Secure REST Services Using Spring Security

by Mrabti Idriss MVB · Sep. 05, 14 · Integration Zone

Visually compose APIs with easy-to-use tooling. Learn how IBM API Connect provides near-universal access to data and services both on-premises and in the cloud, brought to you in partnership with IBM.

Overview :

Recently, I was working on a project which uses a REST services layer to communicate with the client application (GWT application). So I have spent a lot of to time to figure out how to secure the REST services with Spring Security. This article describes the solution I found, and I have implemented. I hope that this solution will be helpful to someone and will save a much valuable time.

The solution :

In a normal web application, whenever a secured resource is accessed Spring Security check the security context for the current user and will decide either to forward him to login page (if the user is not authenticated), or to forward him to the resource not authorised page (if he doesn't have the required permissions).

In our scenario this is different, because we don't have pages to forward to, we need to adapt and override Spring Security to communicate using HTTP protocols status only, below I liste the things to do to make Spring Security works best :

- The authentication is going to be managed by the normal form login, the only difference is that the response will be on JSON along with an HTTP status which can either code 200 (if the autentication passed) or code 401 (if the authentication failed) ;
- Override the **AuthenticationFailureHandler** to return the code 401 UNAUTHORIZED ;
- Override the **AuthenticationSuccessHandler** to return the code 20 OK, the body of the HTTP response contain the JSON data of the current authenticated user ;
- Override the **AuthenticationEntryPoint** to always return the code 401 UNAUTHORIZED. This will override the default behavior of Spring Security which is forwarding the user to the login page if he don't meet the security requirements, because on REST we don't have any login page ;

- Override the **LogoutSuccessHandler** to return the code 20 OK ;

Like a normal web application secured by Spring Security, before accessing a protected service, it is mandatory to first authenticate by submitting the password and username to the Login URL.

Note: The following solution requires Spring Security in version minimum 3.2.

Overriding the AuthenticationEntryPoint :

Class extends org.springframework.security.web.AuthenticationEntryPoint, and implements only one method, which sends response error (with 401 status code) in cause of unauthorized attempt.

```
@Component
public class HttpAuthenticationEntryPoint implements AuthenticationEntryPoint {
    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
            AuthenticationException authException) throws IOException {
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, authException.getMess
age());
    }
}
```

Overriding the AuthenticationSuccessHandler :

The AuthenticationSuccessHandler is responsible of what to do after a successful authentication, by default it will redirect to an URL, but in our case we want it to send an HTTP response with data.

```
@Component
public class AuthSuccessHandler extends SavedRequestAwareAuthenticationSuccessHandler
 {
    private static final Logger LOGGER = LoggerFactory.getLogger(AuthSuccessHandler.c
lass);

    private final ObjectMapper mapper;

    @Autowired
    AuthSuccessHandler(MappingJackson2HttpMessageConverter messageConverter) {
        this.mapper = messageConverter.getObjectMapper();
    }

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request, HttpServletRespon
se response,
            Authentication authentication) throws IOException, ServletException {
        response.setStatus(HttpServletResponse.SC_OK);

        NuvolaUserDetails userDetails = (NuvolaUserDetails) authentication.getPrincip
al();
```

```
        User user = userDetails.getUser();
        userDetails.setUser(user);

        LOGGER.info(userDetails.getUsername() + " got is connected ");

        PrintWriter writer = response.getWriter();
        mapper.writeValue(writer, user);
        writer.flush();
    }
}
```

Overriding the AuthenticationFailureHandler :

The AuthenticationFaillureHandler is responsible of what to after a failed authentication, by default it will redirect to the login page URL, but in our case we just want it to send an HTTP response with the 401 UNAUTHORIZED code.

```
@Component
public class AuthFailureHandler extends SimpleUrlAuthenticationFailureHandler {
    @Override
    public void onAuthenticationFailure(HttpServletRequest request, HttpServletRespon
se response,
            AuthenticationException exception) throws IOException, ServletException {
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        PrintWriter writer = response.getWriter();
        writer.write(exception.getMessage());
        writer.flush();
    }
}
```

Overriding the LogoutSuccessHandler :

The LogoutSuccessHandler decide what to do if the user logged out successfully, by default it will redirect to the login page URL, because we don't have that I did override it to return an HTTP response with the 20 OK code.

```
@Component
public class HttpLogoutSuccessHandler implements LogoutSuccessHandler {
    @Override
    public void onLogoutSuccess(HttpServletRequest request, HttpServletResponse respo
nse, Authentication authentication)
            throws IOException {
        response.setStatus(HttpServletResponse.SC_OK);
        response.getWriter().flush();
    }
}
```

Spring security configuration :

This is the final step, to put all what we did together, I prefer using the new way to configure Spring Security which is with Java no XML, but you can easily adapt this configuration to XML.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    private static final String LOGIN_PATH = ApiPaths.ROOT + ApiPaths.User.ROOT + Api
Paths.User.LOGIN;

    @Autowired
    private NuvolaUserDetailsService userDetailsService;
    @Autowired
    private HttpAuthenticationEntryPoint authenticationEntryPoint;
    @Autowired
    private AuthSuccessHandler authSuccessHandler;
    @Autowired
    private AuthFailureHandler authFailureHandler;
    @Autowired
    private HttpLogoutSuccessHandler logoutSuccessHandler;

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }

    @Bean
    @Override
    public UserDetailsService userDetailsServiceBean() throws Exception {
        return super.userDetailsServiceBean();
    }

    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvi
der();
        authenticationProvider.setUserDetailsService(userDetailsService);
        authenticationProvider.setPasswordEncoder(new ShaPasswordEncoder());

        return authenticationProvider;
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.authenticationProvider(authenticationProvider());
    }

    @Override
    protected AuthenticationManager authenticationManager() throws Exception {
        return super.authenticationManager();
    }

    @Override
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .authenticationProvider(authenticationProvider())
                .exceptionHandling()
                .authenticationEntryPoint(authenticationEntryPoint)
                .and()
                .formLogin()
                .permitAll()
                .loginProcessingUrl(LOGIN_PATH)
                .usernameParameter(USERNAME)
                .passwordParameter(PASSWORD)
                .successHandler(authSuccessHandler)
                .failureHandler(authFailureHandler)
                .and()
                .logout()
                .permitAll()
                .logoutRequestMatcher(new AntPathRequestMatcher(LOGIN_PATH, "DELETE")
)
                .logoutSuccessHandler(logoutSuccessHandler)
                .and()
                .sessionManagement()
                .maximumSessions(1);

        http.authorizeRequests().anyRequest().authenticated();
    }
}
```

This was a sneak peak at the overall configuration, I attached in this article a Github repository containing a sample project https://github.com/imrabti/gwtp-spring-security.

I hope this will help some of you developers struggling to figure out a solution, please feel free to ask any questions, or post any enhancements that can make this solution better.

Topics: JAVA, ENTERPRISE-INTEGRATION, SPRING, REST, SECURITY