DAY - 7

**✅ What is AMI?**

**🔹 Definition:**

**AMI (Amazon Machine Image)** is a **pre-configured template** provided by **Amazon Web Services (AWS)** that contains the **operating system, application server, application software, and related settings** needed to **launch a virtual server (EC2 instance)** in the cloud.

📌 Think of AMI as a **blueprint** to create virtual machines in AWS.

**🧠 What Does an AMI Include?**

* **Operating System** (e.g., Linux, Windows)
* **Application Server** (e.g., Apache, Nginx, Tomcat)
* **Software Packages** (e.g., Docker, Python, Java)
* **Custom Configurations** (e.g., environment variables, scripts)
* **Security Settings**

**💡 Scenarios Where We Use AMI**

**1. Auto Scaling**

When traffic increases, AWS auto-scaling launches new EC2 instances using the same AMI to ensure **consistent configuration**.

**2. Disaster Recovery**

Create regular AMIs of your instances. If a server crashes, you can quickly launch a new one using the AMI — like restoring from a backup.

**3. Environment Replication**

Use the same AMI to launch **identical environments** for dev, staging, and production — ensures consistency.

**4. Quick Deployment**

Instead of setting up everything manually, launch a pre-configured EC2 instance using your AMI — saves time.

**5. CI/CD Pipelines**

Build and bake custom AMIs with your application and configurations. Use them in CI/CD pipelines for **immutable deployments**.

**6. Infrastructure as Code (IaC)**

Tools like **Terraform**, **CloudFormation**, and **Packer** use AMIs to provision and manage infrastructure consistently.

**🛠️ Bonus Tips (DevOps Focus):**

* **Create custom AMIs** after setting up a base EC2 instance.
* **Tag and version** your AMIs for tracking.
* Use **Packer** to automate AMI creation as part of your CI/CD.
* Use **AMI IDs** in scripts, templates, and auto-scaling launch templates.

**🌐 What is an IP Address?**

An **IP (Internet Protocol) Address** is a unique number that identifies a **device** on a network. It’s like your device's **postal address** on the internet or within a private network.

An IP address is a unique number given to a device so it can communicate with other devices in a network or on the internet.

**Example: 192.168.1.10, 172.16.0.5, 8.8.8.8**

**🧠 Types of IP Addresses**

| **Type** | **Description** |
| --- | --- |
| **Public IP** | Used to identify devices on the **internet** |
| **Private IP** | Used inside **local networks** (home, cloud VPC) |
| **Static IP** | **Does not change** over time |
| **Dynamic IP** | Assigned **temporarily** (via DHCP) |
| **IPv4** | Old but widely used (e.g., 192.168.1.1) |
| **IPv6** | Newer version, longer format (e.g., 2001:db8::1) |

**📚 IP Address Classes (IPv4)**

IPv4 addresses are divided into **five classes** (A to E) based on the **first few bits** and **network size**.

**📌 Class A**

* **Range**: 1.0.0.0 to 126.255.255.255
* **Default Subnet Mask**: 255.0.0.0
* **Used For**: **Very large networks**
* **Private Range**: 10.0.0.0 – 10.255.255.255

**📌 Class B**

* **Range**: 128.0.0.0 to 191.255.255.255
* **Default Subnet Mask**: 255.255.0.0
* **Used For**: **Medium-sized networks**
* **Private Range**: 172.16.0.0 – 172.31.255.255

**📌 Class C**

* **Range**: 192.0.0.0 to 223.255.255.255
* **Default Subnet Mask**: 255.255.255.0
* **Used For**: **Small networks**
* **Private Range**: 192.168.0.0 – 192.168.255.255

**📌 Class D (Multicast)**

* **Range**: 224.0.0.0 to 239.255.255.255
* **Used For**: **Multicast group communication**, not standard devices

**📌 Class E (Experimental)**

* **Range**: 240.0.0.0 to 255.255.255.255
* **Used For**: **Research, future use**

**🔐 Reserved & Special IPs**

| **IP Address** | **Purpose** |
| --- | --- |
| 127.0.0.1 | Loopback (localhost) |
| 0.0.0.0 | Default or unknown address |
| 255.255.255.255 | Broadcast address |

**📦 Useful Concepts**

| **Term** | **Meaning** |
| --- | --- |
| **Subnet Mask** | Defines how IP is split into **network** and **host** parts |
| **CIDR** | Modern notation, e.g., 192.168.1.0/24 |
| **DHCP** | Assigns **dynamic IPs** to devices automatically |
| **NAT** | Allows **private IPs to access the internet** via Public IP |

**🌐 What is Public IP and Private IP?**

**🔹 Public IP Address**

A **Public IP** is the IP address that is **visible to the internet**. It is assigned by your **Internet Service Provider (ISP)** or **cloud provider** (like AWS) and is **globally unique**.

