Quarkus - Using Security with a JDBC Realm

This guide demonstrates how your Quarkus application can use a database to store your user identities.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+

Architecture

In this example, we build a very simple microservice which offers three endpoints:

- `/api/public`
- `/api/users/me`
- `/api/admin`

The `/api/public` endpoint can be accessed anonymously. The `/api/admin` endpoint is protected with RBAC (Role-Based Access Control) where only users granted with the `admin` role can access. At this endpoint, we use the `@RolesAllowed` annotation to declaratively enforce the access constraint. The `/api/users/me` endpoint is also protected with RBAC (Role-Based Access Control) where only users granted with the `user` role can access. As a response, it returns a JSON document with details about the user.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `security-jdbc-quickstart` directory.

Creating the Maven Project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.CR2:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=security-jdbc-quickstart \
    -Dextensions="elytron-security-jdbc, jdbc-postgresql, resteasy"
cd security-jdbc-quickstart
```

> ℹ️ Don't forget to add the database connector library of choice. Here we are using PostgreSQL as identity store.

This command generates a Maven project, importing the `elytron-security-jdbc` extension which is an `wildfly-elytron-realm-jdbc` adapter for Quarkus applications.

Writing the application

Let's start by implementing the `/api/public` endpoint. As you can see from the source code below, it is just a regular JAX-RS resource:

```java
package org.acme.security.jdbc;

import javax.annotation.security.PermitAll;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/api/public")
public class PublicResource {

    @GET
    @PermitAll
    @Produces(MediaType.TEXT_PLAIN)
    public String publicResource() {
        return "public";
    }
}
```

The source code for the `/api/admin` endpoint is also very simple. The main difference here is that we are using a `@RolesAllowed` annotation to make sure that only users granted with the `admin` role can access the endpoint:

```
package org.acme.security.jdbc;

import javax.annotation.security.RolesAllowed;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/api/admin")
public class AdminResource {

    @GET
    @RolesAllowed("admin")
    @Produces(MediaType.TEXT_PLAIN)
    public String adminResource() {
        return "admin";
    }
}
```

Finally, let's consider the `/api/users/me` endpoint. As you can see from the source code below, we are trusting only users with the `user` role. We are using `SecurityContext` to get access to the current authenticated Principal and we return the user's name. This information is loaded from the database.

```
package org.acme.security.jdbc;

import javax.annotation.security.RolesAllowed;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.SecurityContext;

@Path("/api/users")
public class UserResource {

    @GET
    @RolesAllowed("user")
    @Path("/me")
    @Produces(MediaType.APPLICATION_JSON)
    public String me(@Context SecurityContext securityContext) {
        return securityContext.getUserPrincipal().getName();
    }
}
```

Configuring the Application

The `elytron-security-jdbc` extension requires at least one datasource to access to your database.

```
quarkus.datasource.url=jdbc:postgresql:elytron-security-jdbc
quarkus.datasource.driver=org.postgresql.Driver
quarkus.datasource.username=quarkus
quarkus.datasource.password=quarkus
```

In our context, we are using PostgreSQL as identity store and we init the database with users and roles.

```sql
CREATE TABLE test_user (
  id INT,
  username VARCHAR(255),
  password VARCHAR(255),
  role VARCHAR(255)
);

INSERT INTO test_user (id, username, password, role) VALUES (1,
'admin', 'admin', 'admin');
INSERT INTO test_user (id, username, password, role) VALUES (2,
'user','user', 'user');
```

> ℹ️ It is probably useless but we kindly remind you that you must not store clear-text passwords in production environment ;-). The `elytron-security-jdbc` offers a built-in bcrypt password mapper.

We can now configure the Elytron JDBC Realm.

```
quarkus.security.jdbc.enabled=true
quarkus.security.jdbc.principal-query.sql=SELECT u.password, u.role
FROM test_user u WHERE u.username=? ①
quarkus.security.jdbc.principal-query.clear-password-
mapper.enabled=true ②
quarkus.security.jdbc.principal-query.clear-password-
mapper.password-index=1
quarkus.security.jdbc.principal-query.attribute-mappings.0.index=2
③
quarkus.security.jdbc.principal-query.attribute-
mappings.0.to=groups
```

The `elytron-security-jdbc` extension requires at least one principal query to autenticate the user and its identity.

① We define a parameterized SQL statement (with exactly 1 parameter) which should return the user's password plus any additional information you want to load.

② We configure the password mapper with the position of the password field in the `SELECT` fields and other information like salt, hash encoding, etc.

③ We use `attribute-mappings` to bind the `SELECT` projection fields (ie. `u.role` here) to the target Principal representation attributes.

> **ℹ** In the `principal-query` configuration all the `index` properties start at 1 (rather than 0).

Testing the Application

The application is now protected and the identities are provided by our database. The very first thing to check is to ensure the anonymous access works.

```
$ curl -i -X GET http://localhost:8080/api/public
HTTP/1.1 200 OK
Content-Length: 6
Content-Type: text/plain;charset=UTF-8

public%
```

Now, let's try a to hit a protected resource anonymously.

```
$ curl -i -X GET http://localhost:8080/api/admin
HTTP/1.1 401 Unauthorized
Content-Length: 14
Content-Type: text/html;charset=UTF-8

Not authorized%
```

So far so good, now let's try with an allowed user.

