Sure! Here's a **simple explanation** of what Andrew Mallett is talking about in this introduction to **Managing systemd Services and Targets**.

**👋 What’s This Module About?**

This module teaches you how to **manage services and system settings** on modern Linux systems using something called **systemd**.

Systemd is the tool that controls **how your Linux system starts, stops, and runs background programs (called services)** — like your web server, database, or networking.

**💡 What is systemd?**

* It's a **standard system manager** used by most modern Linux systems (like Ubuntu, Red Hat, Fedora, etc.).
* It replaces older systems like init.
* The best part: It works **the same way** across most Linux distributions, so you learn it once and use it anywhere!

**📦 Tools You'll Use**

All systemd-related commands **end with ctl**, which stands for “control” — that’s how you can recognize them.

**🔧 1. systemctl – The Main Tool**

* Use it to **start, stop, enable, disable, and check the status of services**.
* Example:
* sudo systemctl start apache2

**📛 2. hostnamectl – Set the System’s Hostname**

* Changes the name of your computer.
* Works for both:
  + **Static hostname** (saved in a config file)
  + **Transient hostname** (only in memory, resets on reboot)

**⏰ 3. timedatectl – Manage Time and Time Zone**

* Set the system’s **time zone**, **time**, and **enable NTP** (network time syncing).
* Example:
* timedatectl set-timezone Asia/Kolkata

**🌐 4. localectl – Set Language and Keyboard Settings**

* Set your system’s **language (locale)** and **keyboard layout**.

**🛠️ Services and Targets**

**📌 What are Services?**

* Services are background programs (like a web server or database).
* You can control them with systemctl.

**🎯 What are Targets?**

* Targets are **groups of services**.
* They help define **runlevels** (like what should start when the system boots).
* Example: A "graphical" target starts the desktop, while "multi-user" is for servers.

**📜 Checking Logs with journalctl**

* This command lets you read logs from the systemd journal.
* You can filter logs by time, service, or other options.
* Example:
* journalctl -u apache2

**✅ Summary (Easy Mode)**

| **What You Can Do** | **Command Used** |
| --- | --- |
| Manage services (start/stop/etc.) | systemctl |
| Set your system's name | hostnamectl |
| Set time zone and time | timedatectl |
| Set language/keyboard | localectl |
| View system logs | journalctl |

You're now ready to learn how to **control services, system settings, and logs** the smart way — using systemd, which works on most Linux systems.

Let me know if you want a cheat sheet or practice questions for this!

Absolutely! Here’s a **simple explanation** of what Andrew is doing in this demo using **systemd tools** like hostnamectl, timedatectl, and localectl.

**🖥️ What’s Happening Here?**

You're learning how to **manage key system settings** (like the hostname, time zone, and language) in Linux using commands that work the same on almost all modern Linux systems (Ubuntu, Red Hat, Fedora, etc.).

**🔧 1. hostnamectl – Set or View Your Computer’s Name**

* This command shows:
  + The system’s name (hostname)
  + The operating system
  + Virtualization info (e.g., if it’s running in VirtualBox)

✅ **Example usage:**

hostnamectl

Shows your current system name and OS info.

sudo hostnamectl set-hostname main

Sets the system's name to main. (You need sudo to change it.)

📝 Even though your prompt may still say the old name, it’s already changed in the background. You can check it with hostnamectl again.

**🕒 2. timedatectl – Set Time, Time Zone, and Sync Settings**

This command shows:

* Current date and time
* Time zone
* If your clock syncs automatically from the internet (NTP)

✅ **Example usage:**

timedatectl

Shows the current settings.

sudo timedatectl set-timezone Europe/London

Changes your time zone to London.

💡 You can press Tab to auto-complete region and city names!

**🌍 3. localectl – Set Language and Keyboard Layout**

This sets what **language and character set** your system uses.

✅ **Example usage:**

localectl

Shows the current locale settings.

sudo localectl set-locale LANG=en\_GB.utf8

Sets your system to use **British English UTF-8**.

🇺🇸 Want US English? Use LANG=en\_US.utf8 instead.

**🎯 Why Are These Commands Great?**

* ✅ They **work the same** across many Linux distributions (like Ubuntu, Red Hat, Arch).
* ✅ You **don’t need to remember where each distro stores these settings** — systemd handles that for you.
* ✅ It’s **simple and consistent**, even for beginners.

**📝 Summary Table**

| **Task** | **Command example** |
| --- | --- |
| View hostname & OS info | hostnamectl |
| Set hostname | sudo hostnamectl set-hostname myname |
| View time/date settings | timedatectl |
| Set time zone | sudo timedatectl set-timezone Europe/London |
| View language/locale | localectl |
| Set language/locale | sudo localectl set-locale LANG=en\_GB.utf8 |

Let me know if you'd like practice tasks or a cheat sheet for these commands!