📡 It's like your **house address** on the internet — anyone in the world can find and communicate with it.

**🔹 Private IP Address**

A **Private IP** is used **inside a local network** (like your home Wi-Fi, office network, or cloud VPC). It is **not visible to the internet** directly and is used for internal communication.

🏠 It's like a **room number inside your house** — only useful within the building.

**🛠️ Technical Ranges (Private IPs):**

According to **RFC 1918**, private IPs come from these ranges:

* 10.0.0.0 – 10.255.255.255
* 172.16.0.0 – 172.31.255.255
* 192.168.0.0 – 192.168.255.255

**✅ Understanding the Relationship:**

**"Linux is a server, and Nginx is installed in it but we can't directly see it."**

**🔸1. Linux is a Server (Operating System + Host Environment)**

* When we say “Linux is a server,” we usually mean:
  + A **machine running the Linux operating system** (like Ubuntu, CentOS).
  + It can be a **physical server**, a **virtual machine**, or a **cloud instance** (like AWS EC2).
* This Linux server can host and manage various **applications/services**.

**🔸2. Nginx is a Web Server Software (Installed in Linux)**

* **Nginx** is a **software** that runs as a **background service (daemon)** inside the Linux OS.
* After installing Nginx, it doesn’t create a separate "window" or GUI – it works **silently in the background**.
* It listens on ports like **80 (HTTP)** or **443 (HTTPS)** to serve web content.

**🔹 Real-Time Analogy:**

Imagine:

* **Linux** = A shop (with an address and open sign)
* **Nginx** = A chef working inside the kitchen
* From outside, customers see the shop (Linux server).
* But they **don’t see the chef** (Nginx), even though the chef is **preparing and serving food** (web content).

**🔹 Real-Time Scenario:**

**Step-by-Step:**

1. You launch an **EC2 instance with Ubuntu (Linux)**
2. You install Nginx:
3. sudo apt/dnf update
4. sudo apt/dnf install nginx
5. Nginx starts running in the **background**
   * You won’t "see" Nginx unless you check:
   * ps aux | grep nginx
   * systemctl status nginx
6. You access the server via a browser:
   * http://<your-server-ip>
   * The **Nginx welcome page** appears.
   * This confirms Nginx is **working**, even though it’s **not visible** like a desktop app.

**✅ Conclusion:**

* **Linux is a server** that acts as the **platform/host**.
* **Nginx is a web server software**, installed and running **inside** Linux.
* You don’t see Nginx like a graphical app — it runs silently in the background and **responds to requests**.

**DNF** is the **modern, faster, and improved version of YUM**, introduced in newer Red Hat-based Linux distributions.

| **Feature** | **YUM (Yellowdog Updater Modified)** | **DNF (Dandified YUM)** |
| --- | --- | --- |
| **Used in** | RHEL/CentOS 6 & 7 | RHEL/CentOS 8+, Fedora 22+ |
| **Speed** | Slower dependency resolution | Faster & better performance |
| **Dependency Handling** | Less efficient, can be buggy | Better dependency resolution |
| **Python Version** | Python 2 | Python 3 |
| **Features** | Limited | Modern CLI, better API, rollback support |
| **Actively Maintained?** | No | Yes (DNF replaces YUM in newer systems) |
| **Modular Support** | ❌ No modularity support | ✅ Fully supports **modular repositories** |

* A **module** in DNF is a collection of packages grouped together with multiple **versions or streams**.
* It allows you to **choose which version** (stream) of a software you want to install.

dnf supports **modular package management**, allowing **version selection**, while yum does **not** support modules.

**01-Frontend**

The frontend is the service in **RoboShop** to serve the web content over Nginx. This will have the webframe for the web application.

This is a static content and to serve static content we need a web server. This server

Developer has chosen Nginx as a web server and thus we will install Nginx Web Server.

**🔧 Step-by-Step Explanation**

**📌 1. dnf module list nginx**

**Purpose:**

* Lists all available **streams (versions)** of the Nginx module in the DNF modular repository.

**Output Example:**

Name Stream Profiles Summary

nginx 1.20 [d] common nginx webserver

nginx 1.24 common nginx webserver

Shows different versions (streams) of Nginx you can install.

