Spring6 Web Security

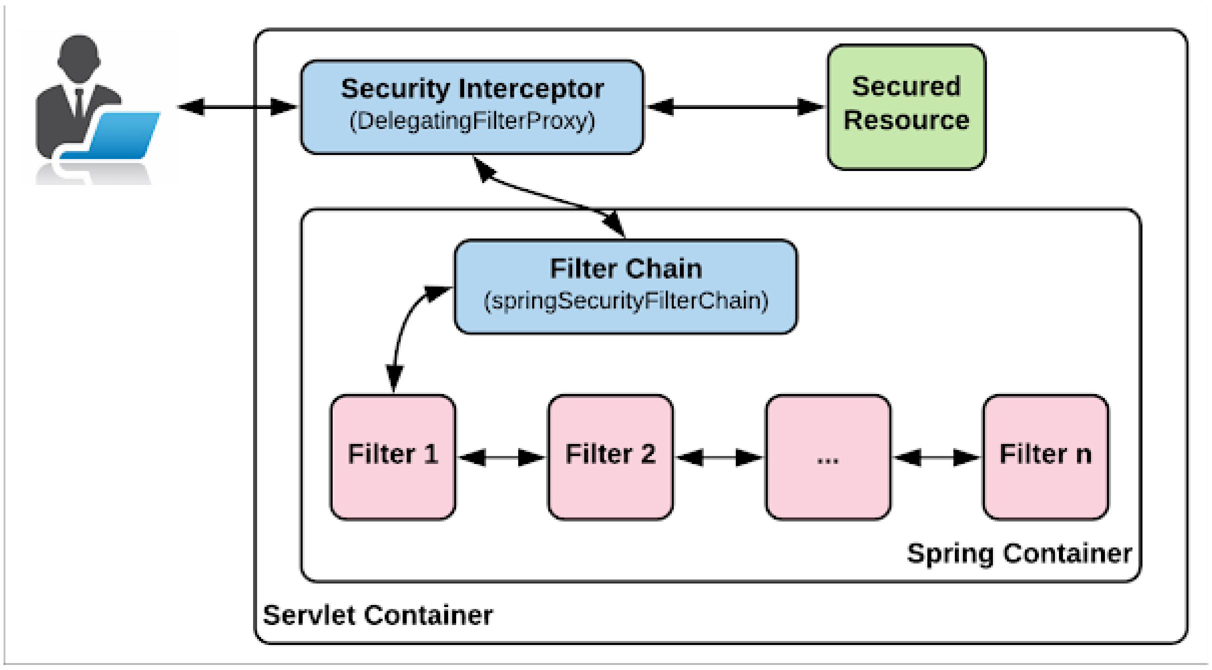
Autor : Hong Le Nguyen   Update : **05.2024**

Code to example on Github :
https://github.com/hong1234/spring-boot3-mvc-jdbc-restApi
https://github.com/hong1234/spring-boot3-mvc-jpa-restApi

How Spring Security works internally ?

An HTTP request is first passed through a chain of Spring Security filters for authentication, authorization and maintaining SecurityContext in HttpSession *before* it reaches the DispatcherServlet.

Forwarding HTTP requests from Servlet container to *security filters in Spring WebApplicationContext* is achieved through a (servlet container) filter called *DelegatingFilterProxy*, which is configured to intercept all requests by specifying a wildcard (*) in its url-pattern. It delegates HTTP requests to the *FilterChainProxy* (a bean named springSecurityFilterChain), which determines the filters to invoke based on the security configuration in beans of type *SecurityFilterChain.*



SecurityFilterChain

The bean of type SecurityFilterChain is used to implement the Spring *Security policy*. The security filters in the SecurityFilterChain are beans registered with FilterChainProxy. An application can have multiple SecurityFilterChain. FilterChainProxy uses the RequestMatcher interface on HttpServletRequest to determine *which chain needs to be called*.

The request is processed by *filters in the chain in a pre-defined order* so that authentication occurs before authorization. *How many filters a particular request will go through* depends upon your Spring security configuration. If you enable more functionalities, more filters will be employed.

You can even create your own *custom filters* and attach it into Spring security filter chain at some pre-defined position e.g. before filter X or after filter X.

