Linux - Fundamentals

m2iformation.fr

Program

- Introduction
  - Origins of GNU/Linux.
  - Definitions of free software and open-source software.
  - Description of relevant organizations.
  - Overview of various operating systems and open-source solutions.
- System Architecture
  - Basics of architecture.
  - General characteristics of system layers.
  - Graphical layers.
  - Overview of different Shells.
  - Presentation of main distributions.
  - Choosing and installing a distribution
- File System
  - File system hierarchy.
  - Different file system types.
  - Useful commands and interaction.
- Getting Started
  - Introduction to Shell and environment.
  - Introduction to graphical interface.
  - Using terminals and applications.
  - Navigating and interacting with files/folders.
  - Managing user and administrator accounts.

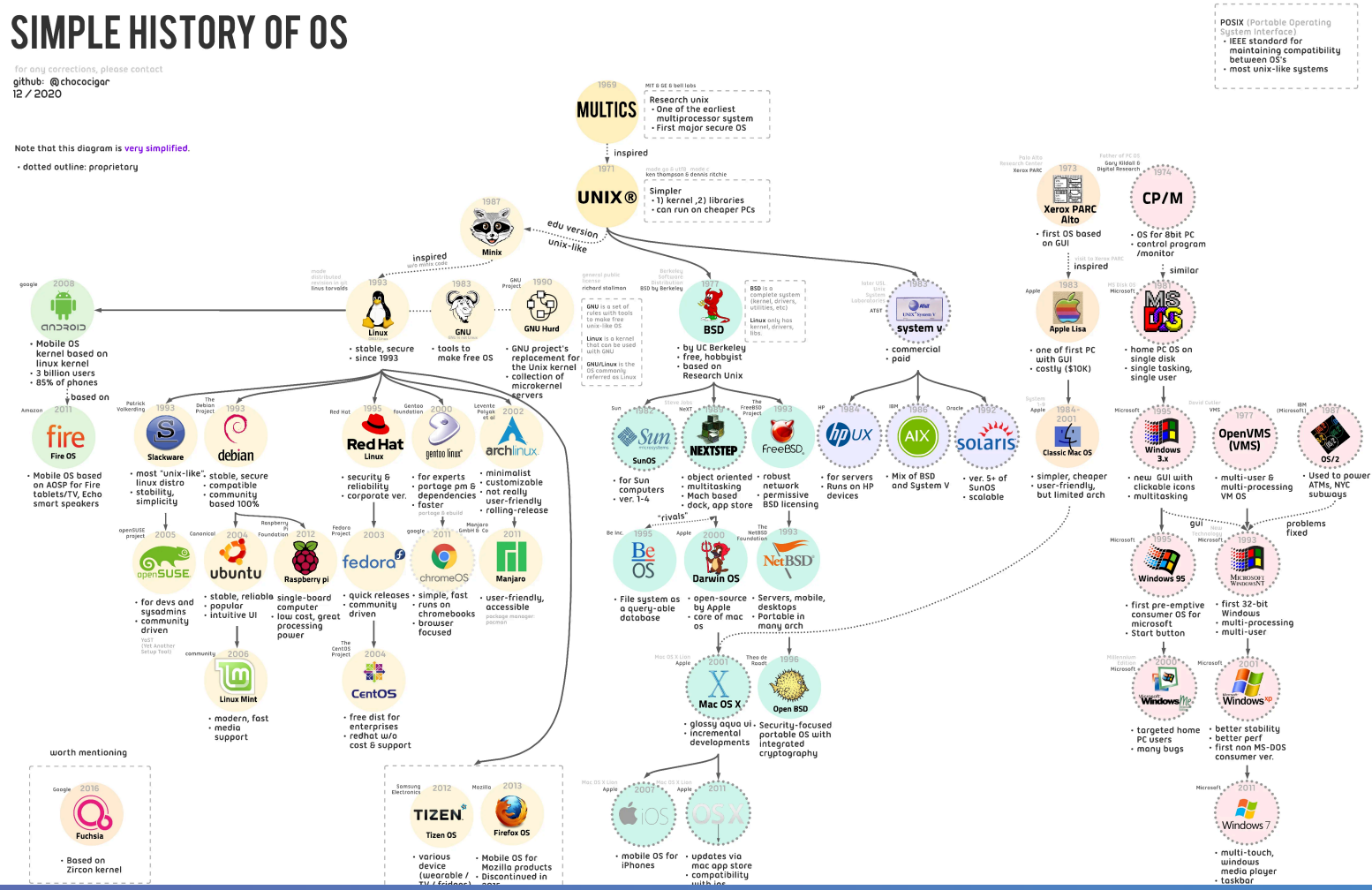
Program

- File Management
  - Commands to manage folders.
  - Reading and interacting with file content.
  - Commands for managing files.
  - Managing aliases and symbolic or hard links.
- Shell/Bash Scripting
  - Shell basics and instructions.
  - Main commands:
    - Search, capture, creation.
  - Using help and command history.
  - Managing Shell variables and exporting them.
  - Capturing command results.
  - Character escaping and protection.
  - Using pipes
- Networking
  - Networking basics.
  - Network configuration.
  - File transfers and remote connections.
- Scripting and Redirectors
  - Introduction to instructions and loops.
  - Introduction to scripting.
  - Overview of standard input (stdin), output (stdout), and error (stderr) streams.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction

Origins of GNU/Linux



The Simple History of Linux

1. The Roots – From MULTICS to UNIX

- In the **1960s**, the MULTICS project (Multiplexed Information and Computing Service) inspired the need for simpler, more efficient operating systems.
- This led to the development of **UNIX** at Bell Labs in **1969**, which became the foundation for many future operating systems.

2. MINIX – Educational UNIX

- In **1987**, **MINIX** was created as a small UNIX-like system for educational purposes by Andrew Tanenbaum.
- MINIX inspired **Linus Torvalds**, who wanted to make his own kernel.

🐧 3. Birth of the Linux Kernel (1991)

- In **1991**, Linus Torvalds released the **Linux kernel** as a personal project.
- He combined it with tools from the **GNU Project** (launched in 1983 by Richard Stallman) to create a complete **free and open-source operating system**.
- This combination is often called **GNU/Linux**.

🌱 4. The Growth of Linux Distributions

From the kernel, different communities and companies created **Linux distributions** for various use cases:

🧩 Slackware (1993)

- One of the oldest distributions.
- Aimed at simplicity and minimalism.
- Inspired many others.

📦 Debian (1993)

- Community-driven, very stable.
- 100% free software by philosophy.
- Became the **base for many other distributions** (Ubuntu, Kali, etc.).

🧱 Red Hat (1995)

- Commercial focus, support for enterprises.

🌿 5. Branching Out – Popular Distros and Their Purposes

Here's how some key distros evolved and what they bring:

🟠 Ubuntu (2004)

- Based on Debian.
- Created to be **user-friendly**, with regular releases and long-term support (LTS).
- Aimed at both desktops and servers.

🔵 Fedora

- Sponsored by Red Hat.
- Focuses on **innovation**, cutting-edge features.
- Acts as a testing ground for **Red Hat Enterprise Linux (RHEL)**.

🌿 5. Branching Out – Popular Distros and Their Purposes

🟡 CentOS

- Rebuilt version of RHEL but **free to use**.
- Designed for production environments without Red Hat subscription.

🟩 Linux Mint

- Based on Ubuntu.
- Designed to look and feel like Windows – **great for beginners**.

🧊 Arch Linux

- For advanced users.
- **Rolling release**, minimal by default, highly customizable.

🌿 5. Branching Out – Popular Distros and Their Purposes

⚔️ Kali Linux

- Built on Debian.
- Designed for **security experts and penetration testing**.

☁️ 6. Alpine Linux

- Not directly shown in detail, but Alpine fits into the minimalism-focused distributions.
- Used widely in containers (Docker).
- **Extremely small and secure**.

📱 7. Linux Beyond the Desktop

Linux powers much more than just computers:

📱 Android

- Based on the **Linux kernel**, Android is the most used mobile OS in the world.
- Over 1 billion devices run Android.

📺 Fire OS

- A fork of Android used in Amazon devices (Fire TV, Echo Show, etc.).

🧠 Tizen / Firefox OS

- Linux-based systems for IoT, smart TVs, and embedded systems.

Summary: Why Linux Matters

Aspect	Description
Open-source	Anyone can view, modify, and redistribute the code
Flexible	Used in desktops, servers, smartphones, routers, containers, supercomputers
Diverse	Many distributions tailored to different users and needs
Community	Strong support through communities, forums, and collaborative development

🔍 Definitions of Free Software and Open-Source Software

🧠 Why This Matters

When we talk about **Linux**, we're really talking about **free and open-source software (FOSS)**. But *"free software"* and *"open-source software"* are not exactly the same — they come from **different philosophies**, even though they often describe the **same software**.

🟢 What is Free Software? (According to the Free Software Foundation)

Free software means **freedom**, not price.
The **Free Software Foundation (FSF)** defines free software based on **four essential freedoms**:

1. **Freedom to run** the program as you wish, for any purpose.
2. **Freedom to study** how the program works, and change it to make it do what you wish (requires access to source code).
3. **Freedom to redistribute** copies so you can help others.
4. **Freedom to distribute** copies of your modified versions to others.

➡️ Example: The GNU tools used in Linux (like `bash`, `gcc`, and `make`) are all **free software**.
💡 **Important**: "Free" here doesn't mean "zero cost" — it means **free as in freedom**, like *free speech*, not *free beer*.

