# Network Basic Learning

## 🔌 1. What is a Computer Network?

A **network** is when multiple devices (computers, phones, servers) are connected together to **communicate** and **share resources** (data, applications, files).
In software development, networks let your backend service talk to:

- Other services (e.g., Kafka, databases, APIs)
- Frontend apps (mobile, web)
- External users (via the internet)

## 🌐 2. Internet vs Intranet

- **Internet**: A public network (like highways) that connects everyone globally.
- **Intranet**: A private network (like a company's internal road system).

In cloud:

- Internet = access from users.
- Intranet = internal service communication (e.g., microservices within a VPC in AWS).

# 🧭 3. IP Address (Internet Protocol Address)

## 📌 What is it?

An **IP address** is a unique identifier assigned to each device on a network — like a phone number for your computer, mobile, or server.
There are two main types:

- **IPv4**: e.g., 192.168.1.100 (most common)
- **IPv6**: e.g., 2001:0db8:85a3::8a2e:0370:7334 (newer, more addresses)

## 📁 Types of IPs:

| Type | Used For |
|---|---|
| **Public IP** | Accessible from anywhere over the Internet |
| **Private IP** | Used inside a private network (e.g., in AWS VPC or office network) |

## 🧑‍💻 Developer Example:

You're building a backend in Spring Boot and hosting it on AWS:

- The **EC2 instance** might have:
  - Private IP: 10.0.0.5 (used to talk to DB inside VPC)
  - Public IP: 3.108.23.101 (used for external access)

So when a frontend React app calls your API over the internet, it uses the **public IP** (or mapped domain).

# 🔠 4. DNS (Domain Name System)

## 📌 What is it?

DNS is like a **phonebook for the Internet**. It converts **domain names** (easy to remember) into **IP addresses** (used by machines).

- You type: www.google.com
- DNS resolves it to: 142.250.193.4
- Your browser connects to that IP.

## 🧠 How it works:

1. Browser checks DNS cache.
2. If not found, it asks a **DNS server** (like Google DNS 8.8.8.8).
3. Gets IP in response and sends the request.

## 🧑‍💻 Developer Example:

Let's say you deploy a Spring Boot app with an API:

## https://api.myparcelapp.com/orders

Internally:

- DNS resolves api.myparcelapp.com to your EC2 instance IP (e.g., 13.233.44.55)
- Then browser sends request to 13.233.44.55 over port 443

You can also use tools like **nslookup** or **dig** to see DNS records.

# 🧱 5. Proxy Server

## 📌 What is it?

A **proxy server** acts as a **middleman** between the client and the real server.
Clients don't talk directly to the backend — they go through a proxy.

## 📦 Why use a Proxy?

- 🔒 **Security**: Hide server IPs
- 🚀 **Caching**: Serve repeated responses quickly
- 📊 **Monitoring**: Log and filter requests
- 🚫 **Control**: Block unwanted content (parental filters, enterprise)

## 🔄 Types:

- **Forward Proxy**: Client-side (e.g., browser uses a proxy to access websites)
- **Reverse Proxy**: Server-side (used in backend or cloud setup)

## 🧑‍💻 Developer Example:

In cloud/backend:

- You deploy your app behind **Nginx** or **API Gateway (AWS)**.
- Clients hit the **proxy endpoint**, which forwards the request to your real Spring Boot service.

## Client --> [Nginx proxy at api.myparcelapp.com] --> [Spring Boot App running on localhost:8080]

In code (Nginx):

```
location /api/ {
    proxy_pass http://localhost:8080/;
}
```

## 📡 8. HTTP/HTTPS

## 📌 What is HTTP?

**HTTP (Hypertext Transfer Protocol)** is the protocol used for communication between a **client** (browser, Postman) and a **server** (e.g., your Spring Boot backend).

## ✅ HTTP is:

- **Stateless**: Each request is independent.
- **Plaintext**: Not secure. Anyone can read the content in transit (bad for passwords, tokens).

## 🔒 HTTPS = HTTP + TLS (Transport Layer Security)

- Encrypts all communication.
- Data is secure between client and server.
- Used by banks, login forms, secure APIs.

