# Spring Vault

## What is Vault?

- Vault is a software of the company HashiCorp
- Vault is an identity-based secret and encryption management system
  - A secret is anything that we don’t want to let known to others; for example systems credentials, API encryption keys, certificates, and many other
- Vault provides a secure and convenient way to manage secrets

Further info about Vault on https://www.vaultproject.io/

## Why Vault?

- There are several best practices to use in keeping secrets, Vault does this for us by providing a simple CLI, API, or a simple Web UI

- It can be integrated with a lot of systems like DB, Github, Kubernetes, AWS and many others

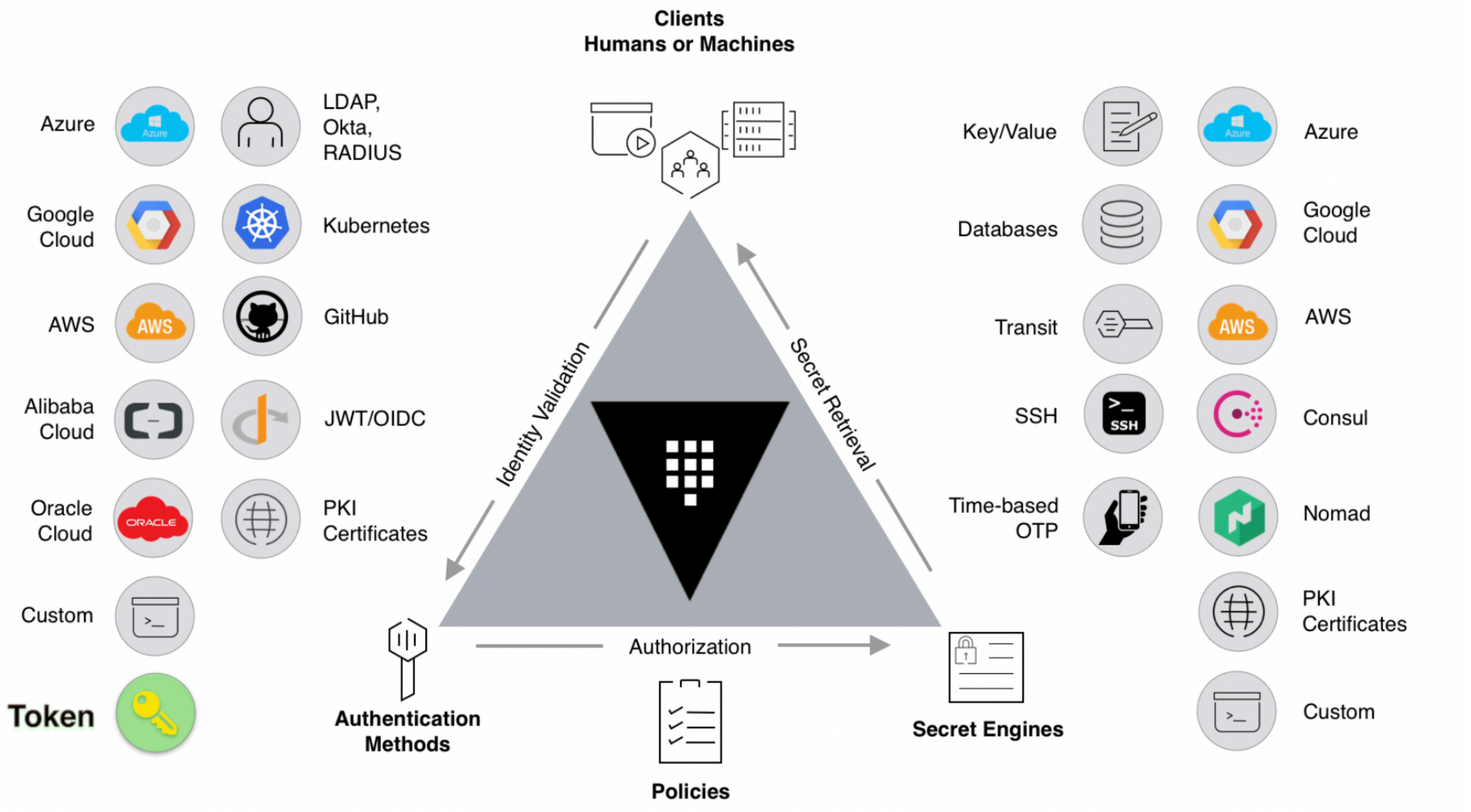

Image from learn.hashicorp.com/tutorials/vault/getting-started-intro

## Exploring Vault

- The first step is to install and use Vault from the command line, so is possible to explore some features that can easily be re-used in Spring applications