🔵 What is Open-Source Software? (According to the Open Source Initiative)

Open-source software focuses more on the **practical advantages** of sharing code, like:

- Faster development
- Better collaboration
- Code transparency

The **Open Source Initiative (OSI)** created the **Open Source Definition** with 10 criteria, such as:

- Free redistribution
- Source code availability
- No discrimination against users or fields of use

➡️ Example: Projects like **Firefox**, **Kubernetes**, and **VS Code** are open-source, and they follow these rules.

Main Difference: Philosophy

Aspect	Free Software	Open Source Software
Focus	**Ethics and user freedom**	**Practical benefits and collaboration**
Founder	Richard Stallman (FSF)	Eric Raymond, Bruce Perens (OSI)
Terms used	“Free as in freedom”	“Open for inspection and contribution”
Example attitude	“You should have control over software”	“Open code helps us make better software”

Even though they often apply to the same software, the **philosophy behind them is different**.

🔧 How It Connects to Linux

- **The Linux kernel** is licensed under the **GNU General Public License (GPL)**, which is a **free software license**.
- This means **Linux is both free and open-source**, but the people who promote it may emphasize different values:
  - Some care about **freedom** (the FSF).
  - Some care about **collaboration and innovation** (the OSI).

🟢 1. Free Software Foundation (FSF)

📌 Who they are:

- Founded in **1985** by **Richard Stallman**
- Non-profit organization based in the USA
- Created to **promote computer user freedom**

🎯 Their mission:

- Protect and promote the **freedoms of software users**
- Develop and maintain the **GNU Project**
- Educate the public on why **software freedom matters**

🛠️ What they do:

- Created the **GNU General Public License (GPL)**
- Host and support key tools like **Bash**, **GCC**, **GIMP**, and more
- Run campaigns like “Defective by Design” against DRM

🌍 Relevance to Linux:

- The **GNU tools** provided by FSF are part of what makes Linux usable (GNU/Linux).
- FSF ensures that Linux remains **free as in freedom**.

🔵 2. Open Source Initiative (OSI)

📌 Who they are:

- Founded in **1998** by **Eric S. Raymond** and **Bruce Perens**
- Non-profit focused on **defining and promoting open-source software**

🎯 Their mission:

- Provide a **clear, practical definition of open source**
- Approve and maintain the list of **OSI-approved licenses**
- Encourage businesses and developers to adopt open-source practices

🛠️ What they do:

- Maintain the official **Open Source Definition (OSD)**
- Certify licenses like **MIT, Apache, BSD, Mozilla Public License**

🌍 Relevance to Linux:

- While the Linux kernel uses a **GPL license** (from FSF), many tools in the Linux ecosystem follow **OSI-approved open-source licenses**.
- The OSI helps **increase trust and adoption** in enterprise and academic environments.

● 3. The Linux Foundation

📌 Who they are:

- Founded in **2000** (originally as Open Source Development Labs)
- Rebranded as **The Linux Foundation** in 2007
- Non-profit consortium of companies like Intel, IBM, Google, Microsoft

🎯 Their mission:

- Promote the growth of **Linux and collaborative software**
- **Fund and support Linus Torvalds** (the creator of the Linux kernel)
- Host major open-source projects (Kubernetes, Node.js, Hyperledger)

🛠️ What they do:

- Provide **certifications** (e.g. LFCS, LFCE)
- Organize conferences like **Open Source Summit**
- Maintain **technical infrastructure** for key open-source projects

🌍 Relevance to Linux:

- The Linux Foundation plays a **central role in coordinating kernel development** and ensuring Linux is **enterprise-ready**.
- It bridges the gap between **community and industry**.

🟡 4. The Debian Project

📌Who they are:

- Founded in **1993** by **Ian Murdock**
- Community-led organization that develops **Debian GNU/Linux**

🎯 Their mission:

- Build a **completely free, stable, and universal OS**
- Maintain the **Social Contract** and **Debian Free Software Guidelines (DFSG)**

🛠️ What they do:

- Develop Debian – the base for Ubuntu, Kali, and many others
- Use a **transparent, democratic governance** model

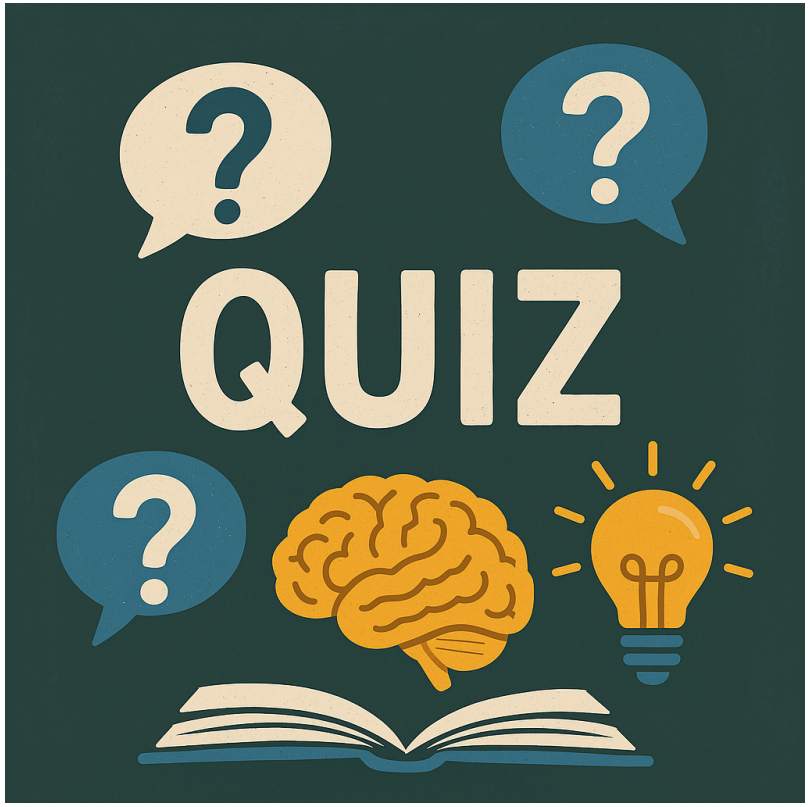
🌍 Relevance to Linux:

- Debian is a **key distribution** in the Linux ecosystem
- Its community values influence many Linux decisions worldwide

Summary Table

Organization	Year	Main Role	Key Contribution
FSF	1985	Promote software freedom	GNU tools, GPL license
OSI	1998	Define open source, certify licenses	Open Source Definition, license list
Linux Foundation	2000	Support Linux and related projects	Linux kernel funding, certifications
Debian Project	1993	Community-driven Linux distro	Debian OS, social contract

Quiz Time



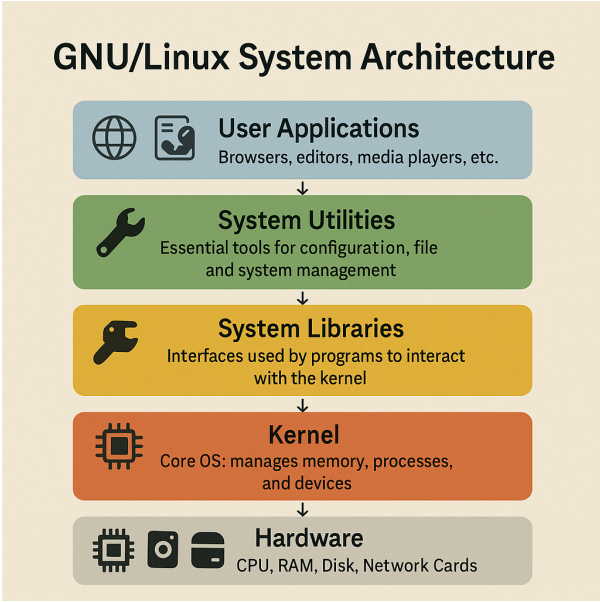
UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

System Architecture

1. Basics of Architecture

System architecture refers to the **high-level structure** of a computer system:
how its components — both **hardware and software** — are **organized and interact**.
In a GNU/Linux system, the architecture typically includes:

1. **Hardware**
   – Physical components: CPU, RAM, disk, network cards.
2. **Kernel**
   – The **core** of the OS. Manages memory, processes, devices.
3. **System Libraries**
   – Tools used by software to interact with the kernel.
4. **System Utilities**
   – Essential programs for configuration, file management, etc.
5. **User Applications**
   – Software like web browsers, text editors, media players.



1. Basics of Architecture

When you open Firefox:

- The app interacts with system libraries (like GTK).
- These libraries use system calls to request resources from the **kernel**.
- The kernel gives access to the **hardware** (screen, network, memory).

