LEMP STACK IMPLEMENTATION

**Github Link:** [**https://github.com/Macbeth111/Dev-Ops-.git**](https://github.com/Macbeth111/Dev-Ops-.git)

**LinkedIn:** [**https://www.linkedin.com/feed/update/urn:li:activity:7254893625946333184/**](https://www.linkedin.com/feed/update/urn:li:activity:7254893625946333184/)

[**PREREQUISITE: 2**](#_tainxf5l3yxv)

[**STEP 1: INSTALL THE NGINX WEB SERVER 3**](#_5hnolu8l09ar)

[##Install Nginx server 3](#_pg7bfkp6yq34)

[###Ensuring EC2 Was Receiving Traffic 4](#_ybzhpjmelhr9)

[###Checked Subnet Internet Access 4](#_evhvk89d73e1)

[###Allocated an Elastic IP 4](#_8uilhmfbas3w)

[###Configured Security Group and Network ACL 5](#_ghl6w5jm3n7e)

[###Tested Network Connectivity 5](#_vty8lr2hzzzv)

[##Install Nginx server 6](#_dlohzovbjptw)

[**STEP 2: INSTALL MYSQL SERVER 7**](#_2p375emet21j)

[## Install MySQL Server 8](#_83lbkpkrkd5h)

[## Log into the MySQL Console 8](#_f63eo7ylondz)

[## Set the Root User Password 8](#_7bboq3ngfxls)

[## Exit the MySQL Console 9](#_58pww6p9t9ux)

[## Run the MySQL Secure Installation Script 9](#_m0frnj187iqx)

[## Configure the VALIDATE PASSWORD PLUGIN 10](#_32ruxbbixo84)

[## Select the Level of Password Validation 10](#_l9ofq121862m)

[## Set and Confirm Root User Password 11](#_s8v6mmyfk58b)

[**STEP 3: INSTALL PHP SERVER 11**](#_aobcllylazia)

[**STEP 4: CONFIGURING NGINX TO USE PHP PROCESSOR 12**](#_ojo94kbtwcz2)

[##Create root directory and change ownership 12](#_h1mkg063zr9h)

[##Create new server block for your project 12](#_at4rmgt63slv)

[##Explanation of Directives and Location Blocks: 13](#_ok9363xvkfni)

[##Activate configuration 14](#_cqpmf3igsasm)

[##Disable default Nginx host and reload new configuration 15](#_eyljgrqfv39r)

[**STEP 5: TESTING PHP WITH NGINX 16**](#_8ksi46dfjo1c)

[##Configure Nginx to process PHP files 16](#_87jeowh3vkz3)

[**STEP 6: RETRIEVING DATA FROM MYSQL DATABASE WITH PHP 17**](#_1qedjb6aksc7)

[##Connect to MySQL with root privileges and create database 17](#_w4j65g8yphnl)

[##Create user and assign privileges 18](#_6rohw7u5mrxu)

[**WHAT I LEARNED, CHALLENGES I FACED AND HOW I OVERCAME THEM: 22**](#_auvcuplnlmnd)

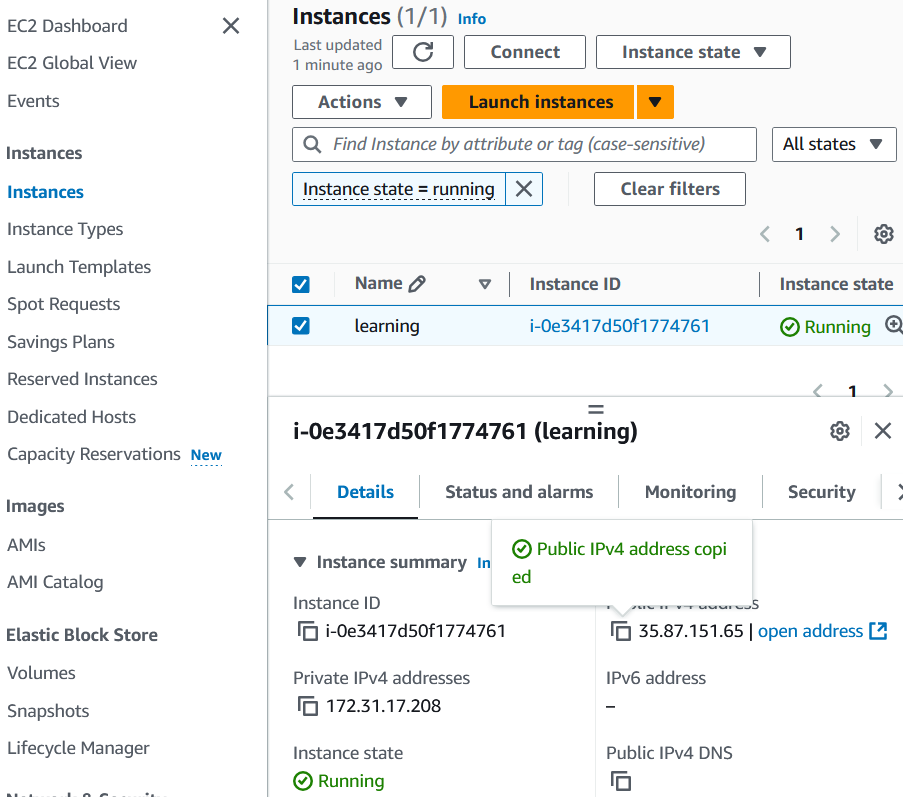
# **PREREQUISITE:**

Pre-requisite: Connecting to EC2 Instance via Gitbash

In this study, I initiated the process by connecting to my EC2 instance using Gitbash, a terminal emulator for Git that allows SSH connections. To connect, I followed these steps:

1. Accessing the EC2 instance: I selected my EC2 instance from the AWS Management Console, clicked on the "Connect" button, and chose the "SSH Client" option.
2. Running the SSH command: AWS provided a sample SSH command, which I executed in Gitbash as follows:

- ssh -i "your-key-pair-name.pem" *ubuntu@ec2-172-31-17-208.us-west-2.compute.amazonaws.com*

****

# 

# STEP 1: INSTALL THE NGINX WEB SERVER

## ##Install Nginx server

I first attempted to install the Nginx web server. The initial step in installing any new software on a Linux machine involves updating the system’s package list. This ensures that the most up-to-date versions of the required packages are installed.