- Vault can be downloaded following the instructions reported in the official documentation:

https://learn.hashicorp.com/tutorials/vault/getting-started-install

## Secrets engines

- Secrets engines are components that store, generate, or encrypt data
- Some secrets engines simply store and read data, other secrets engines connect to other services and generate dynamic credentials on demand
- Secrets engines are enabled at a "path" in Vault
- When a request comes to Vault, the router automatically routes anything with the route prefix to the secrets engine
- Create, set up and start the development server

# Secrets engines

```
$ vault server -dev

   WARNING! dev mode is enabled! In this mode, Vault runs entirely in-memory
   and starts unsealed with a single unseal key. The root token is already
   authenticated to the CLI, so you can immediately begin using Vault.

   You may need to set the following environment variable:

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

   The unseal key and root token are displayed below in case you want to
   seal/unseal the Vault or re-authenticate.

   Unseal Key: 1+yv+v5mz+aSCK67X6slL3ECxb4UDL8ujWZU/ONBpn0=
   Root Token: s.XmpNPoi9sRhYtdKHaQhkHP6x ] ─────────────────────────────────────▶ Auth token

   Development mode should NOT be used in production installations!

   ==> Vault server started! Log data will stream in below:
```

Open a new terminal..

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

$ export VAULT_TOKEN="s.XmpNPoi9sRhYtdKHaQhkHP6x"
```

## Key/Value secrets engine

- By default, running the dev server, the key/value secrets engine is enabled at ‘`secret/`’ path

- This secrets engine provides a safe environment where is possible to save a set of key:value couples, obviously these secrets are encrypted

- On the right, there is an example of how is possible to store and retrieve these types of secrets

```
$ vault kv put secret/hello foo=world excited=yes

  Key                  Value
  ---                  -----
  created_time         2022-01-15T01:40:09.888293Z
  custom_metadata      <nil>
  deletion_time        n/a
  destroyed            false
  version              2


$ vault kv get secret/hello

  == Secret Path ==
  secret/data/hello

  ======= Metadata =======
  Key                  Value
  ---                  -----
  created_time         2022-03-30T11:14:30.632633Z
  custom_metadata      <nil>
  deletion_time        n/a
  destroyed            false
  version              1

  ===== Data =====
  Key          Value
  ---          -----
  excited      yes
  foo          world
```

## Enable a secrets engine

- A new secrets engine can
  be enabled with the `enable` command specifying path and secrets engine type

- All enabled secret engines and their relative path can be listed with `list` command

- Is possible to insert and read value as already seen before, and finally, disable the secrets engine

## Enable a secrets engine

```
$ vault secrets enable -path=polito/ kv


   Success! Enabled the kv secrets engine at: polito/


$ vault secrets list

   Path          Type         Accessor              Description
   ----          ----         --------              -----------
   cubbyhole/    cubbyhole    cubbyhole_4982ced5    per-token private secret storage
   database/     database     database_4dd7d497     n/a
   identity/     identity     identity_224c4ac0     identity store
   polito/       kv           kv_755c1aec           n/a
   secret/       kv           kv_be8bc258           key/value secret storage
   sys/          system       system_b259db0d       system endpoints used for…


$ vault kv put polito/credentials username=politouser password=s3cr37

   Success! Data written to: polito/credentials

$ vault kv get polito/credentials

   ====== Data ======
   Key         Value
   ---         -----
   password    s3cr37
   username    politouser

$ vault secrets disable polito/

   Success! Disabled the secrets engine (if it existed) at: polito/
```

## Enable new secrets engine

- The read/write/delete/list operations are forwarded to the corresponding secrets engine, and the secrets engine decides how to react to those operations

- It enables Vault to interface directly with physical systems, databases, HSMs, etc. But in addition to these physical systems, Vault can interact with more unique environments like AWS IAM, dynamic SQL user creation, GitHub auth, etc. all while using the same read/write interface

## From CLI to Spring

- Vault project was been introduced and was learned how to interact with it
- Spring provides strong integration with Vault that, makes it simple to manage secrets
- All the learned concept until now is valid for this purpose because Spring Vault library provides some interfaces to perform the same operations made by CLI
- Spring Vault can be used without bringing up all SpringApplicationContext, but using it provides all Spring’s IoC benefits

## From CLI to Spring

- Add the following Gradle dependency