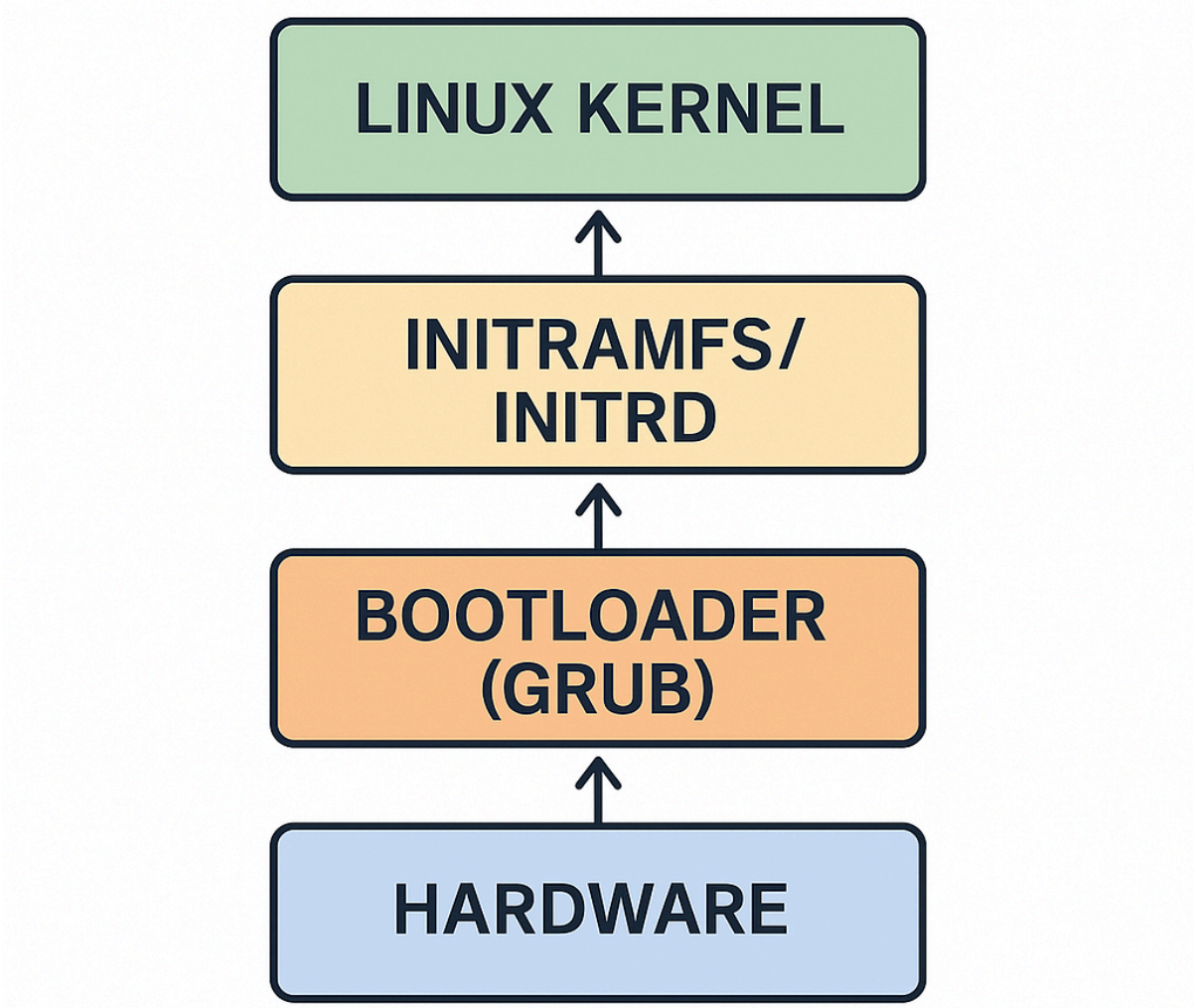
2. General Characteristics of System Layers

Think of a system as **layered** — each one depending on the layer below.

Layer	Role
Hardware	Physical machine
Kernel	Communicates with hardware and manages system calls
Shell / CLI	User interface to interact with the kernel
Utilities	Command-line and graphical tools
Applications	Software for the user (LibreOffice, Firefox, etc.)

Each layer **abstracts** the one below it — making the system easier to use without needing to interact directly with the hardware.

Full Sequence Between Hardware and the Linux Kernel



🔹 UEFI (Unified Extensible Firmware Interface)

- Modern replacement for BIOS
- Supports secure boot, GPT partitioning, graphical interface
- Loads bootloaders from the EFI partition (`/boot/efi`)
- Much more flexible and powerful than BIOS

✅ **Role:**

- Initializes hardware (POST: Power-On Self-Test)
- Finds a valid boot device (hard disk, USB, CD, etc.)
- Loads the **bootloader**

⚙ 3. Bootloader (GRUB)

What it is:
A small program that **launches the operating system**.

🔹 GRUB (GRand Unified Bootloader):

- The default bootloader on most Linux systems
- Supports:
  - Multiple OS choices (dual boot)
  - Custom kernel arguments
  - Rescue/recovery modes

✅ Role:

- Loads the **kernel image** (`vmlinuz`, `bzImage`)
- Loads the **initramfs** (optional)
- Passes **parameters** to the kernel (`root=`, `quiet`, `nomodeset`, etc.)

📦 4. Initrd / Initramfs

What it is:
A small **temporary root filesystem** loaded into **RAM** before the real root (`/`) is mounted.

- `initrd`: Initial RAM Disk (older method)
- `initramfs`: Initial RAM Filesystem (modern, more flexible)

✅ **Role:**

- Contains:
  - Drivers for disk and file systems
  - Tools for LVM, encryption (LUKS), RAID
  - Scripts to mount the actual root filesystem
- Prepares the system environment for the kernel

💡 Without `initramfs`, the kernel might not recognize your disk or file system.
🛠️ You can view or rebuild it:

```
ls /boot/init*
sudo update-initramfs -u
```

🧠 5. The Linux Kernel

What it is:
The **core** of the operating system. It's now fully loaded into memory and running.

✅ **Role:**

- Manages:
  - Memory
  - CPU scheduling
  - File systems
  - Networking
  - Drivers
- Initializes PID 1 (usually `systemd`)

After the kernel finishes booting, it **starts the first user-space process**, which is usually the **init system** (`systemd`, `SysVinit`, or others).

🔹 3. Graphical Layers (Display Stack)

Linux can operate purely from the **command line**, but most users install a **graphical layer**.

🖼️ Components of the Graphical Stack:

Component	Description
X Window System (X11)	The traditional graphics server
Wayland	A modern replacement for X11 (faster and secure)
Display Manager	Login screen (e.g., GDM, LightDM, SDDM)
Window Manager / Desktop Environment	KDE Plasma, GNOME, XFCE

🧠 Example:

- GNOME uses Wayland or X11.
- KDE Plasma can run on either.
- These components **talk to the kernel** (via drivers) to display content on your screen.

4. Presentation of Main Distributions

GNU/Linux exists in **many versions**, called **distributions**, each combining:

- The Linux kernel
- GNU tools
- A package manager
- A graphical or minimal environment

Distribution	Focus / Audience
Ubuntu	User-friendly, good for beginners
Debian	Stability, server-grade reliability
Fedora	Cutting-edge, developer-focused
Arch Linux	Rolling-release, for advanced users
Linux Mint	Windows-like experience, beginner-friendly
Kali Linux	Security testing and ethical hacking
Alpine	Minimal, container-based environments

🔹 5. Choosing and Installing a Distribution

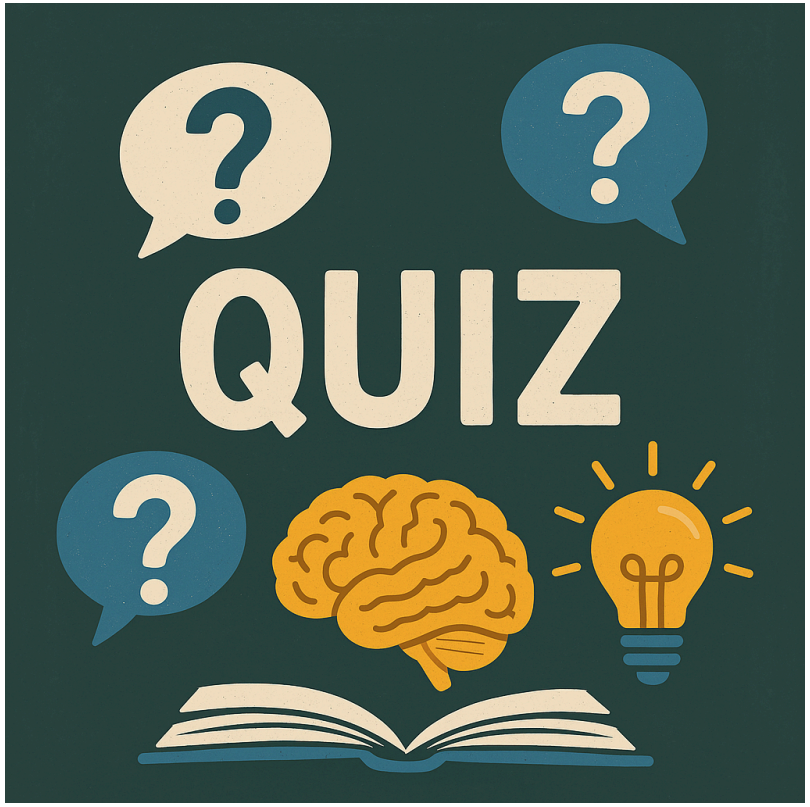
✅ How to Choose a Distribution:

Ask yourself:

- Do I need **stability** or **latest features**?
- Am I a **beginner** or **experienced user**?
- Is it for a **desktop**, **server**, or **embedded system**?
- Do I care about **privacy**, **performance**, or **security tools**?

Use Case	Recommended Distro
Desktop (new user)	Ubuntu, Linux Mint
Server	Debian, CentOS, Ubuntu Server
Security testing	Kali Linux
Cloud / Docker	Alpine Linux
Custom configuration	Arch Linux

Quiz Time



⚙ Installation Steps (general)

1. **Download ISO** from the distro's website.
2. **Create a bootable USB** with tools like Rufus or BalenaEtcher.
3. **Boot from USB** and run the live installer.
4. Choose:
    - Language
    - Keyboard layout
    - Time zone
    - Disk partitioning
    - Username and password
5. **Install and reboot!**

🧠 Advanced Linux Installation Methods

🛠️ 1. Automated Installation (Unattended / Preseed / Kickstart)

This method allows you to **automate the entire installation** process using a configuration file — no need to click through menus manually.

