Web Servers All In One

❖ Introduction to Web Servers

Definition and purpose of web servers

A **web server** is a specialized software application or hardware system that handles and fulfills requests for web content, typically over the HTTP (Hypertext Transfer Protocol) or HTTPS (HTTP Secure) protocols. When a user enters a URL into a browser, the web server retrieves the requested resources—such as HTML pages, images, or applications—and delivers them to the user's device, enabling web browsing and interaction with online services.

There are two key components to a web server:

- **Software:** A web server program (such as Apache HTTPD or NGINX) that handles incoming requests, processes them, and serves the correct resources.
- **Hardware:** A physical or virtual machine where the web server software and the website's content are hosted.

Purpose of Web Servers:

1. **Hosting and Delivering Web Content:** Web servers store, manage, and deliver website files to clients (browsers) upon request. This includes HTML pages, CSS stylesheets, JavaScript files, media (images, videos), and other resources needed for displaying and interacting with a website.
2. **Dynamic Content Generation:** Modern web servers work alongside server-side technologies (like PHP, Python, Ruby, or Node.js) to generate dynamic content. This means they don't just serve static files, but also handle requests that require processing data (like user logins or database queries) and return customized or updated content in real-time.
3. **Handling and Routing Web Traffic:** Web servers are responsible for managing incoming requests, efficiently routing them to the correct files or services. Through configurations like **virtual hosting**, web servers can host multiple websites on a single server. Features like **load balancing** distribute traffic across multiple servers to ensure optimal performance and availability.
4. **Security and Encryption:** Using HTTPS, web servers provide a secure channel between the user's browser and the server, encrypting data to protect against eavesdropping or tampering. Web servers can also integrate additional security mechanisms, such as firewalls, access controls, and secure headers, to safeguard web applications and their users.
5. **Supporting APIs and Web Services:** Beyond just serving web pages, modern web servers also act as endpoints for **APIs (Application Programming Interfaces)**. They facilitate machine-to-machine communication, powering back-end services and mobile applications by processing data requests and sending responses.
6. **Scalability and High Availability:** Web servers support high-traffic applications through scaling techniques such as clustering and load balancing. By distributing traffic across multiple servers or data centers, web servers ensure that applications remain available and responsive even during traffic surges or failures.

Common web servers: Apache HTTPD and NGINX

- **On Ubuntu and Debian-based systems**, the package name is typically **apache2** (this includes the Apache HTTP Server software).
- **On RHEL (Red Hat Enterprise Linux), CentOS, Fedora, and other RHEL-based systems**, the package name is **httpd**, which stands for "HTTP Daemon" and refers to the Apache HTTP Server as well.

  The underlying software is the same (Apache HTTP Server), but the package name and how it's managed differ between distributions:

- On **Ubuntu/Debian**, you use commands like sudo apt install apache2 to install Apache.
- On **RHEL/CentOS/Fedora**, you use sudo yum install httpd (or dnf on newer versions of Fedora) to install Apache.

  Both manage and refer to the same Apache HTTP server, just named differently according to their package management systems

So now Apache HTTPD and NGINX are two of the most widely used web servers in the world, but they differ in architecture, performance, and typical use cases. Here's a comparison of the two:

Apache HTTPD (HTTP Daemon):

Overview:

- Released in 1995, Apache HTTPD is one of the oldest and most widely used web server software.
- It's modular, highly configurable, and widely supported on different platforms.
- Apache processes incoming requests with a variety of multi-processing modules (MPMs), such as prefork, worker, and event, which define how it handles concurrent requests.

  NGINX:

  Overview:

- Released in 2004, NGINX was designed with performance in mind, especially to handle high concurrency. Its name comes from "Engine X."
- NGINX is both a web server and a reverse proxy server.
- It uses an asynchronous, non-blocking, event-driven architecture that makes it extremely efficient in handling a large number of simultaneous requests.

Key Differences Between Apache HTTPD and NGINX:

1. **Architecture**:
    - **Apache**: Works with a process-driven architecture. Each connection is handled by a separate thread or process depending on the MPM (Multi-Processing Module) used. This architecture can struggle under heavy loads due to the overhead of managing multiple threads or processes.

- **NGINX**: Uses an event-driven, asynchronous, non-blocking architecture. A single worker can handle thousands of connections simultaneously, making it more scalable and efficient in high-traffic scenarios.

2. **Performance**:
   - **Apache**: Works well for dynamic content and smaller sites where high concurrency isn't critical. However, under heavy traffic or when handling many concurrent connections, Apache can become resource-intensive and slower compared to NGINX.
   - **NGINX**: Optimized for handling static content (like images, CSS, JavaScript). It performs exceptionally well under high-traffic conditions, managing large numbers of simultaneous connections efficiently.
3. **Handling Dynamic Content**:
   - **Apache**: Can directly handle dynamic content (such as PHP, Python, or Ruby) through embedded modules like mod_php, mod_python, etc. It can execute scripts and return dynamic content natively within its process.
   - **NGINX**: Does not directly execute dynamic content like PHP. Instead, it serves as a reverse proxy, forwarding dynamic requests to an external processor (like PHP-FPM). This separation of concerns (static vs dynamic) enhances performance.
4. **Configuration**:
   - **Apache**: Offers more flexibility and power through .htaccess files, which allow per-directory overrides of server settings. This gives users greater control over specific website configurations without restarting the server.
   - **NGINX**: Does not support .htaccess or per-directory overrides. All configurations are done in a centralized configuration file, making it less flexible in this regard but also faster, as it avoids reading multiple configuration files on every request.
5. **Modules and Extensibility**:
   - **Apache**: Has a rich ecosystem of modules (mod_rewrite, mod_security, mod_ssl, etc.) that can be enabled or disabled to extend its functionality. It supports a wide range of use cases with its modules.
   - **NGINX**: Also supports modules, but they must be compiled into the core at build time. NGINX's module system is less flexible than Apache's dynamic module loading.
6. **Reverse Proxy and Load Balancing**:
   - **Apache**: Can act as a reverse proxy and load balancer, but it isn't its primary strength. Modules like mod_proxy and mod_balancer enable these functions, but they are more basic compared to NGINX.
   - **NGINX**: Designed from the ground up to be a highly efficient reverse proxy and load balancer. It's commonly used as a reverse proxy in front of other web servers or application servers due to its ability to manage traffic effectively.