```
implementation("org.springframework.cloud:spring-cloud-
starter-vault-config")
```

## First steps

- Start the web server in dev mode through the CLI

- The simplest way to use Spring Vault is without the `SpringApplicationContext`
  - Instantiating a `VaultTemplate` object which takes a `VaultEndpoint` and a `TokenAuthentication`
  - As token provides the token printed out when the server was started

- The `VaultTemplate` object provides a simple interface for using Vault functionality

```
data class Secrets(val username: String,
                   val password: String)

fun main(args: Array<String>) {
  val endpoint = VaultEndpoint()
  endpoint.schema = “http"

  val vaultTemplate = VaultTemplate(
    VaultEndpoint(),
    TokenAuthentication(“<your_token>”))

  val secrets = Secrets("hello", "world")

  vaultTemplate.write("secret/myapp", secrets)

  val response = vaultTemplate
    .read("secret/myapp", Secrets::class.java)

  if(Objects.nonNull(response?.data)){

    println(response?.data)
    vaultTemplate.delete("secret/myapp")

  } else {

    println("No secrets found")
  }
}
```

- `VaultEndpoint` represents the vault server, by default, it is configured on `https://localhost:8200/` as the vault development server default configuration
- `TokenAuthentication` represents the authentication strategy chosen, the vault development server by default provides a token auth strategy, the token used for the authentication is provided on the server startup
- `VaultOperations` is another interface that provides access to Vault features which is also implemented by the `VaultTemplate` class
- Is very common and useful to use Spring Vault inside the Spring Context
- A Vault configuration class can be instantiated so that the dependency injection of Spring Boot can provide the configured Vault implementation

- A common way to specify project properties is by spring configuration files, an example of `application.yml`:

```yaml
config:
  vault:
    host: "localhost"
    port: "8200"
    token: "hvs.EqzW9jLKDpjCMm0aJn992nWU"
```

- These values are read by the java configuration class

- The VaultConfig class must extend the `AbstractVaultConfiguration` which is in charge to provide the config into the `ApplicationContext`
- Overriding the `vaultEndpoint` method is possible to provide a custom `VaultEndpoint` implementation
- The same is for `clientAuthentication` method, it returns a `ClientAuthentication` of which `TokenAuthentication` is an implementation

```kotlin
@Configuration
class VaultConfig: AbstractVaultConfiguration() {


    @Value("\${config.vault.host}")
    var vaultHost: String = "localhost"


    @Value("\${config.vault.port}")
    var vaultPort: Int = 8200


    @Value("\${config.vault.token}")
    var token: String = ""

    /**
     * Return us implementation of VaultEndpoint.
     */
    override fun vaultEndpoint(): VaultEndpoint {
        val endpoint = VaultEndpoint.create(vaultHost,
vaultPort)
        endpoint.scheme = "http"
        return endpoint
    }

    /**
     * Configure a Token authentication.
     */
    override fun clientAuthentication(): ClientAuthentication {
        return TokenAuthentication(token)
    }

}
```

- With `VaultConfig` class the same `VaultTemplate` (or `VaultOperations`) instance is provided through dependency injection

- Thus, the service class can use the `VaultTemplate` instance to perform some tasks like write, read and revoke secrets

```kotlin
@Service
class VaultService(val vaultTemplate: VaultTemplate) {

    fun writeSecret(name: String, secret: SecretDTO) : Boolean {
        val path = "polito/$name"
        try {
            vaultTemplate.write(path, secret)

        } catch (e: Exception) {
            logger.error("Some error occurred writing a secret")
            return false
        }

        return true
    }

    fun readSecret(name: String) : SecretDTO? {
        val path = "polito/$name"
        try {
            val response = vaultTemplate
                    .read(path, SecretDTO::class.java)?.data

             //Delete secret
             vaultTemplate.delete(path);

