Spring6 Securing Web Applications

written by Hong Le Nguyen, last update: 01.2025

Code to example on Github :
https://github.com/hong1234/Spring6Boot3-JWT-Authen-APIh
https://github.com/hong1234/spring-boot3-mvc-jdbc-restApi
https://github.com/hong1234/spring-boot3-mvc-jpa-restApi

1) Spring Security integration in Spring MVC

Terminology ---
Filters : all filters implement the jakarta.servlet.Filter interface
Servlet filters : filters managed by Servlet Container
FilterChain : sequence of Servlet filters. There is a filter chain in an application
Spring Security filters : filters are also beans managed by Spring Container
SecurityFilterChain : a sequence of some Spring Security filters
There are *1-n SecurityFilterChains* in an application.

The HTTP request is first passed through filters (of the FilterChain including a appropriate SecurityFilterChain) for authentication, authorization, and protection against common attacks *before* it reaches the DispatcherServlet. Figure 1.

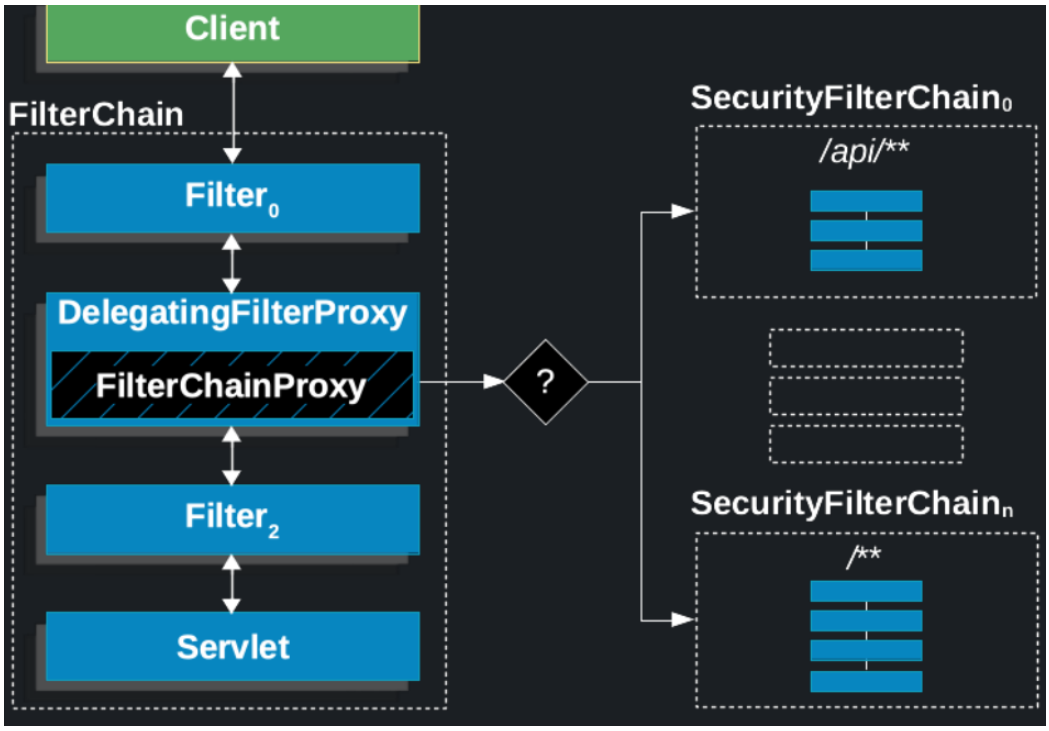


Figure 1. Spring Security Filter architecture

DelegatingFilterProxy Filter

Since the Spring Security filters are not registered via Servlet Container standards (configured in e.g web.xml), the Servlet Container is not aware of them. But Spring Security filters should be injected into the filter chain of the Java Servlet API.

Spring provides a servlet filter DelegatingFilterProxy that acts as a bridge between the Servlet Container and the Spring Application Context. Every request will be going through this filter and it will delegate the request to the FilterChainProxy bean named "springSecurityFilterChain". Figure 2.