Method	Description
Preseed (Debian/Ubuntu)	Use a `.cfg` file with all answers to the installer questions. Often used with PXE or ISO customization.
Kickstart (Red Hat/CentOS/Fedora)	Use a `.ks` file to automate RHEL-based installations.
Autoinstall (Ubuntu 20.04+)	YAML-based replacement for Preseed. Works with cloud-init.
YaST AutoYaST (SUSE)	Use XML profiles to automate SUSE installations.

🌐 2. Network Boot (PXE Boot + TFTP + NFS or HTTP)

Boot the Linux installer over the network — no USB, CD, or ISO needed.
✅ Useful for:

- Installing Linux on multiple computers at once
- Old machines with no USB/DVD support

1. The target machine sends a PXE request via the LAN
2. A **PXE server (TFTP)** provides boot files (like GRUB, kernel)
3. An NFS or HTTP server delivers the installer and/or ISO
4. The machine boots the installer from the network

🛠️ Required Components:

- DHCP server
- TFTP server
- HTTP or NFS server for installation media
- Optional: Kickstart or Preseed file for automation

🗂️ 3. Custom ISO / Remastered Linux

You can **create a custom Linux ISO** that already includes:

- Your configuration
- Pre-installed software
- Predefined users
- Your branding

✅ Great for:

- Organizations distributing internal Linux distros
- Training or lab environments
- Rescue and recovery ISOs

🧩 Tools to build custom ISOs:

Tool	Description
Cubic (Ubuntu)	GUI tool to customize Ubuntu ISOs
live-build (Debian)	Build your own Debian ISO from scratch
Linux Live Kit	Turn an existing system into a bootable ISO

💻 4. Virtual Machine Templates (OVA, VDI, QCOW2, etc.)

Instead of installing Linux repeatedly, you can **use or build virtual machine images** that are ready to go.
✅ Useful for:

- Developers and trainers
- Students in sandboxed environments
- CI/CD test automation

🔧 Common formats:

Format	Used By
`.vdi`	VirtualBox
`.vmdk`	VMware
`.qcow2`	KVM, QEMU
`.ova`	Open Virtual Appliance

💡 Tip: You can export a configured Linux VM and let others import it directly — skipping installation entirely.

5. Cloud Images / Cloud-init Based Installation

Used in cloud platforms (AWS, Azure, GCP, OpenStack), Linux is installed from **pre-built cloud images**.

🔹 Examples:

- `ubuntu-22.04-server-cloudimg-amd64.img`
- `centos-stream-9-cloud.qcow2`

Cloud-init handles:

- SSH key setup
- User creation
- Networking
- Initial scripts

✅ Ideal for:

- Server provisioning
- Infrastructure-as-Code setups
- DevOps pipelines

🔹 6. Overview of Different Shells

A **shell** is a user interface that lets you interact with the operating system — usually through a **command-line interface (CLI)**.

🐚 Common Linux Shells:

Shell	Description
bash	Bourne Again Shell – default in most systems ✅
sh	Original Bourne Shell – very basic
zsh	Like bash but more powerful/flexible
fish	Friendly Interactive Shell – modern, easy to use
csh/tcsh	C Shell variants – C-like syntax

🔹 1. `sh` – Bourne Shell

- The **original Unix shell**, created in the 1970s by Stephen Bourne.
- Still present on all UNIX and Linux systems as `/bin/sh`.
- 🛠️ **Features:**
  - Very basic syntax.
  - No auto-completion.
  - No command history.
- 📌 **When to use:**
  - **Legacy scripts** that require POSIX compliance.
  - When writing highly portable shell scripts.
- ✅ **Why:**
  - It's minimal and supported **everywhere** — good for low-level scripting.

🔹 2. `bash` – Bourne Again Shell

- Developed as part of the GNU Project.
- **Default shell** on most Linux distributions.
- 🛠️ **Features:**
  - Command history (`Up/Down` keys)
  - Tab auto-completion
  - Command-line editing
  - Scripting features: functions, loops, arrays
  - Built-in arithmetic and variables
- 📌 **When to use:**
  - **General scripting**
  - Everyday system administration
  - Most learning/tutorials assume `bash`

- ✅ **Why:**
  - Powerful, widely supported, and **compatible with** `sh`

🔹 3. `zsh` – Z Shell

- A **feature-rich shell** designed to extend `bash`.
- Popular in the developer community.
- 🛠️ **Features:**
  - Everything `bash` has +
  - **Powerful autocomplete**
  - **Spelling correction**
  - **Plugins and themes** (Oh My Zsh)
  - Enhanced globbing (`**`, `**/*.txt`)
  - Customizable prompt
  - Sharing command history between sessions
- 📌 **When to use:**
  - For **power users or developers** who want productivity features
- When customizing your terminal experience
- ✅ **Why:**
  - Clean, fast, and very user-friendly with tools like Oh My Zsh.

🔹 4. `fish` – Friendly Interactive Shell

- A **modern shell** focused on user-friendliness and usability.

🛠️ **Features:**

- **Syntax highlighting** in real time
- **Autosuggestions** as you type
- **Built-in help** system
- No need for `.bashrc` or `.zshrc` (uses functions and universal variables)
- Simpler scripting syntax (but **not POSIX-compatible**)
- 📌 **When to use:**
  - For **new users** or those who want a beautiful and easy terminal
  - For interactive use (not scripting)
- ✅ **Why:**
  - Ideal for beginners and productivity fans. It “just works.”

🔹 5. `csh` and `tcsh` – C Shell

- `csh` was created to **look like the C programming language**
- `tcsh` is an enhanced version of `csh`
- 🛠️ **Features:**
  - C-like syntax (`if (x) then`)
  - Command history and aliasing (in `tcsh`)
- Job control
- 📌 **When to use:**
  - If you work in an environment that already uses C shell scripts
  - Some BSD systems still use `tcsh` as default
- ✅ Why:
  - Legacy use — not recommended for modern scripting

📊 Summary Table

Shell	POSIX?	Interactive Features	Script-friendly?	Ideal Use Case
sh	✅	❌ None	✅ Very portable	Legacy, highly portable scripts
bash	✅	✅ History, autocomplete	✅ Robust	General-purpose scripting & terminal
zsh	✅	✅ Plugins, themes, autocorrect	✅ Powerful	Custom workflows, dev environments
fish	❌	✅ Highlighting, suggestions	❌ Simplified	Interactive use, beginners
tcsh	❌	✅ History, C-like syntax	❌ Limited	BSD users, legacy systems

🔧 Which Shell Should You Use?

You are...	Recommended Shell
A beginner	`fish` or `bash`
A system admin	`bash`
A developer	`zsh` + Oh My Zsh
Writing scripts for portability	`sh`
Working on BSD or legacy systems	`tcsh`

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

FILE SYSTEM

🔹 1. File System Hierarchy

The **Linux File System Hierarchy** describes **how files and folders are organized** on a Linux system.
Everything in Linux starts from the **root directory**: `/`
From there, the file system branches out like a **tree**.

🧭 Main Directories in `/` (Root):

Directory	Purpose
`/bin`	Essential system commands (like `ls`, `cp`, `mv`)
`/sbin`	System binaries (used by root/admins)
`/etc`	Configuration files for the system and apps
`/home`	Home directories for each user (`/home/alice`)
`/var`	Variable data like logs and caches
`/usr`	User programs, libraries (`/usr/bin`, `/usr/lib`)
`/lib`	Essential shared libraries needed at boot

Directory	Purpose
`/tmp`	Temporary files (cleared at reboot)
`/boot`	Files needed to boot the system (kernel, GRUB)
`/dev`	Device files (e.g., `/dev/sda`, `/dev/null`)
`/mnt`, `/media`	Mount points for external devices (USB, CD-ROM)
`/root`	Home directory for the `root` user (admin)
`/proc`, `/sys`	Virtual filesystems for system info

📌 **Everything is a file in Linux** — even devices and system interfaces.

🔹 2. Different File System Types

Linux can support **many types of file systems**, each with different purposes.

File System	Description
ext4	Most common Linux file system; stable and fast ✅
FAT32	Compatible with Windows/USB drives, but limited to 4GB file size
NTFS	Windows default file system; Linux can read/write with extra drivers
XFS	High-performance, good for large-scale storage
Btrfs	Modern, with built-in snapshots and checksums
tmpfs	Temporary filesystem stored in RAM (`/tmp`)

➡️ When you install Linux, the system usually uses **ext4** unless you choose otherwise.