Absolutely! Here's a **simple explanation** of what was covered about **Systemd and analyzing the boot process** on Linux systems like Ubuntu.

**🚀 What Is Systemd?**

When your Linux system starts, it has to **launch all the core programs and services** (like networking, login, etc.). The first program that gets launched after the Linux kernel is called **systemd**.

* It becomes **process ID 1** (PID 1) – the first running program on your system.
* Its job is to **start and manage everything else**.

This makes systemd the **heart of the system** after boot.

**🧱 How Boot Works (In Simple Steps):**

1. **Bootloader** runs first (like GRUB) – lets you choose what OS or kernel to boot.
2. **Kernel** loads and starts systemd (PID 1).
3. **Systemd** starts all services (networking, logging, display, etc.).

**🧾 How to Check All This**

**✅ 1. See who is PID 1 (should be systemd):**

ps -p 1 -f

It shows that the process with ID 1 is owned by root and is actually systemd (even if it says /sbin/init, it's just a **shortcut/symlink** to systemd).

You can confirm this with:

ls -l /sbin/init

You'll see it's pointing to /lib/systemd/systemd.

**✅ 2. Show the Boot Process Tree:**

pstree

* Shows you the "family tree" of processes.
* At the very top: systemd
* Below it: everything else systemd started (your desktop, services, etc.)

**✅ 3. Analyze Boot Time:**

systemd-analyze

This tells you:

* 🧠 How long the **kernel** took to load.
* 🛠️ How long **user space services** took to start.
* 🖥️ When the system was ready for you to log in.

Example output:

Startup finished in 2.867s (kernel) + 9.948s (userspace) = 12.815s

So total boot time = around 13 seconds.

**✅ 4. See Which Services Took Longest:**

systemd-analyze blame

* Shows a list of services from **slowest to fastest**.
* Helps you figure out **what’s slowing down your boot**.

🔍 Tip: Just because a service took 2 seconds doesn’t mean removing it saves 2 seconds — many services start **in parallel** (at the same time).

**📌 Why This Is Useful:**

* Helps you understand **what happens when Linux boots**.
* Lets you **identify and optimize** services that slow down your system.
* systemd gives us **tools that work the same way** across many Linux distros (Ubuntu, Red Hat, Arch, etc.).

**📝 Summary**

| **What You Want to Do** | **Command** |
| --- | --- |
| See PID 1 (should be systemd) | ps -p 1 -f |
| View the process tree | pstree |
| See total boot time | systemd-analyze |
| Find slowest-starting services | systemd-analyze blame |
| Confirm init is actually systemd | ls -l /sbin/init |

Let me know if you want to **practice this on your system**, or need help with **tuning slow services**!

Absolutely! Let’s break all that down into **simple, clear explanations** with examples so you can understand exactly what’s going on — and feel confident using systemctl.

**💡 What is systemctl?**

systemctl is the **main command** used to **control services** in Linux systems that use **Systemd** (most modern Linux distros like Ubuntu, Red Hat, etc.).

You use systemctl to:

✅ Start services  
✅ Stop services  
✅ Check status  
✅ Enable/disable services at boot  
✅ View logs  
✅ Edit service settings

**🔍 Basic Commands & What They Do**

**1. ✅ Check if a service is running:**

systemctl status cron

* Shows if the **cron** service is running
* Also shows:
  + If it will **auto-start** on boot
  + The **process ID**
  + Recent **log entries**

**Example output:**

● cron.service - Regular background program processing daemon

Loaded: loaded (/lib/systemd/system/cron.service; enabled)

Active: active (running)

...

**2. 🔄 Start or Stop a service manually**

**Start cron:**

sudo systemctl start cron

**Stop cron:**

sudo systemctl stop cron

**3. ⚙️ Enable a service to start at boot**

**Enable cron to auto-start:**

sudo systemctl enable cron

This means next time the system boots, cron will start **automatically**.

**4. ✅ Enable and start right now**

**Do both at once (recommended):**

sudo systemctl enable --now cron

This command:

* **Starts** cron **now**
* **Enables** it for the next boot

**5. ⛔ Disable and stop a service**

**Stop cron now and disable auto-start:**

sudo systemctl disable --now cron

This is handy if you want to **completely turn off** a service.

**6. 📄 See the unit file (service settings)**

systemctl cat cron

This shows the contents of the **service unit file**, which defines how the service behaves.

**7. 📝 Edit a service file**

sudo systemctl edit cron

This opens an editor (like nano) where you can **change** or **override** settings.

💡 It creates a **custom override**, so the original file is untouched.