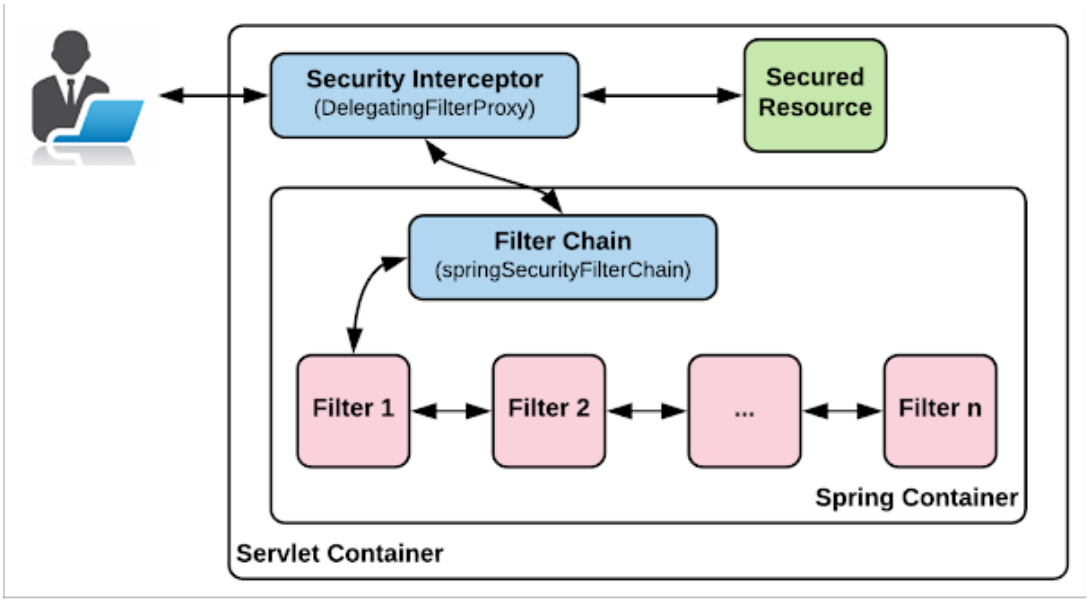


Figure 2.

FilterChainProxy Filter

The filter bean provided by Spring Security, is entry point that enables spring security for a web application.

FilterChainProxy is a GenericFilterBean *that manages all the SecurityFilterChain injected into the Spring IoC container*. There could be multiple SecurityFilterChains present.

FilterChainProxy can determine which SecurityFilterChain a particular request will go through based on matching between request path and url-pattern configured for SecurityFilterChain.

If you inspect the FilterChainProxy class, you'd notice how it invokes the security filter chains and in turn trigger the stack of security filters. When a request HttpServletRequest enters SecurityFilterChain, the matches method is used to determine whether the conditions are met to enter the filter chain.

```
// org.springframework.security.web.FilterChainProxy class
public class FilterChainProxy extends GenericFilterBean {
   …
   private List<SecurityFilterChain> filterChains;
   public FilterChainProxy(List<SecurityFilterChain> filterChains) { this.filterChains = filterChains; }

   private List<Filter> getFilters(HttpServletRequest request) {
      …
      for (SecurityFilterChain chain : this.filterChains) {
         if (chain.matches(request)) {
            return chain.getFilters();
         }
      }
      return null;
   }
```

FilterChainProxy is configured whenever we use the @EnableWebSecurity annotation in the security config.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

By Spring boot is the class
// org.springframework.boot.autoconfigure.security.servlet.SpringBootWebSecurityConfiguration

SecurityFilterChain

The security filter chain is a sequence of filters that Spring Security applies to incoming HTTP request. These filters work together to perform various security-related tasks.

```
public interface SecurityFilterChain {
   boolean matches(HttpServletRequest request);
   List<Filter> getFilters();
}
```

HttpSecurity
is a build class, and its mission is to build a SecurityFilterChain. *HttpSecurity bean gives us a handle on defining rules of the Security policy.*

The Spring Security Default configuration

The auto-configuration SpringBootWebSecurityConfiguration class provides a default set of Spring Security configurations for Spring Boot applications.

```
class SpringBootWebSecurityConfiguration {
   ...
   @Bean
   SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
      http.authorizeHttpRequests(authorize -> authorize
         .anyRequest().authenticated()
      );
      http.formLogin(Customizer.withDefaults());
      http.httpBasic(Customizer.withDefaults());
      return http.build();
   }
```

The configuration here is that all requests must be initiated by an authenticated user, with form login and Http Basic Authentication enabled.