*sudo apt update*

*sudo apt install nginx*

Unfortunately, the Nginx installation failed. To troubleshoot, I conducted a connectivity test by running the following command:

*ping google.com*

The result indicated a 100% packet loss, suggesting that the instance was not receiving traffic. Recognizing this as a networking issue, I embarked on problem-solving measures to rectify the situation.

### ###Ensuring EC2 Was Receiving Traffic

### ###Checked Subnet Internet Access

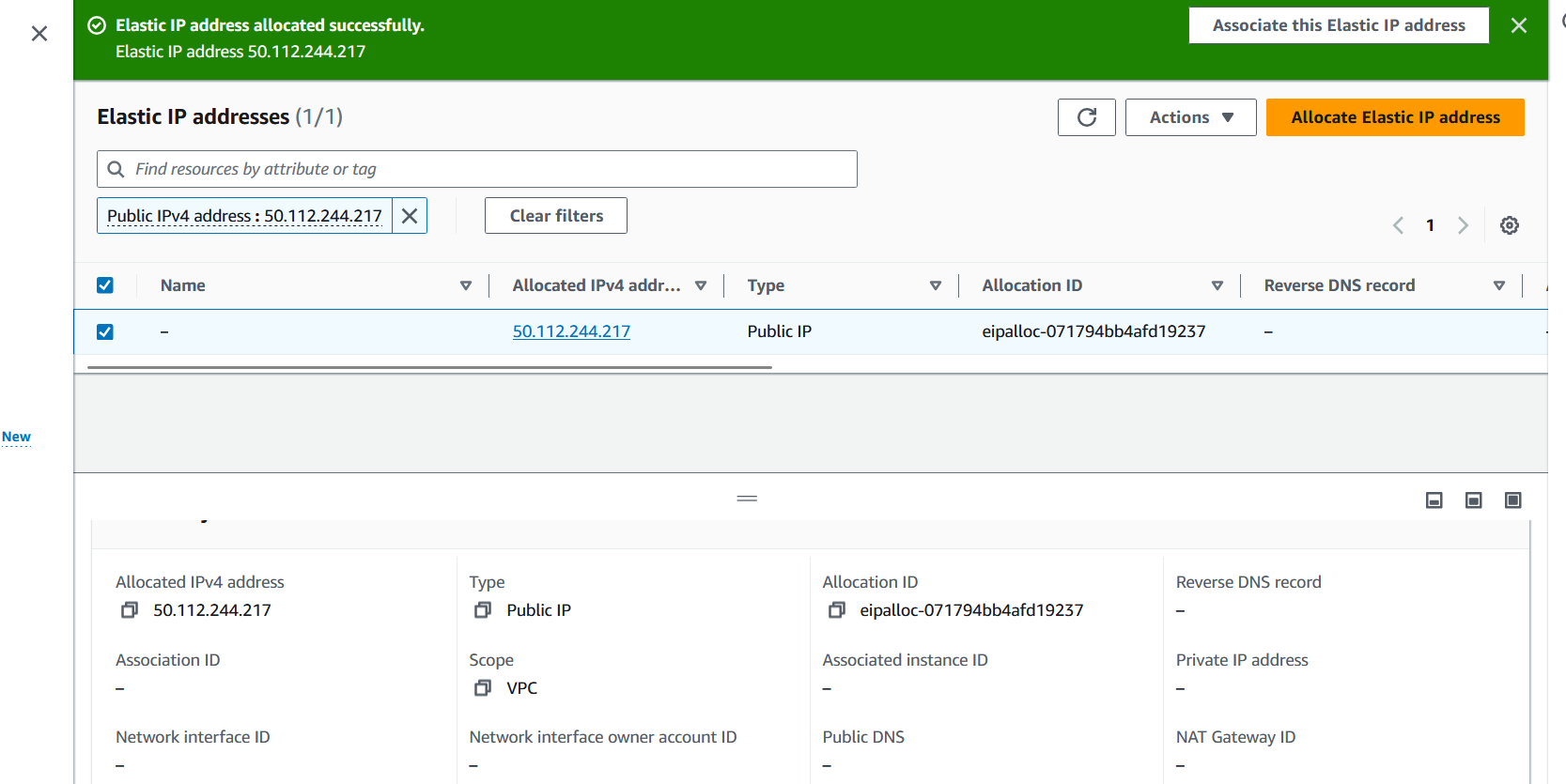
I began by checking if my EC2 instance was in a public subnet with proper internet access.

* **Verified the Public Subnet:**
  + I navigated to the **VPC Dashboard** in AWS.
  + Under the **Subnets** section, I checked the subnet my instance was in, ensuring it had a **Route Table** with a route to the **Internet Gateway (IGW)**.
  + I verified that the route table included a route directing 0.0.0.0/0 traffic to the Internet Gateway (IGW).
* **Checked the Internet Gateway:**
  + In the **VPC Dashboard**, I went to **Internet Gateways**.
  + I confirmed that the Internet Gateway was attached to the VPC where my instance resided.

### ###Allocated an Elastic IP

I realized that the EC2 Connect key was using the private IP address, and my instance didn’t have public internet access. This is where I needed to assign an **Elastic IP** to my instance to access it properly over the internet.

* **Checked for Elastic IP:**
  + I went to the **EC2 Dashboard** and selected my instance.
  + Under the **Description** tab, I confirmed that there was no **Public IPv4 address** assigned, which explained why I couldn’t access the instance from outside.
* **Allocated an Elastic IP:**
  + I went to **Elastic IPs** under the EC2 Dashboard.
  + I clicked on **Allocate Elastic IP Address** and followed the steps to allocate one.
  + Once I had the Elastic IP, I **associated** it with my EC2 instance.
* **Reason for Elastic IP:**
  + Without a public IP, the EC2 Connect key was only using the **private IP**, which meant that traffic couldn’t reach the instance from outside the VPC.



### ###Configured Security Group and Network ACL

Once the Elastic IP was allocated, I verified that both **inbound** and **outbound traffic** were allowed through the security groups and network ACLs.

