1. **Create a VPC by enabling hostname**
2. **Create two subnets by allowing public IP**
3. **Create internet gateway and attach your VPC**
4. **Create route by using 0.0.0.0/0**
5. **Go to Subnet Associate and associate your subnet**
6. **Create a security group with inbound & outbound rules**
7. **Create a database by enabling public IP**
8. **Create a Dockerfile for Your Spring Boot App**

FROM openjdk:17-jdk-slim

WORKDIR /app

COPY target/your-app.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

8. **Build and Push Docker Image to Amazon ECR**

**a. Create an ECR Repository**

aws ecr create-repository --repository-name springboot-app --region ap-south-1

**b. Authenticate Docker with ECR**

aws ecr get-login-password --region ap-south-1 | docker login --username AWS --password-stdin 172236798922.dkr.ecr.ap-south-1.amazonaws.com

**c. Build and Tag the Docker Image**

docker build -t springboot-app .

docker tag springboot-app:latest 172236798922.dkr.ecr.ap-south-1.amazonaws.com/springboot-app:latest

d**. Push the Image to ECR**

docker push 172236798922.dkr.ecr.ap-south-1.amazonaws.com/springboot-app:latest

9. **Set Up Amazon RDS MySQL and create database using EC2 console**

10. **Create ECS cluster, network type faregate only**

**11. create task definition** with port mapping

12. Run Task: Go to Clusters → MyCluster → Tasks → Run new task- Launch type: Fargate

13. now go to Tasks → Your task-> In Network section, note Public IP

Open browser: http://<public-ip>:8080

For loadbalancer, enable load balancing in ECS->cluster->services.

For Eureka client, use Discovery to communicate using host. So for this in code, use **eureka.instance.prefer-ip-address=false** in all microservices except API gateway.

##################### Steps for CI/CD###########################

1. Optional, Create a repo for per microservice.
2. Create a buildspec.yml within microservice
3. This file tells AWS CodeBuild **how to build and push your Docker image to Amazon ECR**:
4. **Push to GitHub**
5. Create ECR Repository if not already created
6. you should already have a task definition
7. Create CodePipeline
8. **Source Stage (GitHub)**

* **Source provider**: GitHub (via GitHub App or OAuth)
* **Connect your GitHub repo**
* Choose your user-service repo
* Select the correct branch (e.g. main)
* This will trigger the pipeline on every push

1. **Build Stage (CodeBuild)**

* Choose **“Other build providers” → AWS CodeBuild**
* Click **“Create a new project”**
* **Environment image**: Managed image
  + OS: Ubuntu
  + Runtime: Standard
* **Enable "Privileged" mode** (✅ required for Docker)
* **Buildspec file**: Use buildspec.yml from your repo
* Save the project

1. **Deploy Stage (ECS)**

* **Provider**: Amazon ECS (Fargate)
* **Cluster**: Select your ECS cluster (e.g. user-service-cluster)
* **Service**: Select your ECS service (e.g. user-service)
* **Image definitions file**: imagedefinitions.json (this is auto-created in build step)

########### Configure AWS S3 in Spring Boot###############

* **Add AWS SDK dependency** to your project
* **Configure AWS credentials and region** (preferably via environment variables or AWS IAM roles; avoid hardcoding in code):
  1. **aws.region=ap-south-1**
  2. **aws.access-key=YOUR\_ACCESS\_KEY**
  3. **aws.secret-key=YOUR\_SECRET\_KEY**
  4. **aws.s3.bucket-name=your-bucket-name**
* **Create an S3 client bean in a Spring configuration class:**
* **Use the S3 client to upload/download files** by calling methods like **putObject**() or **getObject**() on the S3Client.
* **Best practices:**
* Use **IAM roles** with proper permissions instead of static credentials.
* Handle exceptions and edge cases for better reliability.
* Consider using **pre-signed URLs** for secure temporary access to objects.

**What is SSO?**

Single Sign-On (SSO) allows users to log in once and gain access to multiple applications without logging in again. Common SSO providers include Okta, Keycloak, AWS Cognito, and others.

1. Add dependency

* + spring-boot-starter-oauth2-client
  + spring-boot-starter-oauth2-resource-server

2. Configure application.yml/properties (**clientId, secret, scope, redirect-uri, authorization-grant-type, issuer-uri**)

3. **Security Configuration:** You can use default Spring Security behavior, or customize:

4. **How it works:**

* User accesses your app.
* They are redirected to the identity provider (Okta).
* After successful login, user is redirected back with a token.
* Spring Security validates the token, grants access.

|  |  |
| --- | --- |
| **OAuth 2.0:** | Authorization (access control) |
| **OIDC** (on top): | Authentication (who the user is) |
| **SSO** | :Experience (login once, access many) |

When you configure **OAuth 2.0 with Okta**, you're almost always using **OIDC** behind the scenes.  
OIDC adds:

* ID tokens (JWT) for **user identity**
* UserInfo endpoint for profile
* Standard scopes like openid, email, profile

**Scenario**

* **App A** and **App B** are two separate Spring Boot applications
* Both use **OAuth 2.0 with OIDC** and are registered with **Okta** as separate clients
* Okta handles **authentication**, and both apps are configured to **trust Okta**

**When a user logs into App A:**

1. User is redirected to Okta login page
2. After successful login, Okta:
   * Creates a **session cookie** in the browser (sid)
   * Issues an **ID token** and **access token** to App A
3. App A uses the ID token to identify the user
4. Okta session is now active (browser still holds Okta cookie)

**When the same user opens App B (in same browser):**

1. App B redirects the user to Okta for authentication
2. Since the **Okta session is already active** (browser has valid Okta cookie):
   * Okta **does not prompt login again**
   * It immediately issues **new ID/access tokens for App B**
3. App B gets user info → ✅ **SSO achieved**

**Authentication authentication = SecurityContextHolder.getContext().getAuthentication();**

**Jwt jwt = (Jwt) authentication.getPrincipal();**

**Note: get UserId using JWT object: String userId = jwt.getClaimAsString("sub");**

**What is SAML?**

**SAML (Security Assertion Markup Language)** is an XML-based open standard for exchanging authentication and authorization data between parties — specifically, between an Identity Provider (IdP) and a Service Provider (SP).

* **Identity Provider (IdP):** The system that authenticates the user (e.g., Okta, ADFS, PingFederate).
* **Service Provider (SP):** The application that the user wants to access (your Spring Boot app).

Get the **X.509 certificate** from your IdP (e.g., Okta) and place it in src/main/resources/okta.cer

**How does SAML SSO work in a nutshell?**

1. User tries to access your app (SP).
2. SP redirects the user to IdP for authentication.
3. User logs in at the IdP.
4. IdP sends a signed XML SAML assertion (authentication response) back to the SP.
5. SP validates the assertion and grants access.

**Step-by-step for SAML config (using Spring Security SAML 2.0):**

1. Add dependencies (Gradle/Maven)

* spring-boot-starter-security
* spring-security-saml2-service-provider

1. Configure application.properties/yaml

* **entity-id:** The IdP’s unique identifier.
* **web-sso-url:** Where to send authentication requests (IdP SSO endpoint).
* **certificate-location:** IdP’s public certificate to verify signatures.
* **assertion-consumer-service-location:** Your SP endpoint for receiving SAML responses.

1. Security Configuration Class

http.authorizeRequests(authorize -> authorize.**anyRequest**().**authenticated**()).**saml2Login**();

**Key points:**

* You must exchange metadata between your SP (your app) and the IdP.
* Import IdP's certificate to verify SAML assertions.
* Spring Security handles SAML request/response under the hood.
* After login, Spring Security creates an authenticated user principal.

################### LDAP ##################

LDAP (Lightweight Directory Access Protocol) is often used in enterprise applications for **authentication and authorization**, commonly with tools like **Active Directory**.

Step-by-Step LDAP Integration in Spring Boot

1. **Add Dependencies**
   1. spring-boot-starter-security
   2. spring-boot-starter-data-ldap
2. Configure properties
   1. urls: ldap://localhost:8389/
   2. base: dc=springframework,dc=org
   3. username: cn=admin,dc=springframework,dc=org
   4. password: admin
3. **Create LDAP Security Configuration**

http .**authorizeHttpRequests**(authz -authz.anyRequest().**fullyAuthenticated**()).**formLogin** (**Customizer.withDefaults**()).**ldapAuthentication**(auth -> auth.**userDnPatterns**("uid={0},ou=people"). **groupSearchBase**("ou=groups").**contextSource**().url("ldap://localhost:8389/dc= **springframework**, dc=org").**managerDn**("cn=admin,dc=**springframework**,dc=org").**managerPassword**("admin")

**Why Use LDAP *with* OAuth 2.0?**

Because they serve **different roles** in a modern authentication system:

|  |  |  |
| --- | --- | --- |
| **Concern** | **Handled by** | **Purpose** |
| **Authentication Source** | LDAP / Active Directory | Verifies user credentials (username/password) |
| **Token Management + Authorization** | OAuth 2.0 / OIDC | Issues tokens, enables SSO, scopes, etc. |

Note: The **OAuth provider** ( Okta) uses **LDAP/AD** internally to authenticate the user.

App (Spring Boot) → OAuth 2.0 (e.g., Okta)

↓

Uses LDAP behind the scenes

**When you might still want LDAP configured in the app**

* You want **hybrid authentication**:
  + Some internal users authenticate directly against LDAP.
  + External users authenticate via Okta OAuth 2.0.

Install mysql:

sudo yum update -y

sudo yum install mysql -y