Reactive Authorization in Spring Security

Mario Gray Follow
Jun 30, 2018 · 3 min read

This demonstration explores configuring Spring Security for apps that want to also use reactive WebFlux. Source code for this example is on GitHub.

Applying effective security to applications helps to insulate from the ill effects of malicious, and accidental intent in many aspects of software development. Things like network security can only go so far in isolating harm to distributed computing applications. Prevent malicious hackers from gaining access to your systems by ensuring the tools meet the standards for your application.

Spring Security WebFlux is the framework that lets us declare security constructs to our ordinary WebFlux applications. This is similar to classical Spring Security and WebMVC ,with the major difference being the use of functional and reactive techniques.

```
@Bean
RouterFunction<?> routes() {
    return RouterFunctions
            .route(RequestPredicates.GET( pattern: "/admin"),
                    r -> ServerResponse
                            .ok()
                            .body(r.principal()
                                    .repeat()
                                    .zipWith(
                                            Mono.just(" has access."),
                                            (pp, str) -> pp.getName() + str),
                                    String.class)
            );
};
```

Sample: a WebFlux Endpoint.

Spring Security WebFlux provides us the functional/reactive techniques for engaging the security of a WebFlux web environment.

Reactive Authorization Components

How does Spring Security Webflux let us describe our security details?

ServerHttpSecurity surfaces components for customizing security behavior across our web-stack through a DSL-like, fluent API. ServerHttpSecurity ultimately builds the state of a SecurityWebFilterChain. A specialized component that holds an ordered list of the resultant filters, and a matcher for detecting when the request/response lifecycle should get filtered.

ServerHttpSecurity provides a number of ways to lock down our web stack. We can explore the possibilities by listing the configuration specifications, and their effects. We are given a variety of specifications letting us configure policies for different request/response (hence, ServerWebExchange) phases.

Component	ServerHttpSecurity method	handling use cases
AuthorizeExchangeSpec	.authorizeExchange()	pathMatchers, RBAC, custom Authorization
HeadersSpec	.headers()	Cross Site Scriptiong, Strict Transport Security, cache-control, frame options, etc...
CsrfSpec	.csrf()	setup handler and token repository
ExceptionHandlingSpec	.exceptionHandling()	handler for authentication entry point and denial
HttpBasicSpec	.httpBasic()	custom AuthenticationManager, authentication context config
RequestCacheSpec	.requestCache()	handle saving httpRequest prior to authentication
FormLoginSpec	.formLogin()	set login page, authentication behaviour on success/deny

LogoutSpec	.logout()	set logout page and handler

NOTE: Any of the above components may be disabled using it's .disable() method!

Configuring Authorization

We can setup authorization by invoking AuthorizeExchangeSpec's `authorizeExchange()` method. This will let us issue a regular expression for matching URI via PathPattern's, and then several methods to apply Access Control rules on matched service route.

For example, methods `hasRole()` and `hasAuthority()` will check the user's (via UserDetails interface) GrantedAuthority list for a specific value. The simplest privilege is SimpleGrantedAuthority, and is a String representation of the authority. For example, the `hasRole()` method is really a shorthand for `hasAuthority()` method, but requires authorities be prefixed with 'ROLE_'.

Finally, there is the `access()` method that takes a anonymous or otherwise custom implementation of ReactiveAuthorizationManager. This is useful for in-house authorization authorization implementations.

CSRF Configuration

When configuring SecurityWebFilterChain, the CsrfSpec is enabled by calling the `csrf()` method. This lets us configure CSRF tokens and handlers, or exclude CSRF entirely.

To customize CSRF behavior, create a bean of type ServerCsrfTokenRepository and set header and/or parameter attribute names as shown.

In this case, we give our customized ServerCsrfTokenRepository, and configure HTTP Basic. Calling `build()` returns the completed SecurityWebFilterChain.

HttpServerSecurity: Customizing CSRF behavior

Additionally the `and()` and `or()` and `disable()` methods lets us build another component's (e.g. FormLoginSpec) filter on the filter chain. Calling

`build()` compiles the state of the `SecurityWebFilterChain`.

Review

This brief overview should set you up for engaging the ServerHttpSecurity components. Following it's fluent API is a breeze once we get to know the components that we will visit.

The full source code for this sample can be found in my Github repository.

Spring Security Reactive Functional Java

14 claps



WRITTEN BY

Mario Gray

Follow

See responses (1)

More From Medium

Related reads

Customize your serialization using Jackson annotations

Marcos Abel in Trabe
Mar 4 · 4 min read ★

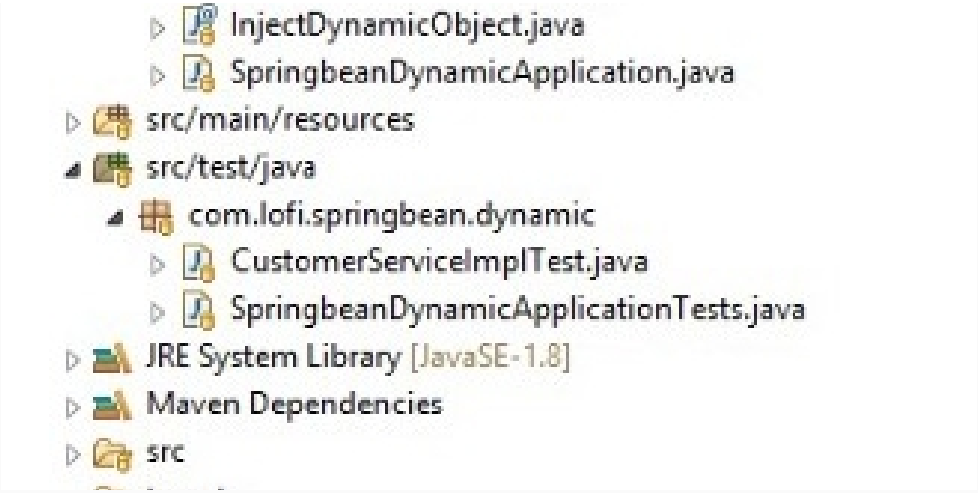
272

Related reads

Creating Spring Bean dynamically in the Runtime

Dr. Lofi Dewanto
Nov 11, 2018 · 3 min read ★

130



Related reads

Tutorial: Drools Decision Tables in Excel for a Product Proposal

Felix Kuestahler
May 10, 2018 · 7 min read ★

211