03

ITE229 (SERVER-SIDE / LINUX WEB INFRASTRUCTURE)

Proof-of-Concept Technical Solution: Secured WordPress Website for Marconi Law Firm

A three-VM LAMP, Docker/Ghost, and Nginx reverse-proxy build with defense-in-depth hardening on a custom 10.10.229.0/24 network

COURSE
ITE229 (Server-side / Linux web infrastructure)

DATE
2025

ENVIRONMENT
Custom network ITE229: subnet 10.10.229.0/24, mask 255.255.255.0, DNS/gateway 10.10.229.1

PAGES (REPORT)
54

STUDENT
Cody Richard

ORGANIZATION
Black Sands Security LLC (client: Marconi Law Firm, LLC – Orlando, Florida)

TECHNICAL BRIEF · EXECUTIVE SUMMARY

Proof-of-Concept Technical Solution: Secured WordPress Website for Marconi Law Firm

I authored a 54-page proof-of-concept and audit-documentation package delivering a production-style WordPress website for the Marconi Law Firm, built across three Linux VMs on a custom 10.10.229.0/24 network behind a firewall/router VM gateway (10.10.229.1). I stood up a full LAMP stack on Ubuntu (10.10.229.12) running WordPress cloned from GitHub against a dedicated MySQL database, deployed a containerized Ghost CMS on a Rocky 8 Docker host (10.10.229.11), and fronted the Ghost site with an Nginx reverse proxy on a second Rocky 8 host (10.10.229.10) that routes /blog traffic to the container. I then applied defense-in-depth hardening: chmod/chown lockdown of /var/www/html (755 dirs, 644 files), relocating wp-config.php above the web root with 600 permissions, and installing the Wordfence Layer-7 WAF plugin — resolving a 403 error and a redirect loop in the process. Every step is documented as graded audit evidence with commands, explanations, and verification screenshots.

▸ Objectives

- Deliver a working WordPress website for a law-firm client as a documented proof of concept and audit artifact
- Design and build a segmented custom network (ITE229, 10.10.229.0/24) with a firewall/router VM gateway
- Deploy a full LAMP stack (Linux/Apache2/MySQL/PHP) on Ubuntu hosting WordPress
- Containerize a Ghost (Node.js) CMS on a Rocky 8 Docker host and expose it via an Nginx reverse proxy
- Harden the WordPress deployment with defense-in-depth controls (file permissions, config secrecy, application firewall)
- Produce comprehensive audit documentation supporting compliance, validation, records maintenance, and decision-making

▸ Environment

```
Custom network ITE229: subnet 10.10.229.0/24, mask 255.255.255.0, DNS/gateway 10.10.229.1
```

```
Firewall/Router VM gateway at 10.10.229.1
```

```
Rocky Linux 8 Docker host 'DockerRichard' at 10.10.229.11 (Ghost container)
```

```
Rocky Linux 8 Nginx reverse-proxy host 'NGINXRichard' at 10.10.229.10
```

```
Ubuntu LAMP host 'UbuntuXRichard' at 10.10.229.12 running WordPress
```

```
CLI-only VMs administered over OpenSSH; Firefox on Ubuntu used for browser verification
```

WALKTHROUGH & EVIDENCE

This is the proof-of-concept build I delivered for the Marconi Law Firm: a production-style, secured WordPress site spanning three Linux VMs on a custom 10.10.229.0/24 network behind a firewall/router gateway at 10.10.229.1. The work covers a full LAMP stack on Ubuntu (10.10.229.12), a containerized Ghost CMS on a Rocky 8 Docker host (10.10.229.11), an Nginx reverse proxy on a second Rocky 8 host (10.10.229.10), and defense-in-depth hardening of the WordPress deployment. Every figure below is the original graded audit evidence, re-presented with commands, IPs, and outcomes preserved exactly as captured.

ARCHITECTURE

Network Topology

Three-tier segmented design on the custom ITE229 network (10.10.229.0/24, mask 255.255.255.0) with the firewall/router VM as gateway and DNS at 10.10.229.1.

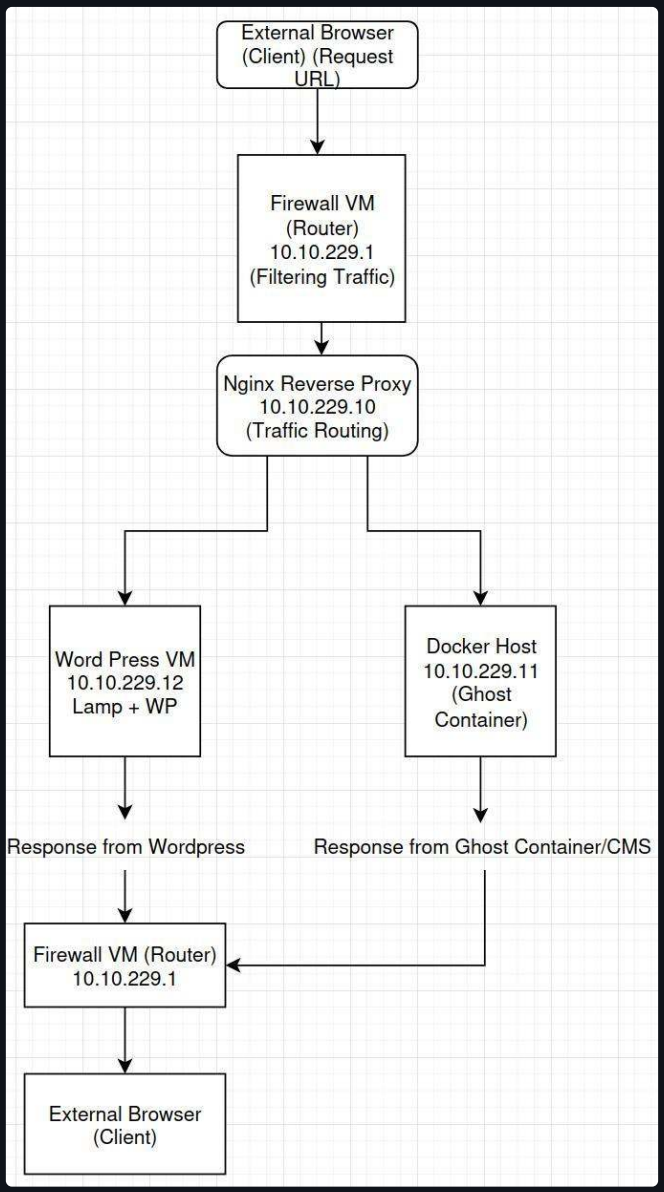

FIG 01 — Network topology diagram for the ITE229 build: firewall/router gateway at 10.10.229.1 fronting the Docker host (.11), Nginx reverse proxy (.10), and Ubuntu LAMP/WordPress host (.12) on the 10.10.229.0/24 subnet.

SSH ACCESS

OpenSSH Client and Host Login

Installed the OpenSSH client on Ubuntu to administer the CLI-only VMs remotely.

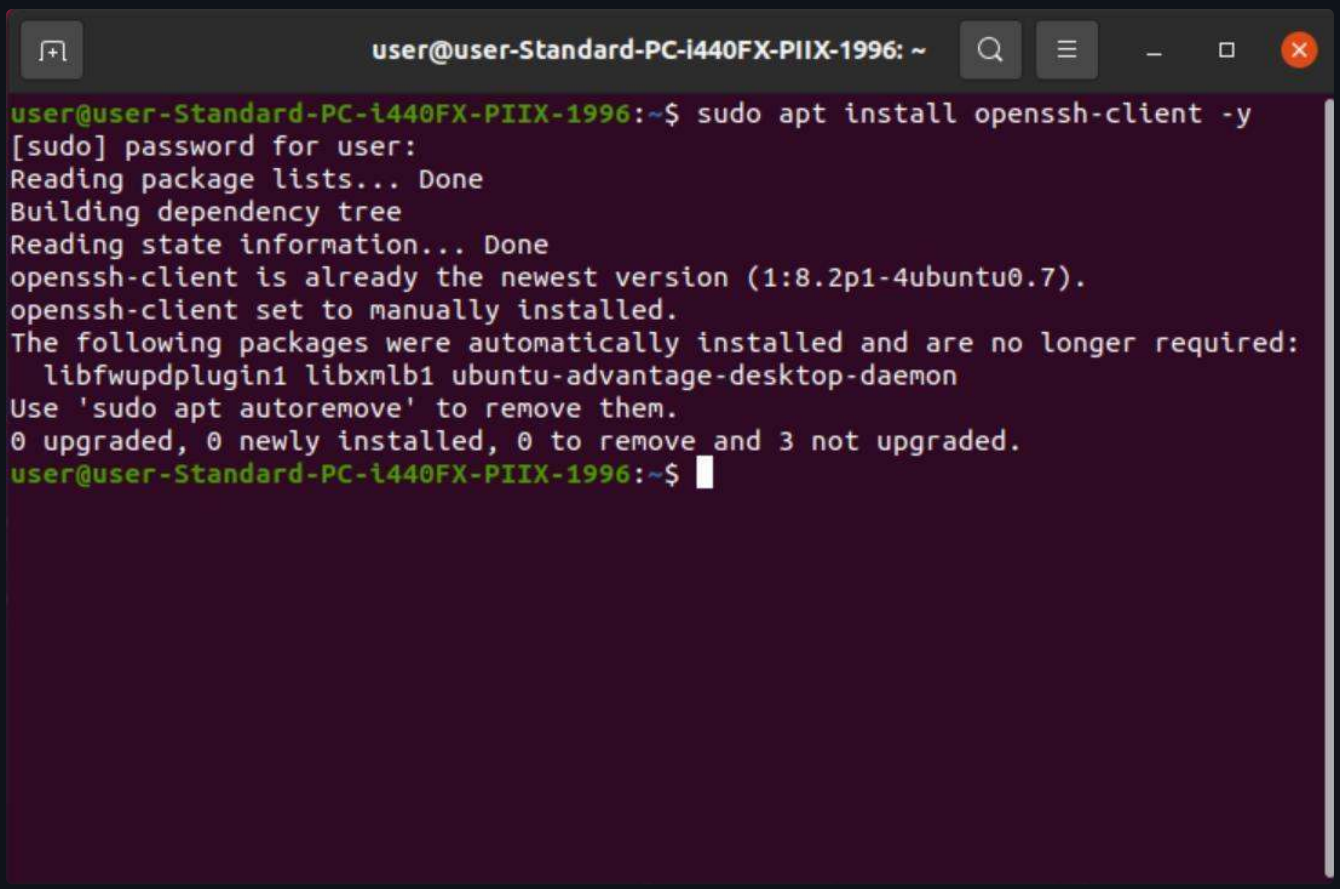

FIG 02 — Installing the OpenSSH client on Ubuntu with sudo apt install openssh-client -y to enable remote administration of the Docker and Nginx hosts.

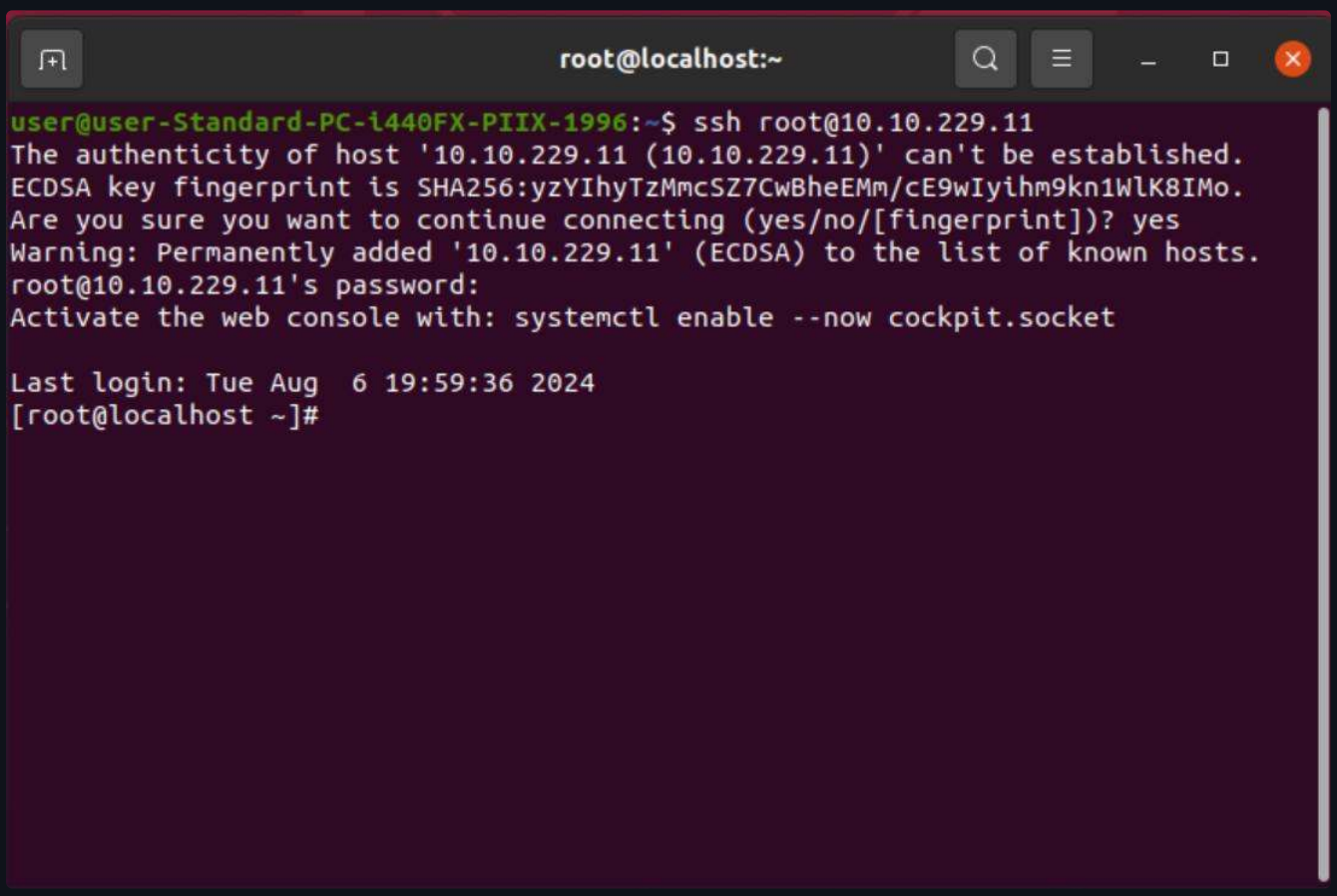

FIG 03 — SSH session to the Docker host: ssh root@10.10.229.11 establishing a remote login to the Rocky 8 container host.

SSH ACCESS

Reverse-Proxy Host Login and Docker Hostname

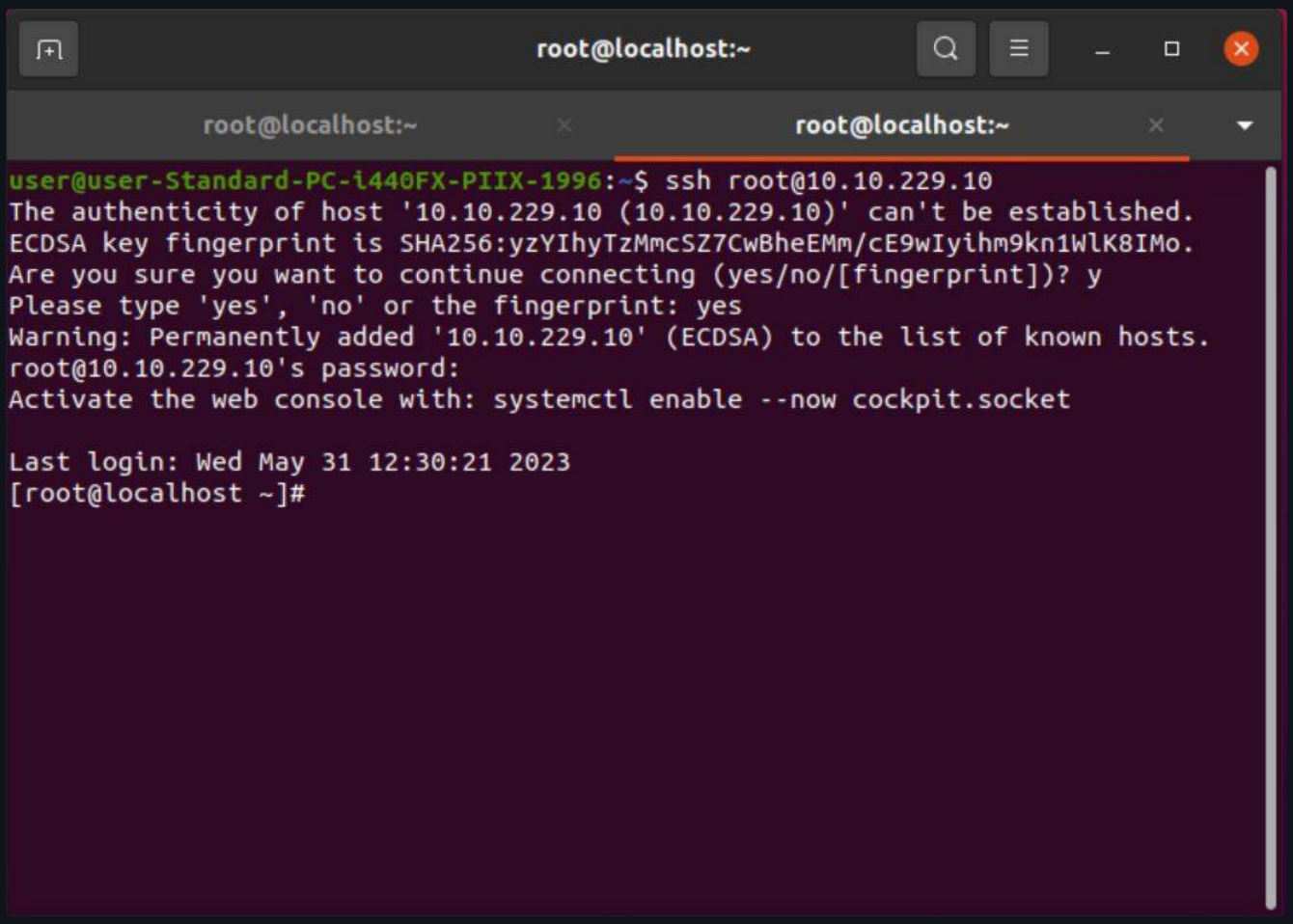

FIG 04 — SSH session to the Nginx reverse-proxy host: ssh root@10.10.229.10 confirming remote access to the second Rocky 8 VM.

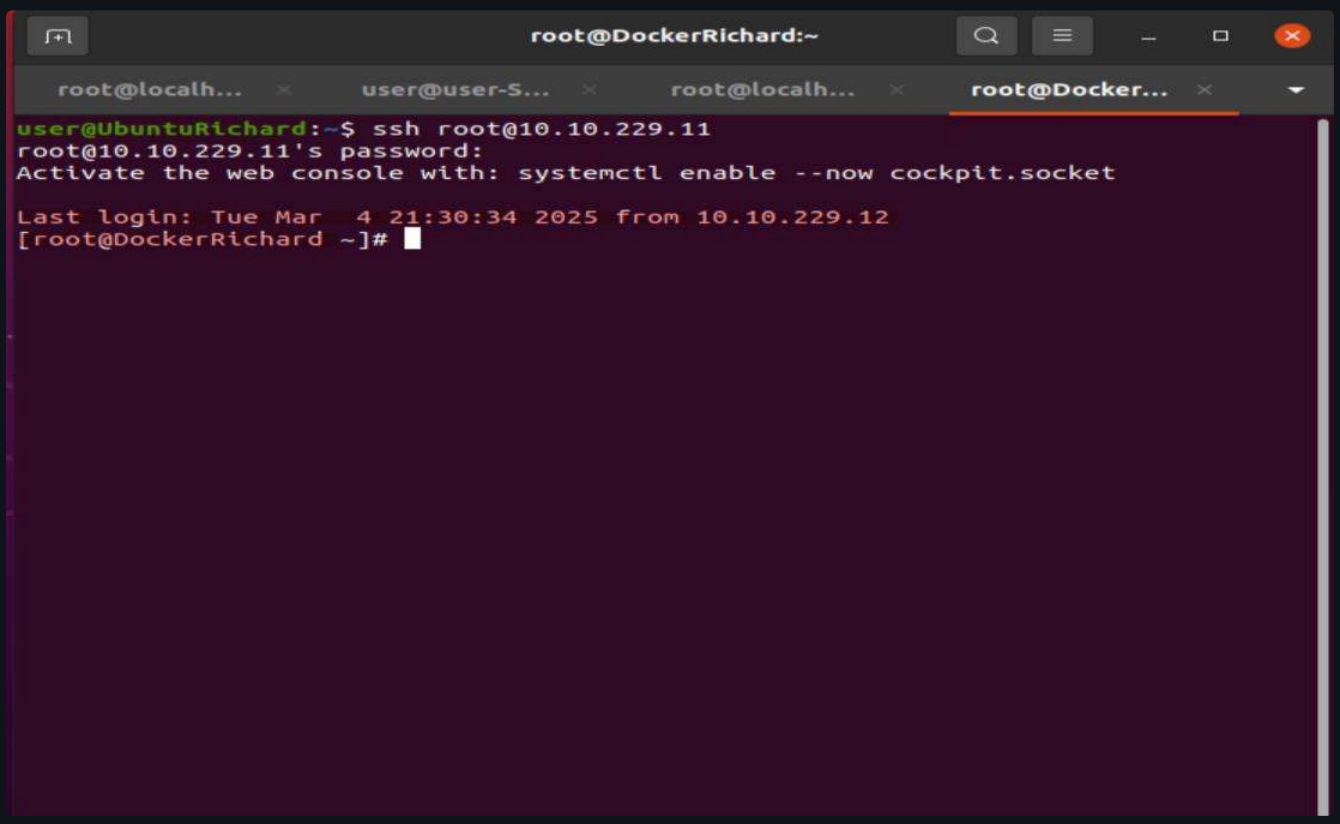

FIG 05 — nmtui System Hostname editor on the Docker host, preparing to set a meaningful hostname for the CLI-only VM.

DOCKER HOST

Set Hostname and Update Packages

Renamed the Docker host to DockerRichard and brought the system fully up to date.

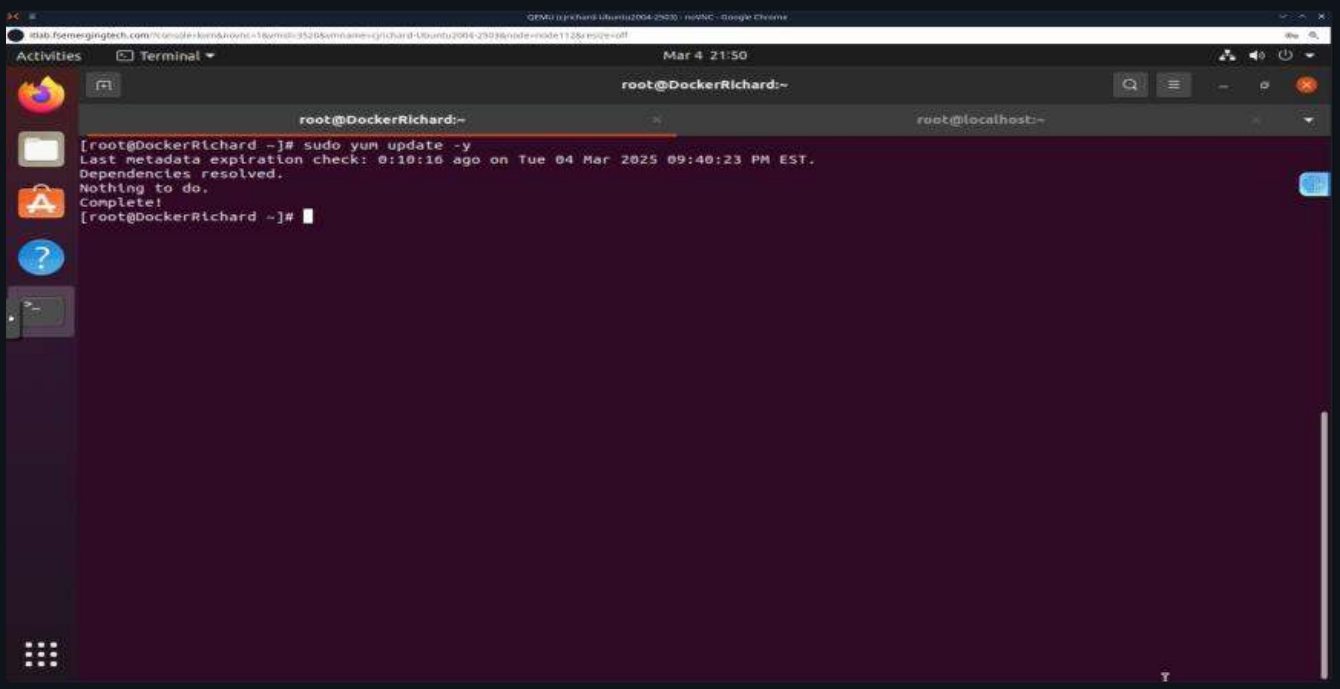

FIG 06 — Docker host renamed to DockerRichard via nmtui, simplifying SSH identification across the VM fleet.

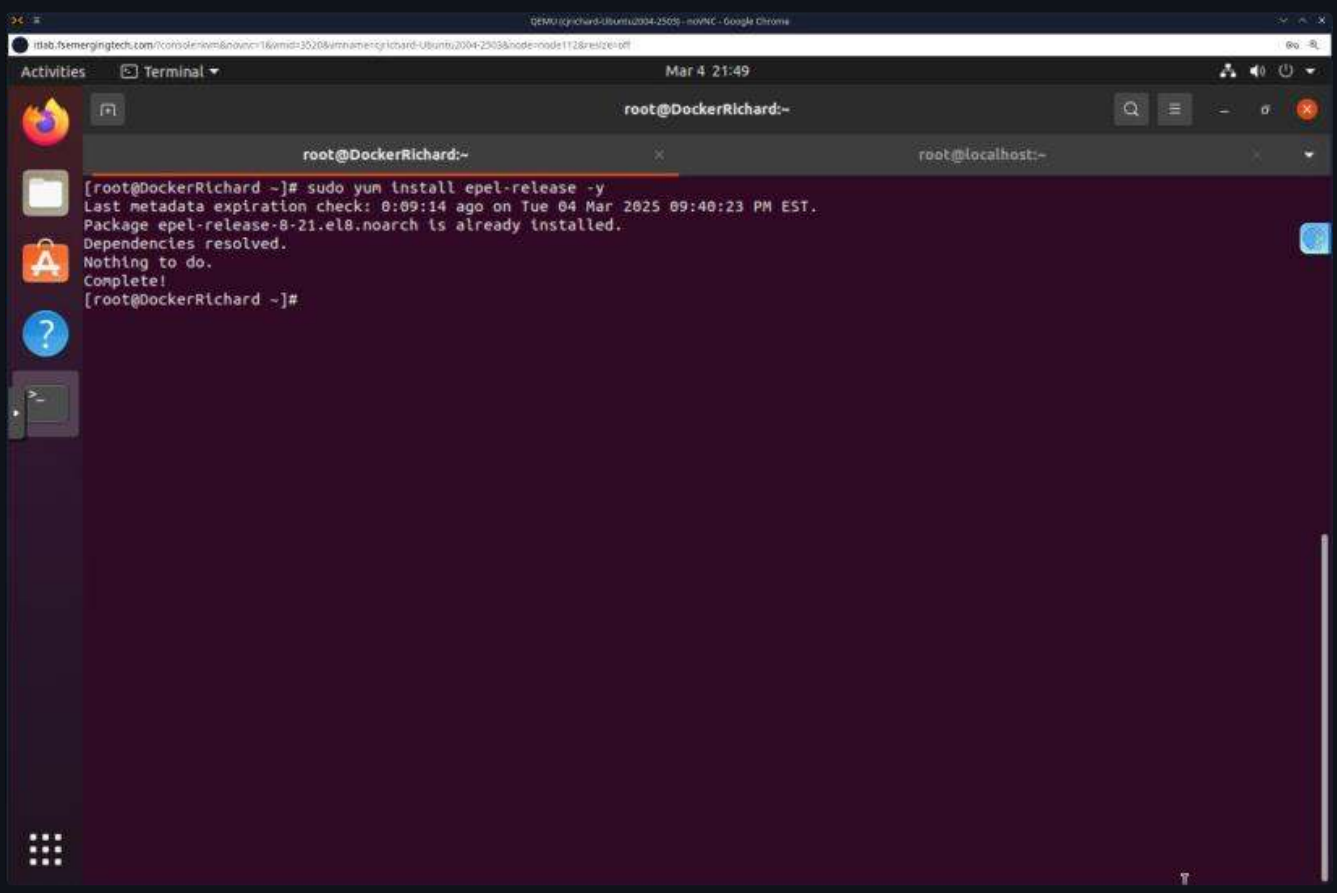

FIG 07 — sudo yum update -y on the Rocky 8 Docker host, refreshing all installed packages for security and stability.

DOCKER HOST

EPEL and Nano

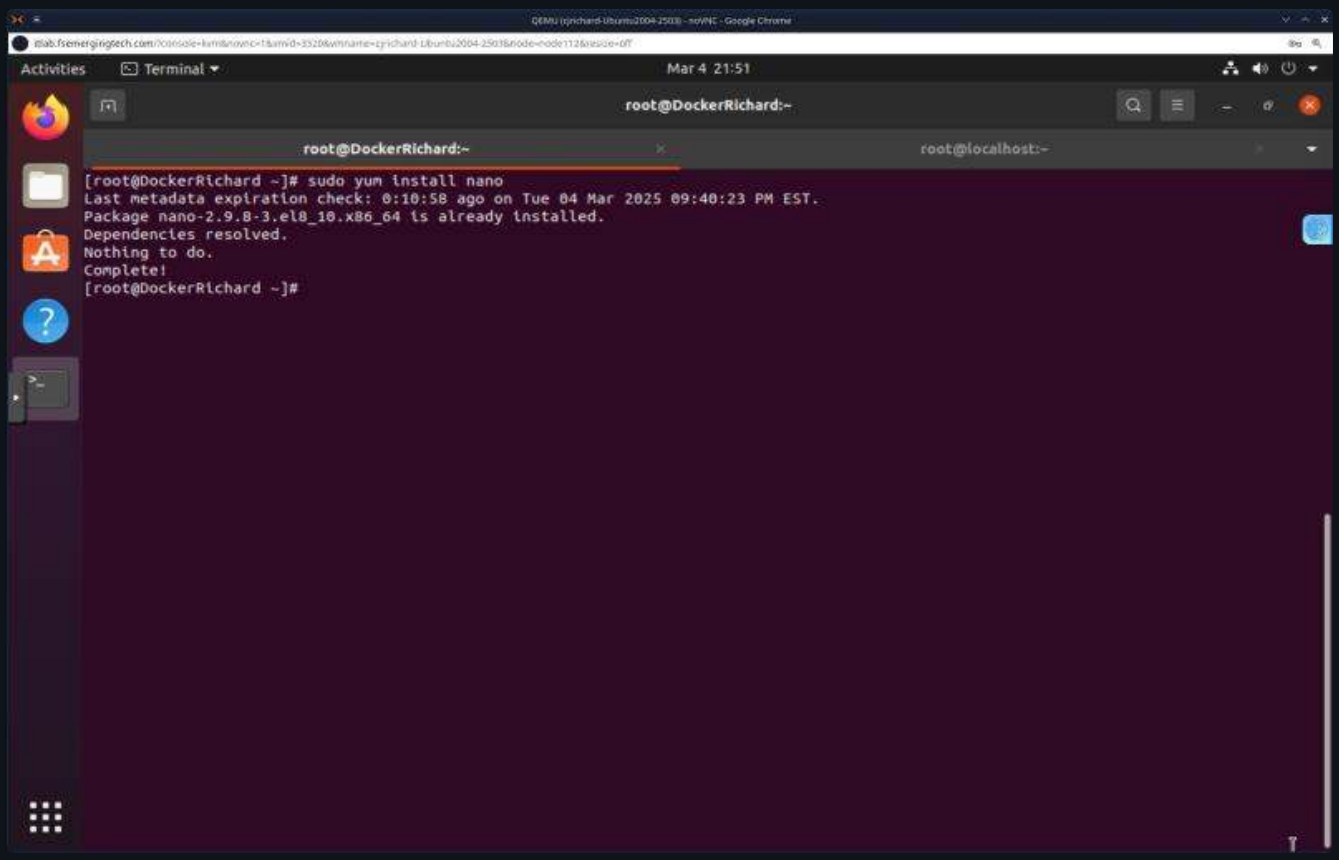

FIG 08 — Installing the EPEL repository with sudo yum install epel-release -y to make additional packages available on Rocky 8.

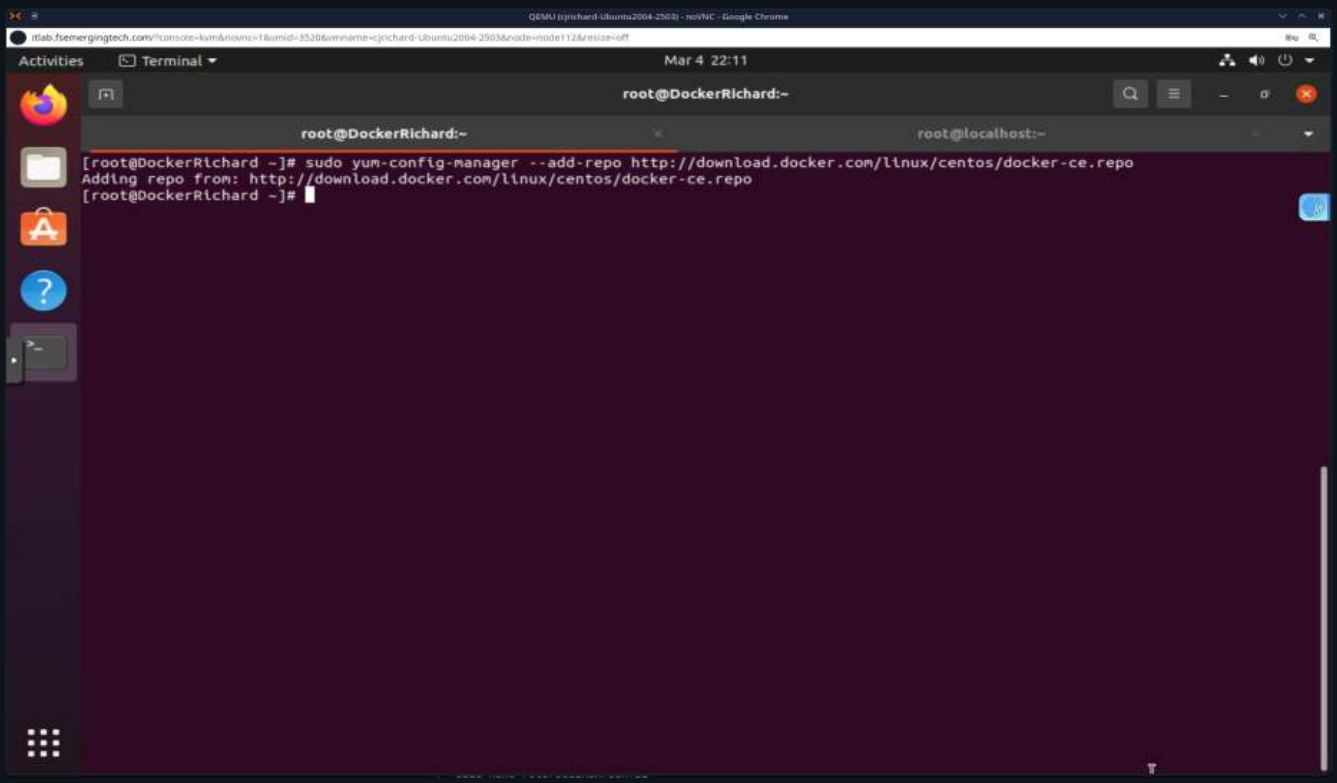

FIG 09 — Adding the Docker CE repository with sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo for installs straight from the official source.

DOCKER HOST

Install Docker CE and Verify Version

Installed Docker CE with its CLI, runtime, buildx, and compose plugins, then confirmed the version.

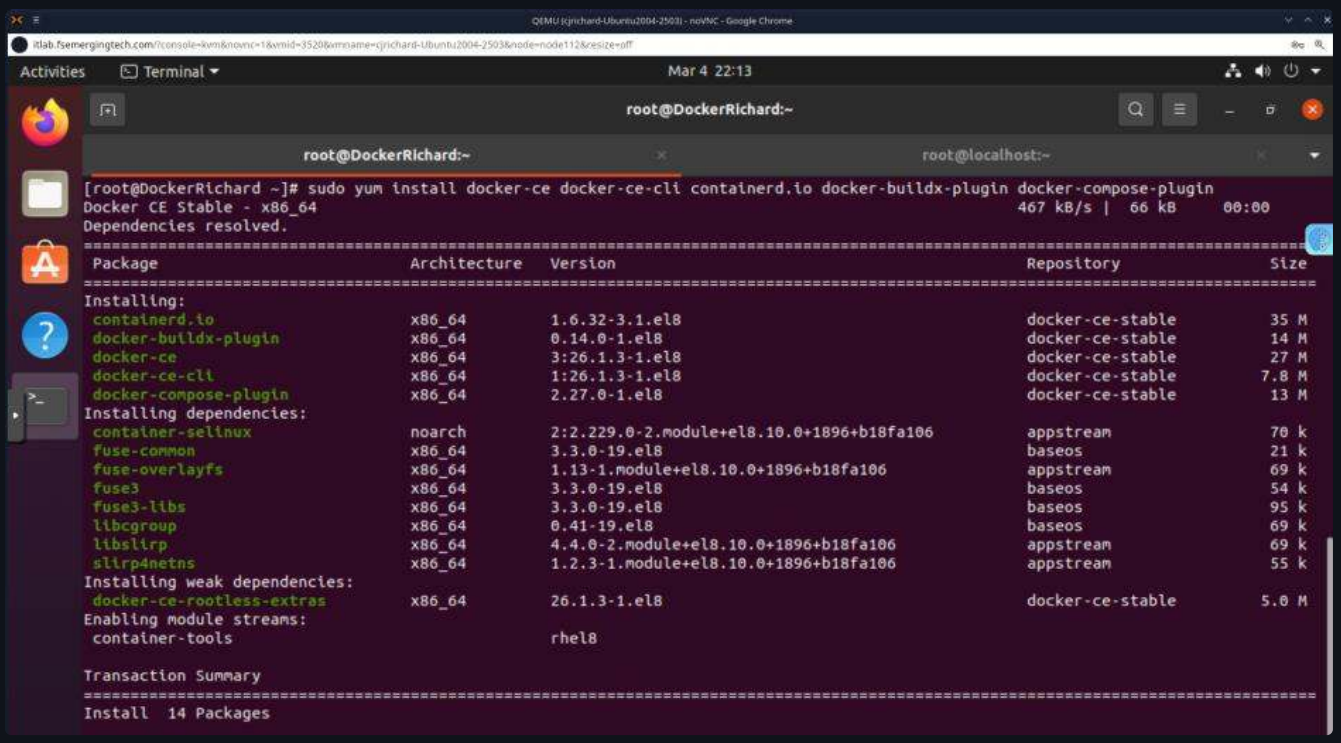

FIG 10 — Installing Docker CE and components: sudo yum install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin.

