

Quarkus - Using HashiCorp Vault with Databases

The most common use case when working with Vault is to keep confidential the database connection credentials. There are several approaches that are supported in Quarkus, with different levels of sophistication and security:

- Property fetched from the KV Secret Engine using the Vault MicroProfile Config Source
- Quarkus Credentials Provider
- Vault Dynamic DB Credentials

This guide aims at providing examples for each of those approaches. We will reuse the application implemented in the [Vault guide](#) and enhance it with a simple persistence use case.



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:

- to complete the [Vault guide](#)
- roughly 20 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- Docker installed

Application

We assume the [Vault guide](#) application has been developed, a Vault container is running, and the root token is known. In this section we are going to start a PostgreSQL database, and add a persistence service in the application.

Add the *Hibernate* and *PostgreSQL* extensions to the application:

```
mvn quarkus:add-extension -Dextensions="io.quarkus:quarkus
-hibernate-orm,io.quarkus:quarkus-jdbc-postgresql"
```

Create a simple service:

```
@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:

```
@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**:

```
@Inject
SantaClausService santaClausService;

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

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
```

Now we are ready to configure Vault and Quarkus to be able to connect to this database from the application.

Vault MicroProfile Config Source

The simplest approach is to write the database password in the KV secret engine under the path that is fetched from the Vault MicroProfile Config Source.

Open a new shell, **docker exec** in the Vault container and set the root token:

```
docker exec -it dev-vault sh
export VAULT_TOKEN=s.5VUS8pte13RqekCB2fmMT3u2
```

Add a **dbpassword** property in the **config** path of the KV secret engine, beside the original **a-private-key** property:

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

Add the following configuration in Quarkus to use the value of property **dbpassword** as our database password:

```
# configure your datasource
quarkus.datasource.db-kind = postgresql
quarkus.datasource.username = sarah
quarkus.datasource.password = ${dbpassword}
quarkus.datasource.jdbc.url =
jdbc:postgresql://localhost:5432/mydatabase

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

Compile and start the application:

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

Test it with the `gift-count` endpoint:

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

You should see:

```
0
```

This approach is certainly the simplest of all. It has also the big advantage of working with any subsystem that requires a secret information in the configuration (i.e. not just *Agroal*). 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 every time a connection creation is attempted.

Credentials Provider

In this approach we introduce a new abstraction called the *Credentials Provider* that acts as an intermediary component between the *Agroal* datasource and Vault. The additional configuration required is small, and it has the major advantage of handling gracefully database password change while the application is running, without any restart. Since all new connections go through the *Credentials Provider* to fetch their password, we make sure we have a fresh value every time.

Create a new path (different than the `config` path) in Vault where the database password will be added:

```
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.db-kind = postgresql
quarkus.datasource.username = sarah
quarkus.datasource.credentials-provider = mydatabase
quarkus.datasource.jdbc.url =
jdbc:postgresql://localhost:5432/mydatabase
```

Recompile, start and test the `gift-count` endpoint. You should see `0` again.

Dynamic Database Credentials

The two previous approaches work well and are very popular. However they rely on a well known user configured in the application (i.e. the database user), and the security comes from the confidentiality of the password. If the password was stolen, we would have to change it in the database and in Vault. Regular rotating passwords is actually a very good practice to limit (in time) the impact of getting the password stolen.

A more sophisticated approach consists in letting Vault create and retire database accounts on a regular basis. This is supported in Vault with the [Database secret engine](#). A number of databases are supported, such as [PostgreSQL](#).

First we need to enable the `database` secret engine, configure the `postgresql-database-plugin` and create the database role `mydbrole` (replace `10.0.0.3` by the actual host that is running the PostgreSQL container; for simplicity we disabled `TLS` between Vault and the PostgreSQL database):

```

vault secrets enable database

vault write database/config/mydb \
    plugin_name=postgresql-database-plugin \
    allowed_roles=mydbrole \

connection_url=postgresql://{{username}}:{{password}}@10.0.0.3:5432
/mydatabase?sslmode=disable \
    username=sarah \
    password=connor

cat <<EOF > vault-postgres-creation.sql
CREATE ROLE "{{name}}" WITH LOGIN PASSWORD '{{password}}' VALID
UNTIL '{{expiration}}';
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public
TO "{{name}}";
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA public to
 "{{name}}";
EOF

vault write database/roles/mydbrole \
    db_name=mydb creation_statements=@vault-postgres-creation.sql \
    default_ttl=1h \
    max_ttl=24h \
    revocation_statements="ALTER ROLE \"{{name}}\" NOLOGIN;" \
    renew_statements="ALTER ROLE \"{{name}}\" VALID UNTIL
'{{expiration}}';"

```



For this use case, user **sarah** configured above needs to be a PostgreSQL super user with the capability to create users.

Then we need to give a read capability to the Quarkus application on path **database/creds/mydbrole**.

```

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

```

Now that Vault knows how to create users in PostgreSQL, we just need to change the **mydatabase** credentials provider to use a **database-credentials-role**.

```
quarkus.vault.credentials-provider.mydatabase.database-credentials-  
role=mydbrole
```



When using `quarkus.hibernate-orm.database.generation=drop-create`, objects get created with the applicative user. Since a user will be created every time the applications starts, database objects will be created with the first created user, then we will attempt to drop them on the second run with a different user that is not the owner. As expected this will fail. As a result, it is recommended to use `quarkus.hibernate-orm.database.generation=update` in this section.

Recompile with `./mvnw package`, start and test the `gift-count` endpoint. You should see `0` again.

Notice in the logs:

```
2020-04-22 14:29:48,522 DEBUG [io.qua.vau.run.VaultDbManager]  
(Agroal_682171661) generated mydbrole credentials: {leaseId:  
database/creds/mydbrole/L6PxoI68gZDeVPXP0RAA4c0a, renewable: true,  
leaseDuration: 60s, valid_until: Wed Apr 22 14:30:48 CEST 2020,  
username: v-userpass-mydbrole-He0MJCmy9coEn02my2AR-1587558588,  
password:***}
```

If you connect to the PostgreSQL database, and list all users configured on `mydatabase`, you will see the `sarah` super user, but also the technical users dynamically created by Vault:

```
docker exec -it postgres-quarkus-hibernate bash  
psql mydatabase sarah  
  
mydatabase=# \du  
  
List of  
roles  
  
Attributes | Role name | Member of |  
-----+-----+-----+  
-----+-----+-----+  
sarah | Superuser,  
Create role, Create DB, Replication, Bypass RLS | {}  
v-userpass-mydbrole-He0MJCmy9coEn02my2AR-1587558588 | Password  
valid until 2020-04-22 12:30:53+00 | {}  
v-userpass-mydbrole-N2ITbBXxoqMQ3A3cZL88-1587558572 | Cannot login  
+ | {}  
 | Password  
valid until 2020-04-22 12:30:37+00 |
```

As you can see 2 users have been created:

- `v-userpass-mydbrole-N2ITbBXxoqMQ3A3cZL88-1587558572` that has expired, which was created while we were executing the tests.
- `v-userpass-mydbrole-He0MJCmy9coEn02my2AR-1587558588` that is valid until `12:30:53`.

As expected, users have been created dynamically by Vault, with expiration dates, allowing a rotation to occur, without breaking existing connections, allowing a greater level of security than the traditional *password* based approaches.