🔐 Ngelmak Auth - Vault Integration with Spring Boot

This documentation explains how to integrate **HashiCorp Vault** with a **Spring Boot authentication service**.
The goal is to securely manage:

- A **JWT signing key** (for JJWT).
- **Postgres database credentials** (dynamic secrets).

📦 Prerequisites

- HashiCorp Vault installed and running (e.g., see install for Linux).
- Postgres database accessible.
- Spring Boot project with `spring-cloud-starter-vault-config` dependency.

🏗 Vault Deployment Mode Setup

📂 Recommended Folder Layout

```
/etc/vault/
├── config/         # Vault server configs (vault.hcl, etc.)
├── policies/       # Policy files (.hcl)
├── approles/       # AppRole definitions (optional JSON/HCL)
├── tls/            # Certificates if TLS enabled
└── logs/           # Vault logs
```

PROF

```
# Create the main Vault directory
sudo mkdir -p /etc/vault/data
sudo mkdir -p /etc/vault/config
sudo mkdir -p /etc/vault/policies
sudo mkdir -p /etc/vault/approles
sudo mkdir -p /etc/vault/tls
sudo mkdir -p /etc/vault/logs
```

1. Create a Vault Configuration File (config.hcl)

```
storage "raft" {
  path    = "/etc/vault/data"
  node_id = "node1"
}
```

```
listener "tcp" {
  address     = "0.0.0.0:8200"
  tls_disable = "true"   # disable TLS for testing; enable TLS in
production
}

api_addr = "http://127.0.0.1:8200"
cluster_addr = "http://127.0.0.1:8201"
disable_mlock = true
ui = true   # enables the Vault web UI
```

- **storage "file"** → tells Vault to use the local filesystem (see also **storage "raft"**).
- **listener "tcp"** → network listener; configure TLS in real deployments.
- **api_addr / cluster_addr** → advertise addresses for clients and cluster peers.
- **ui = true** → turns on the built-in web interface.

```
sudo nano /etc/vault/config/config.hcl
```

2. Start Vault in Server Mode

```
sudo vault server -config=/etc/vault/config/config.hcl
```

- **-config** → points to your configuration file. Unlike -dev, this persists secrets in the backend.

2.1. Export Vault Address

To ensure the client can communicate with the Vault server, export the Vault address

PROF

```
export VAULT_ADDR='http://127.0.0.1:8200'
```

3. Initialize Vault

```
vault operator init -key-shares=5 -key-threshold=3
```

- **-key-shares=5** → generate 5 unseal keys.
- **-key-threshold=3** → require 3 of them to unseal Vault.
  This will show:
  - Unseal Keys → must be stored securely (e.g., HSM, secret manager).
  - Initial Root Token → used for initial setup; rotate and revoke after creating policies.

4. Unseal Vault (or use the ui) and save the root token

```
vault operator unseal <unseal-key-1>
vault operator unseal <unseal-key-2>
vault operator unseal <unseal-key-3>

vault status
Key                     Value
---                     -----
Seal Type               shamir
Initialized             true
Sealed                  false
...
```

Provide the threshold number of keys (here 3/5).

5. Secure the Root Token (optional)

Use the root token only to create admin policies and AppRoles.

Then revoke or rotate it.

Applications (like Spring Boot) should authenticate via AppRole, not the root token.

Use the root token to login

You need to connect to vault for the following commands

```
vault login
```

PROF

```
Success! You are now authenticated. The token information displayed
below
is already stored in the token helper. You do NOT need to run "vault
login"
again. Future Vault requests will automatically use this token.

Key                  Value
---                  -----
token                <token>
token_accessor       <token>
token_duration       ∞
token_renewable      false
token_policies       ["root"]
identity_policies    []
policies             ["root"]
```

Enable KV Secrets Engine

The KV engine stores static secrets such as your JWT signing key.

```
vault secrets enable -path=secret kv
```

- **-path=secret** → mount point name; secrets will live under `secret/`.
- **kv** → type of secrets engine (key-value).

Store the JWT secret:

```
vault kv put secret/jjwt jwt-secret-key="super-secret"
```

- **secret/jjwt** → path where the secret is stored.
- **jwt-secret-key="super-secret"** → field name and value stored at that path.

Check for the secret key with `vault kv get secret/jjwt`.

Enable Database Secrets Engine

Vault can generate **dynamic Postgres credentials**.

```
vault secrets enable database
```

- **database** → type of secrets engine for databases.

Configure Postgres connection:

PROF

```
vault write database/config/my-postgres \
  plugin_name=postgresql-database-plugin \
  connection_url="postgresql://{{username}}:{{password}}@db-
host:5432/postgres" \
  username="vaultadmin" \
  password="vaultadminpassword" \
  allowed_roles="springboot-db-role"
```

- **database/config/my-postgres** → name of this DB config.
- **plugin_name** → database driver (for Postgres: `postgresql-database-plugin`).
- **connection_url** → template Vault uses, inserting generated creds into `{{username}}` and `{{password}}`.
- **username/password** → privileged DB account Vault uses to create/revoke users.
- **allowed_roles** → which Vault roles can request creds from this config.

