Spring Vault - Reference Documentation

Mark Paluch

Version 2.3.1, 2021-02-16

© 2016-2021 The original authors.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Preface

The Spring Vault project applies core Spring concepts to the development of solutions using HashiCorp Vault. We provide a "template" as a high-level abstraction for storing and querying documents. You will notice similarities to the REST support in the Spring Framework.

This document is the reference guide for Spring Vault. It explains Vault concepts and semantics and the syntax.

This part of the reference documentation explains the core functionality offered by Spring Vault.

Vault support introduces the Vault module feature set.

Chapter 1. Document Structure

This section provides basic introduction to Spring and Vault. It contains details about following development and how to get support.

The rest of the document refers to Spring Vault features and assumes the user is familiar with HashiCorp Vault as well as Spring concepts.

Chapter 2. Knowing Spring

Spring Vault uses Spring framework's core functionality, such as IoC container. While it is not important to know the Spring APIs, understanding the concepts behind them is. At a minimum, the idea behind IoC should be familiar for whatever IoC container you choose to use.

The core functionality of the Vault support can be used directly, with no need to invoke the IoC services of the Spring Container. This is much like `RestTemplate` which can be used 'standalone' without any other services of the Spring container. To leverage all the features of Spring Vault document, such as the session support, you will need to configure some parts of the library using Spring.

To learn more about Spring, you can refer to the comprehensive (and sometimes disarming) documentation that explains in detail the Spring Framework. There are a lot of articles, blog entries and books on the matter - take a look at the Spring framework home page for more information.

Chapter 3. Knowing Vault

Security and working with secrets is a concern of every developer working with databases, user credentials or API keys. Vault steps in by providing a secure storage combined with access control, revocation, key rolling and auditing. In short: Vault is a service for securely accessing and storing secrets. A secret is anything that you want to tightly control access to, such as API keys, passwords, certificates, and more.

The jumping off ground for learning about Vault is www.vaultproject.io. Here is a list of useful resources:

- The manual introduces Vault and contains links to getting started guides, reference documentation and tutorials.
- The online shell provides a convenient way to interact with a Vault instance in combination with the online tutorial.
- HashiCorp Vault Introduction
- HashiCorp Vault Documentation

Spring Vault provides client-side support for accessing, storing and revoking secrets. With HashiCorp's Vault you have a central place to manage external secret data for applications across all environments. Vault can manage static and dynamic secrets such as application data, username/password for remote applications/resources and provide credentials for external services such as MySQL, PostgreSQL, Apache Cassandra, Consul, AWS and more.

Chapter 4. Requirements

Spring Vault 2.x binaries requires JDK level 8.0 and above, and Spring Framework 5.3.4 and above.

In terms of Vault, Vault at least 0.6.

Chapter 5. Additional Help Resources

Learning a new framework is not always straight forward. In this section, we try to provide what we think is an easy to follow guide for starting with Spring Vault module. However, if you encounter issues or you are just looking for advice, feel free to use one of the links below:

5.1. Support

There are a few support options available:

5.1.1. Community Forum

Post questions questions regarding Spring Vault on Stackoverflow to share information and help each other. Note that registration is needed **only** for posting.

5.1.2. Professional Support

Professional, from-the-source support, with guaranteed response time, is available from Pivotal Sofware, Inc., the company behind Spring Vault and Spring.

5.2. Following Development

For information on the Spring Vault source code repository, nightly builds and snapshot artifacts please see the Spring Vault homepage. You can help make Spring Vault best serve the needs of the Spring community by interacting with developers through the Community on Stackoverflow. If you encounter a bug or want to suggest an improvement, please create a ticket on the Spring Vault issue tracker. To stay up to date with the latest news and announcements in the Spring ecosystem, subscribe to the Spring Community Portal. Lastly, you can follow the Spring blog or the project team on Twitter (SpringCentral).

Chapter 6. New & Noteworthy

6.1. What's new in Spring Vault 2.3

- Support for PEM-encoded certificates for keystore and truststore usage.
- `ReactiveVaultEndpointProvider` for non-blocking lookup of `VaultEndpoint`.
- `VaultKeyValueMetadataOperations` for Key-Value metadata interaction.
- Support for `transform` backend (Enterprise Feature).
- Documentation of how to use Vault secret backends.
- Login credentials for Kubernetes and PCF authentication are reloaded for each login attempt.
- `SecretLeaseContainer` publishes `SecretLeaseRotatedEvent` instead of `SecretLeaseExpiredEvent` and `SecretLeaseCreatedEvent` on successful secret rotation.
- `AbstractVaultConfiguration.threadPoolTaskScheduler()` bean type changed to `TaskSchedulerWrapper` instead of `ThreadPoolTaskScheduler`.

6.2. What's new in Spring Vault 2.2

- Support for Key-Value v2 (versioned backend) secrets through `@VaultPropertySource`.
- SpEL support in `@Secret`.
- Add support for Jetty as reactive HttpClient.
- `LifecycleAwareSessionManager` and `ReactiveLifecycleAwareSessionManager` emit now `AuthenticationEvent`s.
- PCF authentication.
- Deprecation of `AppIdAuthentication`. Use `AppRoleAuthentication` instead as recommended by HashiCorp Vault.
- `CubbyholeAuthentication` and wrapped `AppRoleAuthentication` now use `sys/wrapping/unwrap` endpoints by default.
- Kotlin Coroutines support for `ReactiveVaultOperations`.

6.3. What's new in Spring Vault 2.1

- GCP Compute, GCP IAM, and Azure authentication.
- Template API support for versioned and unversioned Key/Value backends and for Vault wrapping operations.
- Support full pull mode in reactive AppRole authentication.
- Improved Exception hierarchy for Vault login failures.

6.4. What's new in Spring Vault 2.0

- Authentication steps DSL to compose authentication flows.
- Reactive Vault client via `ReactiveVaultOperations`.
- Vault repository support based on Spring Data KeyValue.
- Transit batch encrypt and decrypt support.
- Policy management for policies stored as JSON.
- Support CSR signing, certificate revocation and CRL retrieval.
- Kubernetes authentication.
- RoleId/SecretId unwrapping for AppRole authentication.
- Spring Security integration with transit backend-based `BytesKeyGenerator` and `BytesEncryptor`.

6.5. What's new in Spring Vault 1.1.0

- AWS IAM authentication.
- Configuration of encryption/decryption versions for transit keys.
- Pull mode for AppRole authentication.
- Transit batch encrypt and decrypt support.
- TTL-based generic secret rotation.

6.6. What's new in Spring Vault 1.0

- Initial Vault support.

Reference documentation

Chapter 7. Vault support

The Vault support contains a wide range of features which are summarized below.

- Spring configuration support using Java based @Configuration classes
- `VaultTemplate` helper class that increases productivity performing common Vault operations. Includes integrated object mapping between Vault responses and POJOs.

For most tasks, you will find yourself using `VaultTemplate` that leverages the rich communication functionality. `VaultTemplate` is the place to look for accessing functionality such as reading data from Vault or issuing administrative commands. `VaultTemplate` also provides callback methods so that it is easy for you to get a hold of the low-level API artifacts such as `RestTemplate` to communicate directly with Vault.

7.1. Dependencies

The easiest way to find compatible versions of Spring Vault dependencies is by relying on the Spring Vault BOM we ship with the compatible versions defined. In a Maven project you would declare this dependency in the `<dependencyManagement />` section of your `pom.xml`:

Example 1. Using the Spring Vault BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.vault</groupId>
      <artifactId>spring-vault-dependencies</artifactId>
      <version>2.3.1</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current version is `2.3.1`. The version name follows the following patterns: `${version}` for GA and service releases and `${version}-${release}` for snapshots and milestones. `release` can be one of the following:

- `SNAPSHOT` - current snapshots
- `M1`, `M2` etc. - milestones
- `RC1`, `RC2` etc. - release candidates

Example 2. Declaring a dependency to Spring Vault

```
<dependencies>
    <dependency>
        <groupId>org.springframework.vault</groupId>
        <artifactId>spring-vault-core</artifactId>
    </dependency>
</dependencies>
```

7.2. Spring Framework

The current version of Spring Vault requires Spring Framework in version 5.3.4 or better. The modules might also work with an older bugfix version of that minor version. However, using the most recent version within that generation is highly recommended.

Chapter 8. Getting Started

Spring Vault support requires Vault 0.6 or higher and Java SE 6 or higher. An easy way to bootstrap setting up a working environment is to create a Spring based project in STS.

First you need to set up a running Vault server. Refer to the Vault for an explanation on how to startup a Vault instance.

To create a Spring project in STS go to File → New → Spring Template Project → Simple Spring Utility Project → press Yes when prompted. Then enter a project and a package name such as `org.spring.vault.example`.

Then add the following to `pom.xml` dependencies section.

Example 3. Adding Spring Vault dependency

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.vault</groupId>
    <artifactId>spring-vault-core</artifactId>
    <version>2.3.1</version>
  </dependency>

</dependencies>
```

If you are using a milestone or release candidate, you will also need to add the location of the Spring Milestone repository to your maven `pom.xml` which is at the same level of your `<dependencies/>` element.

```
<repositories>
  <repository>
    <id>spring-milestone</id>
    <name>Spring Maven MILESTONE Repository</name>
    <url>https://repo.spring.io/libs-milestone</url>
  </repository>
</repositories>
```

The repository is also browseable here.

If you are using a SNAPSHOT, you will also need to add the location of the Spring Snapshot repository to your maven `pom.xml` which is at the same level of your `<dependencies/>` element.