Similarities Between Apache HTTPD and NGINX:

1. **Both Are Open Source**:

- Apache and NGINX are both free and open-source projects, making them highly accessible and widely supported by large communities.

2. **Cross-Platform**:
    - Both web servers are available for various platforms, including Linux, Windows, and macOS, and are used in a wide range of environments, from small personal websites to large-scale enterprise systems.
3. **SSL/TLS Support**:
    - Both Apache and NGINX support SSL/TLS and can handle HTTPS traffic securely. They offer features for managing SSL certificates and supporting modern protocols like HTTP/2.
4. **Reverse Proxy Capabilities**:
    - Both Apache and NGINX can function as reverse proxies, routing requests to other servers or applications.

Which Is Most Used For What?

- **Apache**:
    - **Best for Dynamic Content**: Apache is often used for sites that heavily rely on dynamic content (like PHP-based applications, including WordPress, Drupal, and Joomla).
    - **Flexible Configuration**: Apache's .htaccess files and per-directory configuration make it ideal for shared hosting environments where users need control over specific directories.
    - **Legacy Support**: Apache is often chosen for older projects or environments that require compatibility with legacy systems or modules.
- **NGINX**:
    - **High Performance & Concurrency**: NGINX is the go-to choice for handling high traffic volumes or sites that serve a lot of static content. It is commonly used by large-scale websites and applications (such as Netflix, Dropbox, and WordPress.com) that require high performance and scalability.
    - **Reverse Proxy & Load Balancer**: NGINX shines as a reverse proxy or load balancer due to its efficient resource management and event-driven design.
    - **Static Content**: If a site primarily serves static files (like media, images, and static HTML pages), NGINX is often preferred due to its faster content delivery.

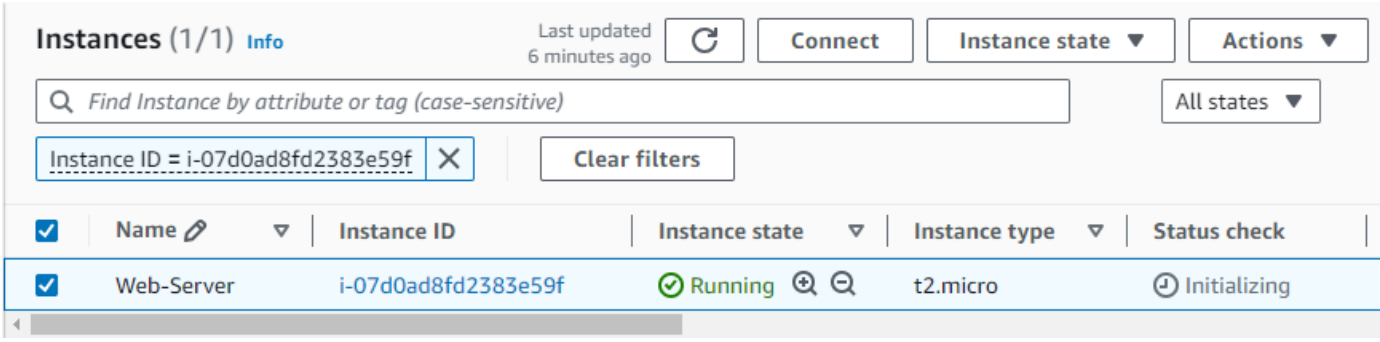
Which Is Most Used?

- In terms of usage:
    - **Apache** was historically the most popular web server. It still holds a significant portion of the market, especially for dynamic content and legacy sites.
    - **NGINX** has rapidly gained popularity in recent years due to its superior performance under high loads. It is often used in modern, large-scale web applications and as a reverse proxy.

Both Apache and NGINX have their strengths, and they are sometimes used together in a **hybrid setup**: NGINX is deployed as a reverse proxy for serving static content and handling traffic, while Apache handles dynamic content generation behind the scenes.

- **Apache HTTPD Server**
  - **Installation and configuration of Apache HTTPD (Linux RHEL-9 OS)**

Step-1 (Create an Instance Machine)



Step-2 (Install HTTPD package and start/enable the service)