HttpSecurity

Spring Security HttpSecurity bean gives us a handle on defining rules of the Security policy. With these settings, HttpSecurity is then used to render SecurityFilterChain through which all servlet requests will be routed.

A configuration example:

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {
        …
        @Bean
        public SecurityFilterChain defaultFilterChain(HttpSecurity http) throws Exception {
                http
                .authorizeHttpRequests(authorize -> authorize
                        .anyRequest().authenticated()
                )
                .formLogin(Customizer.withDefaults())
                .httpBasic(Customizer.withDefaults());
                return http.build();
        }

}
```

Method authorizeRequests(…) defines exactly how we will authorize requests.
The *formLogin* and *httpBasic* directives are switched on, enabling two standard authentication mechanisms.
The above configuration will result in the following *Filter ordering*:

Filter	Added by
UsernamePasswordAuthenticationFilter	HttpSecurity#formLogin (1)
BasicAuthenticationFilter	HttpSecurity#httpBasic (2)
AuthorizationFilter	HttpSecurity#authorizeHttpRequests (3)

Depending on the authentication method, the Authentication-filter (1) or (2) is invoked to authenticate the request. Then the AuthorizationFilter is invoked to authorize the request.

We can decompose the above chain into two chains, e.g. one for Web, the other for rest-API security

```
@Bean
@Order(1)
public SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
        http
        .securityMatcher("/api/**")
        .authorizeHttpRequests(authorize -> authorize
                .anyRequest().hasRole("ADMIN")
        )
        .httpBasic(withDefaults());
        return http.build();
}

@Bean
public SecurityFilterChain formLoginFilterChain(HttpSecurity http) throws Exception {
        http
        .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/login").permitAll()
                .anyRequest().authenticated()
        )
        .formLogin(withDefaults());
        return http.build();
}
```

Using what we've learned to write down *a suitable Security policy* for our Book Reviews *RESTful-API* project

Everyone must log in to access anything
The authenticated users can query everything
Only AUTOR users can add, update, delete books
USER users can add, update, delete reviews
Any other forms of access will be disabled
These rules should apply to command-line interactions
Basic Authentication is the way of authentication when accessing API

We implement the politic with a SecurityFilterChain bean

```
@Configuration
public class WebSecurityConfig {
        ...

        @Bean
        SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
                http
                .csrf((csrf) -> csrf.disable())
                .sessionManagement((session) -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .securityMatcher("/api/**")
                .authorizeHttpRequests(authorize -> authorize
                        .requestMatchers(HttpMethod.GET, "/api/**").authenticated()
                        .requestMatchers("/api/books/**").hasRole("AUTOR")
                        .requestMatchers("/api/reviews/**").hasRole("USER")
                        .anyRequest().denyAll()
                )
                .httpBasic(basic -> basic.authenticationEntryPoint(authEntryPoint))
                .exceptionHandling(customizer -> customizer.accessDeniedHandler(accessDeniedHandler));
                return http.build();
        }