**Step-by-Step Process:**

* **Checked Outbound Rules in Security Groups:**
  + In the **EC2 Dashboard**, I went to **Security Groups**.
  + I confirmed that the **outbound rules** allowed all traffic to 0.0.0.0/0 on all ports.
* **Checked Network Access Control List (NACL):**
  + I went to the **VPC Dashboard** and checked the **NACL** for the subnet where my instance was located.
  + I verified that both **inbound** and **outbound rules** allowed traffic from 0.0.0.0/0 for all ports.

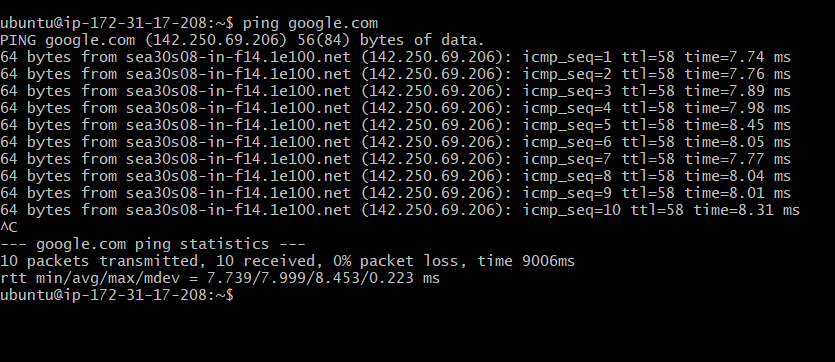
### ###Tested Network Connectivity

After configuring the subnet and assigning the Elastic IP, I tested the connectivity to ensure everything was working.

**Step-by-Step Process:**

* **Pinged a website:**
  + I ran the following command to ping a known external site (Google) from my instance:

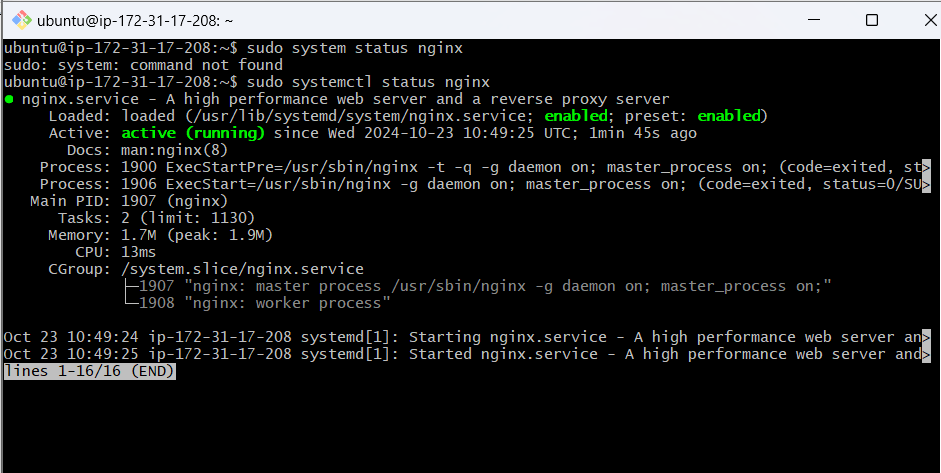
*ping google.com*

**

## ##Install Nginx server

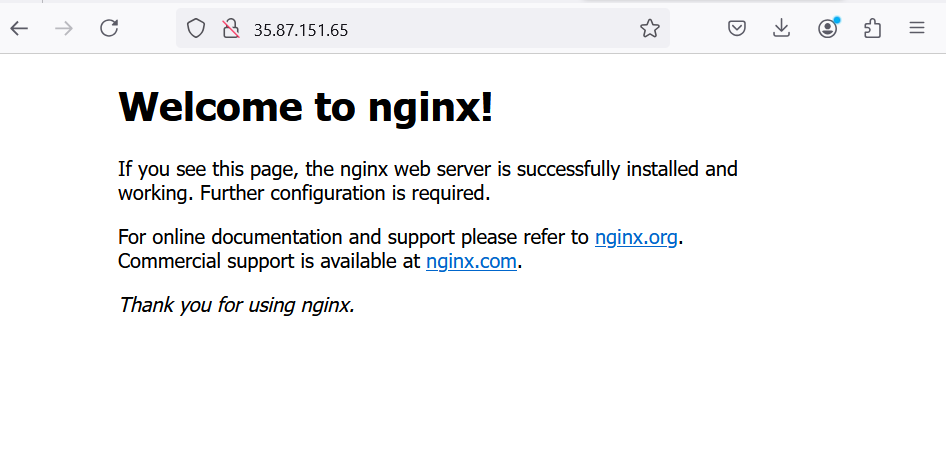
Now I run the following codes to update linux, install nginx and ensure that nginx was instaled, active and running ;

* *sudo apt update*
* *sudo apt install nginx*
* *sudo systemctl status nginx*

**

Now, I checked if my nginx server could respond to request from the internet using

[**https://35.87.151.65:80**](https://35.87.151.65:80)

****

# STEP 2: INSTALL MYSQL SERVER

Here is a step-by-step guide for installing MySQL on a Linux-based system.

To start, I installed the MySQL server using the following command:

## ## Install MySQL Server

* *$ sudo apt install mysql-server*

Here, sudo gives me administrator privileges to install software, apt is the package manager used by Debian-based systems (like Ubuntu), and mysql-server is the actual package I'm installing. When prompted, I confirm the installation by typing y and pressing ENTER.