```
<repositories>
  <repository>
    <id>spring-snapshot</id>
    <name>Spring Maven SNAPSHOT Repository</name>
    <url>https://repo.spring.io/libs-snapshot</url>
  </repository>
</repositories>
```

The repository is also browseable here.

Create a simple `Secrets` class to persist:

Example 4. Mapped data object

```
package org.spring.vault.example;

public class Secrets {

    String username;
    String password;

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }
}
```

And a main application to run

Example 5. Example application using Spring Vault

```
package org.springframework.vault.example;

import org.springframework.vault.authentication.TokenAuthentication;
import org.springframework.vault.client.VaultEndpoint;
import org.springframework.vault.core.VaultTemplate;
import org.springframework.vault.support.VaultResponseSupport;

public class VaultApp {

    public static void main(String[] args) {

        VaultTemplate vaultTemplate = new VaultTemplate(new VaultEndpoint(),
                new TokenAuthentication("00000000-0000-0000-0000-000000000000"));

        Secrets secrets = new Secrets();
        secrets.username = "hello";
        secrets.password = "world";

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

        VaultResponseSupport<Secrets> response = vaultTemplate.read("secret/myapp
", Secrets.class);
        System.out.println(response.getData().getUsername());

        vaultTemplate.delete("secret/myapp");
    }
}
```

Even in this simple example, there are few things to take notice of

- You can instantiate the central class of Spring Vault, `VaultTemplate`, using the `org.springframework.vault.client.VaultEndpoint` object and the `ClientAuthentication`. You are not required to spin up a Spring Context to use Spring Vault.
- Vault is expected to be configured with a root token of `00000000-0000-0000-0000-000000000000` to run this application.
- The mapper works against standard POJO objects without the need for any additional metadata (though you can optionally provide that information).
- Mapping conventions can use field access. Notice the `Secrets` class has only getters.
- If the constructor argument names match the field names of the stored document, they will be used to instantiate the object.

Chapter 9. Introduction to VaultTemplate

The class `VaultTemplate`, located in the package `org.springframework.vault.core`, is the central class of the Spring's Vault support providing a rich feature set to interact with Vault. The template offers convenience operations to read, write and delete data in Vault and provides a mapping between your domain objects and Vault data.

> Once configured, `VaultTemplate` is thread-safe and can be reused across multiple instances.

The mapping between Vault documents and domain classes is done by delegating to `RestTemplate`. Spring Web support provides the mapping infrastructure.

The `VaultTemplate` class implements the interface `VaultOperations`. In as much as possible, the methods on `VaultOperations` are named after methods available on the Vault API to make the API familiar to existing Vault developers who are used to the API and CLI. For example, you will find methods such as "write", "delete", "read", and "revoke". The design goal was to make it as easy as possible to transition between the use of the Vault API and `VaultOperations`. A major difference in between the two APIs is that `VaultOperations` can be passed domain objects instead of JSON Key-Value pairs.

> The preferred way to reference the operations on `VaultTemplate` instance is via its interface `VaultOperations`.

While there are many convenience methods on `VaultTemplate` to help you easily perform common tasks if you should need to access the Vault API directly to access functionality not explicitly exposed by the `VaultTemplate` you can use one of several execute callback methods to access underlying APIs. The execute callbacks will give you a reference to a `RestOperations` object. Please see the section Execution Callbacks for more information.

Now let's look at a examples of how to work with Vault in the context of the Spring container.

9.1. Registering and configuring Spring Vault beans

Using Spring Vault does not require a Spring Context. However, instances of `VaultTemplate` and `SessionManager` registered inside a managed context will participate in lifecycle events provided by the Spring IoC container. This is useful to dispose active Vault sessions upon application shutdown. You also benefit from reusing the same `VaultTemplate` instance across your application.

Spring Vault comes with a supporting configuration class that provides bean definitions for use inside a Spring context. Application configuration classes typically extend from `AbstractVaultConfiguration` and are required to provide additional details that are environment specific.

Extending from `AbstractVaultConfiguration` requires to implement ` VaultEndpoint vaultEndpoint()` and `ClientAuthentication clientAuthentication()` methods.

Example 6. Registering Spring Vault objects using Java based bean metadata

```
@Configuration
public class AppConfig extends AbstractVaultConfiguration {

    /**
     * Specify an endpoint for connecting to Vault.
     */
    @Override
    public VaultEndpoint vaultEndpoint() {
        return new VaultEndpoint();                          ①
    }

    /**
     * Configure a client authentication.
     * Please consider a more secure authentication method
     * for production use.
     */
    @Override
    public ClientAuthentication clientAuthentication() {
        return new TokenAuthentication("…");                 ②
    }
}
```

① Create a new `VaultEndpoint` that points by default to `https://localhost:8200`.

② This sample uses `TokenAuthentication` to get started quickly. See Authentication Methods for details on supported authentication methods.

Example 7. Registering Spring Vault applying injected properties

```
@Configuration
public class AppConfig extends AbstractVaultConfiguration {

    @Value("${vault.uri}")
    URI vaultUri;

    /**
     * Specify an endpoint that was injected as URI.
     */
    @Override
    public VaultEndpoint vaultEndpoint() {
        return VaultEndpoint.from(vaultUri);                        ①
    }

    /**
     * Configure a Client Certificate authentication.
     * {@link RestOperations} can be obtained from {@link #restOperations()}.
     */
    @Override
    public ClientAuthentication clientAuthentication() {
        return new ClientCertificateAuthentication(restOperations()); ②
    }
}
```

① `VaultEndpoint` can be constructed using various factory methods such as `from(URI uri)` or `VaultEndpoint.create(String host, int port)`.

② Dependencies for `ClientAuthentication` methods can be obtained either from `AbstractVaultConfiguration` or provided by your configuration.

Creating a custom configuration class might be cumbersome in some cases. Take a look at `EnvironmentVaultConfiguration` that allows configuration by using properties from existing property sources and Spring's `Environment`. Read more in Using `EnvironmentVaultConfiguration`.

9.2. Session Management

Spring Vault requires a `ClientAuthentication` to login and access Vault. See Authentication Methods on details regarding authentication. 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. Spring Vault comes with two implementations:

- `SimpleSessionManager`: Just obtains tokens from the supplied `ClientAuthentication` without refresh and revocation

- `LifecycleAwareSessionManager`: This `SessionManager` schedules token renewal if a token is renewable and revoke a login token on disposal. Renewal is scheduled with an `AsyncTaskExecutor`. `LifecycleAwareSessionManager` is configured by default if using `AbstractVaultConfiguration`.

9.3. Using `EnvironmentVaultConfiguration`

Spring Vault includes `EnvironmentVaultConfiguration` configure the Vault client from Spring's `Environment` and a set of predefined property keys. `EnvironmentVaultConfiguration` supports frequently applied configurations. Other configurations are supported by deriving from the most appropriate configuration class. Include `EnvironmentVaultConfiguration` with `@Import(EnvironmentVaultConfiguration.class)` to existing Java-based configuration classes and supply configuration properties through any of Spring's `PropertySource`s.

Example 8. Using EnvironmentVaultConfiguration with a property file

Java-based configuration class

```
@PropertySource("vault.properties")
@Import(EnvironmentVaultConfiguration.class)
public class MyConfiguration{
}
```

vault.properties

```
vault.uri=https://localhost:8200
vault.token=00000000-0000-0000-0000-000000000000
```

Property keys

- Vault URI: `vault.uri`
- SSL Configuration
  - Keystore resource: `vault.ssl.key-store` (optional)
  - Keystore password: `vault.ssl.key-store-password` (optional)
  - Keystore type: `vault.ssl.key-store-type` (optional, typically `jks`, supports also `pem`)
  - Truststore resource: `vault.ssl.trust-store` (optional)
  - Truststore password: `vault.ssl.trust-store-password` (optional)
  - Truststore type: `vault.ssl.trust-store-type` (optional, typically `jks`, supports also `pem`)
- Authentication method: `vault.authentication` (defaults to `TOKEN`, supported authentication methods are: `TOKEN`, `APPID`, `APPROLE`, `AWS_EC2`, `AZURE`, `CERT`, `CUBBYHOLE`, `KUBERNETES`)

Authentication-specific property keys

Token authentication

- Vault Token: `vault.token`

AppId authentication

- AppId path: `vault.app-id.app-id-path` (defaults to `app-id`)
- AppId: `vault.app-id.app-id`
- UserId: `vault.app-id.user-id`. `MAC_ADDRESS` and `IP_ADDRESS` use `MacAddressUserId`, respective `IpAddressUserId` user id mechanisms. Any other value is used with `StaticUserId`.

AppRole authentication

- AppRole path: `vault.app-role.app-role-path` (defaults to `approle`)
- RoleId: `vault.app-role.role-id`
- SecretId: `vault.app-role.secret-id` (optional)

AWS-EC2 authentication

- AWS EC2 path: `vault.aws-ec2.aws-ec2-path` (defaults to `aws-ec2`)
- Role: `vault.aws-ec2.role`
- RoleId: `vault.aws-ec2.role-id` (**deprecated:** use `vault.aws-ec2.role` instead)
- Identity Document URL: `vault.aws-ec2.identity-document` (defaults to `http://169.254.169.254/latest/dynamic/instance-identity/pkcs7`)

Azure (MSI) authentication

- Azure MSI path: `vault.azure-msi.azure-path` (defaults to `azure`)
- Role: `vault.azure-msi.role`
- Metadata Service URL: `vault.azure-msi.metadata-service` (defaults to `http://169.254.169.254/metadata/instance?api-version=2017-08-01`)
- Identity TokenService URL: `vault.azure-msi.identity-token-service` (defaults to `http://169.254.169.254/metadata/identity/oauth2/token?resource=https://vault.hashicorp.com&api-version=2018-02-01`)

TLS certificate authentication

No configuration options.

Cubbyhole authentication

- Initial Vault Token: `vault.token`

Kubernetes authentication

- Kubernetes path: `vault.kubernetes.kubernetes-path` (defaults to `kubernetes`)
- Role: `vault.kubernetes.role`
- Path to service account token file: `vault.kubernetes.service-account-token-file` (defaults to `/var/run/secrets/kubernetes.io/serviceaccount/token`)

9.4. Execution callbacks

One common design feature of all Spring template classes is that all functionality is routed into one of the templates execute callback methods. This helps ensure that exceptions and any resource management that maybe required are performed consistency. While this was of much greater need in the case of JDBC and JMS than with Vault, it still offers a single spot for access and logging to occur. As such, using the execute callback is the preferred way to access the Vault API to perform uncommon operations that we've not exposed as methods on `VaultTemplate`.

Here is a list of execute callback methods.

- `<T> T` **doWithVault** `(RestOperationsCallback<T> callback)` Executes the given `RestOperationsCallback`, allows to interact with Vault using `RestOperations` without requiring a session.
- `<T> T` **doWithSession** `(RestOperationsCallback<T> callback)` Executes the given `RestOperationsCallback`, allows to interact with Vault in an authenticated session.

Here is an example that uses the `ClientCallback` to initialize Vault:

```
vaultOperations.doWithVault(new RestOperationsCallback<
VaultInitializationResponse>() {

  @Override
  public VaultInitializationResponse doWithRestOperations(RestOperations
restOperations) {

    ResponseEntity<VaultInitializationResponse> exchange = restOperations
                        .exchange("/sys/init", HttpMethod.PUT,
                                  new HttpEntity<Object>(request),
                                  VaultInitializationResponse.class);

    return exchange.getBody();
    }
});
```

Chapter 10. Supporting for Vault's Secret Engines

Spring Vault ships with several extensions to support Vault's various secret engines.

Specifically, Spring Vault ships with extensions for:

- Key-Value Version 1 ("unversioned secrets")
- Key-Value Version 2 ("versioned secrets")
- PKI (Public Key Infrastructure)
- Token Authentication Backend
- Transform (Enterprise Feature)
- Transit Backend
- System Backend

You can use all other backends through methods on `VaultTemplate` directly (`VaultTemplate.read(…)`, `VaultTemplate.write(…)`).

10.1. Key-Value Version 1 ("unversioned secrets")

The `kv` secrets engine is used to store arbitrary secrets within the configured physical storage for Vault.

When running the `kv` secrets engine in a non-versioned way, only the most recently written value for a key is preserved. The benefits of non-versioned kv is a reduced storage size for each key, since no additional metadata or history is stored. Additionally, requests going to a backend configured this way are more performant because there are fewer storage calls and no locking for any given request.

Spring Vault ships with a dedicated Key-Value API to encapsulate differences between the individual Key-Value API implementations. `VaultKeyValueOperations` follows the Vault CLI design. That's the primary command line tool for Vault providing commands such as `vault kv get`, `vault kv put` and so on.

You can use this API with both Key-Value engine versions by specifying the version and mount path. The following example uses the Key-Value version 1:

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());
VaultKeyValueOperations keyValueOperations = operations.opsForKeyValue("secret",
                            VaultKeyValueOperationsSupport.KeyValueBackend.KV_1);

keyValueOperations.put("elvis", Collections.singletonMap("password", "409-52-2002
"));

VaultResponse read = keyValueOperations.get("elvis");
read.getRequiredData().get("social-security-number");
```

`VaultKeyValueOperations` supports all Key-Value operations such as `put`, `get`, `delete`, `list`.

Alternatively, the API can be used through `VaultTemplate` because of its direct mapping and simple use, as keys and responses map directly to input and output keys. The following example illustrates writing and reading a secret at `mykey`. The `kv` secrets engine is mounted at `secret`:

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());

operations.write("secret/elvis", Collections.singletonMap("social-security-number
", "409-52-2002"));

VaultResponse read = operations.read("secret/elvis");
read.getRequiredData().get("social-security-number");
```

You can find more details about the Vault Key-Value version 1 API in the Vault reference documentation.

10.2. Key-Value Version 2 ("versioned secrets")

You can run the `kv` secrets engine in one of two versions. This section explains using version 2. When running version 2 of the `kv` backend a key can retain a configurable number of versions. You can retrieve the metadata and data of the older versions. Additionally, you can use check-and-set operations to avoid unintentionally overwriting data.

Similar to Key-Value Version 1 ("unversioned secrets"), Spring Vault ships with a dedicated Key-Value API to encapsulate differences between the individual Key-Value API implementations. Spring Vault ships with a dedicated Key-Value API to encapsulate differences between the individual Key-Value API implementations. `VaultKeyValueOperations` follows the Vault CLI design. That is the primary command line tool for Vault, providing commands such as `vault kv get`, `vault kv put`, and so on.

You can use this API with both Key-Value engine versions by specifying the version and mount path. The following example uses Key-Value version 2:

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());
VaultKeyValueOperations keyValueOperations = operations.opsForKeyValue("secret",
                            VaultKeyValueOperationsSupport.KeyValueBackend.KV_2);

keyValueOperations.put("elvis", Collections.singletonMap("social-security-number",
"409-52-2002"));

VaultResponse read = keyValueOperations.get("elvis");
read.getRequiredData().get("social-security-number");
```

`VaultKeyValueOperations` supports all Key-Value operations, such as `put`, `get`, `delete`, `list`.

You can also interact with the specifics of the versioned key-value API. This is useful if you want to obtain a specific secret or you need access to the metadata.

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());
VaultVersionedKeyValueOperations versionedOperations = operations
.opsForVersionedKeyValue("secret");

Versioned.Metadata metadata = versionedOperations.put("elvis",
①
                    Collections.singletonMap("social-security-number", "409-52-
2002"));

Version version = metadata.getVersion();
②

Versioned<Object> ssn = versionedOperations.get("elvis", Version.from(42));
③

Versioned<SocialSecurityNumber> mappedSsn = versionedOperations.get("elvis",
④
                                            Version.from(42),
SocialSecurityNumber.class);

Versioned<Map<String,String>> versioned = Versioned.create(Collections
⑤
                        .singletonMap("social-security-number", "409-52-2002"),
                        Version.from(42));

versionedOperations.put("elvis", version);
```

① Store secrets at `elvis` in that is available under the `secret/` mount.

② Storing data in the versioned backend returns metadata such as the version number.

③ The versioned Key-Value API allows retrieval of specific versions identified by the version number.

④ Versioned key-value secrets can be mapped into value objects.

⑤ When updating versioned secrets using CAS, the input must refer to the previously obtained version.

While using the `kv` v2 secrets engine through `VaultTemplate` is possible. It is not the most convenient approach since the API offers a different approach to context paths and how input/output is represented. Specifically, interaction with the actual secrets requires wrapping and unwrapping of the data section and introducing a `data/` path segment between the mount and the secrets key.

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());

operations.write("secret/data/elvis", Collections.singletonMap("data",
            Collections.singletonMap("social-security-number", "409-52-2002")));

VaultResponse read = operations.read("secret/data/ykey");
Map<String,String> data = (Map<String, String>) read.getRequiredData().get("data"
);
data.get("social-security-number");
```

You can find more details about the Vault Key-Value version 2 API in the Vault reference documentation.

10.3. PKI (Public Key Infrastructure)

The `pki` secrets engine 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 to a CA, and waiting for a verification and signing process to complete. Vault's built-in authentication and authorization mechanisms provide the verification functionality.

Spring Vault supports issuing, signing, revoking certificates, and CRL retrieval through `VaultPkiOperations`. All other PKI functionality can be used through `VaultOperations`.

The following examples explain briefly the use of how to issue and revoke certificates:

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());
VaultPkiOperations pkiOperations = operations.opsForPki("pki");

VaultCertificateRequest request = VaultCertificateRequest.builder()
①
            .ttl(Duration.ofHours(48))
            .altNames(Arrays.asList("prod.dc-1.example.com", "prod.dc-
2.example.com"))
            .withIpSubjectAltName("1.2.3.4")
            .commonName("hello.example.com")
            .build();

VaultCertificateResponse response = pkiOperations.issueCertificate("production",
request);      ②
CertificateBundle certificateBundle = response.getRequiredData();

KeyStore keyStore = certificateBundle.createKeyStore("my-keystore");
③

KeySpec privateKey = certificateBundle.getPrivateKeySpec();
④
X509Certificate certificate = certificateBundle.getX509Certificate();
X509Certificate caCertificate = certificateBundle.getX509IssuerCertificate();

pkiOperations.revoke(certificateBundle.getSerialNumber());
⑤
```

① Construct a certificate request by using the `VaultCertificateRequest` builder.

② Request a certificate from Vault. Vault acts as certificate authority and responds with a signed X.509 certificate. The actual response is a `CertificateBundle`.

③ You can obtain generated certificates directly as Java KeyStore that contains public and private keys as well as the issuer certificate. KeyStore has a wide range of uses, which makes this format suitable to configure (for example a HTTP client, a database driver, or an SSL-secured HTTP server).

④ `CertificateBundle` allows accessing the private key and the public and issuer certificates directly through the Java Cryptography Extension API.

⑤ Once a certificate is no longer in use (or it was compromised), you can revoke it through its serial number. Vault includes the revoked certificate in its CRL.

You can find more details about the Vault PKI secrets API in the Vault reference documentation.

10.4. Token Authentication Backend

This backend is an authentication backend that does not interact with actual secrets. Rather, it gives access to access token management. You can read more about Token-based authentication in the

authentication methods chapter.

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.

When any other auth method returns an identity, Vault core invokes the token method to create a new unique token for that identity.

You can also use the token store to bypass any other auth method. You can create tokens directly, as well as perform a variety of other operations on tokens, such as renewal and revocation.

Spring Vault uses this backend to renew and revoke the session tokens supplied by the configured authentication method.

The following examples show how to request, renew and revoke a Vault token from within your application:

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());
VaultTokenOperations tokenOperations = operations.opsForToken();

VaultTokenResponse tokenResponse = tokenOperations.create();
①
VaultToken justAToken = tokenResponse.getToken();

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

VaultTokenResponse appTokenResponse = tokenOperations.create(tokenRequest);
②
VaultToken appToken = appTokenResponse.getToken();

tokenOperations.renew(appToken);
③

tokenOperations.revoke(appToken);
④
```

① Create an token by applying role defaults.

