Sure! Here's a **simple explanation** of what Andrew Mallett is demonstrating in this part of the course:

**🛠️ What’s being done here?**

You're learning how to **set up an SSH server** on one virtual machine (ubuntu2) and use another machine (ubuntu1) to **connect to it remotely** using SSH.

This is a hands-on guide where:

* ubuntu2 is the **server** (where you’ll configure SSH)
* ubuntu1 is the **client** (where you’ll connect from)

**🧑‍💻 What are the steps being shown?**

1. **Start both machines** – ubuntu1 (client) and ubuntu2 (server) must be running.
2. **Create a new user** on ubuntu2:
   * Command: sudo useradd -m -s /bin/bash tux
   * This creates a user named tux with a home directory and bash shell.
3. **Set a password** for the user:
   * Command: sudo passwd tux

**🔐 Check and configure SSH settings**

Now on ubuntu2, you're configuring the **SSH server** so it behaves the way you want.

**Check the current SSH config:**

* Command: sudo sshd -T | grep -E '(password|permit)'
  + This shows current settings for:
    - PasswordAuthentication – allows logging in with a password
    - PermitRootLogin – controls whether root can log in via SSH

**What do we want?**

* Allow password logins for now (PasswordAuthentication yes)
* **Block root login** completely (PermitRootLogin no), for better security

**Edit the SSH config file:**

* File: /etc/ssh/sshd\_config
* Use a text editor to:
  + Make sure PasswordAuthentication yes is set (so you can log in as tux)
  + Change PermitRootLogin to no (to stop root logins)

**Restart SSH to apply changes:**

* Command: sudo systemctl restart sshd

Now your SSH server (ubuntu2) is configured safely to:

* Let user tux log in with a password
* **Not** allow root logins at all

**✅ What’s next?**

You’ll switch to ubuntu1 (client) to test connecting to ubuntu2 using SSH and eventually set up **key-based authentication** (so you won’t need passwords anymore).

**💡 In simple terms:**

You're getting your ubuntu2 system ready so someone (you!) can **remotely log in** using SSH as a safe, normal user (tux) — **not** as root, and with password login allowed just for now. Later, you'll improve security further by using SSH keys.

Let me know if you'd like help setting up or testing the SSH connection from ubuntu1!

**🧠 First, What is DNS?**

DNS stands for **Domain Name System**.

Think of DNS like the **contacts app** on your phone. Instead of remembering phone numbers, you save names like "Mom" or "Pizza Hut." DNS does the same for the internet:

* You type google.com
* DNS finds its IP address (like 142.250.180.78)
* Your computer uses the IP address to talk to Google's servers

**📘 What Are We Learning in This Lesson?**

You're learning how to:

1. **Understand how DNS works on Ubuntu 20.04**
2. **Turn your Ubuntu machine into a DNS server**
3. **Make another Ubuntu machine use your DNS server**
4. **Check DNS settings**
5. **Change which DNS servers Ubuntu uses**

**🖥️ Ubuntu’s Default DNS Behavior**

Every Ubuntu 20.04 system already has a small **built-in DNS server** using a service called systemd-resolved.

**Here's what it does:**

* Listens on 127.0.0.53 (that’s your local machine only)
* Uses **port 53**, which is the default DNS port
* Helps your system resolve names like ubuntu.com to IP addresses

**Files & Tools involved:**

* /etc/resolv.conf: This file shows which DNS server your computer is using
* resolvectl: A command that shows detailed DNS info (like IPs, DNS servers used, etc.)
* dig: A tool you can install and use to test DNS lookups

**🔧 Goal: Make Ubuntu2 a Real DNS Server**

**You have 2 machines:**

* **Ubuntu1** = DNS client
* **Ubuntu2** = DNS server (it gives answers to Ubuntu1)

We’ll install a DNS server software called **BIND9** on Ubuntu2.

**⚙️ Step-by-Step Process (Simplified)**

**✅ Step 1: Understand What You Already Have**

On Ubuntu, run this command to check what DNS is doing:

bash

CopyEdit

resolvectl status

You'll see:

* Which DNS servers you're using
* Which interfaces (like eth0) are using them

You can also check:

bash

CopyEdit

cat /etc/resolv.conf

⚠️ This file is a *symlink*, meaning it points to another file. Ubuntu uses it to store current DNS settings.