 # yum install -y httpd

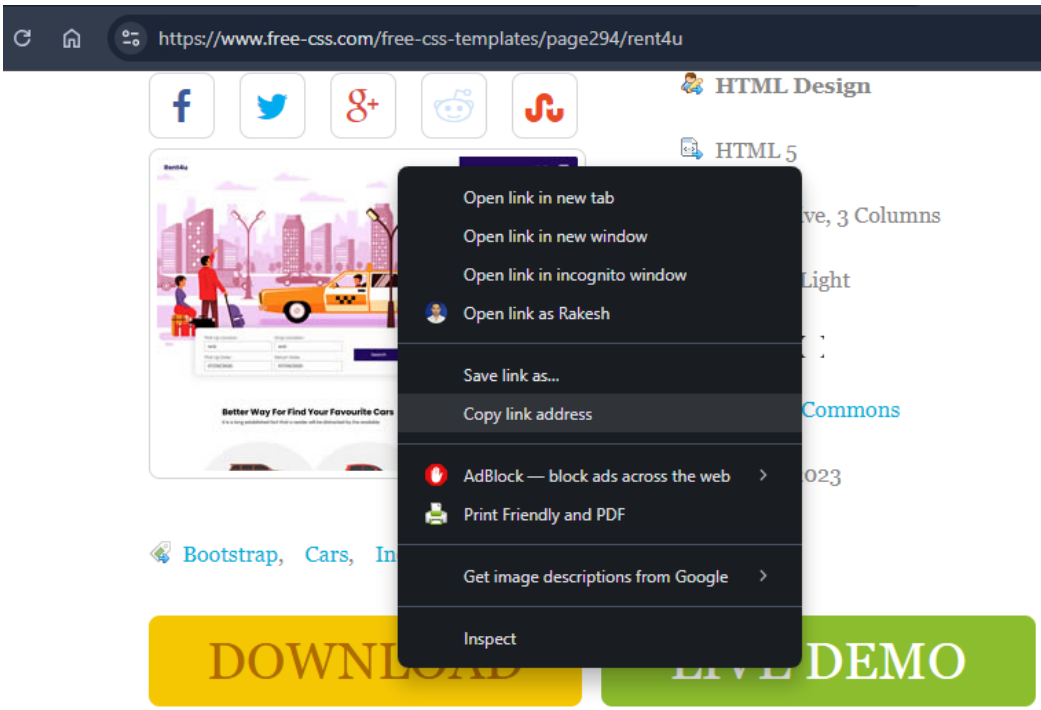
```
[root@testing-server ~]# yum install -y httpd
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use

Last metadata expiration check: 0:01:06 ago on Wed 16 Oct 2024 05:50:54
Dependencies resolved.
================================================================================
 Package                    Architecture        Version
================================================================================
Installing:
 httpd                      x86_64              2.4.57-11.el9_4.1
Installing dependencies:
 apr                        x86_64              1.7.0-12.el9_3
 apr-util                   x86_64              1.6.1-23.el9
```

 # systemctl enable --now httpd

```
[root@testing-server ~]# systemctl enable --now httpd
Created symlink /etc/systemd/system/multi-user.target.wants/httpd.service → /usr/
[root@testing-server ~]#
```

Step-3 (Add some static data to the HTTPD document root directory)

There is a free website named "https://www.free-css.com", download a free template using "copy link address" from download button



Now download this website template directly through "wget" command

```
[root@testing-server ~]# wget https://www.free-css.com/assets/files/free-css-templates/download/page294/rent4u.zip
--2024-10-16 06:12:50--  https://www.free-css.com/assets/files/free-css-templates/download/page294/rent4u.zip
Resolving www.free-css.com (www.free-css.com)... 217.160.0.242, 2001:8d8:100f:f000::28f
Connecting to www.free-css.com (www.free-css.com)|217.160.0.242|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 919720 (898K) [application/zip]
Saving to: 'rent4u.zip'

rent4u.zip                    100%[=========================================================>] 898.16K  1.29MB/s

2024-10-16 06:12:51 (1.29 MB/s) - 'rent4u.zip' saved [919720/919720]

[root@testing-server ~]#
```

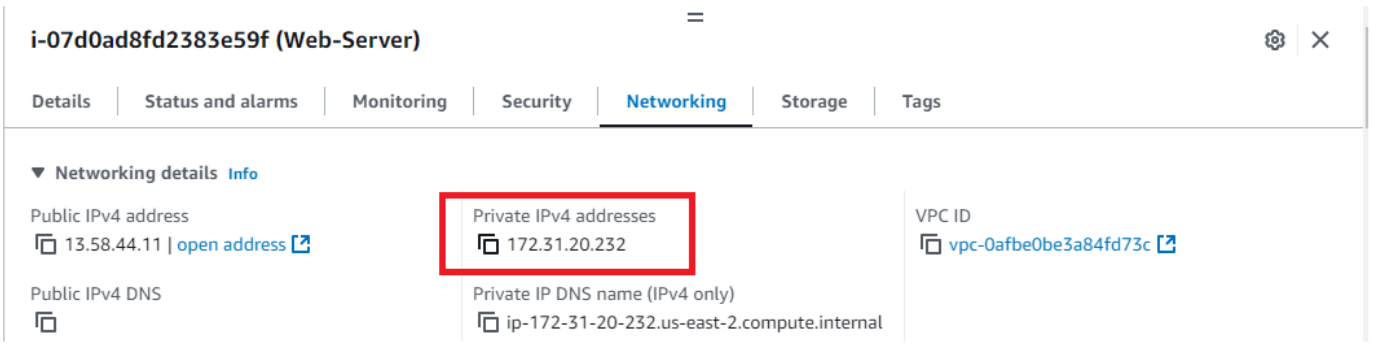
Extract and paste it on /var/www/html. Note: Please check the SELinux label status "httpd_sys_content_t" on document directory /var/www/html/* .

```
[root@testing-server ~]# cp -r rent4u-html/* /var/www/html/
[root@testing-server ~]#
```