**📌 2. dnf module disable nginx -y**

**Purpose:**

* Disables the **default stream (e.g., 1.20)** of the Nginx module, so you can enable a different version.

Without disabling the default, you can't enable another stream.

**📌 3. dnf module enable nginx:1.24 -y**

**Purpose:**

* Enables the specific Nginx **stream version 1.24**.

This prepares DNF to install **Nginx 1.24**, instead of the default version.

**📌 4. dnf install nginx -y**

**Purpose:**

* Installs **Nginx version 1.24** from the module stream you just enabled.

The -y flag auto-confirms the installation.

**📌 5. systemctl enable nginx**

**Purpose:**

* Enables **Nginx service** to **start automatically on boot**.

This means after a restart, Nginx will start by itself.

You can check which version of **Nginx** is installed using this command:

nginx -v

**📌 6. systemctl start nginx**

**Purpose:**

* Starts the **Nginx web server** **now**, without rebooting.

Nginx is now actively listening on port **80** (HTTP).

**📌 7. ps -ef | grep nginx**

**Purpose:**

* Checks if the Nginx process is running.

**Sample Output:**

root 1234 1 0 10:00 ? 00:00:00 nginx: master process /usr/sbin/nginx

nginx 1235 1234 0 10:00 ? 00:00:00 nginx: worker process

Confirms that Nginx is active and running in the background.

**📌 8. netstat -lntp**

**Purpose:**

* Lists **listening ports**, **TCP connections**, and **associated services**.

**Sample Output:**

Proto Recv-Q Send-Q Local Address Foreign Address State PID/Program name

tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 1234/nginx

Shows that Nginx is **listening on port 80**, ready to serve HTTP requests.

**📌 9. You open the public IP in a browser:**

http://<your-ec2-public-ip>

**What Happens:**

* Your browser sends an HTTP request to the public IP.
* The request hits **port 80**.
* Nginx is listening and serves the **default welcome page**.

✅ You now **see the Nginx welcome screen**, which confirms the server is working properly.

**🧠 Final Summary (One Line Each):**

| **Command** | **What It Does** |
| --- | --- |
| dnf module list nginx | Shows available Nginx versions |
| dnf module disable nginx | Disables default version |
| dnf module enable nginx:1.24 | Enables specific version |
| dnf install nginx | Installs Nginx |
| systemctl enable nginx | Auto-starts Nginx on boot |
| systemctl start nginx | Starts Nginx service now |
| `ps -ef | grep nginx` |
| netstat -lntp | Confirms Nginx is listening on port 80 |
| Open IP in browser | Shows Nginx welcome page |

**📁 Contents of /etc/nginx/**

Here’s what you typically find inside:

| **File/Directory** | **Purpose** |
| --- | --- |
| nginx.conf | 🔧 **Main configuration file** — it controls global settings, logging, events, and which sites are served. |

**📌 What is /etc/nginx/nginx.conf?**

This is the **main configuration file** for the Nginx web server. It tells Nginx:

* How to handle client requests
* What ports to listen on
* Where your website files are
* Where to store logs
* How many worker processes to use
* And much more...

**📂 Location:**

/etc/nginx/nginx.conf

You can open it using:

sudo vi /etc/nginx/nginx.conf

**🔍 Key Configurations You Can Do**

| **Configuration** | **Purpose** |
| --- | --- |
| listen 80; | Defines which port Nginx listens on (default is 80) |
| server\_name | Name of the website (like example.com) |
| root | Folder path where website files (HTML, etc.) are stored |
| index | Default file to load (e.g., index.html) |
| access\_log, error\_log | Log file locations |
| worker\_processes, worker\_connections | Performance tuning |
| location block | URL pattern handling (routing, reverse proxy, etc.) |

**🛠️ How to Change Nginx Port Number**

Let’s say you want to change the port from **80 to 8080**.

**✅ Step-by-step:**

1. Open the config:

sudo vi /etc/nginx/nginx.conf

1. Look for the server block under the http section:

server {

listen 80;

...

}

1. Change it to:

server {

listen 8080;

...