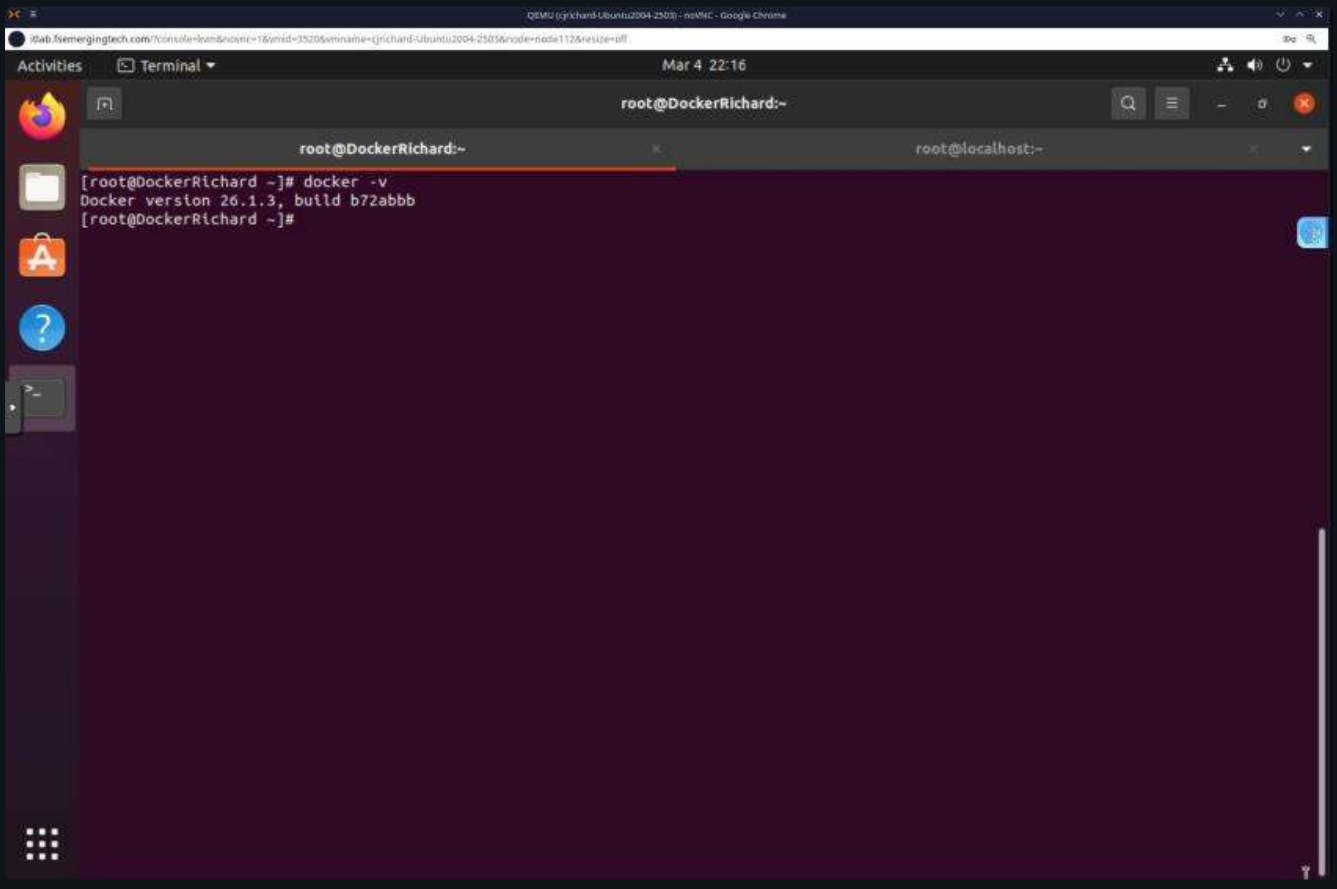

FIG 11 — Verifying the install with docker -v, confirming the Docker CE version was deployed correctly.

DOCKER HOST

Start and Enable the Docker Service

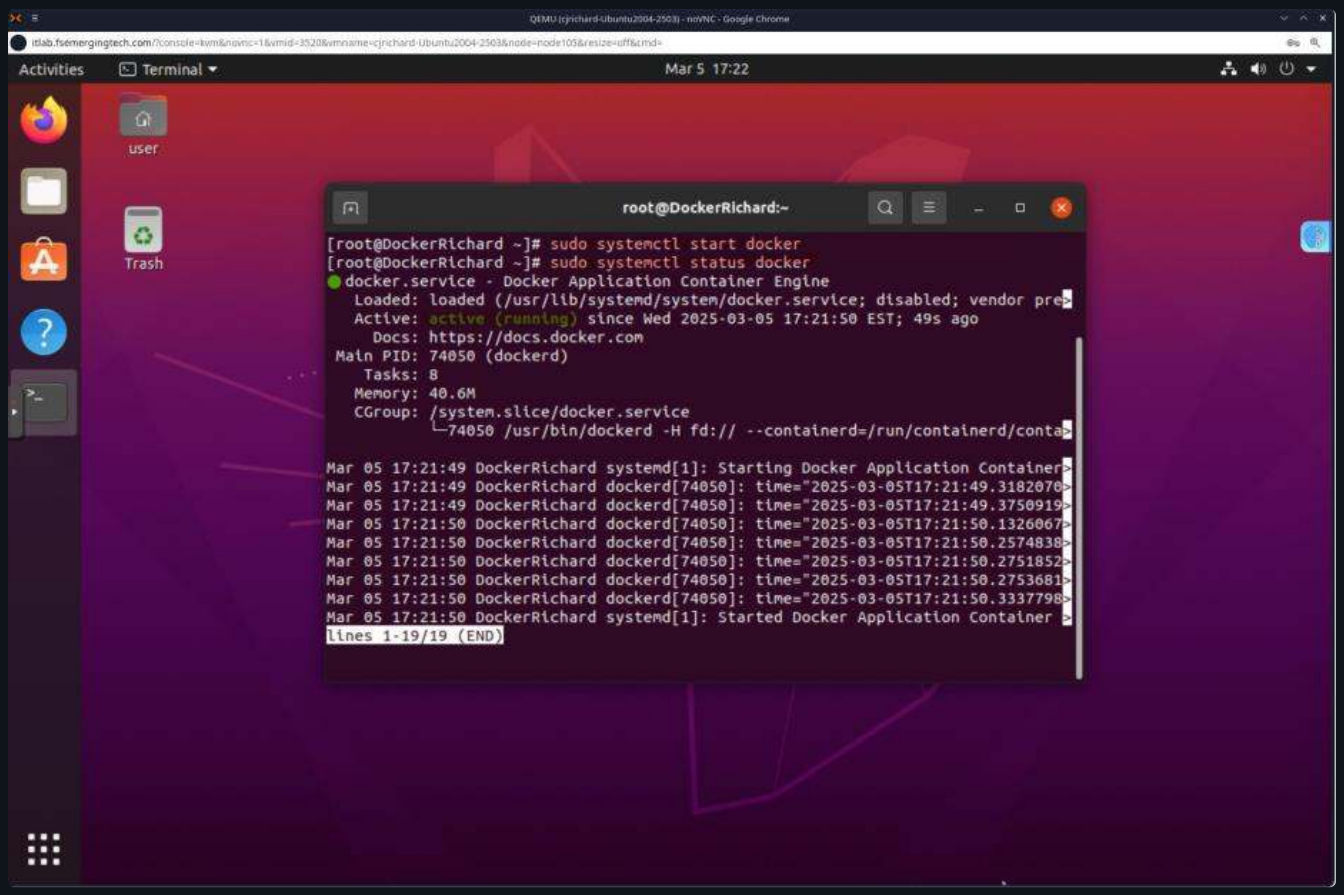

FIG 12 — Starting the daemon with sudo systemctl start docker and checking sudo systemctl status docker to confirm it is active and running.

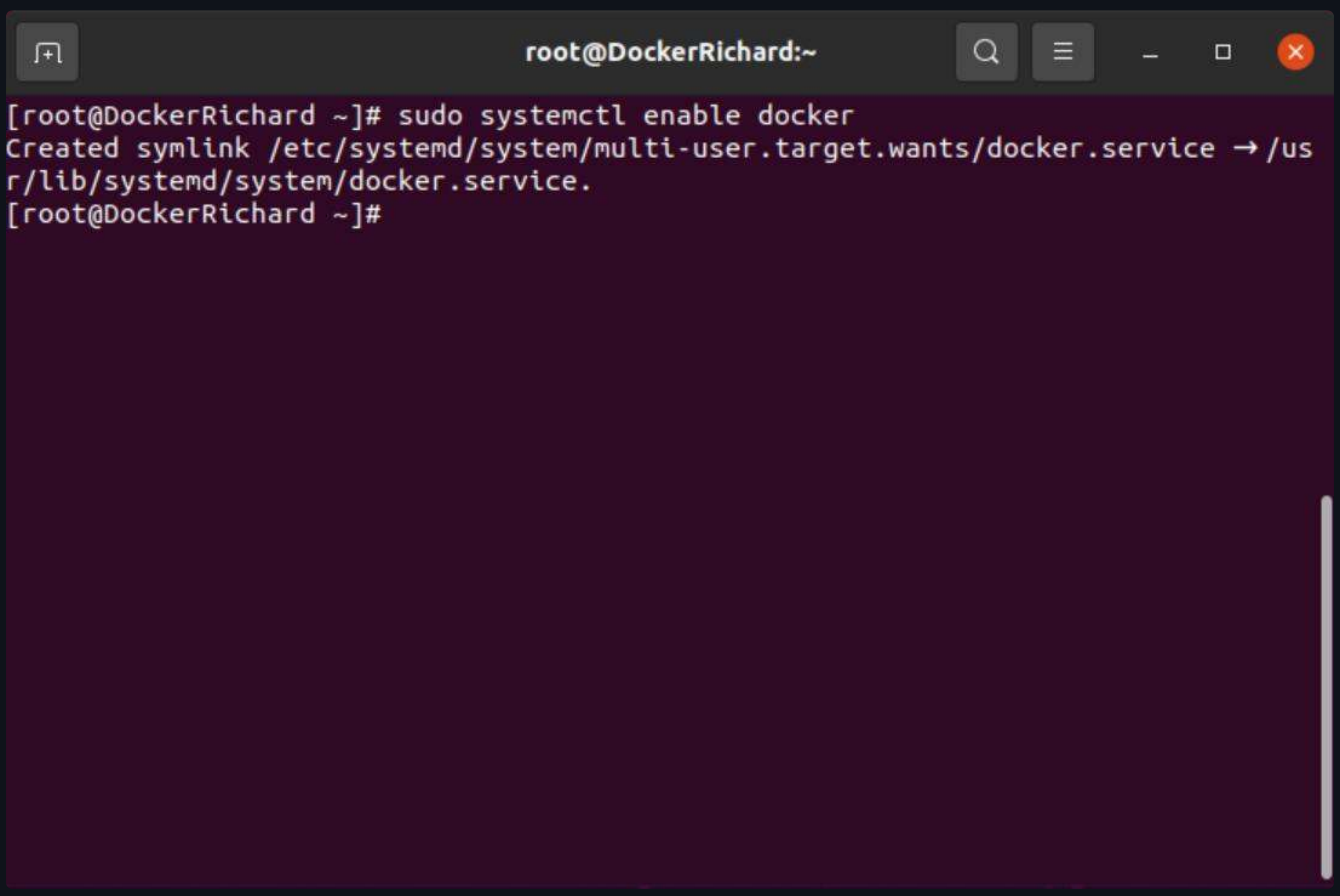

FIG 13 — Persisting the service across reboots with sudo systemctl enable docker.

DOCKER HOST

Validate Docker and Edit SELinux

Ran the hello-world container to prove the install, then disabled SELinux to avoid conflicts for this build.

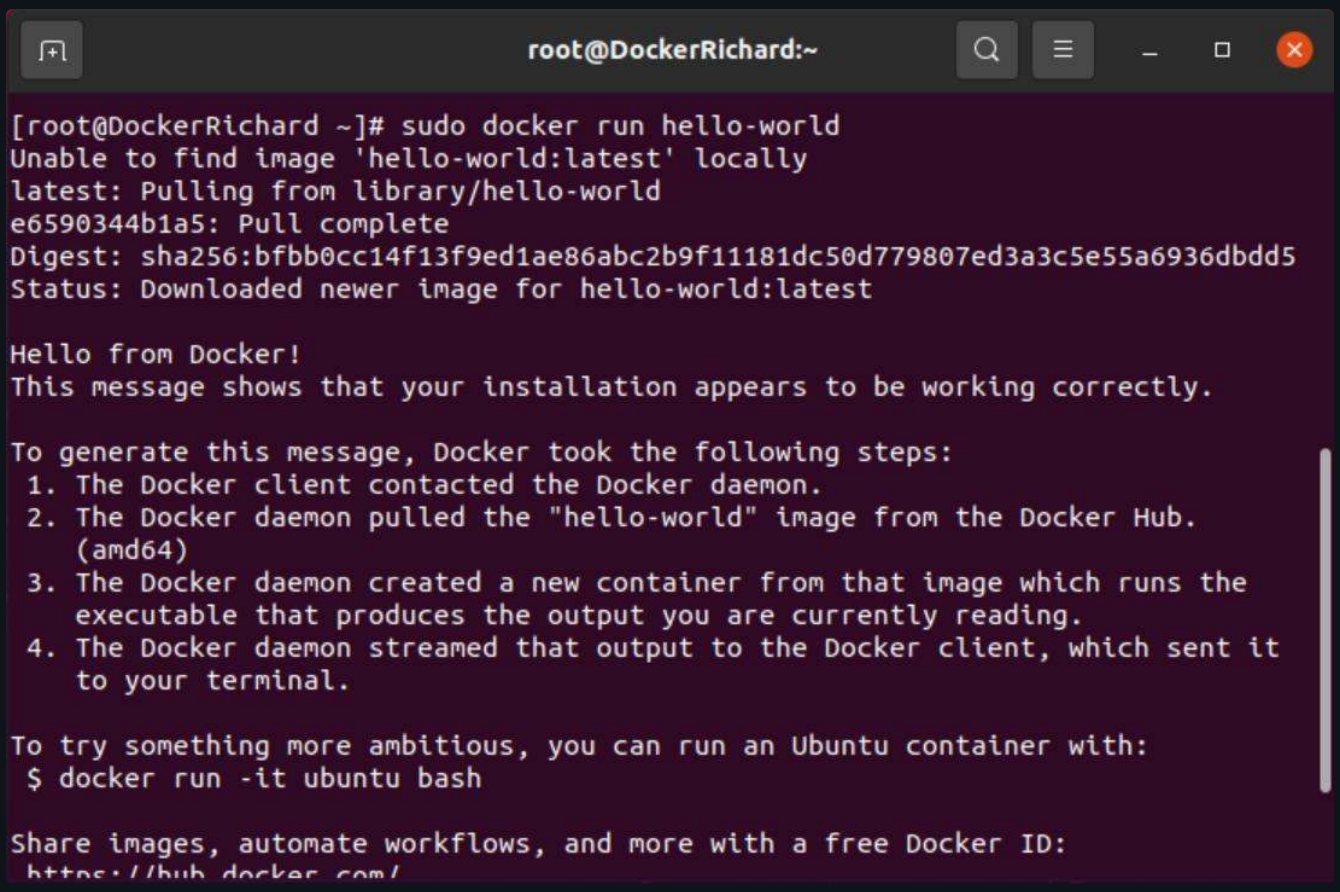

FIG 14 — sudo docker run hello-world returning the success message, validating that Docker is installed and operating correctly.

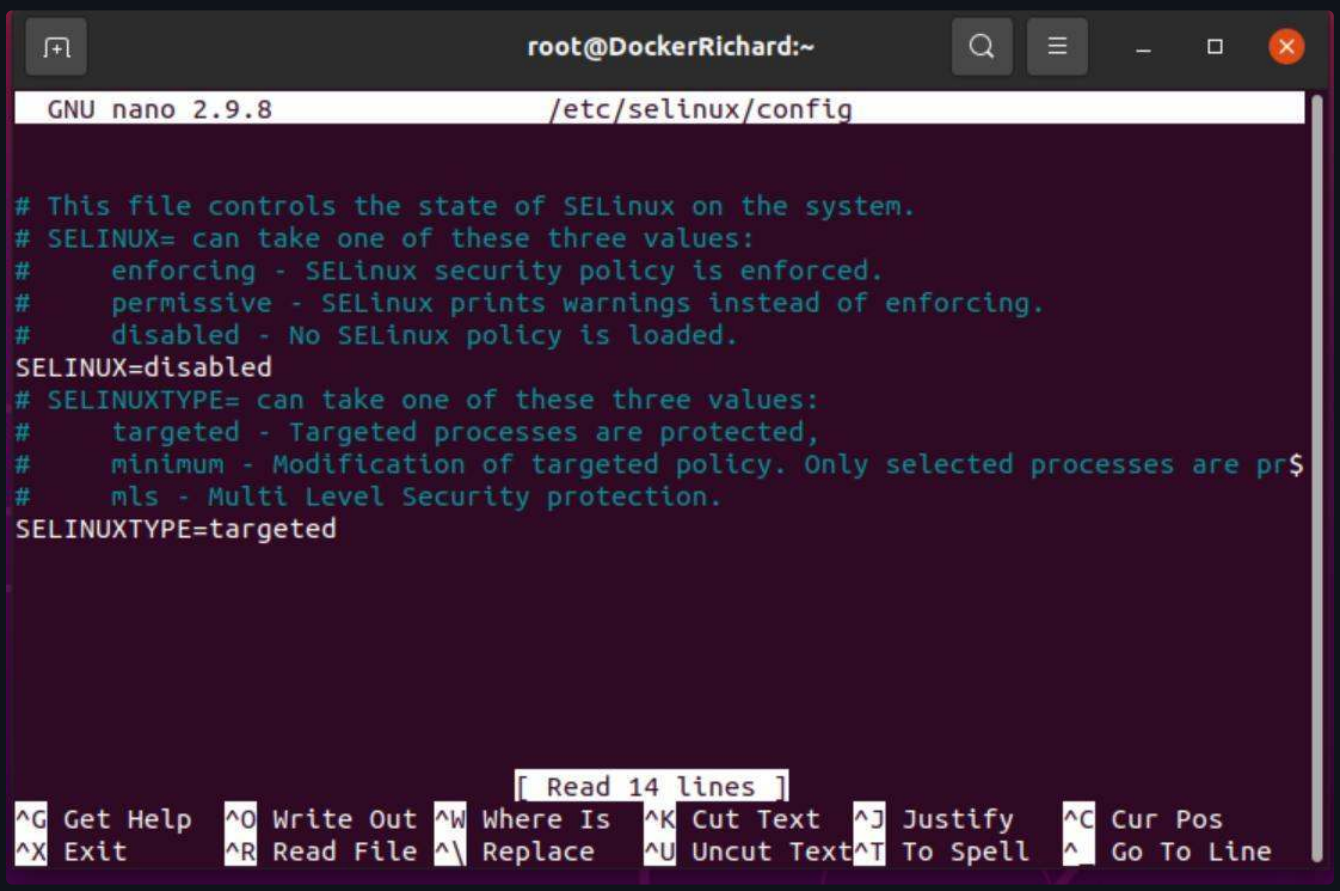

FIG 15 — Editing /etc/selinux/config with nano, changing enforcing to disabled for this assignment.

DOCKER HOST

Reboot and Confirm SELinux Status

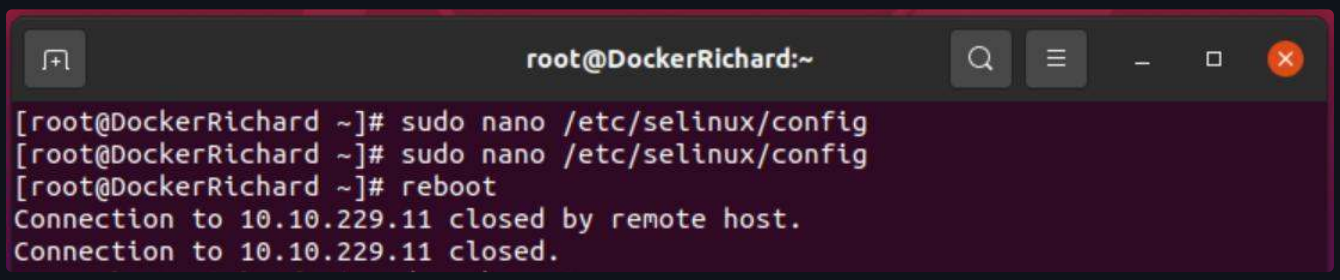

FIG 16 — Issuing reboot to apply the SELinux configuration change on the Docker host.

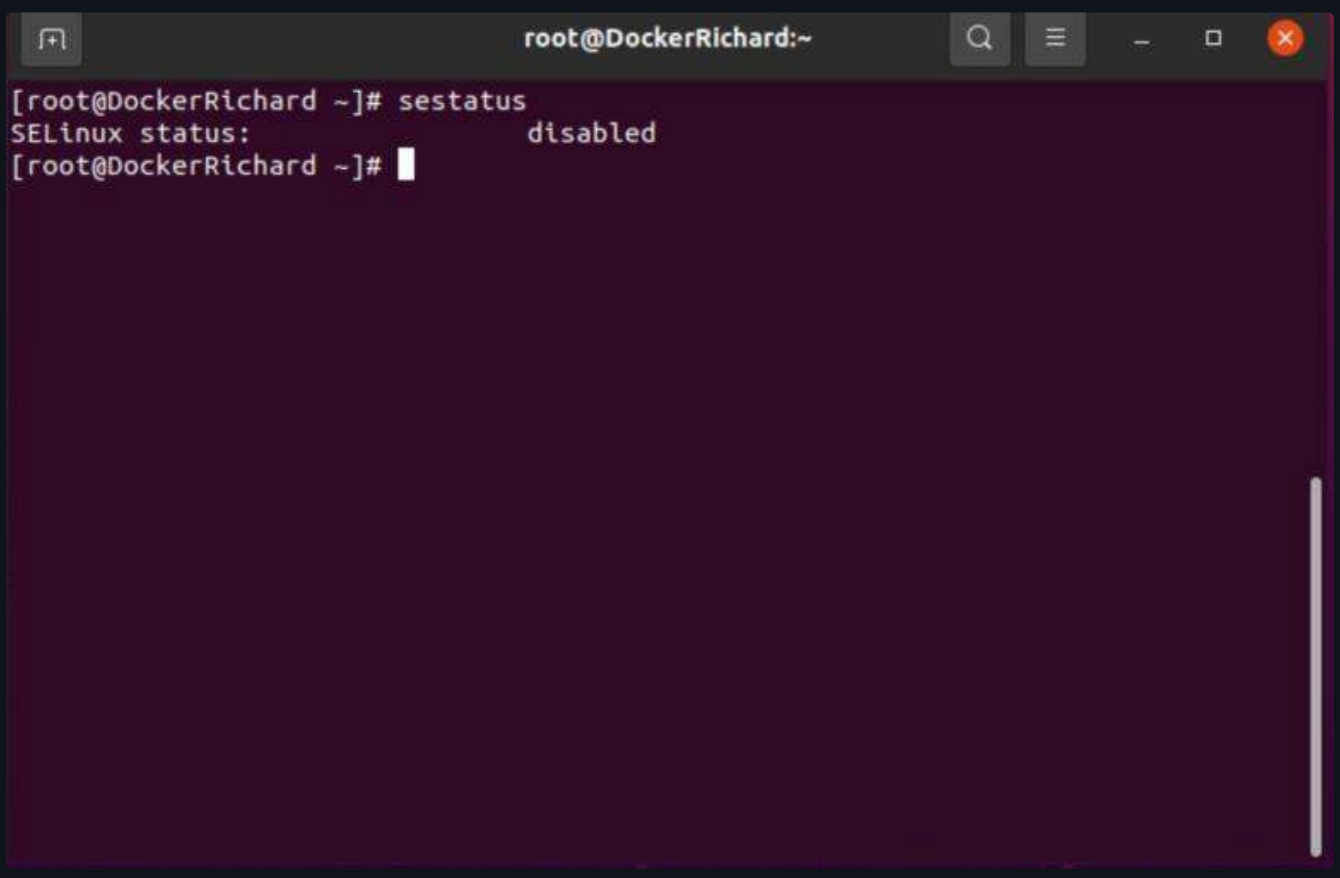

FIG 17 — sestatus after reboot confirming SELinux is now disabled.

GHOST CMS

Deploy the Ghost Container

Stood up the Ghost CMS container, mapping host port 3001 to the container's 2368.

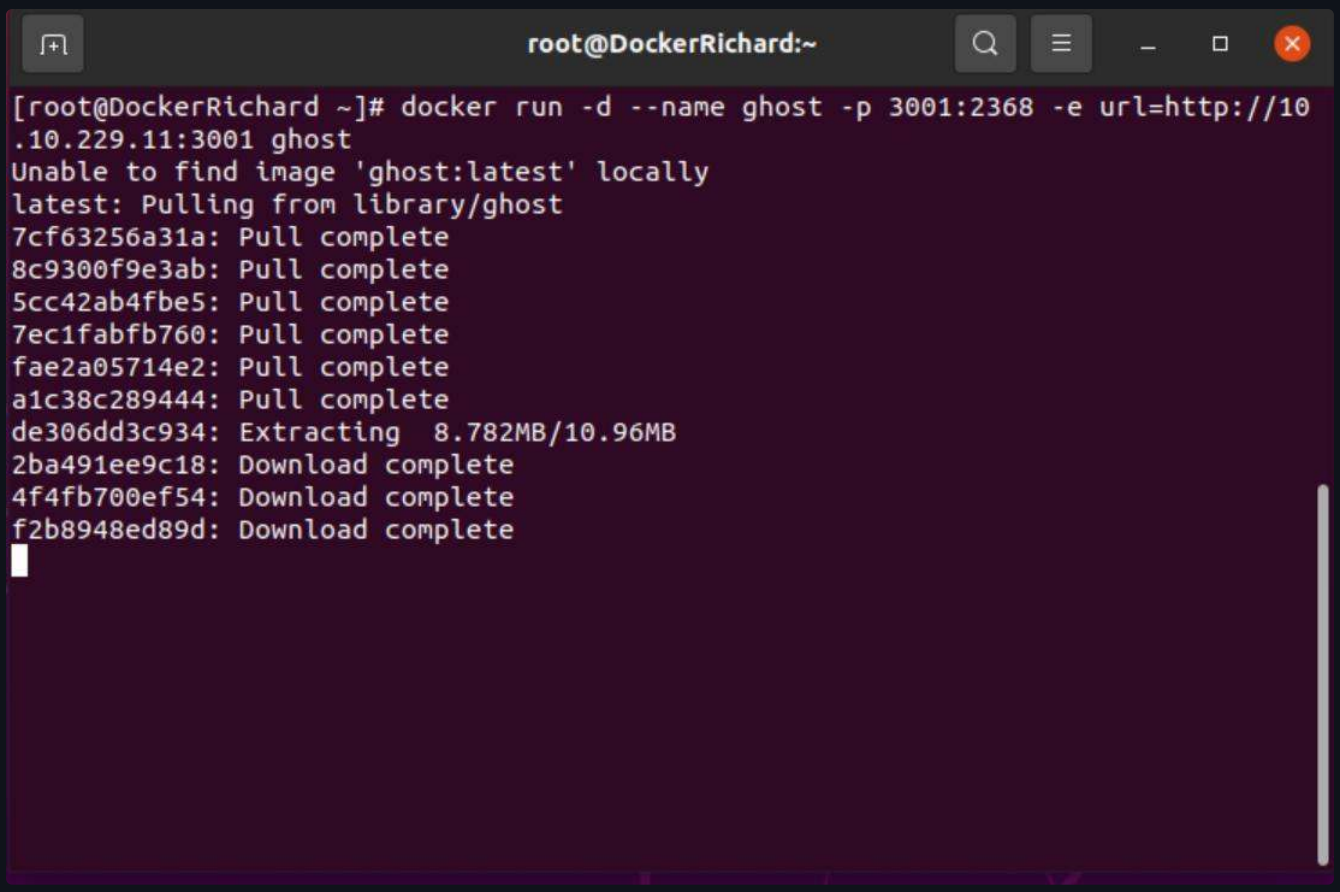

FIG 18 — Launching Ghost: docker run -d --name ghost -p 3001:2368 -e url=http://10.10.229.11:3001 ghost, creating and starting the blog container.

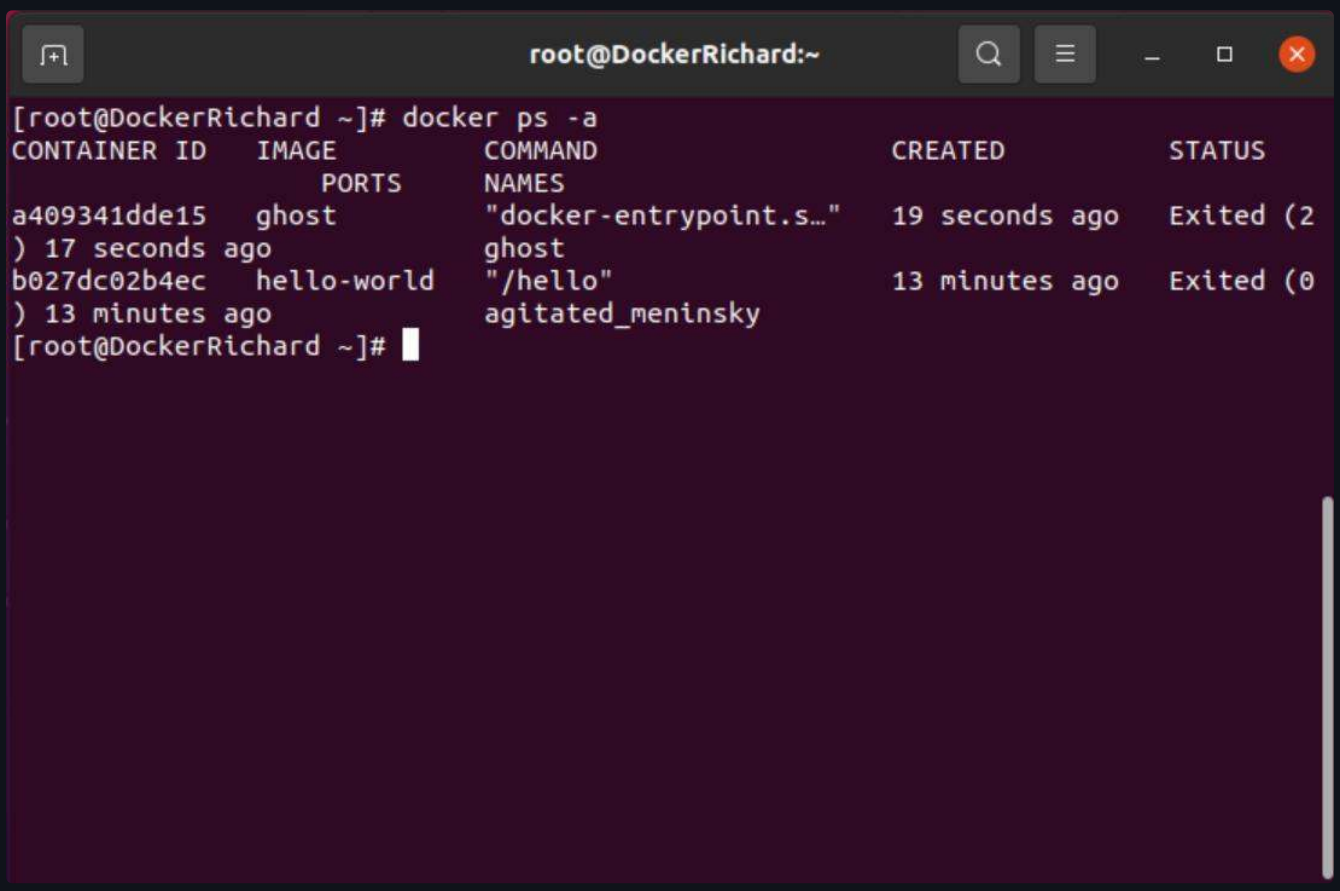

FIG 19 — docker ps -a listing all containers and capturing the Ghost container ID for verification.

PROXY HOST

Set Hostname and Update Packages

Prepared the second Rocky 8 host (the reverse proxy): renamed to NGINXRichard and updated.

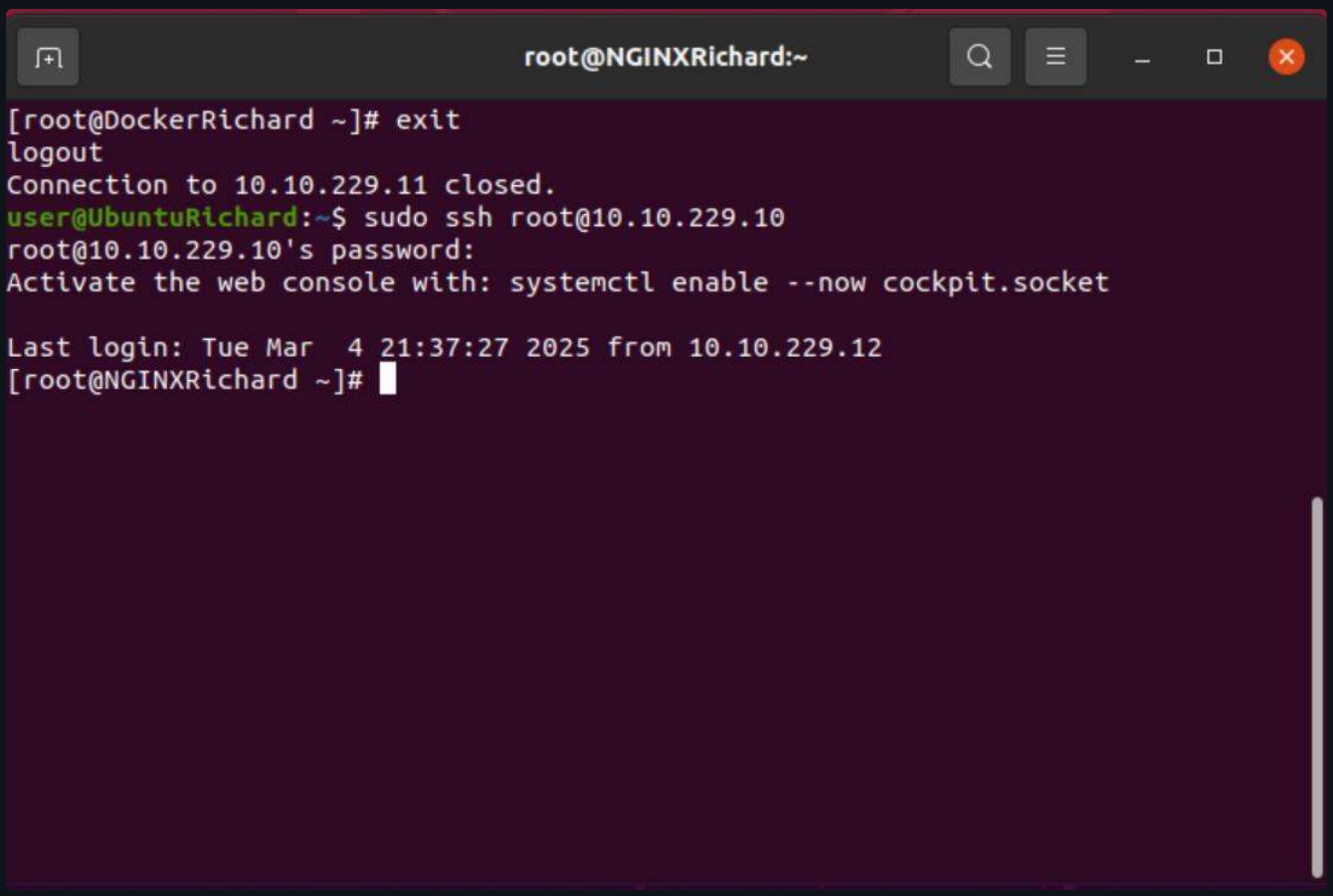

FIG 20 — Reverse-proxy host renamed to NGINXRichard via nmtui for clear identification across the fleet.

FIG 21 — sudo yum update -y on the Nginx host, refreshing all packages for security and stability.

PROXY HOST

EPEL and Nano

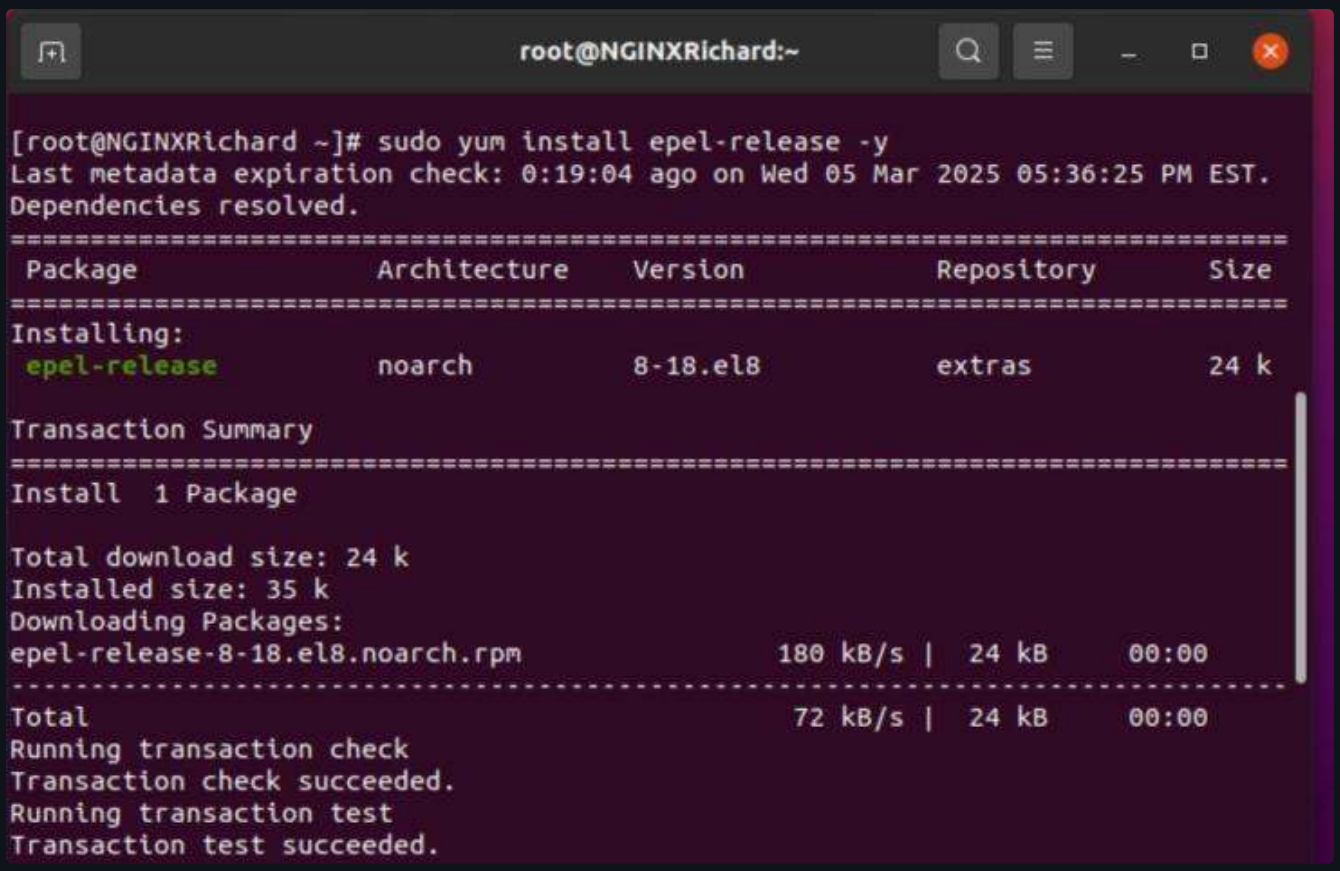

FIG 22 — Installing the EPEL repository with sudo yum install epel-release -y on the Nginx host.