② Using the builder API, you can define fine-grained settings for the token to request. Requesting a token returns a `VaultToken`, which is used as value object for Vault tokens.

③ You can renew tokens through the Token API. Typically, that is done by `SessionManager` to keep track of the Vault session token.

④ Tokens can be revoked if needed through the Token API. Typically, that is done by `SessionManager` to keep track of the Vault session token.

You can find more details about the Vault Token Auth Method API in the Vault reference documentation.

10.5. Transit Backend

The transit secrets engine handles cryptographic functions on data in-transit. Vault does not store the data sent to this secrets engine. It can also be seen as "cryptography as a service" or "encryption as a service". The transit secrets engine can also sign and verify data, generate hashes and HMACs of data, and act as a random bytes source.

The primary use case for transit is to encrypt data from applications while still storing that encrypted data in some primary data store. This relieves the burden of proper encryption and decryption from application developers and pushes the burden onto the operators of Vault.

Spring Vault supports a wide range of Transit operations:

- Key creation
- Key reconfiguration
- Encryption/Decryption/Rewrapping
- HMAC computation
- Signing and signature verification

All operations within `transit` are centered around keys. The Transit engine supports the versioning of keys and a variety of key types. Note that the key type may impose a limitation on which operations can used.

The following examples shows how to create a key and how to encrypt and decrypt data:

```
VaultOperations operations = new VaultTemplate(new VaultEndpoint());
VaultTransitOperations transitOperations = operations.opsForTransit("transit");

transitOperations.createKey("my-aes-key", VaultTransitKeyCreationRequest.
ofKeyType("aes128-gcm96"));    ①

String ciphertext = transitOperations.encrypt("my-aes-key", "plaintext to encrypt
");                    ②

String plaintext = transitOperations.decrypt("my-aes-key", ciphertext);
③
```

① First, we need a key to begin with. Each key requires the type to be specified. `aes128-gcm96` supports encryption, decryption, key derivation, and convergent encryption, of which we need encryption and decryption for this example.

② Next, we encrypt a `String` that contains the plain text that should be encrypted. The input `String` uses the default `Charset` to encode the string into its binary representation. Requesting a token returns a `VaultToken`, which is used as value object for Vault tokens. The `encrypt` method returns Base64-encoded ciphertext, typically starting with `vault:`.

③ To decrypt ciphertext into plain text, call the `decrypt` method. It decrypts the ciphertext and returns a `String` that is decoded using the default charset.

The preceeding example uses simple strings for cryptographic operations. While it is a simple approach, it bears the risk of charset misconfiguration and is not binary-safe. Binary safety is required when the plain text uses a binary representation for data such as images, compressed data, or binary data structures.

To encrypt and decrypt binary data, use the `Plaintext` and `Ciphertext` value objects that can hold binary values:

```
byte [] plaintext = "plaintext to encrypt".getBytes();

Ciphertext ciphertext = transitOperations.encrypt("my-aes-key", Plaintext.of
(plaintext));           ①

Plaintext decrypttedPlaintext = transitOperations.decrypt("my-aes-key",
ciphertext);                ②
```

① Assuming a key `my-aes-key` is already in place, we're encrypting the `Plaintext` object. In return, the `encrypt` method returns a `Ciphertext` object.

② The `Ciphertext` object can be used directly for decryption and returns a `Plaintext` object.

`Plaintext` and `Ciphertext` come with a contextual object, `VaultTransitContext`. It is used to supply a nonce value for convergent encryption and for a context value to make use of key derivation.

Transit allows for signing plain text and verifying the signature for a given plain text. Sign operations require an asymmetric key, typically using Elliptic Curve Cryptography or RSA.

> Signatures use the public/private key split to ensure authenticity.
> The signer uses its private key to create a signature. Otherwise, anybody would be able to sign messages in your name. The verifier uses the public key part to verify the signature. The actual signature is typically a hash value.
>
> Internally, the hash gets computed and encrypted using the private key to create the final signature. The verification decrypts the signature message, computes their own hash for the plain text and compares both hash values to check whether the signature is valid or not.

```
byte [] plaintext = "plaintext to sign".getBytes();

transitOperations.createKey("my-ed25519-key", VaultTransitKeyCreationRequest
.ofKeyType("ed25519")); ①

Signature signature = transitOperations.sign("my-ed25519-key", Plaintext.of
(plaintext));            ②

boolean valid = transitOperations.verify("my-ed25519-key", Plaintext.of(plaintext
), signature);     ③
```

① Signing requires an asymmetric key. You can use any Elliptic Curve Cryptography or RSA key type. Once the key is created, you have all the prerequisites in place to create a signature.

② The signature gets created for a plain text message. The returned `Signature` contains an ASCII-safe string that uses Base64 characters.

③ To verify the signature, the verification requires a Signature object and the plain text message. As the return value, you get whether the signature was valid or not.

You can find more details about the Vault Transit Backend in the Vault reference documentation.

Chapter 11. Introduction to ReactiveVaultTemplate

This section covers basic information on the reactive programming support using Spring Vault.

11.1. What is Reactive Programming?

In plain terms reactive programming is about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale vertically (i.e. within the JVM) rather than horizontally (i.e. through clustering).

A key aspect of reactive applications is the concept of backpressure which is a mechanism to ensure producers don't overwhelm consumers. For example in a pipeline of reactive components extending from the database to the HTTP response when the HTTP connection is too slow the data repository can also slow down or stop completely until network capacity frees up.

11.2. Reactive Vault Client

Spring Vault's reactive client support is built on top of composable authentication steps and Spring's functional `WebClient` via Reactor Netty or Jetty, which feature both a fully non-blocking, event-driven HTTP client.

It exposes `VaultTokenSupplier` as supplier of `VaultToken` to authenticate HTTP requests and `ReactiveVaultOperations` as the primary entry point. The core configuration of `VaultEndpoint`, `ClientOptions` and SSL are reused across the various client implementation.

The class `ReactiveVaultTemplate`, located in the package `org.springframework.vault.core`, is the central class of the Spring's reactive Vault support providing a rich feature set to interact with Vault. The template offers convenience operations to read, write and delete data in Vault and provides a mapping between your domain objects and Vault data.

> Once configured, `ReactiveVaultTemplate` is thread-safe and can be reused across multiple instances.

The mapping between Vault documents and domain classes is done by delegating to `WebClient` and its codecs.

The `ReactiveVaultTemplate` class implements the interface `ReactiveVaultOperations`. In as much as possible, the methods on `ReactiveVaultOperations` are named after methods available on the Vault API to make the API familiar to existing Vault developers who are used to the API and CLI. For example, you will find methods such as "write", "delete", and "read". The design goal was to make it as easy as possible to transition between the use of the Vault API and `ReactiveVaultOperations`. A major difference in between the two APIs is that `ReactiveVaultOperations` can be passed domain objects instead of JSON Key-Value pairs.

> The preferred way to reference the operations on `ReactiveVaultTemplate` instance is via its interface `ReactiveVaultOperations`.

Functionality not explicitly exposed by the `ReactiveVaultTemplate` you can use one of several execute callback methods to access underlying APIs. The execute callbacks will give you a reference to a `WebClient` object. Please see the section Execution Callbacks for more information.

Now let's look at a examples of how to work with Vault in the context of the Spring container.

11.3. Registering and configuring Spring Vault beans

Using Spring Vault does not require a Spring Context. However, instances of `ReactiveVaultTemplate` and `VaultTokenSupplier` registered inside a managed context will participate in lifecycle events provided by the Spring IoC container. This is useful to dispose active Vault sessions upon application shutdown. You also benefit from reusing the same `ReactiveVaultTemplate` instance across your application.

Spring Vault comes with a supporting configuration class that provides bean definitions for use inside a Spring context. Application configuration classes typically extend from `AbstractVaultConfiguration` and are required to provide additional details that are environment specific.

Extending from `AbstractVaultConfiguration` requires to implement ` VaultEndpoint vaultEndpoint()` and `ClientAuthentication clientAuthentication()` methods.

Example 9. Registering Spring Vault objects using Java based bean metadata

```
@Configuration
public class AppConfig extends AbstractReactiveVaultConfiguration {

    /**
     * Specify an endpoint for connecting to Vault.
     */
    @Override
    public VaultEndpoint vaultEndpoint() {
        return new VaultEndpoint();                          ①
    }

    /**
     * Configure a client authentication.
     * Please consider a more secure authentication method
     * for production use.
     */
    @Override
    public ClientAuthentication clientAuthentication() {
        return new TokenAuthentication("…");                 ②
    }
}
```

① Create a new `VaultEndpoint` that points by default to `https://localhost:8200`.

② This sample uses `TokenAuthentication` to get started quickly. See Authentication Methods for details on supported authentication methods.

11.4. Session Management

Spring Vault requires a token to authenticate Vault requests. See Authentication Methods on details regarding authentication. The reactive client requires a non-blocking token supplier whose contract is defined in `VaultTokenSupplier`. Tokens can be static or obtained through a declared authentication flow. Vault login should not occur on each authenticated Vault interaction but the session token should be kept across a session. This aspect is handled by a session manager implementing `ReactiveSessionManager`, such as `ReactiveLifecycleAwareSessionManager`.

11.5. Execution callbacks

One common design feature of all Spring template classes is that all functionality is routed into one of the templates execute callback methods. This helps ensure that exceptions and any resource management that maybe required are performed consistency. While this was of much greater need in the case of JDBC and JMS than with Vault, it still offers a single spot for access and logging to occur. As such, using the execute callback is the preferred way to access the Vault API to perform uncommon operations that we've not exposed as methods on `ReactiveVaultTemplate`.