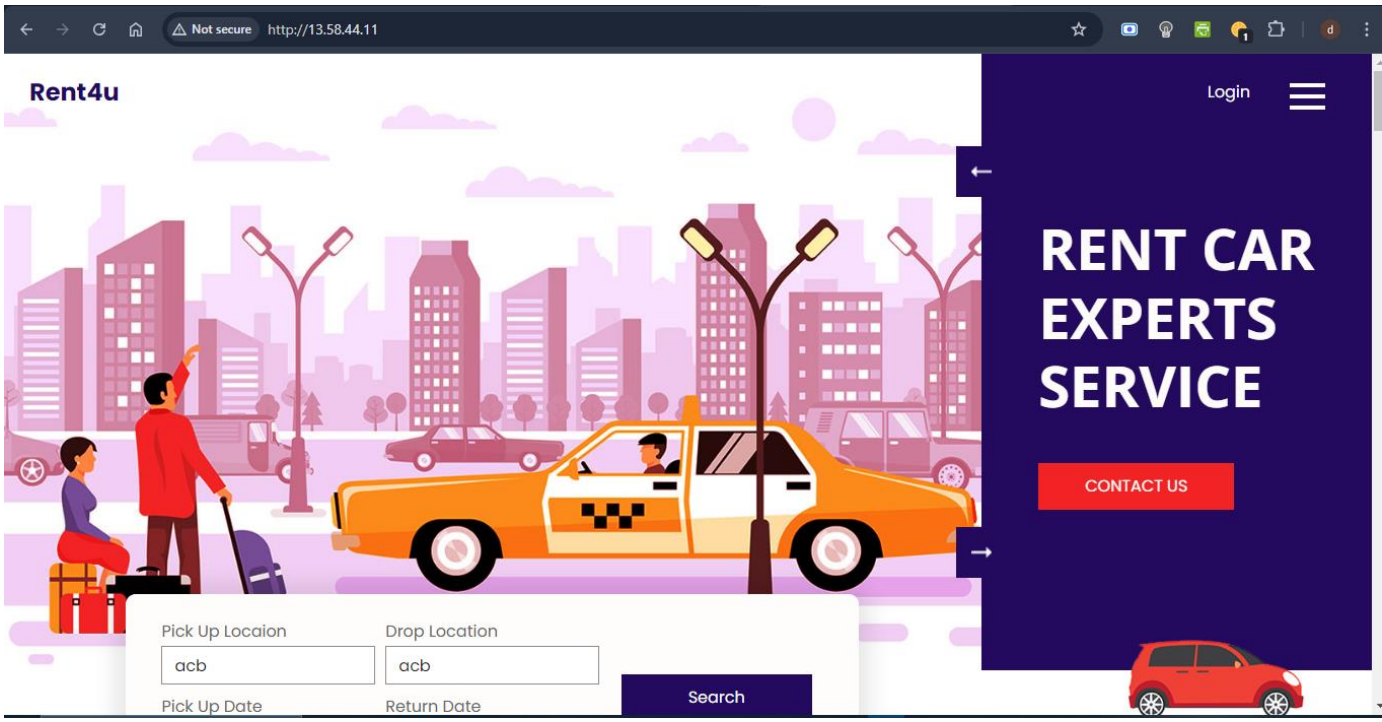
```
[root@testing-server ~]# ls -lZ /var/www/html/
total 56
-rw-r--r--. 1 root root unconfined_u:object_r:httpd_sys_content_t:s0  3732 Oct 16 06:17 about.html
-rw-r--r--. 1 root root unconfined_u:object_r:httpd_sys_content_t:s0  6163 Oct 16 06:17 blog.html
-rw-r--r--. 1 root root unconfined_u:object_r:httpd_sys_content_t:s0  7186 Oct 16 06:17 car.html
-rw-r--r--. 1 root root unconfined_u:object_r:httpd_sys_content_t:s0  5601 Oct 16 06:17 contact.html
drwxr-xr-x. 2 root root unconfined_u:object_r:httpd_sys_content_t:s0   105 Oct 16 06:17 css
drwxr-xr-x. 2 root root unconfined_u:object_r:httpd_sys_content_t:s0  4096 Oct 16 06:17 images
-rw-r--r--. 1 root root unconfined_u:object_r:httpd_sys_content_t:s0 24420 Oct 16 06:17 index.html
drwxr-xr-x. 2 root root unconfined_u:object_r:httpd_sys_content_t:s0    70 Oct 16 06:17 js
```

Your web content is now ready to serve. But need to add an 80/TCP port in the inbound rule of instances security group. This is a non-secure website, and it listens on port 80/TCP from public.

Step-4 Access the website using instances public IP.



172.31.20.232 is my instance's public IP. So paste this on a web browser like Chrome. Our website is ready to serve



- **Basic Apache directives and configuration files**

Main Configuration File

- /etc/httpd/conf/httpd.conf: This is the main configuration file for Apache (httpd) on RHEL-based systems like RHEL 9. It contains global server configuration settings, directives, and virtual host configurations. Most server settings, like port numbers, server names, and document root, are found here.

Key Apache Directives

1. **Listen:**
   This directive specifies the IP address and port number on which Apache listens for incoming connections.
   - Example: Listening on port 80 for HTTP and 443 for HTTPS.

```
#Listen 12.34.56.78:80
Listen 80
```

2. **DocumentRoot:**
   Specifies the directory where the server looks for files to serve. This is where the website's files (HTML, CSS, JS) are stored.

- Example: Set to "/var/www/html" by default.

```
# symbolic links and aliases may be used to point to other locations.
#
DocumentRoot "/var/www/html"

#
# Relax access to content within /var/www.
#
```

3. **<Directory></directory>:**
   This directive sets rules for specific directories. It is used to control access and configure permissions for web server folders.
   - Example: Configure options for "/var/www/html" to control directory permissions and behavior.

```
# Further relax access to the default document root:
<Directory "/var/www/html">
    Options Indexes FollowSymLinks
    AllowOverride None
    Require all granted
</Directory>
```

4. **Options:**
   Controls what features are enabled for a particular directory. This includes settings for allowing script execution, indexes, and symbolic links.
   - Example: Set to "Indexes FollowSymLinks" to allow directory listings and symbolic links.

```
    #
    Options Indexes FollowSymLinks
```

5. **AllowOverride:**
   Specifies whether .htaccess files can override configuration settings for directories.
   - Example: "None" disables .htaccess overrides, while "All" allows them.

```
    #
    # AllowOverride controls what directives may be placed in .htaccess files.
    # It can be "All", "None", or any combination of the keywords:
    #   Options FileInfo AuthConfig Limit
    #
    AllowOverride None
```