```
$ curl -i -X GET -u admin:admin http://localhost:8080/api/admin
HTTP/1.1 200 OK
Content-Length: 5
Content-Type: text/plain;charset=UTF-8

admin%
```

By providing the `admin:admin` credentials, the extension authenticated the user and loaded their roles. The `admin` user is authorized to access to the protected resources.

The user `admin` should be forbidden to access a resource protected with `@RolesAllowed("user")`

because it doesn't have this role.

```
$ curl -i -X GET -u admin:admin http://localhost:8080/api/users/me
HTTP/1.1 403 Forbidden
Content-Length: 34
Content-Type: text/html;charset=UTF-8

Forbidden%
```

Finally, using the user `user` works and the security context contains the principal details (username for instance).

```
curl -i -X GET -u user:user http://localhost:8080/api/users/me
HTTP/1.1 200 OK
Content-Length: 4
Content-Type: text/plain;charset=UTF-8

user%
```

Advanced Configuration

This guide only covered an easy use case, the extension offers multiple datasources, multiple principal queries configuration as well as a bcrypt password mapper.

```
quarkus.datasource.url=jdbc:postgresql:multiple-data-sources-users
quarkus.datasource.driver=org.postgresql.Driver
quarkus.datasource.username=quarkus
quarkus.datasource.password=quarkus

quarkus.datasource.permissions.url=jdbc:postgresql:multiple-data-
sources-permissions
quarkus.datasource.permissions.driver=org.postgresql.Driver
quarkus.datasource.permissions.username=quarkus
quarkus.datasource.permissions.password=quarkus

quarkus.security.jdbc.enabled=true
quarkus.security.jdbc.principal-query.sql=SELECT u.password FROM
test_user u WHERE u.username=?
quarkus.security.jdbc.principal-query.clear-password-
mapper.enabled=true
quarkus.security.jdbc.principal-query.clear-password-
mapper.password-index=1

quarkus.security.jdbc.principal-query.roles.sql=SELECT r.role_name
FROM test_role r, test_user_role ur WHERE ur.username=? AND
ur.role_id = r.id
quarkus.security.jdbc.principal-query.roles.datasource=permissions
quarkus.security.jdbc.principal-query.roles.attribute-
mappings.0.index=1
quarkus.security.jdbc.principal-query.roles.attribute-
mappings.0.to=groups
```

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.security.jdbc.realm-name` The realm name	string	`Quarkus`
🔒 `quarkus.security.jdbc.enabled` If the properties store is enabled.	boolean	`false`
🔒 `quarkus.security.jdbc.principal-query.sql` The sql query to find the password	string	

🔒 `quarkus.security.jdbc.principal-query.datasource` The data source to use	string	
🔒 `quarkus.security.jdbc.principal-query.clear-password-mapper.enabled` If the clear-password-mapper is enabled.	boolean	`false`
🔒 `quarkus.security.jdbc.principal-query.clear-password-mapper.password-index` The index (1 based numbering) of the column containing the clear password	int	`1`
🔒 `quarkus.security.jdbc.principal-query.bcrypt-password-mapper.enabled` If the bcrypt-password-mapper is enabled.	boolean	`false`
🔒 `quarkus.security.jdbc.principal-query.bcrypt-password-mapper.password-index` The index (1 based numbering) of the column containing the password hash	int	`0`
🔒 `quarkus.security.jdbc.principal-query.bcrypt-password-mapper.hash-encoding` A string referencing the password hash encoding ("BASE64" or "HEX")	`base64`, `hex`	`BASE64`
🔒 `quarkus.security.jdbc.principal-query.bcrypt-password-mapper.salt-index` The index (1 based numbering) of the column containing the Bcrypt salt	int	`0`
🔒 `quarkus.security.jdbc.principal-query.bcrypt-password-mapper.salt-encoding` A string referencing the salt encoding ("BASE64" or "HEX")	`base64`, `hex`	`BASE64`
🔒 `quarkus.security.jdbc.principal-query.bcrypt-password-mapper.iteration-count-index` The index (1 based numbering) of the column containing the Bcrypt iteration count	int	`0`

🔒 `quarkus.security.jdbc.principal-query.attribute-mappings."attribute-mappings".index` The index (1 based numbering) of column to map	int	0
🔒 `quarkus.security.jdbc.principal-query.attribute-mappings."attribute-mappings".to` The target attribute name	string	required ❗
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".sql` The sql query to find the password	string	
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".datasource` The data source to use	string	
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".attribute-mappings."attribute-mappings".index` The index (1 based numbering) of column to map	int	0
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".attribute-mappings."attribute-mappings".to` The target attribute name	string	required ❗
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".clear-password-mapper.enabled` If the clear-password-mapper is enabled.	boolean	false
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".clear-password-mapper.password-index` The index (1 based numbering) of the column containing the clear password	int	1
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".bcrypt-password-mapper.enabled` If the bcrypt-password-mapper is enabled.	boolean	false

🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".bcrypt-password-mapper.password-index` The index (1 based numbering) of the column containing the password hash	int	0
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".bcrypt-password-mapper.hash-encoding` A string referencing the password hash encoding ("BASE64" or "HEX")	`base64`, `hex`	BASE64
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".bcrypt-password-mapper.salt-index` The index (1 based numbering) of the column containing the Bcrypt salt	int	0
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".bcrypt-password-mapper.salt-encoding` A string referencing the salt encoding ("BASE64" or "HEX")	`base64`, `hex`	BASE64
🔒 `quarkus.security.jdbc.principal-query."named-principal-queries".bcrypt-password-mapper.iteration-count-index` The index (1 based numbering) of the column containing the Bcrypt iteration count	int	0

Future Work

- Propose more password mappers.

- Provide an opinionated configuration.