3. Useful Commands and Interaction

Here are key **file system commands** and what they do:

Command	Purpose	Example
`ls`	List directory contents	`ls -l /etc`
`cd`	Change directory	`cd /home/user`
`pwd`	Show current directory path	`pwd`
`mkdir`	Create a new directory	`mkdir myfolder`
`rmdir` / `rm -r`	Remove a directory	`rm -r myfolder`
`touch`	Create an empty file	`touch file.txt`
`cp`	Copy files	`cp file.txt backup.txt`
`mv`	Move or rename files	`mv file.txt /tmp/`
`rm`	Delete a file	`rm file.txt`

Command	Purpose	Example
`find`, `locate`	Search for files	`find / -name "*.log"`
`df -h`	Show disk usage by partition	`df -h`
`du -sh`	Show space used by a file or folder	`du -sh Documents/`

You interact with the file system **via the shell or file explorer**.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

GETTING STARTED

🔹 1. Introduction to Shell and Environment

A **shell** is a program that lets you interact with the system by typing commands.
The **environment** is a collection of:

- Variables (`$PATH`, `$HOME`, etc.)
- Shell configuration (`.bashrc`, `.zshrc`)
- Aliases and functions

💡 **Shells** include: `bash`, `zsh`, `fish` (explained earlier)
You can open a shell by launching:

- **Terminal** (on Desktop)
- **TTY** (Ctrl + Alt + F2 on some systems)

2. Introduction to Graphical Interface

The **graphical interface** is also known as the **GUI** (Graphical User Interface).
A Linux GUI usually includes:

Component	Description
Display server	Manages the screen (X11 or Wayland)
Window manager	Controls window borders, layout
Desktop environment	Complete graphical system (GNOME, KDE, XFCE, etc.)
File manager	Explorer for browsing files (`Nautilus`, `Dolphin`, etc.)

You use it to:

- Open applications
- Use menus
- Drag and drop files
- Use visual tools (settings, printer setup, etc.)

🔹 3. Using Terminals and Applications

🖥️ Terminal

A terminal is a **text interface** to the system.
You type commands, the system responds.
Examples:

```
ls
mkdir project
sudo apt install firefox
```

🧩 Applications

You can launch apps:

- From a **menu** (like “Activities” or “Applications”)
- With a command (like `firefox &`)
- Or from a **desktop shortcut**

🔹 4. Navigating and Interacting with Files/Folders

📁 GUI Method:

- Double-click to open folders
- Drag/drop files
- Right-click for options

💻 Terminal Method:

```
cd ~/Documents
ls -lh
cp myfile.txt /tmp/
mv /tmp/myfile.txt ~/Desktop/
```

💡 Tips:

- `~` = your home directory
- `.` = current directory
- `..` = parent directory

Use `tab` to autocomplete names.

🔹 5. Managing User and Administrator Accounts

👤 Users

Each person using Linux has a **user account**.
Your files live in `/home/username/`.
To add a user (as root or with `sudo`):

```
sudo adduser alice
```

🔐 Administrator / Root

The **root user** is the superuser with full control.
To run something as root:

```
sudo command
```

Example:

```
sudo apt update
sudo rm -rf /important/folder
```

🔹 5. Managing User and Administrator Accounts

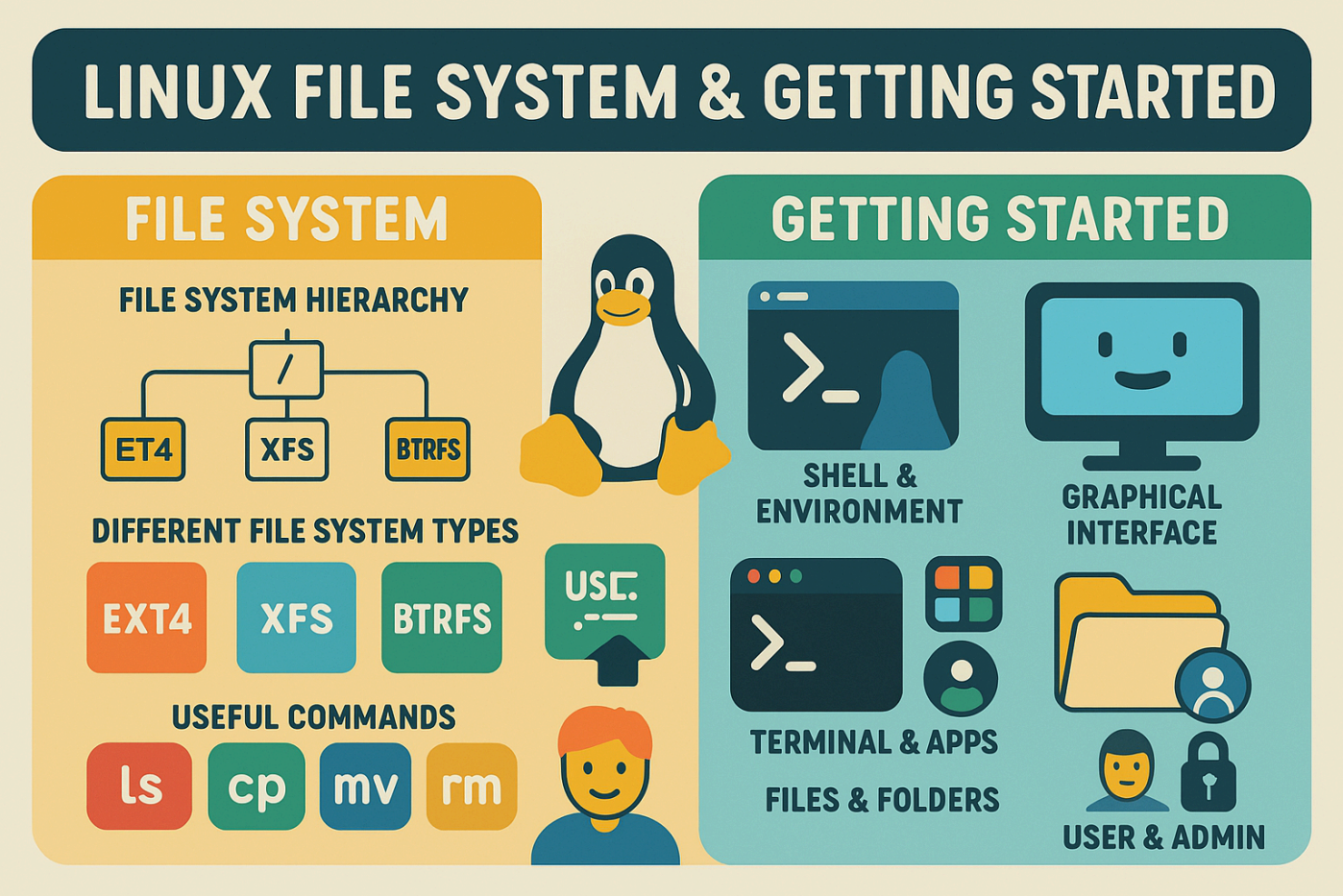
👥 Managing users:

```
sudo useradd john
sudo passwd john
sudo deluser john
```

You can also assign users to **groups**, like `sudo`:

```
sudo usermod -aG sudo john
```

✅ Summary



UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Exercises

🧱 Exercise 1 – Identify System Layers and Shell Type

Goal: Understand what shell, kernel, and distribution you're using.

1. Open a terminal and type the following commands:

```
echo $SHELL
uname -r
cat /etc/os-release
```

2. Questions:

- What is your current shell?
- What is the kernel version?
- What Linux distribution are you using?

3. Bonus:

```
ps -p $$
```

What is the name of the process listed? (Hint: It's your shell.)

👨‍💻 Exercise 2 – Switch Users and Use sudo

Goal: Understand and practice privilege escalation.

1. Try using a command that needs admin:

```
apt update
```

→ What happens?

2. Now use:

```
sudo apt update
```

→ Does it work?

👨‍💻 Exercise 3 – Switch Users and Use sudo

3. Switch to root (if password is set):

```
su -
```

4. Exit back to your user:

```
exit
```

5. Questions:

- What's the difference between `su` and `sudo`?
- When would you use each one?

👥 Exercise 4 – Create and Manage Users

Goal: Learn to add, delete, and assign users to groups.

1. Create a new user:

```
sudo adduser student
```

2. Set a password:

```
sudo passwd student
```

3. Add user to the `sudo` group:

```
sudo usermod -aG sudo student
```

👥 Exercise 5 – Create and Manage Users

4. Delete the user:

```
sudo deluser student
```

5. Questions:

- What folder is automatically created for the new user?
- How do you check what groups a user belongs to?

💾 Exercise 6 – Check File System Type and Mount Points

Goal: Explore partitions and file systems.

1. Show mounted partitions:

```
lsblk -f
df -h
mount | grep "^/"
```

2. Questions:

- What is the format type of your root (`/`) partition?
- Is `/home` on a separate partition?
- What type of file system is your USB key (if connected)?

3. Extra: View system disk usage:

```
sudo du -sh /*
```

🗂️ Exercise 7 – Navigate and Manipulate Files

Goal: Practice core file system commands.

1. Create a directory and file:

```
mkdir ~/mytest
touch ~/mytest/info.txt
```

2. Copy, rename, and delete:

```
cp ~/mytest/info.txt ~/Desktop/
mv ~/Desktop/info.txt ~/mytest/newname.txt
rm ~/mytest/newname.txt
```

3. Questions:

- What command shows your current folder?
- How do you go one level up in the file system?

🐚 Exercise 8 – Explore Shells

Goal: Identify and switch shells.

1. List available shells:

```
cat /etc/shells
```

2. Install `zsh` and switch:

```
sudo apt install zsh
chsh -s $(which zsh)
```

3. Logout/login again, then:

```
echo $SHELL
```

Exercise 9 – POSIX Compatibility (Bonus)

Goal: Learn how to write scripts that work across systems.

1. Create a POSIX-compatible script:

```
nano hello.sh
```

```
#!/bin/sh
echo "Hello, POSIX world!"
```

2. Make it executable:

```
chmod +x hello.sh
./hello.sh
```

3. Run it using different shells:

```
bash hello.sh
sh hello.sh
zsh hello.sh
```

📊 Exercise 10 – Understand Your Distribution

Goal: Use commands to learn system characteristics.