Here is a list of execute callback methods.

- `<T> T` **doWithVault** `(Function<WebClient, ? extends T> clientCallback)` Composes a reactive sequence the given `WebClient`, allows to interact with Vault without a session context.
- `<T> T` **doWithSession** `(Function<WebClient, ? extends T> clientCallback)` Composes a reactive sequence the given `WebClient`, allows to interact with Vault in an authenticated session.

Here is an example that uses the callback to initialize Vault:

```
reactiveVaultOperations.doWithVault(webClient -> {

    return webClient.put()
                    .uri("/sys/init")
                    .syncBody(request)
                    .retrieve()
                    .toEntity(VaultInitializationResponse.class);
});
```

Chapter 12. Vault Property Source Support

Vault can be used in many different ways. One specific use-case is using Vault to store encrypted properties. Spring Vault supports Vault as property source to obtain configuration properties using Spring's PropertySource abstraction.

> You can reference properties stored inside Vault in other property sources or use value injection with `@Value(…)`. Special attention is required when bootstrapping beans that require data stored inside of Vault. A `VaultPropertySource` must be initialized at that time to retrieve properties from Vault.

> Spring Boot/Spring Cloud users can benefit from Spring Cloud Vault's configuration integration that initializes various property sources during application startup.

12.1. Registering `VaultPropertySource`

Spring Vault provides a `VaultPropertySource` to be used with Vault to obtain properties. It uses the nested `data` element to expose properties stored and encrypted in Vault.

```
ConfigurableApplicationContext ctx = new GenericApplicationContext();
MutablePropertySources sources = ctx.getEnvironment().getPropertySources();
sources.addFirst(new VaultPropertySource(vaultTemplate, "secret/my-application"));
```

In the code above, `VaultPropertySource` has been added with highest precedence in the search. If it contains a ´foo` property, it will be detected and returned ahead of any `foo` property in any other `PropertySource`. `MutablePropertySources` exposes a number of methods that allow for precise manipulation of the set of property sources.

12.2. @VaultPropertySource

The `@VaultPropertySource` annotation provides a convenient and declarative mechanism for adding a `PropertySource` to Spring's `Environment` to be used in conjunction with `@Configuration` classes.

`@VaultPropertySource` takes a Vault path such as `secret/my-application` and exposes the data stored at the node in a `PropertySource`. `@VaultPropertySource` supports lease renewal for secrets associated with a lease (i. e. credentials from the `mysql` backend) and credential rotation upon terminal lease expiration. Lease renewal is disabled by default.

Example 10. Properties stored in Vault

```
{
  // …

  "data": {
    "database": {
      "password": ...
    },
    "user.name": ...,
  }

  // …
}
```

Example 11. Declaring a `@VaultPropertySource`

```
@Configuration
@VaultPropertySource("secret/my-application")
public class AppConfig {

    @Autowired Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setUser(env.getProperty("user.name"));
        testBean.setPassword(env.getProperty("database.password"));
        return testBean;
    }
}
```

Example 12. Declaring a `@VaultPropertySource` *with credential rotation and prefix*

```
@Configuration
@VaultPropertySource(value = "aws/creds/s3-access",
                     propertyNamePrefix = "aws.",
                     renewal = Renewal.ROTATE)
public class AppConfig {
  // provides aws.access_key and aws.secret_key properties
}
```

Secrets obtained from `generic` secret backends are associated with a TTL (`refresh_interval`) but not a lease Id. Spring Vault's `PropertySource` rotates generic secrets when reaching its TTL.

You can use `@VaultPropertySource` to obtain the newest secret version from the versioned Key-Value backend. Make sure to not include the `data/` segment in the path.

Any `${…}` placeholders present in a `@VaultPropertySource` path are resolved against the set of property sources already registered against the environment, as the following example shows:

Example 13. Declaring a `@VaultPropertySource` path using placeholders

```
@Configuration
@VaultPropertySource(value = "aws/creds/${my.placeholder:fallback/value}",
                     propertyNamePrefix = "aws.",
                     renewal = Renewal.ROTATE)
public class AppConfig {
}
```

Assuming that `my.placeholder` is present in one of the property sources already registered (for example, system properties or environment variables), the placeholder is resolved to the corresponding value. If not, then `fallback/value` is used as a default. If no default is specified and a property cannot be resolved, an `IllegalArgumentException` is thrown.

In certain situations, it may not be possible or practical to tightly control property source ordering when using `@VaultPropertySource` annotations. For example, if the `@Configuration` classes above were registered via component-scanning, the ordering is difficult to predict. In such cases - and if overriding is important - it is recommended that the user fall back to using the programmatic PropertySource API. See `ConfigurableEnvironment` and `MutablePropertySources` for details.

Chapter 13. Vault Repositories

Working with `VaultTemplate` and responses mapped to Java classes allows basic data operations like read, write and delete. Vault repositories apply Spring Data's repository concept on top of Vault. A Vault repository exposes basic CRUD functionality and supports query derivation with predicates constraining the Id property, paging and sorting.

> Read more about Spring Data Repositories in the Spring Data Commons reference documentation. The reference documentation will give you an introduction to Spring Data repositories.

13.1. Usage

To access domain entities stored in Vault you can leverage repository support that eases implementing those quite significantly.

Example 14. Sample Credentials Entity

```
@Secret
public class Credentials {

  @Id String id;
  String password;
  String socialSecurityNumber;
  Address address;
}
```

We have a pretty simple domain object here. Note that it has a property named `id` annotated with `org.springframework.data.annotation.Id` and a `@Secret` annotation on its type. Those two are responsible for creating the actual key used to persist the object as JSON inside Vault.

> Properties annotated with `@Id` as well as those named `id` are considered as the identifier properties. Those with the annotation are favored over others.

The next step is to declare a repository interface that uses the domain object.

Example 15. Basic Repository Interface for `Credentials` entities

```
public interface CredentialsRepository extends CrudRepository<Credentials, String>
{

}
```

As our repository extends `CrudRepository` it provides basic CRUD and query methods. Vault

repositories require Spring Data components. Make sure to include `spring-data-commons` and `spring-data-keyvalue` artifacts in your class path.

The easiest way to achive this, is by setting up dependency management and adding the artifacts to your `pom.xml`:

Then add the following to `pom.xml` dependencies section.

Example 16. Using the Spring Data BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-bom</artifactId>
      <version>2020.0.2</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.vault</groupId>
    <artifactId>spring-vault-core</artifactId>
    <version>2.3.1</version>
  </dependency>

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-keyvalue</artifactId>
    <!-- Version inherited from the BOM -->
  </dependency>

</dependencies>
```

The thing we need in between to glue things together is the according Spring configuration.

Example 17. JavaConfig for Vault Repositories

```
@Configuration
@EnableVaultRepositories
public class ApplicationConfig {

  @Bean
  public VaultTemplate vaultTemplate() {
    return new VaultTemplate(…);
  }
}
```

Given the setup above we can go on and inject `CredentialsRepository` into our components.

Example 18. Access to Person Entities

```
@Autowired CredentialsRepository repo;

public void basicCrudOperations() {

  Credentials creds = new Credentials("heisenberg", "327215", "AAA-GG-SSSS");
  rand.setAddress(new Address("308 Negra Arroyo Lane", "Albuquerque", "New Mexico
", "87104"));

  repo.save(creds);                                          ①

  repo.findOne(creds.getId());                               ②

  repo.count();                                              ③

  repo.delete(creds);                                        ④
}
```

① Stores properties of `Credentials` inside Vault Hash with a key pattern `keyspace/id`, in this case `credentials/heisenberg`, in the generic secret backend.

② Uses the provided id to retrieve the object stored at `keyspace/id`.

③ Counts the total number of entities available within the keyspace *credentials* defined by `@Secret` on `Credentials`.

④ Removes the key for the given object from Vault.

13.2. Object to Vault JSON Mapping

Vault repositories store objects in Vault using JSON as interchange format. Object mapping between JSON and the entity is done by `VaultConverter`. The converter reads and writes `SecretDocument` that contains the body from a `VaultResponse`. `VaultResponse`s are read from Vault and the body is

deserialized by Jackson into a `Map` of `String` and `Object`. The default `VaultConverter` implementation reads the `Map` with nested values, `List` and `Map` objects and converts these to entities and vice versa.

Given the `Credentials` type from the previous sections the default mapping is as follows:

```
{
  "_class": "org.example.Credentials",                ①
  "password", "327215",                               ②
  "socialSecurityNumber": "AAA-GG-SSSS",
  "address": {                                        ③
    "street": "308 Negra Arroyo Lane",
    "city": "Albuquerque",
    "state": "New Mexico",
    "zip":"87104"
  }
}
```

① The `_class` attribute is included on root level as well as on any nested interface or abstract types.

② Simple property values are mapped by path.

③ Properties of complex types are mapped as nested objects.

The `@Id` property must be mapped to `String`.

Table 1. Default Mapping Rules

Type	Sample	Mapped Value
Simple Type (eg. String)	String firstname = "Walter";	firstname = "Walter"
Complex Type (eg. Address)	Address adress = new Address("308 Negra Arroyo Lane");	address: { "street": "308 Negra Arroyo Lane" }
List of Simple Type	List<String> nicknames = asList("walt", "heisenberg");	nicknames: ["walt", "heisenberg"]
Map of Simple Type	Map<String, Integer> atts = asMap("age", 51)	atts : {"age" : 51}
List of Complex Type	List<Address> addresses = asList(new Address("308...	address: [{ "street": "308 Negra Arroyo Lane" }, ...]

You can customize the mapping behavior by registering a `Converter` in `VaultCustomConversions`. Those converters can take care of converting from/to a type such as `LocalDate` as well as `SecretDocument` whereas the first one is suitable for converting simple properties and the last one complex types to their JSON representation. The second option offers full control over the resulting `SecretDocument`. Writing objects to `Vault` will delete the content and re-create the whole entry, so not mapped data will be lost.

13.3. Queries and Query Methods

Query methods allow automatic derivation of simple queries from the method name. Vault has no query engine but requires direct access of HTTP context paths. Vault query methods translate Vault's API possibilities to queries. A query method execution lists children under a context path, applies filtering to the Id, optionally limits the Id stream with offset/limit and applies sorting after fetching the results.

Example 19. Sample Repository Query Method

