Method-Level Security versus Filter Chains

Combine OAuth 2.0 (OpenID Connect)

with Spring @PreAuthorize

**Abstract**

Servlet filters are a Java standard dating back to the beginning of Java. They, along with Servlets and the Catalina Servlet Container are an integral part of Tomcat.

This document covers an application that turns-off any

ServletFilter interposition of HTTP Security policies and allows

method-level, role-based security policies to be applied

Tomcat is the reference-standard implementation of the Servlet 3.0 Standards/Specs. Any Web Server purporting to be compliant with the Servlet 3.0 Standards/Specs must exhibit the same capabilities as does Tomcat.

Additionally, Tomcat is the default Web Server engine powering Spring boot Applications, with ServletFilters acting as container-wide interceptors for Servlets – ServletFilters fire both before *and after* Servlets.

Spring’s DispatcherServlet is not new, nor is it exempt from the architecture described by the Servlet 3.0 Specification.

And, since “Controllers” are just adjuncts to Servlets/Filters, affording a specificity to Servlets, they to, comply with the HTTP Security policies which are inherent in all Servlet-based applications.

This means that any security policies inforce at the (typically) Filter level supersede (override) security policies applied at the Controller level. By extension, member methods of any Controller are subject to the same ServletFilter security policies.

Okta is being as the OAuth2.0 identity provider.

This document covers an application that turns-off any

ServletFilter interposition of HTTP Security policies and allows

method-level, role-based security policies to be applied

****

**When Things are Amiss**

A complement of error pages, spanning all HTTP Error Codes, has been created. Each links to a httpstatuses.com page that offers a more formal explanation.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

****

**Enabling Method-level Security (Spring @PreAuthorize)**

In a Spring boot application, the preferred approach to specifying Authentication + Authorization is to use the @PreAuthorize annotation directly at the method-level rather than to use the fluent, but centralized WebSecurityConfigurerAdapter API to accomplish the same Authentication + Authorization.

For one, the advantage to the annotation approach is that the libraries are more than likely better integrated between JDKs, and Web Containers.

To configure Spring Boot to selectively permit requests to say, a home endpoint, while also restricting requests to other endpoints, a Spring annotation will be used in a class that extends the abstract class: WebSecurityConfigurerAdapter.java

The annotation (to be used in conjunction with the WebSecurityConfigurerAdapter abstract class) looks like this:

@EnableGlobalMethodSecurity(prePostEnabled = true)

The above annotation explicitly enables method-level security annotations, while also disabling the default HttpBuilder implementation, which by default is assessed as part of the standard chain invocation chain, and so takes precedence.

Add the following SecurityConfig class shown on the next page – which must both have the annotation and extend the WebSecurityConfigurerAdapter abstract class – to achieve both of the above goals.

****

A screenshot of a cell phone

Description automatically generated

The above class explicitly enables method-level security annotations, while also disabling the default HttpBuilder implementation, which by default is assessed as part of the standard chain invocation chain, and so takes precedence.

NB Both the annotation and the extension of the WebSecurityConfigurerAdapter abstract class are necessary to enable method-level security annotations.

****

**Login to a Restricted (or Unrestricted) Endpoint**

Attempts to directly access an endpoint that may/may not have been granted (in this case, admin) access, results in (the commonplace) OAuth 2.0 challenge/response page.

The page below shows an OAuth 2.0 challenge/response page that appears when you are using Okta as the 2-Factor Authenticator.

A screenshot of a cell phone

Description automatically generated

By the way Okta is a formal, metrological unit of measure for “cloudiness”

(Okta is based in the micro-climatic city of S.F.

What is the mechanism behind this OAuth 2.0 challenge response behavior (something you see on sites like Grub Hub)?

There are two provisions that need to be in place for this behavior to work.

A. Look immediately below, to the Spring @PreAuthorize annotation within the WebController class, on line 20.

A screenshot of a computer screen

Description automatically generated

B. Then, look at the screen shot on the next page, where the Spring annotation topping the Spring SecurityConfig class is:

@EnableGlobalMethodSecurity

This annotation allows the Spring framework to act as an HTTP “interceptor” (likely leveraging AOP).

It is this @EnableGlobalMethodSecurity annotation, coupled with the Spring @PreAuthorize annotation within the WebController class which together, enables OAuth 2.0 evaluation (in a Spring app, anyway).

Notice in the second screen shot below, the use of the ßuilder Design Pattern, and the “fluent” construction of the HTTPSecurity instance that it affords.

In “glourious” (sic) Spring style – at the least, lose the long vain-glory springframework package appendage, folks – here is the full package declaration for the HTTPSecurity class:

org.springframework.security.config.annotation.web.builders.HTTPSecurity.java

A screenshot of a cell phone

Description automatically generated

****

This document does not cover *Groups* and the attendant *Roles* to which *Users* can be assigned.

Okta supports this type of Authorization in a Dashboard way.

**Metadata**

As you add users and assign Roles to them (perhaps within a Group), a custom metadata endpoint can help discern whether the mappings are as expected.

The custom endpoint is modeled in the UserMetaController.java <<Controller>> class.

The class (shown on next page) is admittedly creaky in concept/design.

Kinda Model One-ish in JSP terminology – logic and presentation are mashed together. <https://bit.ly/2zdEzHq>

It’s hack-ish – like the brackish water found in the S.F. Bay – but it serves only peripheral/diagnostic purpose, anyway.

A screenshot of a computer

Description automatically generated

A screenshot of a cell phone

Description automatically generated

**Testing Strategy**

A browser is one of many kinds of HTTP User-Agents.

Because of the nature of browser cache, and the way a browser typically works, you may find that – as you modify the Authentication/Authorization aspects of the application-under-test – you have to establish a new browser session to get the expected reaction to your changes.

Can be confusing.

For instance, even stopping the Chrome app (in addition to any browser windows) will allow the OAuth 2.0 access to remain active/viable after re-launching Chrome-the-app.

The easiest way is to start an Incognito Session (but still utilizing the same HTTP endpoints for the same A-B testing).

****

**Lifecycle Scripts**

Occasionally you launch two applications claiming the same port, simultaneously.

Bad times.

The run ßash script in this Spring boot demo app has a fail-safe provision which allows you to ignore Spring’s (embedded Tomcat) lifecycle.

The run ßash script calls the cullany ßash script, which reaps any running Tomcat/Catalina images, thus freeing-up resources – critically, Ports.

This lifecycle cleanup routine is done before running Tomcat/Catalina (as the first executable statement in the run script).

You can invoke the CLI-style Tomcat launch run script to find-and-destroy any/all running instances of Tomcat/Catalina (Java JVM processes, actually).

The cullany script appears on the next page, along with a full examination of the logic it embodies.

A screenshot of a computer screen

Description automatically generated

Line 1 The canonical Command-interpreter directive

Line 3 Redirect the output of a standard Linux ps command (line 5) to a file named after the

user-supplied ($1) runtime identifier (as injected into the loop construct on line 12)

Line 5 Using the default token value space as the Input Field Separator (IFS), read each line

from the file named after the user-supplied runtime identifier

Line 7 Specify the scalar/integral value of ‘2’ as the position/displacement in the columnar

output of the ps command (line 5) which contains the PID (as it is rendered on a MAC).

Line 8 Dereference the PID Variable in a call which leverages the canonical ßash cut utility to

identify the actual PID value, then reassign the actual value back to the PID Variable.

Line 10 The kill is made against the actual PID Value (as the PID Variable is once again dereferenced)