}
```

The preceding security policy can be described as follows:

.csrf((csrf) -> csrf.disable())
CSRF protections aren't needed when doing API calls in a stateless scenario.

SessionCreationPolicy.STATELESS
Stateless authentication doesn't require you to track sessions on the server.

The http.securityMatcher states that this HttpSecurity is applicable only to URLs that start with "/api"

Method authorizeHttpRequests, signaling checks for url-/user-role- matching (access rules)

.requestMatchers(HttpMethod.GET, "/api/**").authenticated()
The rule grants query to the base URL "/api/**" to anyone who is authenticated

.requestMatchers("/api/books/**").hasRole("AUTOR")
The rule restricts POST, PUT, DELETE access to url "/api/books/**" only to authenticated users who also have the AUTOR role.

.requestMatchers("/api/reviews/**").hasRole("USER")
The rule restricts POST, PUT, DELETE access to url "/api/reviews/**" only to authenticated users who also have the USER role.

.anyRequest().denyAll()
Any other forms of access will be disabled

.httpBasic(basic -> basic.authenticationEntryPoint(authEntryPoint))
The basic authentication is enabled

How (Spring Security) Username/Password Authentication works internally ?

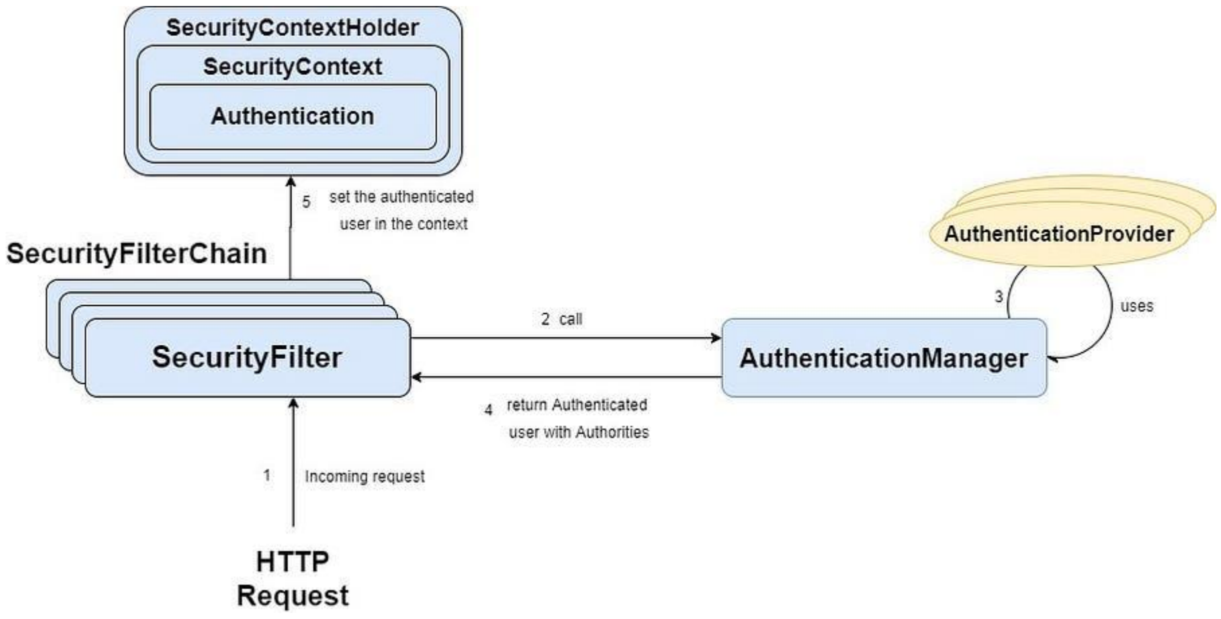
Authentication is the process of verifying a user's credentials and ensuring their validity. Authorization depends on authentication. A user has to be first authenticated, for authorization to take place.

Spring Security supports many authentication mechanisms (Username and Password, OAuth 2.0 Login, SAML 2.0 Login ...). Here we take a closer look at Username/Password Authentication.

When an authentication attempt is made, the *Authentication-filter* e.g UsernamePasswordAuthenticationFilter creates a *UsernamePasswordAuthenticationToken* by extracting the username and password from the HttpServletRequest. Next, the Token is passed to an *AuthenticationManager* to be authenticated by an appropriate *AuthenticationProvider* e.g DaoAuthenticationProvider.

The AuthenticationProvider retrieves user details from *UserDetailsService*. The loadUserByUsername method of the UserDetailsService returns *UserDetails* object that contains user data. If no user is found with the given username, *UsernameNotFoundException* is thrown.

On successful authentication, a fully authenticated *Authentication* object is returned and *SecurityContext* is updated with the currently authenticated user. If authentication fails, *AuthenticationException* is thrown.



BasicAuthenticationFilter

this filter is responsible for processing any request that has a HTTP request header of Authorization with an authentication scheme of *Basic* and *a Base64-encoded username:password token*. For example, to authenticate user "Aladdin" with password "open sesame" the following header would be presented:

Authorization: *Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==*

If authentication is successful, the resulting Authentication object will be placed into the *SecurityContextHolder*. If authentication fails and ignoreFailure is false (the default), an *AuthenticationEntryPoint* implementation is called. Basic authentication is simple and widely deployed. It still transmits *a password in clear text* and as such is undesirable in many situations.

UsernamePasswordAuthenticationFilter

Processes an authentication form submission. Login forms must present two parameters to this filter: a username and password. This filter by default responds to the URL /login.

AuthenticationManager

as a coordinator where you can register multiple authentication providers, and based on the request type, it will deliver an authentication request to the correct provider.

DaoAuthenticationProvider

an AuthenticationProvider implementation that retrieves user details from a UserDetailsService. This Provider is used in Username-Password Authentication as formLogin, HttpBasic.

UserDetailsService

Spring Security's interface for defining *a source of users*. Known Subinterface UserDetailsManager. Known Implementing Classes: *JdbcUserDetailsManager*, *InMemoryUserDetailsManager*.

UserDetailsService Implementation

Solution #1 : Custom Implementation of Method : UserDetails loadUserByUsername(String username)