Define a role for dynamic users:

```
vault write database/roles/springboot-db-role \
  db_name=my-postgres \
  creation_statements="CREATE ROLE \"{{name}}\" WITH LOGIN PASSWORD
'{{password}}' VALID UNTIL '{{expiration}}';" \
  default_ttl="1h" \
  max_ttl="24h"
```

- **database/roles/springboot-db-role** → Vault role name.
- **db_name** → links to the DB config (`my-postgres`).
- **creation_statements** → SQL template for creating users; Vault injects `{{name}}`, `{{password}}`, `{{expiration}}`.
- **default_ttl** → default credential lifetime.
- **max_ttl** → maximum lifetime of credentials.

---

3. Define a Policy

Policies restrict what your app can access.

Create `springboot-policy.hcl`:

```
# Allow reading JWT secret
path "secret/data/jjwt" {
  capabilities = ["read"]
}

# Allow reading dynamic Postgres creds
path "database/creds/springboot-db-role" {
  capabilities = ["read"]
}
```

PROF

- **path** → Vault API path to control (KV v2 reads use `secret/data/<name>`).
- **capabilities** → allowed actions (e.g., `read`, `create`, `update`, `delete`, `list`).

Load the policy:

```
vault policy write springboot-policy springboot-policy.hcl
```

- **springboot-policy** → name of the policy.
- **springboot-policy.hcl** → file containing the rules.

---

4. Enable AppRole Authentication

AppRole is the recommended auth method for applications.

`vault auth enable approle`

- **approle** → type of auth method.

Create an AppRole:

```
vault write auth/approle/role/springboot \
  policies="springboot-policy" \
  secret_id_ttl=24h \
  token_ttl=1h \
  token_max_ttl=4h
```

- **policies** → attach the policy you created (defines what the app can access).
- **secret_id_ttl** → how long the Secret ID remains valid before rotation.
- **token_ttl** → default lifetime of the Vault token issued to the app.
- **token_max_ttl** → maximum lifetime; tokens cannot be renewed beyond this limit.

Fetch Role ID:

```
vault read auth/approle/role/springboot/role-id
```

- **auth/approle/role/springboot/role-id** → path that returns the Role ID (non-secret identifier).

Generate Secret ID:

```
vault write -f auth/approle/role/springboot/secret-id
```

PROF

- **-f** → force create; no payload needed.
- **auth/approle/role/springboot/secret-id** → path that issues a Secret ID (secret half of AppRole).

---

5. Spring Boot Configuration

Add dependency in `pom.xml`:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-vault-config</artifactId>
  <version>2.1.3.RELEASE</version>
</dependency>
```

- **spring-cloud-starter-vault-config** → integrates Spring Boot with Vault, exposing secrets as properties.

Configure `application.yml`:

```
spring:
  application:
    name: auth-service

  datasource:
    url: jdbc:postgresql://db-host:5432/postgres
    driver-class-name: org.postgresql.Driver
    username: ${spring.cloud.vault.database.username}
    password: ${spring.cloud.vault.database.password}

  cloud:
    vault:
      uri: http://localhost:8200
      authentication: approle
      app-role:
        role-id: ${VAULT_ROLE_ID}
        secret-id: ${VAULT_SECRET_ID}
      kv:
        enabled: true
        backend: secret
        default-context: jjwt
      database:
        enabled: true
        role: springboot-db-role
```

- **uri** → Vault server address.
- **authentication** → auth method (`approle`).
- **app-role.role-id / secret-id** → values retrieved from Vault (supply via env vars).
- **kv.enabled** → enables KV property source.
- **kv.backend** → KV mount path (`secret`).
- **kv.default-context** → subpath for the secret (`jjwt` → resolves to `secret/data/jjwt`).
- The KV entry `jwt-secret-key="super-secret"` becomes a Spring property named `jwt-secret-key`, which can be injected with:

```
@Value("${jwt-secret-key}")
private String jwtSecret;
```

- **spring.cloud.vault.database.enabled: true** → tells Spring Cloud Vault to fetch dynamic DB creds.
- **spring.cloud.vault.database.role: springboot-db-role** → matches the Vault role you defined.
- Spring Cloud Vault injects those values into `spring.cloud.vault.database.username/password`, then DataSource uses Vault-issued

PROF

credentials to authenticate.

🗝 At runtime

1. Spring Boot starts.
2. Spring Cloud Vault authenticates to Vault (AppRole).
3. Vault issues a token.
4. Spring Cloud Vault fetches Database creds → exposed as `spring.cloud.vault.database.username/password`.
5. Spring Boot's DataSource picks up those properties and connects to Postgres.

✅ Summary

- **KV engine** → stores JWT secret.
- **Database engine** → generates dynamic Postgres credentials.
- **Policy** → restricts access to only required paths.
- **AppRole** → authenticates Spring Boot app.
- **Spring Boot config** → fetches secrets at runtime, no hardcoding.