**✅ Step 2: Test DNS Resolution**

Use the dig command like this:

bash

CopyEdit

dig google.com

It shows:

* Which DNS server was used
* How fast the response was
* The IP address of google.com

**🎯 End Goal: What are we trying to do?**

We want to:

✅ **Set up a secure DNS server** on **Ubuntu 2**,  
✅ So that only **trusted machines** (like Ubuntu 1 and localhost) can ask it DNS questions (queries),  
✅ And **block everyone else** from using our DNS server.

**🧠 Think of it like this:**

You’re setting up a **phone directory** (DNS server) in your house (Ubuntu 2).

* You want your **family** (Ubuntu 1) to be able to look up phone numbers.
* But you **don’t want strangers** knocking on your door asking for phone numbers.

So, you create a **guest list** of people who are allowed to ask questions.

That guest list is called an **ACL (Access Control List)** — a list of IP addresses that are **trusted**.

Of course! 😊 Here's a **very simple explanation** of what you just read — summarizing the **end of your DNS module**:

**🧠 What Was This Module About?**

You’ve just learned **how DNS (Domain Name System)** works on Linux, and how to **set up your own DNS server**.

**✅ Here’s What You Learned in Simple Steps:**

**1. Systemd-resolved is your system’s default DNS service**

* Ubuntu already has a built-in DNS tool called systemd-resolved.
* You can **check it with:**
* resolvectl status

**2. You tested DNS lookups with dig**

* This command checks how names are being resolved (turned into IP addresses).
* If you only want the IP and not the full output:
* dig example.com +short

**3. You learned about /etc/resolv.conf**

* This file tells your system **which DNS servers to use**.
* But nowadays, it's often **just a shortcut (symbolic link)** to another file managed by systemd or Netplan.

**4. You installed your own DNS server (BIND9)**

* bind9 is a real DNS server, like what companies use.
* You learned to:
  + Install it
  + Configure it in /etc/bind/named.conf
  + Check its status with:
  + sudo systemctl status bind9

**5. You created your own DNS zones**

* A **zone** is like your **own little DNS database** (for example, .local domain).
* You created:
  + A **forward zone** (name → IP)
  + A **reverse zone** (IP → name)

**6. You did reverse lookups**

* Using this command:
* dig -x 192.168.56.101
* This tells you which hostname matches a given IP.

**🧾 Big Picture Summary:**

| **You did this** | **Why it's useful** |
| --- | --- |
| Used built-in DNS | Good for learning how Ubuntu handles DNS |
| Used dig | To test name resolution |
| Changed /etc/resolv.conf | To control which DNS your system uses |
| Installed BIND9 | So you have your **own DNS server** |
| Created zones | So you can **name your own computers** on your local network |
| Tested everything | To make sure it works! |

**🎉 What’s Next?**

You’re now ready to **use your DNS server** — for example, when setting up an **Apache web server**, which is coming in the next module.

That means you'll be able to:

* Host websites like ubuntu2.local
* Use DNS instead of typing IPs

Let me know if you want a **visual diagram** or a **cheat sheet** to summarize all of this. You're doing great! 🚀

**What is a DNS record and why do you need it?**

When you install Apache on your server (like **ubuntu2**), your web server has an **IP address** — for example: 192.168.1.100.

But IP addresses are hard to remember, so we use **domain names** (like www.local or myserver.local) to access the server.

**DNS records** link these domain names to IP addresses or other names.

**Common DNS records you’ll use for Apache:**

* **A record:** Links a domain name directly to an IP address.  
  Example:  
  ubuntu2.local → 192.168.1.100
* **CNAME record:** An alias that points one name to another name.  
  Example:  
  www.local → ubuntu2.local

**Example setup step-by-step:**

**1. You have Apache running on ubuntu2 with IP 192.168.1.100.**

**2. Set an A record in your DNS zone file:**

In your DNS server config (usually a file like /etc/bind/zones/db.local), add:

css

CopyEdit

ubuntu2.local. IN A 192.168.1.100

This means ubuntu2.local points to your server’s IP.

**3. Set a CNAME record so you can use www.local as an alias:**

In the same file, add:

lua

CopyEdit

www.local. IN CNAME ubuntu2.local.

This means www.local is just another name for ubuntu2.local.