```
UserAccount implements UserDetails {
        private Long id;
        private String username;
        private String password;
        private List<GrantedAuthority> authorities = new ArrayList<>();

UserRepository implements Repository<UserAccount, Long> {
        UserAccount findByUsername(String username);

@Bean
UserDetailsService userService(UserRepository repo) {
        return username -> repo.findByUsername(username);
}

// users init
repository.save(new UserAccount("user", "user", "ROLE_USER"));
repository.save(new UserAccount("admin", "admin", "ROLE_USER", "ROLE_ADMIN"));
```

Solution #2 : Using (build-in) Classes JdbcUserDetailsManager, InMemoryUserDetailsManager

--- Using Class JdbcUserDetailsManager ---

```
// src/main/resources/schema.sql
create table users(
        username varchar_ignorecase(50) not null primary key,
        password varchar_ignorecase(500) not null,
        enabled boolean not null
);
create table authorities (
        username varchar_ignorecase(50) not null,
        authority varchar_ignorecase(50) not null,
        constraint fk_authorities_users foreign key(username) references users(username)
);
create unique index ix_auth_username on authorities (username,authority);
```

```
@Bean
UserDetailsService userService(DataSource dataSource) {
    UserDetailsManager userManager = new JdbcUserDetailsManager(dataSource);
        return userManager;
}

// users init
import org.springframework.security.core.userdetails.User;

User.UserBuilder userBuilder = User.builder().passwordEncoder(passwordEncoder()::encode);
UserDetails hong  = userBuilder.username("hong").password("hong").roles("USER").build();
UserDetails admin = userBuilder.username("admin").password("admin").roles("USER", "AUTOR",
"ADMIN").build();
```

--- Using Class InMemoryUserDetailsManager ---

```
@Bean
public UserDetailsService userDetailsService(){
        UserDetails hong   = …;
        UserDetails admin = …;
        return new InMemoryUserDetailsManager(hong, admin);
}
```

Handling Security Exceptions

ExceptionTranslationFilter

handles any *AccessDeniedException* and *AuthenticationException* thrown within the filter chain. This filter is necessary because it provides the bridge between Java exceptions and HTTP responses. It is solely concerned with maintaining the user interface.

If an AuthenticationException is detected, the filter will launch the *AuthenticationEntryPoint*.

If an AccessDeniedException is detected, the filter will determine whether or not the user is an anonymous user. If they are an anonymous user, the AuthenticationEntryPoint will be launched. If they are not an anonymous user, the filter will delegate to the *AccessDeniedHandler*.

AuthenticationEntryPoint

is used by the ExceptionTranslationFilter to commence authentication. By default, the BasicAuthenticationEntryPoint returns a full page for a 401 Unauthorized response back to the client.

To customize the default authentication error page used by basic auth, we can implement the AuthenticationEntryPoint interface.

```
@Component
public class CustomAuthenticationEntryPoint implements AuthenticationEntryPoint {
  @Override
  public void commence(HttpServletRequest request, HttpServletResponse response,
                              AuthenticationException authException) throws IOException, ServletException {
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
         response.getWriter().write("Bad Credentiales.");
  }
```

AccessDeniedHandler

is used by the ExceptionTranslationFilter to commence access refusal. To customize the access refusal used by basic auth, we can implement the AccessDeniedHandler interface.