**🧠 Real Example: Practice**

Let’s say you're on Ubuntu, and you want to:

1. Make sure the **cron** service is running
2. Set it to auto-start every time
3. Check its logs

Here’s what you’d do:

# Check the status

systemctl status cron

# Enable and start it

sudo systemctl enable --now cron

# Check logs for cron

journalctl -u cron

**🔥 Tip: Commands Ending in ctl**

Systemd tools follow a pattern — they all end in ctl, which means “control”.

| **Command** | **What It Controls** |
| --- | --- |
| systemctl | Services and system units |
| hostnamectl | Hostname |
| timedatectl | Time, date, time zone |
| localectl | Language/locale |
| loginctl | User sessions (advanced) |
| journalctl | Logs for services |

**🏁 Summary**

* systemctl is your main tool to manage services.
* Use status, start, stop, enable, disable, and --now.
* Use cat to view a service config, and edit to override settings.
* Works on **almost all modern Linux systems**.

Would you like a short cheat sheet PDF or a quick hands-on practice guide? Let me know!

Sure! Let me explain everything in **simple terms**. The person is teaching you how to **manage services** on a Linux system (like Ubuntu) using the **systemctl command**. Here's a breakdown of what he's saying:

**🌟 What is systemctl?**

systemctl is a command used to **control system services** on Linux. These services are background programs that run automatically (like cron jobs, network services, etc.).

**🧑‍💻 Basic Commands with systemctl**

**✅ 1. Check status**

systemctl status cron

* This checks the **status of the cron service** (a job scheduler).
* If you don’t write .service, it assumes you're talking about a service.

**🧍 2. Standard User vs Root User**

* If you're a **normal user**, you might **not see all logs** or info.
* So he uses:

sudo -i

* This opens a **root shell** where you don’t have to keep typing sudo.

**🔁 Enabling and Disabling Services**

**▶️ 3. Enable a service**

systemctl enable cron

* This makes sure the service **starts automatically on boot**.

**⏹️ 4. Disable a service**

systemctl disable cron

* This stops the service from **starting on boot**, but **doesn't stop it now**.

**🛑 Stopping and Starting Services**

**⏸️ 5. Stop a service**

systemctl stop cron

* This **immediately stops** the service.

**🔁 6. Enable AND start**

systemctl enable --now cron

* This enables it for future boots and starts it **right now**.

**❌ 7. Disable AND stop**

systemctl disable --now cron

* Disables for future boots and stops it now.

**🔒 Masking and Unmasking Services**

**🚫 8. Mask a service**

systemctl mask --now cron

* Prevents the service from being started **in any way**.
* It's like locking it down completely.

**🔓 9. Unmask a service**

systemctl unmask cron

* Allows the service to be started again.

**📝 Viewing and Editing Service Files**

**👀 10. View service file**

systemctl cat cron

* Shows the configuration of the service.

**✏️ 11. Edit service file**

systemctl edit cron

* Lets you change the config in a safe way.
* It creates a copy (override) so you don’t mess with the original.

**🔄 12. Reload systemd after editing**

systemctl daemon-reload

* Needed to apply the changes you made.

**Summary: What You’re Learning**

You're learning how to:

* Check if services are running.
* Start, stop, enable, or disable them.
* Prevent them from running (mask).
* View and customize service configurations.

If you’re preparing for an exam or just want to get better at Linux, this is **super useful stuff**. Let me know if you want me to help you **practice** with examples or make a **cheat sheet**!

Absolutely! 😊 Let's break it down into **super simple words** so you can understand everything clearly.

**🧾 What This Whole Lesson Is About:**

You're learning how **Linux stores and manages log messages**, and how to **read them** using both old and new methods.

**🧱 Two Types of Logging Systems in Linux:**

**1. Traditional (Old Style) Logging:**

* Uses **log files** in the folder:  
  /var/log
* Uses a program called **rsyslog** to write logs.
* Example log file: /var/log/syslog
* You read these using commands like:
* sudo less /var/log/syslog
* sudo tail -f /var/log/syslog

**2. Modern Logging (Systemd Journal):**

* Uses a tool called **journalctl**
* Stores logs in a special **binary format** (not regular text files).
* Easier to **filter** logs (by service, time, etc.)
* You read logs using:
* sudo journalctl

**🛠️ Using journalctl - The New Way**

**View all logs:**

sudo journalctl

**View logs for a specific service, like SSH:**

sudo journalctl --unit=ssh

**View logs from today only:**

sudo journalctl --since today

You can even use:

--since "2024-07-01" --until "2024-07-19"

To filter by date range.

**📁 Persistent Logs vs Volatile Logs**

By **default**, journal logs:

* **Do not** stay after a reboot (they're saved in memory only).
* So when you restart the system, you lose the logs.

To **keep logs after reboot**, you must change a setting.

**🔧 How to Make Journal Logs Persistent**

1. Open the config file:

sudo nano /etc/systemd/journald.conf

1. Find the line:

#Storage=auto

1. Change it to:

Storage=persistent

1. Save and close the file.
2. Restart the journal service:

sudo systemctl restart systemd-journald

Now your logs will be saved even **after reboot**! 🎉

**💡 Faster Way: Use sed to do this automatically**

Instead of editing manually, you can run this command:

sudo sed -i 's|#Storage=auto|Storage=persistent|' /etc/systemd/journald.conf

This replaces the line in one shot.