## 🧑‍💻 Developer Example:

**Backend API**

```
@GetMapping("/orders")
public List<Order> getAllOrders() {
    return orderService.getOrders();
}
```

**Frontend call (React)**

```
fetch("https://api.myparcelapp.com/orders")
```

If the API runs over **HTTPS**, here's what happens:

1. TLS handshake establishes a secure connection.
2. Browser encrypts the request.
3. Server decrypts it and sends response securely.

HTTPS requires an **SSL certificate**, like Let's Encrypt or AWS ACM.

## 🌐 Real Request Flow Summary:

Here's what happens when a user accesses your app:

```
User browser
  |
  |--- types api.myparcelapp.com
  |
DNS lookup
  |
  |--- resolves to 13.233.44.55
  |
HTTP/HTTPS request
  |
  |--- hits Load Balancer / Nginx Proxy
  |
  |--- forwards to Spring Boot App (with IP:Port)
  |
  |--- responds back to browser
```

## 🔐 6. Firewall

A **firewall** allows or blocks traffic based on rules.
As a backend dev:

- You set **security groups** (AWS) or **network policies (Kubernetes)** to allow only specific IPs or ports.

## 🚪 7. Port

A **port** is like a **door on a server** used by applications.

- Web servers → Port 80 (HTTP), Port 443 (HTTPS)
- DB like MySQL → Port 3306

IP tells **where** to go, Port tells **which application** to reach.

## 📡 8. HTTP/HTTPS

These are **protocols** (rules) for web communication.

- **HTTP**: Not secure
- **HTTPS**: Secure (encrypted using TLS/SSL)

When you call a REST API, you're using HTTP or HTTPS.

## 📫 9. TCP vs UDP

- **TCP**: Reliable, ordered delivery (e.g., API calls, DB communication)
- **UDP**: Faster, less reliable (e.g., video streaming, DNS)

TCP is like postal service with receipt; UDP is like throwing a flyer.

## 💼 10. Packet

When you send data over a network, it gets broken into **packets**.
- Your 1 MB image → becomes multiple small packets.
- These packets are reassembled by the receiver.

## 🕸️ 11. Load Balancer

Distributes incoming traffic across multiple servers (like traffic police).
Used in:
- Cloud apps (e.g., AWS Elastic Load Balancer)
- Scaling microservices

## 🧊 12. NAT (Network Address Translation)

Used to allow **private IPs** to communicate with the public internet by mapping to a **public IP**.
In cloud:
- Your EC2 inside VPC uses NAT to call internet without exposing itself directly.

## 💼 13. VPN (Virtual Private Network)

Secure tunnel between two networks or between a client and a network.
- Developers use VPN to access production servers.
- Companies use VPN to connect remote teams securely.

## 🛡️ 14. CDN (Content Delivery Network)

Distributes content (images, videos, JS/CSS files) closer to users.
- Faster page loads.
- Cloudflare, Akamai, AWS CloudFront are examples.

| Term | Think Of It As... |
|---|---|
| IP Address | Phone number of a device |
| DNS | Contact list that maps names to IPs |
| Proxy | Middleman for requests |
| Firewall | Security guard for your system |
| Port | Doorway to a specific app |
| Load Balancer | Traffic controller |
| NAT | Translator between private and public world |
| CDN | Fast delivery service |
| TCP | Safe parcel with receipt |
| UDP | Fast flyer, no tracking |

## 🔁 Scenario: You type **www.google.com** in your browser and hit Enter

Let's break this into clear stages:

## 🧠 Step 1: You Type the URL

You type **www.google.com** in your browser.

### What your system does:

- Parses the URL:
  - **Protocol**: https
  - **Domain**: www.google.com
  - **Resource**: / (default home page)
- Your browser prepares to send an **HTTP(S)** request.

## 🔠 Step 2: DNS Resolution (Converting Domain → IP Address)

### What happens:

- Your browser asks: *"What is the IP of www.google.com?"*

### DNS lookup process:

1. **Browser cache**: Has this domain been resolved recently?
2. **OS cache**: If not, check local operating system cache.
3. **Router cache**: Still not found? Ask the router's DNS cache.
4. **ISP DNS server**: Your internet provider (like Airtel, Jio) has a DNS server (e.g., 1.1.1.1 or 8.8.8.8).
5. **Recursive query**: If not cached, DNS server asks root, TLD, and authoritative DNS servers.