**4. Update the serial number in the DNS zone file (important for changes to take effect):**

Example:

pgsql

CopyEdit

@ IN SOA ns.local. admin.local. (

4 ; serial number — increase this number each time you change the file

3600 ; refresh

1800 ; retry

604800 ; expire

86400 ; minimum TTL

)

Make sure to **increase the serial number** each time you edit the file (e.g., from 3 to 4).

**5. Check your DNS zone for errors:**

bash

CopyEdit

sudo named-checkzone local /etc/bind/zones/db.local

**6. Reload your DNS server:**

bash

CopyEdit

sudo rndc reload local.

This tells DNS to update with your new records.

No problem at all! Let's break it down even **simpler** — like you're learning it for the first time. Think of it like setting up a website using tools already built into Ubuntu. Here’s what you need to understand, step by step:

**🧩 Imagine This:**

You’re building a little **website** using **Apache** on your Ubuntu machine (called ubuntu2).  
You want:

* To **configure** Apache so it works your way.
* To **control** which parts of Apache are running.
* To make your website **show your own page** instead of the default one.

**💡 Part 1: Where is Apache's Configuration?**

Apache uses a **folder** to store all its settings:

/etc/apache2

Inside this folder:

| **Name** | **What it is** |
| --- | --- |
| apache2.conf | Main settings file (like the brain of Apache) |
| ports.conf | Tells Apache which ports to listen to (like port 80 for web) |
| sites-available/ | All the websites you can run |
| sites-enabled/ | The websites you're currently running |
| mods-available/ | All the Apache features you *can* turn on |
| mods-enabled/ | The features that are *currently* turned on |

**⚙️ Part 2: Apache Control Commands**

Apache gives you some helpful commands to control it:

| **Command** | **What it does** |
| --- | --- |
| a2enmod | Turn on a feature (like showing server status) |
| a2dismod | Turn off a feature |
| a2ensite | Turn on a website |
| a2dissite | Turn off a website |
| apache2ctl -t | Check for mistakes in your settings |

**🌐 Part 3: Fixing a Common Warning (ServerName)**

Sometimes when you test your Apache settings:

apache2ctl -t

You’ll see:

AH00558: apache2: Could not reliably determine the server's fully qualified domain name

That just means:

"I don’t know what name to call this server."

**To fix it:**

1. Open the main Apache file:

sudo nano /etc/apache2/apache2.conf

1. Go to the bottom and add:

ServerName www.local

1. Also go to the website file:

sudo nano /etc/apache2/sites-enabled/000-default.conf

And **uncomment** (or add) this line:

ServerName www.local

Then save and **restart Apache**:

sudo systemctl restart apache2

✅ The warning is now gone!

**📁 Part 4: Showing Your Own Web Page**

Apache shows files from this folder:

/var/www/html

The main page it shows is:

index.html

**Want to show your own message?**

Run:

echo "Hello, this is MY page!" | sudo tee /var/www/html/index.html

Now go to your browser or use w3m:

w3m www.local

🎉 You’ll see **your custom page** instead of Apache’s default one!

**🔁 Part 5: Do I Need to Restart Apache?**

| **What you do** | **Restart Apache?** |
| --- | --- |
| Change settings or enable modules | ✅ YES |
| Just change HTML files or your website text | ❌ NO |

**✅ Final Summary (in ultra-simple terms):**

You:

1. Learned where Apache settings live (/etc/apache2).
2. Used commands to turn parts of Apache on/off (a2enmod, a2ensite, etc.).
3. Fixed a warning by telling Apache what name to use (ServerName).
4. Made your own simple web page.
5. Learned when you need to restart Apache and when you don’t.

Would you like me to give you a **short script** you can copy-paste to try this out on your machine? Or maybe a simple drawing?

Absolutely! Let’s break this **Apache access log configuration** into **very simple words**, step-by-step, so it’s easy to understand.

**🧠 What’s Happening?**

You're learning how to:

✅ Understand what gets logged when someone visits your website  
✅ Customize the **Apache access log** to only show the parts you care about  
✅ Test how those logs record visits when someone clicks a link on your website

**🔍 Part 1: Apache Makes Logs When People Visit Your Site**

Apache keeps track of visits to your website. These are saved in a file called:

/var/log/apache2/access.log

Each time someone opens a page, Apache writes a **log entry** there.