6. **Require:**
   This directive defines access control by specifying who is allowed or denied access to the server or specific resources.
   - Example: "Require all granted" allows access to all users.

```
    #
    Require all granted
```

7. **ErrorLog and CustomLog:**

- ErrorLog: Defines where server errors are logged, such as issues with configurations or server failures.
- CustomLog: Specifies where access logs (incoming requests) are stored, recording information about visitors and the requests they make.

```
#
ErrorLog "logs/error_log"
```

8. **VirtualHost:**
   Used for configuring virtual hosting, allowing multiple websites to be served from a single server. You can configure either IP-based or name-based virtual hosting.
   - Example: A virtual host for www.example.com can define the ServerName, DocumentRoot, ErrorLog, and CustomLog specifically for that site.

```
<virtualhost *:80>
servername photos.wordworld.tech
documentroot /var/www/photos
</virtualhost>
```

Key Configuration Files

1. /etc/httpd/conf/httpd.conf:
   This is the main Apache configuration file where most global settings and directives are defined.
2. /etc/httpd/conf.d/*.conf:
   This directory contains additional configuration files for specific modules or features. For example, the ssl.conf file in this directory is used for configuring SSL/TLS settings.
3. /etc/httpd/conf.modules.d/*.conf:
   This directory contains module configuration files. Apache modules are used to extend the server's functionality (e.g., enabling PHP, SSL, or caching modules). These files define which modules are loaded at runtime.
4. /var/log/httpd/:
   This is the default directory for storing Apache logs. Two important log files are stored here:
   - error_log: Stores error messages related to server failures, permission issues, or configuration problems.
   - access_log: Stores details of incoming HTTP requests, such as IP addresses, request URLs, and response codes.
5. /var/www/html/:
   This is the default document root where website files are stored. Apache serves content from this directory unless otherwise specified in the httpd.conf file.

- **Virtual Hosting**

  - **What is Virtual Hosting?**

Virtual hosting is a method used by web servers (like Apache HTTPD or NGINX) to host multiple websites (or domains) on a single physical server or IP address. This allows web hosting providers or system administrators to efficiently use server resources and run multiple websites without needing separate servers for each domain. Virtual hosting can be done in two main ways: IP-based virtual hosting and name-based virtual hosting.

  - **IP-Based Virtual Hosting**

- **Definition**: In IP-based virtual hosting, each website is assigned a unique IP address on the same server. The server listens on different IP addresses, and when a request is made, the server uses the IP address to determine which website to serve.
- **How it works**: The web server identifies which site to serve based on the destination IP address in the request.
- **Use case**: This method is used when each site requires its own IP address, such as when using SSL certificates that require individual IPs (though modern TLS with Server Name Indication (SNI) reduces this need).
- **Drawbacks**: Requires multiple IP addresses, which may not be feasible if IP addresses are limited.

Example use case:

Hosting multiple websites where each site has its own dedicated IP address, for example:

- Site 1: IP 192.168.1.10
- Site 2: IP 192.168.1.20

Step-by-step guide to setting up IP-Based Virtual Hosting

Name-Based Virtual Hosting

- **Definition:** In name-based virtual hosting, multiple websites share a single IP address, and the web server uses the requested domain name (host header) to determine which site to serve.
- **How it works:** The web server relies on the Host header in the HTTP request to distinguish between different websites hosted on the same IP address.
- **Use case:** This is the most common method used today, especially for shared hosting environments, because it allows hosting multiple websites on a single IP address.
- **Drawbacks:** Older browsers without SNI support might not be able to handle multiple SSL websites on a single IP (though this is mostly no longer an issue).
- **Example Use Case:**

Hosting multiple websites with the same IP address, for example:

Site 1: www.example.com (IP: 192.168.1.10)

Site 2: www.example2.com (IP: 192.168.1.10)

Step-by-step guide to setting up name-based virtual hosts

To set up **name-based virtual hosting**, you need two or more domain names that will map to different web content on the same physical server. Each domain will point to the same IP address, but the web server will serve different content based on the domain name in the request.

If you are managing your own custom **DNS server**, you will need to add DNS records for each domain that point to the same server IP address. This can be done by editing the appropriate zone file, typically located in /var/named/ on a **BIND** DNS server.

Configuring DocumentRoot for Each Domain

For each domain, you will need to create an individual **DocumentRoot** directory where the website files will be stored. For example:

- /var/www/site1 for **site1.rakamodify.online**
- /var/www/site2 for **site2.rakamodify.online**
- /var/www/html for rakamodify.online and www.rakamodify.online

Next, you will create individual virtual host configuration files for each domain under /etc/httpd/conf.d/. Each virtual host file should specify the domain's ServerName and the corresponding DocumentRoot directory. Here's an example of the configuration:

Save each virtual host configuration file under **/etc/httpd/conf.d/** with a .conf extension, such as example1.conf and example2.conf.

```
[root@testing-server conf.d]# cd /etc/httpd/conf.d/
[root@testing-server conf.d]# ls
autoindex.conf  html.conf  README  site1.conf  site2.conf  userdir.conf  welcome.conf
[root@testing-server conf.d]#
```

```
<VirtualHost *:80>
ServerName      rakamodify.online
DocumentRoot    /var/www/html
</VirtualHost>
```

vim /etc/httpd/conf.d/html.conf

<virtualhost *:80>

servername **rakamodify.online**

documentroot /var/www/html

</virtualhost>

```
<VirtualHost *:80>
ServerName      site1.rakamodify.online
DocumentRoot    /var/www/site1
</VirtualHost>
~
~
```

vim /etc/httpd/conf.d/site1.conf

<virtualhost *:80>

servername **site1.rakamodify.online**

documentroot /var/www/site1

</virtualhost>

```
<VirtualHost *:80>
ServerName      site2.rakamodify.online
DocumentRoot    /var/www/site2
</VirtualHost>
~
~
```