## ## Log into the MySQL Console

Once MySQL is installed, I log into the MySQL console to begin configuring the server:

* *$ sudo mysql*

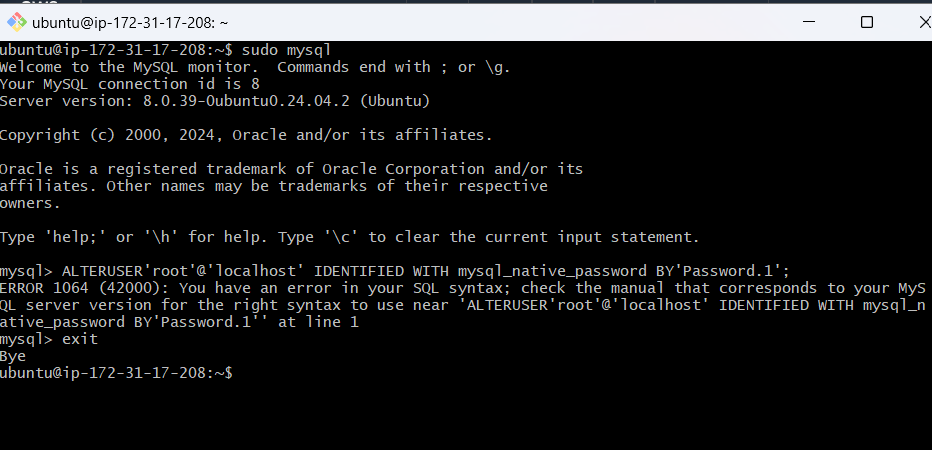
This command starts the MySQL client. Since I use sudo, I’m logging in as the root user (the administrative user for MySQL), giving me full access to the server’s management interface.

## ## Set the Root User Password

Next, I secure my MySQL installation by setting a password for the root user. Inside the MySQL console, I run this command:

* *ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql\_native\_password BY 'PassWord.1';*

This command changes the password for the root user (root), who can connect only from the local machine (localhost). The password is set to 'PassWord.1'. It's important to set a strong password at this stage to prevent unauthorized access to the server.



## ## Exit the MySQL Console

After setting the root password, I exit the MySQL console by typing:

* *mysql> exit*

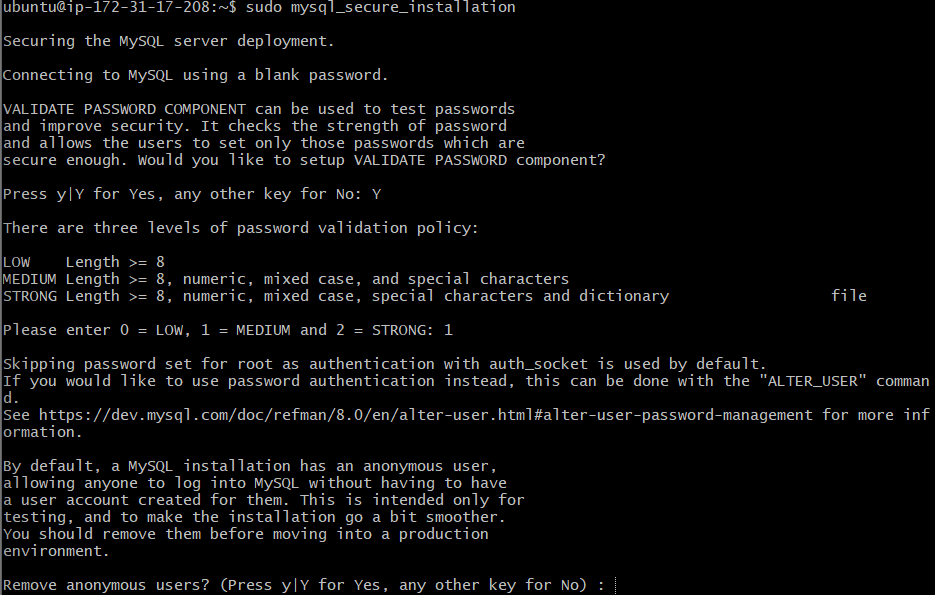
Once the root password is set, the user exits the MySQL console by typing exit. This closes the connection to the MySQL server and returns the user to the regular terminal prompt.

## ## Run the MySQL Secure Installation Script

To further enhance security, I run the secure installation script

* *$ sudo mysql\_secure\_installation*

This script performs several tasks, such as removing anonymous users and disabling remote root login. It’s a crucial step for locking down unnecessary features and improving the overall security of MySQL. I recommend accepting the default options for most questions during this process.



## ## Configure the VALIDATE PASSWORD PLUGIN

During the secure installation script, I’m asked if I want to enable the VALIDATE PASSWORD PLUGIN. This plugin enforces stronger passwords by testing their strength. I choose whether to enable it by typing:

*Press y|Y for Yes, any other key for No:*

This plugin is optional but highly recommended for production environments. It ensures that any passwords I create in the future meet certain strength criteria.

## ## Select the Level of Password Validation

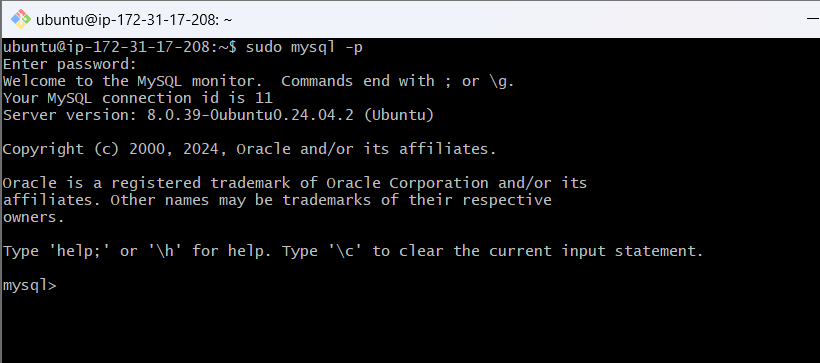
If I enable the plugin, I’m prompted to choose the level of password validation. The three levels are:

1. **LOW**: Requires passwords to be at least 8 characters long.
2. **MEDIUM**: Requires passwords to be at least 8 characters long, with a mix of numeric digits, uppercase and lowercase letters, and special characters.
3. **STRONG**: Requires all of the above and prevents passwords from being common dictionary words.

## ## Set and Confirm Root User Password

Regardless of whether I enabled the password validation plugin, the script asks me to set and confirm the password for the **MySQL root user**. This should not be confused with the system’s root user.

Setting a strong password for the MySQL root user ensures that no one can access the MySQL server without authorization. Even though I set the password earlier, this step adds an extra layer of security. After setting the password, I used it to login to mySQL to ensure that it is working.



**NOW MYSQL IS WORKING PROPERLY**

# STEP 3: INSTALL PHP SERVER

**Here is a step-by-step guide for installing PHP on a Linux-based system.**

**To start, I install the PHP server using the following command:**

* *$ sudo apt install php-fpm php-mysql*

# STEP 4: CONFIGURING NGINX TO USE PHP PROCESSOR

In this step, I’ll be configuring the Nginx web server to use the PHP processor. By setting up server blocks (similar to virtual hosts in Apache), I can host multiple domains on the same server. For the purposes of this guide, I’ll use projectLEMP as the domain name example.

