Quarkus - Using 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, several use cases are supported:

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

- fetching credentials from Vault when configuring an Agroal datasource, as documented in the Vault Datasource Guide

- accessing Vault *kv secret engine* programmatically

- support for the TOTP Secret Engine

- support for the Transit Secret Engine as documented in the Vault Transit Secret Engine Guide

- support for several authentication methods as documented in the Vault Authentication guide

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*.

> 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 3.6.3

- 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/`

> ℹ️ By default Quarkus assumes that a *kv secret engine* in version **2** mounted at path `secret/` will be used. If that is not the case, please use properties `quarkus.vault.kv-secret-engine-version` and `quarkus.vault.kv-secret-engine-mount-path` accordingly.

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
```

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/data/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 is `secret/data/myapps/vault-quickstart/*` since we are working with a *kv secret engine version 2*. It would be `secret/myapps/vault-quickstart/*` with a *kv secret engine version 1*.

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 Quarkus supports additional Vault Authentication methods. Check the Vault Authentication guide for details.

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:1.7.2.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=vault-quickstart \
    -DclassName="org.acme.quickstart.GreetingResource" \
    -Dpath="/hello" \
    -Dextensions="vault"
cd vault-quickstart
```

If you already have your Quarkus project configured, you can add the vault extension to your project

by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="vault"
```

This will add the following to your `pom.xml`:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-vault</artifactId>
</dependency>
```

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:

```java
@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
```

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:

```
@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();
}
```

Add a new key in Vault:

```
vault kv put secret/myapps/vault-quickstart/private mysecret=abc
```

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

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

You should see:

```
{mysecret=abc}
```

TOTP Secrets Engine

The Vault TOTP secrets engine generates time-based credentials according to the TOTP standard.

Vault TOTP supports both the generator scenario (like Google Authenticator) and the provider scenario (like the Google.com sign in).

The Vault extension integrates with the Vault TOTP secret engine by providing an `io.quarkus.vault.VaultTOTPSecretEngine` class.

```java
import io.quarkus.vault.VaultTOTPSecretEngine;
import io.quarkus.vault.secrets.totp.CreateKeyParameters;
import io.quarkus.vault.secrets.totp.KeyConfiguration;
import io.quarkus.vault.secrets.totp.KeyDefinition;

@Inject
VaultTOTPSecretEngine vaultTOTPSecretEngine;

CreateKeyParameters createKeyParameters = new
CreateKeyParameters("Google", "test@gmail.com");
createKeyParameters.setPeriod("30m");

/** Google Authentication logic */

final Optional<KeyDefinition> myKey = vaultTOTPSecretEngine
                                        .createKey("my_key_2",
createKeyParameters); ① ②

final String keyCode =
vaultTOTPSecretEngine.generateCode("my_key_2"); ③

/** Google Login logic */

boolean valid = vaultTOTPSecretEngine.validateCode("my_key_2",
keyCode); ④
```

① Create a key to generate codes.

② `KeyDefinition` class contains an embeddable base64 QR code that can be used by third-party code generators.

③ Generates a code (not using third-party generator).

④ Validates that the code is valid.

Vault Health Check

If you are using the `quarkus-smallrye-health` extension, `quarkus-vault` can add a readiness health check to validate the connection to the Vault server. This is disabled by default.

If enabled, when you access the `/health/ready` endpoint of your application you will have information about the connection validation status.

This behavior can be enabled by setting the `quarkus.vault.health.enabled` property to `true` in your `application.properties`.

Only if Vault is initialized, unsealed and active, the health endpoint returns that Vault is ready to serve requests.

You can change a bit this behaviour by using `quarkus.vault.health.stand-by-ok` and `quarkus.vault.health.performance-stand-by-ok` to `true` in your `application.properties`.

stand-by-ok

Specifies if being a standby should still return the active status code instead of the standby status code.

performance-stand-by-ok

Specifies if being a performance standby should still return the active status code instead of the performance standby status code.

You can inject `io.quarkus.vault.VaultSystemBackendEngine` to run system operations programmatically.

> ⚠️ When the readiness probe is failing in Kubernetes, then the application is not reachable. This means that if Vault is failing, all services depending on Vault will become unreachable and maybe this is not the desired state, so use this flag according to your requirements.

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.

Vault Provisioning

Beside the typical client use cases, the Quarkus extension can be used to provision Vault as well, for instance as part of a CD pipeline. Specific CDI beans support this scenario:

- `VaultSystemBackendEngine`: create Vault Policies. See the Vault documentation.

- `VaultKubernetesAuthService`. See the Vault documentation.

