

# Quarkus - Quarkus Extension for Spring Security API

While users are encouraged to use [Java standard annotations for security authorizations](#), Quarkus provides a compatibility layer for Spring Security in the form of the `spring-security` extension.

This guide explains how a Quarkus application can leverage the well known Spring Security annotations to define authorizations on RESTful services using roles.



This technology is considered preview.

In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

## Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- Some familiarity with the Spring Web extension

## 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 `spring-security-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:1.7.0.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=spring-security-quickstart \
  -DclassName="org.acme.spring.security.GreetingController" \
  -Dpath="/greeting" \
  -Dextensions="spring-web, spring-security, quarkus-elytron-
security-properties-file"
cd spring-security-quickstart
```

This command generates a Maven project with a REST endpoint and imports the `spring-web`, `spring-security` and `security-properties-file` extensions.

If you already have your Quarkus project configured, you can add the `spring-web`, `spring-security` and `security-properties-file` extensions to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="spring-web, spring-
security, quarkus-elytron-security-properties-file"
```

This will add the following to your `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-spring-web</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-spring-security</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-elytron-security-properties-
file</artifactId>
</dependency>
```

For more information about `security-properties-file`, you can check out the guide of the [quarkus-elytron-security-properties-file](#) extension.

## GreetingController

The Quarkus Maven plugin automatically generated a controller with the Spring Web annotations to define our REST endpoint (instead of the JAX-RS ones used by default). The `src/main/java/org/acme/spring/web/GreetingController.java` file looks as follows:

```

package org.acme.spring.security;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.PathVariable;

@RestController
@RequestMapping("/greeting")
public class GreetingController {

    @GetMapping
    public String hello() {
        return "hello";
    }
}

```

## GreetingControllerTest

Note that a test for the controller has been created as well:

```

package org.acme.spring.security;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class GreetingControllerTest {

    @Test
    public void testHelloEndpoint() {
        given()
            .when().get("/greeting")
            .then()
                .statusCode(200)
                .body(is("hello"));
    }
}

```

# Package and run the application

Run the application with: `./mvnw quarkus:dev`. Open your browser to <http://localhost:8080/greeting>.

The result should be: `{"message": "hello"}`.

## Modify the controller to secure the `hello` method

In order to restrict access to the `hello` method to users with certain roles, the `@Secured` annotation will be utilized. The updated controller will be:

```
package org.acme.spring.security;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.PathVariable;

@RestController
@RequestMapping("/greeting")
public class GreetingController {

    @Secured("admin")
    @GetMapping
    public String hello() {
        return "hello";
    }
}
```

The easiest way to setup users and roles for our example is to use the `security-properties-file` extension. This extension essentially allows users and roles to be defined in the main Quarkus configuration file - `application.properties`. For more information about this extension check [the associated guide](#). An example configuration would be the following:

```
quarkus.security.users.embedded.enabled=true
quarkus.security.users.embedded.plain-text=true
quarkus.security.users.embedded.users.scott=jb0ss
quarkus.security.users.embedded.roles.scott=admin,user
quarkus.security.users.embedded.users.stuart=test
quarkus.security.users.embedded.roles.stuart=user
```

Note that the test also needs to be updated. It could look like:

# GreetingControllerTest

```
package org.acme.spring.security;

import io.quarkus.test.junit.QuarkusTest;
import org.junit.jupiter.api.Test;

import static io.restassured.RestAssured.given;
import static org.hamcrest.CoreMatchers.is;

@QuarkusTest
public class GreetingControllerTest {

    @Test
    public void testHelloEndpointForbidden() {
        given().auth().preemptive().basic("stuart", "test")
            .when().get("/greeting")
            .then()
            .statusCode(403);
    }

    @Test
    public void testHelloEndpoint() {
        given().auth().preemptive().basic("scott", "jb0ss")
            .when().get("/greeting")
            .then()
            .statusCode(200)
            .body(is("hello"));
    }
}
```

## Test the changes

### Access allowed

Open your browser again to <http://localhost:8080/greeting> and introduce **scott** and **jb0ss** in the dialog displayed.

The word **hello** should be displayed.

### Access forbidden

Open your browser again to <http://localhost:8080/greeting> and let empty the dialog displayed.

The result should be:

Access to localhost was denied  
You don't have authorization to view this page.  
HTTP ERROR 403

## Run the application as a native executable

You can of course create a native image using the instructions of the [Building a native executable guide](#).

## Supported Spring Security functionalities

Quarkus currently only supports a subset of the functionalities that Spring Security provides with more features being planned. More specifically, Quarkus supports the security related features of role-based authorization semantics (think of `@Secured` instead of `@RolesAllowed`).

### Annotations

The table below summarizes the supported annotations:

Table 1. Supported Spring Security annotations

| Name          | Comments                          |
|---------------|-----------------------------------|
| @Secured      |                                   |
| @PreAuthorize | See next section for more details |

### @PreAuthorize

Quarkus provides support for some of the most used features of Spring Security's `@PreAuthorize` annotation. The expressions that are supported are the following:

#### hasRole

To test if the current user has a specific role, the `hasRole` expression can be used inside `@PreAuthorize`.

Some examples are: `@PreAuthorize("hasRole('admin')")`, `@PreAuthorize("hasRole(@roles.USER)")` where the `roles` is a bean that could be defined like so:

```
import org.springframework.stereotype.Component;

@Component
public class Roles {

    public final String ADMIN = "admin";
    public final String USER = "user";
}
```

### hasAnyRole

In the same fashion as `hasRole`, users can use `hasAnyRole` to check if the logged in user has any of the specified roles.

Some examples are: `@PreAuthorize("hasAnyRole('admin')")`,  
`@PreAuthorize("hasAnyRole(@roles.USER, 'view')")`

### permitAll

Adding `@PreAuthorize("permitAll()")` to a method will ensure that that method is accessible by any user (including anonymous users). Adding it to a class will ensure that all public methods of the class that are not annotated with any other Spring Security annotation will be accessible.

### denyAll

Adding `@PreAuthorize("denyAll()")` to a method will ensure that that method is not accessible by any user. Adding it to a class will ensure that all public methods of the class that are not annotated with any other Spring Security annotation will not be accessible to any user.

### isAnonymous

When annotating a bean method with `@PreAuthorize("isAnonymous()")` the method will only be accessible if the current user is anonymous - i.e. a non logged in user.

### isAuthenticated

When annotating a bean method with `@PreAuthorize("isAuthenticated()")` the method will only be accessible if the current user is a logged in user. Essentially the method is only unavailable for anonymous users.

### #paramName == authentication.principal.username

This syntax allows users to check if a parameter (or a field of the parameter) of the secured method is equal to the logged in username.

Examples of this use case are:

```

public class Person {

    private final String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

@Component
public class MyComponent {

    @PreAuthorize("#username ==
authentication.principal.username") ①
    public void doSomething(String username, String other){

    }

    @PreAuthorize("#person.name ==
authentication.principal.username") ②
    public void doSomethingElse(Person person){

    }
}

```

① `doSomething` can be executed if the current logged in user is the same as the `username` method parameter

② `doSomethingElse` can be executed if the current logged in user is the same as the `name` field of `person` method parameter



the use of `authentication.` is optional, so using `principal.username` has the same result.

### **#paramName != authentication.principal.username**

This is similar to the previous expression with the difference being that the method parameter must be different than the logged in username.

### **@beanName.method()**

This syntax allows developers to specify that the execution of method of a specific bean will determine if the current user can access the secured method.

The syntax is best explained with an example. Let's assume that a `MyComponent` bean has been created like so:

```
@Component
public class MyComponent {

    @PreAuthorize("@personChecker.check(#person,
authentication.principal.username)")
    public void doSomething(Person person){

    }

}
```

The `doSomething` method has been annotated with `@PreAuthorize` using an expression that indicates that method `check` of a bean named `personChecker` needs to be invoked to determine whether the current user is authorized to invoke the `doSomething` method.

An example of the `PersonChecker` could be:

```
@Component
public class PersonChecker {

    @Override
    public boolean check(Person person, String username) {
        return person.getName().equals(username);
    }

}
```

Note that for the `check` method the parameter types must match what is specified in `@PreAuthorize` and that the return type must be a `boolean`.

## Combining expressions

The `@PreAuthorize` annotations allows for the combination of expressions using logical `AND` / `OR`. Currently there is a limitation where only a single logical operation can be used (meaning mixing `AND` and `OR` isn't allowed).

Some examples of allowed expressions are:

```

    @PreAuthorize("hasAnyRole('user', 'admin') AND #user ==
principal.username")
    public void allowedForUser(String user) {

    }

    @PreAuthorize("hasRole('user') OR hasRole('admin')")
    public void allowedForUserOrAdmin() {

    }

    @PreAuthorize("hasAnyRole('view1', 'view2') OR isAnonymous() OR
hasRole('test')")
    public void allowedForAdminOrAnonymous() {

    }

```

Also to be noted that currently parentheses are not supported and expressions are evaluated from left to right when needed.

## Important Technical Note

Please note that the Spring support in Quarkus does not start a Spring Application Context nor are any Spring infrastructure classes run. Spring classes and annotations are only used for reading metadata and / or are used as user code method return types or parameter types. What that means for end users, is that adding arbitrary Spring libraries will not have any effect. Moreover Spring infrastructure classes (like `org.springframework.beans.factory.config.BeanPostProcessor` for example) will not be executed.

## Conversion Table

The following table shows how Spring Security annotations can be converted to JAX-RS annotations.

| Spring            | JAX-RS                 | Comments |
|-------------------|------------------------|----------|
| @Secured("admin") | @RolesAllowed("admin") |          |

## More Spring guides

Quarkus has more Spring compatibility features. See the following guides for more details:

- [Quarkus - Extension for Spring DI](#)
- [Quarkus - Extension for Spring Web](#)
- [Quarkus - Extension for Spring Data JPA](#)

- [Quarkus - Reading properties from Spring Cloud Config Server](#)
- [Quarkus - Extension for Spring Boot properties](#)
- [Quarkus - Extension for Spring Cache](#)
- [Quarkus - Extension for Spring Scheduled](#)