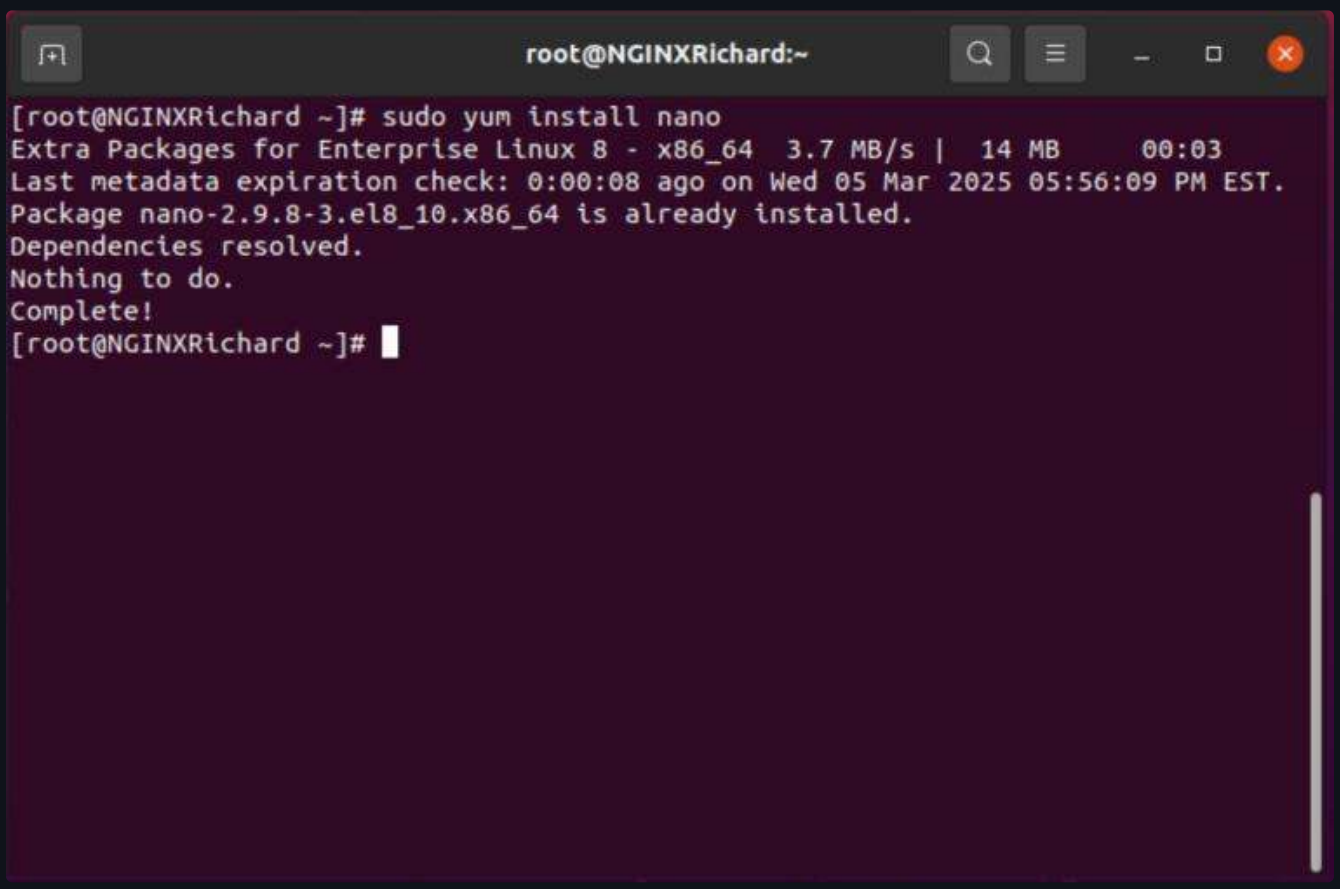

FIG 23 — Installing the nano editor with sudo yum install nano for config file editing.

PROXY HOST

Disable SELinux and Reboot

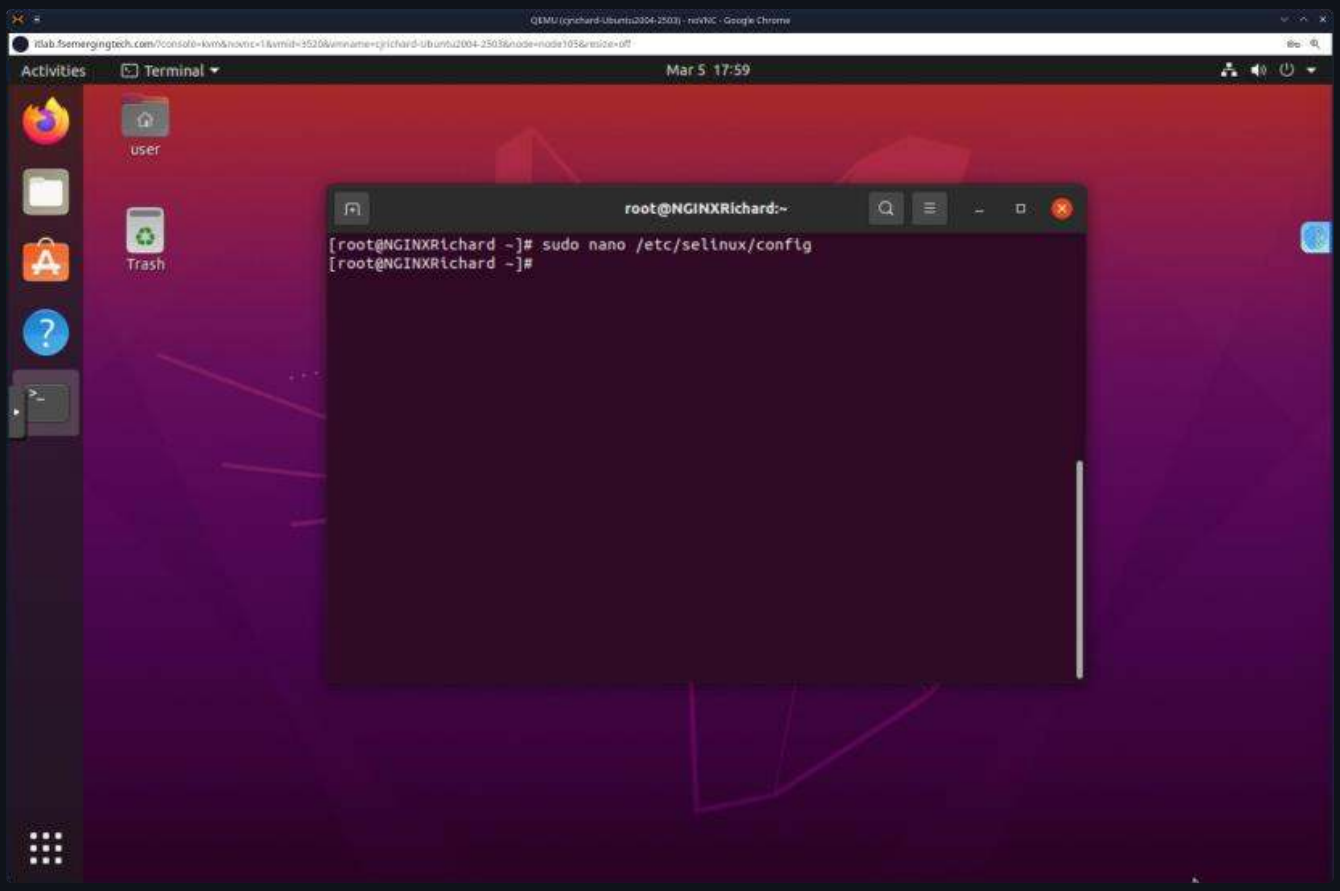

FIG 24 — Editing /etc/selinux/config on the Nginx host, switching enforcing to disabled.

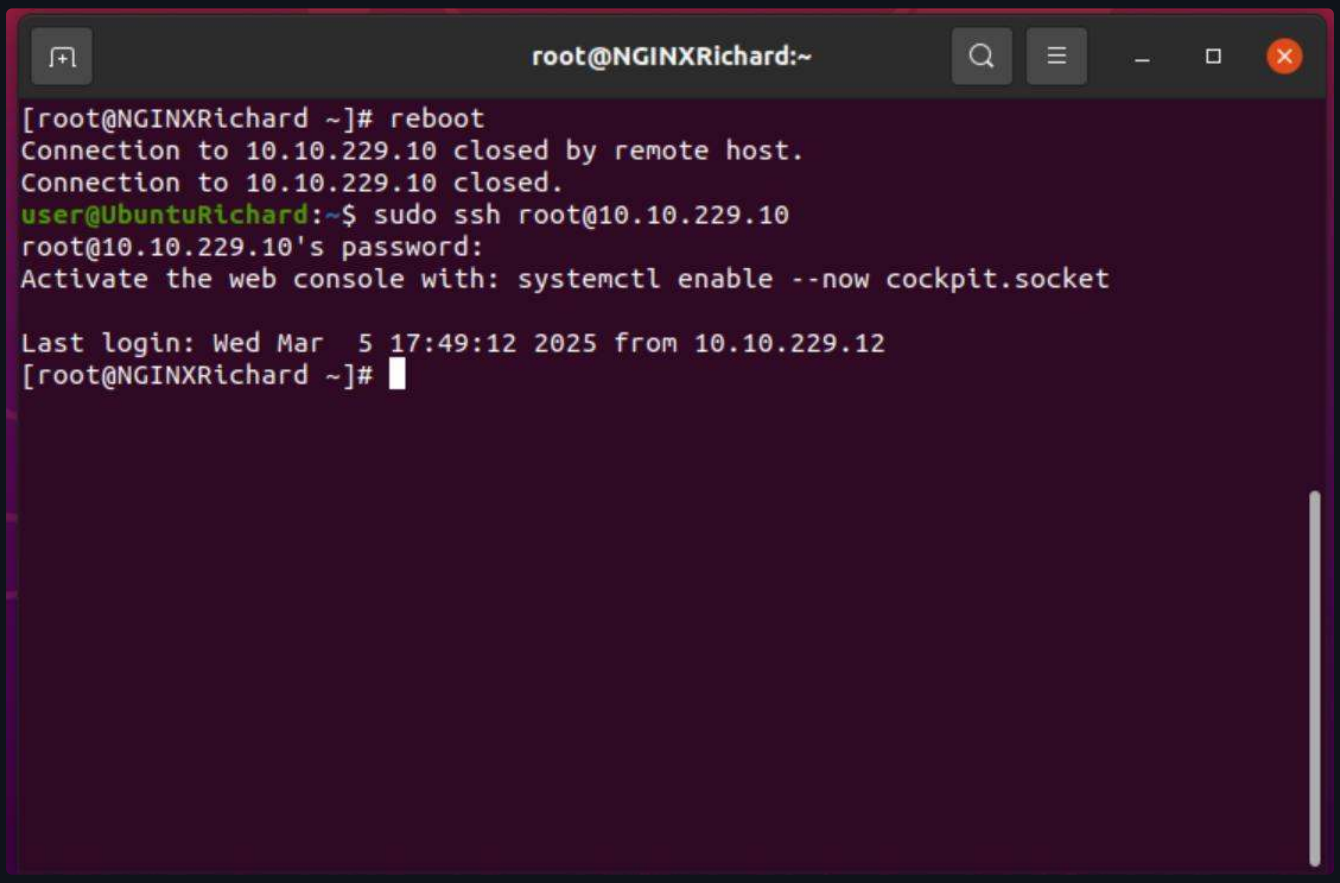

FIG 25 — Issuing reboot to apply the SELinux change on the reverse-proxy host.

PROXY HOST

Confirm SELinux and Stop firewalld

Verified SELinux disabled, then stopped firewalld so the proxy could receive HTTP/HTTPS on ports 80 and 443.

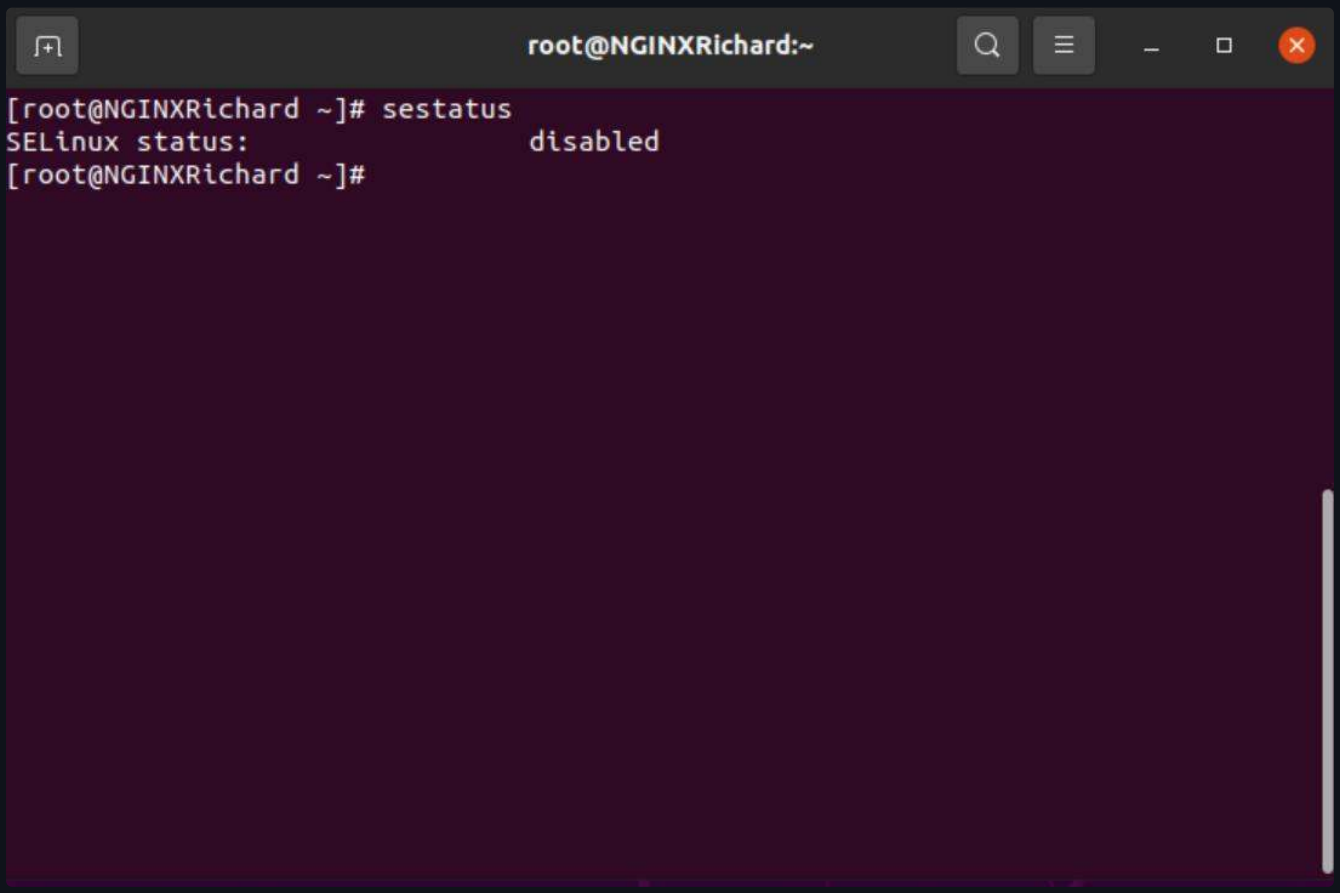

FIG 26 — sestatus confirming SELinux is disabled on the Nginx host after reboot.

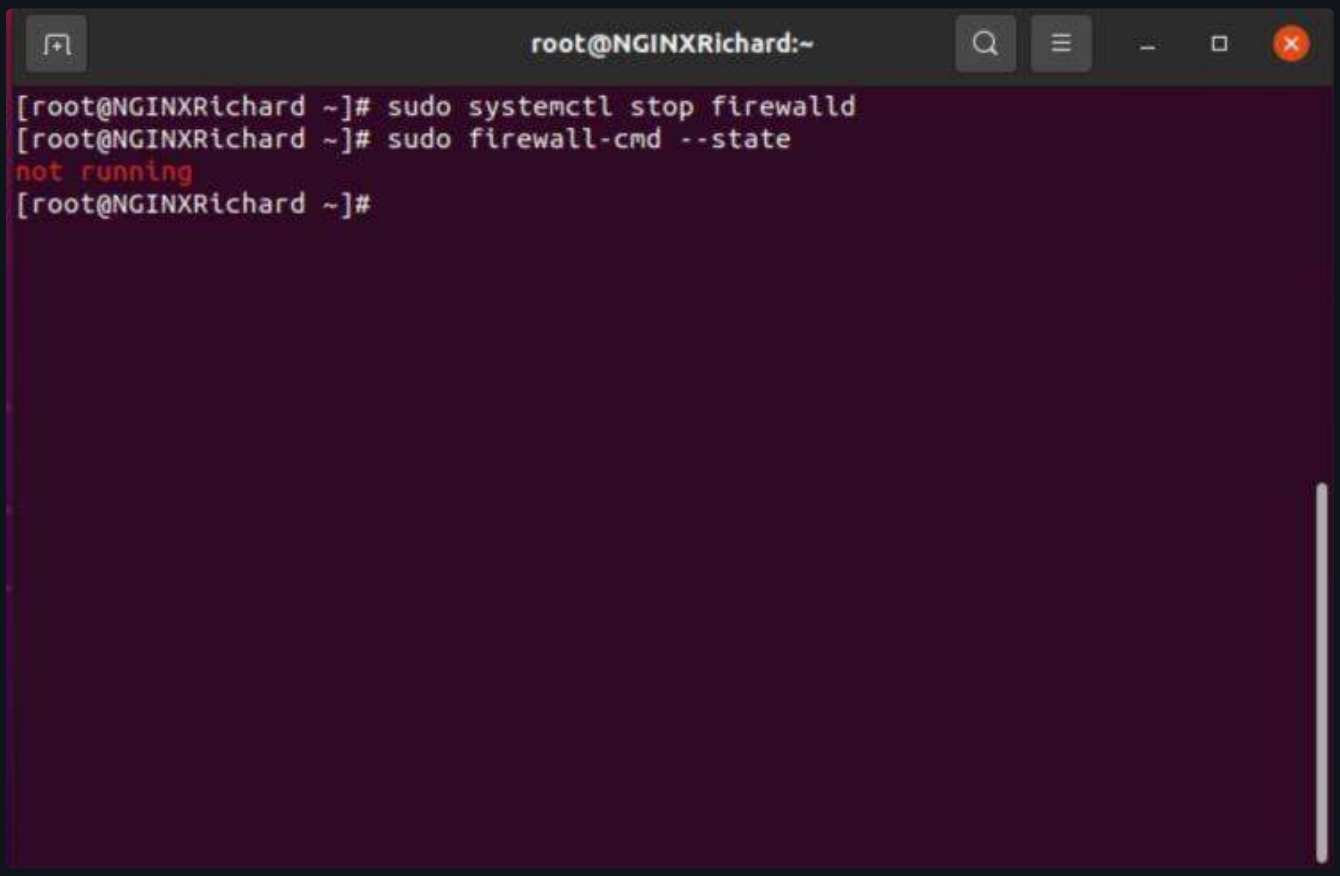

FIG 27 — Stopping the firewall with sudo systemctl stop firewalld and checking sudo firewall-cmd --state to clear ports 80/443 for Nginx.

PROXY HOST

Disable firewalld and Install Nginx

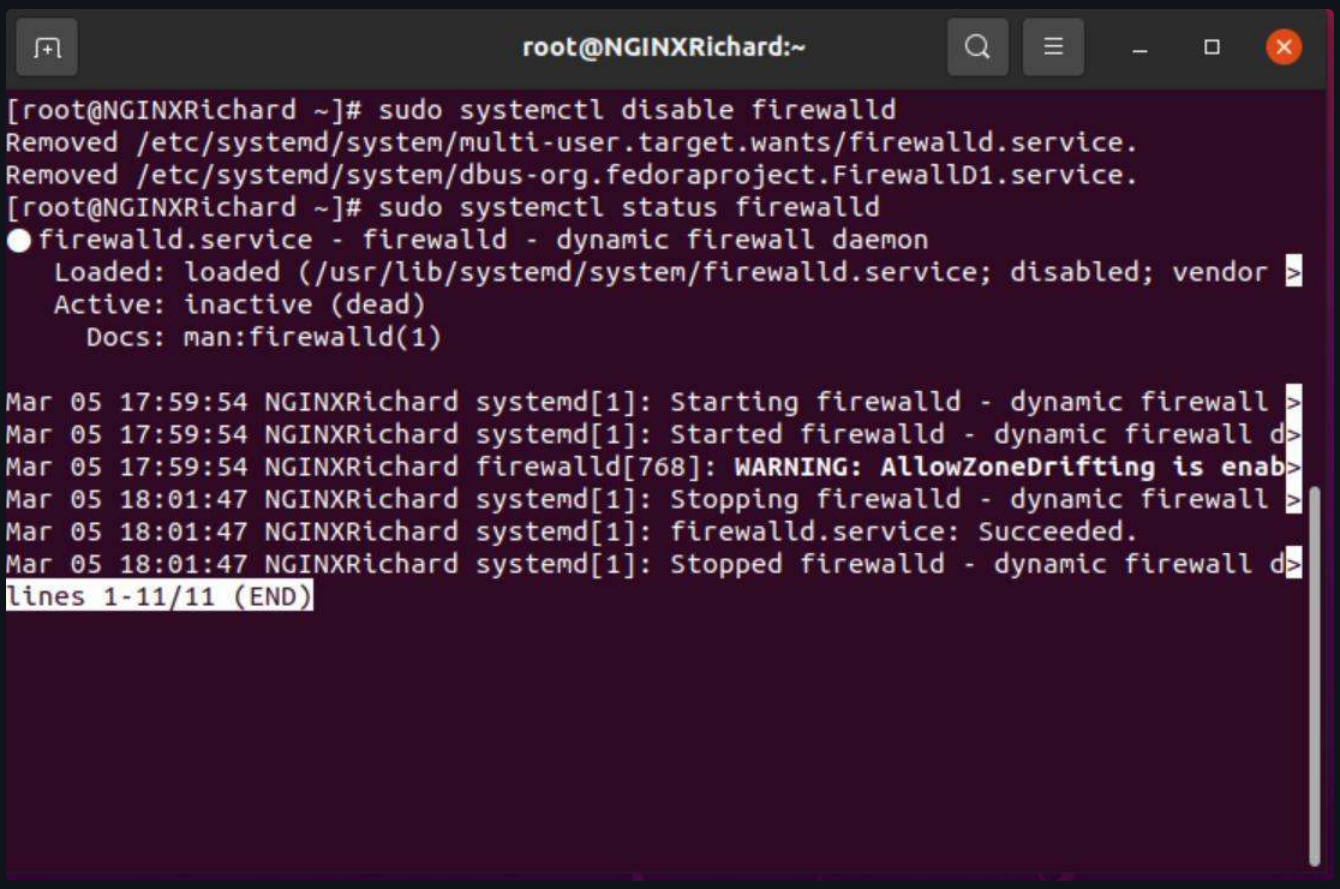

FIG 28 — Disabling the firewall at boot with sudo systemctl disable firewalld so it cannot block reverse-proxy traffic.

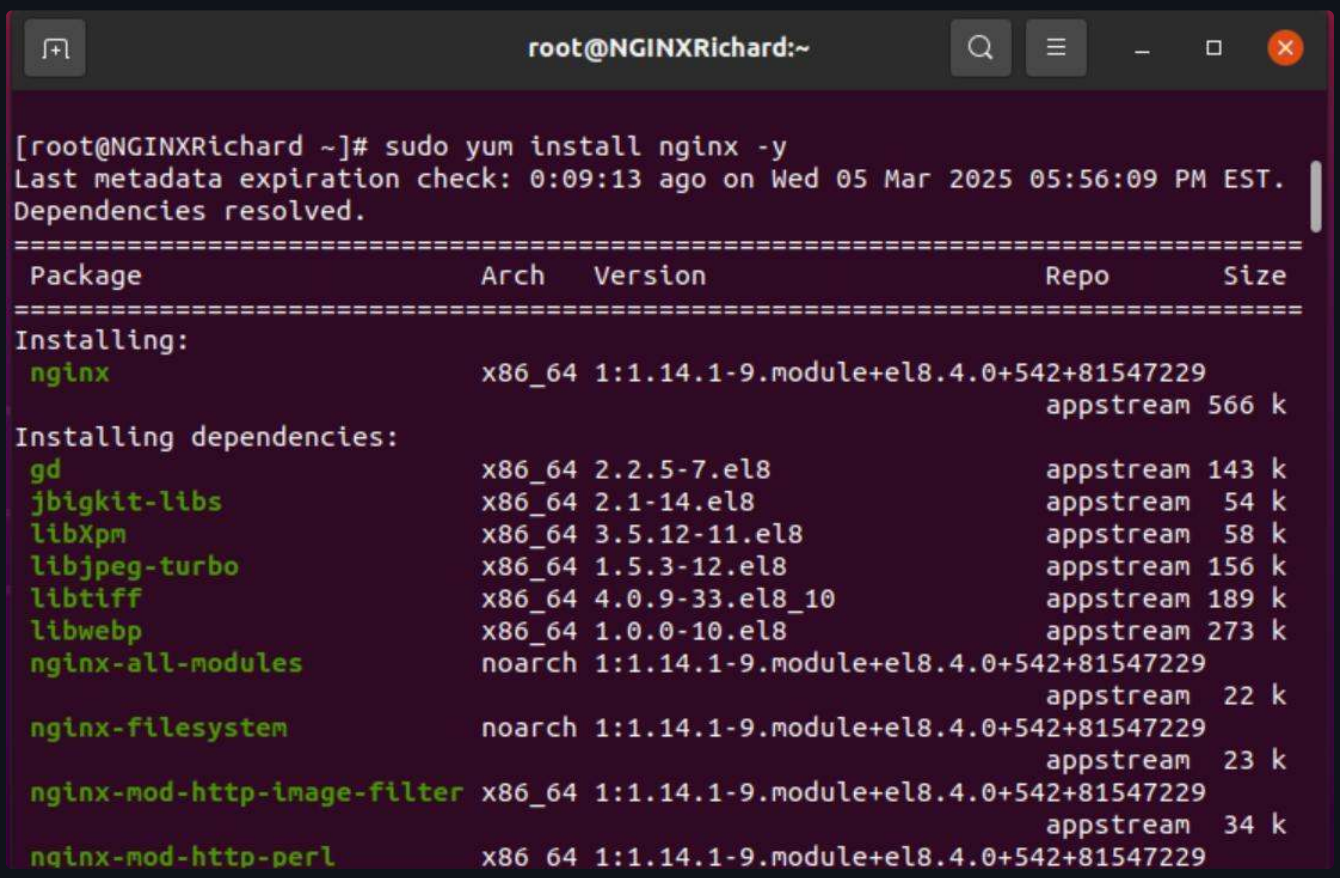

FIG 29 — Installing the web server with sudo yum install nginx -y.

PROXY HOST

Start and Enable Nginx

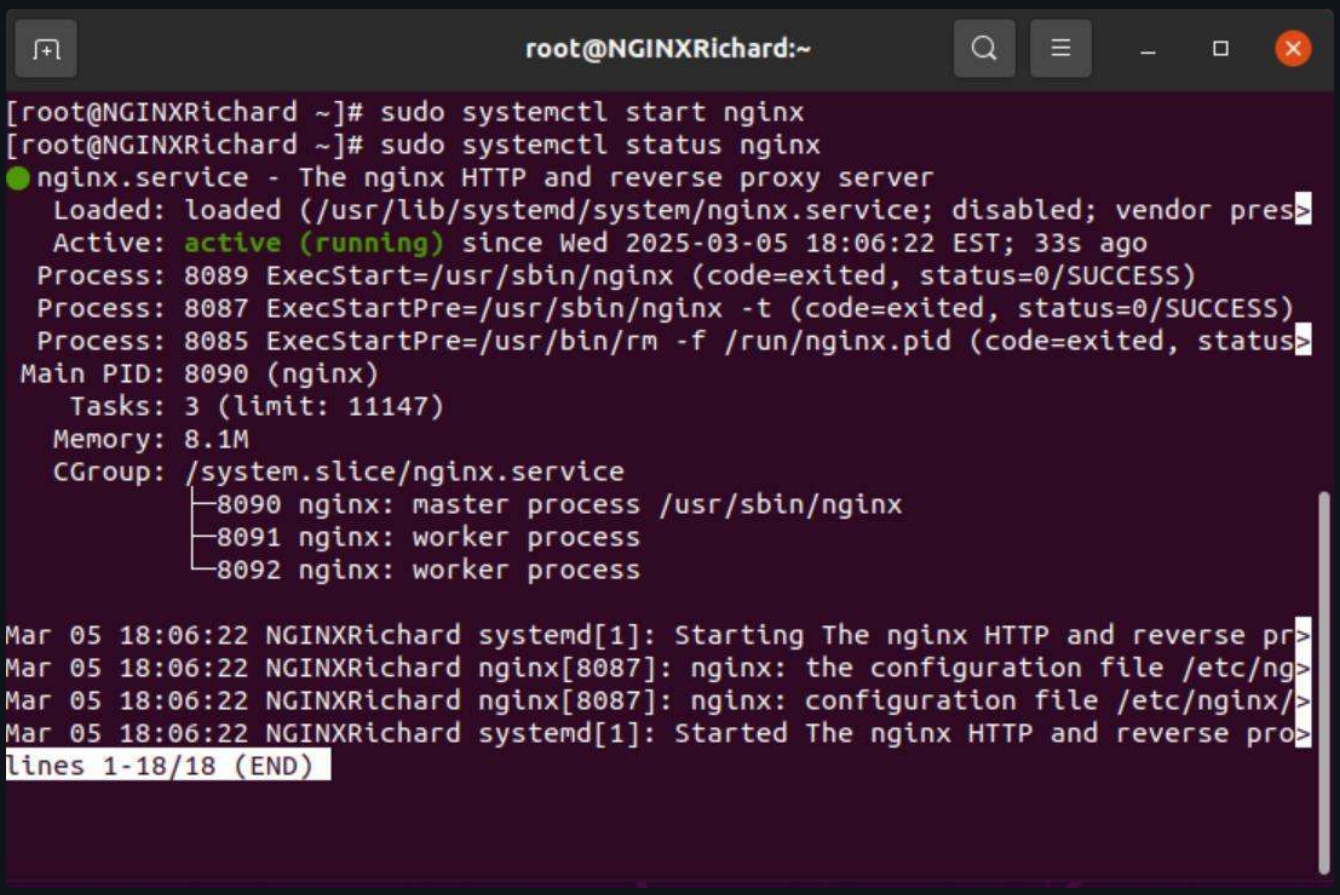

FIG 30 — sudo systemctl start nginx and sudo systemctl status nginx confirming the service is active and operating properly.

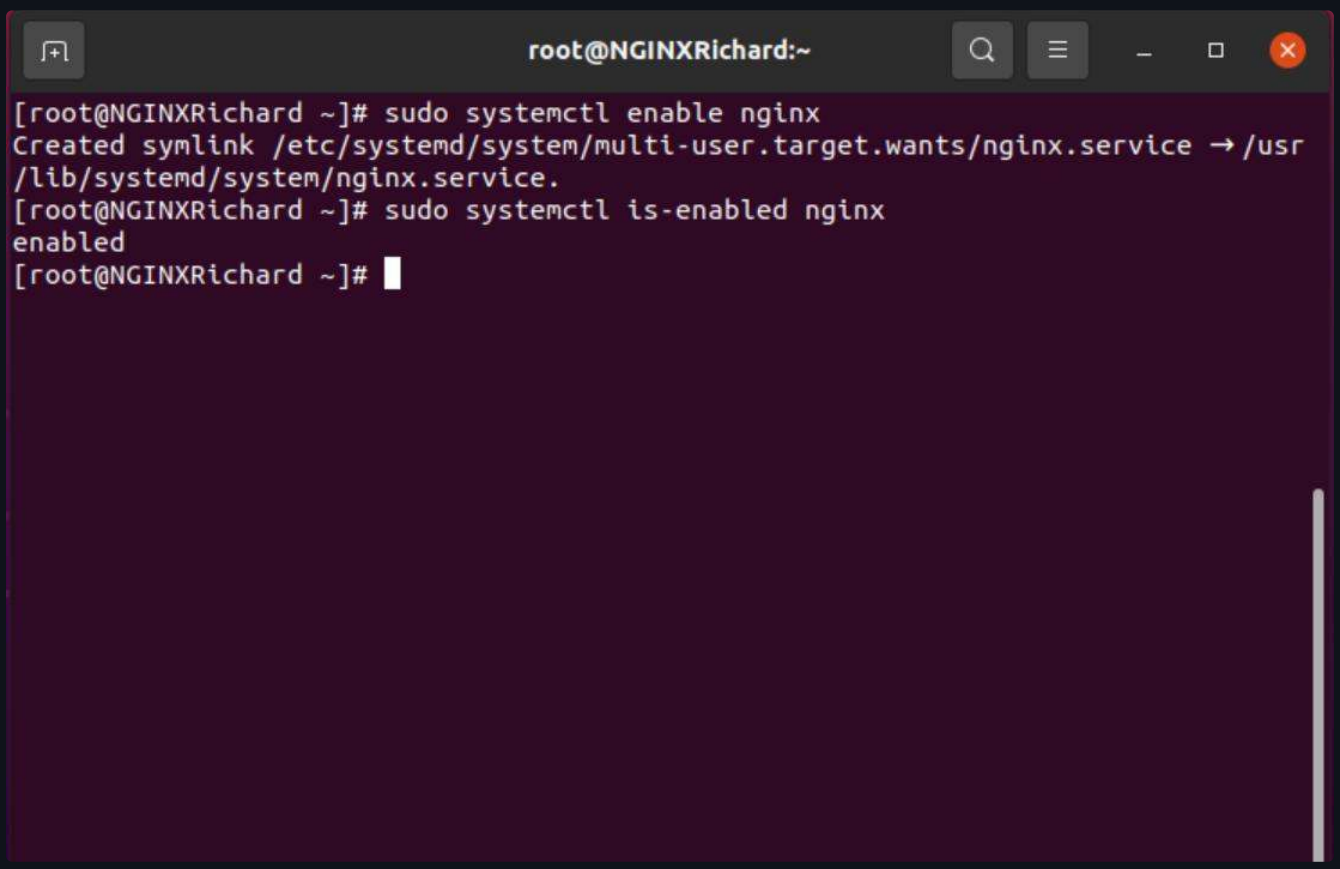

FIG 31 — sudo systemctl enable nginx and sudo systemctl is-enabled nginx confirming Nginx is set to start at boot.

REVERSE PROXY

Confirm Status and Edit nginx.conf

Confirmed the service, then edited the Nginx config to forward /blog traffic to the Ghost backend.

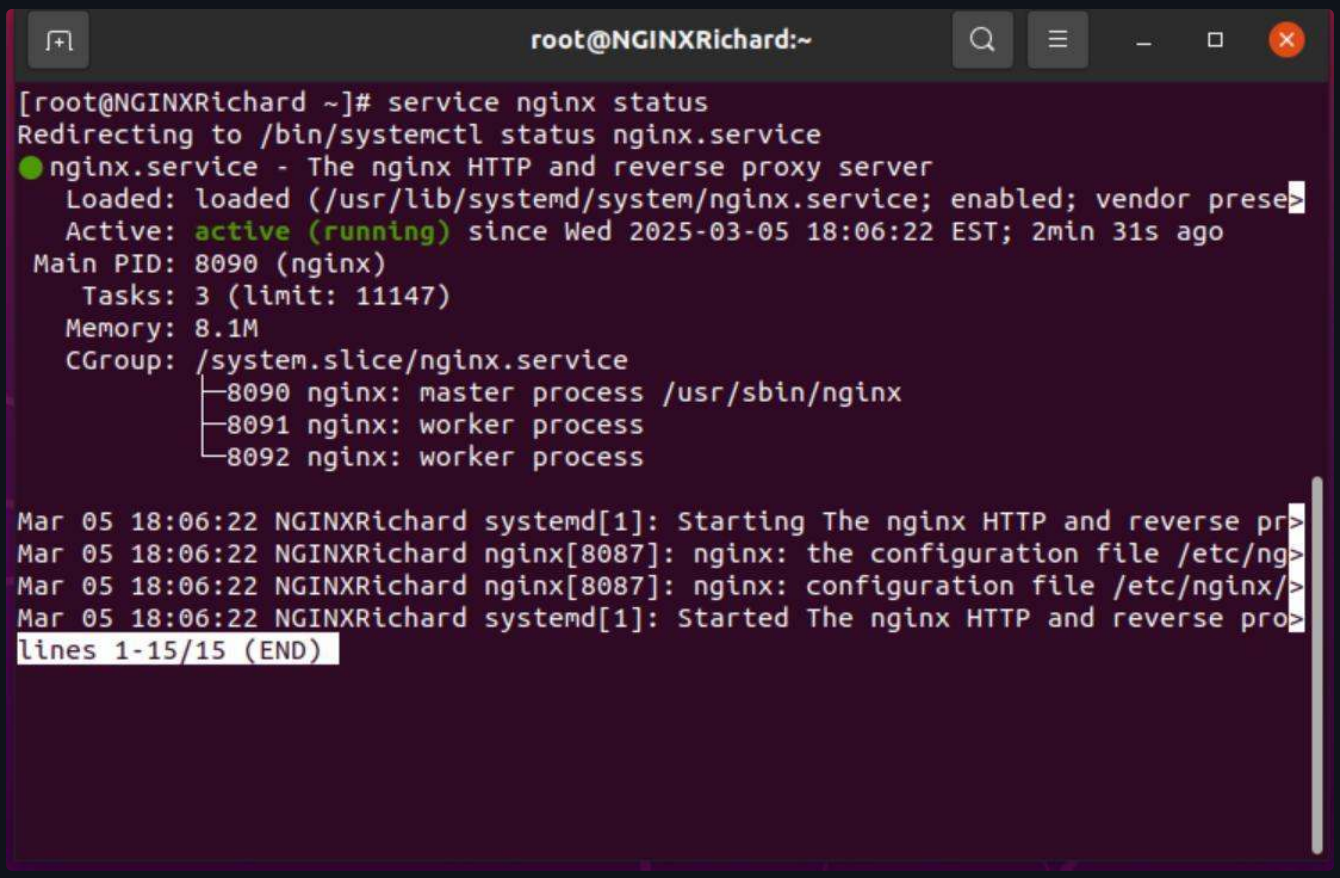

FIG 32 — service nginx status confirming Nginx is active and working properly.

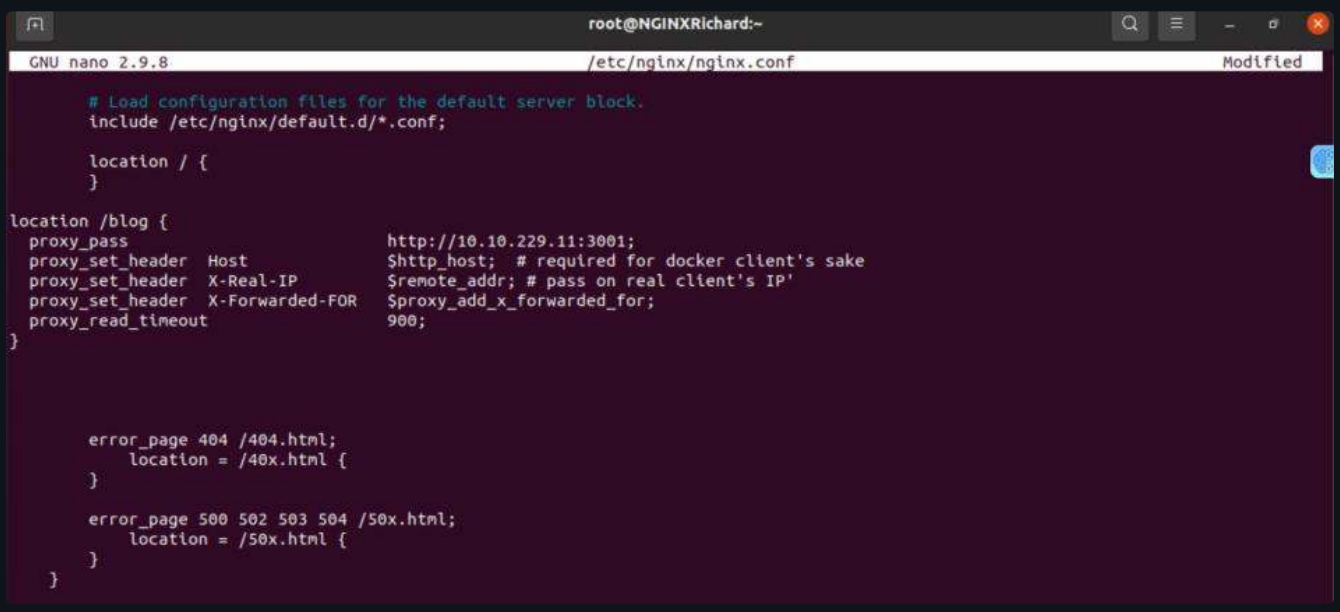

FIG 33 — Editing /etc/nginx/nginx.conf with nano to configure Nginx as a reverse proxy for the Ghost site.