Since I’m using Ubuntu 20.04, I know that Nginx has a single server block enabled by default, serving files from the /var/www/html directory. While this works well for one site, managing multiple sites would be difficult. Instead of modifying /var/www/html, I’ll create a new directory for my domain within /var/www/, leaving /var/www/html as the default fallback directory for any unmatched requests.

## ##Create root directory and change ownership

First, I created the root web directory for my domain as follows:

* *$ sudo mkdir /var/www/projectLEMP*

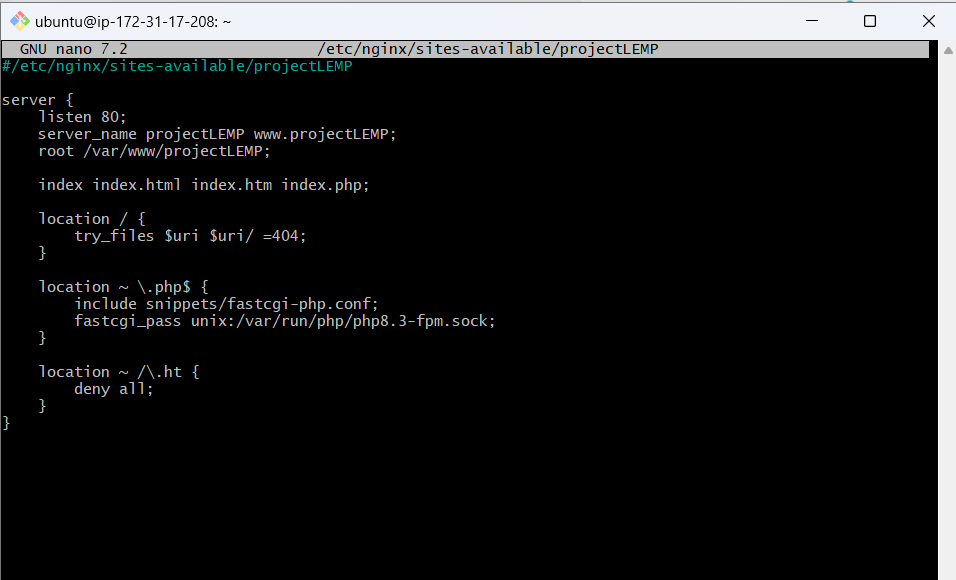
Next, I changed the ownership of the directory to my current system user by using the $USER environment variable:

* *$ sudo chown -R $USER:$USER /var/www/projectLEMP*

## ##Create new server block for your project

Then, I created a new server block configuration file for **projectLEMP** in Nginx's sites-available directory. I’ll use nano as my text editor:

* *$ sudo nano /etc/nginx/sites-available/projectLEMP*

**

## ##Explanation of Directives and Location Blocks:

1. **listen 80**:

o This tells Nginx to listen for incoming HTTP connections on port 80 (the default HTTP port).

1. **server\_name projectLEMP www.projectLEMP**:

o Specifies the domain names (projectLEMP and www.projectLEMP) that this server block should respond to.

1. **root /var/www/projectLEMP**:

o Defines the root directory where the site’s files are located. All requests will look for files in /var/www/projectLEMP.

1. **index index.html index.htm index.php**:

o Defines the default index files to serve when accessing a directory. If index.html isn't found, it will try index.htm, and then index.php.

1. **location / { try\_files $uri $uri/ =404; }**:

o This block handles all requests to the root ("/").

o try\_files $uri $uri/ =404; tries to serve the requested URI as a file. If the file doesn’t exist, it tries to find a directory. If neither is found, it returns a 404 error.

1. **location ~ .php$ { include snippets/fastcgi-php.conf; fastcgi\_pass unix:/var/run/php/php8.3-fpm.sock; }**:

o This block matches PHP files (with .php extension).

o include snippets/fastcgi-php.conf; loads configuration for processing PHP files.

o fastcgi\_pass unix:/var/run/php/php8.3-fpm.sock; passes PHP requests to the PHP FastCGI processor via a UNIX socket.

1. **location ~ /.ht { deny all; }**:

o This block denies access to any files starting with .ht (like .htaccess). This is a security measure to prevent exposure of sensitive files.

By doing this, I set up Nginx to host my **projectLEMP** domain, ensuring that it doesn’t interfere with any existing sites or configurations.

## ##Activate configuration

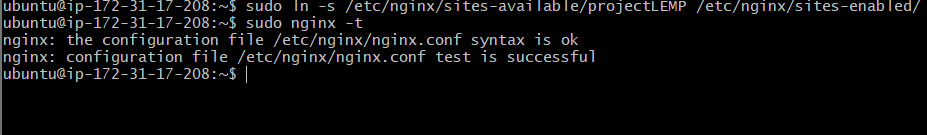
Afterwards, I went ahead to activate the configuration I had created for my Nginx server. To do this, I created a symbolic link from the configuration file in the sites-available directory to the sites-enabled directory. This step was necessary because Nginx reads configuration files from sites-enabled, and linking my custom configuration ensured it would be loaded the next time Nginx restarted. I used the code:

* *sudo ln -s /etc/nginx/sites-available/projectLEMP /etc/nginx/sites-enabled/*

After linking the configuration, I wanted to ensure everything was set up correctly without any syntax errors. I ran the Nginx syntax test command:

* *sudo nginx -t*

The output confirmed that the syntax was correct and the test was successful, so I knew the configuration was ready for use:



## ##Disable default Nginx host and reload new configuration

Next, I needed to disable the default Nginx host, which was currently listening on port 80. This was important because it could conflict with my newly created configuration if both were active. To disable the default configuration, I ran:

* *sudo unlink /etc/nginx/sites-enabled/default*

With the default configuration removed, I reloaded Nginx to apply the changes and load the new configuration:

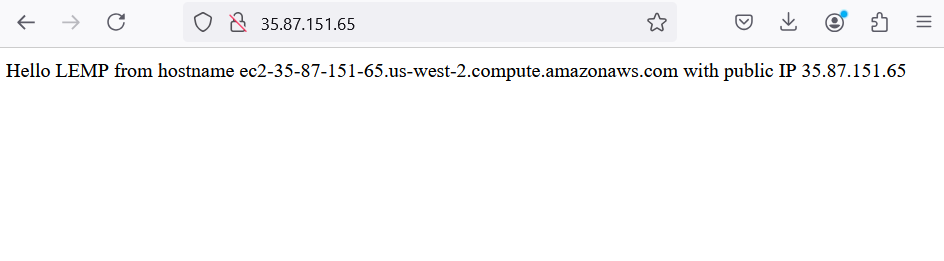
* *sudo systemctl reload nginx*

At this point, my new server block was active, but the web root directory (/var/www/projectLEMP) was still empty. To test that the server was functioning correctly, I created a simple index.html file. This file would serve as a placeholder to verify that the server was properly responding to requests.