```
public interface CredentialsRepository extends CrudRepository<Credentials, String>
{

  List<Credentials> findByIdStartsWith(String prefix);
}
```

Query methods for Vault repositories support only queries with predicates on the `@Id` property.

Here's an overview of the keywords supported for Vault.

Table 2. Supported keywords for query methods

Keyword	Sample
`After`, `GreaterThan`	`findByIdGreaterThan(String id)`
`GreaterThanEqual`	`findByIdGreaterThanEqual(String id)`
`Before`, `LessThan`	`findByIdLessThan(String id)`
`LessThanEqual`	`findByIdLessThanEqual(String id)`
`Between`	`findByIdBetween(String from, String to)`
`In`	`findByIdIn(Collection ids)`
`NotIn`	`findByIdNotIn(Collection ids)`
`Like`, `StartingWith`, `EndingWith`	`findByIdLike(String id)`
`NotLike`, `IsNotLike`	`findByIdNotLike(String id)`
`Containing`	`findByFirstnameContaining(String id)`
`NotContaining`	`findByFirstnameNotContaining(String name)`
`Regex`	`findByIdRegex(String id)`
`(No keyword)`	`findById(String name)`
`Not`	`findByIdNot(String id)`
`And`	`findByLastnameAndFirstname`
`Or`	`findByLastnameOrFirstname`
`Is,Equals`	`findByFirstname`,`findByFirstnameIs`,`findByFirstnameEquals`
`Top,First`	`findFirst10ByFirstname`,`findTop5ByFirstname`

13.3.1. Sorting and Paging

Query methods support sorting and paging by selecting in memory a sublist (offset/limit) Id's retrieved from a Vault context path. Sorting has is not limited to a particular field, unlike query method predicates. Unpaged sorting is applied after Id filtering and all resulting secrets are fetched from Vault. This way a query method fetches only results that are also returned as part of the result.

Using paging and sorting requires secret fetching before filtering the Id's which impacts performance. Sorting and paging guarantees to return the same result even if the natural order of Id returned by Vault changes. Therefore, all Id's are fetched from Vault first, then sorting is applied and afterwards filtering and offset/limiting.

Example 20. Paging and Sorting Repository

```
public interface CredentialsRepository extends PagingAndSortingRepository
<Credentials, String> {

  List<Credentials> findTop10ByIdStartsWithOrderBySocialSecurityNumberDesc(String
prefix);

  List<Credentials> findByIdStarts(String prefix, Pageable pageRequest);
}
```

Chapter 14. Client support

Spring Vault supports various HTTP clients to access Vault's HTTP API. Spring Vault uses `RestTemplate` as primary interface accessing Vault. Dedicated client support originates from customized SSL configuration that is scoped only to Spring Vault's client components.

Spring Vault supports following HTTP imperative clients:

- Java's builtin `HttpURLConnection` (default client)
- Apache Http Components
- Netty
- OkHttp 3

Spring Vault's reactive integration supports the following reactive HTTP clients:

- Reactor Netty
- Jetty

Using a specific client requires the according dependency to be available on the classpath so Spring Vault can use the available client for communicating with Vault.

14.1. Java's builtin `HttpURLConnection`

Java's builtin `HttpURLConnection` is available out-of-the-box without additional configuration. Using `HttpURLConnection` comes with a limitation regarding SSL configuration. Spring Vault won't apply customized SSL configuration as it would require a deep reconfiguration of the JVM. This configuration would affect all components relying on the default SSL context. Configuring SSL settings using `HttpURLConnection` requires you providing these settings as System Properties. See Customizing JSSE for further details.

14.2. External Clients

You can use external clients to access Vault's API. Simply add one of the following dependencies to your project. You can omit the version number if using Spring Vault's Dependency BOM

Example 21. Apache Http Components Dependency

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
</dependency>
```

Apache HttpClient's wire logging can be enabled through logging configuration. Make sure to not accidentally enable wire logging as logs may expose traffic (tokens and secrets) between your application and Vault in plain text.

Example 22. Netty Dependency

```
<dependency>
  <groupId>io.netty</groupId>
  <artifactId>netty-all</artifactId>
</dependency>
```

Example 23. Square OkHttp 3

```
<dependency>
  <groupId>com.squareup.okhttp3</groupId>
  <artifactId>okhttp</artifactId>
</dependency>
```

Example 24. Reactor Netty

```
<dependency>
  <groupId>io.projectreactor.netty</groupId>
  <artifactId>reactor-netty</artifactId>
</dependency>
```

Example 25. Jetty

```
<dependency>
  <groupId>org.eclipse.jetty</groupId>
  <artifactId>jetty-reactive-httpclient</artifactId>
</dependency>
```

14.3. Vault Client SSL configuration

SSL can be configured using `SslConfiguration` by setting various properties. You can set either `javax.net.ssl.trustStore` to configure JVM-wide SSL settings or configure `SslConfiguration` to set SSL settings only for Spring Vault.

```
SslConfiguration sslConfiguration = SslConfiguration.create(            ①
        new FileSystemResource("client-cert.jks"), "changeit".toCharArray(),
        new FileSystemResource("truststore.jks"), "changeit".toCharArray());

SslConfiguration.forTrustStore(new FileSystemResource("keystore.jks"),  ②
                                    "changeit".toCharArray())

SslConfiguration.forKeyStore(new FileSystemResource("keystore.jks"),    ③
                                    "changeit".toCharArray())

SslConfiguration.forKeyStore(new FileSystemResource("keystore.jks"),    ④
                                    "changeit".toCharArray(),
                                    KeyConfiguration.of("key-password"
.toCharArray(),
                                    "my-key-alias"))
```

① Full configuration.

② Configuring only trust store settings.

③ Configuring only key store settings.

④ Configuring only key store settings with providing a key-configuration.

Please note that providing `SslConfiguration` can be only applied when either Apache Http Components or the OkHttp client is on your class-path.

The SSL configuration supports also PEM-encoded certificates as alternative to a Java Key Store.

```
KeyStoreConfiguration keystore = KeyStoreConfiguration
        .of(new ClassPathResource("ca.pem")).withStoreType("PEM");
SslConfiguration configuration = SslConfiguration.forTrustStore(keystore);
```

PEM files may contain one or more certificates (blocks of `-----BEGIN CERTIFICATE-----` and `-----END CERTIFICATE-----`). Certificates added to the underlying `KeyStore` use the full subject name as alias.

Chapter 15. Authentication Methods

Different organizations have different requirements for security and authentication. Vault reflects that need by shipping multiple authentication methods. Spring Vault supports multiple authentications mechanisms.

15.1. Externalizing login credentials

Obtaining first-time access to a secured system is known as secure introduction. Any client requires ephemeral or permanent credentials to access Vault. Externalizing credentials is a good pattern to keep code maintainability high but comes at a risk of increased disclosure.

Disclosure of login credentials to any party allows login to Vault and access secrets that are permitted by the underlying role. Picking the appropriate client authentication and injecting credentials into the application is subject to risk evaluation.

Spring's PropertySource abstraction is a natural fit to keep configuration outside the application code. You can use system properties, environment variables or property files to store login credentials. Each approach comes with its own properties. Keep in mind that the command line and environment properties can be introspected with appropriate OS access levels.

Example 26. Externalizing `vault.token` to a properties file

```
@PropertySource("configuration.properties")
@Configuration
public class Config extends AbstractVaultConfiguration {

    @Override
    public ClientAuthentication clientAuthentication() {
        return new TokenAuthentication(getEnvironment().getProperty("vault.token"
));
    }
}
```

Spring allows multiple ways to obtain `Environment`. When using `VaultPropertySource`, injection via `@Autowired Environment environment` will not provide the `Environment` as the environment bean is still in construction and autowiring comes at a later stage. Your configuration class should rather implement `ApplicationContextAware` and obtain the `Environment` from `ApplicationContext`.

See `SecurePropertyUsage.java` for a sample on referencing properties in components and other property sources.

15.2. Token authentication

Tokens are the core method for authentication within Vault. Token authentication requires a static token to be provided.

> Token authentication is the default authentication method. If a token is disclosed an unintended party, it gains access to Vault and can access secrets for the intended client.

Typically, Token authentication is used in scenarios in which the token is created and renewed externally (such as HashiCorp Vault service broker). Depending on the actual setup, you may or may not want token renewal and revocation. See `LifecycleAwareSessionManager` for details about TTL and token revocation.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ...

    @Override
    public ClientAuthentication clientAuthentication() {
        return new TokenAuthentication("...");
    }

    // ...
}
```

See also:

- Vault Documentation: Tokens
- Vault Documentation: Using the Token auth backend

15.3. AppId authentication

> AppId authentication is deprecated by Vault. Use AppRole authentication instead.

Vault supports AppId authentication that consists of two hard to guess tokens. The AppId defaults to `spring.application.name` that is statically configured. The second token is the UserId which is a part determined by the application, usually related to the runtime environment. IP address, Mac address or a Docker container name are good examples. Spring Vault supports IP address, Mac address and static UserId's (e.g. supplied via System properties). The IP and Mac address are represented as Hex-encoded SHA256 hash.

IP address-based UserId's use the local host's IP address.