**📜 Part 2: What’s Inside the Log?**

The log shows:

| **Code** | **What it means** |
| --- | --- |
| %h | IP address of the visitor |
| %l | Remote log name (usually -) |
| %u | Username (if login is used, usually -) |
| %t | Time of the visit |
| %r | The request (like GET /page2 HTTP/1.1) |
| %>s | HTTP status code (like 200 OK) |
| %O | Number of bytes sent to the user |
| %{Referer}i | Where the visitor came from (link source) |
| %{User-Agent}i | The browser or tool used (e.g., Chrome, w3m, etc.) |

**🏷️ Part 3: What is “combined”?**

Apache uses a **shortcut** called combined to represent all those log fields.

So instead of writing a long format every time, we just say:

CustomLog ${APACHE\_LOG\_DIR}/access.log combined

This means:  
➡️ "Log this info in the access.log file using the combined format."

**✏️ Part 4: Let’s Make Our Own Log Format**

You can create your **own version** of that log format (maybe simpler).

**Steps:**

1. Open the main config:

sudo nano /etc/apache2/apache2.conf

1. Find the line that says LogFormat ... combined.
2. Copy and paste that line, and **edit it** to remove things you don’t need (like %l and %u).
3. Rename it to something like:

LogFormat "%h %t \"%r\" %>s %O \"%{Referer}i\" \"%{User-Agent}i\"" my-combined

🧠 This removes:

* Remote log name
* Username  
  So it's cleaner and easier to read.

1. Save and close the file.

**📄 Part 5: Use Your Custom Format**

Next, tell Apache to **use your new log format**.

1. Open the default site config:

sudo nano /etc/apache2/sites-enabled/000-default.conf

1. Find the line:

CustomLog ${APACHE\_LOG\_DIR}/access.log combined

1. Change it to:

CustomLog ${APACHE\_LOG\_DIR}/access.log my-combined

1. Save and close the file.

**🔁 Part 6: Restart Apache**

Because you changed Apache settings, restart the server:

sudo systemctl restart apache2

**🌐 Part 7: Add a Link to See the Referer Working**

Let’s test the **referer** (where a visitor comes from).

1. Open your index page:

sudo nano /var/www/html/index.html

1. Add this inside the file:

<h1>My Page</h1>

<a href="page2.html">See page 2</a>

1. Create page2.html:

sudo nano /var/www/html/page2.html

Put this inside:

<h1>Page 2</h1>

1. Save and close both files.

**🧪 Part 8: Visit the Pages Using w3m**

w3m www.local

* Open the homepage
* Follow the link to Page 2
* Quit the browser (Q and then Y)

**📂 Part 9: Check the Logs**

See the last few lines in the log file:

sudo tail /var/log/apache2/access.log

You will see:

* One line for visiting the index page — **no referer**.
* One line for visiting Page 2 — **referer shows** [**www.local**](http://www.local/) (because you clicked the link).

**✅ Final Summary (Super Simple):**

* Apache keeps track of who visits your site in access.log.
* You can create your own logging format called my-combined to remove things you don’t need.
* The **referer** shows which page someone came from.
* You tested this by adding a link from Page 1 (index) to Page 2.
* After clicking the link, the log shows Page 2 was visited from index.

Would you like me to send you a **ready-made list of commands** you can copy and paste to try this yourself?

Absolutely! Let's break it down **step by step in very simple words**, so you clearly understand what’s going on here.

**🧠 The Goal**

We want to **protect a special page** on our web server called:

/server-status

This page shows **sensitive details** like:

* How busy your web server is
* What pages are being accessed
* How long the server has been running

This is **not something we want everyone to see**, especially over the internet.

**✅ What We Want**

We want to **restrict who can view /server-status**.

* ✅ Allow: **Only localhost (ubuntu2)**
* ❌ Block: **Other computers (like ubuntu1)**

**🔧 Step-by-Step (Plain & Simple)**

**🔍 1. Check if the status module is enabled**

Apache uses "modules" for extra features. The /server-status page comes from the status module.

Check if it's on:

sudo a2query -m status

If it says enabled, you're good.

**📝 2. Edit the configuration for the status page**

Open the file that controls the /server-status page:

sudo nano /etc/apache2/mods-enabled/status.conf

You’ll see something like this:

<Location /server-status>

SetHandler server-status

Require local

</Location>

**What does this mean?**

* <Location /server-status> — This block applies to the /server-status URL
* SetHandler server-status — Tells Apache to treat this as a special status page
* Require local — Only allow computers on the **localhost** (127.0.0.1 or ::1)

✅ This is the **secure setup** — only ubuntu2 can access it.