  - Configure the Kubernetes Auth Method (Kubernetes host, certificates, keys, …)

  - Create Kubernetes Auth Roles (association between Kubernetes service accounts, Kubernetes namespaces and Vault policies)

For instance:

```
@Inject
VaultSystemBackendEngine vaultSystemBackendEngine;

@Inject
VaultKubernetesAuthService vaultKubernetesAuthService;

...

  String rules = "path \"transit/*\" {\n" +
          "  capabilities = [ \"create\", \"read\", \"update\" ]\n"
+
          "}";
  String policyName = "sys-test-policy";
  vaultSystemBackendEngine.createUpdatePolicy(policyName, rules);

  String roleName = "test-auth-k8s";
  List<String> boundServiceAccountNames = asList("vault-auth");
  List<String> boundServiceAccountNamespaces = asList("default");
  List<String> tokenPolicies = asList(policyName);

  vaultKubernetesAuthService.createRole(roleName, new
VaultKubernetesAuthRole()
          .setBoundServiceAccountNames(boundServiceAccountNames)

.setBoundServiceAccountNamespaces(boundServiceAccountNamespaces)
          .setTokenPolicies(tokenPolicies));
```

Like any client, a provisioning program would have to authenticate using one of the supported Auth methods, and get the appropriate grants through one or more policies. Example:

```
# create Policies
path "sys/policy/*" {
  capabilities = ["read", "create", "update", "delete"]
}

# create Kubernetes Auth Roles
path "auth/kubernetes/role/*" {
  capabilities = ["read", "create", "update", "delete"]
}
```

You should adjust to the minimal set of access rights depending on your specific use case.

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 - All other configuration properties are overridable at runtime

Configuration property	Type	Default
`quarkus.vault.url` Vault server url. \<p\> Example: https://localhost:8200 \<p\> See also the documentation for the `kv-secret-engine-mount-path` property for some insights on how the full Vault url gets built.	URL	

`quarkus.vault.renew-grace-period` Renew grace period duration. <p> 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. <p> 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. <p> For instance, if vault contains property `foo`, it will be made available to the quarkus application as `@ConfigProperty(name = "foo") String foo;` <p> If 2 paths contain the same property, the last path will win. <p> For instance if <p> * `secret/base-config` contains foo=bar and * `secret/myapp/config` contains foo=myappbar, then <p> `@ConfigProperty(name = "foo") String  foo` will have value `myappbar` with application properties `quarkus.vault.secret-config-kv-path=base-config,myapp/config` <p> See also the documentation for the `kv-secret-engine-mount-path` property for some insights on how the full Vault url gets built.	list of string	
`quarkus.vault.log-confidentiality-level` Used to hide confidential infos, for logging in particular. Possible values are: <p> * 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. <p> see https://www.vaultproject.io/docs/secrets/kv/index.html	int	2

`quarkus.vault.kv-secret-engine-mount-path` KV secret engine path. \<p\> This value is used when building the url path in the KV secret engine programmatic access (i.e. `VaultKVSecretEngine`) and the vault config source (i.e. fetching configuration properties from Vault). \<p\> For a v2 KV secret engine (default - see `kv-secret-engine-version property`) the full url is built from the expression `<url>/v1/</kv-secret-engine-mount-path>/data/`…. \<p\> With property `quarkus.vault.url=https://localhost:8200`, the following call `vaultKVSecretEngine.readSecret("foo/bar")` would lead eventually to a `GET` on Vault with the following url: `https://localhost:8200/v1/secret/data/foo/bar`. \<p\> With a KV secret engine v1, the url changes to: `<url>/v1/</kv-secret-engine-mount-path>/`…. \<p\> The same logic is applied to the Vault config source. With `quarkus.vault.secret-config-kv-path=config/myapp` The secret properties would be fetched from Vault using a `GET` on `https://localhost:8200/v1/secret/data/config/myapp` for a KV secret engine v2 (or `https://localhost:8200/v1/secret/config/myapp` for a KV secret engine v1). \<p\> 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. \<p\> 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. \<p\> If the same property is available in 2 different paths for the same prefix, the last one will win. \<p\> See also the documentation for the `kv-secret-engine-mount-path` property for some insights on how the full Vault url gets built.	Map<String,List<String>>	required ❗
`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`
Health check configuration	**Type**	**Default**
🔒 `quarkus.vault.health.enabled` Whether or not an health check is published in case the smallrye-health extension is present.	boolean	`false`
🔒 `quarkus.vault.health.stand-by-ok` Specifies if being a standby should still return the active status code instead of the standby status code.	boolean	`false`
🔒 `quarkus.vault.health.performance-stand-by-ok` Specifies if being a performance standby should still return the active status code instead of the performance standby status code.	boolean	`false`
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. Lease renewal does not apply.	string	

`quarkus.vault.authentication.client-token-wrapping-token` Client token wrapped in a wrapping token, such as what is returned by: vault token create -wrap-ttl=60s -policy=myapp client-token and client-token-wrapping-token are exclusive. Lease renewal does not apply.	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	
`quarkus.vault.authentication.app-role.secret-id-wrapping-token` Wrapping token containing a Secret Id, obtained from: vault write -wrap-ttl=60s -f auth/approle/role/myapp/secret-id secret-id and secret-id-wrapping-token are exclusive	string	
`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.userpass.password-wrapping-token` Wrapping token containing a Password, obtained from: vault kv get -wrap-ttl=60s secret/ The key has to be 'password', meaning the password has initially been provisioned with: vault kv put secret/ password= password and password-wrapping-token are exclusive	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`
`quarkus.vault.authentication.kubernetes.auth-mount-path` Allows configure Kubernetes authentication mount path.	string	`auth/kubernetes`
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	
`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:
 	
``
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
`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	

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.

References

- HashiCorp Vault
- Quarkus Security