REVERSE PROXY

Reverse-Proxy Block and Reload

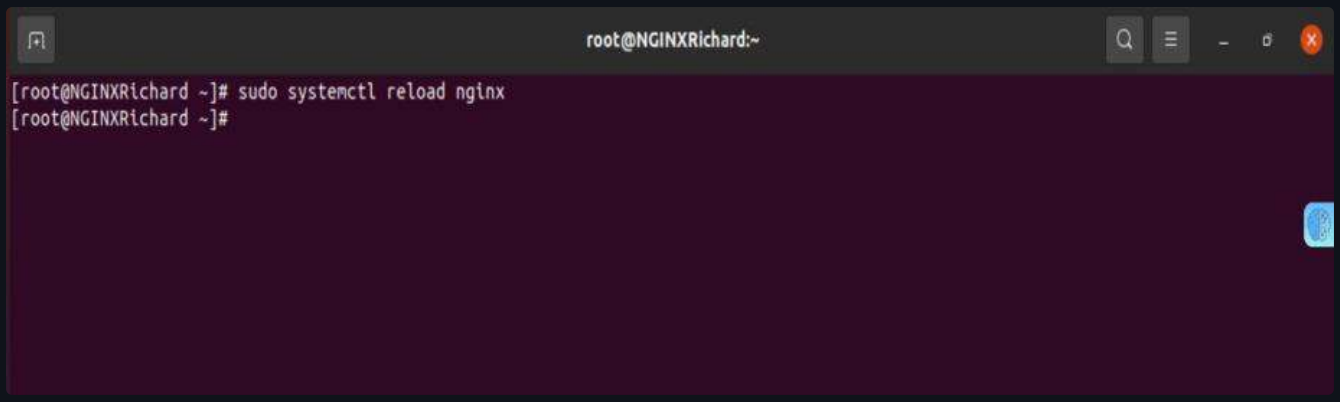

FIG 34 — The reverse-proxy location block in nginx.conf forwarding client requests to the Ghost container backend.

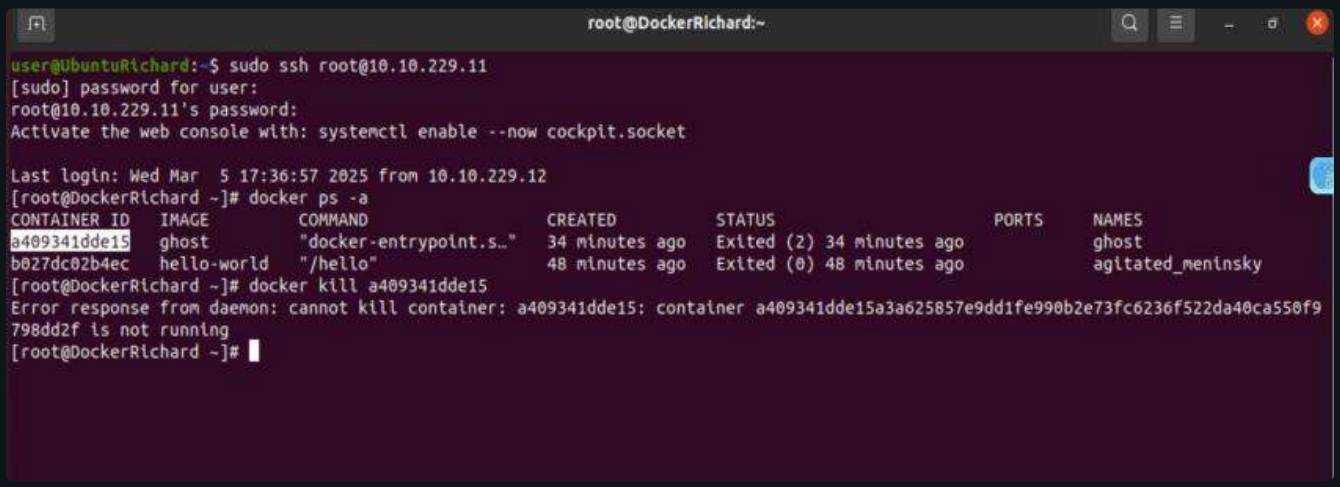

FIG 35 — Applying the new configuration without a restart via sudo systemctl reload nginx.

REVERSE PROXY

Clear the Stale Ghost Container

Cleared the first, misconfigured Ghost container before rebuilding it against the proxy URL.

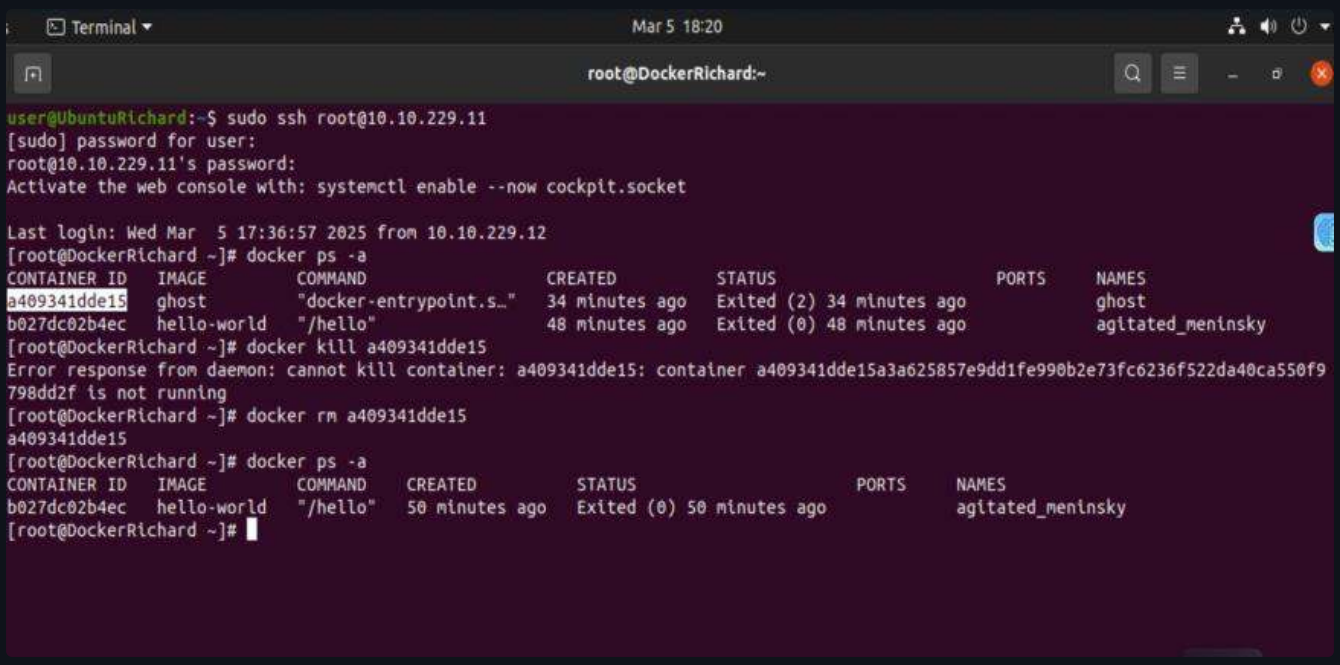

FIG 36 — docker ps -a showing the Ghost container inactive; docker kill on its ID returned an error because it was already stopped.

FIG 37 — Removing the stale container with docker rm <container ID> to clear the earlier misconfiguration.

REVERSE PROXY

Recreate Ghost and Verify Through the Proxy

Recreated the container pointed at the proxy URL, then verified in Firefox that requests reached the reverse proxy.

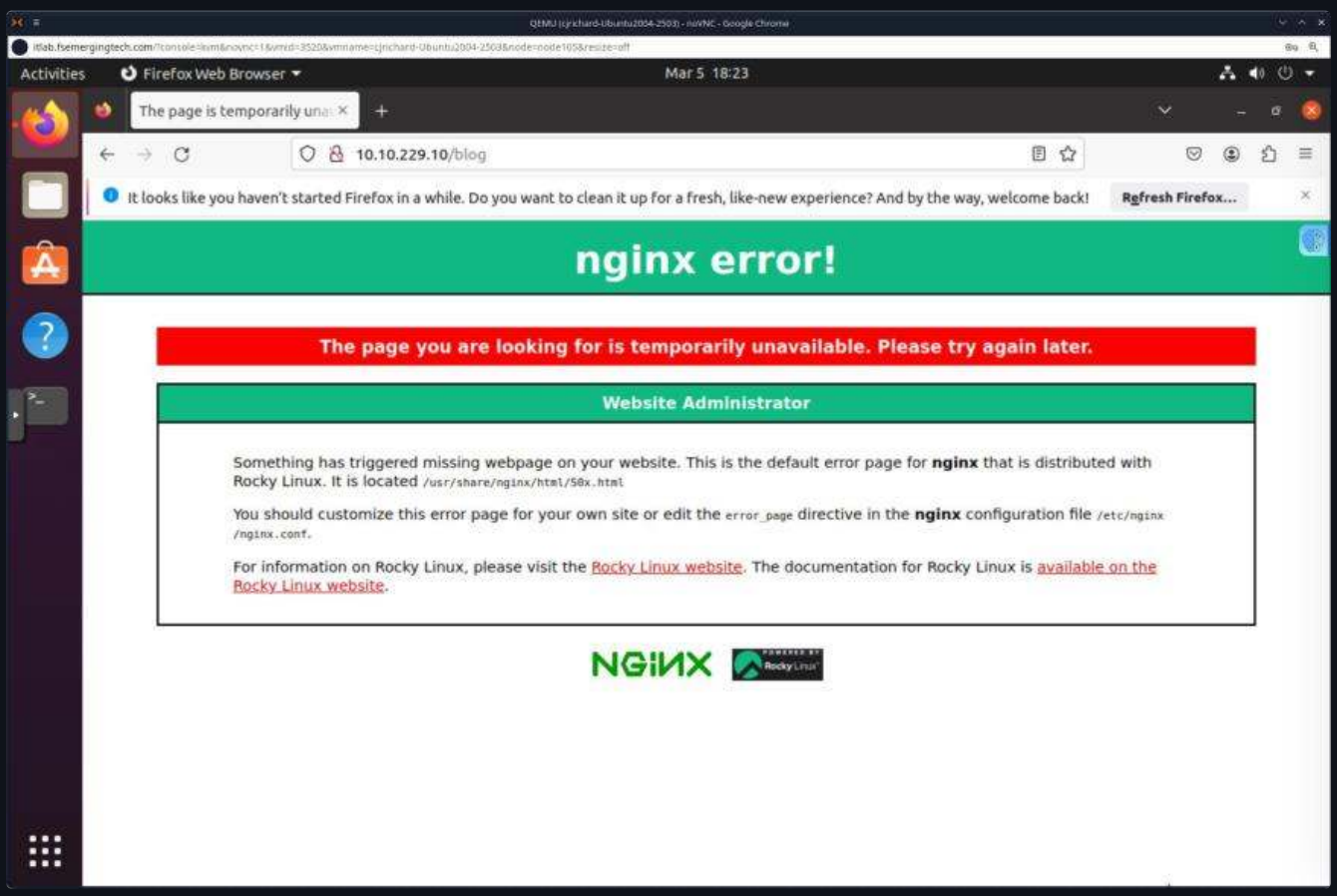

FIG 38 — Browsing http://10.10.229.10/blog in Firefox after recreating Ghost with -e url=http://10.10.229.10/blog; the response confirms requests are reaching the Nginx reverse proxy and routing toward the Ghost backend.

LAMP STACK

Set Hostname and Update Ubuntu

Moved to the Ubuntu LAMP host: renamed to UbuntuXRichard and pulled the latest package lists.

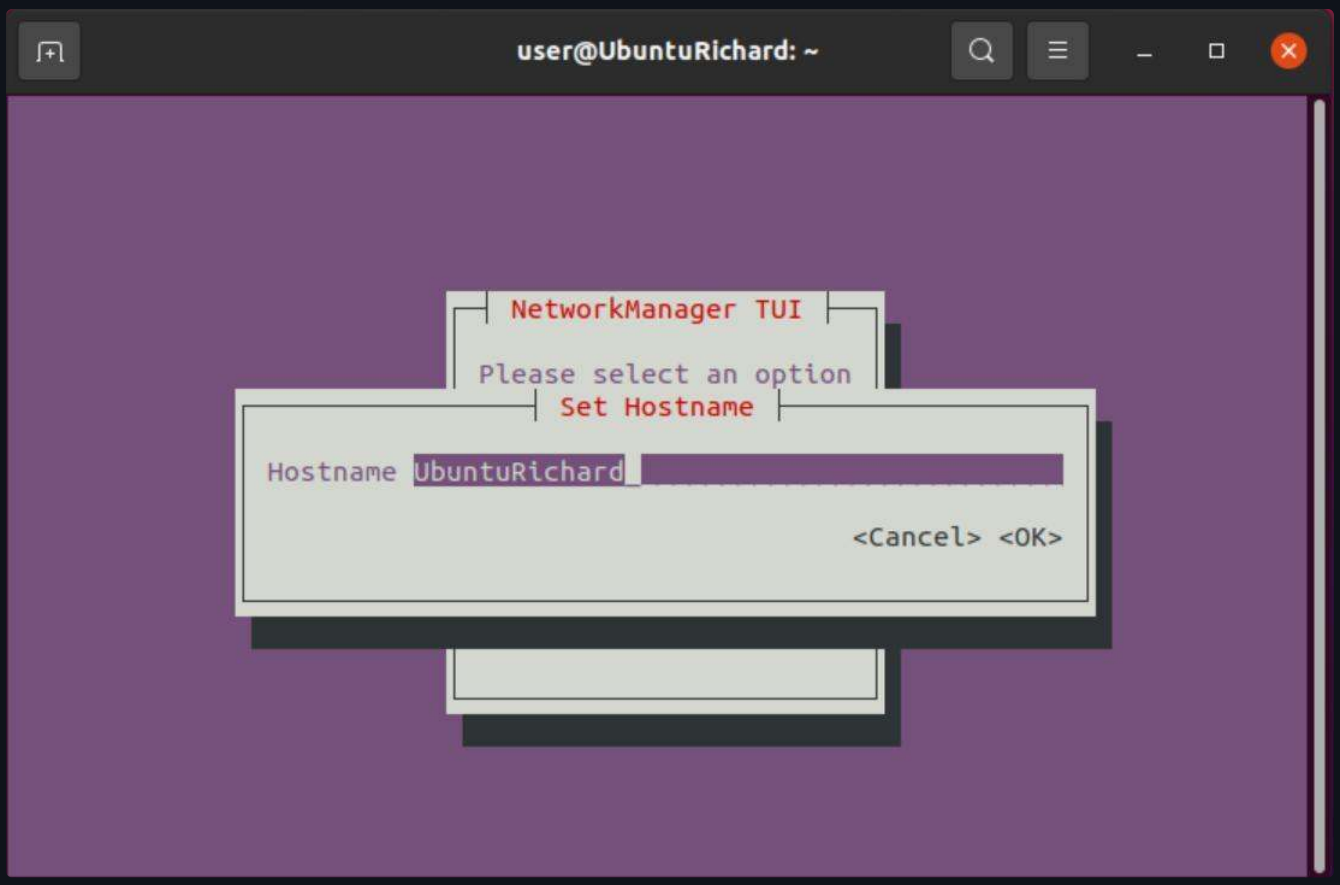

FIG 39 — Ubuntu host renamed to UbuntuXRichard via nmtui for clear identification across the VM fleet.

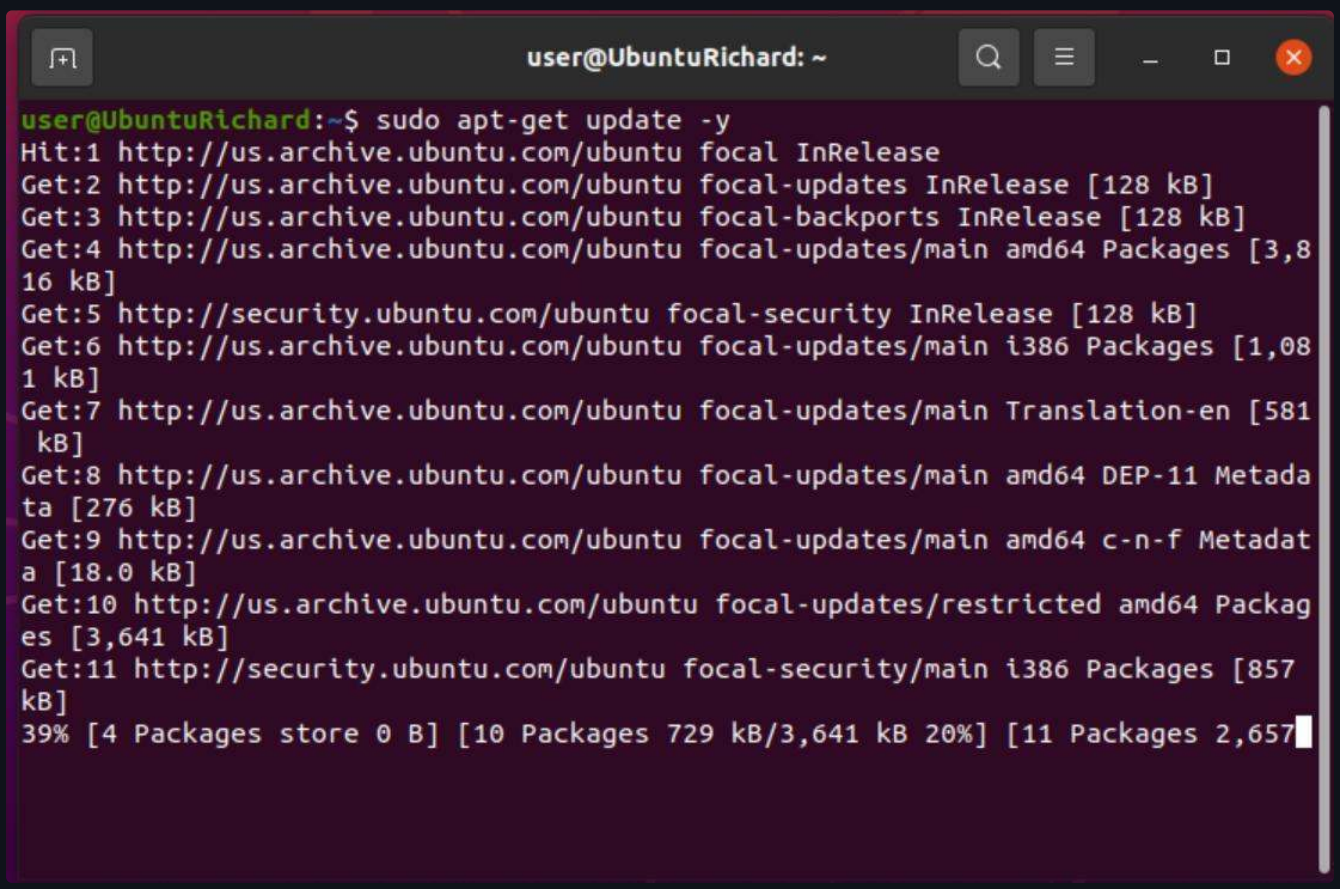

FIG 40 — Refreshing package lists with sudo apt-get update -y on the Ubuntu LAMP host.

LAMP STACK

Upgrade Packages and Install Nano

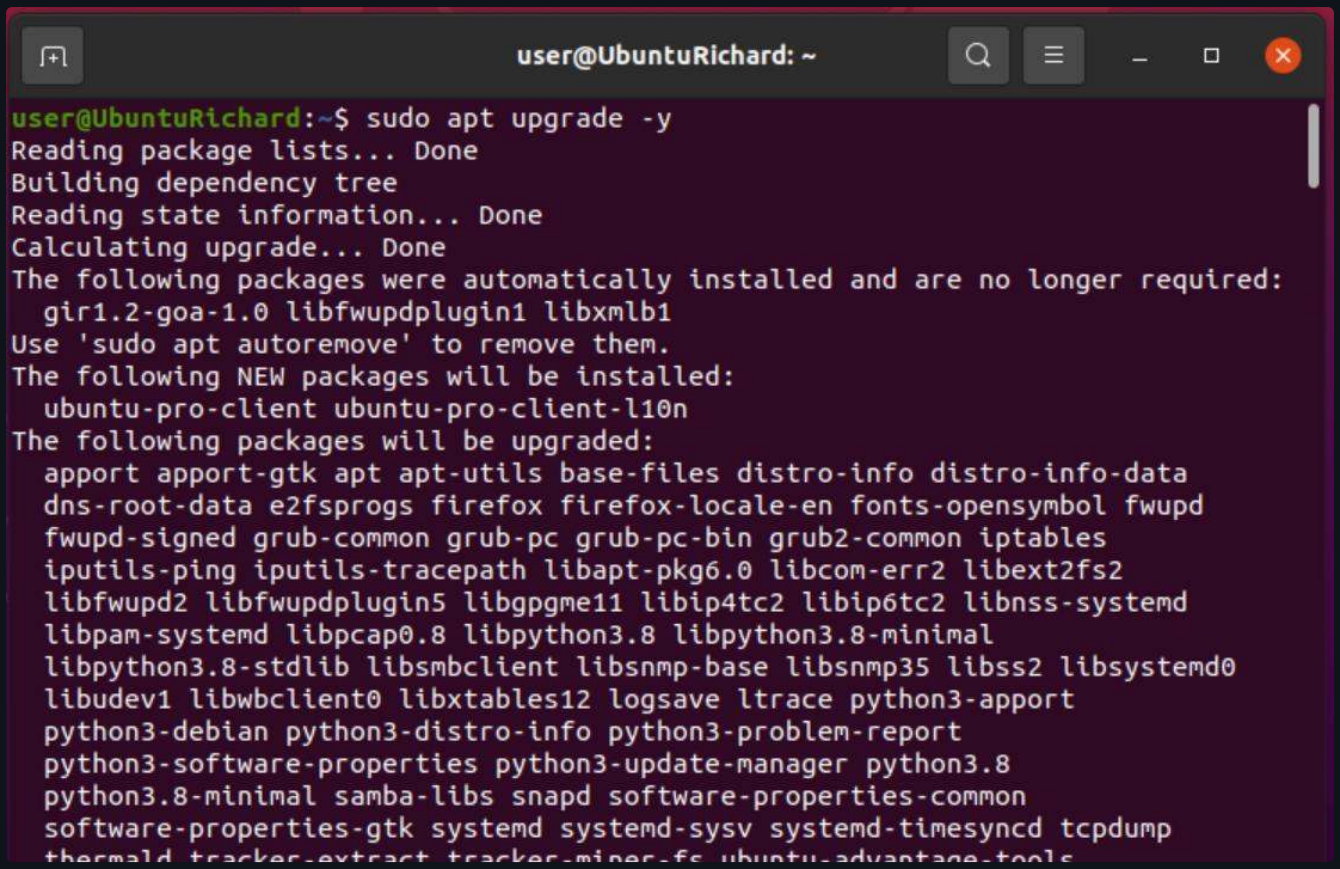

FIG 41 — Upgrading installed packages with sudo apt upgrade -y for system security and stability.

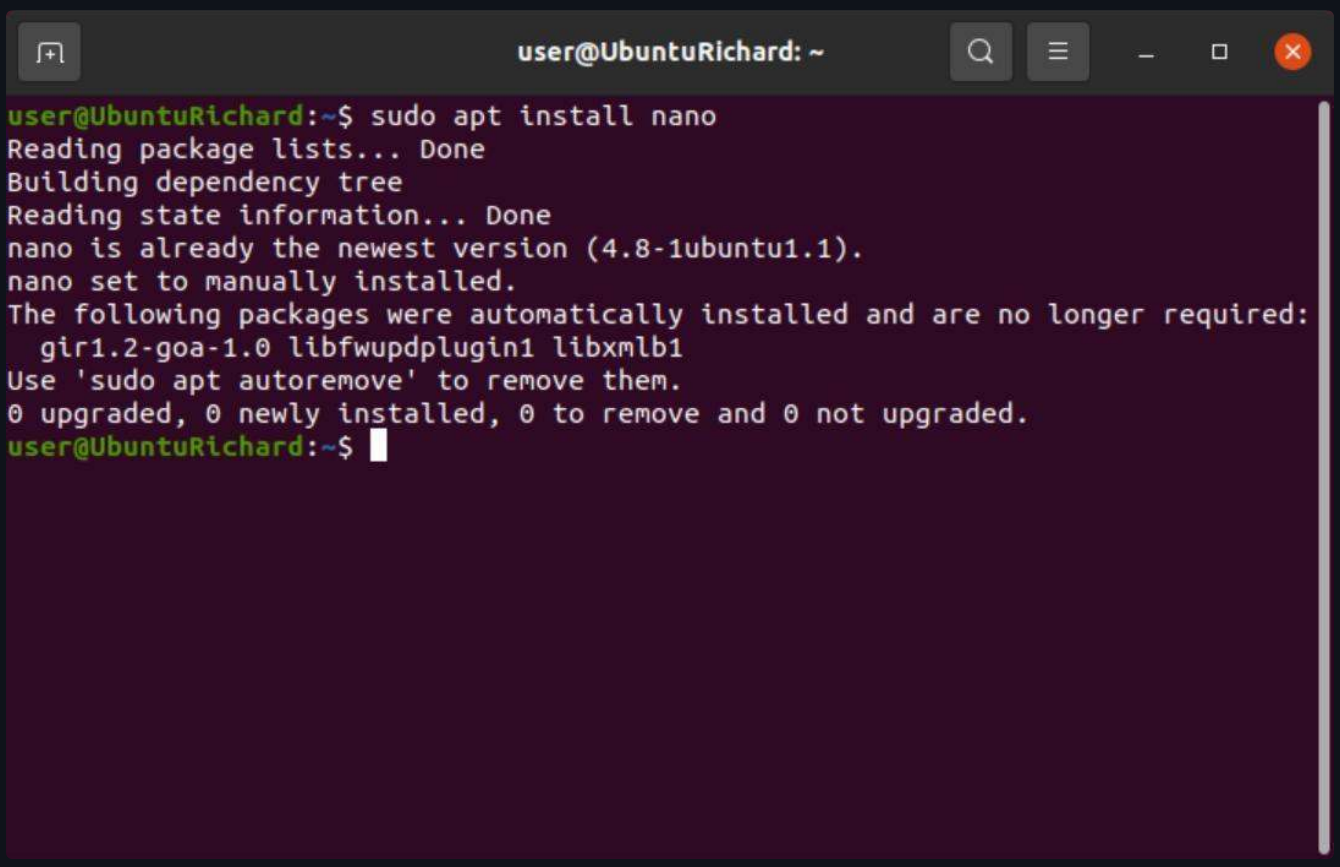

FIG 42 — Installing the nano editor with sudo apt install nano.

LAMP STACK

Install Git and Apache2

Installed Git (to clone WordPress later) and the Apache2 web server.

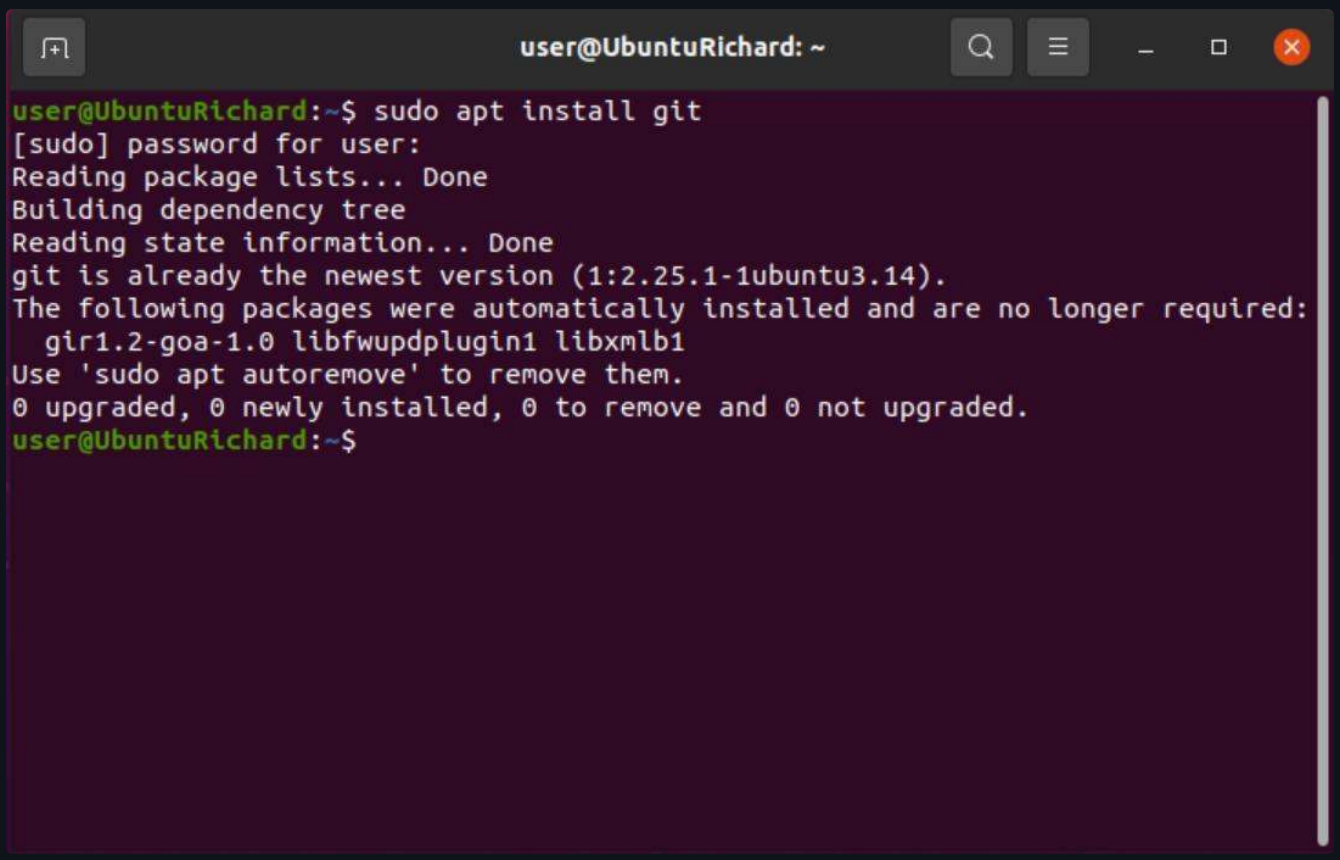

FIG 43 — Installing Git with sudo apt install git to enable cloning WordPress from GitHub.

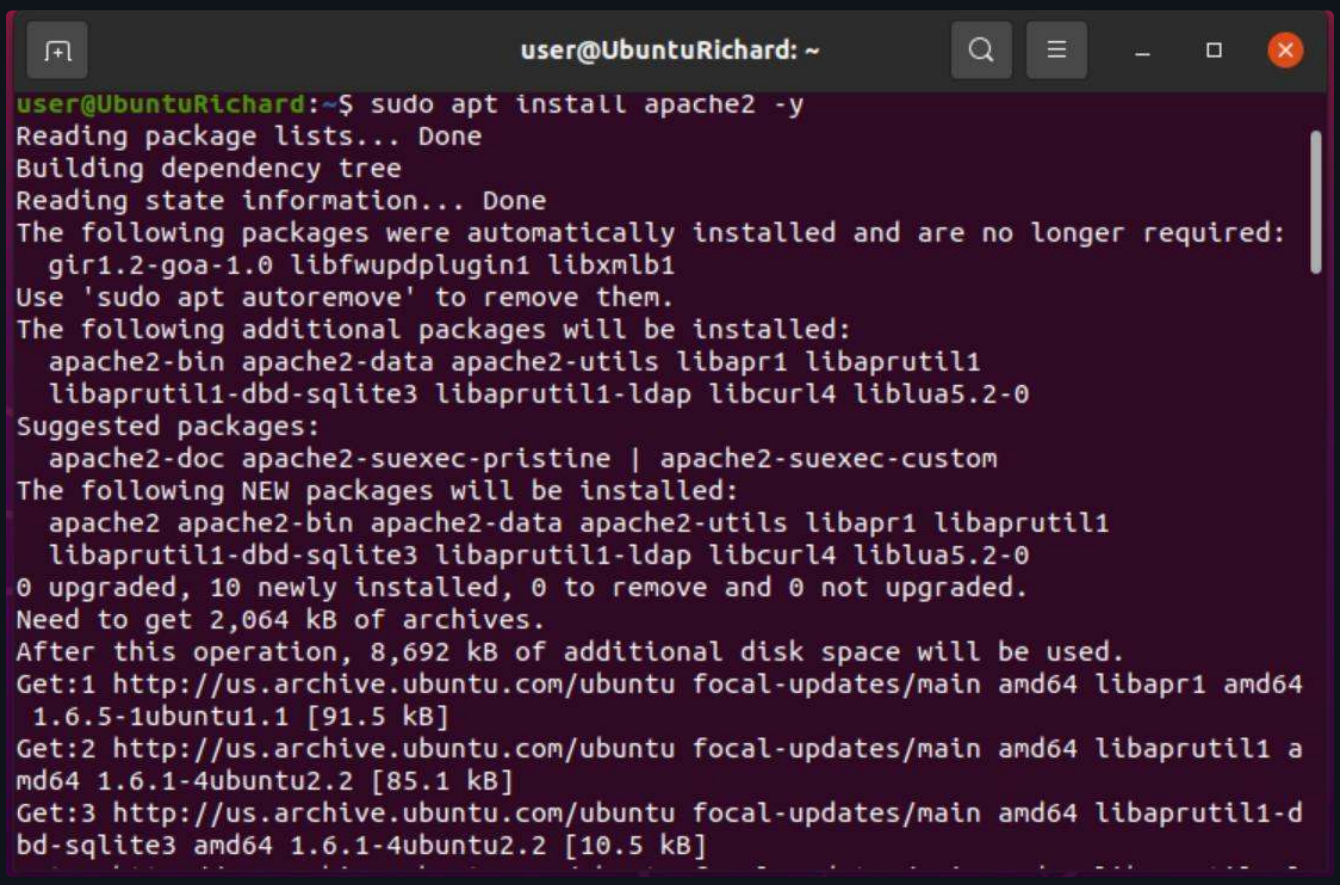

FIG 44 — Installing the web server with sudo apt install apache2 to host the WordPress site.

LAMP STACK

Open Firewall Ports and Verify Apache

Allowed Apache through UFW on ports 80/443 and confirmed the default Apache page over the network.

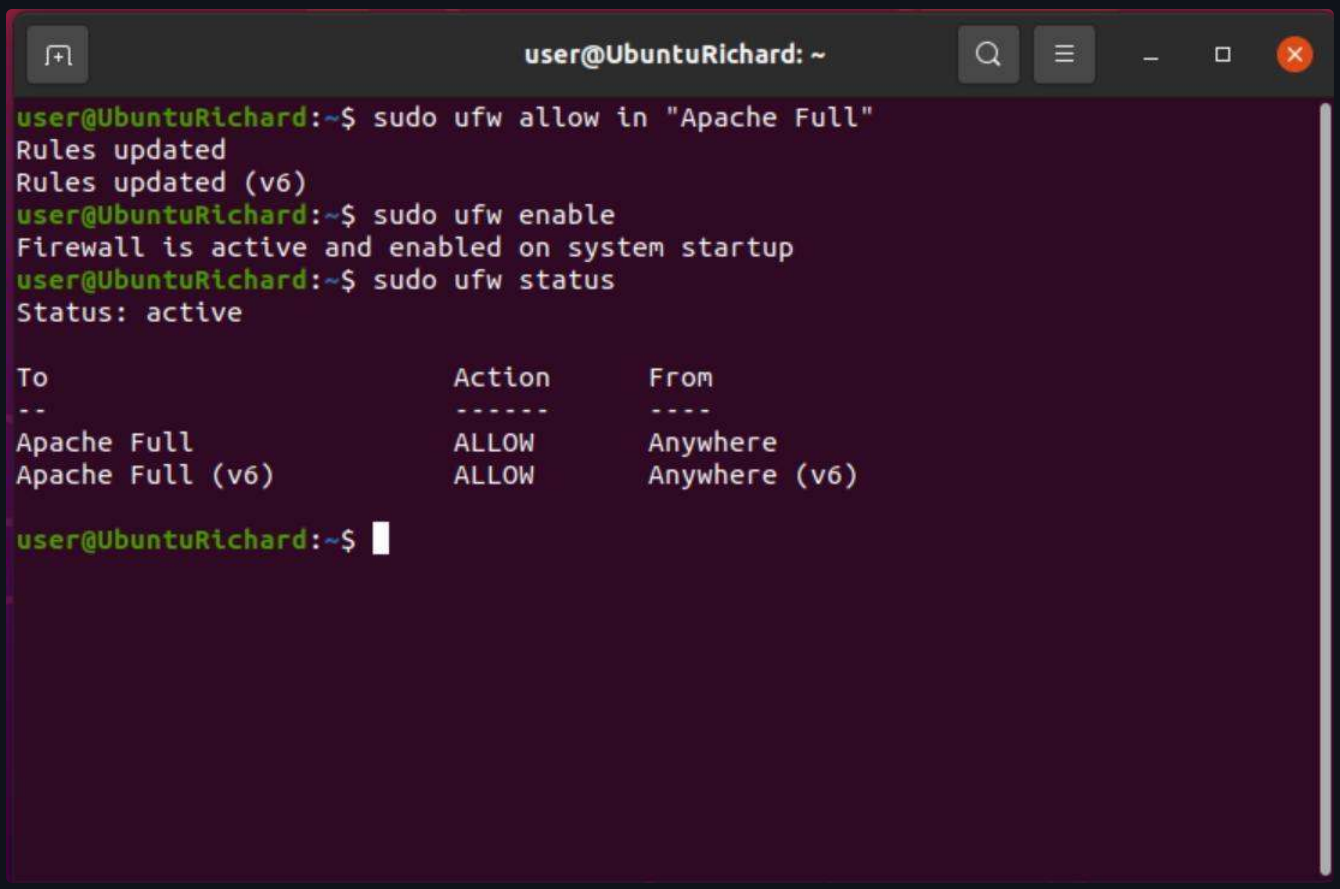

FIG 45 — sudo ufw allow in 'Apache Full', sudo ufw enable, and sudo ufw status opening HTTP/HTTPS ports 80 and 443 and confirming the rules are active.

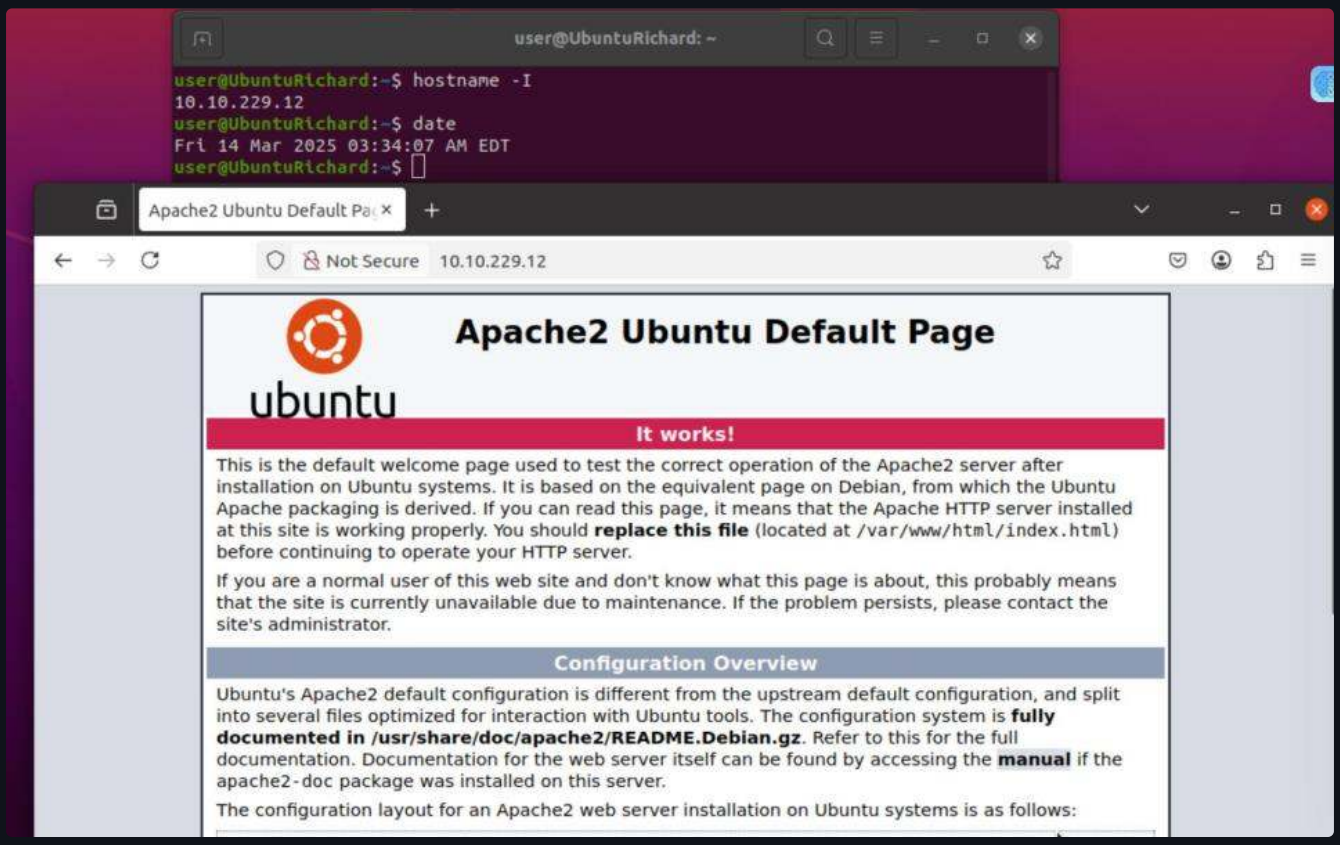

FIG 46 — The default Apache2 Ubuntu page served at http://10.10.229.12, confirming the web server is reachable.