**🔄 3. Restart Apache**

Whenever you change Apache settings, restart it:

sudo systemctl restart apache2

**🔍 4. Test It from Both Systems**

**✅ On ubuntu2 (local machine):**

Try:

w3m localhost/server-status

You should see the status page.

**❌ On ubuntu1 (remote machine):**

Try:

w3m www.local/server-status

You should see a **403 Forbidden** error.

That means the access restriction is **working perfectly**.

**💡 Optional: Allow All Systems on Your Local Network**

If you *do* want to let more machines in, like anything in your local network (e.g., 192.168.56.x), change the config like this:

<Location /server-status>

SetHandler server-status

Require ip 192.168.56.0/24

</Location>

Then restart Apache again:

sudo systemctl restart apache2

Now, any computer in that IP range can view the status page.

**🧾 Summary**

| **Task** | **Command / Description** |
| --- | --- |
| ✅ Check if module is on | sudo a2query -m status |
| ✏️ Edit access rules | sudo nano /etc/apache2/mods-enabled/status.conf |
| 🚫 Allow only localhost | Add Require local |
| 🌐 Allow a network | Add Require ip 192.168.56.0/24 |
| 🔁 Restart Apache | sudo systemctl restart apache2 |
| ✅ Test from localhost | w3m localhost/server-status |
| ❌ Test from another system | w3m www.local/server-status |

Would you like me to give you **all the commands in a copy-paste block** so you can try it on your system?

Absolutely! Here's a **very simple explanation** of what’s happening in this final part about using **Apache as a web proxy server**:

**🧠 What Is a Proxy Server?**

A **proxy server** is like a **middleman** between your computer and the internet.

Instead of your computer going out to websites like www.pluralsight.com, it **asks the proxy server to do it** for you.

**💡 Why Use a Proxy?**

* ✅ Keep clients (like ubuntu1) **off the internet** directly
* ✅ **Control or monitor** what people are accessing
* ✅ Add an **extra layer of security**

**✅ What You Need to Do (In Simple Steps)**

We’ll turn your Apache web server (on ubuntu2) into a **proxy server**, then set your client machine (e.g., ubuntu1) to use that proxy.

**🛠️ On the Server (ubuntu2) – Set Up Apache as a Proxy**

**1. Enable Required Apache Modules**

Run these two commands to turn on proxy features:

sudo a2enmod proxy

sudo a2enmod proxy\_http

These allow Apache to act as a proxy for HTTP traffic.

**2. Edit the Proxy Configuration**

Open the file:

sudo nano /etc/apache2/mods-enabled/proxy.conf

Inside, you’ll see lines like this (all commented out with #):

<Proxy \*>

#Allow from .example.com

#Require all denied

</Proxy>

👉 You need to:

* **Uncomment** (remove # from) the line ProxyRequests On
* **Add a rule to allow your local network** (and prevent an open proxy)

Update it like this:

<Proxy \*>

Require ip 192.168.56.0/24

</Proxy>

🔒 This means:

* Only computers in your local network (like ubuntu1) can use this proxy
* Random people on the internet **cannot** use your proxy (which is good!)

**3. Restart Apache**

Apply your changes by restarting the Apache server:

sudo systemctl restart apache2

Now Apache is a working **proxy server**! 🎉

**💻 On the Client (ubuntu1) – Use the Proxy**

**1. Set the http\_proxy Environment Variable**

This tells command-line tools (like w3m) to use the proxy.

export http\_proxy=http://ubuntu2.local

Or use the IP address if the hostname doesn't work:

export http\_proxy=http://192.168.56.101

**2. Try Using w3m to Browse**

Check if it still works for local pages:

w3m www.local

Then test an **external** site like:

w3m www.pluralsight.com

🎉 If everything is working, you'll see Pluralsight — even though your **client didn’t go to the internet directly**. The **proxy server went for you.**

**📄 Confirm in the Logs (Back on ubuntu2)**

To make sure it worked, check the Apache logs:

tail /var/log/apache2/access.log

You’ll see:

* A log entry for the Pluralsight request from your client (ubuntu1)
* Apache handled it like a middleman

**🔐 Important Security Tip**

Make sure your proxy only allows trusted IPs:

Require ip 192.168.56.0/24

❌ Do **not** use:

Require all granted

...or your server becomes an **open proxy** anyone can abuse!

