Quarkus - Working with HashiCorp Vault

HashiCorp Vault is a multi-purpose tool aiming at protecting sensitive data, such as credentials, certificates, access tokens, encryption keys, … In the context of Quarkus, it is being used for 3 primary use cases:

- mounting a map of properties stored into the Vault kv secret engine as an Eclipse MicroProfile config source

- fetch credentials from Vault when configuring an Agroal datasource

- access the Vault *kv secret engine* programmatically

Under the hood, the Quarkus Vault extension takes care of authentication when negotiating a client Vault token plus any transparent token or lease renewals according to ttl and max-ttl.

> **ℹ** This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

Prerequisites

To complete this guide, you need:

- roughly 20 minutes

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.5.3+

- Docker installed

Starting Vault

Let's start Vault in development mode:

```
docker run --rm --cap-add=IPC_LOCK -e
VAULT_ADDR=http://localhost:8200 -p 8200:8200 -d --name=dev-vault
vault:1.2.2
```

You can check that vault is running with:

```
docker logs dev-vault
```

You should see:

```
==> Vault server configuration:

             Api Address: http://0.0.0.0:8200
                     Cgo: disabled
         Cluster Address: https://0.0.0.0:8201
              Listener 1: tcp (addr: "0.0.0.0:8200", cluster
address: "0.0.0.0:8201", max_request_duration: "1m30s",
max_request_size: "33554432", tls: "disabled")
               Log Level: info
                   Mlock: supported: true, enabled: false
                 Storage: inmem
                 Version: Vault v1.2.2

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

You may need to set the following environment variable:

    $ export VAULT_ADDR='http://0.0.0.0:8200'

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

Unseal Key: 0lZ2/vzpa92pH8gersSn2h9b5tmzd4m5sqIdMC/4PDs=
Root Token: s.5VUS8pte13RqekCB2fmMT3u2

Development mode should NOT be used in production installations!

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

In development mode, Vault gets configured with several options that makes it convenient:

- Vault is already initialized with one key share (whereas in normal mode this has to be done explicitly and the number of key shares is 5 by default)

- the unseal key and the root token are displayed in the logs (please write down the root token, we will need it in the following step)

- Vault is unsealed

- in-memory storage

- TLS is disabled

- a *kv secret engine v2* is mounted at `secret/`

In the following step, we are going to add a `userpass` authentication that we will use from the Quarkus application, to access a secret stored in the *kv secret engine*.

First open a shell inside the vault container:

```
docker exec -it dev-vault sh
```

Set the `VAULT_TOKEN` with the value that was printed in the logs:

```
export VAULT_TOKEN=s.5VUS8pte13RqekCB2fmMT3u2
```

You can check Vault's status using the CLI command `vault status`:

```
Key             Value
---             -----
Seal Type       shamir
Initialized     true
Sealed          false
Total Shares    1
Threshold       1
Version         1.2.2
Cluster Name    vault-cluster-b07e80d8
Cluster ID      55bd74b6-eaaf-3862-f7ce-3473ab86c57f
HA Enabled      false
```

For simplicity reasons, we are going to use a *kv secret engine version 1* (which is the default) mounted at path `secret` instead of the pre-configured *kv version 2* available in dev mode. So let's disable the current kv engine, and recreate a new one:

```
# this will disable the current kv version 2 secret engine mounted
at path 'secret'
vault secrets disable secret

# this will enable a new kv engine version 1 at path 'secret'
vault secrets enable -path=secret kv
```

Now let's add a secret configuration for our application:

```
vault kv put secret/myapps/vault-quickstart/config a-private-
key=123456
```

We have defined a path `secret/myapps/vault-quickstart` in Vault that we need to give access to from the Quarkus application.

Create a policy that gives read access to `secret/myapps/vault-quickstart` and subpaths:

```
cat <<EOF | vault policy write vault-quickstart-policy -
path "secret/myapps/vault-quickstart/*" {
  capabilities = ["read"]
}
EOF
```

ℹ️ When using a *kv secret engine version 2*, secrets are written and fetched at path `<mount>/`**`data`**`/<secret-path>` as opposed to `<mount>/<secret-path>` in a *kv secret engine version 1*. It does not change any of the CLI commands (i.e. you do not specify `data` in your path). However it does change the policies, since capabilities are applied to the real path. In the example above, the path should be `secret/`**`data`**`/myapps/vault-quickstart/*` if you were working with a *kv secret engine version 2*.

And finally, let's enable the *userpass auth secret engine*, and create user `bob` with access to the `vault-quickstart-policy`:

```
vault auth enable userpass
vault write auth/userpass/users/bob password=sinclair
policies=vault-quickstart-policy
```

ℹ️ The Vault extension also supports alternate authentication methods such as:

- approle
- kubernetes when deploying the Quarkus application and Vault into Kubernetes

It is also possible to directly pass a `quarkus.vault.authentication.client-token`, which will bypass the authentication process. This could be handy in development where revocation and ttl are not a concern.

Check the extension configuration documentation for more information.

To check that the configuration is correct, try logging in:

```
vault login —method=userpass username=bob password=sinclair
```

You should see:

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

Key                    Value
---                    -----
token                  s.s93BVzJPzBiIGuYJHBTkG8Uw
token_accessor         OKNipTAgxWbxsrjixASNiwdY
token_duration         768h
token_renewable        true
token_policies         ["default" "vault-quickstart-policy"]
identity_policies      []
policies               ["default" "vault-quickstart-policy"]
token_meta_username    bob
```

Now set `VAULT_TOKEN` to the `token` above (instead of the root token), and try reading the vault-quickstart secret config:

```
export VAULT_TOKEN=s.s93BVzJPzBiIGuYJHBTkG8Uw
vault kv get secret/myapps/vault-quickstart/config
```

You should see:

```
======== Data ========
Key              Value
---              -----
a-private-key    123456
```

Create a quarkus application with a secret configuration

```
mvn io.quarkus:quarkus-maven-plugin:1.2.0.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=vault-quickstart \
    -DclassName="org.acme.quickstart.GreetingResource" \
    -Dpath="/hello" \
    -Dextensions="vault,hibernate-orm,jdbc-postgresql"
cd vault-quickstart
```

Configure access to vault from the `application.properties`:

```
# vault url
quarkus.vault.url=http://localhost:8200

# vault authentication
quarkus.vault.authentication.userpass.username=bob
quarkus.vault.authentication.userpass.password=sinclair

# path within the kv secret engine where is located the vault-
quickstart secret configuration
quarkus.vault.secret-config-kv-path=myapps/vault-quickstart/config
```

This should mount whatever keys are stored in `secret/myapps/vault-quickstart` as MicroProfile config properties.

Let's verify that by adding a new endpoint in GreetingResource:

```
@Path("/hello")
public class GreetingResource {

    @ConfigProperty(name = "a-private-key")
    String privateKey;

    ...

    @GET
    @Path("/private-key")
    @Produces(MediaType.TEXT_PLAIN)
    public String privateKey() {
        return privateKey;
    }
}
```

Now compile the application and run it:

```
./mvnw clean install
java -jar target/vault-quickstart-1.0-SNAPSHOT-runner.jar
```

Finally test the new endpoint:

```
curl http://localhost:8080/hello/private-key
```

You should see:

```
123456
```

Fetching credentials from Vault with Hibernate ORM

Start a PostgreSQL database:

```
docker run --ulimit memlock=-1:-1 -it --rm=true --memory
-swappiness=0 --name postgres-quarkus-hibernate -e
POSTGRES_USER=sarah -e POSTGRES_PASSWORD=connor -e
POSTGRES_DB=mydatabase -p 5432:5432 postgres:10.5
```

Create a simple service:

```java
@ApplicationScoped
public class SantaClausService {

    @Inject
    EntityManager em;

    @Transactional
    public List<Gift> getGifts() {
        return (List<Gift>) em.createQuery("select g from Gift g")
.getResultList();
    }
}
```

With its `Gift` entity:

```java
@Entity
public class Gift {

    private Long id;
    private String name;

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator=
"giftSeq")
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Finally, add a new endpoint in `GreetingResource`:

```java
@Inject
SantaClausService santaClausService;