vim /etc/httpd/conf.d/site2.conf

<virtualhost *:80>

servername **site2.rakamodify.online**

documentroot /var/www/site2

</virtualhost>

```
[root@testing-server conf.d]# systemctl status httpd
● httpd.service - The Apache HTTP Server
     Loaded: loaded (/usr/lib/systemd/system/httpd.service; enabled;
     Active: active (running) since Wed 2024-10-16 15:17:41 UTC; 11m
       Docs: man:httpd.service(8)
   Main PID: 19017 (httpd)
     Status: "Total requests: 4; Idle/Busy workers 100/0;Requests/sec
      Tasks: 177 (limit: 4400)
     Memory: 22.0M
        CPU: 346ms
     CGroup: /system.slice/httpd.service
             ├─19017 /usr/sbin/httpd -DFOREGROUND
             ├─19019 /usr/sbin/httpd -DFOREGROUND
             ├─19020 /usr/sbin/httpd -DFOREGROUND
             ├─19021 /usr/sbin/httpd -DFOREGROUND
             └─19022 /usr/sbin/httpd -DFOREGROUND

Oct 16 15:17:41 testing-server.example.com systemd[1]: Stopped The Ap
Oct 16 15:17:41 testing-server.example.com systemd[1]: Starting The A
Oct 16 15:17:41 testing-server.example.com systemd[1]: Started The Ap
Oct 16 15:17:41 testing-server.example.com httpd[19017]: Server conf
```

systemctl restart httpd

Now check named-based content with curl

```
[root@testing-server ~]# curl http://rakamodify.online
rakamodify.online is working
[root@testing-server ~]# curl http://www.rakamodify.online
rakamodify.online is working
[root@testing-server ~]# curl http://site1.rakamodify.online
site1.rakamodify.online is working
[root@testing-server ~]# curl http://site2.rakamodify.online
site2.rakamodify.online is working
[root@testing-server ~]#
```

- **Secure Web Server Setup**

Make your website connection secure

Step-1: install EPEL repository (Configure swap space in cloud instance)

```
[root@testing-server ~]# dnf install https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm -y
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use "rhc" or "subscription-manager" to registe

Last metadata expiration check: 2:00:25 ago on Wed 16 Oct 2024 01:47:51 PM UTC.
epel-release-latest-9.noarch.rpm
Dependencies resolved.
================================================================================
 Package                    Architecture                Version
================================================================================
Installing:
 epel-release               noarch                      9-8.el9
```

This is a cloud instance, so before configure epel repository I need to configure swap space memory.

```
[root@testing-server ~]# free -h
               total        used        free      shared  buff/cache   available
Mem:           765Mi       321Mi       216Mi       7.0Mi       352Mi       443Mi
Swap:          5.0Gi       0.0Ki       5.0Gi
[root@testing-server ~]#
```

Step-2: Install required packages (On custom DNS server)

yum install -y mod_ssl certbot python3-certbot-apache

```
[root@testing-server ~]# yum install -y mod_ssl  certbot python3-certbot-apache
Updating Subscription Management repositories.
Unable to read consumer identity

This system is not registered with an entitlement server. You can use "rhc" or "subscription-ma

Extra Packages for Enterprise Linux 9 - x86_64
Extra Packages for Enterprise Linux 9 openh264 (From Cisco) - x86_64
Dependencies resolved.
================================================================================
 Package                    Architecture     Version                    Reposit
================================================================================
Installing:
 certbot                    noarch           2.11.0-1.el9               epel
 mod_ssl                    x86_64           1:2.4.57-11.el9_4.1        rhel-9-
 python3-certbot-apache     noarch           2.11.0-1.el9               epel
Installing dependencies:
 augeas-libs                x86_64           1.13.0-6.el
```

certbot --apache

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1: rakamodify.online
2: site1.rakamodify.online
3: site2.rakamodify.online
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Select the appropriate numbers separated by commas and/or spaces, or leave input
blank to select all options shown (Enter 'c' to cancel):
Requesting a certificate for rakamodify.online and 2 more domains

Successfully received certificate.
Certificate is saved at: /etc/letsencrypt/live/rakamodify.online/fullchain.pem
Key is saved at:         /etc/letsencrypt/live/rakamodify.online/privkey.pem
This certificate expires on 2025-01-14.
These files will be updated when the certificate renews.
Certbot has set up a scheduled task to automatically renew this certificate in the background.

Deploying certificate
Successfully deployed certificate for rakamodify.online to /etc/httpd/conf.d/html-le-ssl.conf
Successfully deployed certificate for site1.rakamodify.online to /etc/httpd/conf.d/site1-le-ssl.conf
Successfully deployed certificate for site2.rakamodify.online to /etc/httpd/conf.d/site2-le-ssl.conf
Congratulations! You have successfully enabled HTTPS on https://rakamodify.online, https://site1.rakamodify.onl
e2.rakamodify.online
We were unable to subscribe you the EFF mailing list. You can try again later by visiting https://act.eff.org.
```

What happen when we secure our website with lets encrypt

When you secure your website using the command certbot --apache with Let's Encrypt, the following steps occur:

1. Installation of SSL/TLS Certificates:
   - Let's Encrypt issues a free SSL/TLS certificate for your domain, which enables HTTPS (secure HTTP) for your website.
   - The command certbot --apache automatically handles the certificate request, validation, installation, and renewal processes.
2. Domain Ownership Validation:
   - Let's Encrypt needs to verify that you own the domain for which you're requesting the certificate. This is typically done through an HTTP challenge, where Certbot temporarily places a validation file on your server.
   - Let's Encrypt's servers will access this file to confirm domain ownership.
3. Apache Configuration Update:
   - Once validation is successful, Certbot will:
     - Generate SSL/TLS certificates (both the public and private keys).
     - Place these certificates in a default location, typically /etc/letsencrypt/live/yourdomain/.