🔁 Eventually, **www.google.com** resolves to something like:

### 142.250.193.4

## Dev analogy:

It's like looking up a contact name (**www.google.com**) to get their phone number (**142.250.193.4**) before calling them.

## 📡 Step 3: TCP + TLS Handshake (Creating a Connection)

## Since it's HTTPS:

- Browser initiates a **TCP connection** with Google's server on port 443.
- Then performs **TLS handshake**:
  - Exchange certificates (Google proves it's legit).
  - Negotiate encryption algorithms.
  - Establish a **secure, encrypted channel**.

🔐 This ensures **confidentiality + integrity** of data.

## 🌐 Step 4: Sending the HTTP Request

## Now, your browser sends an encrypted HTTPS GET request like:

## GET / HTTP/1.1

## Host: www.google.com

## User-Agent: Chrome/125.0

This is sent to the IP **142.250.193.4**, on port **443**, using TCP.

## 🏢 Step 5: Request Hits Google's Infrastructure

## Google's Cloud & Load Balancing kicks in:

- **Reverse Proxy / Load Balancer** receives the request.
- Routes it to one of many available web servers (closest region, least load).
- Your request hits a **Google web server**.

## 🧠 Step 6: Google Processes the Request

- The server:
  - Parses your request
  - Checks cookies, user-agent, etc.
  - Generates HTML or dynamic content
  - Calls databases or other internal services if needed

## 📤 Step 7: Google Sends Back a Response

## Response:
## HTTP/1.1 200 OK
## Content-Type: text/html
## Content-Length: 14,567

- Browser receives HTML content.
- Starts downloading linked CSS, JS, and images (each goes through same DNS + HTTPS steps).
- Renders the page visually.

## 🧑‍💻 Step 8: You See Google Homepage

Now the browser has rendered the response — and you see the Google homepage.

## 🎯 Behind-the-Scenes Key Concepts for Developers

| Concept | What's Happening |
|---|---|
| DNS | Converts domain name → IP address |
| IP Address | Google server's location on the internet |
| TCP/IP | Guarantees packet delivery & order |
| HTTPS/TLS | Encrypts the entire communication |
| Ports | Port 443 is used for HTTPS |
| Load Balancer | Routes traffic to correct server in data center |
| CDN | Caches and delivers static assets faster |
| Browser cache | Stores previously visited resources locally |

## 🌍 What is your IP address in this process?

Your **public IP address** is the one assigned by your ISP. Google sees **this IP** when your request hits its server.
You can check it with: https://whatismyipaddress.com/
In a local network:
- Your laptop may have a private IP: 192.168.1.20
- Router does **NAT (Network Address Translation)** to map your private IP → public IP

So, when your request goes out:

## [Your laptop 192.168.1.20] → [Router NAT] → [Public IP 103.23.45.11] → Google Server

## 📦 Complete Summary in Software Dev Flow

**Browser → DNS Lookup (what is google.com?)**

**→ Gets IP: 142.250.193.4**
**→ TCP Handshake + TLS (secure connection)**
**→ HTTPS GET Request to google.com**
**→ Hits Load Balancer/Proxy**
**→ Routed to Web Server**
**→ HTML Response returned**
**→ Browser renders the page**

## ✅ What Are HTTPS Certificates?

An **HTTPS certificate** (a.k.a. **SSL/TLS certificate**) is a **digital identity** for a server. It's used in **HTTPS** to:

1. **Prove the server is who it claims to be** (Authentication)
2. **Encrypt the communication** so it can't be read by attackers (Encryption)
3. **Ensure data wasn't altered during transmission** (Integrity)

## 🌐 Real-Life Analogy

Imagine you're talking to a bank over the phone:

- You want to **know it's really the bank**, not a scammer → (Authentication)
- You speak in a **private language** so no one can listen in → (Encryption)
- You want to be sure your message isn't changed → (Integrity)

That's exactly what an HTTPS certificate ensures between a browser/client and server.

