social networking app, The shopping cart in an e-commerce app, Read/unread state of articles in a news app*