

Quarkus - Security Guide

Quarkus security allows you to define security authorization requirements for your code using annotations and/or configuration, and provides several ways to load security authentication information using Quarkus extensions.

Authentication sources

Quarkus supports several sources to load authentication information from. You need to import at least one of the following extensions in order for Quarkus to know how to find the authentication information to check for authorizations:

Table 1. Security Extensions

Extension	Description
quarkus-elytron-security-properties-file	Provides support for simple properties files that can be used for testing security. This supports both embedding user info in <code>application.properties</code> and standalone properties files.
quarkus-security-jpa	Provides support for authenticating via JPA.
quarkus-elytron-security-jdbc	Provides support for authenticating via JDBC.
quarkus-elytron-security-oauth2	Provides support for OAuth2 flows using Elytron. This extension will likely be deprecated soon and replaced by a reactive Vert.x version.
quarkus-smallrye-jwt	A MicroProfile JWT implementation that provides support for authenticating using Json Web Tokens. This also allows you to inject the token and claims into the application as per the MP JWT spec.
quarkus-oidc	Provides support for authenticating via an OpenID Connect provider such as Keycloak.
quarkus-keycloak-authorization	Provides support for a policy enforcer using Keycloak Authorization Services.

Please see the linked documents above for details on how to setup the various extensions.

Authenticating via HTTP

Quarkus has two built in authentication mechanisms for HTTP based FORM and BASIC auth. This mechanism is pluggable however so extensions can add additional mechanisms (most notably OpenID Connect for Keycloak based auth).

Basic Authentication

To enable basic authentication set `quarkus.http.auth.basic=true`. You must also have at least one extension installed that provides a username/password based `IdentityProvider`, such as [Elytron JDBC](#).

Form Based Authentication

Quarkus provides form based authentication that works in a similar manner to traditional Servlet form based auth. Unlike traditional form authentication the authenticated user is not stored in a HTTP session, as Quarkus does not provide clustered HTTP session support. Instead the authentication information is stored in an encrypted cookie, which can be read by all members of the cluster (provided they all share the same encryption key).

The encryption key can be set using the `quarkus.http.auth.session.encryption-key` property, and it must be at least 16 characters long. This key is hashed using SHA-256 and the resulting digest is used as a key for AES-256 encryption of the cookie value. This cookie contains a expiry time as part of the encrypted value, so all nodes in the cluster must have their clocks synchronised. At one minute intervals a new cookie will be generated with an updated expiry time if the session is in use.

The following properties can be used to configure form based auth:

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

Configuration property	Type	Default
 <code>quarkus.http.auth.form.enabled</code> If form authentication is enabled	boolean	<code>false</code>
 <code>quarkus.http.auth.form.login-page</code> The login page	string	<code>/login.html</code>
 <code>quarkus.http.auth.form.error-page</code> The error page	string	<code>/error.html</code>
 <code>quarkus.http.auth.form.landing-page</code> The landing page to redirect to if there is no saved page to redirect back to	string	<code>/index.html</code>
 <code>quarkus.http.auth.form.redirect-after-login</code> Option to disable redirect to landingPage if there is no saved page to redirect back to. Form Auth POST is followed by redirect to landingPage by default.	boolean	<code>true</code>

<p> <code>quarkus.http.auth.form.timeout</code></p> <p>The inactivity (idle) timeout. When inactivity timeout is reached, cookie is not renewed and a new login is enforced.</p>	<p>Duration </p>	<p>PT30M</p>
<p> <code>quarkus.http.auth.form.new-cookie-interval</code></p> <p>How old a cookie can get before it will be replaced with a new cookie with an updated timeout, also referred to as "renewal-timeout". Note that smaller values will result in slightly more server load (as new encrypted cookies will be generated more often), however larger values affect the inactivity timeout as the timeout is set when a cookie is generated. For example if this is set to 10 minutes, and the inactivity timeout is 30m, if a users last request is when the cookie is 9m old then the actual timeout will happen 21m after the last request, as the timeout is only refreshed when a new cookie is generated. In other words no timeout is tracked on the server side; the timestamp is encoded and encrypted in the cookie itself and it is decrypted and parsed with each request.</p>	<p>Duration </p>	<p>PT1M</p>
<p> <code>quarkus.http.auth.form.cookie-name</code></p> <p>The cookie that is used to store the persistent session</p>	<p>string</p>	<p>quarkus-credential</p>

About the Duration format

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the [Duration#parse\(\) javadoc](#).



You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.

Proactive Authentication

By default Quarkus does what we call proactive authentication. This means that if an incoming request has a credential then that request will always be authenticated (even if the target page does not require authentication).

This means that requests with an invalid credential will always be rejected, even for public pages. You can change this behaviour and only authenticate when required by setting `quarkus.http.auth.proactive=false`.

Authorization in REST endpoints and CDI beans 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("**")`).

SubjectExposingResource Example

```
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.

Authorization of Web Endpoints using configuration

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 permitted 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.

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` build in policy `/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 permit all, deny all and only allow authenticated users respectively.

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 wins

Matching is always done on a longest path 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.

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.

Authenticated representation

For every authenticated resource, you can inject a `SecurityIdentity` instance to get the authenticated identity information.

In some other contexts you may have other parallel representations of the same information (or parts of it) such as `SecurityContext` for JAX-RS or `JsonWebToken` for JWT.

Configuration

There are two configuration settings that alter the RBAC 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`.

Registering Security Providers

When running in native mode the default behavior for Graal native image generation is to only include the main "SUN" provider unless you have enabled SSL, in which case all security providers are registered. If you are not using SSL, then you can selectively register security providers by name using the `quarkus.security.users.security-providers` property. The following example illustrates configuration to register the "SunRsaSign" and "SunJCE" security providers:

Example Security Providers Configuration

```
quarkus.security.security-providers=SunRsaSign,SunJCE
...
```

Security Identity Customization

Internally, the identity providers create and update an instance of the `io.quarkus.security.identity.SecurityIdentity` class which holds the principal, roles, credentials which were used to authenticate the client (user) and other security attributes. An easy option to customize `SecurityIdentity` is to register a custom `SecurityIdentityAugmentor`, for example, the augmentor below adds an addition role:

```

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CompletionStage;
import javax.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;

@ApplicationScoped
public class RolesAugmentor implements SecurityIdentityAugmentor {

    @Override
    public int priority() {
        return 0;
    }

    @Override
    public CompletionStage<SecurityIdentity> augment
    (SecurityIdentity identity, AuthenticationRequestContext context) {

        CompletableFuture<SecurityIdentity> cs = new
    CompletableFuture<>();
        if (identity.isAnonymous()) {
            cs.complete(identity);
        } else {
            // create a new builder and copy principal, attributes,
    credentials and roles from the original
            QuarkusSecurityIdentity.Builder builder =
    QuarkusSecurityIdentity.builder()
                .setPrincipal(identity.getPrincipal())
                .addAttributes(identity.getAttributes())
                .addCredentials(identity.getCredentials())
                .addRoles(identity.getRoles());

            // add custom role source here
            builder.addRole("dummy");

            cs.complete(builder.build());
        }

        return cs;
    }
}

```

Reactive Security

If you are going to use security in a reactive environment, you will likely need SmallRye Context Propagation:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-context-propagation</artifactId>
</dependency>
```

This will allow you to propagate the identity throughout the reactive callbacks. You also need to make sure you are using an executor that is capable of propagating the identity (e.g. no `CompletableFuture.supplyAsync`), to make sure that quarkus can propagate it. For more information see the [Context Propagation Guide](#).