**✅ Summary (Very Easy Table)**

| **Step** | **Command / Action** |
| --- | --- |
| Enable proxy features | sudo a2enmod proxy + proxy\_http |
| Configure who can use it | Edit /etc/apache2/mods-enabled/proxy.conf |
| Restart Apache | sudo systemctl restart apache2 |
| Set proxy on client | export http\_proxy=http://ubuntu2.local |
| Browse through proxy | w3m www.pluralsight.com |
| Check it worked | Look in /var/log/apache2/access.log |

Would you like me to give you all the exact copy-paste commands to try this on your system?

Great question — and **you’re very close** to the right idea! Let me clarify this clearly:

**💡 What is a proxy doing here?**

In this setup **you are using Apache as a forward proxy**. That means:

**✅ Your client (e.g., ubuntu1) does not go directly to the internet.**

Instead:

1. Your **client sends the request to your Apache server** (e.g., ubuntu2).
2. The **Apache server acts as the proxy** — it goes to the internet (like www.pluralsight.com) on behalf of the client.
3. The Apache server **gets the response** from the internet.
4. Apache **sends that response back to the client**.

**🔄 So yes — the proxy "talks" to the internet instead of your client.**

**❌ But no — the server (proxy) is not using the proxy for itself.**

It is acting **as** the proxy **for other clients** like ubuntu1.

**📌 In simple terms:**

| **Who** | **Role** | **What they do** |
| --- | --- | --- |
| ubuntu2 (Apache server) | Proxy server | Talks to the internet **on behalf of clients** |
| ubuntu1 (Client) | Uses the proxy | Sends requests **to the proxy**, not the internet |
| The Internet | Destination | Doesn't know the client, only sees the proxy server |

**📥 Example:**

When ubuntu1 runs:

w3m http://www.pluralsight.com

This happens:

* ubuntu1 sends that request **to ubuntu2** (proxy server)
* ubuntu2 makes the real request to Pluralsight
* ubuntu2 gets the response
* ubuntu2 passes it back to ubuntu1

**✅ Final Summary:**

* **Yes**, the proxy (Apache) is the one talking to the internet.
* **Clients**, like your Ubuntu1 machine, send all their requests **to the proxy**, not the internet directly.
* The **proxy server protects and hides the clients**, and can **control access** too.

Let me know if you'd like a simple drawing or if you want to test it with actual commands.

Excellent follow-up question — and you're absolutely right to think deeper here. Let's clear it up step by step:

**🧭 You're asking:**

What happens **when someone from the internet** visits **my web application**?  
Where does the proxy fit into that — since it seems to be used only **internally**?

**🧩 Two Different Roles Apache Can Play**

There are **two different types of proxy roles** Apache can play — and it’s easy to confuse them:

**🔁 1. Forward Proxy (What you're doing now)**

* Used **by internal clients** (like Ubuntu1) to **access the internet**.
* The **clients are inside** your network.
* The **proxy server (Apache)** fetches internet data **for them**.
* Think of this as a **"client-side helper"**.

✅ This is what you just set up.

**🔃 2. Reverse Proxy (What you're asking about now)**

* Used **by people on the internet** to access your **web application**.
* The **Apache server sits in front of your app**, like a shield.
* Requests from the internet **first go to Apache**, then Apache forwards them **internally to your app**, and then responds.

💡 This is useful for:

* Load balancing
* SSL termination
* Caching
* Hiding internal architecture

✅ This is what you'd set up **if you want Apache to handle incoming traffic from the internet** to your web app.

**🎯 So to answer your question directly:**

**Your current proxy setup (forward proxy)** is **only used internally**.

✔️ It's for internal systems like ubuntu1 to access **external websites** through ubuntu2.

But when **someone from the internet** visits **your web app**, they don’t go through that forward proxy.

❌ They go straight to your **public-facing Apache server**, which **acts as a normal web server** — or if configured — as a **reverse proxy**.