LAMP STACK

Install MySQL and Secure root

Installed the MySQL server and set the root password using native-password authentication.

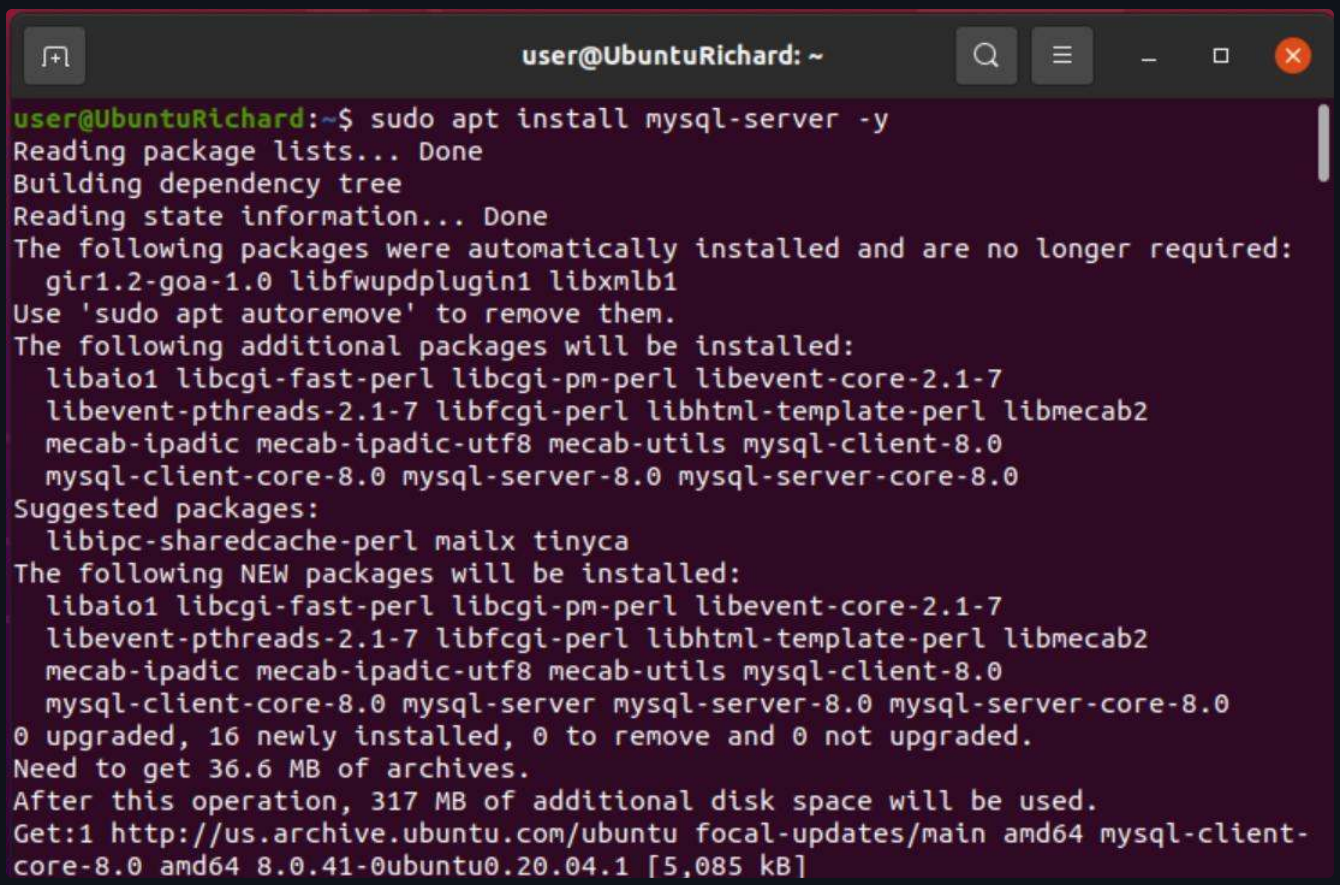

FIG 47 — Installing the database server with sudo apt install mysql-server -y.

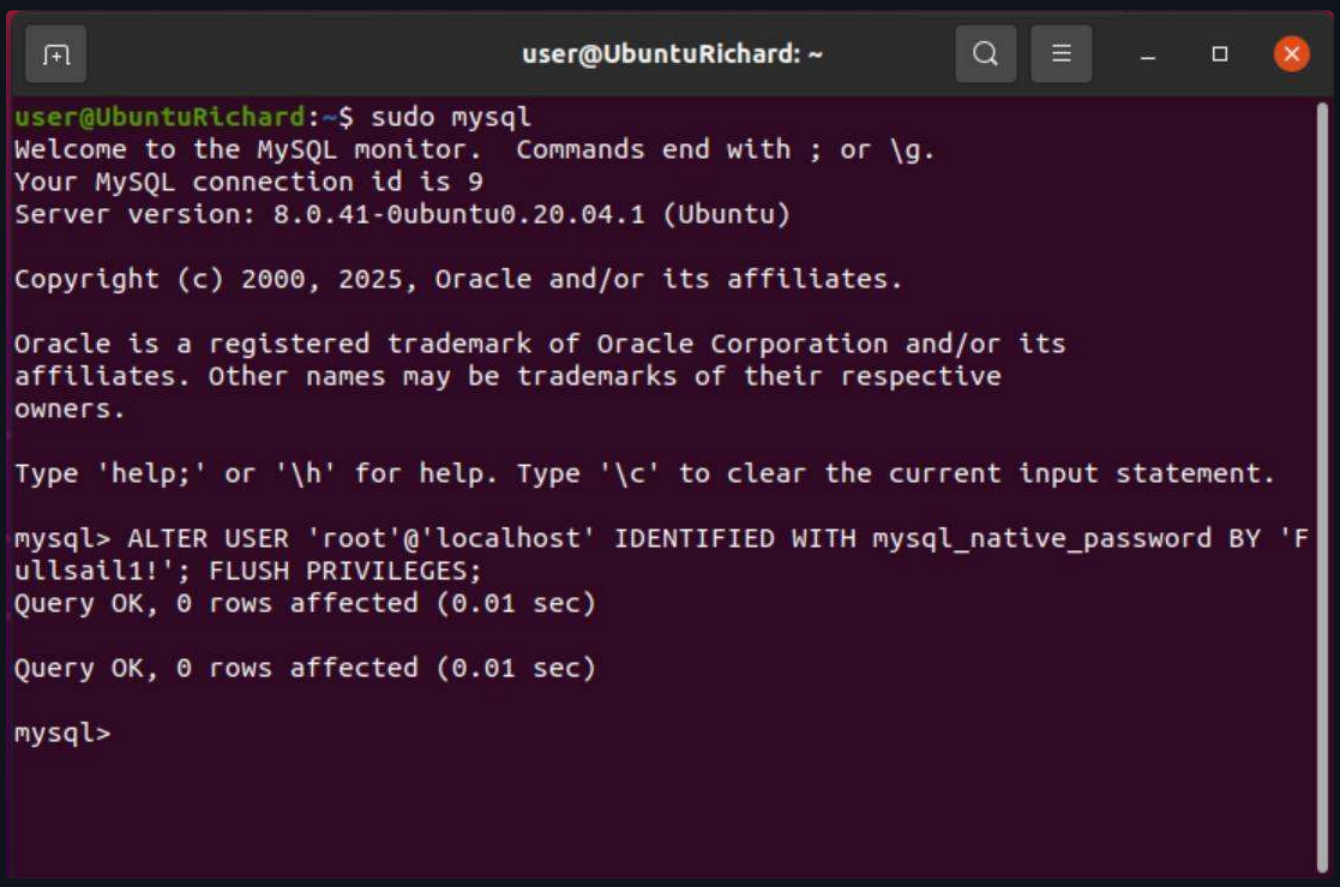

FIG 48 — ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'Fullsail1!'; FLUSH PRIVILEGES; securing the MySQL root account and applying the change.

LAMP STACK

Exit MySQL and Install PHP

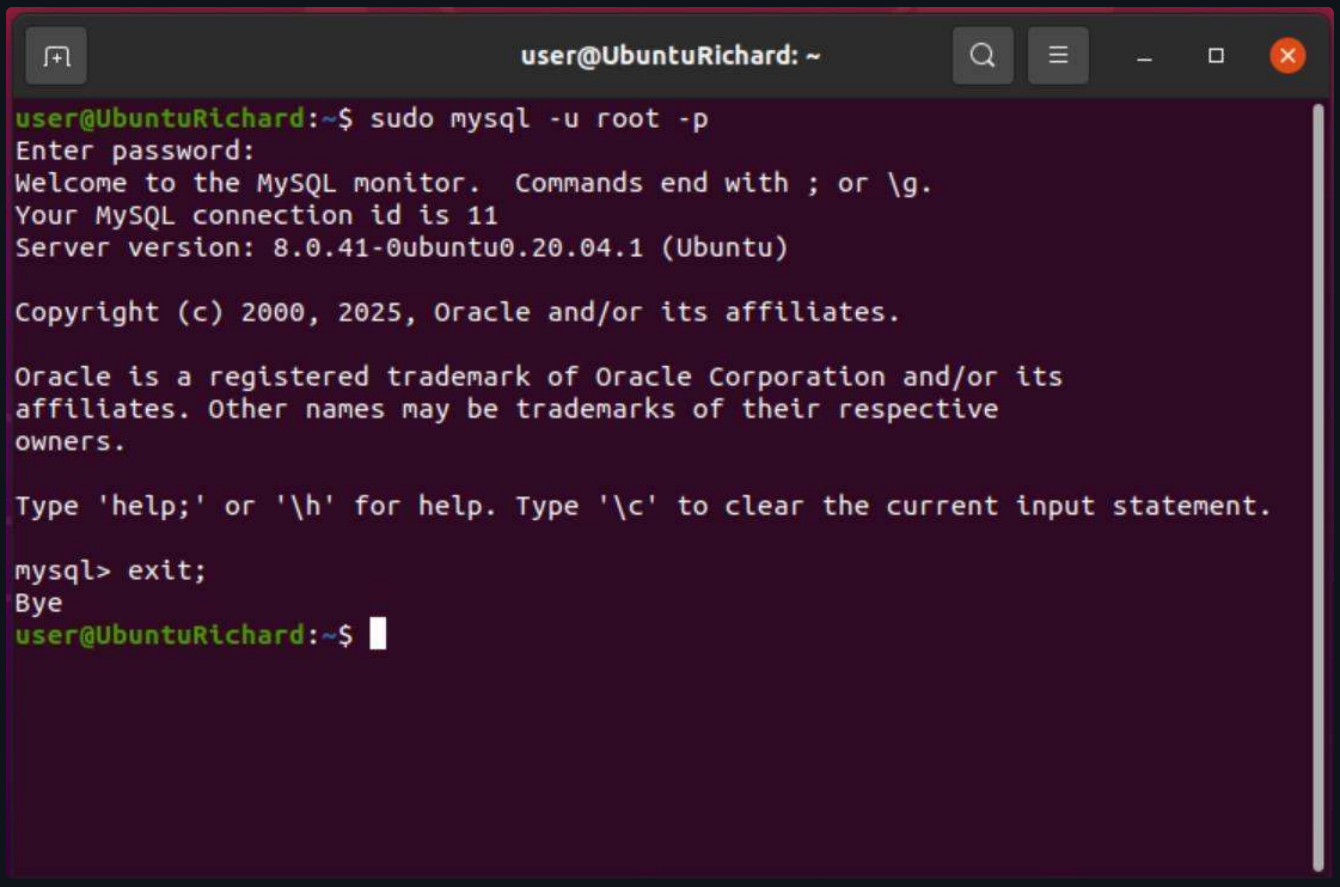

FIG 49 — Leaving the MySQL shell with EXIT; after setting the root password.

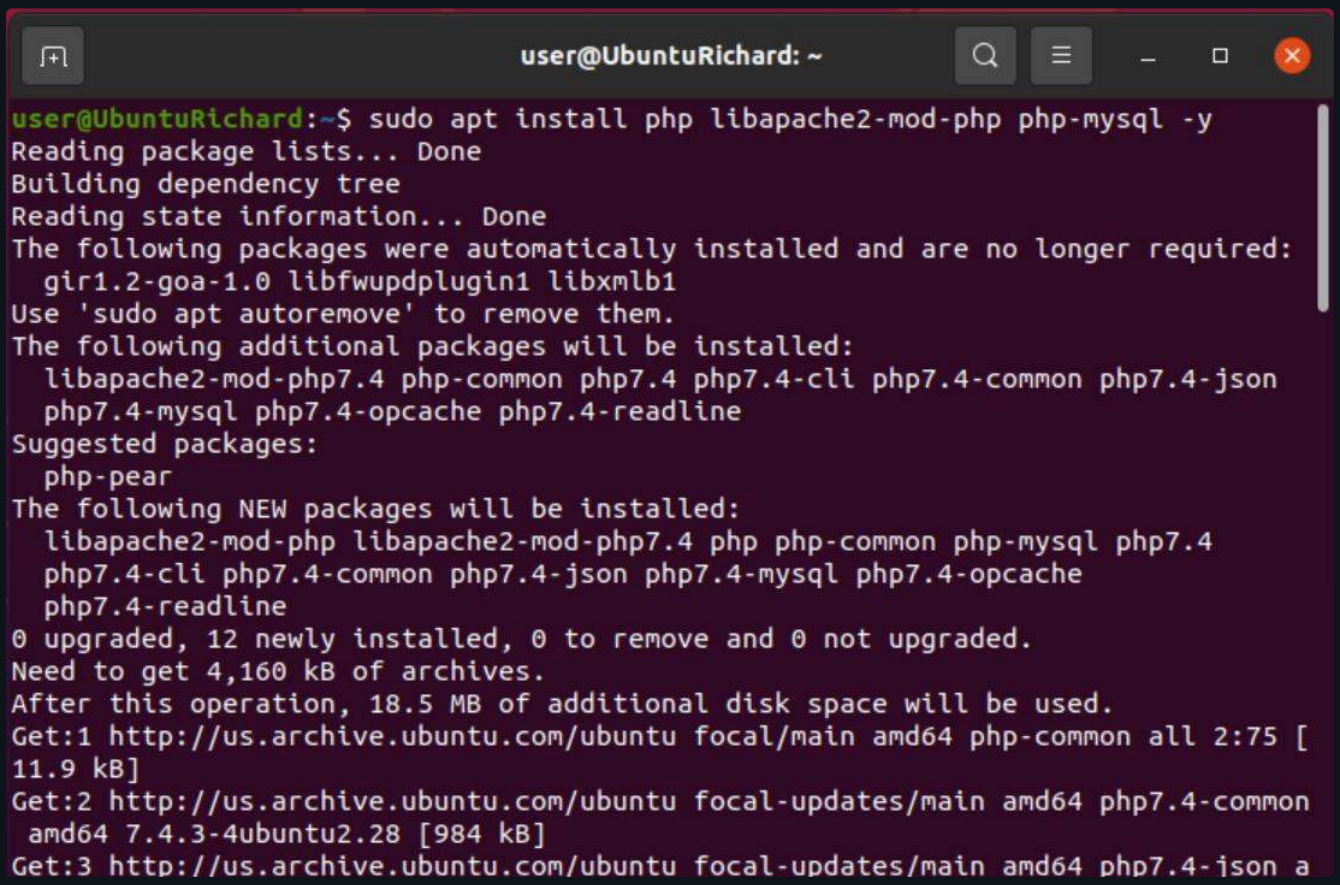

FIG 50 — Installing PHP and the Apache/MySQL bridges: sudo apt install php libapache2-mod-php php-mysql -y.

LAMP STACK

MySQL PHP Libraries and mod_rewrite

Confirmed the PHP/MySQL libraries and enabled Apache's rewrite module for clean WordPress URLs.

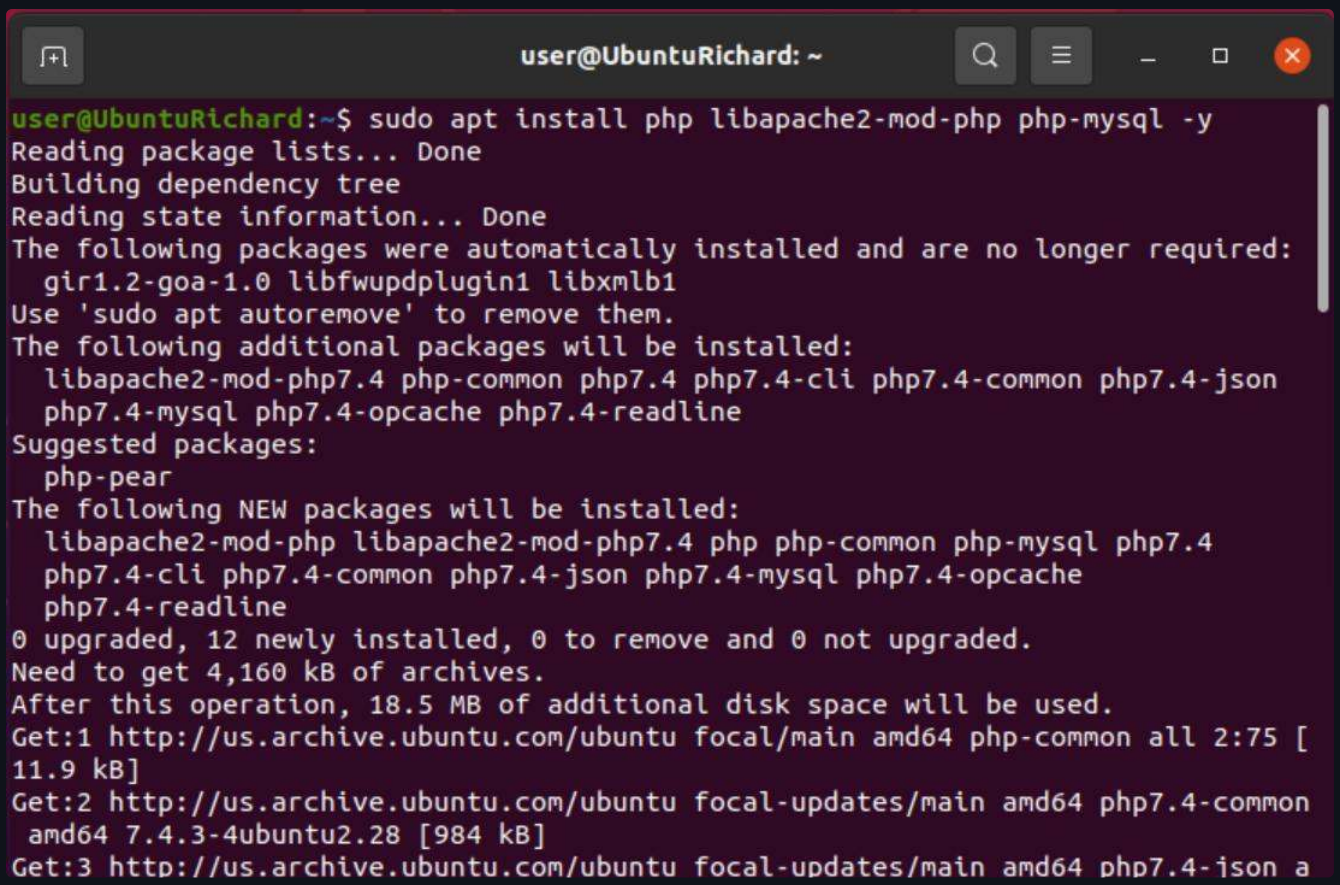

FIG 51 — Ensuring PHP can talk to MySQL via sudo apt install php libapache2-mod-php php-mysql -y for WordPress database queries.

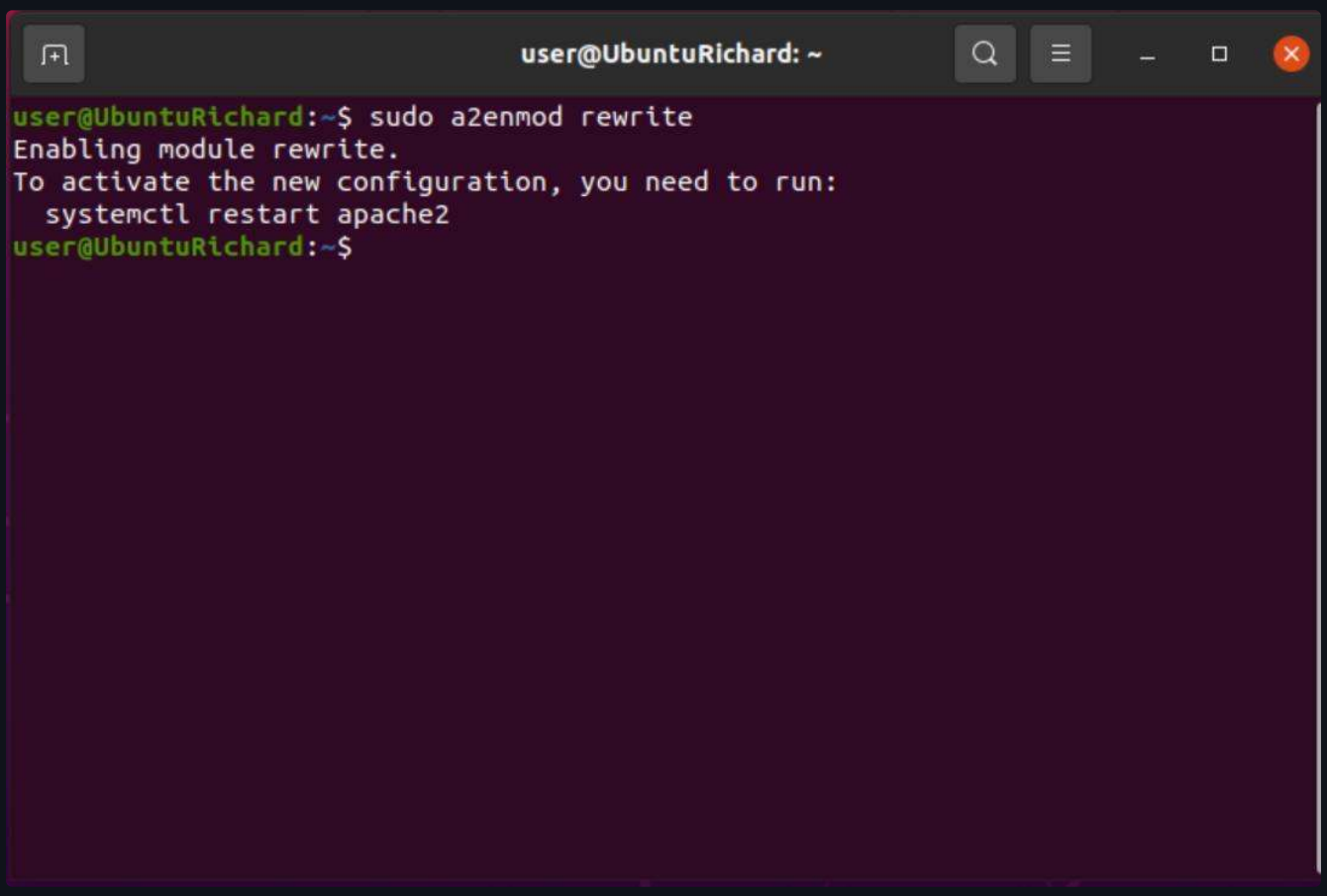

FIG 52 — Enabling clean URLs and permalinks with sudo a2enmod rewrite.

LAMP STACK

Restart Apache and Create test.php

Restarted Apache to load mod_rewrite, then dropped a PHP probe page to confirm execution.

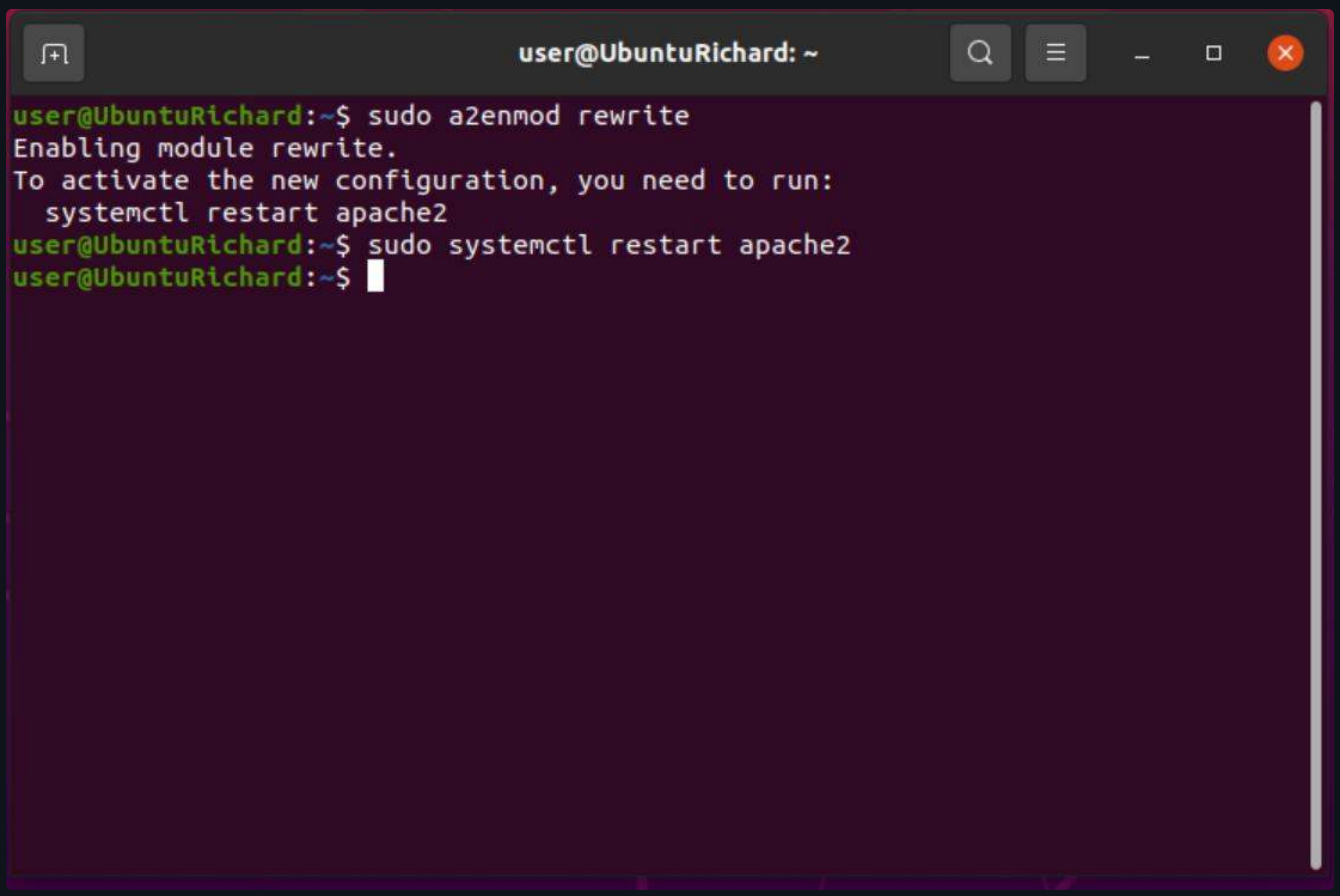

FIG 53 — Restarting the web server with sudo systemctl restart apache2 so the rewrite module takes effect.

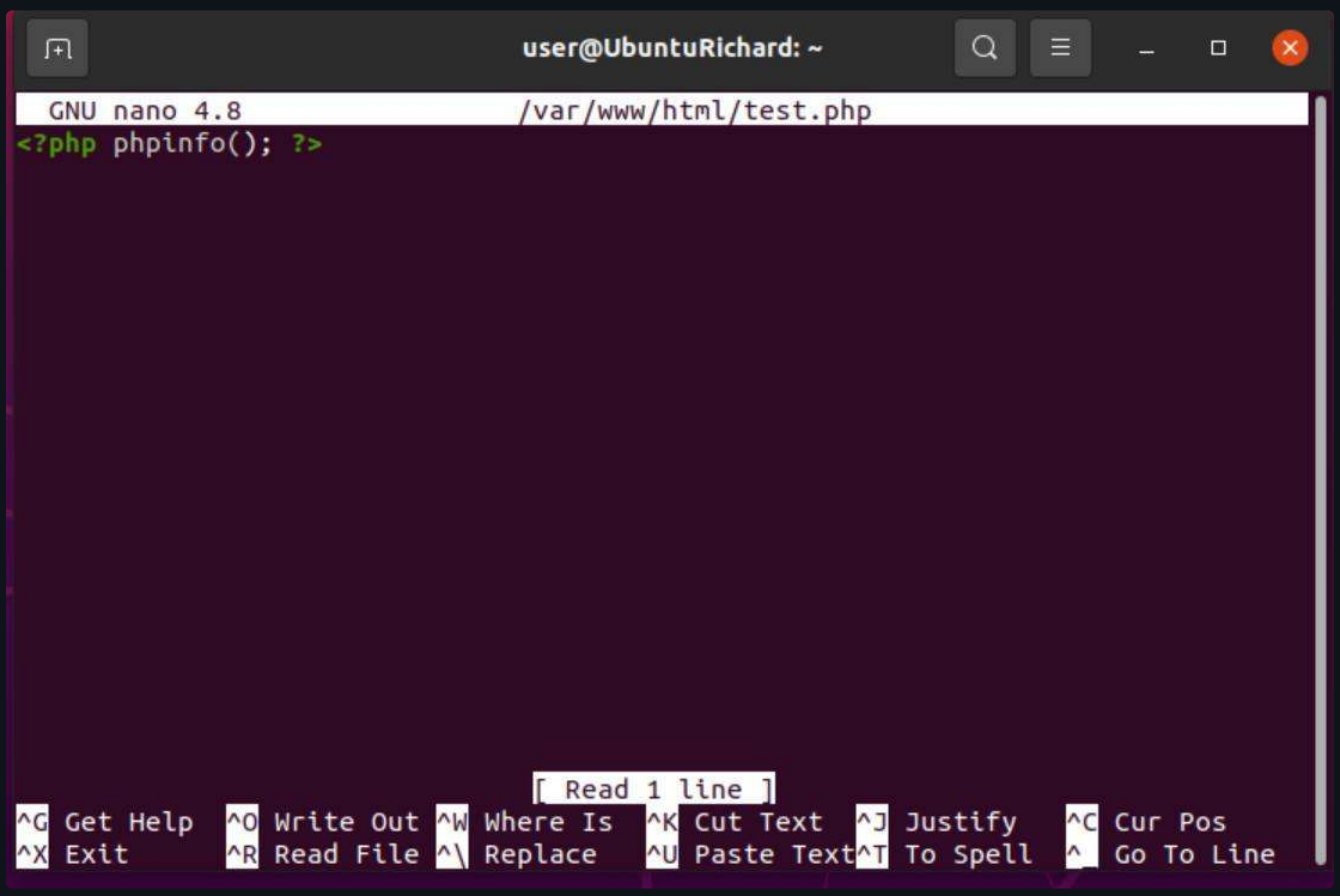

FIG 54 — Creating a probe page with sudo nano /var/www/html/test.php to verify PHP runs on the server.

LAMP STACK

Verify PHP and Log Into MySQL

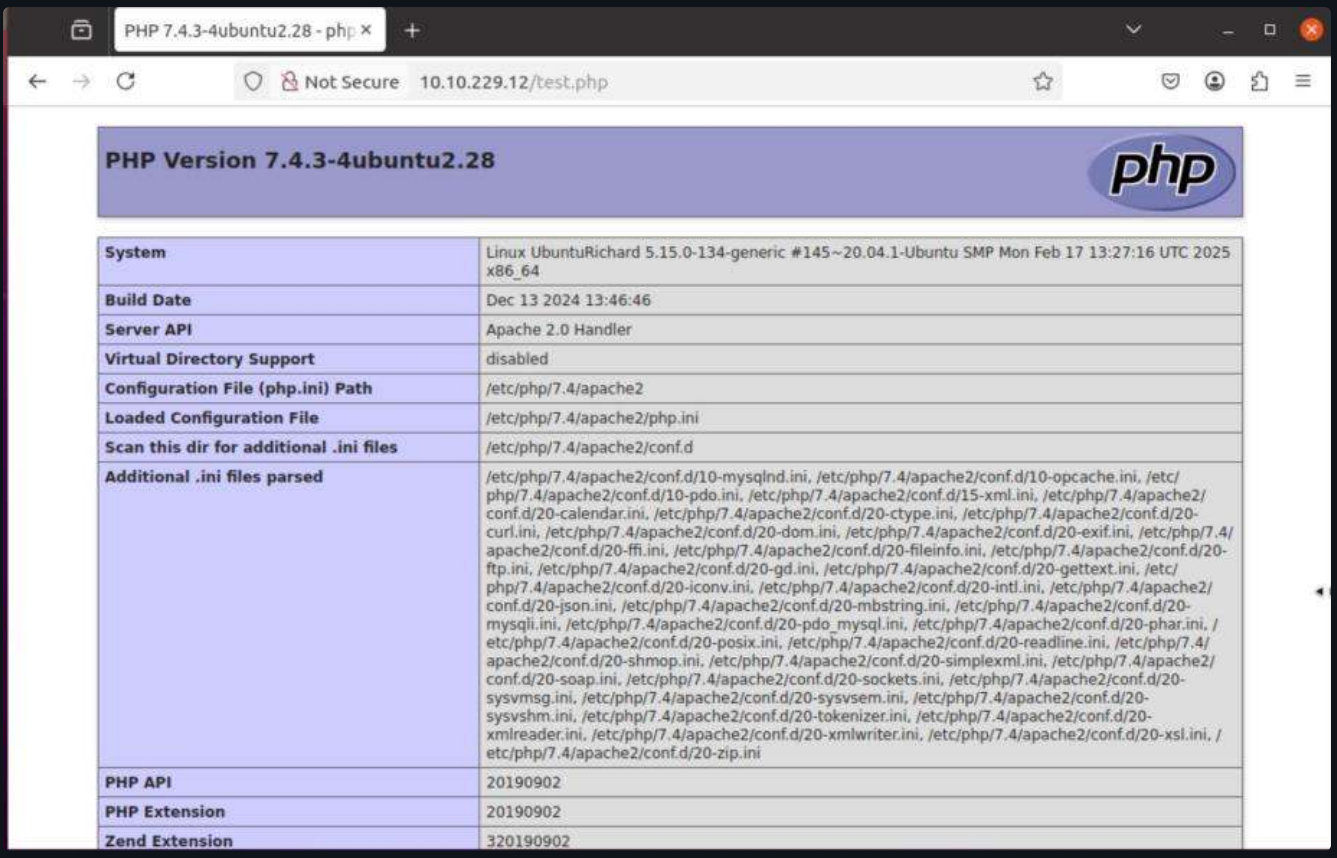

FIG 55 — The phpinfo() output at http://10.10.229.12/test.php, confirming Apache executes PHP and showing the configuration details.

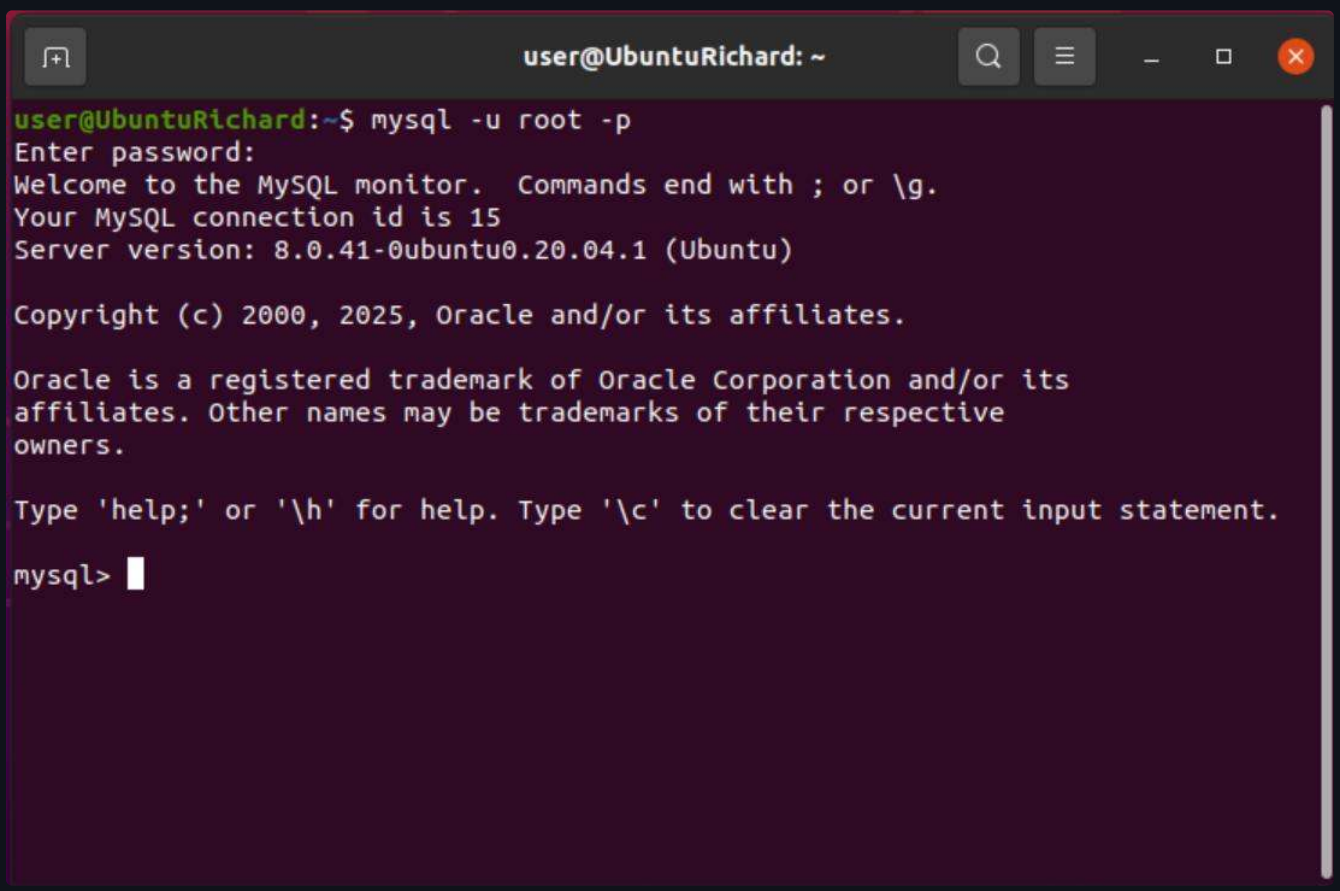

FIG 56 — Logging into the database with mysql -u root -p to begin WordPress database configuration.

DATABASE

Create the WordPress Database and User

Created the WordPressDB database and a dedicated WordPressUser account.