**✅ Summary Table**

| **Task** | **Command** |
| --- | --- |
| View all logs | sudo journalctl |
| View logs for SSH | sudo journalctl --unit=ssh |
| View today’s logs | sudo journalctl --since today |
| Make logs persistent | Edit /etc/systemd/journald.conf and set Storage=persistent |
| Restart journald | sudo systemctl restart systemd-journald |

**🎓 Why This Is Useful**

* You can **diagnose errors** and problems by reading logs.
* journalctl is easier and smarter than reading big old text files.
* You can filter by **date**, **service**, and more.
* Making logs persistent helps keep logs **even after reboot**.

Would you like me to show you how to run some of these on your system? Or create a cheat sheet for you? 😊

Absolutely! Let me explain this in **simple words** — like you're just getting started with **Bash scripting** on Linux.

**👋 What is this all about?**

This module is about **automating tasks in Linux using shell scripts**. Shell scripts are just **a list of commands** saved in a file that can be run like a mini-program.

**🧾 What's a Shell Script?**

A **shell script** is:

* A **text file**
* That contains **Linux commands** (like echo, hostname, etc.)
* You run it to perform a **series of steps automatically**

Example:

echo "This is my script"

hostname

uname -r

**📁 Getting Started**

**1. Create a Directory (just to keep your scripts organized):**

mkdir my-scripts

cd my-scripts

**2. Create a Script File**

You can use a text editor like **Nano**:

nano info.sh

Paste this inside:

echo "======================="

echo "System Info"

echo "======================="

hostname

hostname -I

uname -r

echo "======================="

Save and exit.

**🐚 Running Your Script**

**3. Run it with Bash**

bash info.sh

This runs the script using the Bash shell.

**🔥 Make It a Real Program**

To make your script **runnable like a real command**, you do 2 things:

**✅ 1. Add a special line at the top (called a shebang):**

This tells Linux **which shell** to use.  
You can add it manually, or quickly with this command:

sed -i '1i #!/bin/bash' info.sh

This adds the line #!/bin/bash to the top of your file.

**✅ 2. Make it executable:**

chmod +x info.sh

Now you can run it like this:

./info.sh

If the file is in a directory that’s part of your **PATH**, you could even run it by name alone:

info.sh

But usually, if it’s in your current directory, you need to use ./ before the name.

**🛠️ Why use sed?**

Instead of opening each file and manually adding #!/bin/bash, the sed command lets you **automate** this step.

For example:

sed -i '1i #!/bin/bash' \*.sh

Adds the shebang line to the **first line** of **every .sh file** in your folder. Super useful!

**🏁 Summary (TL;DR):**

* A **shell script** is just a file with a bunch of Linux commands.
* You run it to **automate repetitive tasks**.
* You can add #!/bin/bash at the top to make it run properly.
* Make it executable with chmod +x.
* Run it with ./scriptname.sh.

Would you like a simple step-by-step script challenge next? Like writing one that shows CPU info or disk space? 😊

Of course! Here's a **simple explanation** of everything that was done in that demo about **creating and running shell scripts in Linux**.

**🧾 What is this all about?**

You are learning how to:

* Create and organize shell scripts.
* Make them **runnable** like mini-programs.
* Use sed to edit files quickly.
* Set up your environment (like PATH) to run scripts more easily.

**🛠️ Step-by-Step (In Simple Terms)**

**1. ✅ Create a folder to store scripts**

This is just to stay organized.

mkdir shellscripting

cd shellscripting

You're now inside a folder named shellscripting where you will keep your scripts.

**2. ✏️ Create a simple script**

Open a text editor like **nano** and make a new script:

nano info.sh

Type this inside:

echo "====================="

echo "System Info"

echo "====================="

hostname

hostname -I

uname -r

echo "====================="

Save it:

* Press Ctrl+X to exit
* Press Y to save
* Press Enter to confirm the filename

This is your **first script**! 🎉

**3. ▶️ Run the script (with bash)**

Even if it’s not marked as “executable,” you can still run it with Bash:

bash info.sh

You'll see output like:

=====================

System Info

=====================

ubuntu1

192.168.1.100

5.15.0-xx-generic

=====================

**4. 🚀 Add a shebang line**

The **shebang line** tells Linux **what program** should run this script.  
You add this at the top of your script:

#!/bin/bash

Instead of opening each file, you can do it quickly with sed:

sed -i '1i #!/bin/bash' info.sh

If you had multiple scripts like info.sh and info2.sh, you can do:

sed -i '1i #!/bin/bash' info\*

This adds the shebang to the **first line** of all scripts starting with info.