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.
The request is processed by filters in a pre-defined order so that authentication occurs before authorization.

BasicAuthenticationFilter

Basic authentication is simple and widely deployed. It still transmits a password in clear text and as such is undesirable in many situations.

The BasicAuthenticationFilter 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.

AuthorizationFilter

The final decision about whether a request is allowed or denied is made by the AuthorizationFilter class, which is the last filter in the Spring Security filter chain. Authorization decision is made for a Authentication object, which is stored in the SecurityContextHolder class.

Security Filter Chain Configuration

We can modify the default Spring Security filter chain by providing *a custom configuration*.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

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

An example, 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

```
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

We can configure Multiple Filter Chains for application, 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();
}
```

Custom Security Filter

Custom filters can be implemented to handle specific security requirements not covered by the default filters. Custom filters are created by implementing the interface jakarta.servlet.Filter

```
import jakarta.servlet.Filter;
import jakarta.servlet.FilterChain;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
```

```
public class CustomFilter implements Filter {
  ...
  @Override
  public void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
                                                                    throws ServletException, IOException {
    // filter logic
    ...

   // call the next filter
    chain.doFilter(request, response);
  }
}
```

Each filter in the chain processes the request and response before passing it to the next filter, forming a chain of responsibilities.

To make sure that our filter gets invoked only once for every request. We create class CustomFilter that extends a special filter OncePerRequestFilter.

```
import org.springframework.web.filter.OncePerRequestFilter;

public class CustomFilter extends OncePerRequestFilter {
  @Override
  protected void doFilterInternal(
                    ServletRequest request, ServletResponse response, FilterChain filterChain) {
```

Adding Filters to the Security Filter Chain

Custom filters can be added to the Spring Security filter chain at specific positions relative to existing filters. Methods like addFilterBefore(), addFilterAfter(), and addFilterAt() are used to specify the position of the custom filter in the chain.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .addFilterBefore(new CustomFilter(), UsernamePasswordAuthenticationFilter.class)
        .addFilterAfter(new LoggingFilter(), UsernamePasswordAuthenticationFilter.class)

        // In some cases, you might want to replace an existing filter with a custom one.
        // For example, you might want to replace the default BasicAuthenticationFilter
        .addFilterAt(new CustomAuthenticationFilter(), BasicAuthenticationFilter.class)
```

2) Authentication

Authentication is the process of verifying who user is. In a Spring Boot application, this involves checking user's credentials (like a username and password) against the stored data, and if they match, the user is granted access to the system. A user has to be first authenticated, for authorization to take place.

Spring Security supports many *authentication mechanisms* (Username-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* creates a *Authentication object* by extracting the username and password from the request. Next, the Authentication object is passed to an *AuthenticationManager* to be authenticated by an appropriate *AuthenticationProvider*. Figure 3.

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.

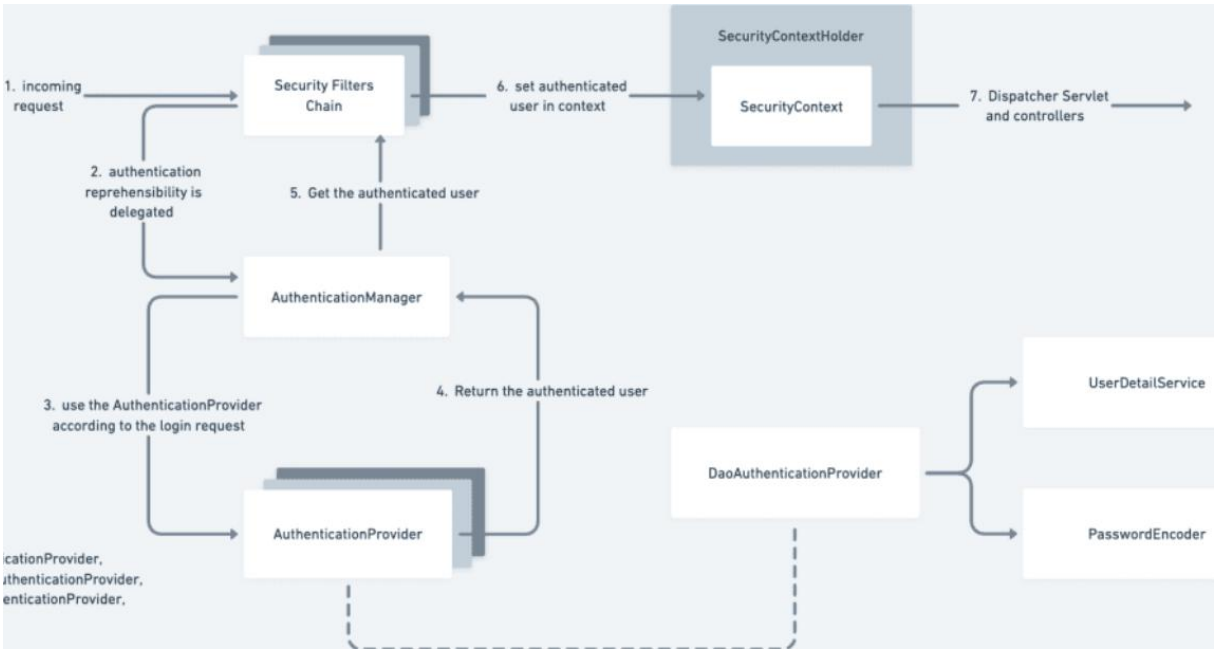


Figure 3.