1. Find:

```
lsb_release -a
uname -a
```

2. Questions:

- What is your kernel version?
- Is your distribution Debian-based or RPM-based?

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

File Management

📁 1. Folder (Directory) Management

🛠️ Creating Folders – `mkdir`

```
mkdir myfolder
```

✅ Creates one directory in the current location.

```
mkdir folder1 folder2 folder3
```

✅ Creates **multiple folders** in one command.

```
mkdir -p parent/child/grandchild
```

✅ `-p` option = **create parent directories** if they don't exist.
Great for scripting or nested structures.

> 🔍 `mkdir` does **not overwrite** existing folders. If the folder exists, it throws an error unless `-p` is used.

📂 Viewing Folders – `ls` and `tree`

```
ls
```

✅ Lists files and directories in the current folder (non-hidden only).

```
ls -l
```

✅ Long format:

- Permissions (rwx)
- Number of links
- Owner / group
- File size
- Modification date

```
ls -a
```

✅ Lists **all files**, including hidden ones (`.bashrc`, `.gitignore`)

📂 Viewing Folders – `ls` and `tree`

```
ls -lh
```

✅ Human-readable sizes like KB, MB instead of bytes.

```
ls -R
```

✅ Recursive: shows subdirectory contents too.

```
ls -la /etc
```

✅ Combine options to list everything in `/etc`.

```
tree
```

✅ Shows folder structure as a **visual tree**.
May need install: `sudo apt install tree`

🔄 Navigating Folders – `cd`, `pwd`

```
cd /path/to/folder
cd ~                    # Home
cd ..                   # Up one level
cd -                    # Back to previous directory
pwd                     # Show current path
```

✏️ Rename and Move Folders – `mv`

```
mv oldfolder newfolder
```

✅ Renames folder.

```
mv folder /tmp/
```

✅ Moves folder to another location.
If destination exists, content will be **merged** inside.

📋 Copy Folders – `cp -r`

```
cp -r folder1 folder2
```

✅ `-r` = recursive (mandatory for directories)
Optional:

```
cp -rv folder1 folder2
```

- `-v` = verbose (show what's being copied)

❌ Delete Folders – `rmdir`, `rm -r`

```
rmdir folder
```

✅ Deletes **only if folder is empty**.

```
rm -r folder
```

✅ Recursively delete folder and contents.

```
rm -rf folder
```

- `-f` = force, no confirmation.
  ⚠️ Dangerous! Use with caution.

📄 2. Reading & Interacting with File Content

👀 Viewing Text Files

```
cat file.txt
```

✅ Prints whole file at once. Great for small files.

```
less file.txt
```

✅ Paged viewer, scrollable (`q` to quit, `/` to search).

```
head -n 20 file.txt
```

✅ First 20 lines (default = 10)

📄 2. Reading & Interacting with File Content

```
tail -n 15 file.txt
```

✅ Last 15 lines (default = 10)

```
tail -f logfile.log
```

✅ Real-time view of log files. Ctrl+C to stop.

🔍 Searching Content – `grep`

```
grep "error" /var/log/syslog
```

✅ Show lines containing “error”.

```
grep -i "ERROR" file.txt
```

✅ `-i` = case-insensitive

```
grep -r "conflict" /etc
```

✅ `-r` = recursive through folders

```
grep -n "hello" file.txt
```

✅ `-n` = show line numbers

```
grep -v "DEBUG" file.txt
```

✅ `-v` = show lines **not** containing “DEBUG”

🔢 Word & Line Counts – `wc`

```
wc file.txt
```

✅ Shows:

- Lines
- Words
- Characters

```
wc -l file.txt       # Only lines
wc -w file.txt       # Only words
wc -c file.txt       # Only characters
```

🧪 File Type – `file`

```
file file.txt
```

✅ Detects file format (text, binary, script, etc.)

📝 3. Managing Files

📄 Create Files

```
touch newfile.txt
```

✅ Creates empty file

```
echo "Hello World" > greeting.txt
```

✅ Writes one line to a file

```
nano file.txt          # Simple editor
vi file.txt            # Vim (advanced)
gedit file.txt         # GUI editor (GNOME)
```

🔁 Move / Rename Files – `mv`

```
mv file.txt /tmp/
mv file.txt renamed.txt
```

📎 Copy Files – `cp`

```
cp file.txt backup.txt
cp file.txt /home/user/backup/
```

With `-v` (verbose), `-u` (update only if newer), `-i` (ask before overwrite):

```
cp -uvi file.txt /home/user/backup/
```

❌ Delete Files – `rm`

```
rm file.txt
rm -i file.txt        # Confirm before delete
rm -f file.txt        # Force delete (no prompt)
```

🔐 Permissions – `chmod`

```
chmod 644 file.txt      # rw-r--r--
chmod 755 script.sh     # rwxr-xr-x
chmod +x file.sh        # Make executable
```

👤 Ownership – `chown`

```
sudo chown alice:alice file.txt
```

Change owner and group.

📊 Compare Files – `diff`, `cmp`

```
diff file1.txt file2.txt
cmp file1.bin file2.bin
```

🔗 4. Aliases and Links

📌 Aliases

🔹 Temporary:

```
alias ll='ls -lh'
alias clr='clear'
```

🔹 Permanent:

Edit `.bashrc` or `.zshrc`:

```
alias up='sudo apt update && sudo apt upgrade'
```

Reload:

```
source ~/.bashrc
```

🔗 Symbolic Links – `ln -s`

```
ln -s /real/path/file.txt shortcut.txt
```

- Behaves like a shortcut.
- Can link across partitions.
- If original is deleted → **broken link**

Check:

```
ls -l shortcut.txt
```

You'll see:

```
shortcut.txt -> /real/path/file.txt
```

🧱 Hard Links – `ln`

```
ln file.txt link.txt
```

- Points to **same inode**
- If original is deleted, file still exists
- Cannot link across partitions

Check with:

```
ls -li file.txt link.txt
```

Same inode = hard link.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Exercises

Exercise 1: Basic Folder Tasks

1. Create a folder named `linux_lab` in your home directory.
2. Inside it, create 3 folders: `notes`, `scripts`, and `logs`.
3. Navigate into the `scripts` folder.
4. Go back to your home directory using only `cd`.
5. Display the full path of your current directory.

Challenge: Create the full structure `linux_lab/backup/2025/april/notes` using one command.

Exercise 2: Viewing and Cleaning Folders

1. List all folders inside `linux_lab` in long format.
2. Display hidden files in your home directory.
3. Use `tree` to view the folder tree (install it if necessary).
4. Delete the `logs` folder.
5. Try to delete `linux_lab` (expect an error). Why?

Challenge: Use a single command to remove `linux_lab` and all of its contents without a prompt.

Exercise 3: Read and Scroll

1. Create a file `poem.txt` and paste a large text (copy/paste or use `curl https://www.gutenberg.org/files/11/11-0.txt > alice.txt`)
2. Display:
   - First 15 lines
   - Last 10 lines
3. Open the file with `less` and scroll to the bottom.
4. Use `tail -f` on a system log file (e.g., `/var/log/syslog`).

Challenge: Search the word “Alice” and count how many lines contain it.

Exercise 4: Search, Count, and Describe

1. Search for the word "rabbit" in `alice.txt`, case-insensitive.
2. Count the number of lines in the file.
3. Count the number of words in the file.
4. Find all lines that **do not** contain "Alice".
5. Use the `file` command on `alice.txt` and describe what it reports.

Exercise 5: Creating and Modifying

1. Use `touch` to create a file `mydata.txt`.
2. Write "Linux is awesome" into the file using `echo`.
3. Append "Always open source" to the same file.
4. Edit the file using `nano` and add a third line manually.
5. Display the full content with `cat`.

Challenge: Use a command to insert a timestamp on a new line each time the file is run.

Exercise 6: Copy, Move, Delete

1. Copy `mydata.txt` to the folder `backup`.
2. Rename `mydata.txt` to `info.txt`.
3. Move `info.txt` into your Desktop.
4. Delete it from your Desktop.
5. Use a command that shows total disk usage of all folders in your home directory.

Exercise 7: Permissions and Ownership

1. Create a file `script.sh` and make it executable.
2. Check permissions with `ls -l`.
3. Change permissions to `rw-r--r--`.
4. Change ownership of the file to another user (if one exists).
5. Set the file so that **only the owner** can read/write.

Exercise 8: Aliases

1. Create an alias `ll` that runs `ls -lh`.
2. Make an alias `goto` that brings you to your `Documents` folder.
3. Remove the alias `goto`.
4. Make your alias `ll` permanent by adding it to `~/.bashrc` or `~/.zshrc`.
5. Reload your shell.

Challenge: Create an alias called `greet` that echoes “Hello, $(whoami)!”.

Exercise 9: Links

1. Create a file `source.txt` with the content “This is the source file.”
2. Create a **symbolic link** to it named `symlink.txt`.
3. Create a **hard link** to it named `hardlink.txt`.
4. Delete `source.txt`.
5. Try reading both links. Which one still works? Why?

Challenge: Use `ls -li` to confirm inode numbers of the original and links. Explain the difference.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Networking

What is a Network?

A **network** is a group of computers/devices that communicate to share resources (files, printers, internet access, etc.).
In Linux, networking is mostly done using:

- **IP addresses**
- **Network interfaces** (like `eth0`, `wlan0`, `enp0s3`)
- **Protocols** like TCP/IP, SSH, FTP, etc.