}

1. Save and exit the file (:wq in vi editor)

**✅ Test the Configuration**

Always test before restarting:

nginx -t

✅ Output should say:

nginx: configuration file /etc/nginx/nginx.conf test is successful

**🔄 Restart or Reload Nginx**

To apply changes:

sudo systemctl reload nginx

Or restart fully:

sudo systemctl restart nginx

**🔍 Accessing Nginx on New Port**

Now access your server via:

http://your-server-ip:8080

**🧠 Final Summary:**

| **Task** | **Command** |
| --- | --- |
| Open config | sudo vi /etc/nginx/nginx.conf |
| Change port | Edit listen directive in server block |
| Test config | nginx -t |
| Reload service | sudo systemctl reload nginx |
| Access web | http://IP:PORT |

**📁 What is /usr/share/nginx/html?**

/usr/share/nginx/html is the **default web root directory** used by the **Nginx web server**. It is the place where you put the website files (like .html, .css, .js) that you want to serve to users when they visit your server's IP or domain.

**📌 Purpose**

| **Key Role** | **Description** |
| --- | --- |
| 🌐 **Web root folder** | It is the default folder where Nginx looks for web pages to serve. |
| 🧾 **Contains static files** | Typically stores files like index.html, error pages, images, etc. |
| 📍 **Referenced in Nginx config** | It is defined in nginx.conf or other .conf files as the root location. |

**📄 What is index.html in /usr/share/nginx/html/?**

index.html is the **default homepage** file that **Nginx serves to users** when they access your server's IP address or domain **without specifying a file**.

**📌 In Simple Words:**

When someone visits:

http://your-server-ip/

Nginx automatically looks for the file:

/usr/share/nginx/html/index.html

and shows it in the browser. This is because it's set as the **default index page** in the Nginx configuration.

**✅ Step-by-Step Developer Actions to Deploy Frontend on Nginx**

**✅ Step 1: Remove default content**

You're removing the existing default index.html and other files:

sudo rm -rf /usr/share/nginx/html/\*

🧹 This ensures that:

* The default "Welcome to nginx!" page is deleted
* You start with a clean directory for your custom frontend

**✅ Step 2: Download the frontend content**

curl -o /tmp/frontend.zip https://roboshop-artifacts.s3.amazonaws.com/frontend-v3.zip

📥 This does the following:

* Uses curl to download a frontend zip file
* Saves it to /tmp/frontend.zip
* curl -o means "output to file"

**✅ Step 3: Extract the frontend content**

cd /usr/share/nginx/html

sudo unzip /tmp/frontend.zip

📦 This unzips the website content into the Nginx default web root. After this:

* index.html and all assets (CSS, JS, images) are extracted into /usr/share/nginx/html
* Nginx will now serve this frontend when users access the public IP

**🌐 Real-Time Test:**

1. ✅ Make sure Nginx is running:

sudo systemctl start nginx

1. ✅ Enable on boot:

sudo systemctl enable nginx

1. ✅ Open in browser:

http://<your-public-ip>

You should now see the **custom frontend interface** from the zip you downloaded.

**✅ Why do we open and edit /etc/nginx/nginx.conf as developers?**

When deploying a real-world application or frontend through Nginx, **developers may provide custom Nginx configuration code**. They usually ask you to **replace the default config** with this custom one.

**📄 Default Nginx Config Location**

* The main configuration file for Nginx is:

/etc/nginx/nginx.conf

* It controls:
  + Which port Nginx listens on (e.g., 80)
  + Which directory to serve content from
  + Log locations
  + Reverse proxy setup
  + Caching, compression, etc.

**🎯 Why Do We Replace the Config?**

| **Reason** | **Explanation** |
| --- | --- |
| ✅ To match app requirements | The default config is very basic. The new one is optimized for your app’s routing, performance, and security. |
| ✅ Custom routing rules | The app may need Nginx to reverse proxy to backend services like /api, /cart, etc. |
| ✅ Enable compression, caching | To make the website load faster |
| ✅ Connect to backend ports | If you're working in a microservices architecture (like Roboshop), the frontend might talk to other services like catalog, user, etc. via Nginx |

**✅ Step 4: Replace the Default Nginx Configuration**

If your developer has provided a custom Nginx configuration, follow these steps:

**🔧 Instructions:**

sudo vim /etc/nginx/nginx.conf

**🛠 Inside vim:**

1. Press Esc, then type :%d → Deletes all lines
2. Press i to enter **Insert Mode**
3. Paste the configuration given by the developer (right-click or Shift+Insert)
4. Press Esc, then type :wq to save and exit

**✅ Step 5: Restart Nginx to Apply the Changes**

Once the config is saved:

sudo systemctl restart nginx

This:

* Reloads the updated config
* Starts serving the custom frontend properly

**✅ Step 6: Domain Purchase from Hostinger**

🛒 **Action**: Purchased a custom domain from **Hostinger** **dkoushik.store**

📌 **Purpose**: To map the domain name to the Nginx web server's public IP so users can access the frontend using a user-friendly name instead of an IP address.