**5. 🔐 Make the script executable**

Give permission to run the script like a program:

chmod +x info.sh

To do this for multiple files:

chmod -v +x info\*

**6. 💻 Run it directly**

Now you can run the script like this:

./info.sh

If you’re inside the shellscripting folder, ./ means “run from the current folder.”

**7. 🔍 Optional: Add folder to PATH**

Right now, you have to type ./info.sh. If you want to just type info.sh from anywhere, you can add the folder to your PATH like this:

export PATH=$PATH:/home/vagrant/shellscripting

Now you can run:

info.sh

from anywhere in the terminal **(for this terminal session only)**.

⚠️ To make this permanent, you'd need to add that export line to your ~/.bashrc or ~/.profile.

**🏁 Summary (TL;DR)**

* You made a folder to store your shell scripts.
* You wrote a script with commands like hostname, uname -r, etc.
* You made it executable with chmod.
* You added a shebang line so the system knows to use Bash.
* You optionally added the folder to your PATH to run the script from anywhere.

Would you like help writing your own custom script next? Like a script to check disk space, memory, or running services?

Absolutely! Here's a **simple explanation** of what just happened — you improved your basic shell script to make it look better, be more readable, and act more like a **"rock star" script**. 🎸✨

**🚀 What did we do in this step?**

We made our script better by:

1. Using **variables** for hostname, IP address, and kernel version.
2. Using a **loop** to create a header/footer line with equal signs (=).
3. Printing everything in a nice, clear format.

Let’s break it down piece by piece:

**🧠 1. Using Variables (To Make Code Clearer and Reusable)**

Instead of repeating the same command multiple times, we store the output in **named variables**:

INFO\_HOSTNAME=$(hostname)

INFO\_IP=$(hostname -I | cut -d ' ' -f2)

INFO\_KERNEL=$(uname -r)

✅ What’s happening:

* INFO\_HOSTNAME: Stores the computer name.
* INFO\_IP: Gets **only the 2nd IP address** using cut, because hostname -I gives multiple IPs.
* INFO\_KERNEL: Gets the Linux kernel version.

📝 Why use variables?

* Makes your script easier to **read and change** later.
* Helps you avoid running the same command multiple times.

**🔁 2. Creating a Header/Footer Line with a Loop**

Instead of manually typing 25 equal signs, we use a **for loop** to create them automatically:

for i in {1..25}

do

echo -n "="

done

✅ What’s happening:

* {1..25} → Means “do this 25 times.”
* echo -n "=" → Prints = without a new line, so they stay on the same line.

This prints:

=========================

We use this as a **header and footer**.

**💬 3. Displaying the System Info Nicely**

Now, instead of just printing raw data, we format it with labels:

echo ""

echo "Host: $INFO\_HOSTNAME"

echo "IP: $INFO\_IP"

echo "Kernel: $INFO\_KERNEL"

echo ""

✅ What’s happening:

* The echo "" adds a blank line to space things out.
* echo "Host: $INFO\_HOSTNAME" prints the hostname clearly with a label.
* We do the same for IP and Kernel version.

**📝 Final Script (How it might look)**

#!/bin/bash

INFO\_HOSTNAME=$(hostname)

INFO\_IP=$(hostname -I | cut -d ' ' -f2)

INFO\_KERNEL=$(uname -r)

for i in {1..25}

do

echo -n "="

done

echo ""

echo "Host: $INFO\_HOSTNAME"

echo "IP: $INFO\_IP"

echo "Kernel: $INFO\_KERNEL"

echo ""

for i in {1..25}

do

echo -n "="

done

echo ""

**▶️ How to Run**

If it's already executable, just run:

./info.sh

**✅ What You Get in Output**

=========================

Host: ubuntu1

IP: 192.168.56.101

Kernel: 5.15.0-91-generic

=========================

🎉 It looks professional, it’s reusable, and it’s easier to edit later.

**🧠 Why This Is "Rock Star" Level**

* ✅ Clean, readable code.
* ✅ Uses best practices like variables and loops.
* ✅ Produces neat output.
* ✅ Can be easily reused or modified.

Would you like me to generate a script file like this for you to try directly? Or help you customize it even further (e.g., show uptime, user count, disk space, etc.)?

Absolutely! Let's simplify this step too. You're now learning to **import system info** into your shell script using a powerful built-in command: source.

**🧾 What’s the Goal Here?**

You're adding **extra system info** like the OS name and version (e.g., Ubuntu 20.04.4 LTS) into your script — without typing it yourself.

You're doing this by reading from a special system file:  
📄 **/etc/os-release**

**✅ What's Inside /etc/os-release?**

This file contains **predefined variables** about the operating system, like this:

NAME="Ubuntu"

VERSION="20.04.4 LTS"

PRETTY\_NAME="Ubuntu 20.04.4 LTS"

ID=ubuntu

So, instead of manually typing "Ubuntu" in your script, you can grab it directly from here using:

source /etc/os-release

**🧠 What Does source Do?**

source is a **built-in shell command** that:

* Reads another file (like a script).
* **Loads its variables** into the current script or shell.