        } catch (e: Exception) {
            logger.error("Some error occurred writing a secret")
            return null
        }

        return response
    }
}
```

# Vault Operations

- VaultOperations is another interface that can be used to access Vault features
- It provides several factory methods, one for every type of secrets engine which returns an object that can be used to access the functionality
- In the current example, a KeyValue engine is used

```
fun writeSecret(name: String, secret: SecretDTO) : BaseResponse {
    val path = "polito/$name"
    val keyValueOperations = vaultOperations.opsForKeyValue(path,
VaultKeyValueOperationsSupport.KeyValueBackend.KV_1)
    try {
        logger.info("Attempting to write secret in vault...")
        keyValueOperations.put(path, secret)
        logger.info("Secret written")
    } catch (e: Exception) {
        logger.error("Some error occurred writing a secret")
        return BaseResponse(true, "Error, secret not created")
    }

    return BaseResponse(false, "Secret created")
}

fun readSecret(pathName: String): BaseResponse {
    val path = "polito/$pathName"
    val keyValueOperations = vaultOperations.opsForKeyValue(path,
 VaultKeyValueOperationsSupport.KeyValueBackend.KV_1)

    try {
        logger.info("Attempting to read secret from vault...")
        val secret = keyValueOperations.get(path, SecretDTO::clas
        if(Objects.isNull(secret)){
            throw NullPointerException()
        }
        logger.info("Secret read")
        return BaseResponse(false, secret)
    } catch (e: NullPointerException) {
        logger.error("Secret not found")
    } catch (e: Exception) {
        logger.error("Some error occurred reading a secret")
    }
    return BaseResponse(true, "No secret found in $path")
}
```

## KeyValue versions

- KeyValue secrets engine is available in two versions, version 1  and version 2
- The difference is that the version 2 secrets are versioned, conversely, for the version 1 secrets only the most recent written value for a key is preserved
- It means that with version 2 KeyValue secrets, is possible to browse between the versions of a secret

## PKI Engine

- Vault provides another secret engine called PKI
- It represents a backend for certificates by implementing certificate authority operations
- The PKI secrets engine generates dynamic X.509 certificates
- With this secrets engine, services can get certificates without going through the usual manual process of generating a private key and CSR, submitting it to a CA, and waiting for a verification and signing process to complete

## Token Engine

- This backend is an authentication backend that does not interact with actual secrets
- It gives access to access token management
- The `token` authentication method is built-in and automatically available at `/auth/token`
    - It lets users authenticate using a token, as well to create new tokens, revoke secrets by token, and more

- A token can be created providing some criteria
- To submit token request with criteria a `VaultTokenRequest` is required
- `.create()` method return a `VaultTokenResponse` which contains the result token
- `.renew()` and `.revoke()` method can be used respectively to renew and revoke a token

```
val operations: VaultOperations =
                VaultTemplate(VaultEndpoint())
val tokenOperations = operations.opsForToken()

val tokenResponse = tokenOperations.create()
val justAToken = tokenResponse.token

val tokenRequest = VaultTokenRequest.builder()
    .withPolicy("policy-for-myapp")
    .displayName("Access tokens for myapp")
    .renewable()
    .ttl(Duration.ofHours(1))
    .build()

val appTokenResponse = tokenOperations
                .create(tokenRequest)
val appToken = appTokenResponse.token

tokenOperations.renew(appToken)

tokenOperations.revoke(appToken)
```

## Session management

- Spring Vault, as seen, requires a `ClientAuthentication` to login and access Vault
- Vault login should not occur on each authenticated Vault interaction but must be reused throughout a session
- This aspect is handled by a `SessionManager` implementation
- A `SessionManager` decides how often it obtains a token, about revocation and renewal

## Session management

- Spring comes up with two implementations of `SessionManager`
    - `SimpleSessionManager` just obtains tokens from the supplied `ClientAuthentication` without refresh and revocation
    - `LifecycleAwareSessionManager` schedules a token renewal if the token is renewable, and revokes a login token on disposal

- `LifecycleAwareSessionManager` is configured by default if using `AbstractVaultConfiguration`

## Further sources

- The complete Spring Vault documentation is available on Spring Website:

  https://docs.spring.io/spring-vault/docs/current/reference/html/

- Some code examples are available on GitHub:

  https://github.com/giovannimirarchi420/vault