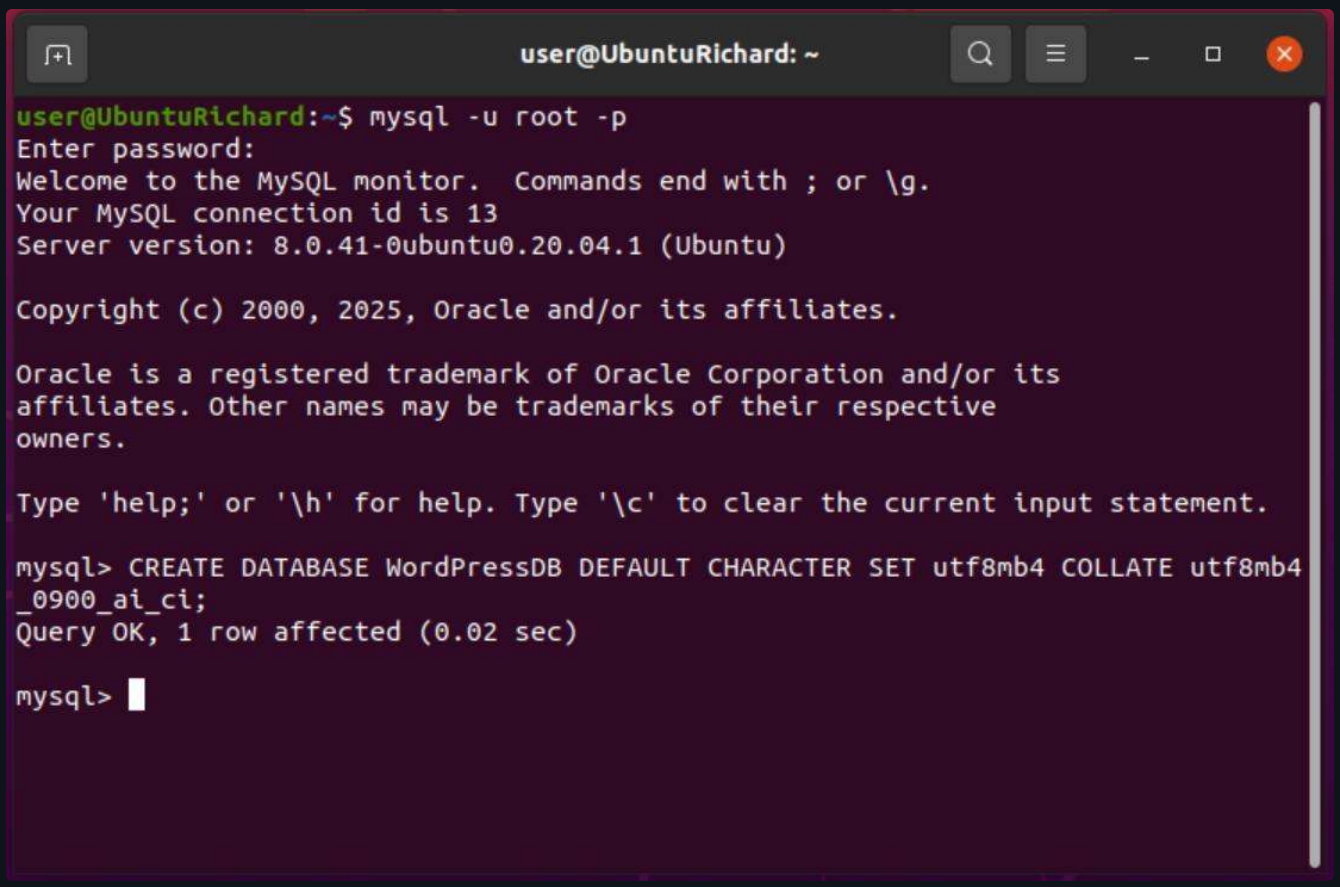

FIG 57 — CREATE DATABASE WordPressDB DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_0900_ai_ci; provisioning the WordPress database with UTF-8 encoding.

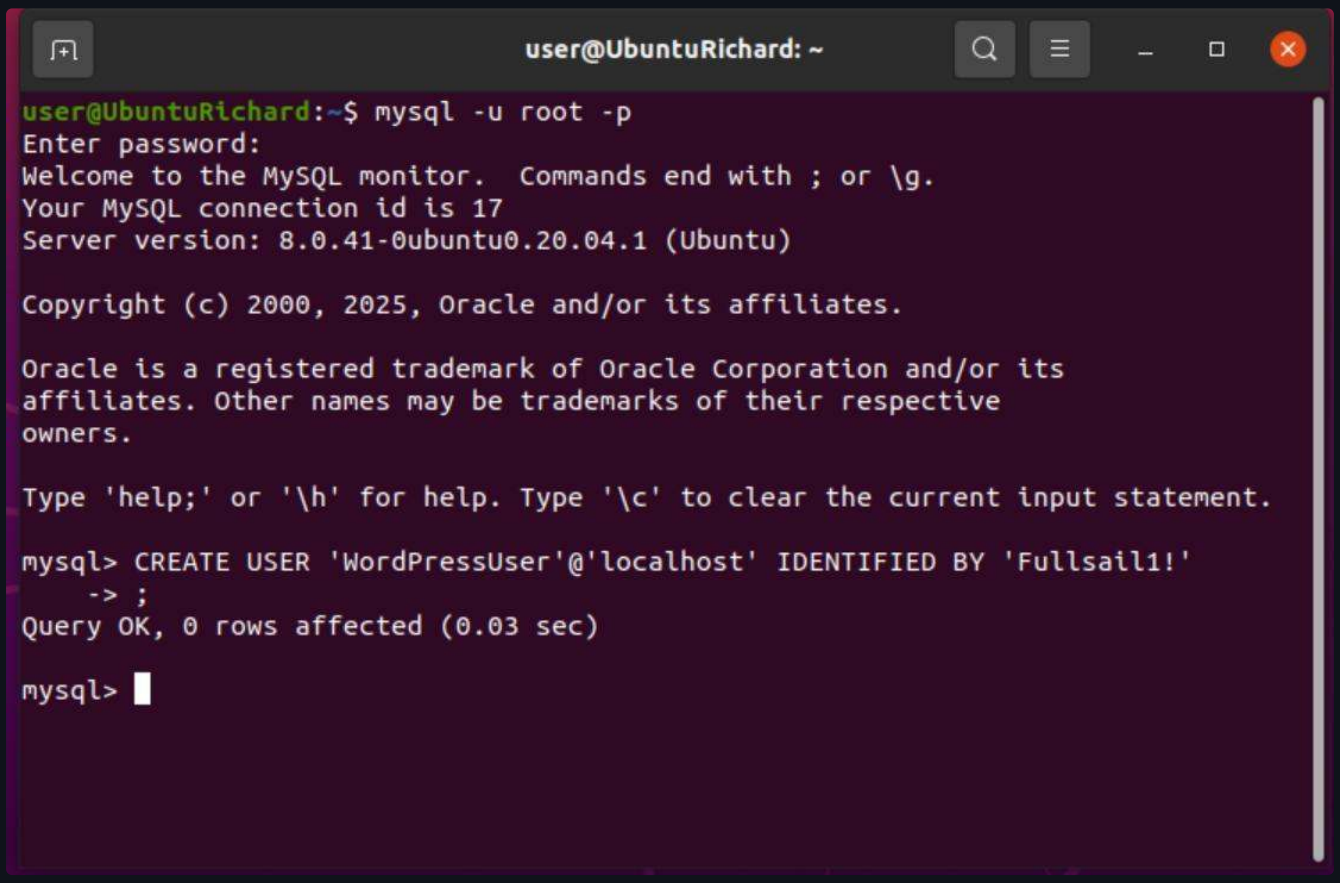

FIG 58 — CREATE USER 'WordPressUser'@'localhost' IDENTIFIED BY 'FullSail1!'; creating the dedicated database account WordPress will use.

DATABASE

Verify User and Grant Privileges

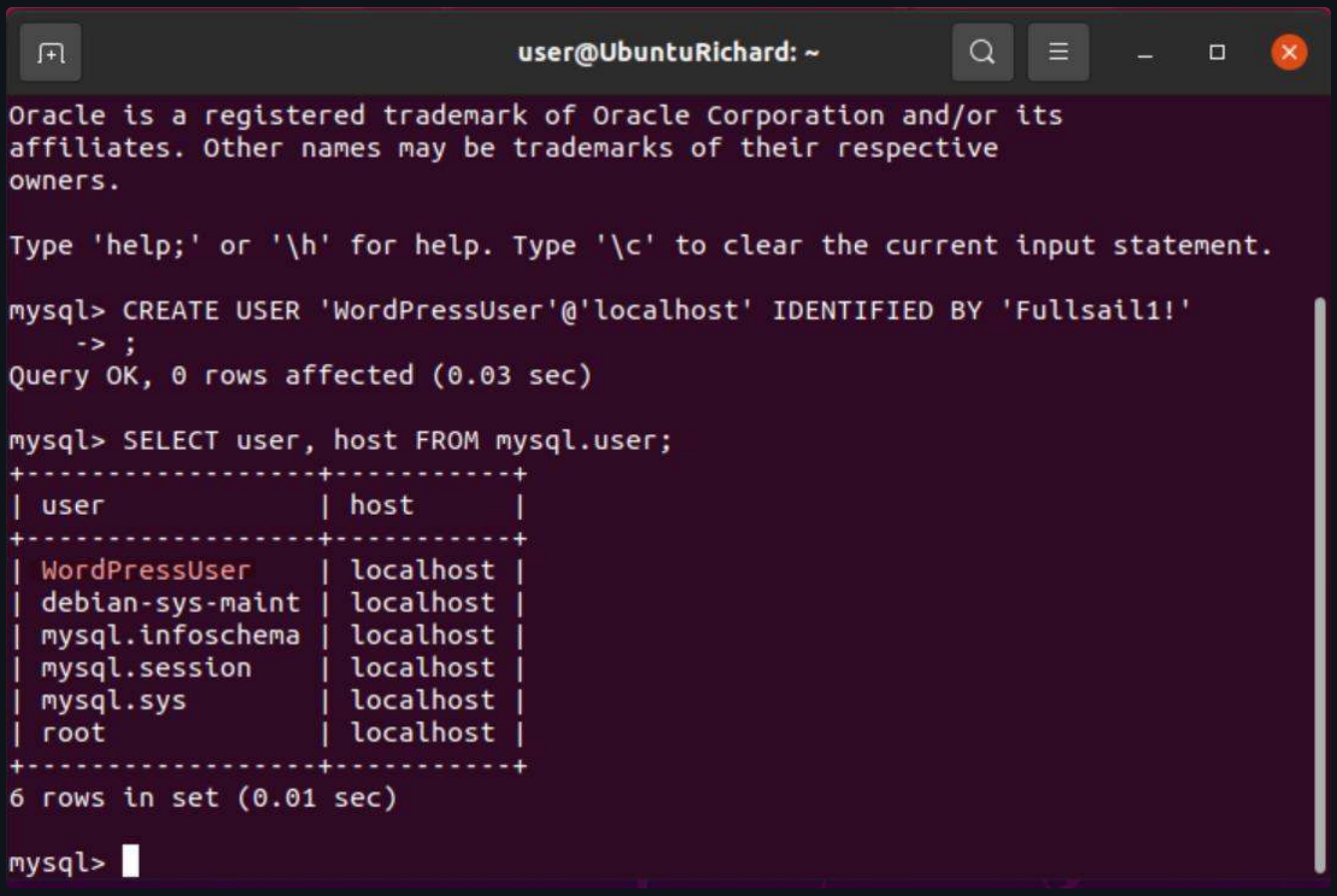

FIG 59 — SELECT user, host FROM mysql.user; confirming WordPressUser was created and bound to localhost.

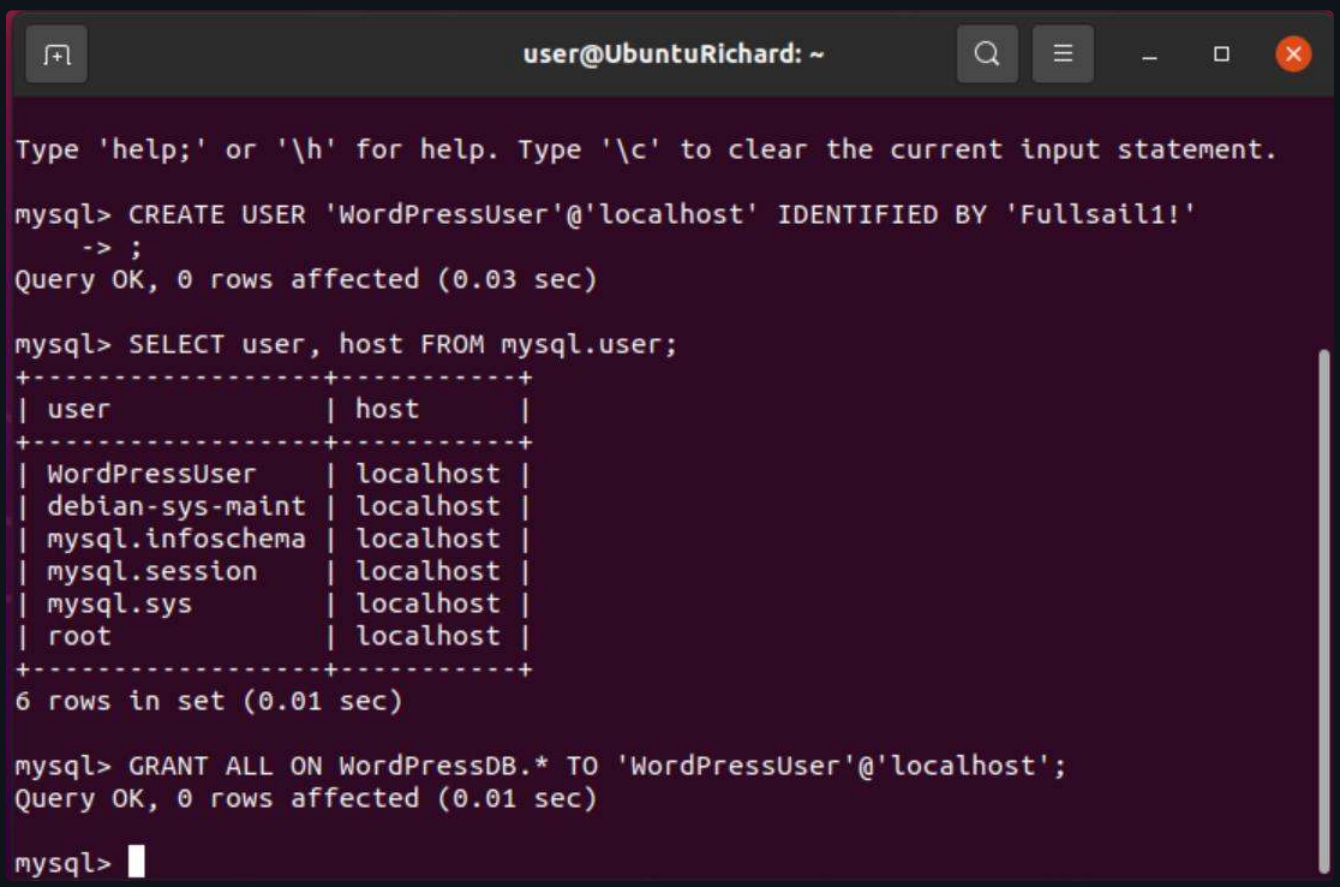

FIG 60 — GRANT ALL ON WordPressDB.* TO 'WordPressUser'@'localhost'; giving the account full read/write access to the WordPress database.

DATABASE

Flush Privileges and Exit

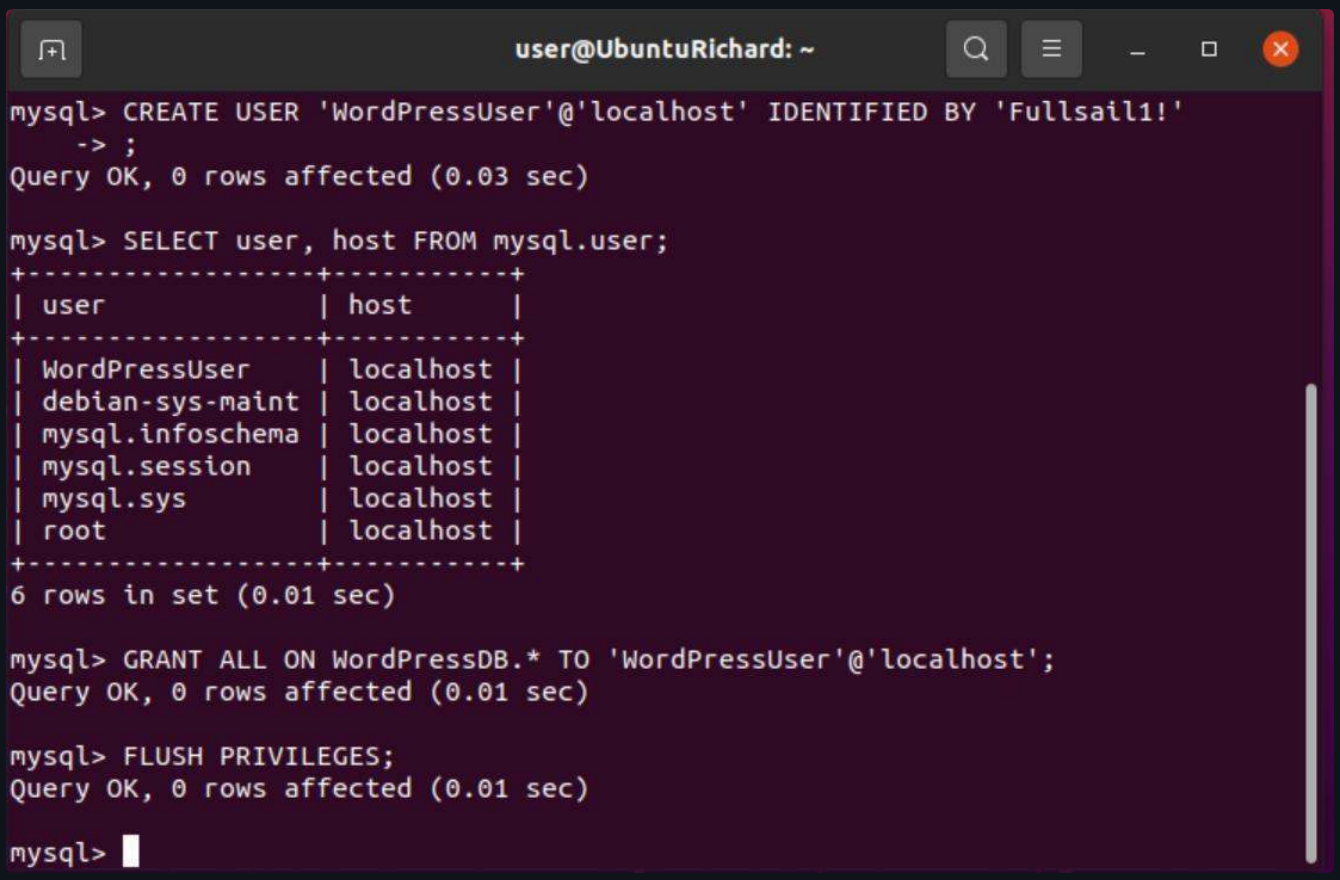

FIG 61 — FLUSH PRIVILEGES; applying the new grant so MySQL recognizes the updated permissions.

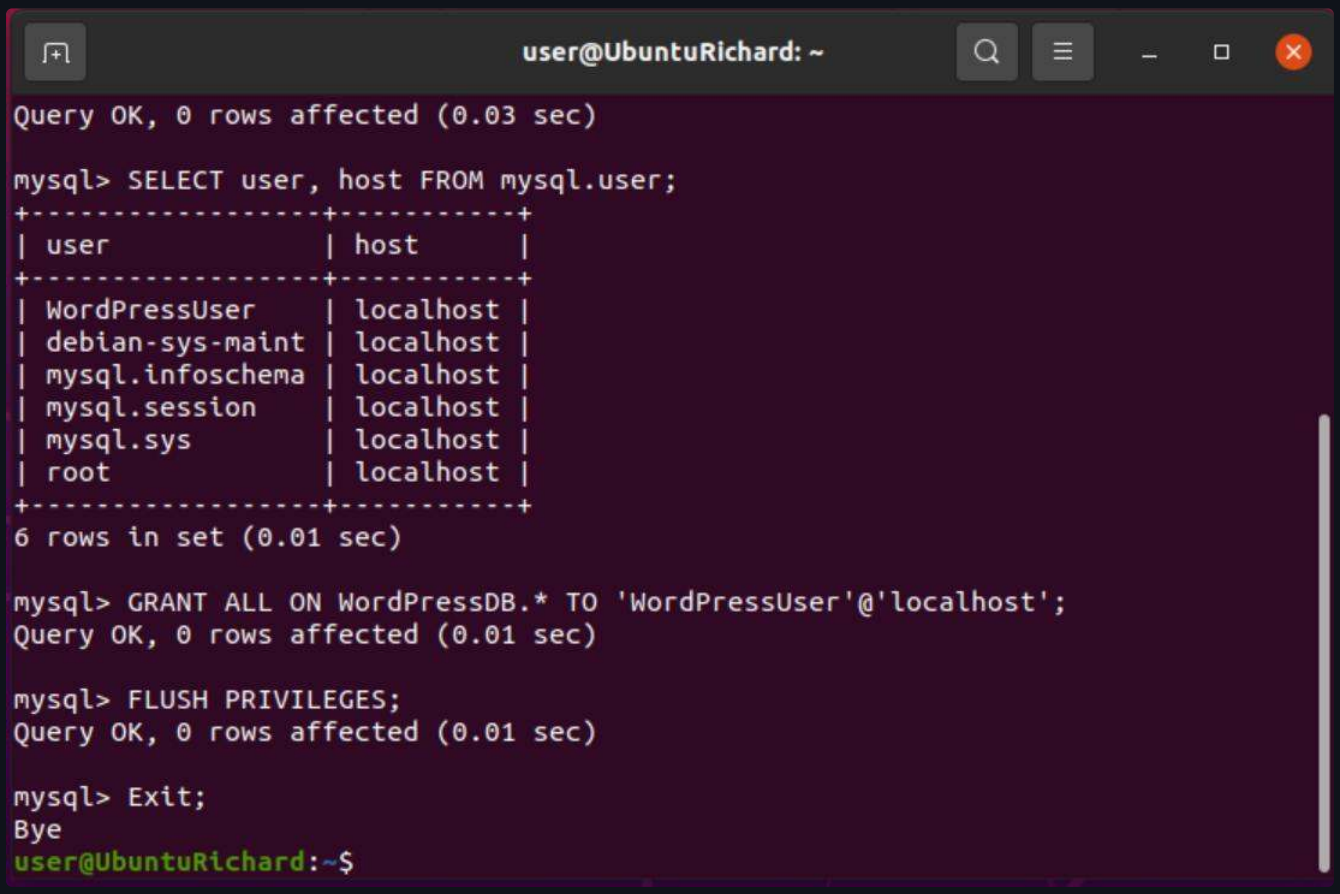

FIG 62 — Leaving the shell with EXIT; after completing database and user configuration.

WORDPRESS

Prepare the Web Root

Took ownership of /var/www/html and cleared it for a fresh WordPress install.

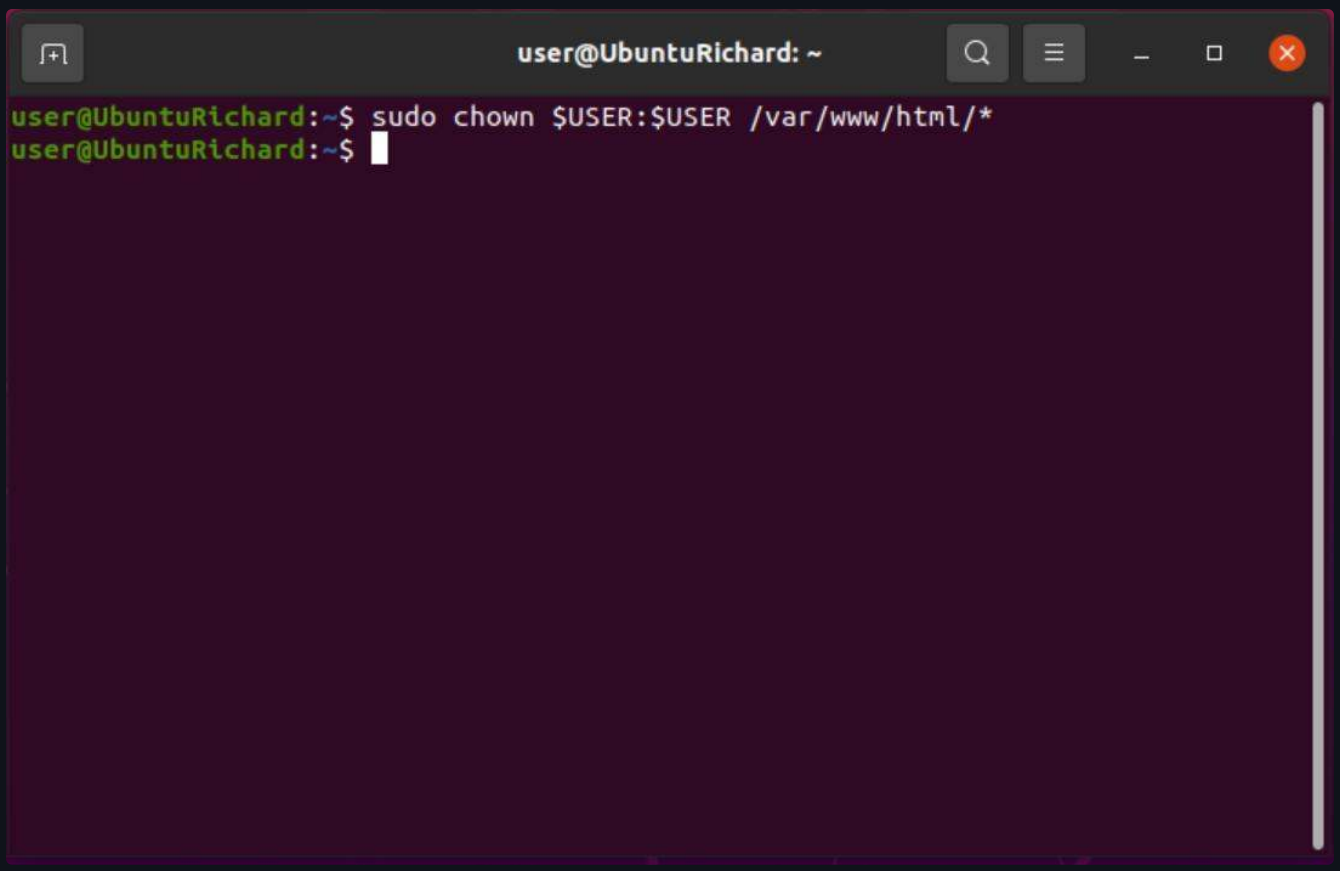

FIG 63 — sudo chown $USER:$USER /var/www/html/* taking ownership of the web root for the install.

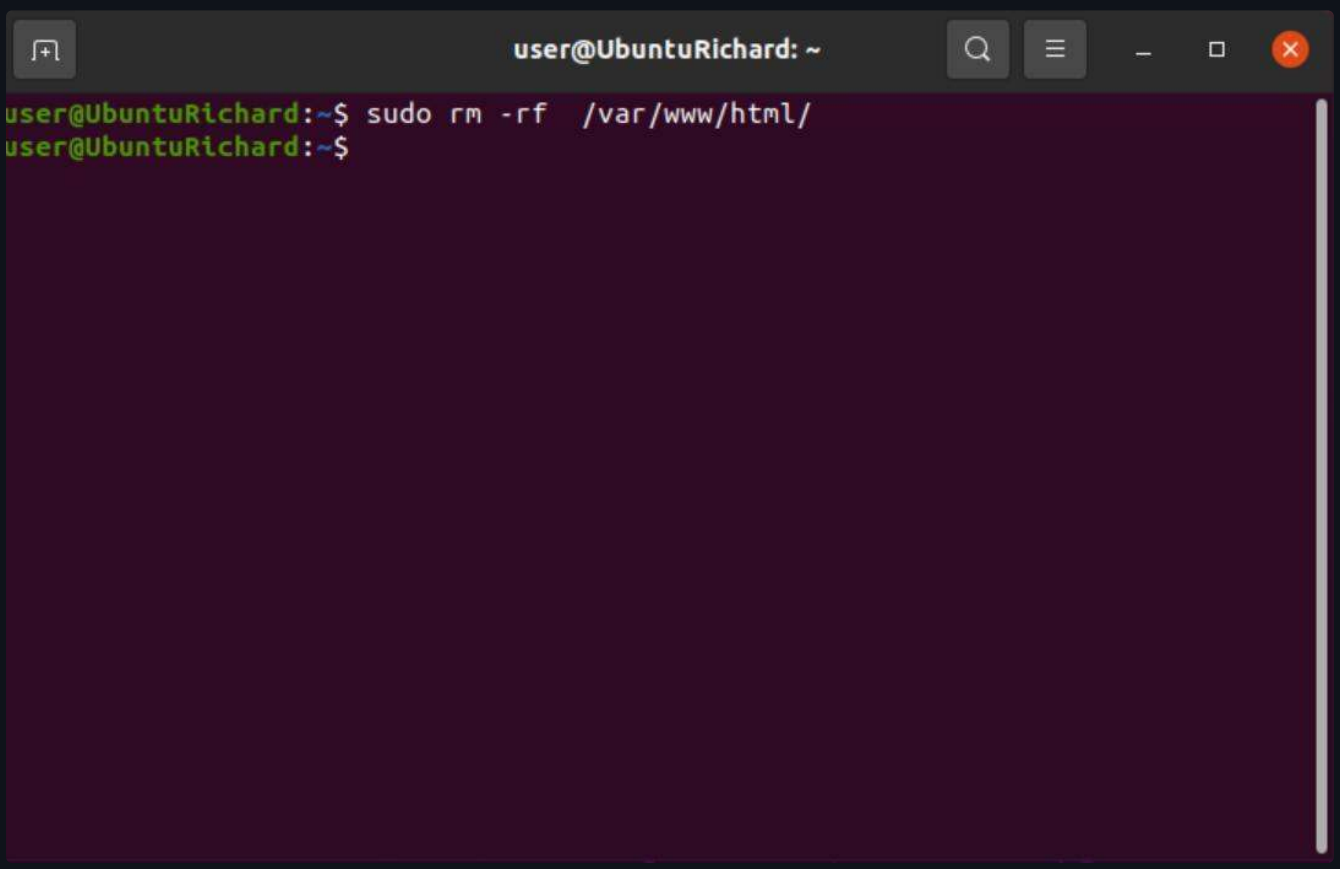

FIG 64 — sudo rm -rf /var/www/html/* clearing the directory to prepare for a clean WordPress deployment.

WORDPRESS

Confirm Empty Root and Clone WordPress

Verified the web root was empty, then cloned WordPress from GitHub into it.

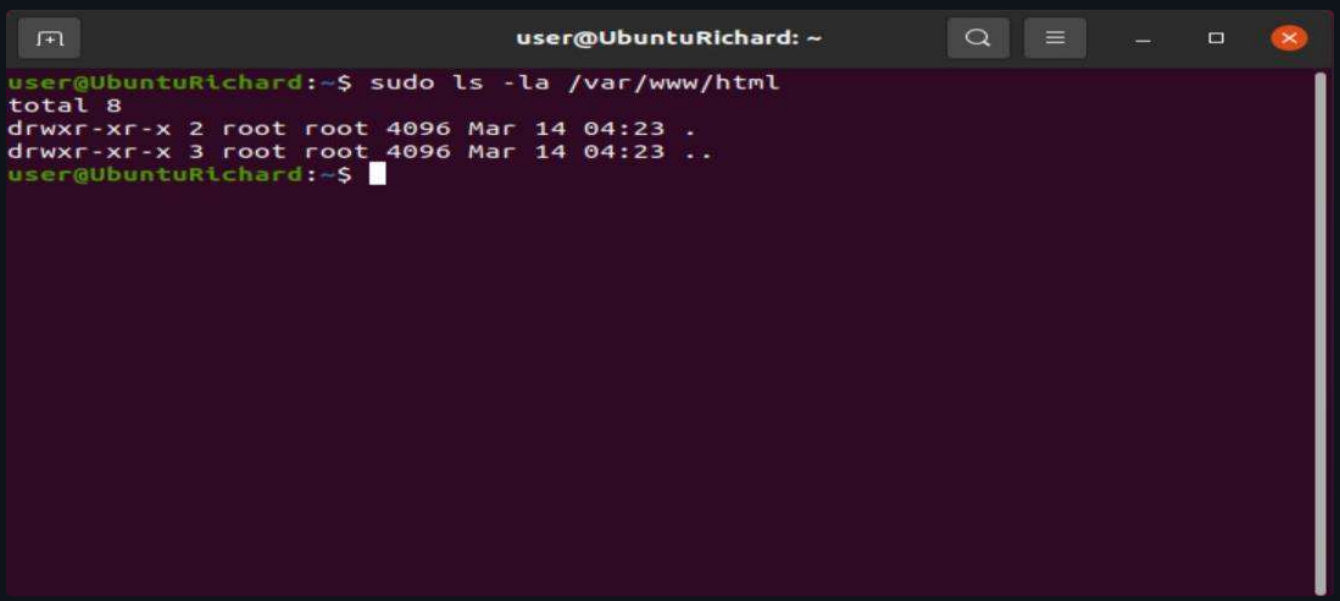

FIG 65 — sudo ls -la /var/www/html confirming the directory is empty before installation.

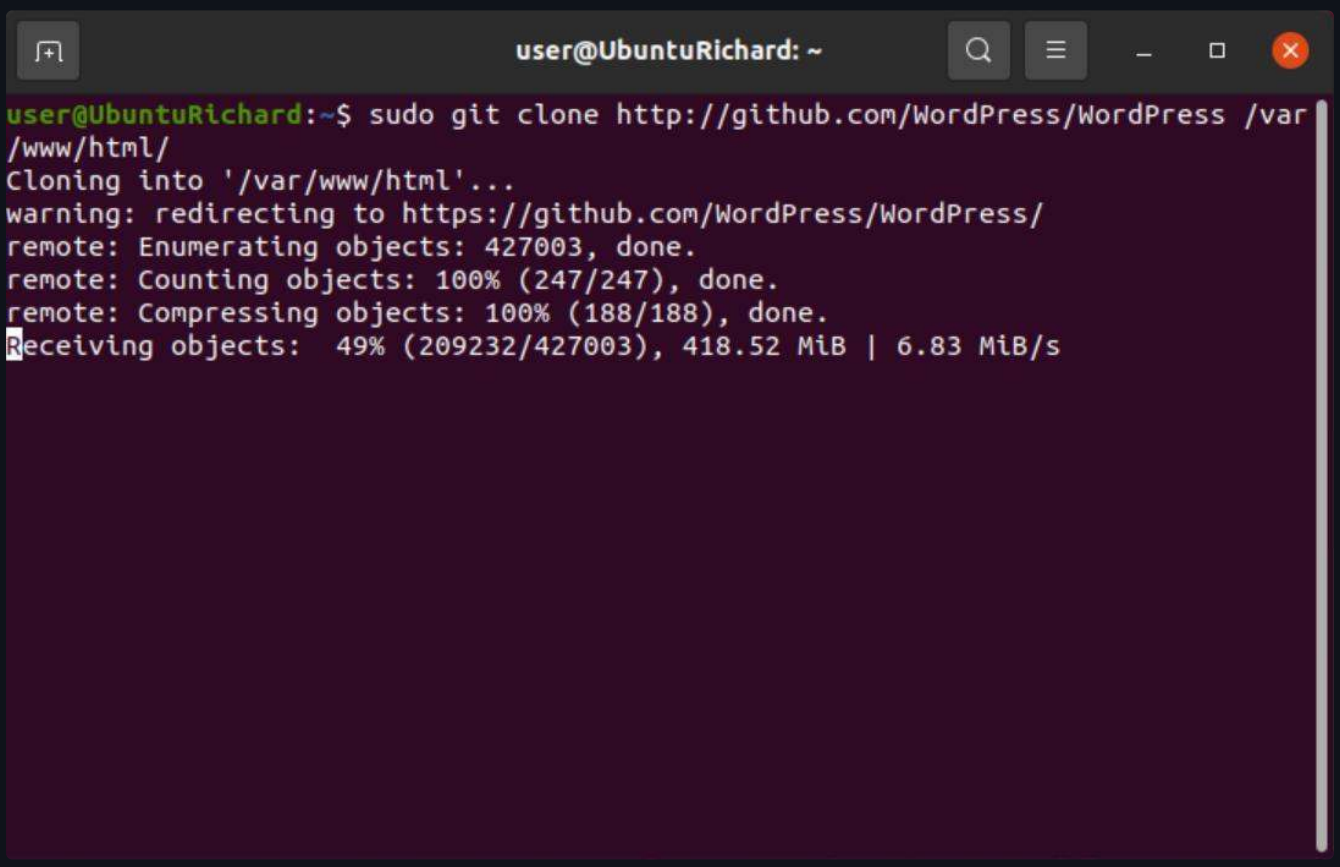

FIG 66 — sudo git clone https://github.com/WordPress/WordPress /var/www/html/ cloning the latest WordPress source into the web root.

WORDPRESS

Verify Clone and Directory Permissions

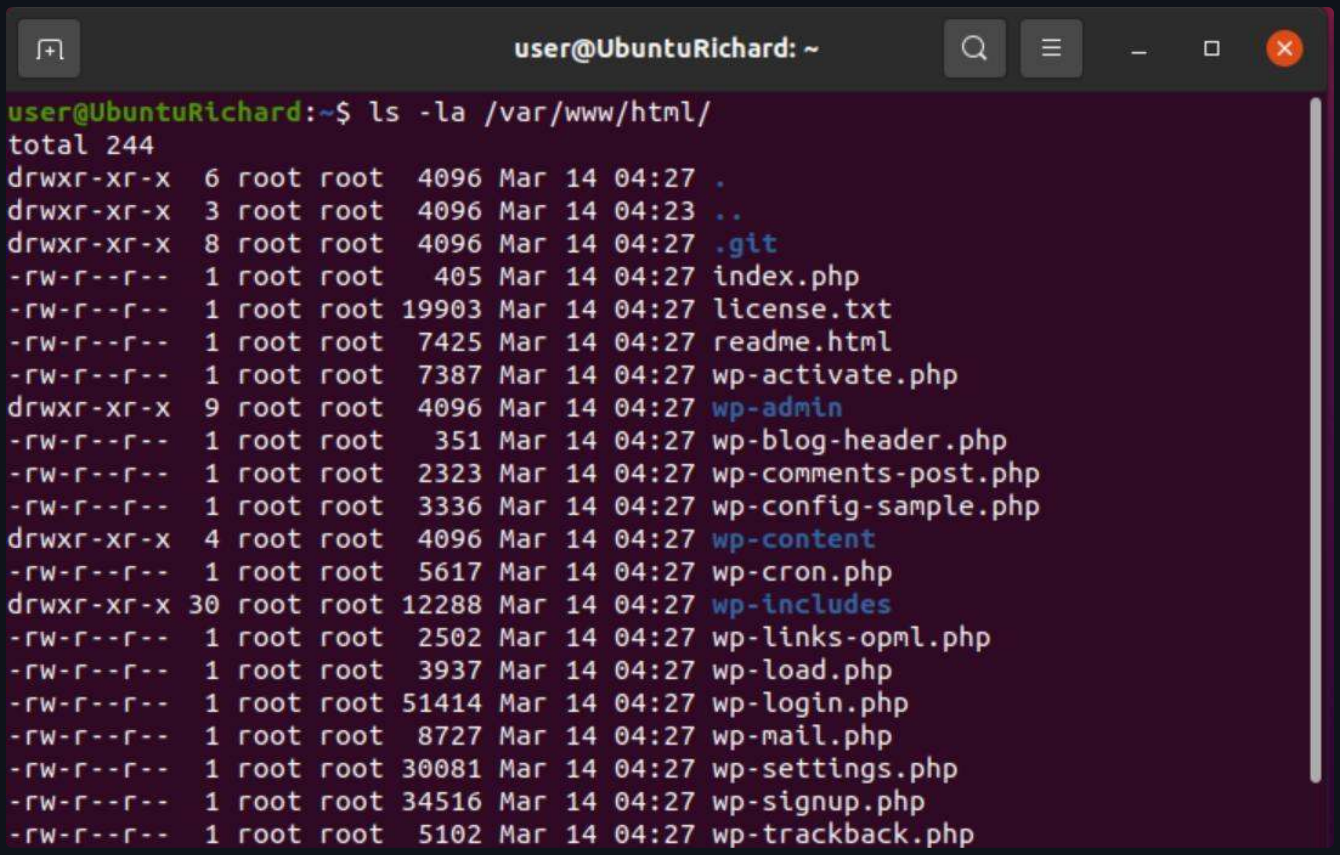

FIG 67 — ls -la /var/www/html/ confirming the WordPress files were cloned successfully.