```
public class CustomAuthenticationFilter extends OncePerRequestFilter {
  …
  @Override
  protected final void doFilterInternal(
                    HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) {
    // Authentication object create
    Authentication authentication = getAuthenticationConverter().convert(request);

    try {
      Authentication existingAuthentication =SecurityContextHolder.getContext().getAuthentication();

      // Do not attempt to authenticate if request is already authenticated
      if (existingAuthentication != null && existingAuthentication.isAuthenticated()) {
        filterChain.doFilter(request, response);
        return;
      }

      // Perform the authentication
      Authentication populatedAuthentication = authenticationManager.authenticate(authentication);

      // and set it in the security context
      SecurityContextHolder.getContext().setAuthentication(populatedAuthentication);

    } catch (Exception e) {
      SecurityContextHolder.getContext().setAuthentication(null);
    }

    filterChain.doFilter(request, response);    // Pass the control to the next filter
  }
```

Authentication object

The Authentication interface serves two main purposes within Spring Security:

An input to AuthenticationManager to provide the credentials a user has provided to authenticate. When used in this scenario, isAuthenticated() returns false.

```
Authentication authentication = new UsernamePasswordAuthenticationToken(username, password);
```

Represent *the currently authenticated user* after perform the authentication and set it in the security context.

```
Authentication populatedAuthentication = authenticationManager.authenticate(authentication);
```

Security Context

In order to grant access to a given resource in the AuthorizationFilter class, the SecurityContextHolder class must be populated with the Authentication object. For that reason, one of the security filters in the Spring Security filter chain must set the Authentication object in the SecurityContextHolder class.

```
SecurityContextHolder.getContext().setAuthentication(populatedAuthentication);
```

Figure 4.

To obtain information about the authenticated user, access the SecurityContextHolder.

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String username = authentication.getName();
Object principal = authentication.getPrincipal();
Collection<? extends GrantedAuthority> authorities = authentication.getAuthorities();
```

AuthenticationManager

Responsible for authenticating a user based on their credentials. The *ProviderManager* is the default implementation. It manages multiple AuthenticationProvider.

At authentication attempt, ProviderManager iterates through a list of configured AuthenticationProviders, delegating the authentication process to each one in order until one successfully authenticates the user or all providers have been tried.