Basic Concepts:

Term	What it means
IP Address	A unique number assigned to each device (like 192.168.0.2)
MAC Address	A unique hardware address of a network card
Subnet Mask	Defines how big the network is (like 255.255.255.0)
Gateway	The router or device used to access the internet
DNS	Resolves names like `google.com` to IP addresses

🧪 Demo: Viewing your network info

```
ip a
```

“ Shows your network interfaces and IP addresses. ”

Example result:

```
2: enp0s3: ...
    inet 192.168.1.25/24 brd 192.168.1.255 scope global dynamic enp0s3
```

Here:

- Interface: `enp0s3`
- IP: `192.168.1.25`
- Subnet: `/24` → equivalent to 255.255.255.0

🧪 Other commands:

```
hostname              # Show your device’s hostname
ip route              # Show routing table and default gateway
ping google.com       # Test if you're connected to the internet
```

🔸 How Linux Manages Network Interfaces

Linux uses:

- Configuration files (older systems use `/etc/network/interfaces`)
- `NetworkManager` service (on many desktop distros)
- `systemd-networkd` (on servers or minimal distros)

🔸 Check active interfaces:

```
ip link
```

> Shows all available network devices and their status.

🔸 Configure a static IP temporarily

```
sudo ip addr add 192.168.1.100/24 dev enp0s3
sudo ip route add default via 192.168.1.1
```

This:

- Sets a new IP
- Adds a default route to the internet

⚠️ These settings are **temporary** and will disappear on reboot.

🔸 Restart your connection:

```
sudo systemctl restart NetworkManager
```

Or use:

```
nmcli device reapply enp0s3
```

“ `nmcli` is the command-line tool for NetworkManager. ”

◆ Make it permanent (Ubuntu with Netplan):

File: `/etc/netplan/01-network-manager-all.yaml`

```
network:
  version: 2
  renderer: NetworkManager
  ethernets:
    enp0s3:
      dhcp4: no
      addresses: [192.168.1.100/24]
      gateway4: 192.168.1.1
      nameservers:
        addresses: [8.8.8.8, 1.1.1.1]
```

Then apply:

```
sudo netplan apply
```

🔄 File Transfers and Remote Connections

🔐 A. SSH – Secure Shell

What is it?
A secure way to **log in** to a remote Linux machine.

📘 Basic usage:

```
ssh user@remote-ip
```

Example:

```
ssh alice@192.168.1.50
```

First time? You'll be asked to **accept the key** (say yes).
You'll now be **inside** the remote computer's terminal!

🔄 File Transfers and Remote Connections

📂 B. SCP – Secure Copy

What is it?
SCP lets you **copy files between machines** over SSH.

📘 Upload a file:

```
scp file.txt user@192.168.1.50:/home/user/
```

📘 Download a file:

```
scp user@192.168.1.50:/home/user/file.txt .
```

🧠 Add `-r` to copy folders:

```
scp -r folder/ user@remote:/tmp/
```

🔄 C. rsync – Advanced Sync Tool

What is it?

`rsync` synchronizes files between two locations — locally or over SSH.

🟦 Sync files locally:

```
rsync -av --delete /src/ /dest/
```

- `-a` → archive (preserves permissions)
- `-v` → verbose
- `--delete` → remove deleted files

🟦 Sync with remote:

```
rsync -av file.txt user@192.168.1.50:/home/user/
```

🌐 D. FTP / SFTP

- `ftp` is old and **not secure**
- `sftp` uses SSH and is preferred

```
sftp user@remote-ip
```

Then:

```
put myfile.txt     # Upload
get otherfile.txt  # Download
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Exercises

Exercise 1 – View All Interfaces

“ Objective: Identify all the network interfaces on your system and observe their current state. ”

Task:

Use a command-line tool to list all available network interfaces, both physical and virtual.
Note the interface name, whether it’s up or down, and whether it has an IP address assigned.

Exercise 2 – Test Internet and Local Connectivity

“ Objective: Verify both internal and external network reachability. ”

Task:

Use tools to:

1. Test if your system can reach your local router.
2. Test if it can reach a public website using its domain name.

◆ Exercise 3 – Assign a Temporary Static IP

“ Objective: Temporarily assign a new static IP to your main interface. ”

Task:

Assign the IP `192.168.100.100/24` to your main Ethernet or wireless interface.

Ensure it doesn't interfere with an existing address and note that this change must be reversible.

◆ Exercise 4 – Enable SSH Access

“ Objective: Make a Linux system accessible remotely via SSH. ”

Task:

Ensure the OpenSSH server is installed, running, and reachable on port 22.

Verify from another system that you can connect using a terminal.

◆ Exercise 5 – Remote Command Execution

“ Objective: Run a command on a remote system without logging in interactively. ”

Task:

Run a command that lists files or shows system uptime on a remote machine, using SSH and without opening a session.

◆ Exercise 6 – Key-Based Authentication

“ Objective: Set up SSH login without a password using key pairs. ”

Task:

Generate an SSH key pair and configure the remote machine to allow key-based login from your system.
Ensure that password authentication is no longer needed.

◆ Exercise 7 – Transfer a File with `scp`

“ Objective: Send a local file to a remote machine. ”

Task:

Copy a file from your current machine to the home directory of a user on a remote machine over SSH.

◆ Exercise 8 – Retrieve a File with `scp`

“ Objective: Download a file from a remote system. ”

Task:

Locate a file on the remote system and download it to a local directory using `scp`.

Make sure the file retains its original content and permissions.

◆ Exercise 9 – Transfer a Directory Recursively

“ Objective: Upload a complete folder and its contents to a remote host. ”

Task:

Send a folder with multiple subfolders and files to the `/tmp/` directory on the remote machine.

◆ Exercise 10 – Synchronize with `rsync`

“ Objective: Use `rsync` to mirror local and remote files. ”

Task:

Synchronize a folder between your system and a remote system.
Make sure deleted files in the source are also deleted in the destination.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Scripting and Redirectors

📜 Shell Scripting — Full Explanation with Detailed Concepts

A **shell script** is a text file that contains a series of commands that are executed **in sequence** by the shell interpreter (such as `bash`, `sh`, `zsh`).
You're not just executing one command — you're **automating logic**.

> “ It's like turning a recipe (step-by-step) into a machine-readable format that the shell follows. ”

🎯 Why Use Scripts?

- To **automate repetitive tasks**
- To **define complex logic** (conditions, loops, functions)
- To handle **files**, **users**, **networks**
- To work **with systems and services**
- To connect and **reuse logic** in modular programs

How Shell Scripts Work

When you type `./myscript.sh`, the system reads it **line by line**, from top to bottom.
Before that, the script **must specify the interpreter**, like this:

```
#!/bin/bash
```

This is called a **shebang**. It tells the OS to use `/bin/bash` to run this file.

✅ 1. Use Variables

A **variable** is like a labeled box in memory that stores a value.
In Shell:

```
name="Alice"
echo "Hello $name"
```

- No space between variable name and `=`
- You retrieve values with `$variable`

Shell supports:

- **Strings** (text)
- **Integers** (basic arithmetic)
- **Arrays** (lists)
- **Associative arrays** (maps — only in Bash)

🧪 Example:

```
colors=("red" "green" "blue")
echo ${colors[1]}    # "green"
```

✅ 2. Use Logic and Conditions

Shell supports logic:

```
if [ $age -ge 18 ]; then
  echo "Adult"
else
  echo "Minor"
fi
```

Explanation:

- `if` starts the block
- `[ condition ]` is a command (called `test`)
- `-ge` means greater or equal
- `then` starts the action
- `fi` ends the condition

This works with:

- File checks (`-f file.txt`)
- String comparisons (`[ "$a" = "$b" ]`)
- Number comparisons (`-eq`, `-lt`, `-gt`, etc.)

✅ 3. Use Loops

A **loop** repeats an action until a condition changes.

`for` loop:

```
for file in *.txt
do
  echo "$file"
done
```

Reads all `.txt` files in the folder and prints each one.

`while` loop:

```
count=1
while [ $count -le 5 ]; do
  echo $count
  count=$((count+1))
done
```

Repeats while the condition is true.

✅ 4. Define and Use Functions

A **function** is a named block of code. You define it once, and use it multiple times.

```
greet() {
  echo "Hello, $1"
}
greet "Alice"
```

- `$1` is the first argument passed to the function
- Functions help keep your code **modular and reusable**

You can also **return values** using `echo` and capture them with `$(...)`.

✅ 5. Use Parameters

Shell scripts can accept arguments from the command line:

```
./myscript.sh Alice 25
```

In your script:

```
echo "Hello $1, age $2"
```

- `$1` = first parameter
- `$@` = all parameters
- `$#` = number of parameters

✅ 6. Include Files and Build Modular Scripts

Modular scripting means breaking large scripts into parts.