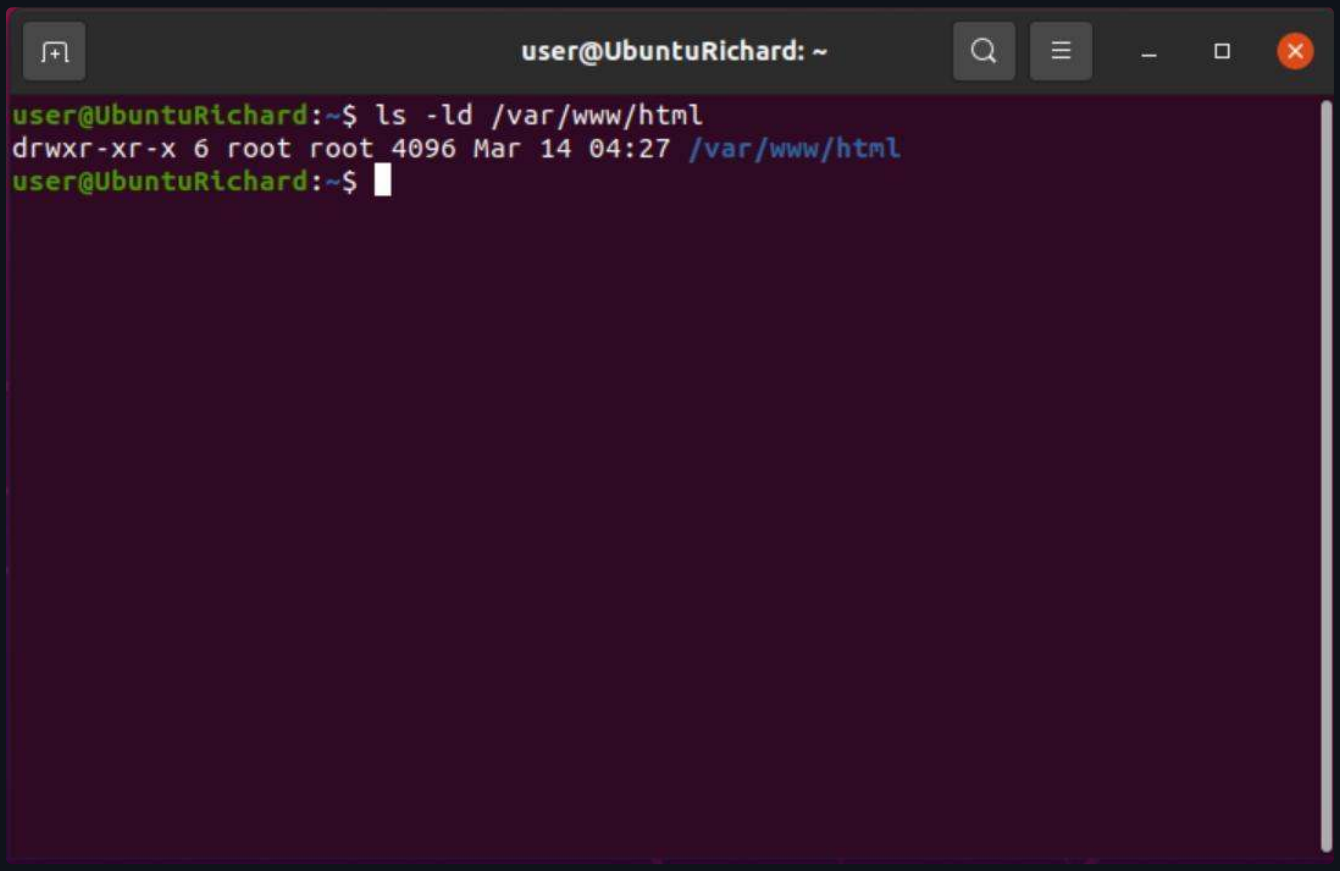

FIG 68 — ls -ld /var/www/html/ checking the directory permissions for web-server accessibility.

WORDPRESS

Set Ownership and Edit apache2.conf

Handed the files to the www-data web-server user and opened the Apache config to allow .htaccess overrides.

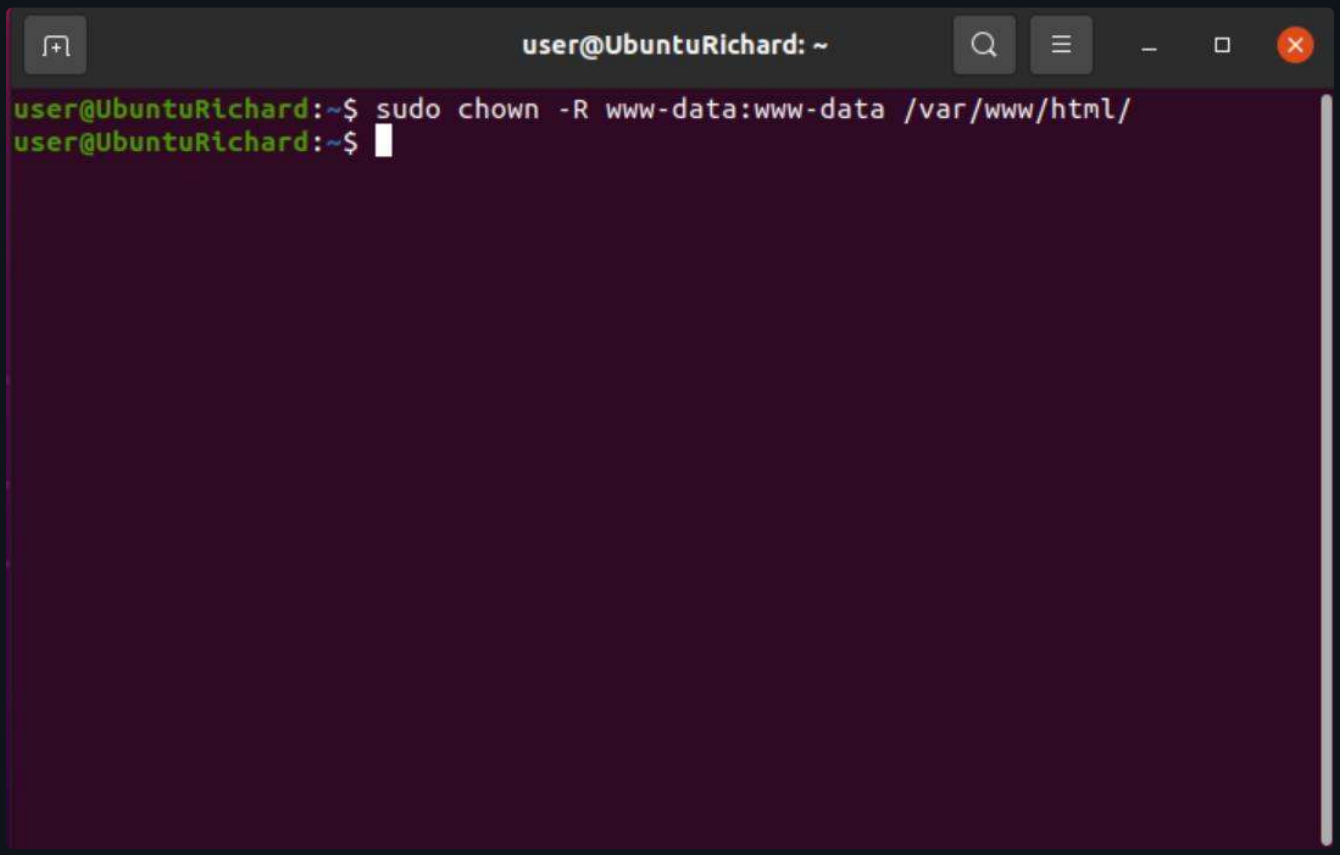

FIG 69 — sudo chown -R www-data:www-data /var/www/html/ giving the Apache user read/write access to the WordPress files.

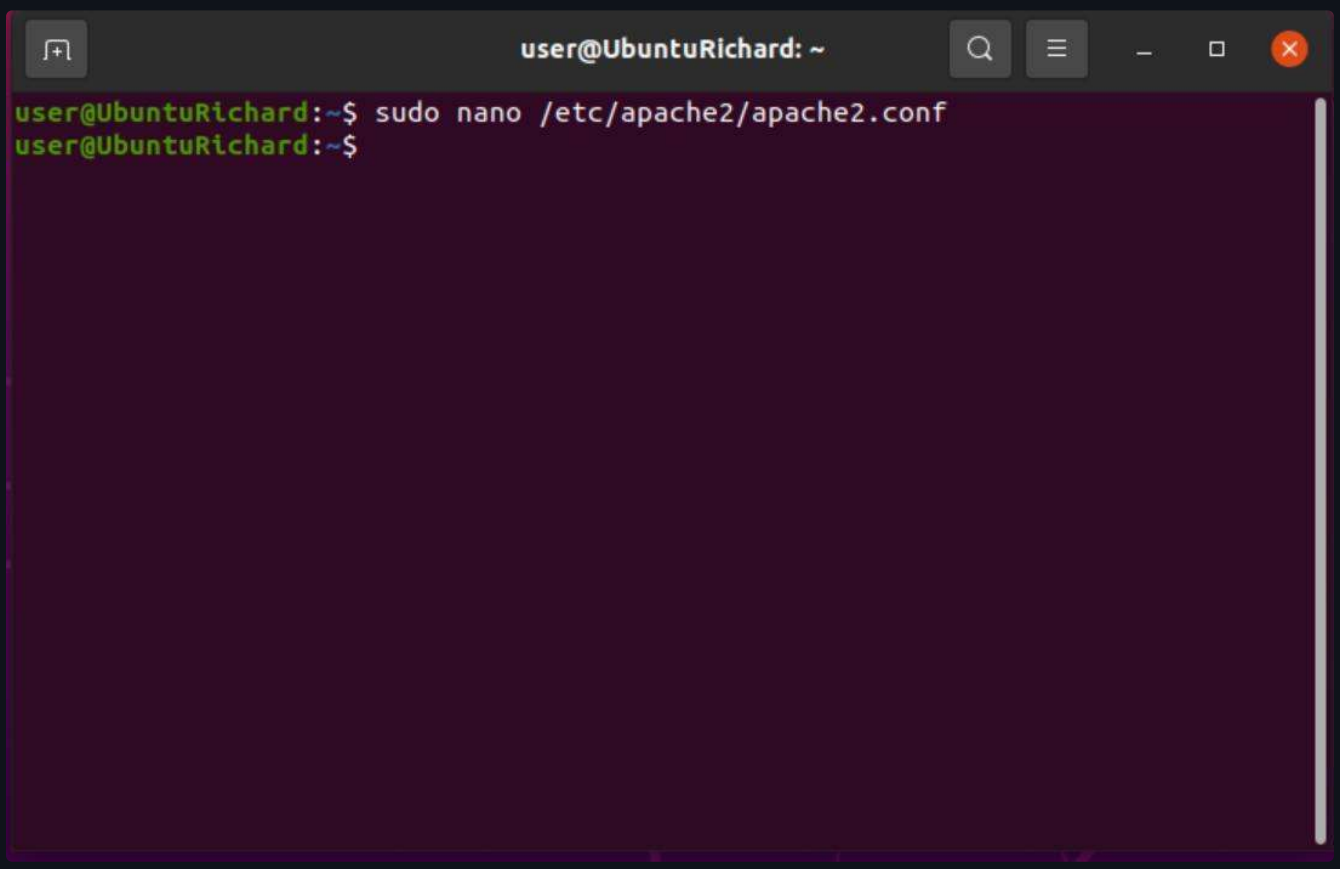

FIG 70 — Opening the Apache config for editing with sudo nano /etc/apache2/apache2.conf.

WORDPRESS

AllowOverride All and Protect .git

Set AllowOverride All for permalinks and added a protective .htaccess to block access to the cloned .git directory.

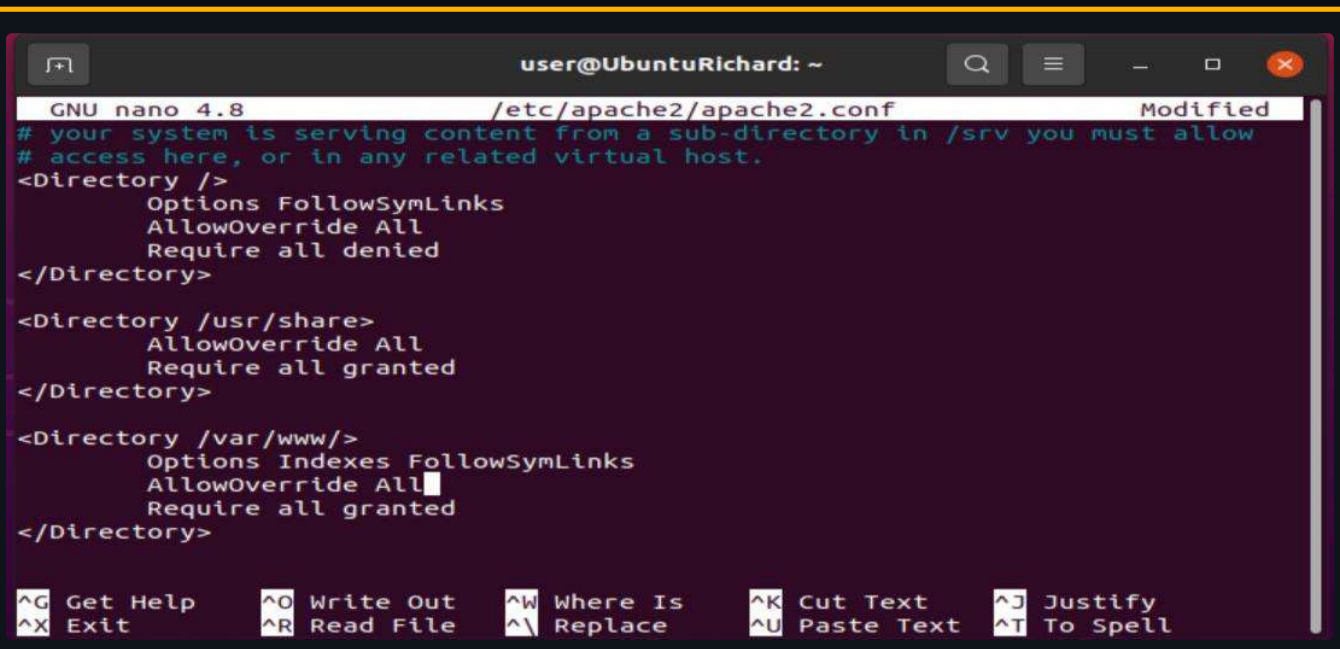

FIG 71 — Setting AllowOverride All in apache2.conf so .htaccess files can govern permalinks and security rules.

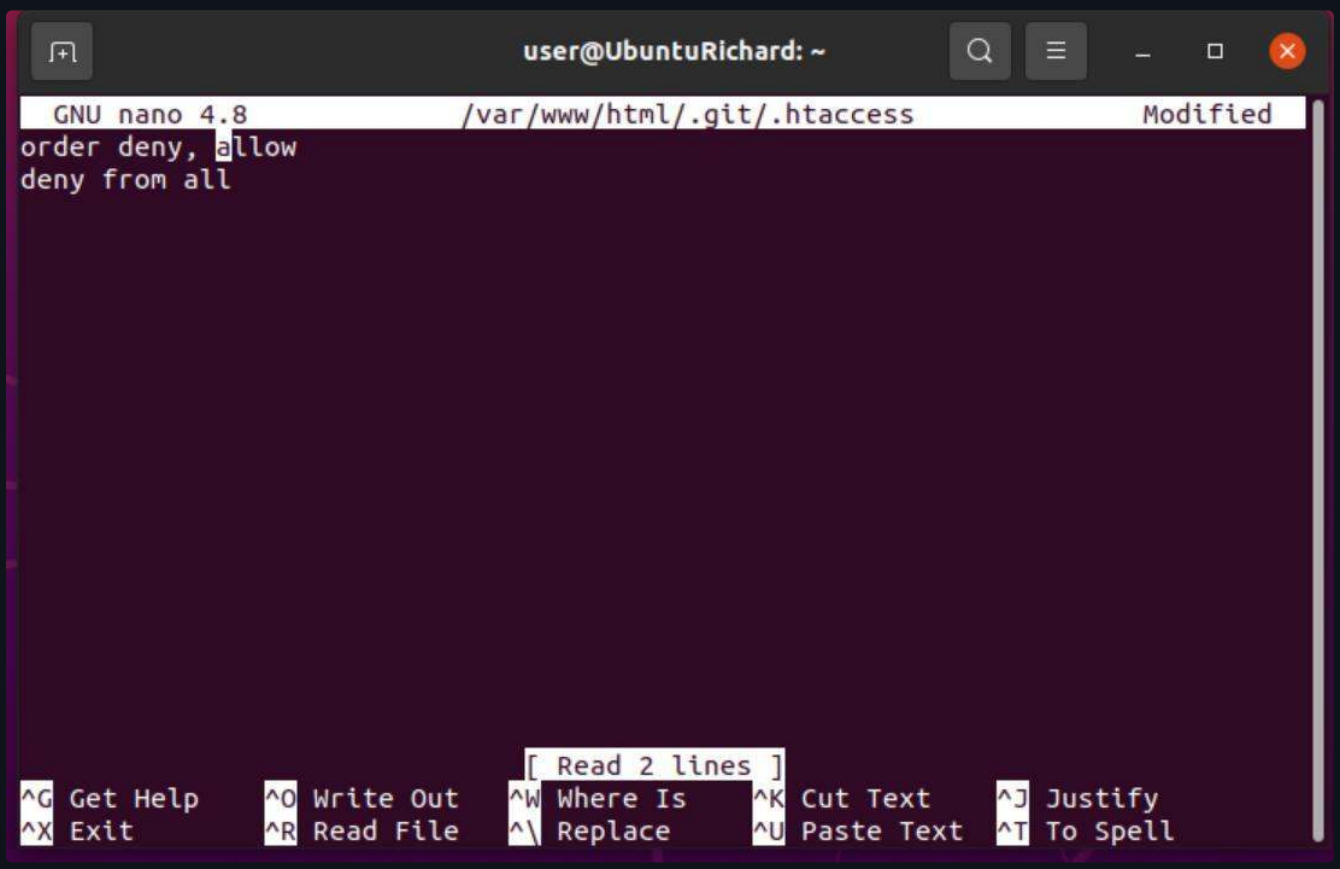

FIG 72 — Creating sudo nano /var/www/html/.git/.htaccess to block unauthorized access to the cloned .git directory.

WORDPRESS

Run the WordPress Installer

Restarted Apache, then ran the web installer against WordPressDB.

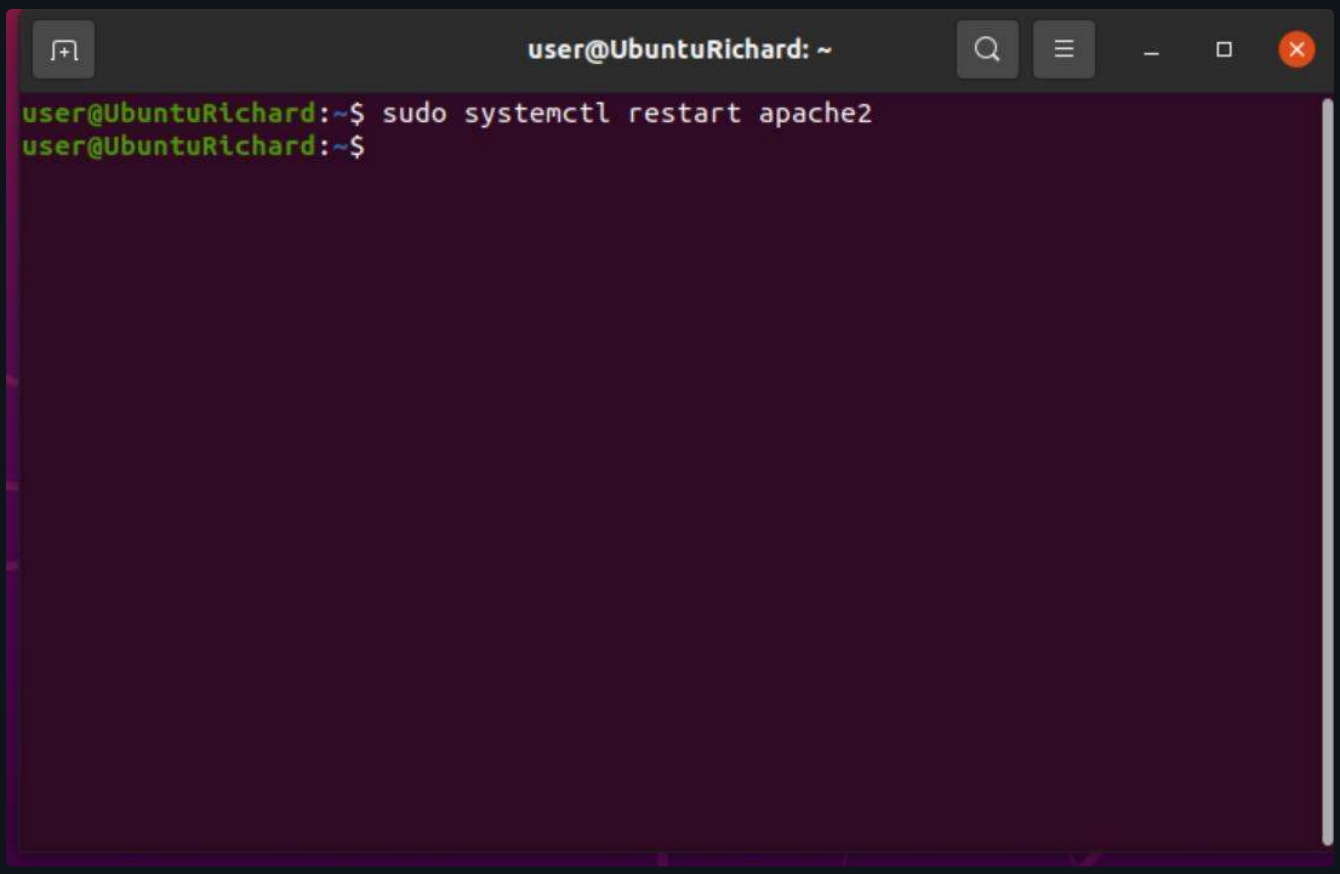

FIG 73 — Restarting the web server with sudo systemctl restart apache2 to apply all configuration changes.

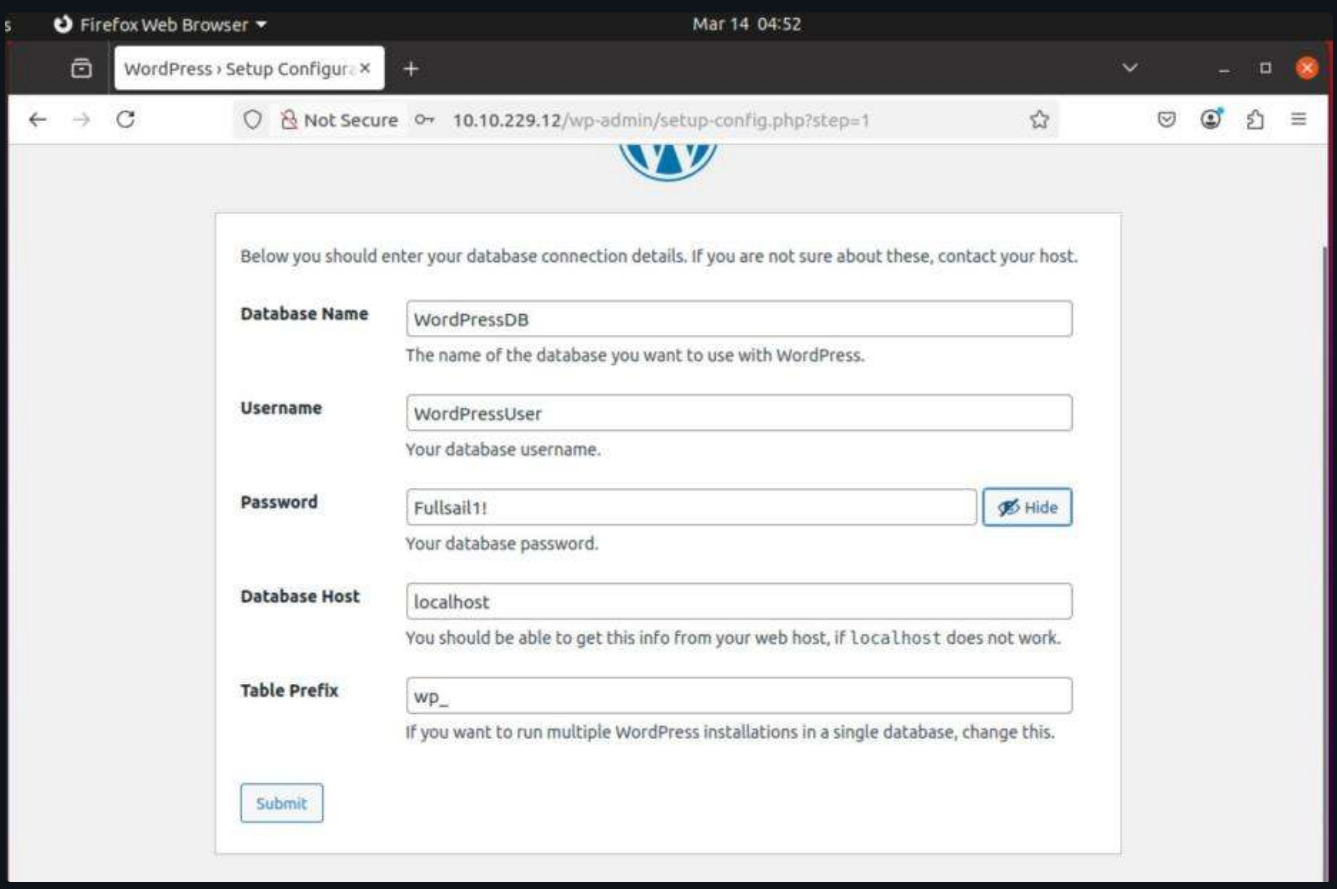

FIG 74 — The WordPress installer database step at http://10.10.229.12, entering WordPressDB, WordPressUser, and the password to connect WordPress to MySQL.

VALIDATION

Installer Success and Live Site

Completed the install with an admin account, applied a theme, and published a post to prove the site is live.

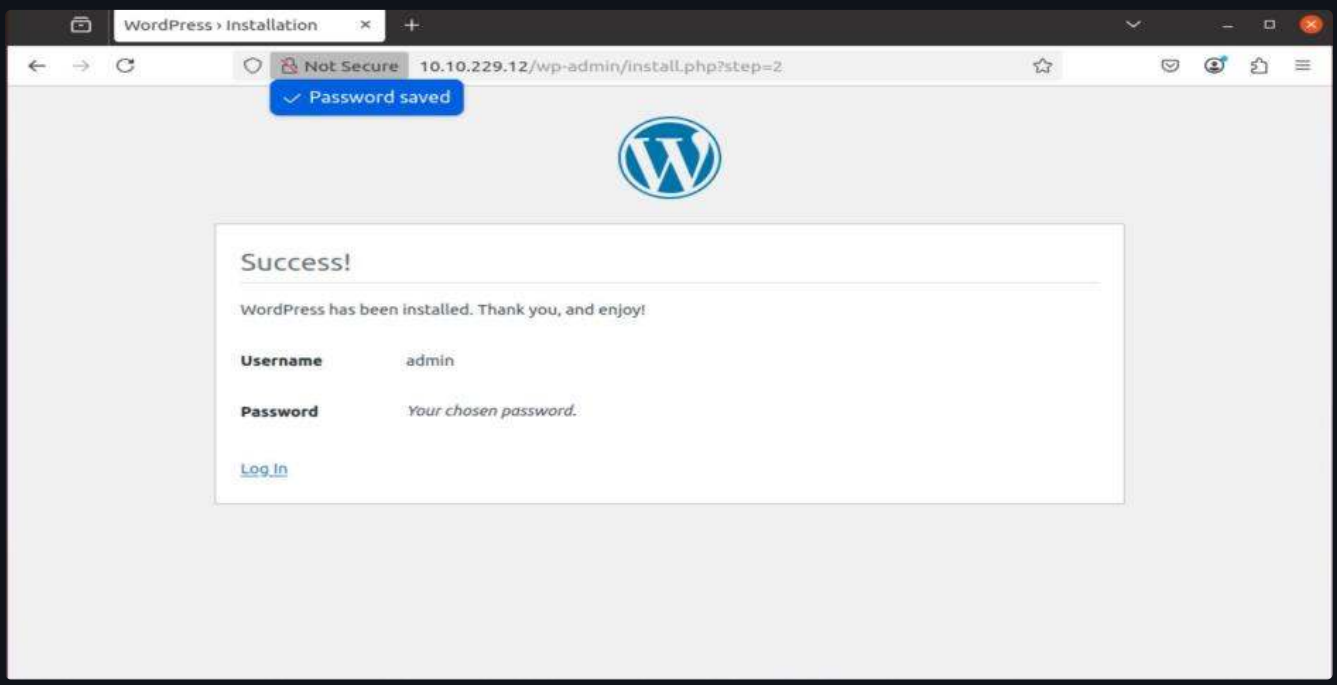

FIG 75 — WordPress installer reporting Success! after creating the admin account and connecting to WordPressDB.

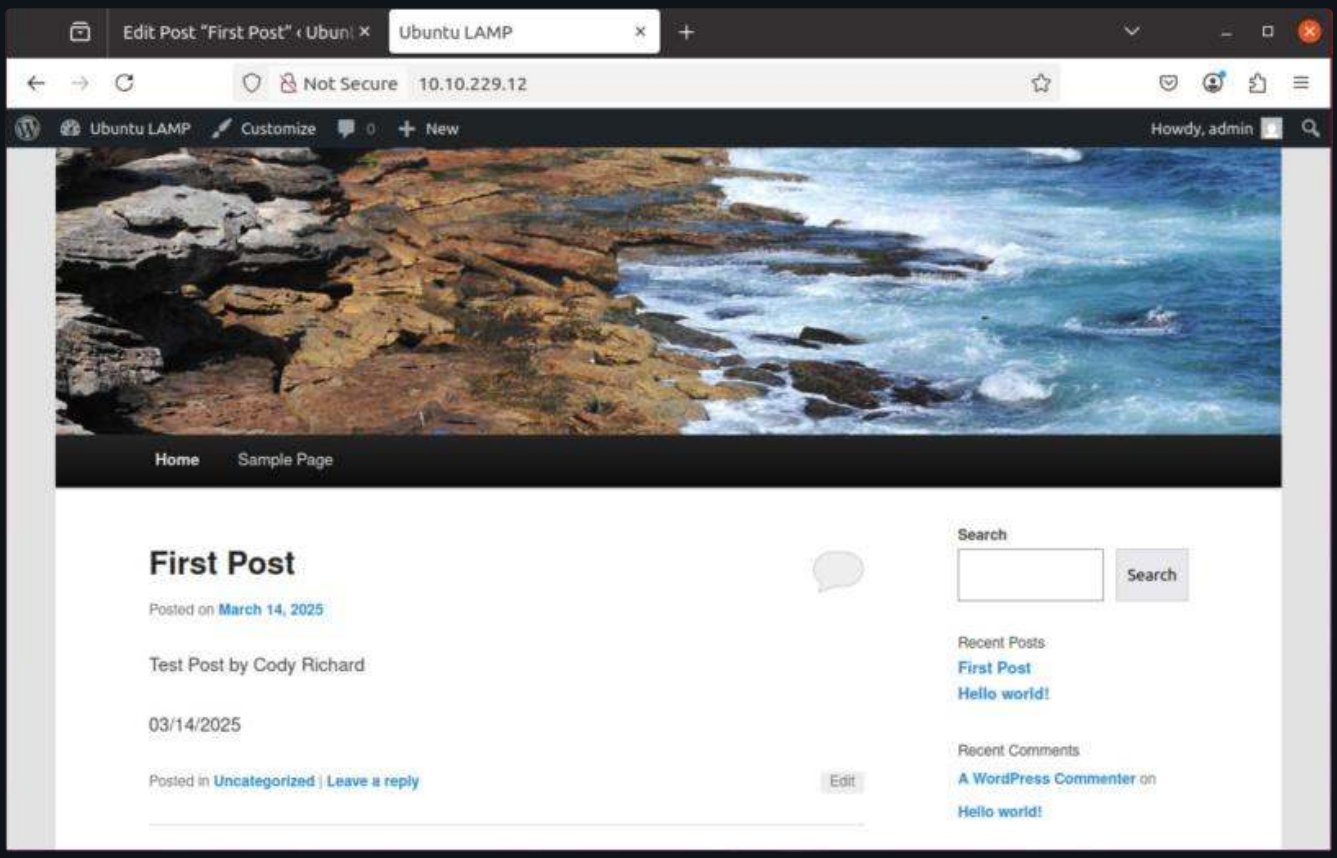

FIG 76 — The live WordPress site at http://10.10.229.12 with a custom theme applied and a published post, confirming full end-to-end functionality.

HARDENING

File Permissions: Lockdown

Defense-in-depth, layer one: enforced 755 on directories and 644 on files across the web root.

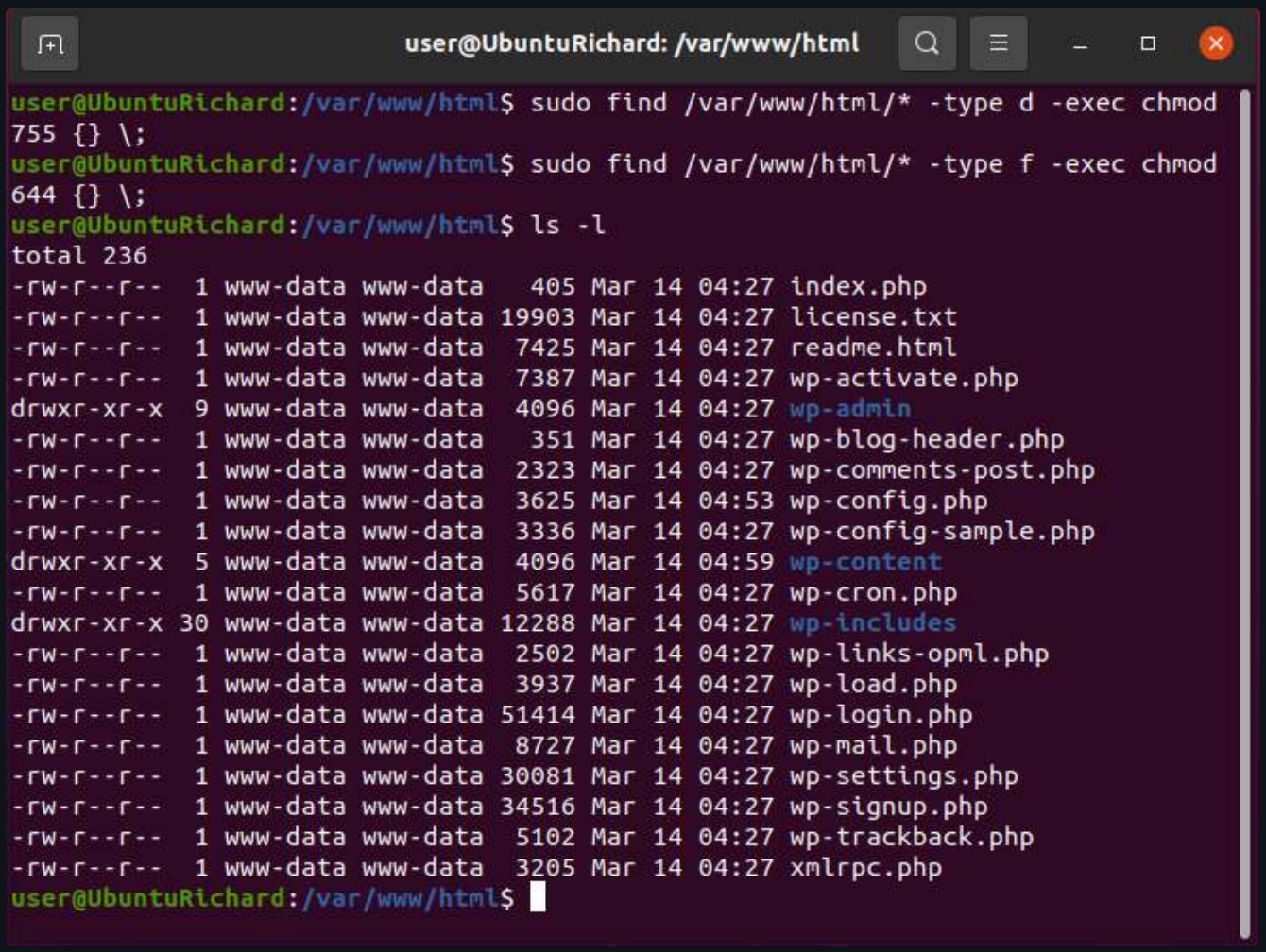

FIG 77 — Baseline permission state of /var/www/html before hardening, establishing the starting point for the lockdown.

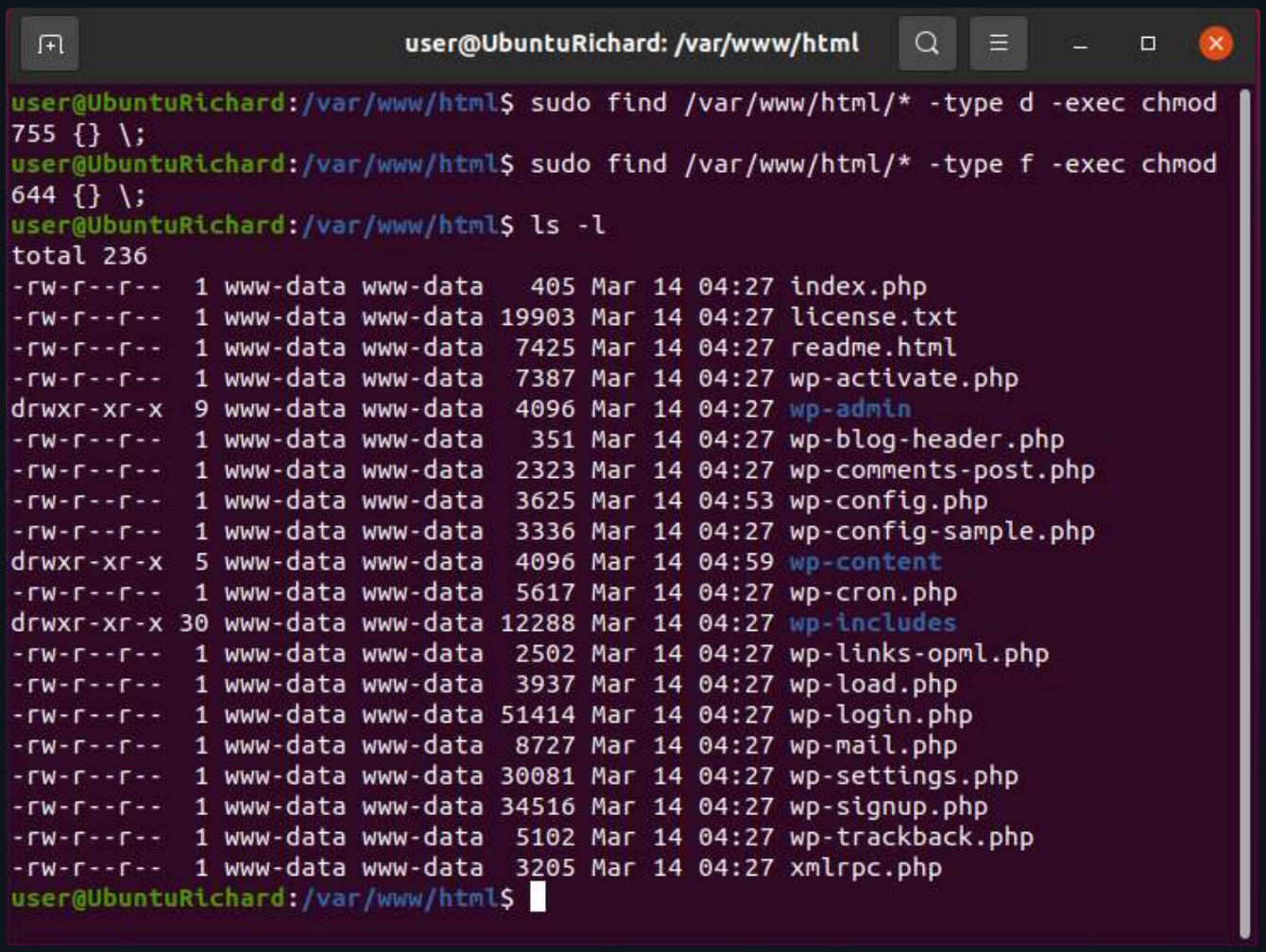

FIG 78 — sudo find /var/www/html/ -type d -exec chmod 755 {} \; and sudo find /var/www/html/ -type f -exec chmod 644 {} \; setting secure defaults (755 directories, 644 files) so only the web-server owner can modify content.