The configuration looks like this

```
@Configuration
public class AuthenticationConfig {
   private final AuthenticationProvider authenticationProvider;
```

```
    @Bean
    public AuthenticationManager authenticationManager() {
      return new ProviderManager(List.of(authenticationProvider));
    }
}
```

AuthenticationProvider

Authentication providers provide logic for authentication.
The method authenticate() accepts an Authentication object containing credentials (password, token etc) and return another Authentication object that's fully populated with user information and authorities of the user.

```
interface AuthenticationProvider {
  Authentication authenticate(Authentication authentication) throws AuthenticationException { ... }
```

Custom authentication logic can be implemented by creating a class that implements the AuthenticationProvider interface.

```
class CustomAuthenProvider implements AuthenticationProvider {
  @Override
  public Authentication authenticate(Authentication authentication) throws AuthenticationException {
    …
```

You can register the CustomAuthenProvider with the AuthenticationManager as follows

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration {
  …
  private final AuthenticationProvider authenProvider;

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
    .authenticationProvider(authenProvider)
    …
  }
```

Common Spring implementations include
DaoAuthenticationProvider for database-backed authentication and
JwtAuthenticationProvider for JWT-based authentication

DaoAuthenticationProvider

An AuthenticationProvider implementation is used in *Username-Password Authentication* as formLogin, HttpBasic, … This Provider needs *UserDetailsService* (load the user from DB and set the *UserDetails*) and *PasswordEncoder* (password will be saved in DB after Encoding)

```
@Configuration
public class SecurityConfiguration {

  @Bean
  UserDetailsService userDetailsService() {
    return username -> userRepository.findByEmail(username)
        .orElseThrow(() -> new UsernameNotFoundException("User not found"));
  }
```

```
@Bean
BCryptPasswordEncoder passwordEncoder() {
   return new BCryptPasswordEncoder();
}

@Bean
AuthenticationProvider authenticationProvider() {
   DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();
   authProvider.setUserDetailsService(userDetailsService());
   authProvider.setPasswordEncoder(passwordEncoder());
```

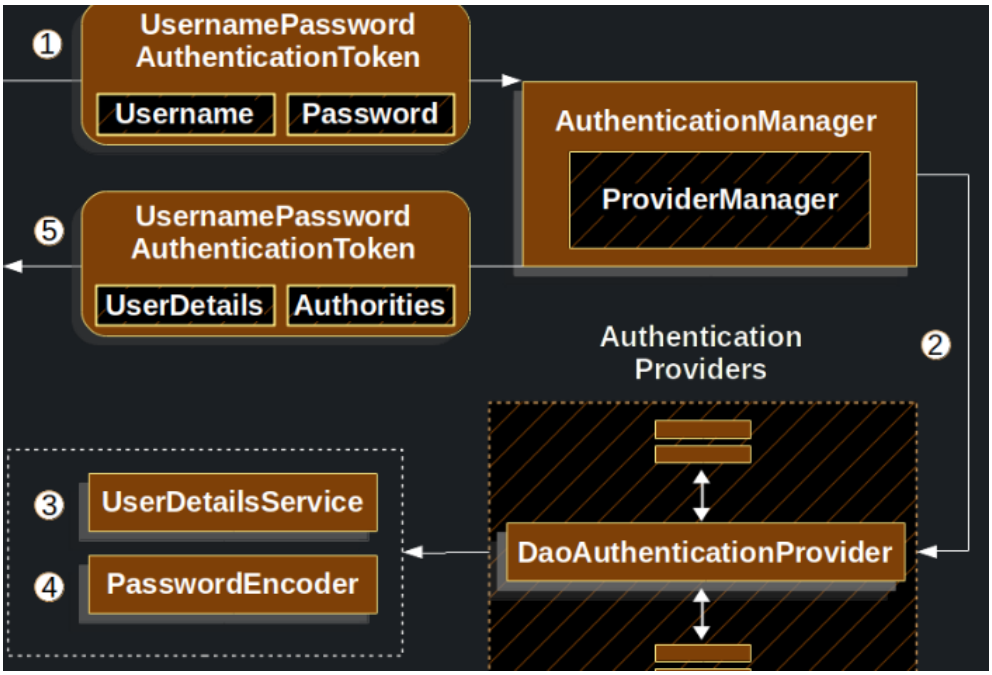


Figure 5. DaoAuthenticationProvider Usage

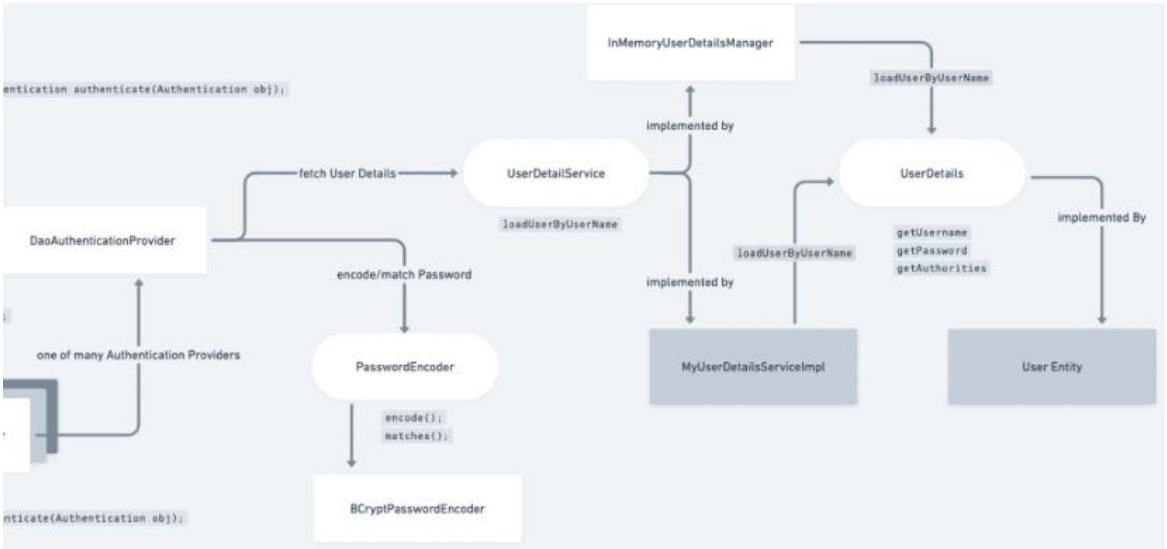


Figure 6.

UserDetailsService

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