```
[root@testing-server ~]# cd /etc/letsencrypt/live/rakamodify.online/
[root@testing-server rakamodify.online]# ls
cert.pem  chain.pem  fullchain.pem  privkey.pem  README
[root@testing-server rakamodify.online]#
```

     - Automatically update your Apache configuration to use these certificates. It creates or modifies your Apache virtual host configuration files to enable SSL.
     - The configuration will include paths to your certificate files:

- SSLCertificateFile: Path to the SSL certificate.
- SSLCertificateKeyFile: Path to the private key.
- SSLCertificateChainFile: Path to the intermediate certificate chain.

```
Successfully received certificate.
Certificate is saved at: /etc/letsencrypt/live/rakamodify.online/fullchain.pem
Key is saved at:         /etc/letsencrypt/live/rakamodify.online/privkey.pem
This certificate expires on 2025-01-14.
These files will be updated when the certificate renews.
Certbot has set up a scheduled task to automatically renew this certificate in the background.

Deploying certificate
Successfully deployed certificate for rakamodify.online to /etc/httpd/conf.d/html-le-ssl.conf
Successfully deployed certificate for site1.rakamodify.online to /etc/httpd/conf.d/site1-le-ssl.conf
Successfully deployed certificate for site2.rakamodify.online to /etc/httpd/conf.d/site2-le-ssl.conf
```

4. Enforce HTTPS:

- Certbot typically enables HTTPS redirection by modifying your Apache configuration. This means any HTTP requests will be automatically redirected to the secure HTTPS version of your site.
- It adds a rewrite rule in your virtual host file or creates a new virtual host for port 443 (HTTPS) with SSL enabled.

5. Automated Certificate Renewal:

- Let's Encrypt certificates are valid for 90 days, but Certbot sets up automatic renewal to ensure your certificate is always up to date.
- A cron job or systemd timer is typically created to check and renew the certificate automatically before it expires, without requiring any manual intervention.

6. Testing the SSL Configuration:

- After completing the installation, Certbot tests the Apache configuration to ensure there are no syntax errors using apachectl configtest or httpd -t and restarts the Apache service to apply the new configuration.

```
[root@testing-server ~]# apachectl configtest
Syntax OK
[root@testing-server ~]# httpd -t
Syntax OK
[root@testing-server ~]#
```

- Certbot also verifies that the SSL certificate is correctly installed and can be served over HTTPS.

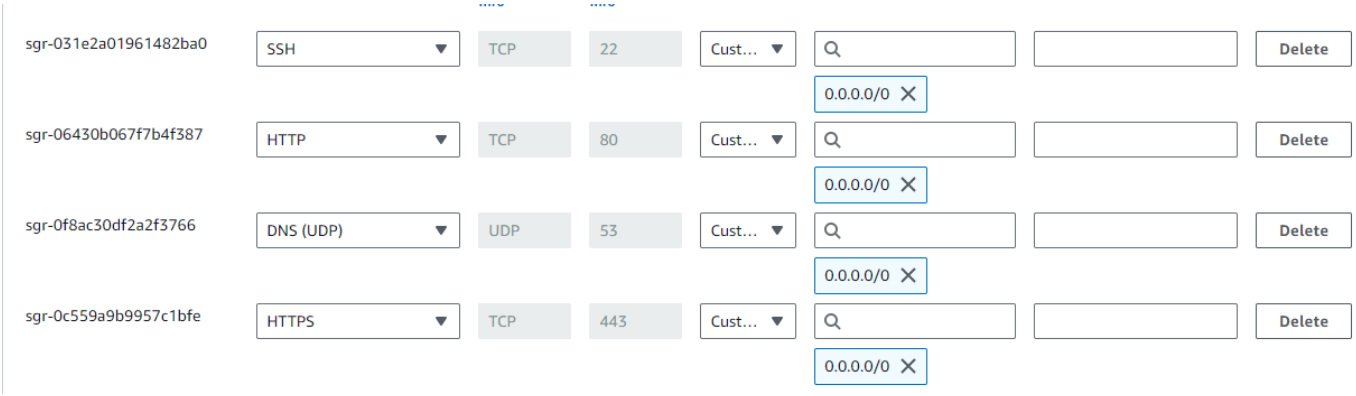
Key Effects of Running certbot --apache:

1. Your website becomes accessible over HTTPS, improving security by encrypting data transmitted between your server and the clients.

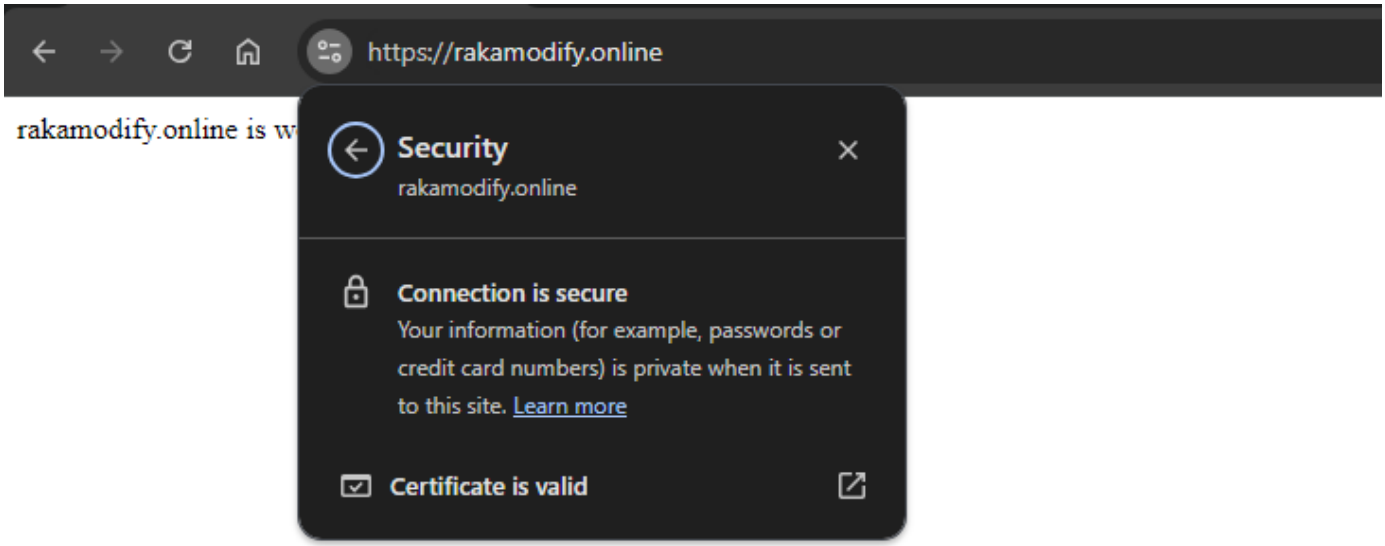
2. Apache is automatically reconfigured to support SSL, without manual editing of virtual host files.
3. HTTP requests are redirected to HTTPS, ensuring that users always use the secure version of your site.
4. Certificates are automatically renewed, keeping your website secure without further manual intervention.

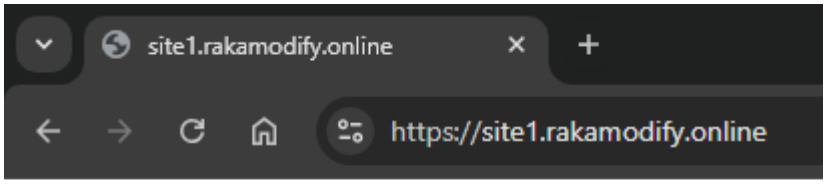
This process provides a seamless way to secure your website with SSL, enhancing security and user trust with minimal effort.

Don't forget to add 443/tcp port on inbound rule and reload the named and httpd service

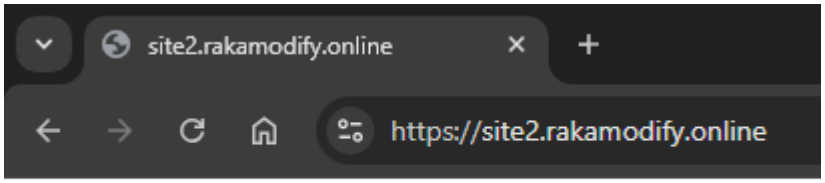


Now access the secure version of the website. (DNS and lets encrypt is)





site1.rakamodify.online is working



site2.rakamodify.online is working

- **NGINX (Installation and configuration of basic setup of NGINX)**

