Red Hat build of Quarkus 3.8

Authorization of web endpoints

Last Updated: 2024-09-19

Legal Notice

Abstract

This guide explores authorization mechanisms for web endpoints, focusing on both configuration-based and annotation-based methods. It examines configuring authorization, including the use of built-in and custom policies, path and method matching, and handling complex path scenarios. It then delves into the nuances of role-based access control and permission management, covering properties for access denial, disabling permissions, and mapping roles to SecurityIdentity. It concludes with a discussion on using annotations for securing RESTful services, highlighting standard security annotations and their application.

Table of Contents

PROVIDING FEEDBACK ON RED HAT BUILD OF QUARKUS DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to **create a ticket**.
2. Enter a brief description of the issue in the **Summary**.
3. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
4. Clicking **Submit** creates and routes the issue to the appropriate documentation team.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. AUTHORIZATION OF WEB ENDPOINTS

Quarkus incorporates a pluggable web security layer. When security is active, the system performs a permission check on all HTTP requests to determine if they should proceed.

Using **@PermitAll** will not open a path if the path is restricted by the **quarkus.http.auth.** configuration. To ensure specific paths are accessible, appropriate configurations must be made within the Quarkus security settings.

NOTE

If you use Jakarta RESTful Web Services, consider using **quarkus.security.jaxrs.deny-unannotated-endpoints** or **quarkus.security.jaxrs.default-roles-allowed** to set default security requirements instead of HTTP path-level matching because annotations can override these properties on an individual endpoint.

Authorization is based on user roles that the security provider provides. To customize these roles, a **SecurityIdentityAugmentor** can be created, see Security Identity Customization.

1.1. AUTHORIZATION USING CONFIGURATION

Permissions are defined in the Quarkus configuration by permission sets, each specifying a policy for access control.

Table 1.1. Red Hat build of Quarkus policies summary

Built-in policy	Description
deny	This policy denies all users.
permit	This policy permits all users.
authenticated	This policy permits only authenticated users.

You can define role-based policies that allow users with specific roles to access the resources.

Example of a role-based policy

```
quarkus.http.auth.policy.role-policy1.roles-allowed=user,admin
```
(1)

(1) This defines a role-based policy that allows users with the **user** and **admin** roles.

You can reference a custom policy by configuring the built-in permission sets that are defined in the **application.properties** file, as outlined in the following configuration example:

Example of policy configuration

```
quarkus.http.auth.permission.permit1.paths=/public/*
quarkus.http.auth.permission.permit1.policy=permit
quarkus.http.auth.permission.permit1.methods=GET
```
(1)

```
quarkus.http.auth.permission.deny1.paths=/forbidden                    2
quarkus.http.auth.permission.deny1.policy=deny

quarkus.http.auth.permission.roles1.paths=/roles-secured/*,/other/*,/api/*    3
quarkus.http.auth.permission.roles1.policy=role-policy1
```

1. This permission references the default built-in **permit** policy to allow **GET** methods to **/public**. In this case, the demonstrated setting would not affect this example because this request is allowed anyway.
2. This permission references the built-in **deny** policy for **/forbidden**. It is an exact path match because it does not end with ***.
3. This permission set references the previously defined policy. **roles1** is an example name; you can call the permission sets whatever you want.

WARNING

The exact path **/forbidden** in the example will not secure the **/forbidden/** path. It is necessary to add a new exact path for the **/forbidden/** path to ensure proper security coverage.

1.1.1. Custom HttpSecurityPolicy

Sometimes it might be useful to register your own named policy. You can get it done by creating application scoped CDI bean that implements the **io.quarkus.vertx.http.runtime.security.HttpSecurityPolicy** interface like in the example below:

```
import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.security.identity.SecurityIdentity;
import io.quarkus.vertx.http.runtime.security.HttpSecurityPolicy;
import io.smallrye.mutiny.Uni;
import io.vertx.ext.web.RoutingContext;

@ApplicationScoped
public class CustomNamedHttpSecPolicy implements HttpSecurityPolicy {
    @Override
    public Uni<CheckResult> checkPermission(RoutingContext event, Uni<SecurityIdentity> identity,
            AuthorizationRequestContext requestContext) {
        if (customRequestAuthorization(event)) {
            return Uni.createFrom().item(CheckResult.PERMIT);
        }
        return Uni.createFrom().item(CheckResult.DENY);
    }

    @Override
    public String name() {
```

```
        return "custom"; (1)
    }

    private static boolean customRequestAuthorization(RoutingContext event) {
        // here comes your own security check
        return !event.request().path().endsWith("denied");
    }
}
```

(1) Named HTTP Security policy will only be applied to requests matched by the **application.properties** path matching rules.

Example of custom named HttpSecurityPolicy referenced from configuration file

```
quarkus.http.auth.permission.custom1.paths=/custom/*
quarkus.http.auth.permission.custom1.policy=custom (1)
```

(1) Custom policy name must match the value returned by the **io.quarkus.vertx.http.runtime.security.HttpSecurityPolicy.name** method.

TIP

You can also create global **HttpSecurityPolicy** invoked on every request. Just do not implement the **io.quarkus.vertx.http.runtime.security.HttpSecurityPolicy.name** method and leave the policy nameless.

1.1.2. Matching on paths and methods

Permission sets can also specify paths and methods as a comma-separated list. If a path ends with the * wildcard, the query it generates matches all sub-paths. Otherwise, it queries for an exact match and only matches that specific path:

```
quarkus.http.auth.permission.permit1.paths=/public*,/css/*,/js/*,/robots.txt (1)
quarkus.http.auth.permission.permit1.policy=permit
quarkus.http.auth.permission.permit1.methods=GET,HEAD
```

(1) The * wildcard at the end of the path matches zero or more path segments, but never any word starting from the /**public** path. For that reason, a path like /**public-info** is not matched by this pattern.

1.1.3. Matching a path but not a method

The request is rejected if it matches one or more permission sets based on the path but none of the required methods.

TIP

Given the preceding permission set, **GET /public/foo** would match both the path and method and therefore be allowed. In contrast, **POST /public/foo** would match the path but not the method, and, therefore, be rejected.

1.1.4. Matching multiple paths: longest path wins

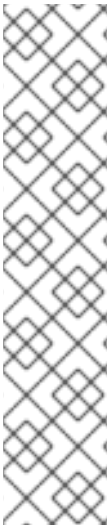
Matching is always done on the "longest path wins" basis. Less specific permission sets are not considered if a more specific one has been matched:

```
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
```

TIP

Given the preceding permission set, **GET /public/forbidden-folder/foo** would match both permission sets' paths. However, because the longer path matches the path of the **deny1** permission set, **deny1** is chosen, and the request is rejected.



NOTE

Subpath permissions precede root path permissions, as the **deny1** versus **permit1** permission example previously illustrated.

This rule is further exemplified by a scenario where subpath permission allows access to a public resource while the root path permission necessitates 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
```

1.1.5. Matching multiple sub-paths: longest path to the * wildcard wins

Previous examples demonstrated matching all sub-paths when a path concludes with the * wildcard.

This wildcard also applies in the middle of a path, representing a single path segment. It cannot be mixed with other path segment characters; thus, path separators always enclose the * wildcard, as seen in the **/public/*/about-us** path.

When several path patterns correspond to the same request path, the system selects the longest sub-path leading to the * wildcard. In this context, every path segment character is more specific than the * wildcard.

Here is a simple example:

```
quarkus.http.auth.permission.secured.paths=/api/*/detail                    1
quarkus.http.auth.permission.secured.policy=authenticated
quarkus.http.auth.permission.public.paths=/api/public-product/detail        2
quarkus.http.auth.permission.public.policy=permit
```

1 Request paths like **/api/product/detail** can only be accessed by authenticated users.

2 The path **/api/public-product/detail** is more specific, therefore accessible by anyone.

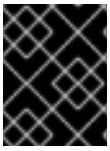
IMPORTANT

All paths secured with the authorization using configuration should be tested. Writing path patterns with multiple wildcards can be cumbersome. Please make sure paths are authorized as you intended.

In the following example, paths are ordered from the most specific to the least specific one:

Request path /one/two/three/four/five matches ordered from the most specific to the least specific path

```
/one/two/three/four/five
/one/two/three/four/*
/one/two/three/*/five
/one/two/three/*/*
/one/two/*/four/five
/one/*/three/four/five
/*/two/three/four/five
/*/two/three/*/five
/*
```



IMPORTANT

The * wildcard at the end of the path matches zero or more path segments. The * wildcard placed anywhere else matches exactly one path segment.

1.1.6. Matching multiple paths: most specific method wins

When a path is registered with multiple permission sets, the permission sets explicitly specifying an HTTP method that matches the request take precedence. In this instance, the permission sets without methods only come into effect if the request method does not match permission sets with the method specification.

```
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
```

NOTE

The preceding permission set shows that **GET /public/foo** matches the paths of both permission sets.However, it specifically aligns with the explicit method of the **permit1** permission set.Therefore, **permit1** is selected, and the request is accepted.

In contrast, **PUT /public/foo** does not match the method permissions of **permit1**. As a result, **deny1** is activated, leading to the rejection of the request.

1.1.7. Matching multiple paths and methods: both win

Sometimes, the previously described rules allow multiple permission sets to win simultaneously. In that case, for the request to proceed, all the permissions must allow access. For this to happen, both must either have specified the method or have no method. Method-specific matches take precedence.

```
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
```

TIP

Given the preceding permission set, **GET /api/foo** would match both permission sets' paths, requiring both the **user** and **admin** roles.

1.1.8. Configuration properties to deny access

The following configuration settings alter the role-based access control (RBAC) denying behavior:

quarkus.security.jaxrs.deny-unannotated-endpoints=true|false

If set to true, access is denied for all Jakarta REST endpoints by default. If a Jakarta REST endpoint has no security annotations, it defaults to the **@DenyAll** behavior. This helps you to avoid accidentally exposing an endpoint that is supposed to be secured. Defaults to **false**.

quarkus.security.jaxrs.default-roles-allowed=role1,role2

Defines the default role requirements for unannotated endpoints. The **** role is a special role that means any authenticated user. This cannot be combined with **deny-unannotated-endpoints** because **deny** takes effect instead.

quarkus.security.deny-unannotated-members=true|false

If set to true, the access is denied to all CDI methods and Jakarta REST endpoints that do not have security annotations but are defined in classes that contain methods with security annotations. Defaults to **false**.

1.1.9. Disabling permissions

Permissions can be disabled at build time with an **enabled** property for each declared permission, such as:

```
quarkus.http.auth.permission.permit1.enabled=false
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
```

Permissions can be reenabled at runtime with a system property or environment variable, such as: **-Dquarkus.http.auth.permission.permit1.enabled=true**.

1.1.10. Permission paths and HTTP root path

The **quarkus.http.root-path** configuration property changes the http endpoint context path.

By default, **quarkus.http.root-path** is prepended automatically to configured permission paths then do not use a forward slash, for example:

```
quarkus.http.auth.permission.permit1.paths=public/*,css/*,js/*,robots.txt
```

This configuration is equivalent to the following:

```
quarkus.http.auth.permission.permit1.paths=${quarkus.http.root-path}/public/*,${quarkus.http.root-path}/css/*,${quarkus.http.root-path}/js/*,${quarkus.http.root-path}/robots.txt
```

A leading slash changes how the configured permission path is interpreted. The configured URL is used as-is, and paths are not adjusted if the value of **quarkus.http.root-path** changes.

Example:

```
quarkus.http.auth.permission.permit1.paths=/public/*,css/*,js/*,robots.txt
```

This configuration only impacts resources served from the fixed or static URL, /**public**, which might not match your application resources if **quarkus.http.root-path** has been set to something other than /.

For more information, see Path Resolution in Quarkus.

1.1.11. Map SecurityIdentity roles

Winning role-based policy can map the **SecurityIdentity** roles to the deployment-specific roles. These roles are then applicable for endpoint authorization by using the **@RolesAllowed** annotation.

```
quarkus.http.auth.policy.admin-policy1.roles.admin=Admin1 1
quarkus.http.auth.permission.roles1.paths=/*
quarkus.http.auth.permission.roles1.policy=admin-policy1
```

1 Map the **admin** role to **Admin1** role. The **SecurityIdentity** will have both **admin** and **Admin1** roles.

1.1.12. Shared permission checks

One important rule for unshared permission checks is that only one path match is applied, the most specific one. Naturally you can specify as many permissions with the same winning path as you want and they will all be applied. However, there can be permission checks you want to apply to many paths without repeating them over and over again. That's where shared permission checks come in, they are always applied when the permission path is matched.

Example of custom named HttpSecurityPolicy applied on every HTTP request

```
quarkus.http.auth.permission.custom1.paths=/*
quarkus.http.auth.permission.custom1.shared=true 1
quarkus.http.auth.permission.custom1.policy=custom

quarkus.http.auth.policy.admin-policy1.roles-allowed=admin
quarkus.http.auth.permission.roles1.paths=/admin/*
quarkus.http.auth.permission.roles1.policy=admin-policy1
```

1 Custom HttpSecurityPolicy will be also applied on the **/admin/1** path together with the **admin-policy1** policy.

TIP

Configuring many shared permission checks is less effective than configuring unshared ones. Use shared permissions to complement unshared permission checks like in the example below.

Map SecurityIdentity roles with shared permission

```
quarkus.http.auth.policy.role-policy1.roles.root=admin,user 1
quarkus.http.auth.permission.roles1.paths=/secured/* 2
quarkus.http.auth.permission.roles1.policy=role-policy1
quarkus.http.auth.permission.roles1.shared=true

quarkus.http.auth.policy.role-policy2.roles-allowed=user 3
quarkus.http.auth.permission.roles2.paths=/secured/user/*
quarkus.http.auth.permission.roles2.policy=role-policy2

quarkus.http.auth.policy.role-policy3.roles-allowed=admin
quarkus.http.auth.permission.roles3.paths=/secured/admin/*
quarkus.http.auth.permission.roles3.policy=role-policy3
```

1 Role **root** will be able to access **/secured/user/*** and **/secured/admin/*** paths.

2 The **/secured/*** path can only be accessed by authenticated users. This way, you have secured the **/secured/all** path and so on.

3 Shared permissions are always applied before unshared ones, therefore a **SecurityIdentity** with the **root** role will have the **user** role as well.

1.2. AUTHORIZATION USING ANNOTATIONS

Red Hat build of Quarkus includes 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.

Table 1.2. Red Hat build of Quarkus annotation types summary

Annotation type	Description
@DenyAll	Specifies that no security roles are allowed to invoke the specified methods.
@PermitAll	Specifies that all security roles are allowed to invoke the specified methods. **@PermitAll** lets everybody in, even without authentication.

Annotation type	Description
@RolesAllowed	Specifies the list of security roles allowed to access methods in an application. As an equivalent to **@RolesAllowed("**")**, Red Hat build of Quarkus also provides the **io.quarkus.security.Authenticated** annotation that permits any authenticated user to access the resource.

The following SubjectExposingResource example demonstrates an endpoint that uses both Jakarta REST and Common Security annotations to describe and secure its endpoints.

SubjectExposingResource example

```
import java.security.Principal;

import jakarta.annotation.security.RolesAllowed;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.core.Context;
import jakarta.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. The /**subject/secured** endpoint requires an authenticated user with the granted "Tester" role through the use of the **@RolesAllowed("Tester")** annotation.
2. The endpoint obtains the user principal from the Jakarta REST **SecurityContext**. This returns **non-null** for a secured endpoint.
3. The /**subject/unsecured** endpoint allows for unauthenticated access by specifying the **@PermitAll** annotation.
4. The call to obtain the user principal returns **null** if the caller is unauthenticated and **non-null** if the caller is authenticated.
5. The /**subject/denied** endpoint declares the **@DenyAll** annotation, disallowing all direct access to it as a REST method, regardless of the user calling it. The method is still invokable internally by other methods in this class.

CAUTION

If you plan to use standard security annotations on the IO thread, review the information in Proactive Authentication.

The **@RolesAllowed** annotation value supports property expressions including default values and nested property expressions. Configuration properties used with the annotation are resolved at runtime.

Table 1.3. Annotation value examples

Annotation	Value explanation
@RolesAllowed("${admin-role}")	The endpoint allows users with the role denoted by the value of the **admin-role** property.
@RolesAllowed("${tester.group}-${tester.role}")	An example showing that the value can contain multiple variables.
@RolesAllowed("${customer:User}")	A default value demonstration. The required role is denoted by the value of the **customer** property. However, if that property is not specified, a role named **User** is required as a default.

Example of a property expressions usage in the @RolesAllowed annotation

```
admin=Administrator
tester.group=Software
tester.role=Tester
%prod.secured=User
%dev.secured=**
all-roles=Administrator,Software,Tester,User
```

Subject access control example

```
import java.security.Principal;
```

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

@Path("subject")
public class SubjectExposingResource {

    @GET
    @Path("admin")
    @RolesAllowed("${admin}") (1)
    public String getSubjectSecuredAdmin(@Context SecurityContext sec) {
        return getUsername(sec);
    }

    @GET
    @Path("software-tester")
    @RolesAllowed("${tester.group}-${tester.role}") (2)
    public String getSubjectSoftwareTester(@Context SecurityContext sec) {
        return getUsername(sec);
    }

    @GET
    @Path("user")
    @RolesAllowed("${customer:User}") (3)
    public String getSubjectUser(@Context SecurityContext sec) {
        return getUsername(sec);
    }

    @GET
    @Path("secured")
    @RolesAllowed("${secured}") (4)
    public String getSubjectSecured(@Context SecurityContext sec) {
        return getUsername(sec);
    }

    @GET
    @Path("list")
    @RolesAllowed("${all-roles}") (5)
    public String getSubjectList(@Context SecurityContext sec) {
        return getUsername(sec);
    }

    private String getUsername(SecurityContext sec) {
        Principal user = sec.getUserPrincipal();
        String name = user != null ? user.getName() : "anonymous";
        return name;
    }
}
```

(1) The **@RolesAllowed** annotation value is set to the value of **Administrator**.

2 This /**subject/software-tester** endpoint requires an authenticated user that has been granted the role of "Software-Tester". It is possible to use multiple expressions in the role definition.

3 This /**subject/user** endpoint requires an authenticated user that has been granted the role "User" through the use of the **@RolesAllowed("${customer:User}")** annotation because we did not set the configuration property **customer**.

4 In production, this /**subject/secured** endpoint requires an authenticated user with the **User** role. In development mode, it allows any authenticated user.

5 Property expression **all-roles** will be treated as a collection type **List**, therefore, the endpoint will be accessible for roles **Administrator**, **Software**, **Tester** and **User**.

1.2.1. Permission annotation

Quarkus also provides the **io.quarkus.security.PermissionsAllowed** annotation, which authorizes any authenticated user with the given permission to access the resource. This annotation is an extension of the common security annotations and checks the permissions granted to a **SecurityIdentity** instance.

Example of endpoints secured with the @PermissionsAllowed annotation

```
package org.acme.crud;

import io.quarkus.arc.Arc;
import io.vertx.ext.web.RoutingContext;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.QueryParam;

import io.quarkus.security.PermissionsAllowed;

import java.security.BasicPermission;
import java.security.Permission;
import java.util.Collection;
import java.util.Collections;

@Path("/crud")
public class CRUDResource {

    @PermissionsAllowed("create") 1
    @PermissionsAllowed("update")
    @POST
    @Path("/modify/repeated")
    public String createOrUpdate() {
        return "modified";
    }

    @PermissionsAllowed(value = {"create", "update"}, inclusive=true) 2
    @POST
    @Path("/modify/inclusive")
    public String createOrUpdate(Long id) {
        return id + " modified";
    }
```

```
    @PermissionsAllowed({"see:detail", "see:all", "read"}) 3
    @GET
    @Path("/id/{id}")
    public String getItem(String id) {
        return "item-detail-" + id;
    }

    @PermissionsAllowed(value = "list", permission = CustomPermission.class) 4
    @Path("/list")
    @GET
    public Collection<String> list(@QueryParam("query-options") String queryOptions) {
        // your business logic comes here
        return Collections.emptySet();
    }

    public static class CustomPermission extends BasicPermission {

        public CustomPermission(String name) {
            super(name);
        }

        @Override
        public boolean implies(Permission permission) {
            var event = Arc.container().instance(RoutingContext.class).get(); 5
            var publicContent = "public-content".equals(event.request().params().get("query-options"));
            var hasPermission = getName().equals(permission.getName());
            return hasPermission && publicContent;
        }
    }
}
```

1. The resource method **createOrUpdate** is only accessible for a user with both **create** and **update** permissions.

2. By default, at least one of the permissions specified through one annotation instance is required. You can require all permissions by setting **inclusive=true**. Both resource methods **createOrUpdate** have equal authorization requirements.

3. Access is granted to **getItem** if **SecurityIdentity** has either **read** permission or **see** permission and one of the **all** or **detail** actions.

4. You can use your preferred **java.security.Permission** implementation. By default, string-based permission is performed by **io.quarkus.security.StringPermission**.

5. Permissions are not beans, therefore the only way to obtain bean instances is programmatically by using **Arc.container()**.

CAUTION

If you plan to use the **@PermissionsAllowed** on the IO thread, review the information in Proactive Authentication.

NOTE

@PermissionsAllowed is not repeatable on the class level due to a limitation with Quarkus interceptors. For more information, see the Repeatable interceptor bindings section of the Quarkus "CDI reference" guide.

The easiest way to add permissions to a role-enabled **SecurityIdentity** instance is to map roles to permissions. Use Authorization using configuration to grant the required **SecurityIdentity** permissions for **CRUDResource** endpoints to authenticated requests, as outlined in the following example:

```
quarkus.http.auth.policy.role-policy1.permissions.user=see:all                    (1)
quarkus.http.auth.policy.role-policy1.permissions.admin=create,update,read        (2)
quarkus.http.auth.permission.roles1.paths=/crud/modify/*,/crud/id/*               (3)
quarkus.http.auth.permission.roles1.policy=role-policy1

quarkus.http.auth.policy.role-policy2.permissions.user=list
quarkus.http.auth.policy.role-policy2.permission-
class=org.acme.crud.CRUDResource$CustomPermission                                 (4)
quarkus.http.auth.permission.roles2.paths=/crud/list
quarkus.http.auth.permission.roles2.policy=role-policy2
```

1. Add the permission **see** and the action **all** to the **SecurityIdentity** instance of the **user** role. Similarly, for the **@PermissionsAllowed** annotation, **io.quarkus.security.StringPermission** is used by default.
2. Permissions **create**, **update**, and **read** are mapped to the role **admin**.
3. The role policy **role-policy1** allows only authenticated requests to access **/crud/modify** and **/crud/id** sub-paths. For more information about the path-matching algorithm, see Matching multiple paths: longest path wins later in this guide.
4. You can specify a custom implementation of the **java.security.Permission** class. Your custom class must define exactly one constructor that accepts the permission name and optionally some actions, for example, **String** array. In this scenario, the permission **list** is added to the **SecurityIdentity** instance as **new CustomPermission("list")**.

You can also create a custom **java.security.Permission** class with additional constructor parameters. These additional parameters get matched with arguments of the method annotated with the **@PermissionsAllowed** annotation. Later, Quarkus instantiates your custom permission with actual arguments, with which the method annotated with the **@PermissionsAllowed** has been invoked.

Example of a custom java.security.Permission class that accepts additional arguments

```
package org.acme.library;

import java.security.Permission;
import java.util.Arrays;
import java.util.Set;

public class LibraryPermission extends Permission {

    private final Set<String> actions;
    private final Library library;
```

```
    public LibraryPermission(String libraryName, String[] actions, Library library) { 1
        super(libraryName);
        this.actions = Set.copyOf(Arrays.asList(actions));
        this.library = library;
    }

    @Override
    public boolean implies(Permission requiredPermission) {
        if (requiredPermission instanceof LibraryPermission) {
            LibraryPermission that = (LibraryPermission) requiredPermission;
            boolean librariesMatch = getName().equals(that.getName());
            boolean requiredLibraryIsSublibrary = library.isParentLibraryOf(that.library);
            boolean hasOneOfRequiredActions = that.actions.stream().anyMatch(actions::contains);
            return (librariesMatch || requiredLibraryIsSublibrary) && hasOneOfRequiredActions;
        }
        return false;
    }

    // here comes your own implementation of the `java.security.Permission` class methods

    public static abstract class Library {

        protected String description;

        abstract boolean isParentLibraryOf(Library library);

    }

    public static class MediaLibrary extends Library {

        @Override
        boolean isParentLibraryOf(Library library) {
            return library instanceof MediaLibrary;
        }
    }

    public static class TvLibrary extends MediaLibrary {
        // TvLibrary specific implementation of the 'isParentLibraryOf' method
    }
}
```

1 There must be exactly one constructor of a custom **Permission** class. The first parameter is always considered to be a permission name and must be of type **String**. Quarkus can optionally pass permission actions to the constructor. For this to happen, declare the second parameter as **String[]**.

The **LibraryPermission** class permits access to the current or parent library if **SecurityIdentity** is allowed to perform one of the required actions, for example, **read**, **write**, or **list**.

The following example shows how the **LibraryPermission** class can be used:

```
package org.acme.library;

import io.quarkus.security.PermissionsAllowed;
import jakarta.enterprise.context.ApplicationScoped;
```

```
import org.acme.library.LibraryPermission.Library;

@ApplicationScoped
public class LibraryService {

    @PermissionsAllowed(value = "tv:write", permission = LibraryPermission.class) 1
    public Library updateLibrary(String newDesc, Library update) {
        update.description = newDesc;
        return update;
    }

    @PermissionsAllowed(value = "tv:write", permission = LibraryPermission.class, params = "library")
2
    @PermissionsAllowed(value = {"tv:read", "tv:list"}, permission = LibraryPermission.class)
    public Library migrateLibrary(Library migrate, Library library) {
        // migrate libraries
        return library;
    }

}
```

1 The formal parameter **update** is identified as the first **Library** parameter and gets passed to the **LibraryPermission** class. However, the **LibraryPermission** must be instantiated each time the **updateLibrary** method is invoked.

2 Here, the first **Library** parameter is **migrate**; therefore, the **library** parameter gets marked explicitly through **PermissionsAllowed#params**. The permission constructor and the annotated method must have the parameter **library** set; otherwise, validation fails.

Example of a resource secured with the LibraryPermission

```
package org.acme.library;

import io.quarkus.security.PermissionsAllowed;
import jakarta.inject.Inject;
import jakarta.ws.rs.PUT;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;
import org.acme.library.LibraryPermission.Library;

@Path("/library")
public class LibraryResource {

    @Inject
    LibraryService libraryService;

    @PermissionsAllowed(value = "tv:write", permission = LibraryPermission.class)
    @PUT
    @Path("/id/{id}")
    public Library updateLibrary(@PathParam("id") Integer id, Library library) {
        ...
    }

    @PUT
    @Path("/service-way/id/{id}")
```

```
    public Library updateLibrarySvc(@PathParam("id") Integer id, Library library) {
        String newDescription = "new description " + id;
        return libraryService.updateLibrary(newDescription, library);
    }

}
```

Similarly to the **CRUDResource** example, the following example shows how you can grant a user with the **admin** role permissions to update **MediaLibrary**:

```
package org.acme.library;

import io.quarkus.runtime.annotations.RegisterForReflection;

@RegisterForReflection 1
public class MediaLibraryPermission extends LibraryPermission {

    public MediaLibraryPermission(String libraryName, String[] actions) {
        super(libraryName, actions, new MediaLibrary()); 2
    }

}
```

1 When building a native executable, the permission class must be registered for reflection unless it is also used in at least one **io.quarkus.security.PermissionsAllowed#name** parameter.

2 We want to pass the **MediaLibrary** instance to the **LibraryPermission** constructor.

```
quarkus.http.auth.policy.role-policy3.permissions.admin=media-library:list,media-library:read,media-library:write 1
quarkus.http.auth.policy.role-policy3.permission-class=org.acme.library.MediaLibraryPermission
quarkus.http.auth.permission.roles3.paths=/library/*
quarkus.http.auth.permission.roles3.policy=role-policy3
```

1 Grants the permission **media-library**, which permits **read**, **write**, and **list** actions. Because **MediaLibrary** is the **TvLibrary** class parent, a user with the **admin** role is also permitted to modify **TvLibrary**.

TIP

The **/library/*** path can be tested from a Keycloak provider Dev UI page, because the user **alice** which is created automatically by the Dev Services for Keycloak has an **admin** role.

The examples provided so far demonstrate role-to-permission mapping. It is also possible to programmatically add permissions to the **SecurityIdentity** instance. In the following example, **SecurityIdentity** is customized to add the same permission that was previously granted with the HTTP role-based policy.

Example of adding the LibraryPermission programmatically to SecurityIdentity

```
import java.security.Permission;
import java.util.function.Function;
```

```
import jakarta.enterprise.context.ApplicationScoped;

import io.quarkus.security.identity.AuthenticationRequestContext;
import io.quarkus.security.identity.SecurityIdentity;
import io.quarkus.security.identity.SecurityIdentityAugmentor;
import io.quarkus.security.runtime.QuarkusSecurityIdentity;
import io.smallrye.mutiny.Uni;

@ApplicationScoped
public class PermissionsIdentityAugmentor implements SecurityIdentityAugmentor {

    @Override
    public Uni<SecurityIdentity> augment(SecurityIdentity identity, AuthenticationRequestContext
context) {
        if (isNotAdmin(identity)) {
            return Uni.createFrom().item(identity);
        }
        return Uni.createFrom().item(build(identity));
    }

    private boolean isNotAdmin(SecurityIdentity identity) {
        return identity.isAnonymous() || !"admin".equals(identity.getPrincipal().getName());
    }

    SecurityIdentity build(SecurityIdentity identity) {
        Permission possessedPermission = new MediaLibraryPermission("media-library",
                new String[] { "read", "write", "list"}); (1)
        return QuarkusSecurityIdentity.builder(identity)
                .addPermissionChecker(new Function<Permission, Uni<Boolean>>() { (2)
                    @Override
                    public Uni<Boolean> apply(Permission requiredPermission) {
                        boolean accessGranted = possessedPermission.implies(requiredPermission);
                        return Uni.createFrom().item(accessGranted);
                    }
                })
                .build();
    }

}
```

(1) The permission **media-library** that was created can perform **read**, **write**, and **list** actions. Because **MediaLibrary** is the **TvLibrary** class parent, a user with the **admin** role is also permitted to modify **TvLibrary**.

(2) You can add a permission checker through **io.quarkus.security.runtime.QuarkusSecurityIdentity.Builder#addPermissionChecker**.

CAUTION

Annotation-based permissions do not work with custom Jakarta REST SecurityContexts because there are no permissions in **jakarta.ws.rs.core.SecurityContext**.

1.3. REFERENCES

- Quarkus Security overview
- Quarkus Security architecture
- Authentication mechanisms in Quarkus
- Basic authentication
- Getting started with Security by using Basic authentication and Jakarta Persistence
- OpenID Connect Bearer Token Scopes And SecurityIdentity Permissions