Spring Vault - Reference Documentation

Mark Paluch

Version 1.0.0.RELEASE, 2017-04-10

Table of Contents

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 the 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 1.x binaries requires JDK level 6.0 and above, and Spring Framework 4.3.7.RELEASE and above. While we maintain Java 6 compatibility, new projects should consider using Java 8 .

In terms of Vault, Vault at least 0.5.

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 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>1.0.0.RELEASE</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

The current version is `1.0.0.RELEASE`. The version name follows the following pattern: `${version}-${release}` where release can be one of the following:

- `BUILD-SNAPSHOT` - current snapshots
- `M1`, `M2` etc. - milestones
- `RC1`, `RC2` etc. - release candidates
- `RELEASE` - GA release
- `SR1`, `SR2` etc. - service releases

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 4.3.7.RELEASE 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 0.5 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. Using the Spring Vault BOM

```
<dependencies>

    <!-- other dependency elements omitted -->

    <dependency>
        <groupId>org.springframework.vault</groupId>
        <artifactId>spring-vault-core</artifactId>
        <version>1.0.0.RELEASE</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>http://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>http://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.

8.1. 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.

NOTE	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.

NOTE	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.

8.1.1. 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.

NOTE 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`.

8.1.2. 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`.

8.2. 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 = new SslConfiguration(               ①
                new FileSystemResource("client-cert.jks"), "changeit",
                new FileSystemResource("truststore.jks"), "changeit");

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

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

① Full configuration.

② Configuring only trust store settings.

③ Configuring only key store settings.

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

8.3. 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)
  - Truststore resource: `vault.ssl.trust-store` (optional)
  - Truststore password: `vault.ssl.trust-store-password` (optional)
- Authentication method: `vault.authentication` (defaults to `TOKEN`, supported authentication methods are: `TOKEN`, `APPID`, `APPROLE`, `AWS_EC2`, `CERT`, `CUBBYHOLE`)

Authentication-specific property keys

Token authentication

- Vault Token: `vault.token`

AppId authentication

- 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

- RoleId: `vault.app-role.role-id`
- SecretId: `vault.app-role.secret-id` (optional)

AWS-EC2 authentication

- RoleId: `vault.aws-ec2.role-id`

- Identity Document URL: `vault.aws-ec2.identity-document` (optional)

TLS certificate authentication

No configuration options.

Cubbyhole authentication

- Initial Vault Token: `vault.token`

8.4. 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.

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

8.4.1. 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.

8.4.2. @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 9. Properties stored in Vault

```
{
  // ⋯

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

  // ⋯
}
```

Example 10. 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 11. 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
}
```

NOTE	Secrets obtained from `generic` secret backends are associated with a TTL (`refresh_interval`) but not a lease Id. Spring Vault's `PropertySource` is not refreshing/flushing these secrets once the TTL expires despite the requested `Renewal` mode.

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.

8.5. 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 `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 9. Client support

Spring Vault supports a 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 clients:

- Java's builtin `HttpURLConnection` (default client)
- Apache Http Components
- Netty
- OkHttp 2
- OkHttp 3

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.

9.1. 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.

9.2. 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 12. Apache Http Components Dependency

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

Example 13. Netty Dependency

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

Example 14. Square OkHttp 2

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

Example 15. Square OkHttp 3

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

Chapter 10. 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.

10.1. 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 16. 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"
));
    }
}
```

NOTE 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.

10.2. Token authentication

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

NOTE	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.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // ···

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

    // ···
}
```

See also:

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

10.3. AppId authentication

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.

```
@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
```

NOTE 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).

```
@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
```

NOTE 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.

10.3.1. 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

10.4. 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 (push or pull mode).

RoleId and optionally SecretId must be provided to `AppRoleAuthenticationOptions`, Spring Vault will not look up these or create a custom SecretId.

```
@Configuration
class AppConfig extends AbstractVaultConfiguration {

    // …

    @Override
    public ClientAuthentication clientAuthentication() {

        AppRoleAuthenticationOptions options = AppRoleAuthenticationOptions
.builder()
                .roleId("…")
                .secretId("…")
                .build();

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

    // …
}
```

See also: Vault Documentation: Using the AppRole auth backend

10.5. 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

10.6. 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() {
        return new ClientCertificateAuthentication(options, restOperations());
    }

    // …
}
```

See also: Vault Documentation: Using the Cert auth backend

10.7. 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

NOTE	Response Wrapping for token creation requires Vault 0.6.0 or higher.

Example 17. 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 18. 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 19. 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]

$ 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 20. 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());
    }

    // ···
}
```

See also:

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