@GET
@Path("/gift-count")
@Produces(MediaType.TEXT_PLAIN)
public int geGiftCount() {
    return santaClausService.getGifts().size();
}
```

Create a new path in Vault where the database password will be added:

```
# set the root token again
export VAULT_TOKEN=s.5VUS8pte13RqekCB2fmMT3u2
vault kv put secret/myapps/vault-quickstart/db password=connor
```

Since we allowed read access on `secret/myapps/vault-quickstart/` subpaths in the policy, there is nothing else we have to do to allow the application to read it.

When fetching credentials from Vault that are intended to be used by the Agroal connection pool, we need first to create a named Vault credentials provider in the application.properties:

```
quarkus.vault.credentials-provider.mydatabase.kv-path=myapps/vault-
quickstart/db
```

This defines a credentials provider `mydatabase` that will fetch the password from key `password` at path `myapps/vault-quickstart/db`.

The credentials provider can now be used in the datasource configuration, in place of the `password` property:

```
# configure your datasource
quarkus.datasource.url =
jdbc:postgresql://localhost:5432/mydatabase
quarkus.datasource.driver = org.postgresql.Driver
quarkus.datasource.username = sarah
#quarkus.datasource.password = connor
quarkus.datasource.credentials-provider = mydatabase

# drop and create the database at startup (use `update` to only
update the schema)
quarkus.hibernate-orm.database.generation=drop-and-create
```

Another way to specify the datasource password is with property indirection. Assuming that vault path `myapps/vault-quickstart/config` contains key `my-db-password`, all is required on the datasource configuration is:

```
quarkus.datasource.username = sarah
quarkus.datasource.password = ${my-db-password}
```

The only drawback is that the password will never be fetched again from vault after the initial property loading. This means that if the db password was changed while running, the application would have to be restarted after vault has been updated with the new password. This contrasts with the credentials provider approach, which fetches the password from vault everytime a connection creation is attempted.

Restart the application after rebuilding it, and test it with the new endpoint:

```
curl http://localhost:8080/hello/gift-count
```

You should see:

```
0
```

> ℹ️ The Vault extension also supports using [dynamic database credentials](#) through the `database-credentials-role` property on the `credentials-provider`. The Vault setup is more involved and goes beyond the scope of that quickstart guide.

Programmatic access to the KV secret engine

Sometimes secrets are retrieved from an arbitrary path that is known only at runtime through an application specific property, or a method argument for instance. In that case it is possible to inject a Quarkus VaultKVSecretEngine, and retrieve secrets programmatically.

For instance, in `GreetingResource`, add:

```java
@Inject
VaultKVSecretEngine kvSecretEngine;

...

@GET
@Path("/secrets/{vault-path}")
@Produces(MediaType.TEXT_PLAIN)
public String getSecrets(@PathParam("vault-path") String vaultPath)
{
    return kvSecretEngine.readSecret("myapps/vault-quickstart/" +
vaultPath).toString();
}
```

Restart the application after rebuilding it, and test it with the new endpoint:

```
curl http://localhost:8080/hello/secrets/db
```

You should see:

```
{password=connor}
```

TLS

In production mode, TLS should be activated between the Quarkus application and Vault to prevent *man-in-the-middle* attacks.

There are several ways to configure the Quarkus application:

- through the standard `javax.net.ssl.trustStore` system property, which should refer to a JKS truststore containing the trusted certificates

- using property `quarkus.vault.tls.ca-cert`, which should refer to a pem encoded file.

If `quarkus.vault.tls.ca-cert` is not set and the Quarkus application is using the Kubernetes authentication, TLS will be active and use the CA certificate bundle located in `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`. If you want to disable this behavior (for instance when using a trust store), you need to set it explicitly using: `quarkus.vault.tls.use-kubernetes-ca-cert=false`.

The last relevant property is `quarkus.vault.tls.skip-verify`, which allows to communicate with Vault using TLS, but without checking the certificate authenticity. This may be convenient in development, but is strongly discouraged in production as it is not more secure than talking to Vault in plain HTTP.

Conclusion

As a general matter, you should consider reading the extensive Vault documentation and apply best practices.

The features exposed today through the Vault extension covers only a small fraction of what the product is capable of. Still, it addresses already some of the most common microservices scenarii (e.g. sensitive configuration and database credentials), which goes a long way towards creating secured applications.

Configuration Reference

▤ Configuration property fixed at build time - ⚙ Configuration property overridable at runtime

Configuration property	Type	Default
⚙ `quarkus.vault.url` Vault server url. Example: https://localhost:8200	URL	

⚙ `quarkus.vault.renew-grace-period` Renew grace period duration. This value if used to extend a lease before it expires its ttl, or recreate a new lease before the current lease reaches its max_ttl. By default Vault leaseDuration is equal to 7 days (ie: 168h or 604800s). If a connection pool maxLifetime is set, it is reasonable to set the renewGracePeriod to be greater than the maxLifetime, so that we are sure we get a chance to renew leases before we reach the ttl. In any case you need to make sure there will be attempts to fetch secrets within the renewGracePeriod, because that is when the renewals will happen. This is particularly important for db dynamic secrets because if the lease reaches its ttl or max_ttl, the password of the db user will become invalid and it will be not longer possible to log in. This value should also be smaller than the ttl, otherwise that would mean that we would try to recreate leases all the time.	Duration ❓	1H
⚙ `quarkus.vault.secret-config-cache-period` Vault config source cache period. Properties fetched from vault as MP config will be kept in a cache, and will not be fetched from vault again until the expiration of that period. This property is ignored if `secret-config-kv-path` is not set.	Duration ❓	10M
⚙ `quarkus.vault.secret-config-kv-path` List of comma separated vault paths in kv store, where all properties will be available as MP config properties **as-is**, with no prefix. For instance, if vault contains property `foo`, it will be made available to the quarkus application as `@ConfigProperty(name = "foo") String foo;` If 2 paths contain the same property, the last path will win. For instance if • `secret/base-config` contains `foo=bar` and • `secret/myapp/config` contains `foo=myappbar`, then `@ConfigProperty(name = "foo") String foo` will have value `myappbar` with application properties `quarkus.vault.secret-config-kv-path=base-config,myapp/config`	list of string	

⚙ `quarkus.vault.log-confidentiality-level` Used to hide confidential infos, for logging in particular. Possible values are: • low: display all secrets. • medium: display only usernames and lease ids (ie: passwords and tokens are masked). • high: hide lease ids and dynamic credentials username.	`low`, `medium`, `high`	`medium`
⚙ `quarkus.vault.kv-secret-engine-version` Kv secret engine version. see https://www.vaultproject.io/docs/secrets/kv/index.html	int	`1`
⚙ `quarkus.vault.kv-secret-engine-mount-path` Kv secret engine path. see https://www.vaultproject.io/docs/secrets/kv/index.html	string	`secret`
⚙ `quarkus.vault.connect-timeout` Timeout to establish a connection with Vault.	Duration ❓	`5S`
⚙ `quarkus.vault.read-timeout` Request timeout on Vault.	Duration ❓	`1S`
⚙ `quarkus.vault.secret-config-kv-path."prefix"` List of comma separated vault paths in kv store, where all properties will be available as **prefixed** MP config properties. For instance if the application properties contains `quarkus.vault.secret-config-kv-path.myprefix=config`, and vault path `secret/config` contains `foo=bar`, then `myprefix.foo` will be available in the MP config. If the same property is available in 2 different paths for the same prefix, the last one will win.	`Map<String,List<String>>`	required ❗