# 🧩 What's Inside an HTTPS Certificate?

Example: A certificate for api.myapp.com

| Field | Example Value |
|---|---|
| **Common Name** | api.myapp.com |
| **Issuer** | Let's Encrypt, GoDaddy, etc. |
| **Public Key** | (used to encrypt data) |
| **Valid From / To** | Expiration details |
| **Signature** | Digital signature by CA |

Think of this like a government-issued ID card for your server.

# 🔐 How HTTPS Works with Certificates (Step by Step)

Let's say a client (browser or another service) calls:
https://api.myapp.com/orders

Here's what happens:

# 🧭 Step 1: TLS Handshake

Before any data is exchanged:

1. **Client connects to server** (e.g., api.myapp.com:443)
2. Server sends its **certificate**
3. Client checks:
   - Is the certificate valid (not expired)?
   - Does it match the domain?
   - Is it signed by a **trusted Certificate Authority (CA)**?

If all checks pass ✅, a secure encrypted session begins.

# 💡 How This Applies to Spring Boot Projects on Two Servers

Let's say you have:

| Service | Domain | Server IP |
|---|---|---|
| order-service | orders.myapp.com | 3.100.45.77 |
| payment-service | payments.myapp.com | 3.105.99.88 |

# 🔄 Communication Flow

order-service (client) needs to call payment-service (server):
GET https://payments.myapp.com/payments/123

# Without certificate:

- Anyone could impersonate payments.myapp.com
- Man-in-the-middle attacks possible
- No encryption
-

# With HTTPS certificate:

- payment-service provides a certificate for payments.myapp.com
- order-service verifies the cert

- Establishes a secure connection

# 🛠️ Example Setup with HTTPS Certificates in Spring Boot

## 🔧 1. Get an HTTPS Certificate

In real world:

- Use **Let's Encrypt** for free certs
- Or buy from CA like GoDaddy, DigiCert

You'll receive:

- A **certificate** (e.g., cert.pem)
- A **private key** (e.g., privkey.pem)
- A **CA chain** (optional, for trust)

## 🔄 2. Convert to a Keystore for Java

Java/Spring Boot uses .p12 or .jks keystore format.
Convert .pem cert + key to .p12:

```
openssl pkcs12 -export \
  -in cert.pem \
  -inkey privkey.pem \
  -out myapp.p12 \
  -name myapp \
  -CAfile chain.pem \
  -caname root
```

## ⚙️ 3. Configure HTTPS in Spring Boot

In application.properties of payment-service:

```
server.port=8443
server.ssl.enabled=true
server.ssl.key-store=classpath:myapp.p12
server.ssl.key-store-password=changeit
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=myapp
```

✅ Now your service is running with HTTPS!

## 📞 4. Call payment-service from order-service

java
CopyEdit

```
RestTemplate restTemplate = new RestTemplate();
String response = restTemplate.getForObject(
    "https://payments.myapp.com/payments/123",
    String.class
);
```

If the certificate is valid, connection is successful.

## 🔒 Use Case: Why Make Two Services Trust Each Other?

Normally with HTTPS:

- **Client verifies server's identity** via server's certificate.
- But the **server does NOT verify the client.**

With **mutual TLS (mTLS)**:

- Both **client and server authenticate each other** using certificates.
- This is very important in **microservices**, **zero-trust networks**, and **secure cloud communication**.

## 🧩 Scenario

You have:

- OrderService running on orders.myapp.com
- PaymentService running on payments.myapp.com

➡️ You want OrderService to **call** PaymentService securely, and **PaymentService should trust only requests from trusted clients**.

## 🔐 What You'll Need

Each service needs:

- A **private key** and **public certificate**
- A **trust store** that contains the **public certificate of the other service**

Let's call them:

```
order-service:
  keystore: order-keystore.p12 (contains order private key + cert)
  truststore: order-truststore.p12 (contains payment's public cert)

payment-service:
  keystore: payment-keystore.p12 (contains payment private key + cert)
  truststore: payment-truststore.p12 (contains order's public cert)
```

## 🛠️ Step-by-Step: How to Set This Up

## 🔧 1. Generate Keystore and Certificate for Each Service

**For OrderService:**

```
keytool -genkeypair -alias order \
  -keyalg RSA -keysize 2048 \
  -storetype PKCS12 \
  -keystore order-keystore.p12 \
  -storepass changeit \
  -validity 365 \
  -dname "CN=orders.myapp.com"
```