* *TOKEN=$(curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-metadata-token-ttl-seconds: 21600")*

*sudo echo "Hello LEMP from hostname $(curl -H "X-aws-ec2-metadata-token: $TOKEN" -s http://169.254.169.254/latest/meta-data/public-hostname) with public IP $(curl -H "X-aws-ec2-metadata-token: $TOKEN" -s http://169.254.169.254/latest/meta-data/public-ipv4)" > /var/www/projectLEMP/index.html*

This command created an index.html file in the root directory, which displayed a message with the hostname and public IP address of the server. This allowed me to verify that the new server block was working as expected by visiting the website in a browser. By the end of this process, I had successfully configured PHP to work with Nginx, ensured the configuration was error-free, and set up a basic placeholder page to verify that the server was functioning correctly.



**AT THIS POINT LEMP WAS FULLY CONFIGURED**

# 

# STEP 5: TESTING PHP WITH NGINX

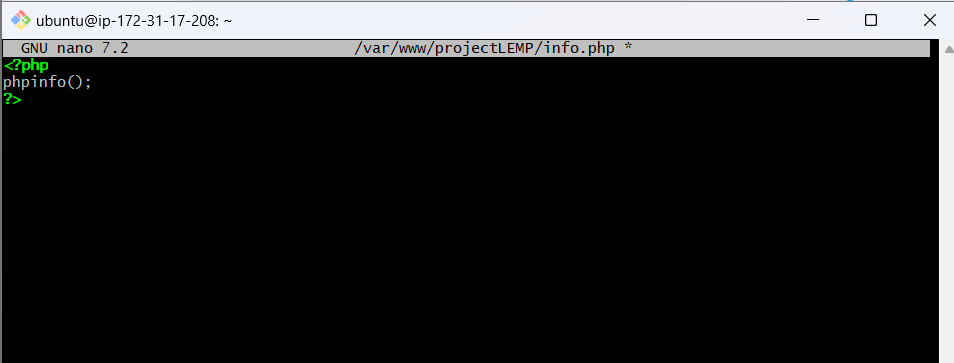
## ##Configure Nginx to process PHP files

Once I had completed setting up my LEMP stack, I needed to test if the installation was fully operational. Specifically, I wanted to confirm that Nginx could correctly process PHP files by handing them off to the PHP processor.

To do this, I decided to create a simple test PHP file. I opened my text editor and created a new file called info.php in the document root of my web directory:

* *nano /var/www/projectLEMP/info.php*

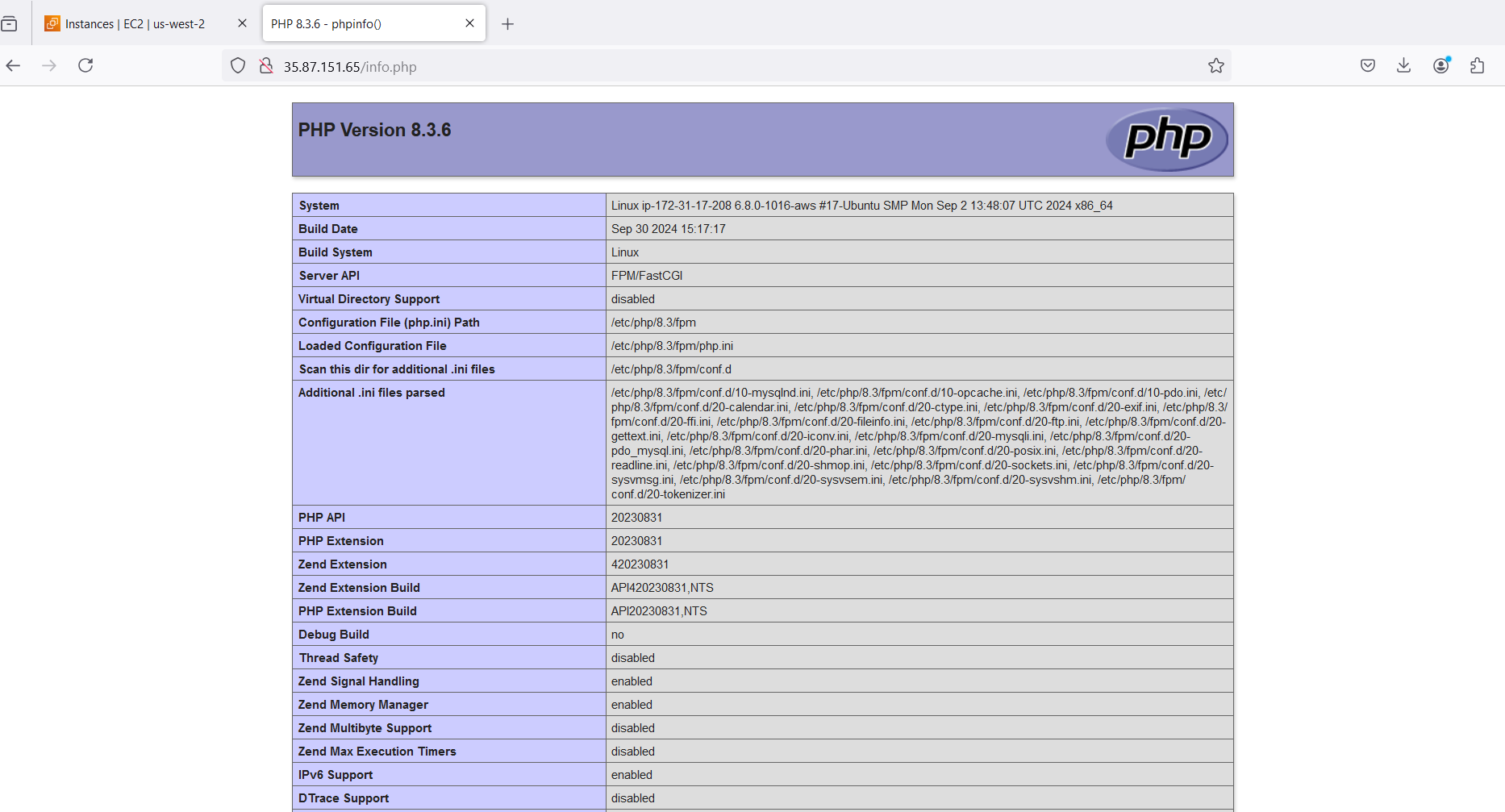
In this file, I added a small block of PHP code that would output information about my server’s PHP configuration when accessed. This would serve as a useful test to see if the PHP processor was working correctly:



After saving the file, I accessed it via my web browser by navigating to the domain name or public IP address of my server, followed by /info.php. The URL looked something like this:

[*http://35.87.151.65/info.php*](http://35.87.151.65/info.php)

When I visited this page, it displayed a detailed PHP information page, which confirmed that Nginx was successfully processing PHP files through the FastCGI processor. This output included all the necessary details about the PHP version, loaded modules, and configuration settings. This step was crucial because it validated that the PHP processor was working as expected, ensuring that future PHP applications or scripts would run without issues on my server. Having this confirmation gave me confidence that my LEMP stack was fully set up and ready for use.



# STEP 6: RETRIEVING DATA FROM MYSQL DATABASE WITH PHP

To retrieve data from MySQL using PHP, I followed a structured process. I first created a test database and configured it for access through an Nginx website. This would allow the site to query data from the database and display it effectively.

## ##Connect to MySQL with root privileges and create database

I started by connecting to the MySQL console with root privileges. This provided me the necessary administrative access to create and manage databases and users.

* *$ sudo mysql*

Once in the MySQL console, I created a new database named example\_database. You can replace this name with a different one if needed, but this was used for demonstration purposes.

* *mysql> CREATE DATABASE `mac\_database`;*

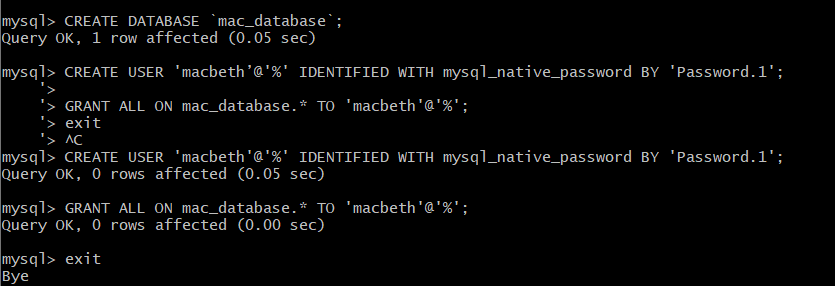
## ##Create user and assign privileges

With the database set up, the next task was to create a new user who would have full privileges over this newly created database. I opted for a user named example\_user. Since MySQL requires authentication, I used the mysql\_native\_password method for this user. For demonstration purposes, I assigned a password of 'Password.1', though it's recommended to choose a stronger, more secure password for real-world applications.

* *mysql> CREATE USER 'macbeth'@'%' IDENTIFIED WITH mysql\_native\_password BY 'Password.1';*

The final step was to grant this new user full permissions to manage the example\_database. This ensures the user can perform all actions such as querying, inserting, updating, and deleting data in the database.

* *mysql> GRANT ALL ON mac\_database.\* TO 'macbeth'@'%';*

**

After granting “macbeth” full privileges over the mac\_database, I exited the MySQL shell to test the new user credentials. This step ensures that the user has the correct permissions to access and manage the database.

To exit the MySQL shell, I ran the following command:

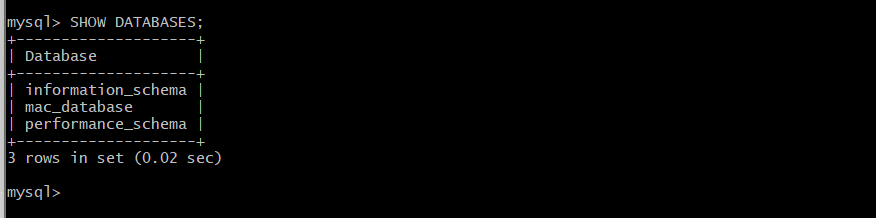
* *Mysql> exit*

Next, I logged back into the MySQL console using the newly created example\_user credentials. The -p flag in the command prompted me to enter the password I had set for this user.

* *$ mysql -u macbeth -p*

After entering the MySQL console, I verified that “macbeth” had access to the database by running the SHOW DATABASES; command. This command lists all the databases that the current user can interact with.

*mysql> SHOW DATABASES;*



The expected output confirmed that “macbeth” had access to mac\_database:

With access confirmed, I proceeded to create a table inside the example\_database. The table was named todo\_list, and it would store to-do items, each with a unique item\_id and a content field for the description of the to-do item. The following SQL command was used to create the table:

*CREATE TABLE mac\_database.todo\_list (*

*item\_id INT AUTO\_INCREMENT,*

*content VARCHAR(255),*

*PRIMARY KEY (item\_id)*

*);*

This setup established a basic structure for storing data, ensuring that the “macbeth” could manage the table and retrieve data using PHP later on. After creating the todo\_list table inside the mac\_database, I proceeded to insert some data into the table to test that everything was working correctly. Each row inserted represents a different to-do item, with the content field storing the item's description.