```
# yum install -y nginx
# systemctl enable --now nginx
# echo "This is website1" > /usr/share/nginx/html/index.html
# curl http://localhost
```

- **Reverse Proxy with NGINX (Nginx Reverse-Proxy)**

```
# vim /etc/nginx/nginx.conf
worker_processes 1;

events {
    worker_connections 1024;
}

http {

    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;
    sendfile            on;
    keepalive_timeout   65;
    # Load modular configuration files from the /etc/nginx/conf.d directory.
    # See http://nginx.org/en/docs/ngx_core_module.html#include
    # for more information.

    server {
        listen       80;
        server_name  192.168.199.168;     ##------------- Main Machine IP

        # Load configuration files for the default server block.

        location = / {
                proxy_pass http://192.168.199.168; ##------- Redirect Server IP
                index index.html index.htm;
        }

        }
    }

# setenforce 0
# systemctl restart nginx.service
# curl http://localhost
```

- **Load Balancing (Nginx Load-Balance, High Availability)**

What is Load Balancer ?

A load balancer is a networking device or software application that distributes and balances the incoming traffic among servers to provide high availability, efficient utilization of servers, and

high performance. It works as a "traffic cop" sitting in front of your server and routing client requests across all servers.

Importance of Load Balancer

1. **Performance and Latency Reduction**: By dividing user requests among multiple servers, user wait time is vastly cut down, resulting in better user experience.
2. **High Availability**: By distributing traffic across multiple servers, load balancers enhance the availability and reliability of applications. If one server fails, the load balancer redirects traffic to healthy servers.
3. **Scalability**: Load balancers facilitate horizontal scaling by easily accommodating new servers or resources to handle increasing traffic demands.

Types of Load Balancer Algorithms

1. **Round Robin**: This algorithm distributes client requests evenly across all servers in the pool. It is simple and effective when all servers have similar capacities.
2. **Weighted Policy**: This algorithm assigns a weight to each server based on its capacity. Servers with higher weights receive more client requests.
3. **Least Connection**: This algorithm directs traffic to the server with the fewest active connections. It is useful when there are significant and potentially variable computational loads on the servers.
4. **IP Hash**: This algorithm uses the client's IP address to determine which server should handle the request. This method ensures that a specific client is always directed to the same server as long as the server is available.

Load Balancer Practical

```
# vim /etc/nginx/conf.d/load.conf

upstream backend {          # backend is a group name for below mentioned servers
   server 54.175.218.1:8081;    # Public IP of Docker-Server:port
   server 54.175.218.1:8082;
      }
server {
   listen 80; # You may want to specify the appropriate port
   server_name 54.175.228.52; ## Localhost server ip
location / {
    proxy_pass http://backend;
      }
   }
```

- Setup MySQL WordPress complete website", so Follow this article of mine on Hashnode https://rakeshkumarjangid.hashnode.dev/deploy-wordpress-mariadb-php-apache-web-server-on-rhel9-with-easy-steps