HARDENING

Relocate and Lock wp-config.php

Layer two: moved wp-config.php above the web root and restricted it to owner-only read/write (600).

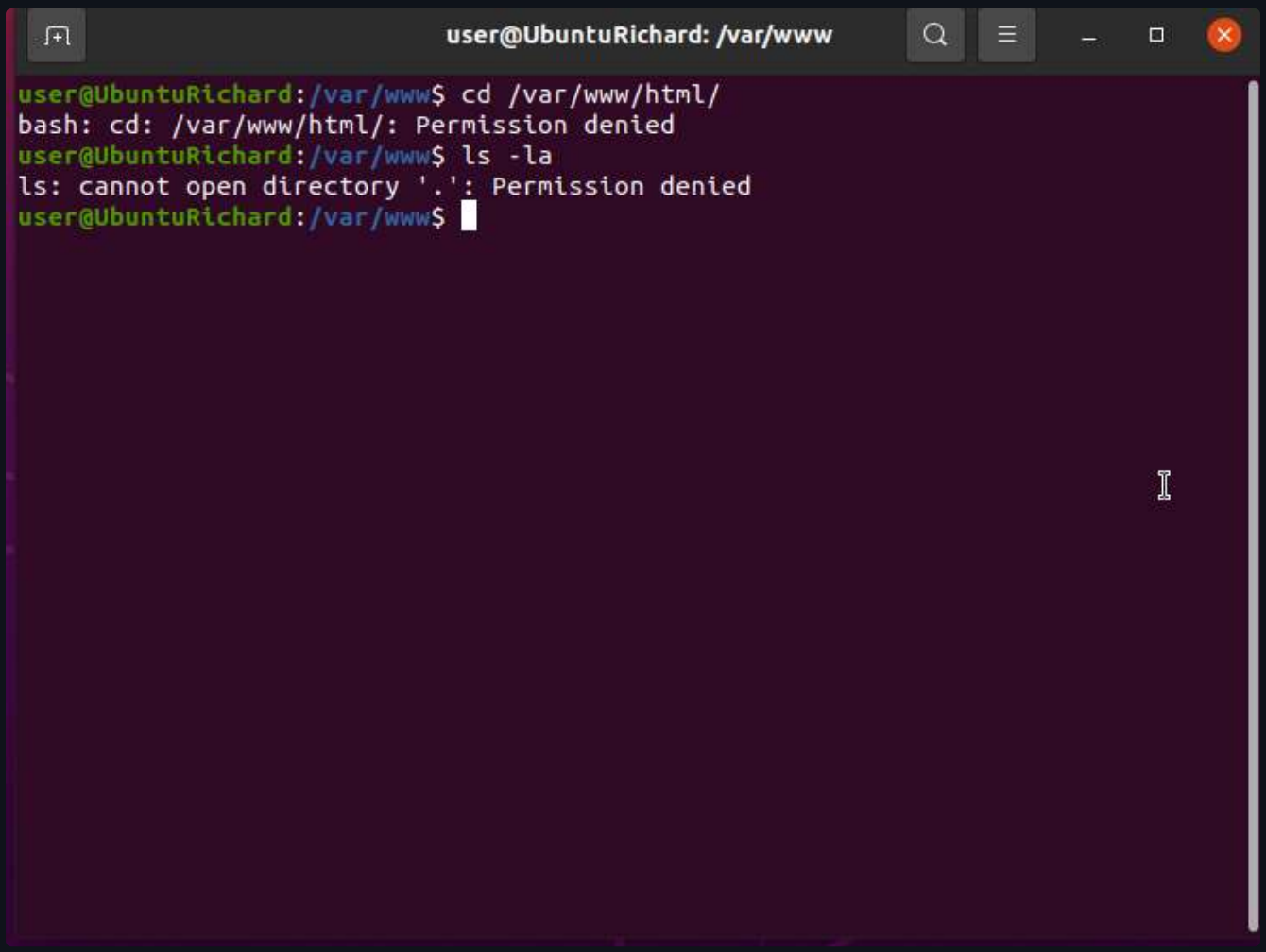

FIG 79 — Permission lockdown validated: world and group write access removed from the web content so attackers cannot tamper with files even with limited shell access.

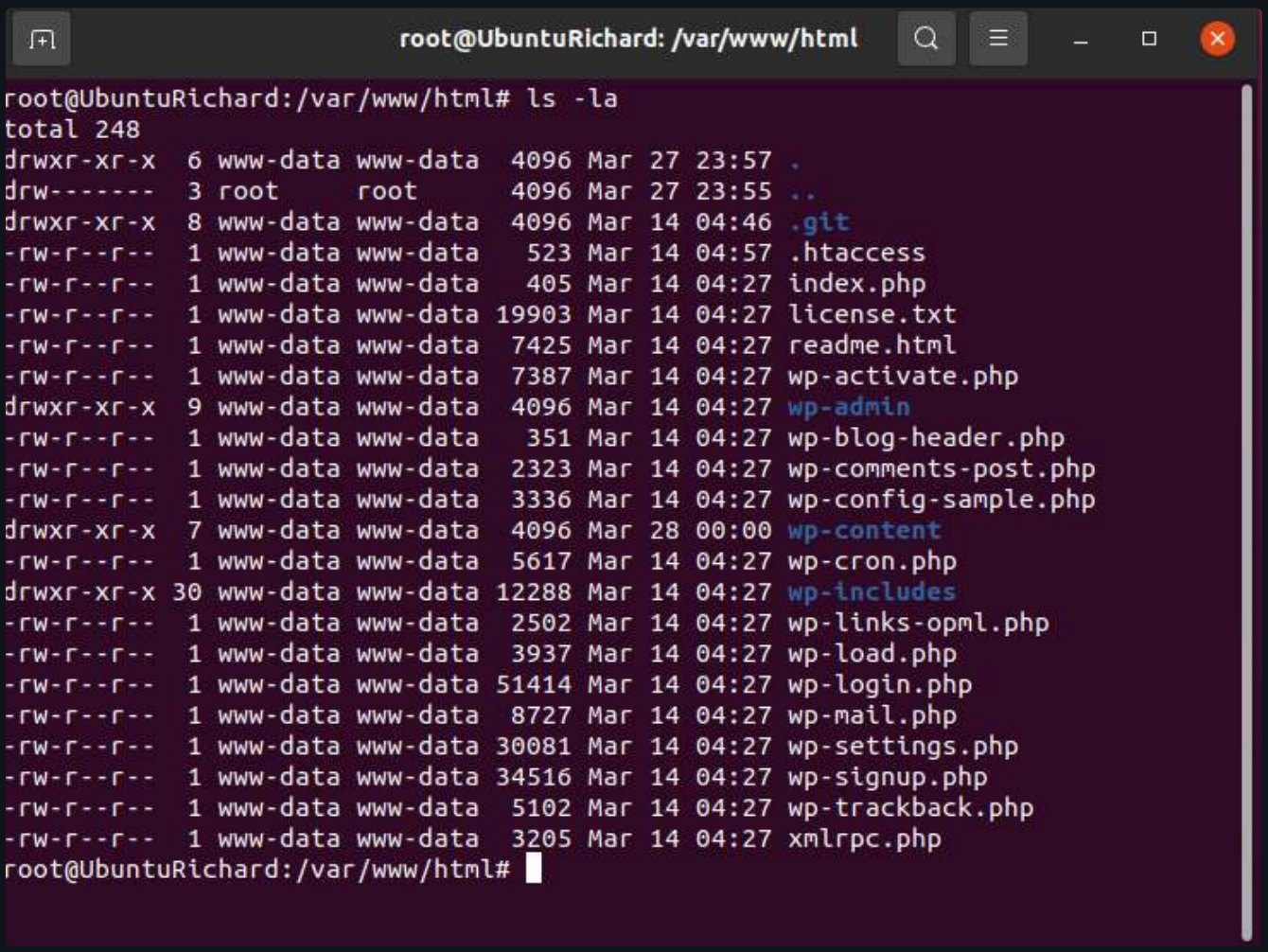

FIG 80 — Moving wp-config.php one directory above the web root so it is unreachable via the browser, applying least-privilege owner-only access.

HARDENING

wp-config.php: Configure and Validate

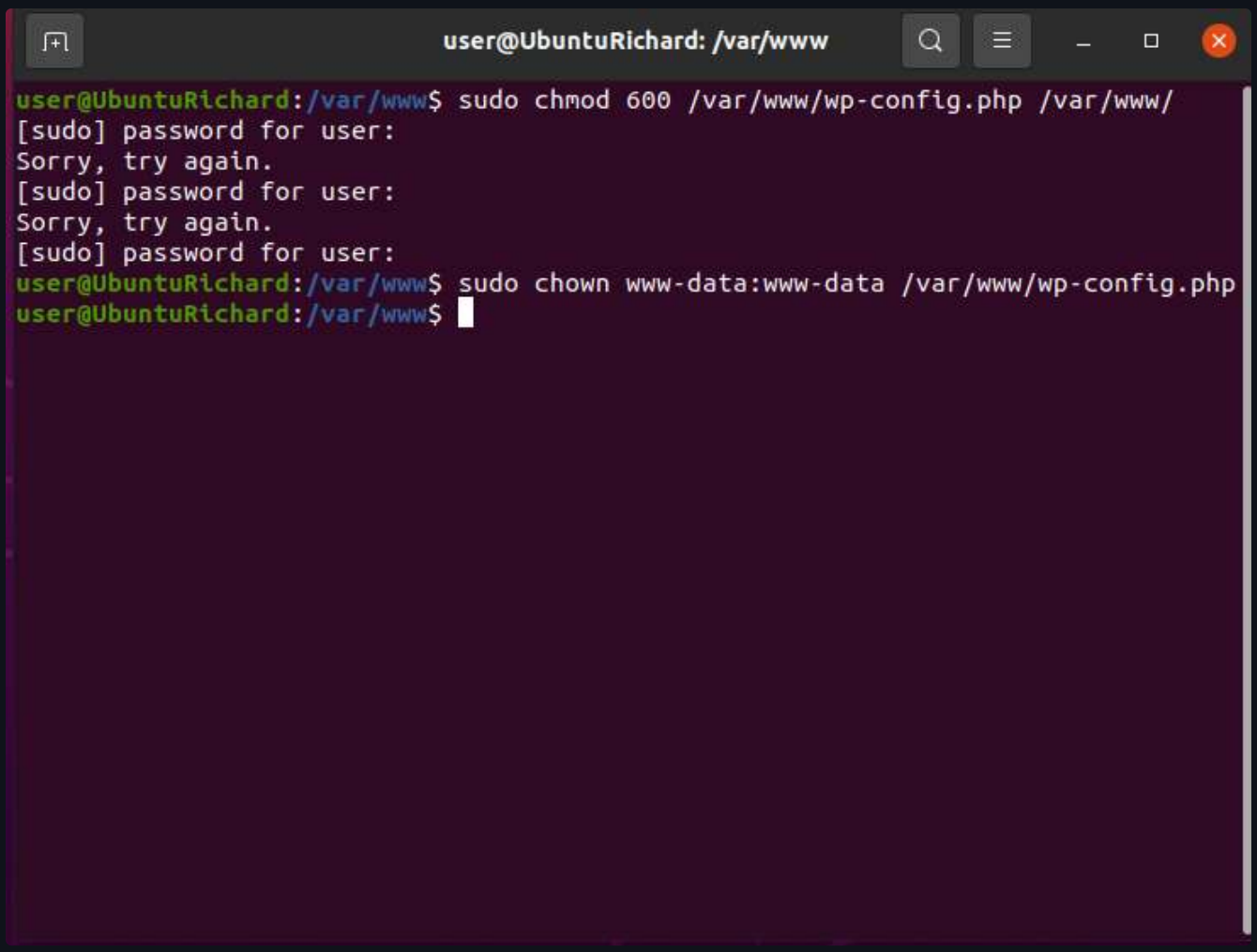

FIG 81 — Applying 600 (owner read/write only) permissions to the relocated wp-config.php to prevent database-credential exposure.

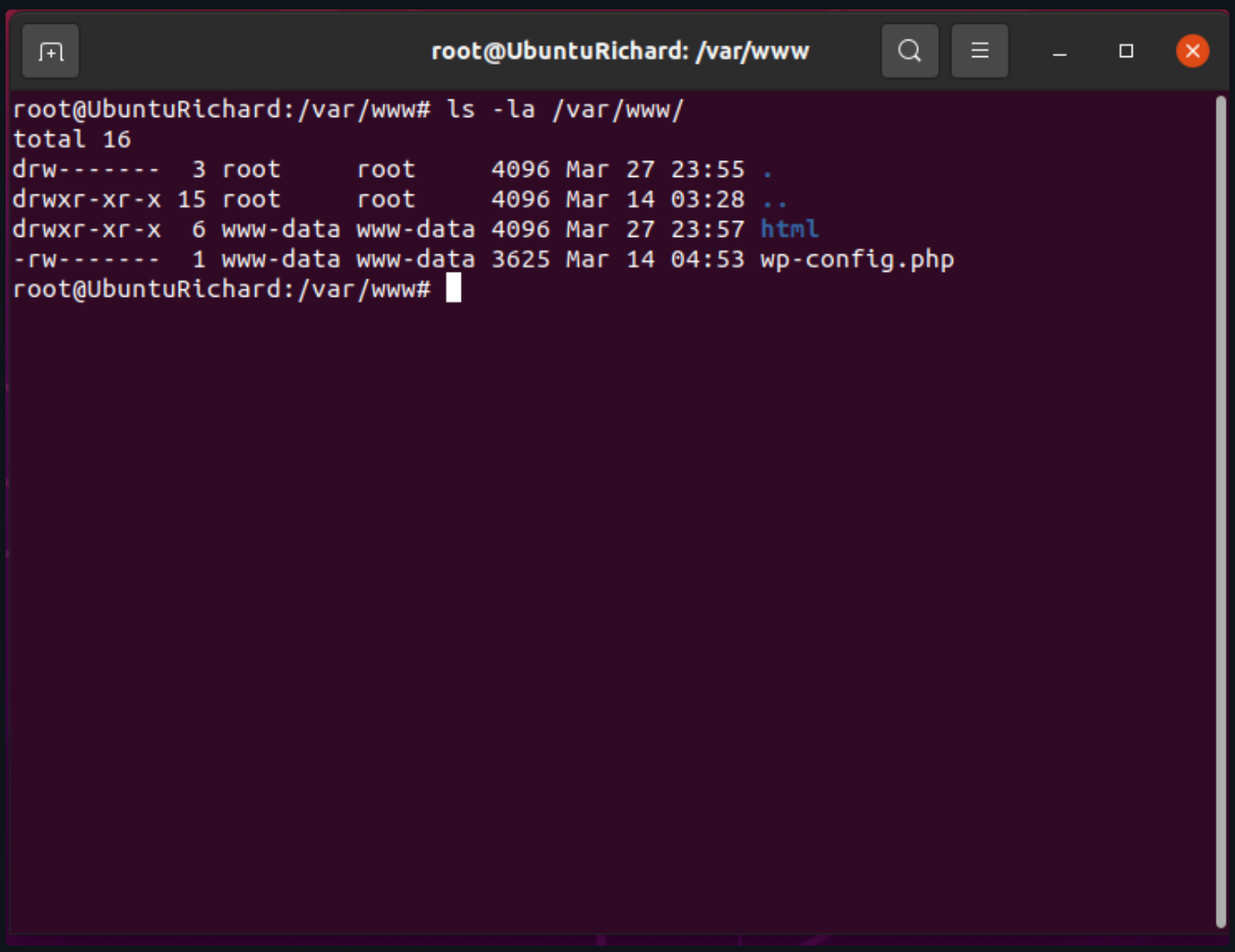

FIG 82 — Validation that wp-config.php now sits above the web root with 600 permissions, eliminating browser access to the credentials file.

HARDENING

Wordfence Layer-7 WAF

Layer three: installed the Wordfence application firewall to inspect and filter HTTP(S) traffic at OSI Layer 7.

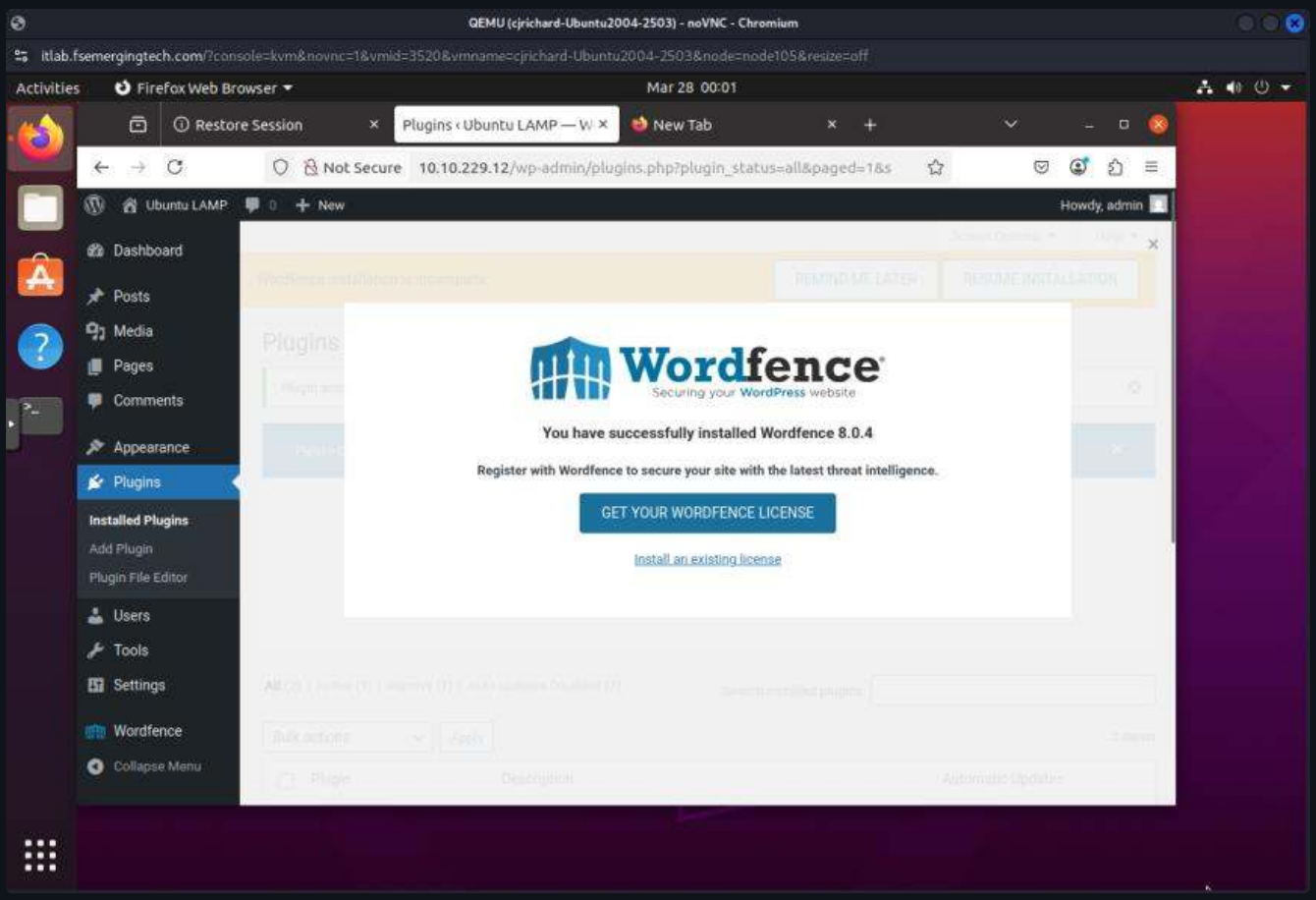

FIG 83 — Installing and configuring the Wordfence WAF plugin, adding Layer-7 inspection of HTTP(S) traffic for malicious requests.

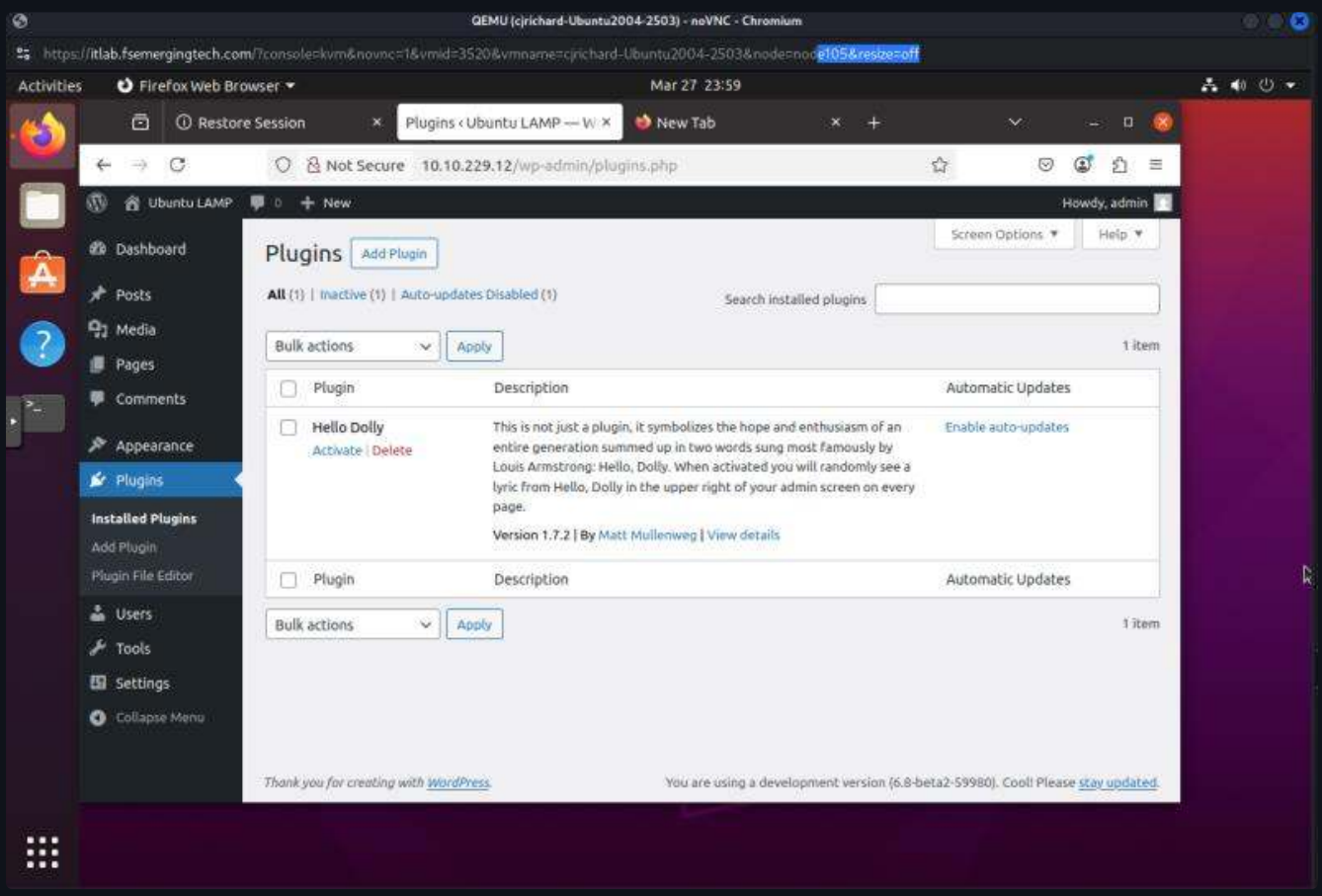

FIG 84 — Wordfence active and protecting against common application-layer attacks such as SQL injection, XSS, and brute-force logins.

CONCLUSION

Defense-in-Depth Summary

The conclusion ties the three layers together and documents resolving a 403 error and a redirect loop by correcting permissions and setting the site URL in wp-config.php.

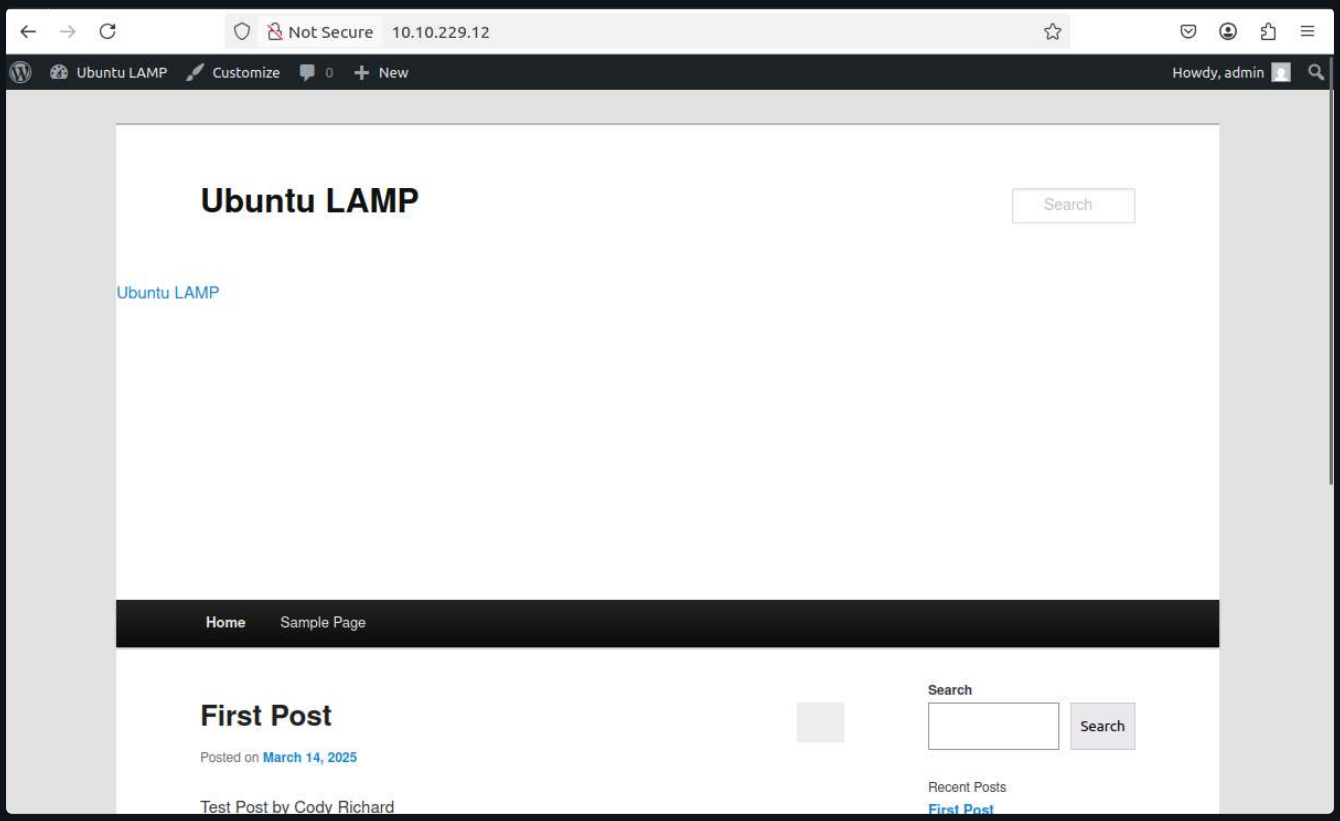

FIG 85 — Conclusion documenting the layered controls: 644/755 file and directory permissions, wp-config.php relocated to /var/www and set to 600, and Wordfence as a Layer-7 defense.

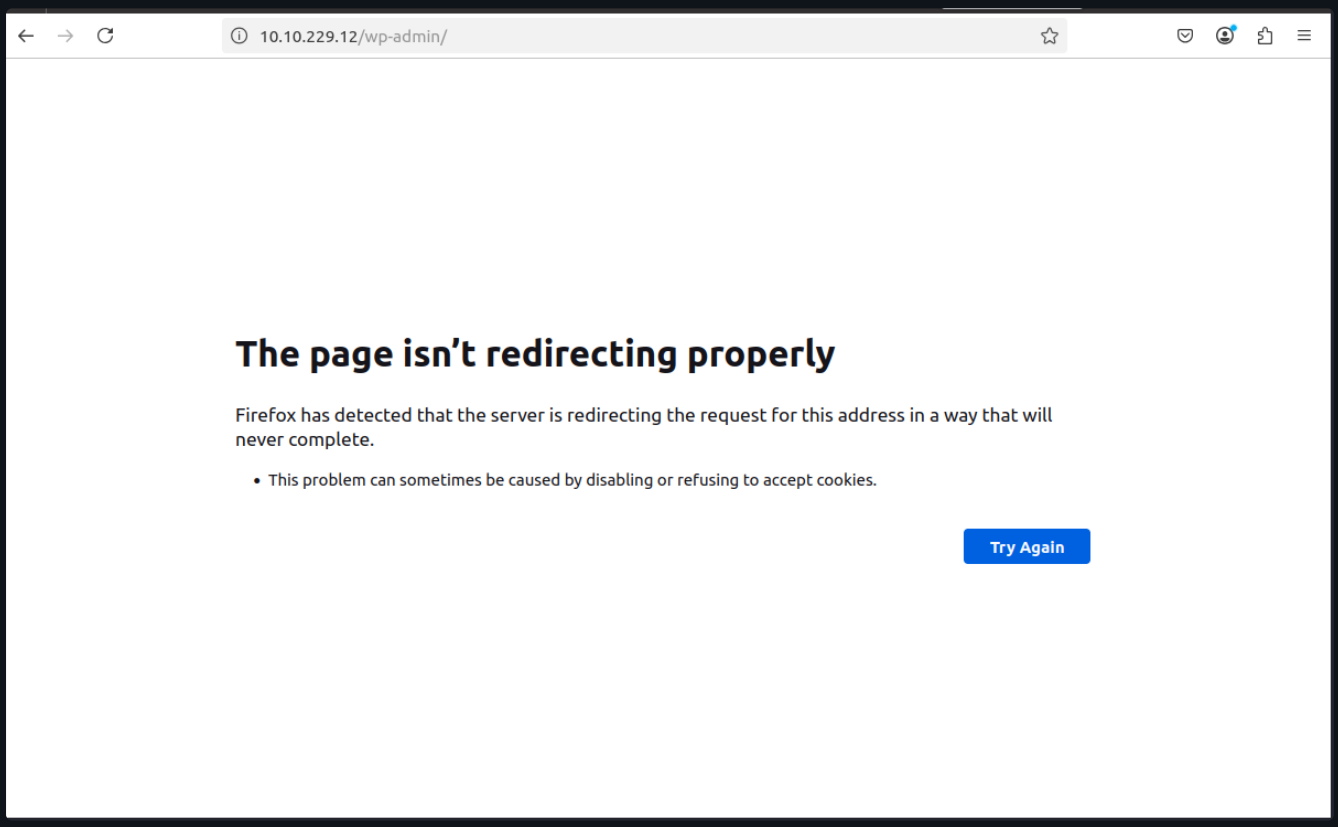

FIG 86 — Continued conclusion narrative on the defense-in-depth approach across file system, configuration, and application layers.

APPENDIX A

Nginx Config and Access Log

Reference appendices: the reverse-proxy configuration forwarding /blog to the Ghost backend, and the last 10 lines of the access log.

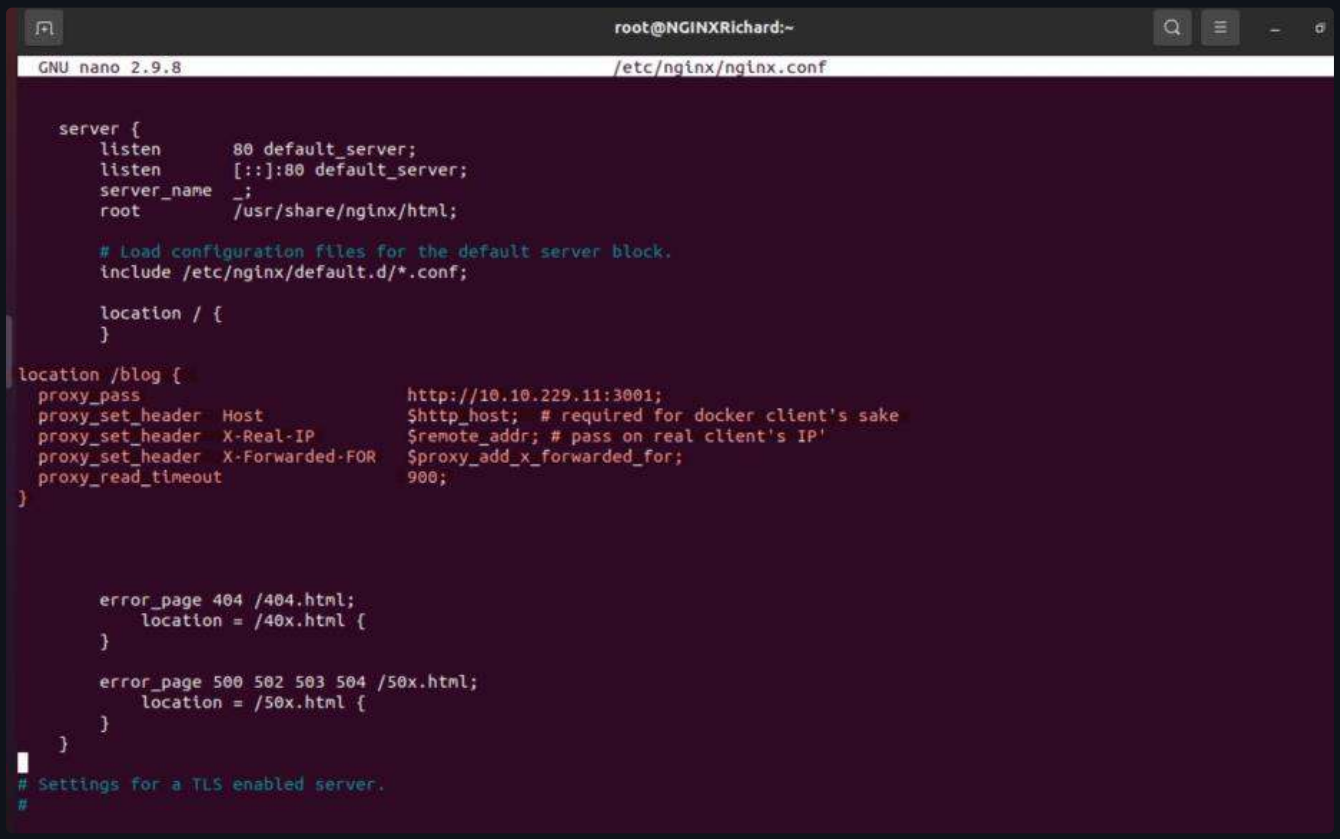

FIG 87 — Appendix A: /etc/nginx/nginx.conf showing the reverse proxy forwarding /blog requests to the Ghost backend.

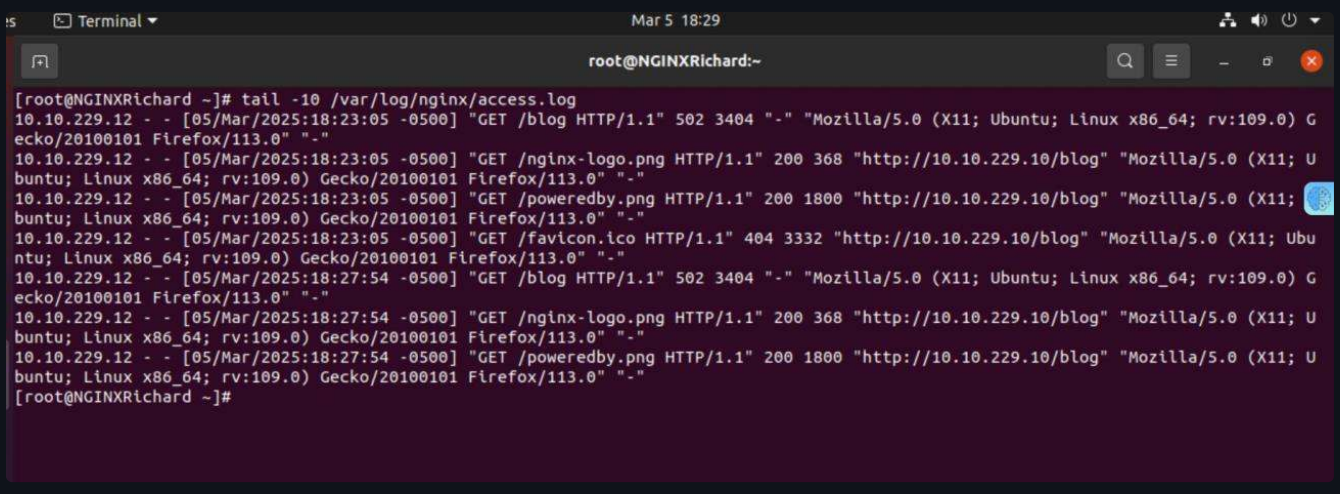

FIG 88 — Appendix B: tail -10 /var/log/nginx/access.log showing the most recent incoming requests to the Nginx server.

APPENDIX B

Nginx Error Log

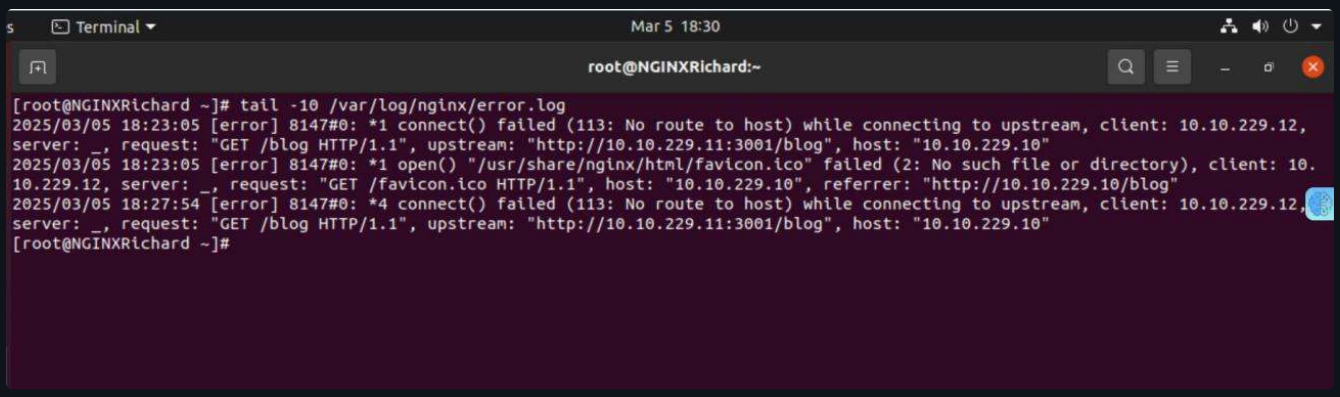

FIG 89 — tail -10 /var/log/nginx/error.log showing the last 10 error-log entries for misconfiguration and failed-request diagnostics.