I used the following SQL command to insert the first row:

* *mysql> INSERT INTO mac\_database.todo\_list (content) VALUES ("My first important item");*

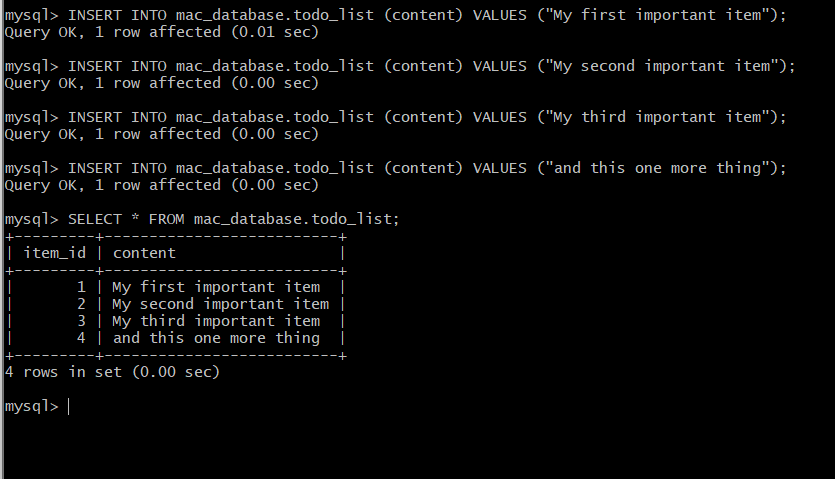
To add more test data, I repeated this command multiple times, changing the content each time. This allowed me to have multiple entries to test the functionality further.

For example, I ran similar commands to add more items:

* *mysql> INSERT INTO mac\_database.todo\_list (content) VALUES ("My second important item");*
* *mysql> INSERT INTO mac\_database.todo\_list (content) VALUES ("My third important item");*
* *mysql> INSERT INTO mac\_database.todo\_list (content) VALUES ("and this one more thing");*

After inserting the data, I wanted to confirm that everything was saved correctly. I used the SELECT command to retrieve and display all the entries from the todo\_list table:

* *mysql> SELECT \* FROM mac\_database.todo\_list;*

**

Seeing this output confirmed that the data had been successfully saved to the database.

After verifying the data, I exited the MySQL console with the following command:

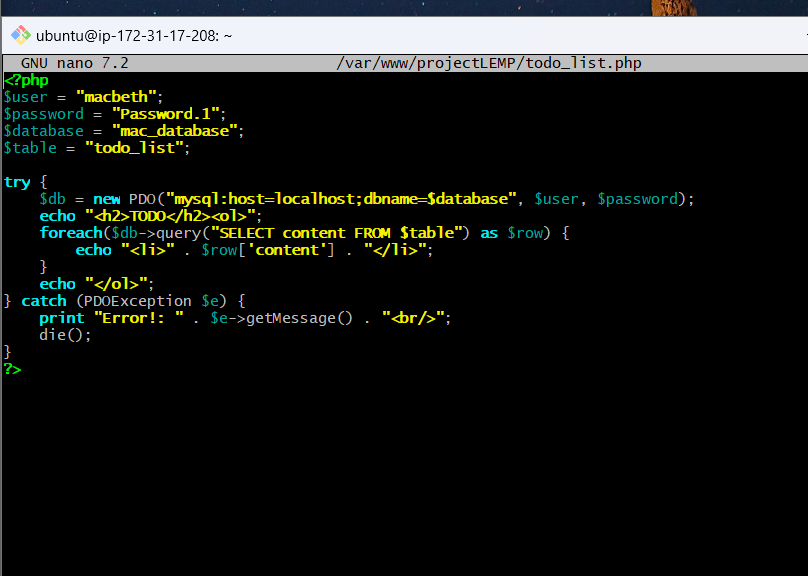
* *mysql> exit*

At this point, I had successfully created a table, inserted data into it, and retrieved the data, confirming that everything was functioning correctly.

After exiting mysql, I created a PHP file using the following code:

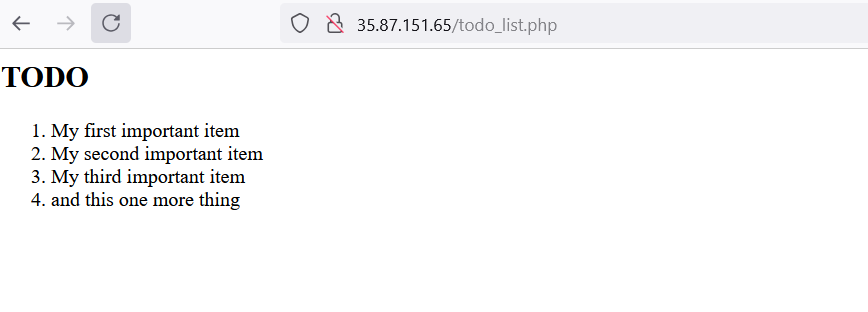
* *nano /var/www/projectLEMP/todo\_list.php*

where I inserted the PHP script to connect to MySQL and query content from the todo\_list table:



Afterwards, I saved and exited the nano page.

Then I went back to my web-browser and accessed the PHP page to see if the changes had taken place. The results indicate that I had successfully configured PHP to retrieve information from MySQL.



# WHAT I LEARNED, CHALLENGES I FACED AND HOW I OVERCAME THEM:

Throughout the process of implementing the LEMP stack, I gained significant insight into the critical components of Linux, Nginx, MySQL, and PHP, which form the foundation for creating a robust web server environment. I learned how to install each of these components, ensuring they work seamlessly together to deliver web applications efficiently. By starting with setting up an Ubuntu server, I developed my Linux command-line skills, followed by configuring Nginx as a web server to handle requests. Understanding how to secure the MySQL database, manage databases, and connect them to PHP was a valuable experience that reinforced my knowledge of database management and server-side scripting. The hands-on approach enabled me to troubleshoot configuration issues and deepen my understanding of server environments, which has been instrumental in developing a comprehensive view of web hosting architecture.

However, the process was not without its challenges. One major difficulty I faced was troubleshooting server errors, particularly when services failed to run after installation. Configuring the firewall to allow HTTP/HTTPS traffic and resolving issues with PHP and MySQL connectivity required detailed research and trial and error. Despite these challenges, I employed self-study and problem-solving strategies, such as consulting online documentation and forums, to overcome them. This experience not only enhanced my technical skills but also improved my ability to work independently and resolve technical issues, thereby reinforcing my confidence in implementing complex server environments.