```java
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ...

    @Override
    public ClientAuthentication clientAuthentication() {
        AppIdAuthenticationOptions options = AppIdAuthenticationOptions.builder()
                .appId("myapp")
                .userIdMechanism(new IpAddressUserId())
                .build();

        return new AppIdAuthentication(options, restOperations());
    }

    // ...
}
```

The corresponding command to generate the IP address UserId from a command line is:

```
$ echo -n 192.168.99.1 | sha256sum
```

Including the line break of `echo` leads to a different hash value so make sure to include the `-n` flag.

Mac address-based UserId's obtain their network device from the localhost-bound device. The configuration also allows specifying a `network-interface` hint to pick the right device. The value of `network-interface` is optional and can be either an interface name or interface index (0-based).

```java
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ...

    @Override
    public ClientAuthentication clientAuthentication() {

        AppIdAuthenticationOptions options = AppIdAuthenticationOptions.builder()
                .appId("myapp")
                .userIdMechanism(new MacAddressUserId())
                .build();

        return new AppIdAuthentication(options, restOperations());
    }

    // ...
}
```

The corresponding command to generate the Mac address UserId from a command line is:

```
$ echo -n 0AFEDE1234AC | sha256sum
```

The Mac address is specified uppercase and without colons. Including the line break of `echo` leads to a different hash value so make sure to include the `-n` flag.

15.3.1. Custom UserId

A more advanced approach lets you implementing your own `AppIdUserIdMechanism`. This class must be on your classpath and must implement the `org.springframework.vault.authentication.AppIdUserIdMechanism` interface and the `createUserId` method. Spring Vault will obtain the UserId by calling `createUserId` each time it authenticates using AppId to obtain a token.

MyUserIdMechanism.java

```
public class MyUserIdMechanism implements AppIdUserIdMechanism {

  @Override
  public String createUserId() {

    String userId = ···
    return userId;
  }
}
```

See also: Vault Documentation: Using the App ID auth backend

15.4. AppRole authentication

AppRole allows machine authentication, like the deprecated (since Vault 0.6.1) AppId authentication. AppRole authentication consists of two hard to guess (secret) tokens: RoleId and SecretId.

Spring Vault supports AppRole authentication by providing either RoleId only or together with a provided SecretId and fetching RoleId/SecretId from Vault (push and pull modes with response unwrapping).

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ···

    @Override
    public ClientAuthentication clientAuthentication() {

        AppRoleAuthenticationOptions options = AppRoleAuthenticationOptions
.builder()
                .roleId(RoleId.provided("···"))
                .secretId(SecretId.wrapped(VaultToken.of("···")))
                .build();

        return new AppRoleAuthentication(options, restOperations());
    }

    // ···
}
```

Spring Vault also support full pull mode: If RoleId and SecretId are not provided, Spring Vault will

retrieve them using the role name and an initial token. The initial token may be associated with a TTL and usage limit.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ···

    @Override
    public ClientAuthentication clientAuthentication() {

        VaultToken initialToken = VaultToken.of("···");
        AppRoleAuthenticationOptions options = AppRoleAuthenticationOptions
.builder()
                .appRole("···")
                .roleId(RoleId.pull(initialToken))
                .secretId(SecretId.pull(initialToken))
                .build();

        return new AppRoleAuthentication(options, restOperations());
    }

    // ···
}
```

See also: Vault Documentation: Using the AppRole auth backend

15.5. AWS-EC2 authentication

The aws-ec2 auth backend provides a secure introduction mechanism for AWS EC2 instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats AWS as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each EC2 instance.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ...

    @Override
    public ClientAuthentication clientAuthentication() {
        return new AwsEc2Authentication(restOperations());
    }

    // ...
}
```

AWS-EC2 authentication enables nonce by default to follow the Trust On First Use (TOFU) principle. Any unintended party that gains access to the PKCS#7 identity metadata can authenticate against Vault.

During the first login, Spring Vault generates a nonce that is stored in the auth backend aside the instance Id. Re-authentication requires the same nonce to be sent. Any other party does not have the nonce and can raise an alert in Vault for further investigation.

The nonce is kept in memory and is lost during application restart.

AWS-EC2 authentication roles are optional and default to the AMI. You can configure the authentication role by setting it in `AwsEc2AuthenticationOptions`.

See also: Vault Documentation: Using the AWS-EC2 auth backend

15.6. AWS-IAM authentication

The aws auth backend allows Vault login by using existing AWS IAM credentials.

AWS IAM authentication creates a signed HTTP request that is executed by Vault to get the identity of the signer using AWS STS `GetCallerIdentity` method. AWSv4 signatures require IAM credentials.

IAM credentials can be obtained from either the runtime environment or supplied externally. Runtime environments such as AWS-EC2, Lambda and ECS with assigned IAM principals do not require client-specific configuration of credentials but can obtain these from its metadata source.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        AwsIamAuthenticationOptions options = AwsIamAuthenticationOptions.builder
()
                .credentials(new BasicAWSCredentials(…)).build();

        return new AwsIamAuthentication(options, restOperations());
    }

    // …
}
```

Example 27. Using AWS-EC2 instance profile as credentials source

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        AwsIamAuthenticationOptions options = AwsIamAuthenticationOptions.builder
()
                .credentialsProvider(InstanceProfileCredentialsProvider
.getInstance()).build();

        return new AwsIamAuthentication(options, restOperations());
    }

    // …
}
```

`AwsIamAuthentication` requires the AWS Java SDK dependency (`com.amazonaws:aws-java-sdk-core`) as the authentication implementation uses AWS SDK types for credentials and request signing.

You can configure the authentication via `AwsIamAuthenticationOptions`.

See also:

- Vault Documentation: Using the AWS auth backend
- AWS Documentation: STS GetCallerIdentity

15.7. Azure (MSI) authentication

The azure auth backend provides a secure introduction mechanism for Azure VM instances, allowing automated retrieval of a Vault token. Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats Azure as a Trusted Third Party and uses the managed service identity and instance metadata information that can be bound to a VM instance.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        AzureMsiAuthenticationOptions options = AzureMsiAuthenticationOptions
.builder()
                    .role(…).build();

        return new AzureMsiAuthentication(options, restOperations());
    }

    // …
}
```

Azure authentication requires details about the VM environment (subscription Id, resource group name, VM name). These details can be either configured through `AzureMsiAuthenticationOptionsBuilder`. If left unconfigured, `AzureMsiAuthentication` queries Azure's instance metadata service to obtain these details.

See also:

- Vault Documentation: Using the Azure auth backend
- Azure Documentation: Managed Service Identity

15.8. GCP-GCE authentication

The gcp auth backend allows Vault login by using existing GCP (Google Cloud Platform) IAM and GCE credentials.

GCP GCE (Google Compute Engine) authentication creates a signature in the form of a JSON Web

Token (JWT) for a service account. A JWT for a Compute Engine instance is obtained from the GCE metadata service using Instance identification. This API creates a JSON Web Token that can be used to confirm the instance identity.

Unlike most Vault authentication backends, this backend does not require first-deploying, or provisioning security-sensitive credentials (tokens, username/password, client certificates, etc.). Instead, it treats GCP as a Trusted Third Party and uses the cryptographically signed dynamic metadata information that uniquely represents each GCP service account.

You can configure the authentication via `GcpComputeAuthenticationOptions`.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        GcpComputeAuthenticationOptions options = GcpComputeAuthenticationOptions
.builder()
                .role(…).build();

        GcpComputeAuthentication authentication = new GcpComputeAuthentication
(options,
                restOperations());
    }

    // …
}
```

See also:

- Vault Documentation: Using the GCP auth backend
- GCP Documentation: Verifying the Identity of Instances

15.9. GCP-IAM authentication

The gcp auth backend allows Vault login by using existing GCP (Google Cloud Platform) IAM and GCE credentials.

GCP IAM authentication creates a signature in the form of a JSON Web Token (JWT) for a service account. A JWT for a service account is obtained by calling GCP IAM's `projects.serviceAccounts.signJwt` API. The caller authenticates against GCP IAM and proves thereby its identity. This Vault backend treats GCP as a Trusted Third Party.

IAM credentials can be obtained from either the runtime environment or supplied externally as e.g.

JSON. JSON is the preferred form as it carries the project id and service account identifier required for calling `projects.serviceAccounts.signJwt`.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        GcpIamAuthenticationOptions options = GcpIamAuthenticationOptions.builder
()
                .role(…).credential(GoogleCredentials.getApplicationDefault())
.build();

        GcpIamAuthentication authentication = new GcpIamAuthentication(options,
                restOperations());
    }

    // …
}
```

`GcpIamAuthenticationOptions` requires the Google Cloud Java SDK dependency (`com.google.apis:google-api-services-iam` and `com.google.auth:google-auth-library-oauth2-http`) as the authentication implementation uses Google APIs for credentials and JWT signing.

You can configure the authentication via `GcpIamAuthenticationOptions`.

> Google credentials require an OAuth 2 token maintaining the token lifecycle. All API is synchronous therefore, `GcpIamAuthentication` does not support `AuthenticationSteps` which is required for reactive usage.

See also:

- Vault Documentation: Using the GCP auth backend
- GCP Documentation: projects.serviceAccounts.signJwt

15.10. PCF authentication

The pcf auth backend allows Vault login for PCF instances. It leverages PCF's App and Container Identity Assurance.

PCF authentication uses the instance key and certificate to create a signature that is validated by Vault. If the signature matches, and potentially bound organization/space/application Id's match, Vault issues an appropriately-scoped token.

Instance credentials are available from files at `CF_INSTANCE_CERT` and `CF_INSTANCE_KEY` variables.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        PcfAuthenticationOptions options = PcfAuthenticationOptions.builder()
                .role(…).build();

        PcfAuthentication authentication = new PcfAuthentication(options,
                restOperations());
    }

    // …
}
```

`PcfAuthenticationOptions` requires the BouncyCastle library for creating RSA-PSS signatures.