**🌐 Who gives the IP address for facebook.com?**

When you type facebook.com, a system called the **DNS (Domain Name System)** is responsible for giving you the correct IP address.

**🔄 Step-by-Step Flow:**

1. **You type** facebook.com in your browser
2. Your browser asks your **Operating System**
3. Your OS asks the **DNS Resolver** (usually your ISP or Google DNS like 8.8.8.8)
4. The Resolver checks if it has the IP cached.
   * If not, it goes to the **Root DNS Server**
5. Root server directs it to **.com TLD server**
6. That directs it to **Facebook’s Authoritative DNS Server**
7. The **Authoritative DNS Server** gives the **actual IP address** of facebook.com

**🌐 What is DNS?**

**DNS** stands for **Domain Name System**.

It is like the **phonebook of the internet** — it helps translate human-friendly domain names (like google.com) into IP addresses (like 142.250.192.14) that computers use to talk to each other.

**🧠 Simple Analogy**

* You type: www.koushikstore.com in your browser
* DNS translates it to something like: 13.235.118.239
* Now your browser knows **where** to send the request to get the website content

**🧠 What is Cache?**

**Cache** is **temporary storage** used to **store frequently accessed data** so that future requests for that data are faster.

**✅ If we type facebook.com in a browser, who gives the IP address?**

Here’s the **detailed step-by-step flow**:

**🔁 Step-by-Step DNS Resolution Flow**

1. **Browser Cache**
   * Your browser checks: "Have I recently resolved facebook.com?"
   * If **yes**, it uses the cached IP address (faster).
   * If **no**, it goes to next step.
2. **OS to OS Cache (Local DNS Cache)**
   * It checks OS then
   * Your operating system checks its own local cache.
   * If IP is stored, it gives the answer.
   * If not found → continue to step 3.
3. **DNS Resolver (from ISP like Airtel, Jio or Google DNS like 8.8.8.8)**
   * This is the first external query.
   * The DNS resolver takes responsibility for finding the IP.

**🌐 The DNS Resolver Works Like This:**

1. **Contacts Root DNS Servers**
   * There are **13 root server clusters** managed globally (IANA/ICANN)
   * Root servers don’t know exact IP, but know where .com info is.
2. **Redirects to TLD DNS Server**
   * For facebook.com, it goes to .com **Top Level Domain (TLD)** server
   * It says: "I know who manages domains ending with .com"
3. **Redirects to Authoritative Name Server**
   * Example: ns1.facebook.com, ns2.facebook.com
   * These are **Facebook’s DNS servers** (name servers)
   * They **own the final record** and give:
   * facebook.com → 157.240.1.35
4. **IP Returned to Browser**
   * The resolver sends this IP back to your OS, then to your browser
   * Your browser now knows where to send the HTTP request

**📌 Summary Table:**

| **Step** | **What Happens** |
| --- | --- |
| Browser Cache | Checks if recently resolved |
| OS Cache | Checks local DNS entries |
| DNS Resolver (ISP) | Starts the resolution journey |
| Root Servers | Directs to .com TLD |
| TLD Servers | Direct to facebook's name servers |
| Name Servers | Give the final IP of facebook.com |
| Response to Browser | Website loads using that IP |

Amazon Route 53 is a highly available and scalable Domain Name System (DNS) web service provided by AWS. It’s used to manage domain names and direct traffic to applications, both inside and outside AWS.

**🔧 What Route 53 Does:**

1. Domain Registration  
   You can register domain names (e.g., example.com) directly through Route 53.
2. DNS Routing (DNS Service)  
   It translates human-readable domain names (like www.example.com) into IP addresses (like 192.0.2.1) that computers use to connect to each other.
3. Health Checking & Failover  
   It can monitor the health of endpoints (like web servers) and automatically route traffic to healthy ones.

**🧭 Types of Routing Policies in Route** 53:

| **Policy Type** | **Description** |
| --- | --- |
| Simple Routing | Routes traffic to a single resource. |
| Weighted Routing | Splits traffic based on weights. Useful for A/B testing. |
| Latency-Based Routing | Routes to the lowest-latency AWS regison. |
| Failover Routing | Directs to a backup if the primary fails. |
| Geolocation Routing | Routes based on user location. |
| Multi-Value Answer | Similar to simple routing, but can return multiple IPs for health checks. |

🌐 **Common Use Cases:**

* Hosting a website with a custom domain (e.g., www.myshop.com)
* Distributing traffic globally to the nearest AWS region
* Setting up automatic failover for high availability
* Managing DNS for hybrid environments (AWS + on-premise)

🏷️ **Why It's Called “Route 53”?**

The number 53 comes from TCP/UDP port 53, which is the port used for DNS communication.

