**📌 Real-Time Scenario: Deploying a Java-based E-commerce App on GCP**

**📝 Problem Statement**

You are building a **Java-based e-commerce application** that allows users to browse products, place orders, and process payments. The application consists of:  
1️.**Frontend & Backend (Java Spring Boot) → Compute Engine**  
2️.**Database (PostgreSQL) → Cloud SQL**

Your key requirements:  
✅ Users should access the application using a **domain name (DNS)**.  
✅ The **backend should be secure** and accessible only via a **Load Balancer**.  
✅ The **database should not be exposed to the internet**, only accessible from the backend.  
✅ Secure **internet access** for backend VMs to connect to external services (e.g., payment gateways).

**🌐 Solution: Implementing Cloud Services for a Secure and Scalable Java App**

We'll use **Compute Engine, Cloud SQL, VPC, Load Balancers, Cloud NAT, DNS, and Firewall Rules** to design a **secure, scalable cloud architecture**.

**Step 1️: Setting Up the VPC (Virtual Private Cloud)**

🔹 Create a **VPC (ecommerce-vpc)** to manage network isolation.  
🔹 Inside this VPC, create **two subnets**:

A screenshot of a computer

AI-generated content may be incorrect.

**Step 2️⃣: Hosting the Java Application on Compute Engine**

🔹 Deploy the **Java Spring Boot application** on **Compute Engine** inside app-subnet.  
🔹 Use **Instance Groups** (if needed) to scale backend instances dynamically.  
🔹 The Compute Engine VMs should be behind an **HTTPS Load Balancer**.  
🔹 The Load Balancer will:

* Handle incoming traffic from users.
* Distribute requests across multiple backend instances.

✅ **Why?**  
✔️ Ensures **scalability** and **high availability**.  
✔️ Allows secure access via a **domain name** instead of a public VM IP.

**Step 3️⃣: Storing Data in Cloud SQL (PostgreSQL)**

🔹 The application stores user data, products, and orders in **Cloud SQL (PostgreSQL)**.  
🔹 Cloud SQL is deployed inside db-subnet and is **only accessible from the backend**.  
🔹 **Private IP connectivity** ensures no internet exposure.

✅ **Why?**  
✔️ Fully managed PostgreSQL with **automatic backups, replication, and failover**.  
✔️ Backend VMs can access the database **without public exposure**.

**Step 4️⃣: Managing Internet Traffic with DNS and Load Balancing**

🔹 Users should access the application using www.myecommerce.com.  
🔹 Use **Cloud DNS** to map www.myecommerce.com to the Load Balancer.  
🔹 The **Load Balancer** routes traffic to backend Compute Engine instances.

✅ **Why?**  
✔️ Provides a **user-friendly domain name** instead of an IP.  
✔️ Improves **performance and security** using HTTPS.

**Step 5️⃣: Enabling Secure Internet Access for Backend**

🔹 The backend might need to call external APIs (e.g., payment gateway like Stripe).  
🔹 Since the backend is inside a private subnet, it needs **Cloud NAT** to access the internet.

✅ **Why?**  
✔️ **Backend remains private** but can still call external APIs.  
✔️ Avoids assigning **public IPs** to backend VMs.

**Step 6️⃣: Implementing Firewall Rules for Security**

🔹 Define **firewall rules** to restrict access:

A screenshot of a computer program

AI-generated content may be incorrect.

✅ **Why?**  
✔️ Restricts access to only **necessary services**.  
✔️ Prevents **unauthorized external access** to backend and database.

**🎯 Summary: How Each Component Helps**

| **Component** | **Use Case in Scenario** |
| --- | --- |
| **VPC & Subnets** | Isolates backend and database for security |
| **Compute Engine (Backend API)** | Hosts the Java backend securely behind an ILB |
| **Cloud SQL** | Fully managed PostgreSQL database |
| **Load Balancer** | Handles incoming traffic and distributes it to backend instances |
| **Cloud NAT** | Allows backend to call external APIs securely |
| **Cloud DNS** | Resolves domain name for the application |
| **Firewall Rules** | Restricts access to necessary services only |