```
@Component
public class CustomAccessDeniedHandler implements AccessDeniedHandler {
   @Override
   public void handle(HttpServletRequest request, HttpServletResponse response,
                 AccessDeniedException accessDeniedException) throws IOException, ServletException {
         response.setStatus(HttpServletResponse.SC_FORBIDDEN);
         response.getWriter().write("Access Denied. You do not have privileges to access this resource.");
   }
```

We register the above merchants in basic authentication as following

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

      @Autowired
      @Qualifier("customAuthenticationEntryPoint")
      private AuthenticationEntryPoint authEntryPoint;

      @Autowired
      @Qualifier("customAccessDeniedHandler")
      private AccessDeniedHandler accessDeniedHandler;
      ...

      @Bean
      SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
            ...
            http.
            httpBasic(basic -> basic.authenticationEntryPoint(authEntryPoint))
            .exceptionHandling(customizer -> customizer.accessDeniedHandler(accessDeniedHandler))
            ;
```

Handling security exceptions with @ExceptionHandler and @ControllerAdvice

This approach allows us to use exactly the same exception handling techniques but in a cleaner and much better way in the controller advice with methods annotated with @ExceptionHandler.

Spring security core exceptions such as AuthenticationException and AccessDeniedException are runtime exceptions. Since these exceptions are thrown by the authentication filters *before* invoking the controller methods, @ControllerAdvice won't be able to catch these exceptions. To handle these exceptions at a global level via @ExceptionHandler and @ControllerAdvice, we need delegate the exception to *HandlerExceptionResolver*.

A new custom implementation of AccessDeniedHandler:

```
@Component
public class DelegatedAccessDeniedHandler implements AccessDeniedHandler {
      ...
      @Override
      public void handle(HttpServletRequest request, HttpServletResponse response,
            AccessDeniedException accessDeniedException) throws IOException, ServletException {
            resolver.resolveException(request, response, null, accessDeniedException);
}
```

A new custom implementation of AuthenticationEntryPoint:

```
@Component
public class DelegatedAuthenticationEntryPoint implements AuthenticationEntryPoint {
   @Autowired
   @Qualifier("handlerExceptionResolver")
   private HandlerExceptionResolver resolver;

   @Override
   public void commence(HttpServletRequest request, HttpServletResponse response,
                        AuthenticationException authException) throws IOException, ServletException {
          resolver.resolveException(request, response, null, authException);
   }
}
```

Now we can handle the exceptions in the class annotated with @RestControllerAdvice.

```
@RestControllerAdvice
public class GlobalExceptionHandler {

        @ExceptionHandler(AuthenticationException.class)
        public ErrorDetails handleAuthenticationException(Exception e) {
                 ...
                errorDetails.setMessage(e.getMessage());
                return errorDetails;
        }

        @ExceptionHandler(AccessDeniedException.class)
        public ErrorDetails forbidden(Exception e) {
                ...
                errorDetails.setMessage(e.getMessage());
                return errorDetails;
        }
```

Spring Web Security Configuration

To configure Spring Security for Spring Web Mvc the developer must take care of three things.

declare the security filter for the application
define the Spring Security context
configure authentication and authorization

if we use Spring MVC, our SecurityWebApplicationInitializer could look something like the following:

```
public class SecurityWebApplicationInitializer extends AbstractSecurityWebApplicationInitializer {}
```

This only registers the *DelegatingFilterProxy* (a servlet-filter) with Servlet Container for every URL in your application.

After that, we need to ensure that WebSecurityConfig was loaded in our existing ApplicationInitializer. For example, if we use Spring MVC it is added in the getServletConfigClasses():

```
public class MvcWebApplicationInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
        ...
        @Override
        protected Class<?>[] getServletConfigClasses() {
                return new Class[] { WebSecurityConfig.class, WebMvcConfig.class };
        }
```

And cofiguring the Web Security

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

    @Bean
    public PasswordEncoder encoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public UserDetailsService userDetailsService(DataSource dataSource) {
        UserDetailsManager userManager = new JdbcUserDetailsManager(dataSource);
        return userManager;
    }

    @Bean
    @Order(1)
    public SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
        …
        return http.build();
    }
```

Spring Boot Default Security Configuration

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
class SpringBootWebSecurityConfiguration {

    @ConditionalOnDefaultWebSecurity
    static class SecurityFilterChainConfiguration {
        @Bean
        @Order(SecurityProperties.BASIC_AUTH_ORDER)
        SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
            http.authorizeHttpRequests((requests) -> requests
                .anyRequest().authenticated()
            );
            http.formLogin(withDefaults());
            http.httpBasic(withDefaults());
            return http.build();
        }
    }
```