

Quarkus - Authorization of Web Endpoints

Quarkus has an integrated pluggable web security layer. If security is enabled all HTTP requests will have a permission check performed to make sure they are allowed to continue.



Configuration authorization checks are executed before any annotation-based authorization check is done, so both checks have to pass for a request to be allowed.

Authorization using Configuration

The default implementation allows you to define permissions using config in `application.properties`. An example config is shown below:

```
quarkus.http.auth.policy.role-policy1.roles-allowed=user,admin
```

①

```
quarkus.http.auth.permission.roles1.paths=/roles-secured/*,/other*/api/*
```

②

```
quarkus.http.auth.permission.roles1.policy=role-policy1
```

```
quarkus.http.auth.permission.permit1.paths=/public/*
```

③

```
quarkus.http.auth.permission.permit1.policy=permit
```

```
quarkus.http.auth.permission.permit1.methods=GET
```

```
quarkus.http.auth.permission.deny1.paths=/forbidden
```

④

```
quarkus.http.auth.permission.deny1.policy=deny
```

- ① This defines a role based policy that allows users with the `user` and `admin` roles. This is referenced by later rules.
- ② This is a permission set that references the previously defined policy. `roles1` is an arbitrary name, you can call the permission sets whatever you want.
- ③ This permission references the default `permit` built-in policy to allow `GET` methods to `/public`. This is actually a no-op in this example, as this request would have been allowed anyway.
- ④ This permission references the built-in `deny` policy for `/forbidden`. This is an exact path match as it does not end with `*`.

Permissions are defined in config using permission sets. These are arbitrarily named permission grouping. Each permission set must specify a policy that is used to control access. There are three built-in policies: `deny`, `permit` and `authenticated`, which respectively permits all, denies all and

only allows authenticated users.

It is also possible to define role based policies, as shown in the example. These policies will only allow users with the specified roles to access the resources.

Matching on paths, methods

Permission sets can also specify paths and methods as a comma separated list. If a path ends with `*` then it is considered to be a wildcard match and will match all sub paths, otherwise it is an exact match and will only match that specific path:

```
quarkus.http.auth.permission.permit1.paths=/public*/,/css*/,/js*/,/  
robots.txt  
quarkus.http.auth.permission.permit1.policy=permit  
quarkus.http.auth.permission.permit1.methods=GET,HEAD
```

Matching path but not method

If a request would match one or more permission sets based on the path, but does not match any due to method requirements then the request is rejected.



Given the above permission set, `GET /public/foo` would match both the path and method and thus be allowed, whereas `POST /public/foo` would match the path but not the method and would thus be rejected.

Matching multiple paths: longest path wins

Matching is always done on a longest path wins basis, less specific permission sets are not considered if a more specific one has been matched:

```
quarkus.http.auth.permission.permit1.paths=/public/*  
quarkus.http.auth.permission.permit1.policy=permit  
quarkus.http.auth.permission.permit1.methods=GET,HEAD  
  
quarkus.http.auth.permission.deny1.paths=/public/forbidden-folder/*  
quarkus.http.auth.permission.deny1.policy=deny
```



Given the above permission set, `GET /public/forbidden-folder/foo` would match both permission sets' paths, but because it matches the `deny1` permission set's path on a longer match, `deny1` will be chosen and the request will be rejected.

Subpath permissions always win against the root path permissions as explained above in the `deny1` versus `permit1` permission example. Here is another example showing a subpath permission allowing a public resource access with the root path permission requiring the authorization:



```
quarkus.http.auth.policy.user-policy.roles-allowed=user
quarkus.http.auth.permission.roles.paths=/api/*
quarkus.http.auth.permission.roles.policy=user-policy

quarkus.http.auth.permission.public.paths=/api/noauth/*
quarkus.http.auth.permission.public.policy=permit
```

Matching multiple paths: most specific method wins

If a path is registered with multiple permission sets then any permission sets that specify a HTTP method will take precedence and permissions sets without a method will not be considered (assuming of course the method matches). In this instance, the permission sets without methods will only come into effect if the request method does not match any of the sets with method permissions.

```
quarkus.http.auth.permission.permit1.paths=/public/*
quarkus.http.auth.permission.permit1.policy=permit
quarkus.http.auth.permission.permit1.methods=GET,HEAD

quarkus.http.auth.permission.deny1.paths=/public/*
quarkus.http.auth.permission.deny1.policy=deny
```



Given the above permission set, `GET /public/foo` would match both permission sets' paths, but because it matches the `permit1` permission set's explicit method, `permit1` will be chosen and the request will be accepted. `PUT /public/foo` on the other hand, will not match the method permissions of `permit1` and so `deny1` will be activated and reject the request.

Matching multiple paths and methods: both win

If multiple permission sets specify the same path and method (or multiple have no method) then both permissions have to allow access for the request to proceed. Note that for this to happen both have to either have specified the method, or have no method, method specific matches take precedence as stated above:

```
quarkus.http.auth.policy.user-policy1.roles-allowed=user
quarkus.http.auth.policy.admin-policy1.roles-allowed=admin

quarkus.http.auth.permission.roles1.paths=/api/*,/restricted/*
quarkus.http.auth.permission.roles1.policy=user-policy1

quarkus.http.auth.permission.roles2.paths=/api/*,/admin/*
quarkus.http.auth.permission.roles2.policy=admin-policy1
```



Given the above permission set, `GET /api/foo` would match both permission sets' paths, so would require both the `user` and `admin` roles.

Configuration Properties to Deny access

There are two configuration settings that alter the RBAC Deny behavior:

- `quarkus.security.jaxrs.deny-unannotated-endpoints=true|false` - if set to true, the access will be denied for all JAX-RS endpoints by default. That is if the security annotations do not define the access control. Defaults to `false`.
- `quarkus.security.deny-unannotated-members=true|false` - if set to true, the access will be denied to all CDI methods and JAX-RS endpoints that do not have security annotations but are defined in classes that contain methods with security annotations. Defaults to `false`.

Authorization using Annotations

Quarkus comes with built-in security to allow for Role-Based Access Control (RBAC) based on the common security annotations `@RolesAllowed`, `@DenyAll`, `@PermitAll` on REST endpoints and CDI beans. An example of an endpoint that makes use of both JAX-RS and Common Security annotations to describe and secure its endpoints is given in [SubjectExposingResource Example](#). Quarkus also provides the `io.quarkus.security.Authenticated` annotation that will permit any authenticated user to access the resource (equivalent to `@RolesAllowed("**")`).

```
import java.security.Principal;

import javax.annotation.security.DenyAll;
import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.SecurityContext;

@Path("subject")
public class SubjectExposingResource {

    @GET
    @Path("secured")
    @RolesAllowed("Tester") ①
    public String getSubjectSecured(@Context SecurityContext sec) {
        Principal user = sec.getUserPrincipal(); ②
        String name = user != null ? user.getName() : "anonymous";
        return name;
    }

    @GET
    @Path("unsecured")
    @PermitAll ③
    public String getSubjectUnsecured(@Context SecurityContext sec)
    {
        Principal user = sec.getUserPrincipal(); ④
        String name = user != null ? user.getName() : "anonymous";
        return name;
    }

    @GET
    @Path("denied")
    @DenyAll ⑤
    public String getSubjectDenied(@Context SecurityContext sec) {
        Principal user = sec.getUserPrincipal();
        String name = user != null ? user.getName() : "anonymous";
        return name;
    }
}
```

- ① This `/subject/secured` endpoint requires an authenticated user that has been granted the role "Tester" through the use of the `@RolesAllowed("Tester")` annotation.
- ② The endpoint obtains the user principal from the JAX-RS `SecurityContext`. This will be non-null for a secured endpoint.

- ③ The `/subject/unsecured` endpoint allows for unauthenticated access by specifying the `@PermitAll` annotation.
- ④ This call to obtain the user principal will return null if the caller is unauthenticated, non-null if the caller is authenticated.
- ⑤ The `/subject/denied` endpoint disallows any access regardless of whether the call is authenticated by specifying the `@DenyAll` annotation.