```
//pre-defined interface from spring security
public interface UserDetailsService {
   UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

UserDetailsService Implementation

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

```
@Entity
@Data
public class UserInfo{

   @Id
   @GeneratedValue(strategy = GenerationType.UUID)
   private String id;

   @Column(unique = true)
   private String username;

   @Column
   private String password;

   @Column
   private String roles;
}
```

Along with above entity class you will need a repository interface extending the Spring Data JPA repository interface.

```
public interface UserInfoRepository extends JpaRepository<UserInfo,String> {
   Optional<UserInfo> findByUsername(String username);
}
```

So we have to create a model class which implements the UserDetails class which we then return from UserDetailsService class implementation.

```
public class UserDetailModel implements UserDetails {

   private String username;
   private String password;
   private List<GrantedAuthority> authorities;

   public UserDetailModel(UserInfo user){
      this.username = user.getUsername();
      this.password = user.getPassword();
      this.authorities =
Stream.of(user.getRoles().split(",")).map(SimpleGrantedAuthority::new).collect(Collectors.toList());
   }

   @Override
   public Collection<? extends GrantedAuthority> getAuthorities() { return this.authorities; }
   @Override
   public String getPassword() { return this.password; }
   @Override
   public String getUsername() { return this.username; }
```

```
    @Override
    public boolean isAccountNonExpired() { return false; }
    @Override
    public boolean isAccountNonLocked() { return false; }
    @Override
    public boolean isCredentialsNonExpired() { return false; }
    @Override
    public boolean isEnabled() { return false; }

}

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UserInfoRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        Optional<UserInfo> user = userRepository.findByUsername(username);
        return user.map(UserDetailModel::new)
                .orElseThrow(()->new UsernameNotFoundException("Invalid Username"));
    }
}

// users init
repository.save(new UserInfo("user", passwordEncoder.encode("user"), "ROLE_USER"));
repository.save(new UserInfo("admin", passwordEncoder.encode("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 ---

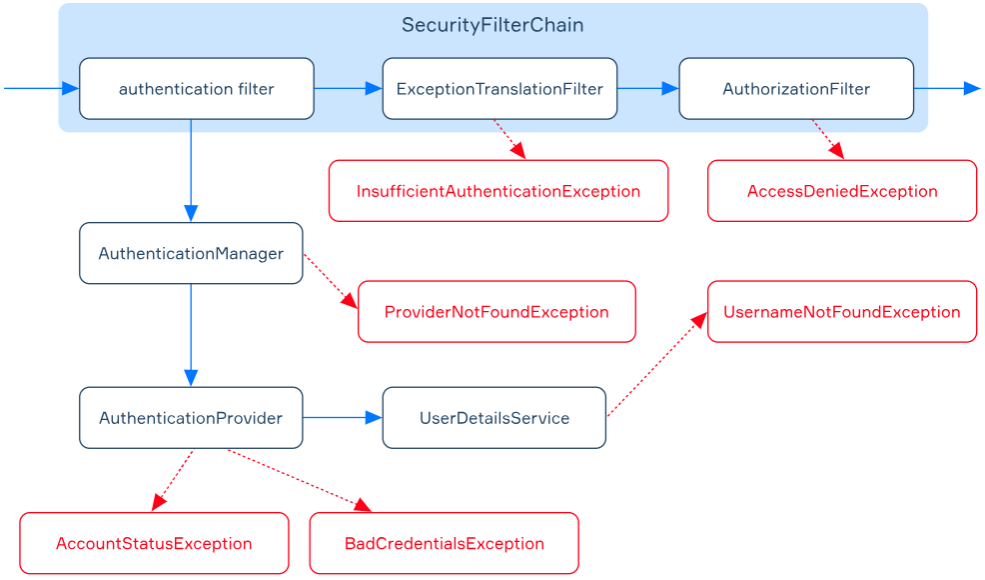
```
@Bean
public UserDetailsService userDetailsService(){
   UserDetails hong   = …;
   UserDetails admin = …;
   return new InMemoryUserDetailsManager(hong, admin);
}
```

PasswordEncoder

The PasswordEncoder (like BCryptPasswordEncoder) contains two methods :
Method encode() is used when a user signs up to create a secure version of their password.
Method matches() is used during login to check if the password provided by the user matches the encoded password stored in the database.

3) Handling Security Exceptions