So after this:

source /etc/os-release

You can now use:

echo $PRETTY\_NAME

✅ And it will print:

Ubuntu 20.04.4 LTS

**🧑‍💻 How You Added It to Your Script**

Here’s the **updated script logic**:

**1. Add source just after the shebang**

#!/bin/bash

source /etc/os-release

This loads all those useful variables like $NAME, $VERSION, and $PRETTY\_NAME.

**2. Use $PRETTY\_NAME in your echo line**

echo "Operating System: $PRETTY\_NAME"

This prints:

Operating System: Ubuntu 20.04.4 LTS

You can also use other variables from /etc/os-release, like:

* $NAME → just "Ubuntu"
* $VERSION\_ID → "20.04"

**🛠️ Final Updated Script (Simplified)**

#!/bin/bash

source /etc/os-release

INFO\_HOSTNAME=$(hostname)

INFO\_IP=$(hostname -I | cut -d ' ' -f2)

INFO\_KERNEL=$(uname -r)

for i in {1..25}

do

echo -n "="

done

echo ""

echo "Operating System: $PRETTY\_NAME"

echo "Host: $INFO\_HOSTNAME"

echo "IP: $INFO\_IP"

echo "Kernel: $INFO\_KERNEL"

echo ""

for i in {1..25}

do

echo -n "="

done

echo ""

**▶️ How to Run It**

Make sure it’s executable:

chmod +x info.sh

Then run it:

./info.sh

**🧾 Output Example**

=========================

Operating System: Ubuntu 20.04.4 LTS

Host: ubuntu1

IP: 192.168.56.101

Kernel: 5.15.0-91-generic

=========================

**🎉 Summary**

* /etc/os-release contains OS info as **variables**.
* source /etc/os-release imports those into your script.
* You can then **print system info** easily using variables like $PRETTY\_NAME.

Let me know if you want to add more system info (e.g. disk space, uptime, users) — I can help you extend this script further!

Of course! Let’s break this down **step-by-step in super simple terms**. You’re now building a **password generator script** using:

* A **WHILE loop**
* **User input**
* **Conditions (IF statements)**
* **Password encryption**

And it’s all in **Bash scripting**. Let’s go! 🚀

**🧾 What Does This Script Do?**

✅ It **asks the user to enter a password**  
✅ It checks that the password is **at least 6 characters long**  
✅ If it’s too short, it **asks again** (this is where the **loop** happens)  
✅ When it’s long enough, it **encrypts** the password and shows the result

**🛠️ Step-by-Step Breakdown**

**1. Start the Script with Shebang**

#!/bin/bash

This tells the system:  
👉 “Use the Bash shell to run this script.”

**2. Set Up a Variable to Control the Loop**

PWD\_OK=false

This variable keeps track of whether the password is acceptable.  
Right now it’s set to false, so the loop will begin.

**3. Create a WHILE Loop**

while [ "$PWD\_OK" != "true" ]; do

This means:  
🔁 “As long as the password is *not* okay, keep looping.”

**4. Ask the User for a Password**

read -s -p "Enter a new password: "

This does a few things:

* read gets input from the user
* -s hides the input (so your password isn’t visible)
* -p shows a message

It stores the entered password in a special variable called REPLY.

**5. Count the Length of the Password**

PWD\_LEN=$(echo -n "$REPLY" | wc -m)

This counts how many characters the user entered.

* -n makes sure we don’t count the "Enter" key as part of the length
* wc -m counts characters

**6. Encrypt the Password**

PASSWORD=$(openssl passwd -6 "$REPLY")

This uses OpenSSL to **encrypt** the password.

* -6 tells it to use **SHA-512** (a strong algorithm)

So PASSWORD now holds the **encrypted version**.

**7. Check If Password Is Long Enough**

if [ "$PWD\_LEN" -gt 6 ]; then

PWD\_OK=true

echo "Encrypted password: \"$PASSWORD\""

fi

* If password is **greater than 6 characters**, then:
  + Set PWD\_OK to true (so the loop stops)
  + Show the encrypted password

If the password is too short, it loops again.

**8. Close the Loop**

done

That ends the while loop.

**🧪 How Do You Use This Script?**

1. **Save the file** as pwgen.sh
2. Make it executable:
3. chmod +x pwgen.sh
4. Run it:
5. ./pwgen.sh

**👀 Example Run:**

**User enters short password:**

Enter a new password: 1234

Enter a new password: abc123

Encrypted password: "$6$randomsalt$encryptedstring"

**💡 Summary**

| **Feature** | **Purpose** |
| --- | --- |
| while | Keep looping until password is okay |
| read -s -p | Get password input secretly |
| wc -m | Count characters |
| if [ condition ] | Check if password is long enough |
| openssl passwd -6 | Encrypt password securely |

If you'd like, I can also:

* 🧪 Help you test this on your system
* 🛠️ Modify it to require numbers or special characters
* 💾 Save passwords to a file (if safe to do)

Just say the word!

