

# Quarkus - Using Security from JPA

This guide demonstrates how your Quarkus application can use a database to store your user identities with [Hibernate ORM](#) or [Hibernate ORM with Panache](#).

## 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-jpa-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.CR1:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=security-jpa-quickstart \
  -Dextensions="security-jpa, jdbc-postgresql, resteasy,
hibernate-orm-panache"
cd security-jpa-quickstart
```



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 `security-jpa` extension which allows you to map your security source to JPA entities.

## 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:

```
package org.acme.security.jpa;

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.jpa;

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.jpa;

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();
    }
}

```

## Defining our user entity

We can now describe how our security information is stored in our model by adding a few annotations to our `User` entity:

```
package org.acme.security.jpa;

import javax.persistence.Entity;
import javax.persistence.Table;

import io.quarkus.hibernate.orm.panache.PanacheEntity;
import io.quarkus.security.common.BcryptUtil;
import io.quarkus.security.jpa.Password;
import io.quarkus.security.jpa.Roles;
import io.quarkus.security.jpa.UserDefinition;
import io.quarkus.security.jpa.Username;

@Entity
@Table(name = "test_user")
@UserDefinition ①
public class User extends PanacheEntity {
    @Username ②
    public String username;
    @Password ③
    public String password;
    @Roles ④
    public String role;

    /**
     * Adds a new user in the database
     * @param username the user name
     * @param password the unencrypted password (it will be
    encrypted with bcrypt)
     * @param role the comma-separated roles
     */
    public static void add(String username, String password, String
    role) { ⑤
        User user = new User();
        user.username = username;
        user.password = BcryptUtil.bcryptHash(password);
        user.role = role;
        user.persist();
    }
}
```

The `security-jpa` extension is only initialized if there is a single entity annotated with `@UserDefinition`.

- ① This annotation must be present on a single entity. It can be a regular Hibernate ORM entity or a Hibernate ORM with Panache entity as in this example.
- ② This indicates the field used for the user name.
- ③ This indicates the field used for the password. This defaults to using bcrypt hashed passwords, but you can also configure it for clear text passwords.
- ④ This indicates the comma-separated list of roles added to the target Principal representation attributes.
- ⑤ This method allows us to add users while hashing the password with the proper bcrypt hash.

## Configuring the Application

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

```
quarkus.datasource.url=jdbc:postgresql:security_jpa
quarkus.datasource.driver=org.postgresql.Driver
quarkus.datasource.username=quarkus
quarkus.datasource.password=quarkus

quarkus.hibernate-orm.database.generation=drop-and-create
```

In our context, we are using PostgreSQL as identity store. The database schema is created by Hibernate ORM automatically on startup (change this in production) and we initialise the database with users and roles in the `Startup` class:

```
package org.acme.security.jpa;

import javax.enterprise.event.Observes;
import javax.inject.Singleton;
import javax.transaction.Transactional;

import io.quarkus.runtime.StartupEvent;

@Singleton
public class Startup {
    @Transactional
    public void loadUsers(@Observes StartupEvent evt) {
        // reset and load all test users
        User.deleteAll();
        User.add("admin", "admin", "admin");
        User.add("user", "user", "user");
    }
}
```



It is probably useless but we kindly remind you that you must not store clear-text passwords in production environments ;-). As a result, the `security-jpa` defaults to using bcrypt-hashed passwords.

## 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 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%
```

## Supported model types

- The `@UserDefinition` class must be a JPA entity (with Panache or not).
- The `@Username` and `@Password` field types must be of type `String`.
- The `@Roles` field must either be of type `String` or `Collection<String>` or alternately a `Collection<X>` where `X` is an entity class with one `String` field annotated with the `@RolesValue` annotation.
- Each `String` role element type will be parsed as a comma-separated list of roles.

## Storing roles in another entity

You can also store roles in another entity:

```

@EntityDefinition
@Table(name = "test_user")
@Entity
public class User extends PanacheEntity {
    @Username
    public String name;

    @Password
    public String pass;

    @ManyToMany
    @Roles
    public List<Role> roles = new ArrayList<>();
}

@Entity
public class Role extends PanacheEntity {

    @ManyToMany(mappedBy = "roles")
    public List<ExternalRolesUserEntity> users;

    @RolesValue
    public String role;
}

```

## Password storage and hashing

By default, we consider passwords to be stored hashed with `bcrypt` under the [Modular Crypt Format \(MCF\)](#).

When you need to create such a hashed password we provide the convenient `String BcryptUtil.bcryptHash(String password)` function, which defaults to creating a random salt and hashing in 10 iterations (though you can specify the iterations and salt too).



with MCF you don't need dedicated columns to store the hashing algo, the iterations count or the salt because they're all stored in the hashed value.

WARN: you can also store passwords in clear text with `@Password(PasswordType.CLEAR)` but we strongly recommend against it in production.