**✅ Goal:**

You're keeping the domain **registered with Hostinger**, but you want **AWS Route 53** to handle **DNS (domain records)** like A, CNAME, etc.

**🔁 Step-by-Step: Point Domain from Hostinger to Route 53**

**🔹 Step 1: Create a Hosted Zone in Route 53**

1. Go to **AWS Console → Route 53**
2. Click **"Hosted Zones"** → **"Create Hosted Zone"**
3. Enter your domain name (e.g., koushikdharavath.com)
4. Type: **Public Hosted Zone**
5. Click **Create**

✅ Route 53 creates a hosted zone with **4 NS records** and a **SOA record**

**🔹 Step 2: Copy Route 53 Name Servers**

* In the hosted zone, look at the **NS record** (e.g., ns-123.awsdns-45.com, etc.)
* Copy **all 4 name servers**

**🔹 Step 3: Update DNS (Name Servers) in Hostinger**

1. Login to your **Hostinger account**
2. Go to **Domains → Your Domain**
3. Click **DNS / Nameservers Settings**
4. Select **Use custom nameservers**
5. Paste the **4 Route 53 NS records** you copied
6. Save the changes

📌 It may take **few minutes to 24 hours** to fully propagate.

**🔹 Step 4: Add Records in Route 53**

* In the **Route 53 Hosted Zone**, add DNS records like:

| **Type** | **Name** | **Value** |
| --- | --- | --- |
| A | @ or koushik.com | Your EC2 IP or Load Balancer IP |

**✅ Most Common DNS Record Types (with Real-Time Use Cases)**

| **Type** | **Full Form** | **Purpose** | **Real-Time Example** |
| --- | --- | --- | --- |
| **A** | Address Record | Maps domain name → **IPv4 address** | facebook.com → 157.240.1.35 |
| **AAAA** | IPv6 Address Record | Maps domain name → **IPv6 address** | google.com → 2404:6800:4009:80f::200e |
| **CNAME** | Canonical Name | Points one domain name to **another domain name** | www.koushik.com → koushik.com |
| **MX** | Mail Exchange | Defines mail server for sending/receiving emails | gmail.com → ALT1.GMAIL-SMTP-IN.L.GOOGLE.COM |
| **TXT** | Text Record | Stores text for SPF, DKIM, domain verification | Used by **Google**, **AWS**, etc. to verify ownership |
| **NS** | Name Server | Shows which DNS servers manage your domain | ns1.route53.aws.com |
| **SRV** | Service Record | Defines services like SIP, VoIP, or MS Teams | \_sip.\_tcp.domain.com → 5060 |
| **CAA** | Certificate Authority Authorization | Controls which SSL provider can issue a cert for your domain | Only Let's Encrypt or AWS allowed to create SSL |

**✅ What is TTL in DNS?**

**TTL** stands for **Time To Live**.  
It defines **how long (in seconds)** a DNS record is **cached** by:

* DNS resolvers (like Google DNS)
* Browsers
* Operating systems

**🔍 Real-Time Example:**

Let’s say you have this A record:

Type: A

Name: www.koushik.com

Value: 54.123.45.67

TTL: 300

This means:

* The IP address 54.123.45.67 will be **cached for 300 seconds** (5 minutes).
* After 5 minutes, if someone tries to visit your domain again, the DNS resolver will **query Route 53** (or your DNS server) again.

**📌 Why is TTL Important?**

| **Use Case** | **TTL Value** | **Reason** |
| --- | --- | --- |
| **Changing servers frequently** | Low (60–300s) | So DNS picks up new IP faster |
| **Stable websites** | High (86400s) | Reduce DNS queries, better performance |
| **Testing new config** | Low (60s) | Helps test changes quickly |

**⚙️ Common TTL Values**

| **TTL (Seconds)** | **Description** |
| --- | --- |
| 60 | 1 minute (great for testing) |
| 300 | 5 minutes (good balance) |
| 3600 | 1 hour |
| 86400 | 1 day |

**🧠 Summary:**

TTL controls **how long DNS results are cached**.  
Lower TTL = **faster updates**, higher TTL = **better performance**.