```
source settings.sh
```

This runs `settings.sh` inside your script, and you can use its variables and functions as if they were declared in your current script.

✅ 7. Use Exit Codes and Error Handling

When any command runs, it sets a **return code**:

- `0` → success
- `1-255` → error

You can check it with:

```
if ls myfile.txt; then
  echo "Found!"
else
  echo "Missing!"
fi
```

Or with:

```
echo $?
```

🧠 What Are Standard Streams?

When a command runs, it uses 3 invisible **channels**:

Stream	Number	Description
stdin	0	Input (usually from keyboard or pipe)
stdout	1	Output (normal result messages)
stderr	2	Error output (problem messages)

🎯 What Happens When You Type a Command?

Let's say:

```
ls -l
```

Here's what's really happening:

- You're typing into **stdin**
- The result goes to **stdout**
- If there's a problem (e.g., bad path), it goes to **stderr**

🔵 Redirecting Output

Goal	Syntax	Meaning
Send stdout to a file	`command > file.txt`	Saves only normal output
Append to file	`command >> file.txt`	Adds at the end
Send stderr to file	`command 2> errors.log`	Saves only errors
Combine stdout and stderr	`command > all.log 2>&1`	Redirect both to one file
Use shorthand	`command &> file.log`	Equivalent to `2>&1`

🧪 Real Example

```
ls goodfile.txt badfile.txt > output.txt 2> error.txt
```

- If `goodfile.txt` exists → goes to `output.txt`
- If `badfile.txt` doesn't → error goes to `error.txt`

🟦 Redirecting Input

You can also use `stdin` to send data into commands.

```
cat < file.txt
```

This feeds `file.txt` into `cat`, as if you typed it.
Also useful with loops:

```
while read line
do
  echo "Line: $line"
done < myfile.txt
```

Here, `myfile.txt` is read line-by-line into `read`.

🧪 Chaining with Pipes

A **pipe** sends the `stdout` of one command into the `stdin` of another:

```
ps aux | grep firefox | wc -l
```

- `ps aux` → list processes
- `grep firefox` → filter
- `wc -l` → count matching lines

Each command **passes its output to the next one**.

🧠 Conceptual Diagram

```
Keyboard → [stdin] → command → [stdout] → Terminal or file
                           ↘→ [stderr] → Terminal or log file
```

When you use `|`, you do this:

```
[stdout of cmd1] → [stdin of cmd2]
```

When you do this:

```
command > out.txt 2> err.txt
```

You're **splitting** stdout and stderr into two different destinations.

🧪 Practical Use Cases

1. **Log success and failure separately**:

```
./build.sh > success.log 2> failure.log
```

2. **Read names from a file**:

```
./greet.sh < names.txt
```

Script:

```
while read name; do
  echo "Hello, $name"
done
```

🧪 Practical Use Cases

3. **Ignore errors but show output**:

```
command 2>/dev/null
```

4. **Silence everything**:

```
command > /dev/null 2>&1
```

This means: "Don't show anything on screen."

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Labs

🧰 Lab 1

🎯 Scenario:

You've just joined a company as a **junior system administrator**. Before being given access to production servers, your manager asks you to **prepare your personal Linux workstation** (WSL, dual boot, or native) as if it were a small server environment. Everything you do will be checked by scripts, files, logs, and command output.

🧩 Tasks:

🔹 1. System Exploration

- Identify your current distribution, kernel version, and system type.
- List your partitions, mount points, and file system types using:
  - `lsblk`, `mount`, `df -h`, `findmnt`, `blkid`
- Check where user data and system configs are stored (`/home`, `/etc`, etc.)

2. User and Permissions Setup

- Create a new user named `adminlocal`.
- Set a password for that user.
- Add the user to the `sudo` group (or configure `sudoers.d`).
- Create an alias called `becomeadmin` that allows switching to `adminlocal` easily.

3. Useful Automation Scripts

- Create a folder `~/bin/` with:
  - A script to archive any directory passed as an argument.
  - A system summary script showing:
    - Current user, date, uptime, CPU load.
- Add `~/bin/` to your `$PATH` so you can call these scripts from anywhere.

4. File Management and Productivity

- Create a symbolic link in your home directory pointing to `~/Documents`, called `docs-link`.
- Create an alias `cleanup` that deletes all `.tmp` and `.bak` files in the current folder (but not in `~/Downloads`).
- In a file `.bash_aliases`, define at least 5 aliases that help with:
  - Navigation (`cd` shortcuts)
  - File inspection (`ls`, `du`, `find`)
  - Network info (`ip`, `ping`, `hostname`)

5. Network Configuration & Check

- Identify your current IP address, network interface, gateway, and DNS.
- Create an alias `netinfo` that displays this in a formatted way.
- Create a command `pingdns` that pings 1.1.1.1 and 8.8.8.8 and logs the output to `~/ping.log`.

🌐 Lab 2

🎯 Scenario:

You work as a freelancer. You often switch between projects. You want to build your own **toolkit to automate daily tasks**, **monitor your files**, and **synchronize logs or backups**, all from your personal machine.

🧩 Tasks:

🔹 1. Startup Automation

- Write a script `startup.sh` that:
  - Displays a welcome banner
  - Checks internet connectivity
  - Lists the 3 most recently changed files in `~/Documents`
  - Logs all output (and errors) to `~/startup.log`
  - Uses: pipes, functions, stderr, `tee`, and logic conditions

🔹 2. Smart Backup Script

- Write a script `save.sh` that:
  - Archives `~/Documents` as `.tar.gz`
  - Saves it to `~/Backups/` with a filename containing the date
  - Displays the archive size
  - Deletes backup files older than 7 days automatically

🔹 3. Directory Structure & Links

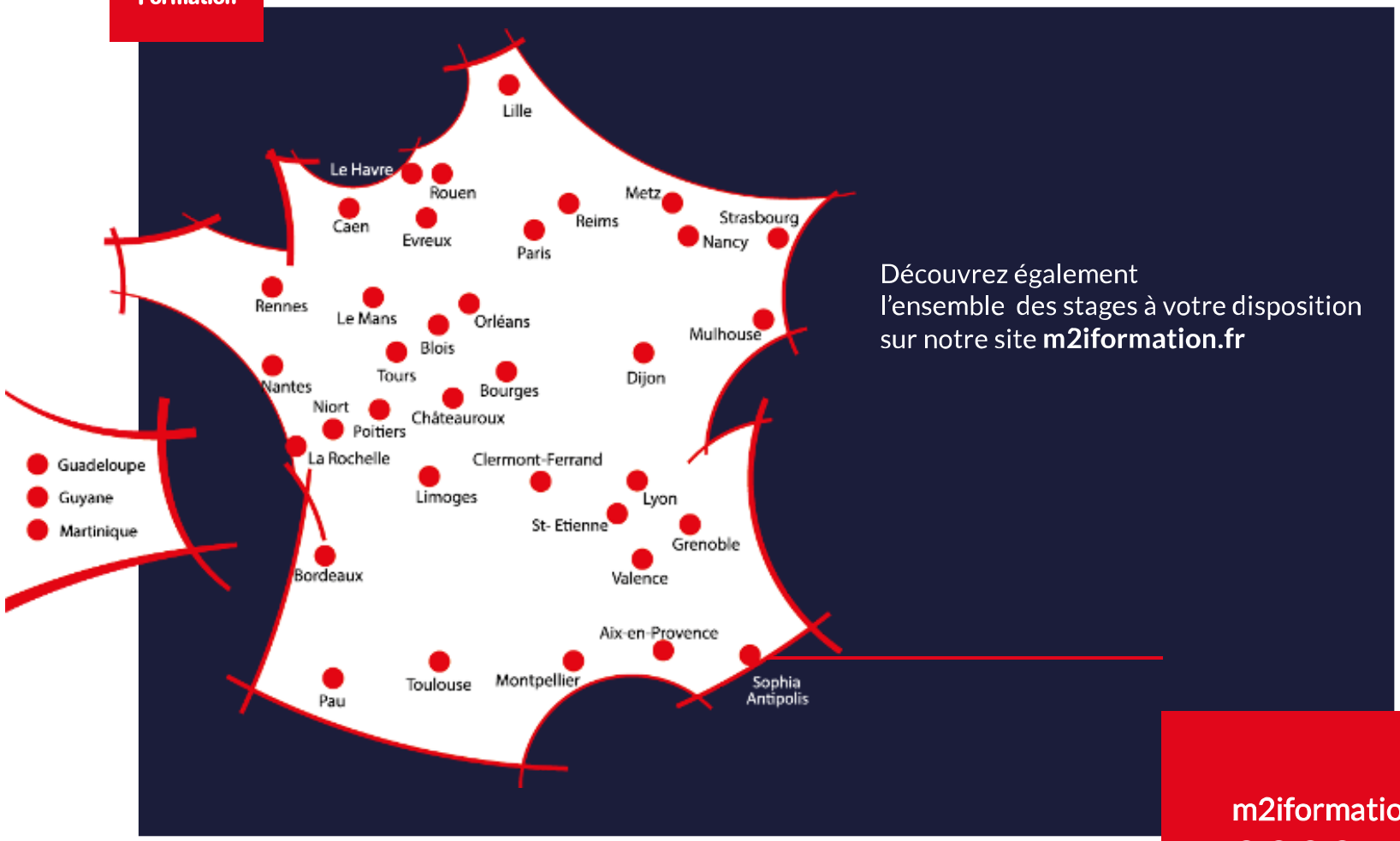
- In your home folder:
  - Create a `~/workspace/` with 3 subfolders: `web`, `admin`, `notes`
  - In `notes/`, create:
    - A hard link between two `.txt` files
    - A symbolic link pointing to your `Downloads`

🔹 4. Shell Enhancements

- In `.bashrc` or `.bash_aliases`, define a function `watchfolder`:
  - It takes a directory name as input
  - Every 5 seconds, it checks if new files appeared and prints them
- Create another command `searchtext` that:
  - Searches for a string passed as an argument in all `.txt` files in the current directory
  - Writes the result to `found.txt`, sorted alphabetically by filename

Optional Add-ons (for both Labs):

- Add a summary script (`status.sh`) that gives you a dashboard:
  - Current network info
  - Disk usage
  - Running processes
- Log each important action (success or failure) using timestamps and `$?`



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr