

## Posts

### Setup and Customize a Login Page With Reactive Spring Security.

=====

🕒 7 minute read ✎ Published: 25 Jul, 2018

> Spring Security provides a intuitive and concise API for managing Authentication aspects within your app.




► Table of Contents

#### Customized WebFlux Form Authentication

=====

This demonstration examines Spring Security WebFlux's Authentication mechanisms. We will look at authentication with HTML forms using Mustache, User Authentication, and customized form-based login / logout configurations.

#### ## The ServerHttpSecurity Configuration

[SecurityWebFilterChain](#)  is the governing chain of [WebFilter]'s that allows us to lock down reactive WebFlux applications. With [@EnableWebFluxSecurity](#)  turned on, we can build this object by issuing commands to the [ServerHttpSecurity](#)  DSL object.

SecurityConfiguration.java:


```

@EnableWebFluxSecurity
@Slf4j
@Configuration
public class SecurityConfiguration {

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
        return http
            .authorizeExchange()

        // ...
    }

```

First, by calling `authorizeExchange()` method to expose [AuthorizeExchangeSpec](#)  lets us proceed with authentication details. With this, we can apply a matcher and permission model to our endpoints. For this example, we want to open several endpoints to everyone. This is where `permitAll()` can be applied to a multi-argument `pathMatchers()` expression.

SecurityConfiguration.java:

```

        .pathMatchers("/login",
            "/bye",
            "/favicon.ico",

```

```
"/images/**")
.permitAll()
```

For this example, we want every endpoint that's not publicly available to require a logged-in user. Wire in a [PathMatcher](#) for all other URL's, and apply the `authenticated()` operator to whatever matches.

SecurityConfiguration.java:

```
.pathMatchers("/**")
.authenticated()
.and()
```

### ### CSRF Configuration

Another component for configuring `SecurityWebFilterChain` is the [csrfSpec](#) enabled by calling `csrf()` method. This lets us configure [CSRF](#) tokens and request matchers, or exclude CSRF entirely.

In this demo we will use the default options, so `'_csrf'` becomes our parameter name, and `'X-CSRF-TOKEN'` is our header.

SecurityConfiguration.java:

```
.csrf()
    //csrfTokenRepository(customCsrfTokenRepository)
    //requireCsrfProtectionMatcher(customCsrfMatcher)
.and()
```

Note: Currently supported token repository exists are:



Class	Function
<a href="#">WebSessionServerCsrfTokenRepository</a>	Store CSRF token in a Web Session
<a href="#">CookieServerCsrfTokenRepository</a> 	Store CSRF token in custom cookie

Later, we will customize how CSRF tokens get included to our web page through custom filtering.

### ### Customizing Login/logout

Spring Security provides login/logout pages on demand whenever one is not already configured. This is provided by the [LoginPageGeneratingWebFilter](#) and [LogoutPageGeneratingWebFilter](#) that get wired in if no login/logout page was specified.

To override this, expose [FormLoginSpec](#) by calling `HttpServerSecurity's` `formLogin()` method. We can then issue the path to our custom login page and declare form-login success/error handlers. We want to redirect successes to the home page by using

[RedirectServerAuthenticationSuccessHandler](#) . Then For logout successes, we'll send the user to the "/bye" endpoint by configuring the [RedirectServerLogoutSuccessHandler](#)  in a separate method since it's constructor doesn't support parameters.

SecurityConfiguration.java:


```

        .and()
        .formLogin()
            .loginPage("/login")
            .authenticationSuccessHandler(new RedirectServerAuthenticationSuc
        .and()
        .logout()
            .logoutUrl("/logout")
            .logoutSuccessHandler(logoutSuccessHandler("/bye"))
        .and()
        .build();
    }




    public ServerLogoutSuccessHandler logoutSuccessHandler(String uri) {
        RedirectServerLogoutSuccessHandler successHandler = new RedirectServerLogouts
        successHandler.setLogoutSuccessUrl(URI.create(uri));
        return successHandler;
    }
}




```

Both handlers do similar things - namely redirect on success. Explicitly constructing the `logoutSuccessHandler` since its constructor only allows no-args.

Next, we will look at how user/pass pairs are authenticated. This is done by a subclass of [ReactiveAuthenticationManager](#) .

### ## Authenticating Users

[UserDetailsRepositoryReactiveAuthenticationManager](#)  bean is provided automatically if there are no other configured [ReactiveAuthenticationManager](#)  `@Bean` definitions. This authentication manager defers principal/credential operations to a [ReactiveUserDetailsService](#)  implementation.

Spring comes with ready-made implementations for storing and looking up users in the [MapReactiveUserDetailsService](#) . We'll complete this section using the map reactive implementation, and by having our users come from the handy [User](#)  object since no other details are necessary for [customizing the user](#) .

UserDetailBeans.java:

```

@Configuration
public class UserDetailServiceBeans {

```

```

@Bean
public MapReactiveUserDetailsService mapReactiveUserDetailsService() {
    return new MapReactiveUserDetailsService(users);
}

private static final PasswordEncoder pw = PasswordEncoderFactories.createDele

private static UserDetails user(String u, String... roles) {

    return User
        .withUsername(u)
        .passwordEncoder(pw::encode)
        .password("pw")
        .authorities(roles)
        .build();
}

private static final Collection<UserDetails> users = new ArrayList<>(
    Arrays.asList(
        user("thor", "ROLE_USER"),
        user("loki", "ROLE_USER"),
        user("odin", "ROLE_ADMIN", "ROLE_USER")
    ));
}

```

## ## View Configuration with Mustache

To enable [Mustache](#)  Views, we need to wire in an appropriate [ViewResolver](#)  so view names are rendered with the Mustache template View.

First, include the `spring-boot-starter-mustache` dependency in your pom.xml.

pom.xml:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mustache</artifactId>
</dependency>

```

Now, create a [WebFluxConfigurer]() to enable Mustache view rendering.

webConfig.java:

```

@Configuration
class webConfig implements WebFluxConfigurer {

```

```

private final MustacheViewResolver resolver;

// The resolver is provided by MustacheAutoConfiguration class
webConfig(MustacheViewResolver resolver) {
    this.resolver = resolver;
}

// order matters; cache will find first and render.
@Override
public void configureViewResolvers(ViewResolverRegistry registry) {
    registry.viewResolver(resolver);
}
}

```

This configuration will populate a [MustacheViewResolver](#)  into the [ViewResolverRegistry](#) .

### ### Mustache Web Content

Mustache lets us use includes for re-usable content. We will create 3 common fragments for our views.

frag/header.html:

```

<!doctype html>
<html lang="en">
<body>

```

frag/footer.html:

```

</body>
</html>

```

frag/logout.html:

```

<form class="form-inline" action="/logout" method="post">
    <input type="hidden" name="_csrf" value="{{_csrf.token}}" />
    <button class="btn btn-lg btn-primary btn-block" type="submit">Escape!</button>
</form>

```

Next we can define the meat of our content. We just need a login, game, and logout page.

game.html:

```

{{>frag/header}}
<h1>West of House</h1>
<div>Hello, {{user.username}}. You are standing in an open field west of a white house, w
    There is a small mailbox here.</div>
<br/>

    <div class="row">
        <div class="col-md-6">
            <form class="form-inline" action="/login" method="post">
                <div class="form-group">
                    <label for="username">What is your name?</label>
                    <input type="text" name="username" id="username" class="form-cont
                </div>
                <div class="form-group">
                    <label for="password">What is the passcode?</label>
                    <input type="password" name="password" id="password" class="form-
                </div>
                <br/>
                <input type="hidden" name="_csrf" value="{{_csrf.token}}" />
                <button class="btn btn-lg btn-primary btn-block" type="submit">who R
            </form>
        </div>
    </div>
</div>
{{>frag/footer}}

```

bye.html:

```

{{>frag/header}}
<h1>Leaving the Great Underground Empire</h1>
<div>
    See you later, dungeon-master!<br/>
</div>
<br/>
{{>frag/footer}}

```


### ### Configuring Mustache Behaviour



To let Mustache view resolver know where to find our templates, how to handle view objects, we set the specific options for this. Using `expose-request-attributes` allows us to access our request's view attributes from within the template.

application.properties

```
spring.mustache.prefix=classpath:/templates/mustache/
spring.mustache.suffix=.html
spring.mustache.expose-request-attributes=true
```

### ## Routing to Views

We need to wire up our views with routing logic, so let's add this with the functional style [RouterFunction](#) .

We need a way to display icons, so first we will wire in a 'favicon.ico' route, and send it to a [ClassPathResource](#)  resource for classpath resolution. Alternately, we could imply that the file exists on the local Filesystem by using a [FileSystemResource](#) .

WebRoutes.java:

```
@Component
public class WebRoutes {

    @Bean
    RouterFunction<?> iconResources() {
        return RouterFunctions
            .resources("/favicon.**", new ClassPathResource("images/favicon.ico"))
    }
}
```

Next we will have a route to the login page. Since we already overrode the default location in `ServerHttpSecurity`, we must provide the route to our new login page. Also included in this example is our logout landing page. It can be any URL you select, for this example we have a single page to tell the user 'goodbye'.

WebRoutes.java:

```
@Bean
RouterFunction<?> viewRoutes() {
    return RouterFunctions
        .route(RequestPredicates.GET("/login"),
            req -> ServerResponse
                .ok()
                .render("login-form",
                    req.exchange().getAttributes())
        )
}
```

```

        .andRoute(RequestPredicates.GET("/bye"),
            req -> ServerResponse.ok().render("bye")
        )
    )
}

```

## ## Route Filtering & CSRF

During `ServerHttpSecurity` configuration, we added the line for `csrf()` that has the effect of implementing request/response filtering. The effect of this Filter - [CsrfWebFilter](#) is to create, store and validate csrf tokens where seen or needed. We can expose the CSRF token by including the form entry `'_csrf'` and accessing our view model to extract the token value. Remember that the parameter and headers names may be changed by additional configuration to the [CsrfSpec](#) during configuration.

This filter allows us to access the CSRF token per request, or as needed if you are explicit about the paths that must be matched. CSRF tokens are stored as the classname `org.springframework.security.web.csrf` within the request attributes map. We can access this and determine how to expose it to the view by calling the token `parameterName()` method (then placing the token as that name in the model map).

```

        .filter((req, resHandler) -> {
            req.exchange()
                .getAttributeOrDefault(
                    CsrfToken.class.getName(),
                    Mono.empty().ofType(CsrfToken.class)
                )
                .flatMap(csrfToken -> {
                    req.exchange()
                        .getAttributes()
                        .put(csrfToken.getParameterName(), csrfToken)
                    return resHandler.handle(req);
                })
        });
    }
}

```

## ## Data Model for Views

The last route will require some information about the user logged in. We can construct the model for our mustache template by including a `Map<String, Object>` as the second argument to the `render()` method.

To get to the logged-in user, we get the principal from the [ServerRequest](#) object, cast it to its value type, and inject it into request attributes Map under the `'user'` key. The attributes map (model object) is then passed in to the `render()` function.

WebRoutes.java:



```

.andRoute(RequestPredicates.GET("/"),
    req -> req.principal()
        .ofType(Authentication.class)
        .flatMap(auth -> {
            User user = User.class.cast(auth.getPrincipal());
            req.exchange()
                .getAttributes()
                .putAll(Collections.singletonMap("user", user));
            return ServerResponse.ok().render("game",
                req.exchange().getAttributes());
        })
)

```

## ## Application execution Entry

This simple app requires no additional configuration beyond [EnableWebFlux](#) and [SpringBootApplication](#) annotations.

App.java:

```

@SpringBootApplication
@EnableWebFlux
public class FormLoginApp {

    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(FormLoginApp.class);
        app.run();
    }
}

```

Execute the application by running it in your IDE or by executing the following command in a bash window.

```
mvn spring-boot:run
```

## Conclusion

=====

This simple web was made to demonstrate key concepts in obtaining successful authentication from a user that is browser-bound. We looked at wiring up CSRF, Mustache views, login/logout customization, and Routing/Filtering in the webFlux environment.

-----

Published by Mario Gray 25 Jul, 2018 in [appsec](#), [reactive](#) and [security](#) and tagged [demo](#), [forms](#), [functional](#), [java](#), [spring](#) and [web](#) using 1412 words.

## Related Content

- [Setup and customize Authentication against a webFlux Application](#) - 4 minutes
- [Sending and consuming messages with Spring and Kafka](#) - 5 minutes
- [Reactive Websocket Client with Spring](#) - 3 minutes
- [Intro to RIFF Is For Functions](#) - 5 minutes
- [Configuring Authorization with Reactive Spring Security 5](#) - 3 minutes
- [Spring Reactive WebSocket Hot Publisher](#) - 2 minutes
- [Spring Reactive WebSocket Cold Publisher](#) - 3 minutes

This page was generated using [After Dark](#)  for [Hugo](#) .