**🖼️ Simple Diagram:**

# Forward Proxy (Internal use)

[ubuntu1 client] ---> [Apache proxy on ubuntu2] ---> [Internet]

# Reverse Proxy (Public use)

[Internet User] ---> [Apache Reverse Proxy] ---> [Your internal web app/server]

**🛠️ If you want to make Apache act as a reverse proxy, you’d enable modules like:**

sudo a2enmod proxy

sudo a2enmod proxy\_http

Then configure something like this in a VirtualHost:

<VirtualHost \*:80>

ServerName www.yoursite.com

ProxyPass / http://127.0.0.1:8080/

ProxyPassReverse / http://127.0.0.1:8080/

</VirtualHost>

This would forward **public traffic** to your internal app.

**✅ Final Summary:**

| **Type** | **Used By** | **Used For** | **Traffic Direction** |
| --- | --- | --- | --- |
| Forward Proxy | Internal clients (e.g. ubuntu1) | Accessing external websites | Outbound |
| Reverse Proxy | Internet users | Accessing your internal web app | Inbound |

Let me know if you want help setting up a reverse proxy or comparing both side by side.

Absolutely! Let's break that down into simple and clear terms so you can walk away feeling confident.

**✅ What You Just Finished Learning (Apache Module Recap)**

You’ve completed a module that taught you **how to install and configure the Apache web server**. Here's what you did, step by step:

**📦 1. Installed Apache and Tools**

* You installed **Apache2** (the web server).
* You installed **w3m**, a **command-line web browser**.
* You installed **Apache's documentation**, which became available on your server at:  
  http://localhost/manual

**⚙️ 2. Learned Apache Configuration Structure (Debian/Ubuntu Style)**

* All Apache config files are in /etc/apache2/.
* You learned special a2 commands like:
  + a2enmod → Enable a module
  + a2dismod → Disable a module
  + a2ensite → Enable a website config
  + a2dissite → Disable a website config

**📄 3. Worked With Logs**

* You found where Apache stores logs (like /var/log/apache2/access.log).
* You learned about **LogFormat** — how Apache decides what to log (IP, time, request, etc.).
* You even created your **own custom log format**.

**🔐 4. Restricted Access to Sensitive Pages**

* For example, the **/server-status** page (shows internal info about Apache).
* You limited who can access it using:
  + Require ip 127.0.0.1 (local access only)
  + Require ip 192.168.56.0/24 (allow internal subnet)
* You tested access from different machines.

**🌐 5. Set Up Apache as a Proxy Server**

* You turned Apache into a **forward proxy**.
* This means internal clients can use Apache to reach the internet.
* You secured it, so **only your internal network** can use it — not outsiders.
* Then, from your Ubuntu1 client, you used http\_proxy=... to send requests through Ubuntu2 (the proxy).
* Apache fetched internet content **on behalf of the client**.

**🎉 What’s Next?**

You're done with Apache for now!

🆕 The **next module** will teach you about **MariaDB**, which is a **database server** (similar to MySQL). You’ll install it on **Ubuntu 20.04**, and learn how to:

* Set it up
* Secure it
* Create databases
* Manage users and access

**💡 TL;DR Summary:**

| **What You Did** | **Why It’s Useful** |
| --- | --- |
| Installed Apache | To serve web pages |
| Used w3m & docs | To test and learn from the command line |
| Enabled/disabled modules/sites | To control Apache features |
| Configured logs | To track who visits your site |
| Restricted access | To protect sensitive pages |
| Made a proxy server | To let internal clients access the internet securely |

Let me know when you're ready for the MariaDB part — I’ll explain it just as clearly!

Of course! Let’s break this down into **very simple words**, so it’s easy to understand.

**🧠 What’s This All About?**

You’re starting a new module where you’ll learn how to **install and use MariaDB** — which is a **database server** — on **Ubuntu 20.04**.

**👨‍🏫 Who’s Teaching?**

The instructor is **Andrew Mallett** from Pluralsight. He’s going to guide you step-by-step through everything.

**✅ What Will You Learn?**

You will:

1. **Install** MariaDB on Ubuntu.
2. **Secure** it (so random people can’t access it).
3. Learn about:
   * **Authentication** (how users log in).
   * **User accounts** (like the root user).
   * **Default databases** that come with MariaDB.
4. **Create your own databases** (where you’ll store data).
5. **Create users** and **give them permission** to use your databases.