	Type	Default
⚙ quarkus.vault.credentials-provider."credentials-provider".database-credentials-role Database credentials role, as defined by https://www.vaultproject.io/docs/secrets/databases/index.html One of `database-credentials-role` or `kv-path` needs to be defined. not both.	string	
⚙ quarkus.vault.credentials-provider."credentials-provider".kv-path A path in vault kv store, where we will find the kv-key. One of `database-credentials-role` or `kv-path` needs to be defined. not both. see https://www.vaultproject.io/docs/secrets/kv/index.html	string	
⚙ quarkus.vault.credentials-provider."credentials-provider".kv-key Key name to search in vault path `kv-path`. The value for that key is the credential. `kv-key` should not be defined if `kv-path` is not. see https://www.vaultproject.io/docs/secrets/kv/index.html	string	`password`
Authentication	**Type**	**Default**
⚙ quarkus.vault.authentication.client-token Vault token, bypassing Vault authentication (kubernetes, userpass or approle). This is useful in development where an authentication mode might not have been set up. In production we will usually prefer some authentication such as userpass, or preferably kubernetes, where Vault tokens get generated with a TTL and some ability to revoke them.	string	
⚙ quarkus.vault.authentication.app-role.role-id Role Id for AppRole auth method. This property is required when selecting the app-role authentication type.	string	
⚙ quarkus.vault.authentication.app-role.secret-id Secret Id for AppRole auth method. This property is required when selecting the app-role authentication type.	string	

	Type	Default
⚙ `quarkus.vault.authentication.userpass.username` User for userpass auth method. This property is required when selecting the userpass authentication type.	string	
⚙ `quarkus.vault.authentication.userpass.password` Password for userpass auth method. This property is required when selecting the userpass authentication type.	string	
⚙ `quarkus.vault.authentication.kubernetes.role` Kubernetes authentication role that has been created in Vault to associate Vault policies, with Kubernetes service accounts and/or Kubernetes namespaces. This property is required when selecting the Kubernetes authentication type.	string	
⚙ `quarkus.vault.authentication.kubernetes.jwt-token-path` Location of the file containing the Kubernetes JWT token to authenticate against in Kubernetes authentication mode.	string	`/var/run/secrets/kubernetes.io/serviceaccount/token`
TLS	**Type**	**Default**
⚙ `quarkus.vault.tls.skip-verify` Allows to bypass certificate validation on TLS communications. If true this will allow TLS communications with Vault, without checking the validity of the certificate presented by Vault. This is discouraged in production because it allows man in the middle type of attacks.	boolean	`false`
⚙ `quarkus.vault.tls.ca-cert` Certificate bundle used to validate TLS communications with Vault. The path to a pem bundle file, if TLS is required, and trusted certificates are not set through javax.net.ssl.trustStore system property.	string	

	Type	Default
⚙ `quarkus.vault.tls.use-kubernetes-ca-cert` If true and Vault authentication type is kubernetes, TLS will be active and the cacert path will be set to /var/run/secrets/kubernetes.io/serviceaccount/ca.crt. If set, this setting will take precedence over property quarkus.vault.tls.ca-cert. This means that if Vault authentication type is kubernetes and we want to use quarkus.vault.tls.ca-cert or system property javax.net.ssl.trustStore, then this property should be set to false.	boolean	`true`
Transit Engine	**Type**	**Default**

⚙ `quarkus.vault.transit.key."key".name`
 	
Specifies the name of the key to use. By default this will be the property key alias. Used when the same transit key is used with different configurations. Such as in:
 	
<pre>quarkus.vault.transit.key.my-foo-key.name=foo
 	
quarkus.vault.transit.key.my-foo-key-with-
prehashed.name=foo
quarkus.vault.transit.key.my-foo-key-with-
prehashed.prehashed=true
...
transitSecretEngine.sign("my-foo-key", "my raw
content");
or</pre>transitSecretEngine.sign("my-foo-key-with-prehashed", "my already hashed content");
	string	

⚙ `quarkus.vault.transit.key."key".prehashed` Set to true when the input is already hashed. Applies to sign operations.	boolean	
⚙ `quarkus.vault.transit.key."key".signature-algorithm` When using a RSA key, specifies the RSA signature algorithm. Applies to sign operations.	string	
⚙ `quarkus.vault.transit.key."key".hash-algorithm` Specifies the hash algorithm to use for supporting key types. Applies to sign operations.	string	

⚙ `quarkus.vault.transit.key."key".type`		
Specifies the type of key to create for the encrypt operation. Applies to encrypt operations.	string	
⚙ `quarkus.vault.transit.key."key".convergent-encryption`		
If enabled, the key will support convergent encryption, where the same plaintext creates the same ciphertext. Applies to encrypt operations.	string	

ℹ

About the Duration format

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.