Absolutely, I'm here to help — let's take it **step by step** and slow it down a bit so it's crystal clear. Here's what we’ll do:

**✅ What is the goal of the script?**

You created a **password generator script (pwgen.sh)** that does this:

1. Accept a password from the user
2. Encrypt the password
3. Show the encrypted version
4. Make sure the password is at least **6 characters long**
5. Allow the password to come from:
   * **User typing it in** (interactive)
   * OR as a **command-line argument** (non-interactive)

**📌 Step-by-step Explanation**

**Step 1: Start the script with a shebang**

#!/bin/bash

This tells Linux, “Run this file using Bash.”

**Step 2: Prepare a variable to control a loop (but we’ll stop using loop soon)**

PWD\_OK=false

This sets a variable that says whether the password is OK (true/false). We use it **to control the loop** (while loop). But later we change it so the script can **exit early** if the password is bad.

**Step 3: Decide where the password is coming from**

if [[ $# != 1 ]]; then

read -s -p "Enter a new password: "

echo

else

REPLY=$1

fi

* [[ $# != 1 ]] means: *If the user did not give exactly 1 argument...*
* Then: read -s will ask the user to type the password (without showing it).
* If they did give an argument, we save it to the variable REPLY.

🧠 **REPLY** = the password, no matter how it was entered.

**Step 4: Measure password length**

PWD\_LEN=$(echo -n "$REPLY" | wc -m)

* echo -n "$REPLY" means: print password but don’t add a newline.
* wc -m means: count characters.

We store the character count in PWD\_LEN.

**Step 5: Encrypt the password**

PASSWORD=$(openssl passwd -6 "$REPLY")

* This encrypts the password using openssl.
* -6 means use **SHA-512 encryption** (a secure algorithm).

**Step 6: Check if password is long enough**

if [ "$PWD\_LEN" -gt 6 ]; then

PWD\_OK=true

echo "Encrypted password: \"$PASSWORD\""

else

echo "Password too short"

exit 1

fi

* If the password is **more than 6 characters**, we say it's okay and show the encrypted version.
* If it’s too short, we print “Password too short” and exit 1.

💡 exit 1 stops the script and says “Something went wrong.”

You can check the result with:

echo $?

* 0 = success
* 1 = password too short

**🔁 Summary in Real-World Language**

| **Situation** | **What the script does** |
| --- | --- |
| You run ./pwgen.sh | It asks you to type a password |
| You run ./pwgen.sh secret123 | It uses secret123 as your password |
| You type a short password | It says “Password too short” and quits |
| You type a good password | It encrypts and shows it |
| You run with a short password like ./pwgen.sh 123 | It says “Password too short” and quits |

**💡 Why All This Is Useful**

* Lets you **choose** how to enter a password (typed or passed in)
* Avoids prompting if you already passed a password
* Exits early if the password is bad — saves time
* Encrypts passwords safely

Would you like me to:

* 🔁 Go over any one part more slowly?
* 📄 Give you the **full final script**?
* 🎥 Walk through an example session like you’re typing it in?

Just let me know — we’ll go at your pace!

Absolutely! Let's break it all down into **simple and clear steps** — like how you'd explain it to a friend who's new to Linux.

**🧠 What Are We Doing?**

We are:

✅ Creating **three user accounts** (jo, jill, sam)  
✅ Giving each user the **same password** (Password123?)  
✅ Encrypting the password using a script we wrote (pwgen.sh)  
✅ Using a **for loop at the command line** to do all this automatically  
✅ Verifying that users were created and that their passwords work

**💡 Why Are We Doing This?**

* We want to learn how to use **loops in the terminal**
* We want to reuse our **password script**
* We want to **automate user creation**

**✅ Step-by-Step (Simple Version)**

**1️⃣ The For Loop in the Command Line**

Here’s the structure:

for user in jo jill sam; do

# commands go here

done

🧠 This means:

* "For each name in the list (jo, jill, sam), do the stuff inside the loop."

**2️⃣ Inside the Loop – Creating the User**

sudo useradd -m -s /bin/bash -p "$(./pwgen.sh Password123?)" "$user"

Let’s break this down:

| **Part** | **Meaning** |
| --- | --- |
| sudo | Run with admin rights (you need this to create users) |
| useradd | Command to create a new user |
| -m | Create a home directory for the user |
| -s /bin/bash | Set the user’s shell to bash |
| -p | Set the password (must be encrypted) |
| "$(./pwgen.sh Password123?)" | Run our script to **generate encrypted password** |
| "$user" | The username for this loop round (jo, jill, or sam) |

So each time, we’re doing:

➡️ "Create a new user (jo/jill/sam), give them a bash shell, a home folder, and a securely encrypted version of the same password."