Then export the certificate:

```
keytool -exportcert -alias order \
  -keystore order-keystore.p12 \
  -rfc -file order-cert.pem \
  -storepass changeit
```

**For PaymentService:**

```
keytool -genkeypair -alias payment \
  -keyalg RSA -keysize 2048 \
  -storetype PKCS12 \
  -keystore payment-keystore.p12 \
  -storepass changeit \
  -validity 365 \
  -dname "CN=payments.myapp.com"
```

Then export its certificate:

```
keytool -exportcert -alias payment \
  -keystore payment-keystore.p12 \
  -rfc -file payment-cert.pem \
  -storepass changeit
```

## 🔗 2. Import Each Other's Certificate into Trust Store

**On order-service, trust payment-service:**

```
keytool -importcert \
  -keystore order-truststore.p12 \
  -storepass changeit \
  -alias payment \
  -file payment-cert.pem \
  -storetype PKCS12 \
  -noprompt
```

**On payment-service, trust order-service:**

```
keytool -importcert \
  -keystore payment-truststore.p12 \
  -storepass changeit \
  -alias order \
  -file order-cert.pem \
  -storetype PKCS12 \
  -noprompt
```

## 📁 3. Configure Each Spring Boot App

**application.properties for order-service:**

```
server.port=8443
server.ssl.key-store=classpath:order-keystore.p12
server.ssl.key-store-password=changeit
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=order

server.ssl.trust-store=classpath:order-truststore.p12
```

server.ssl.trust-store-password=changeit
server.ssl.trust-store-type=PKCS12

server.ssl.client-auth=need

**application.properties for payment-service:**
server.port=8443
server.ssl.key-store=classpath:payment-keystore.p12
server.ssl.key-store-password=changeit
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=payment

server.ssl.trust-store=classpath:payment-truststore.p12
server.ssl.trust-store-password=changeit
server.ssl.trust-store-type=PKCS12

server.ssl.client-auth=need

## 🔁 4. Call PaymentService from OrderService Using RestTemplate

## You must configure RestTemplate to use client certificate

**Example Java config for order-service:**

```
@Bean
public RestTemplate restTemplate() throws Exception {
    char[] password = "changeit".toCharArray();

    KeyStore keyStore = KeyStore.getInstance("PKCS12");
    keyStore.load(new FileInputStream("order-keystore.p12"), password);

    KeyStore trustStore = KeyStore.getInstance("PKCS12");
    trustStore.load(new FileInputStream("order-truststore.p12"), password);

    SSLContext sslContext = SSLContexts.custom()
            .loadKeyMaterial(keyStore, password)
            .loadTrustMaterial(trustStore, null)
            .build();

    HttpClient client = HttpClients.custom()
            .setSSLContext(sslContext)
            .build();

    return new RestTemplate(new HttpComponentsClientHttpRequestFactory(client));
}
```

Now your RestTemplate is using:
- **Client certificate (order's)**
- **Trusting only payment-service's certificate**

## 🔄 What Happens When They Communicate?

1. order-service sends HTTPS request → payment-service
2. payment-service asks for **client cert**
3. order-service presents its cert
4. payment-service checks trust store — if it finds order's cert → accepts

5. Then order-service validates payment’s cert from its trust store
6. If mutual trust exists → request is processed
7. Else → connection is rejected (SSLHandshakeException)

# ✅ Benefits of mTLS

| Security Feature | Purpose |
|---|---|
| 🔒 Encryption | Prevent eavesdropping on communication |
| 🔐 Client Authentication | Ensures only trusted services talk to each other |
| 🔁 Integrity | Prevents tampering of data in transit |

# 🧪 Optional Tools for Testing

- 🧪 openssl s_client -connect host:port — to test handshake
- 🔧 Postman can be configured to send client certificates
- 🔍 Wireshark (for inspecting mTLS if needed)
- 🌐 In Kubernetes, you can use Istio or Linkerd to enforce mTLS between pods

# ✅ Summary

Two Spring Boot services trust each other using **mutual TLS** by **exchanging certificates** and **setting up keystore/truststore** in their HTTPS configurations.