You can configure the authentication via `PcfAuthenticationOptions`.

See also:

- Vault Documentation: Using the PCF auth backend

15.11. TLS certificate authentication

The `cert` auth backend allows authentication using SSL/TLS client certificates that are either signed by a CA or self-signed.

To enable `cert` authentication you need to:

1. Use SSL, see Vault Client SSL configuration
2. Configure a Java `Keystore` that contains the client certificate and the private key

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        ClientCertificateAuthenticationOptions options =
ClientCertificateAuthenticationOptions.builder()
                .path(…).build();

        return new ClientCertificateAuthentication(options, restOperations());
    }

    // …
}
```

See also: Vault Documentation: Using the Cert auth backend

15.12. Cubbyhole authentication

Cubbyhole authentication uses Vault primitives to provide a secured authentication workflow. Cubbyhole authentication uses tokens as primary login method. An ephemeral token is used to obtain a second, login VaultToken from Vault's Cubbyhole secret backend. The login token is usually longer-lived and used to interact with Vault. The login token can be retrieved either from a wrapped response or from the `data` section.

Creating a wrapped token

Response Wrapping for token creation requires Vault 0.6.0 or higher.

Example 28. Crating and storing tokens

```
$ vault token-create -wrap-ttl="10m"
Key                             Value
---                             -----
wrapping_token:                 397ccb93-ff6c-b17b-9389-380b01ca2645
wrapping_token_ttl:             0h10m0s
wrapping_token_creation_time:   2016-09-18 20:29:48.652957077 +0200 CEST
wrapped_accessor:               46b6aebb-187f-932a-26d7-4f3d86a68319
```

Example 29. Wrapped token response usage

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ...

    @Override
    public ClientAuthentication clientAuthentication() {

        CubbyholeAuthenticationOptions options = CubbyholeAuthenticationOptions
                .builder()
                .initialToken(VaultToken.of("..."))
                .wrapped()
                .build();

        return new CubbyholeAuthentication(options, restOperations());
    }

    // ...
}
```

Using stored tokens

Example 30. Crating and storing tokens

```
$ vault token create
Key                     Value
---                     -----
token                   f9e30681-d46a-cdaf-aaa0-2ae0a9ad0819
token_accessor          4eee9bd9-81bb-06d6-af01-723c54a72148
token_duration          0s
token_renewable         false
token_policies          [root]

$ vault token create -use-limit=2 -orphan -no-default-policy -policy=none
Key                     Value
---                     -----
token                   895cb88b-aef4-0e33-ba65-d50007290780
token_accessor          e84b661c-8aa8-2286-b788-f258f30c8325
token_duration          0s
token_renewable         false
token_policies          [none]

$ export VAULT_TOKEN=895cb88b-aef4-0e33-ba65-d50007290780
$ vault write cubbyhole/token token=f9e30681-d46a-cdaf-aaa0-2ae0a9ad0819
```

Example 31. Stored token response usage

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ...

    @Override
    public ClientAuthentication clientAuthentication() {

        CubbyholeAuthenticationOptions options = CubbyholeAuthenticationOptions
                .builder()
                .initialToken(VaultToken.of("..."))
                .path("cubbyhole/token")
                .build();

        return new CubbyholeAuthentication(options, restOperations());
    }

    // ...
}
```

Remaining TTL/Renewability

Tokens retrieved from Cubbyhole associated with a non-zero TTL start their TTL at the time of token creation. That time is not necessarily identical with application startup. To compensate for the initial delay, Cubbyhole authentication performs a self lookup for tokens associated with a non-zero TTL to retrieve the remaining TTL. Cubbyhole authentication will not self-lookup wrapped tokens without a TTL because a zero TTL indicates there is no TTL associated.

Non-wrapped tokens do not provide details regarding renewability and TTL by just retrieving the token. A self-lookup will lookup renewability and the remaining TTL.

See also:

- Vault Documentation: Tokens
- Vault Documentation: Cubbyhole Secret Backend
- Vault Documentation: Response Wrapping

15.13. Kubernetes authentication

Vault supports since 0.8.3 kubernetes-based authentication using Kubernetes tokens.

Using Kubernetes authentication requires a Kubernetes Service Account Token, usually mounted at `/var/run/secrets/kubernetes.io/serviceaccount/token`. The file contains the token which is read and sent to Vault. Vault verifies its validity using Kubernetes' API during login.

Configuring Kubernetes authentication requires at least the role name to be provided:

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        KubernetesAuthenticationOptions options = KubernetesAuthenticationOptions
.builder()
                .role(…).jwtSupplier(…).build();

        return new KubernetesAuthentication(options, restOperations());
    }

    // …
}
```

You can configure the authentication via `KubernetesAuthenticationOptions`.

See also:

- Vault Documentation: Using the Kubernetes auth backend
- Kubernetes Documentation: Configure Service Accounts for Pods

15.14. Authentication Steps

`ClientAuthentication` objects describe the authentication flow and perform the actual authentication steps. Pre-composed authentications are easy to use and to configure with a tight binding to synchronous execution.

The composition of authentication methods and reusing common steps, such as posting login payload to Vault or retrieving authentication input from an HTTP source is not intended with `ClientAuthentication` objects.

Authentication steps provide reusability of common authentication activity. Steps created via `AuthenticationSteps` describe an authentication flow in a functional style leaving the actual authentication execution to specific executors.

Example 32. Stored token authentication flow.

```
AuthenticationSteps.just(VaultToken.of(…));                    ①
```

① Creates `AuthenticationSteps` from just a `VaultToken`.

A single-step authentication flow can be created from a single input. Flows declaring multiple authentication steps start with a `Supplier` or `HttpRequest` that provide an authentication state object which can be used to map or post to Vault for login.

Example 33. AppRole authentication flow

```
AuthenticationSteps.fromSupplier(                              ①

    () -> getAppRoleLogin(options.getRoleId(), options.getSecretId()))  ②

    .login("auth/{mount}/login", options.getPath());           ③
```

① Start declaring `AuthenticationSteps` accepting a `Supplier<T>`. The state object type depends on the `Supplier` response type which can be mapped in a later step.

② The actual `Supplier` implementation. Creating a `Map` in this case.

③ Perform a Vault login by posting the state object (`Map`) to a Vault endpoint for Vault token creation. Note that template variables are subject to URL escaping.

Authentication flows require an executor to perform the actual login. We provide two executors for different execution models:

- `AuthenticationStepsExecutor` as a drop-in replacement for synchronous `ClientAuthentication`.
- `AuthenticationStepsOperator` for reactive execution.

Many `ClientAuthentication`'s come with static factory methods to create `AuthenticationSteps` for their authentication-specific options:

Example 34. Synchronous `AuthenticationSteps` execution

```
CubbyholeAuthenticationOptions options = ⋯
RestOperations restOperations = ⋯

AuthenticationSteps steps = CubbyholeAuthentication.createAuthenticationSteps
(options);

AuthenticationStepsExecutor executor = new AuthenticationStepsExecutor(steps,
restOperations);

VaultToken token = executor.login();
```

15.15. Token Lifecycle

Vault's tokens can be associated with a time to live. Tokens obtained by an authentication method are intended to be used as long as the session is active and should not expire while the application is active.

Spring Vault provides with `LifecycleAwareSessionManager` a session manager that can renew the token until it reaches its terminal TTL to then perform another login to obtain the next token which is associated with the session.

Depending on the authentication method, a login can create two kinds of tokens:

- `VaultToken`: Generic token encapsulating the actual token.
- `LoginToken`: Token associated with renewability/TTL.

Authentication methods such as `TokenAuthentication` just create a `VaultToken` which does not carry any renewability/TTL details. `LifecycleAwareSessionManager` will run a self-lookup on the token to retrieve renewability and TTL from Vault. `VaultToken` are renewed periodically if self-lookup is enabled. Note that `VaultToken` are never revoked, only `LoginToken` are revoked.

Authentication methods creating `LoginToken` directly (all login-based authentication methods) already provide all necessary details to setup token renewal. Tokens obtained from a login are revoked by `LifecycleAwareSessionManager` if the session manager is shut down.

Chapter 16. Miscellaneous

Learn in this chapter about details worth mentioning like the Spring Security integration.

16.1. Spring Security

Spring Vault integrates with Spring Security by providing implementations for `BytesKeyGenerator` and `BytesEncryptor`. Both implementations use Vault's `transit` backend.

Example 35. `VaultBytesKeyGenerator` example

```
VaultOperations operations = ⋯;
VaultBytesKeyGenerator generator = new VaultBytesKeyGenerator(operations);

byte[] key = generator.generateKey();
```

Example 36. `VaultBytesEncryptor` example

```
VaultTransitOperations transit = ⋯;

VaultBytesEncryptor encryptor = new VaultBytesEncryptor(transit, "my-key-name");

byte[] ciphertext = encryptor.encrypt(plaintext);

byte[] result = encryptor.decrypt(ciphertext);
```

Vault encapsulates an entropy source that is decoupled from your JVM along with server-side key-management. This relieves the burden of proper encryption/decryption from application developers and pushes the burden onto the operators of Vault. Operators of Vault commonly include the security team at an organization, which means they can ensure that data is encrypted/decrypted properly. Additionally, since encrypt/decrypt operations must enter the audit log, any decryption event is recorded.

The backend also supports key rotation, which allows a new version of the named key to be generated. All data encrypted with the key will use the newest version of the key; previously encrypted data can be decrypted using old versions of the key. Administrators can control which previous versions of a key are available for decryption, to prevent an attacker gaining an old copy of ciphertext to be able to successfully decrypt it.

Vault is after all a networked service that incurs each operation with a latency. Components heavily using encryption or random bytes generation may experience a difference in throughput and performance.