**3️⃣ End the Loop**

done

That just tells the terminal: “We’re finished looping.”

**4️⃣ What Happens When You Hit Enter?**

It automatically:

* Creates user **jo** with password Password123? (encrypted)
* Then **jill** with the same clear password but different encrypted one
* Then **sam** in the same way

🎉 All done in one go!

**5️⃣ How to Check If It Worked**

tail -n3 /etc/passwd

This shows the last 3 users in the user database — you should see jo, jill, sam.

sudo tail -n3 /etc/shadow

This shows the encrypted passwords (only root can see them). You’ll see different hashes for each user.

Why? 🔐 Because encryption adds **random “salt”**, so even if the password is the same, the encrypted result is unique.

**6️⃣ Test It!**

You can try logging in as a new user:

su - jo

Enter Password123?, and boom — you're now jo!

**✅ Final Summary**

| **What you did** | **Why it matters** |
| --- | --- |
| Used a **loop** to save time | No need to create users one-by-one |
| Used your **own script** (pwgen.sh) | Reused work and made it dynamic |
| Used useradd with options | Created real usable accounts |
| Checked /etc/passwd and /etc/shadow | Verified accounts and security |
| Logged in with su | Confirmed it all worked |

**Want me to give you the full command ready to copy and run? Or do you want a version using a file list of names (like a .txt)? Let me know!**

Absolutely! Here's a **simple explanation** of everything you just read, broken down clearly:

**🐧 What Was Covered in the Course So Far?**

You've been learning how to **write Bash scripts** in Linux. These scripts help automate tasks like showing system info, generating passwords, or creating users.

**🗃️ Where to Get the Example Scripts**

You don’t have to start from scratch. You can **download all the example scripts** used in the course from a **GitHub repository**.

**To download them:**

1. Open your terminal.
2. Type this command:
3. git clone https://github.com/theurbanpenguin/lfcs
4. That will download all course files into a folder called lfcs.
5. Inside lfcs, go to:
6. lfcs/operating/shell-scripting/

That’s where all the example scripts are.

**✅ What Have You Learned So Far?**

Here’s a **summary of key points** from this shell scripting module:

**🔍 1. File & Shebang**

* The file command shows the type of a script.
* If the script has a **shebang** (#!/bin/bash), Linux knows it's a Bash script.

**💬 2. echo command**

* Prints text.
* echo -n means **don't go to the next line** (no newline).

**📥 3. read command**

* Takes user input and stores it.
* If you use read -s, it hides what you're typing (useful for passwords).
* read -p "Prompt text" shows a message before waiting for input.

**🔁 4. Loops**

* **For loop**: Repeats actions for a list or a range.
* for i in {1..25}; do echo -n "="; done

→ Prints 25 equal signs.

* **While loop**: Keeps going **while a condition is true**.
* **Until loop**: Keeps going **until a condition becomes true**.

**🧠 5. Variables and Commands**

* You can store the result of a command into a variable:
* KERNEL=$(uname -r)
* Then use it later:
* echo "Kernel version: $KERNEL"

**📂 6. source command**

* Loads another script or file into the current one.
* Used to import system info from /etc/os-release:
* source /etc/os-release
* echo $PRETTY\_NAME

**🔒 7. Password Generator Script**

* You wrote (or downloaded) a script pwgen.sh to **encrypt passwords** using openssl.
* This lets you safely create new users with secure passwords.

**🧪 8. Special Bash Variables**

* $# → Number of arguments passed to the script
* $0 → Script name
* $1, $2 → First, second argument, etc.
* $? → Exit status of the last command (0 = success, not 0 = error)

**👤 9. Creating Users in a Loop**

* You used a for loop at the command line to create multiple users (jo, jill, sam) using:
* for user in jo jill sam; do
* sudo useradd -m -s /bin/bash -p $(./pwgen.sh Password123?) $user
* done
* Each gets the same clear-text password (Password123?), but their **encrypted passwords are different** for security.

**🕒 What’s Coming Next?**

The next topic in the course is:

**✅ Scheduling tasks**

* Learn to automatically run tasks at specific times.
* Tools you'll see:
  + cron (runs tasks on schedule)
  + at (runs once at a specific time)
  + systemd timers (modern way to schedule tasks)

**👣 What Should You Do Now?**

* Try creating the scripts **yourself**.
* Play around with:
  + for loops
  + read and echo
  + argument handling ($1, $#)
* Clone the GitHub repo:
* git clone https://github.com/theurbanpenguin/lfcs
* Practice using and modifying the example scripts.

Would you like help navigating the repo or modifying one of the example scripts to suit your own use case?