**🧰 What Is MariaDB?**

* It’s a **database server** — it stores and manages data.
* It’s a **free and open-source version** of MySQL.
* It was created by the same person who made MySQL: **Michael "Monty" Widenius**.

**🧑‍💻 What Will You Use?**

* **Ubuntu2** will be your **database server** (it runs MariaDB).
* **Ubuntu1** will act as a **client** (it connects to Ubuntu2 to use the database).

**🖥️ Installing and Using MariaDB**

Once MariaDB is installed:

1. You’ll connect to it using this command:
2. sudo mysql -u root
   * This logs in as the **database root user**.
   * You must use sudo because the root user only works from **Linux root** (for security reasons).
3. After logging in, you can run commands like:
   * show databases; → to see all databases.
   * use mysql; → go into a specific database.
   * describe user; → see the structure of the user table (like columns and types).
   * select user, plugin from user; → see login methods and which users exist.

**🔒 Important: Authentication**

* The **root user** in MariaDB is set up to use unix\_socket authentication.
* This means:
  + You must be logged in to **Ubuntu as root** to connect.
  + You can **only connect from the same machine** (localhost).
  + You can’t log in as root remotely.

**📦 Coming Up Next**

* You’ll **install MariaDB** on **Ubuntu2**.
* Then you’ll start working with it by connecting, creating databases, and setting up users.

**📌 TL;DR (Too Long; Didn’t Read)**

| **What You’re Learning** | **In Simple Words** |
| --- | --- |
| MariaDB | A system to store and manage data (like contacts, logins, orders, etc.) |
| Installation | Putting MariaDB on Ubuntu2 |
| Security | Making sure only allowed users can connect |
| Authentication | Controlling who can log in and how |
| Client/Server | Ubuntu1 connects to Ubuntu2 (the database server) |
| Root User | Admin account that only works from local system using sudo |

Let me know when you're ready to start the MariaDB installation — I can walk you through that part too!

Absolutely! Let me break this down for you in **very simple words**, step by step.

**🧑‍🏫 What’s Happening Here?**

You are learning how to **install and use MariaDB** (a database system) on your **Ubuntu2** machine.

**🛠️ What You Do in This Demo:**

1. **Install MariaDB** server.
2. **Try to connect** to MariaDB as the root user.
3. **See why it doesn’t work at first**.
4. **Learn how to properly connect** using sudo.
5. **Look at the databases** and how to explore them.

**🔄 Step-by-Step in Simple Words:**

**✅ Step 1: Install MariaDB**

In the terminal, run:

sudo apt install mariadb-server

This installs:

* The **database server** (MariaDB)
* The **client** (so you can run commands)

**❌ Step 2: Try to connect as root (Wrong way)**

You try:

mysql -u root

But it fails:

* You get **"Access denied"** because **root access is protected**.

**✅ Step 3: Connect as root (Correct way)**

Now you try:

sudo mysql -u root

And it works!

Why? Because:

* MariaDB is using **unix\_socket** authentication.
* That means: only the **Linux root user** can log in as **MariaDB’s root user** — and **only from the same computer** (localhost).

**🧠 Why is this important?**

This makes MariaDB **more secure**:

* No one can connect as root from another computer.
* No password is needed because it checks if you're the Linux root user.

**📂 Step 4: View and Use Databases**

Once inside MariaDB (after sudo mysql -u root), you can:

**Show all databases:**

show databases;

You’ll see default databases like:

* information\_schema
* mysql
* performance\_schema

**Use a database:**

use mysql;

This switches you into the mysql database so you can explore its tables.

**Look at a table’s structure:**

describe user;

You’ll see the **columns** in the user table (like username, plugin, etc.).

**See which users exist:**

select user, plugin from user;

You’ll probably see just one user:

* root using unix\_socket plugin

**🚪 Exit MariaDB:**

exit;

You are now back at the Linux terminal.

**🔒 Summary: Why use sudo?**

* sudo gives **Linux root access**.
* **MariaDB root user** is connected only if you are **Linux root**.
* So: sudo mysql -u root is the correct way to log in.

**✅ Final Tips:**

| **Action** | **Command** |
| --- | --- |
| Install MariaDB | sudo apt install mariadb-server |
| Log in to MariaDB | sudo mysql -u root |
| Show databases | show databases; |
| Use a database | use mysql; |
| Exit MariaDB | exit; |

Let me know when you’re ready for the next step — creating your own users and databases!