Spring Security divides exceptions into two categories: authentication-related and authorization-related.
The base exception for authentication-related issues is *AuthenticationException*, an abstract class that extends RuntimeException. There are numerous concrete implementations that extend AuthenticationException.
For authorization-related issues, the base class is *AccessDeniedException*, a concrete class that also extends RuntimeException.
Let's explore some of these exceptions that basic HTTP authentication can throw.



Exception Handling in Spring Filters

Effective Strategy : Manual Exception Handling within the Filter

Directly catch exceptions within the doFilter method of your filter.
Log the exception details for debugging purposes.
Set appropriate HTTP status codes on the response object (e.g., 400 Bad Request, 500 Internal Server Error).
Optionally, write a custom error response body in the desired format (JSON, XML, etc.).

Code Example:

```
@Component
public class CustomFilter extends OncePerRequestFilter {

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                  FilterChain filterChain) throws ServletException, IOException {
    try {
      // Filter logic...
      filterChain.doFilter(request, response);
    } catch (Exception e) {
      logger.error("Error during filtering:", e);
      response.setStatus(HttpServletResponse.SC_BAD_REQUEST); // Set appropriate status code
      response.getWriter().write("An error occurred while processing your request.");
    }
  }
}
```

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.
The pseudocode for ExceptionTranslationFilter looks something like this:

```
try {
  filterChain.doFilter(request, response);
} catch (AccessDeniedException | AuthenticationException ex) {
  if (!authenticated || ex instanceof AuthenticationException) {
    startAuthentication();
  } else {
    accessDenied();
  }
}
```

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
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
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 as following

```
@Configuration
public class WebSecurityConfig {
  @Autowired
  @Qualifier("customAuthenticationEntryPoint")
  private AuthenticationEntryPoint authEntryPoint;

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

  @Bean
  SecurityFilterChain apiFilterChain(HttpSecurity http) throws Exception {
    http
    // in basic authentication ----
    .httpBasic(basic -> basic.authenticationEntryPoint(authEntryPoint))
    .exceptionHandling(customizer -> customizer.accessDeniedHandler(accessDeniedHandler))
    ;
     // or for custom filter e.g. jwtAuthenticationFilter ----
    .addFilterAfter(jwtAuthenticationFilter, ExceptionTranslationFilter.class)

    .exceptionHandling(exception -> exception
      .authenticationEntryPoint(authEntryPoint)
      .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*.

The adjustment at filter

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

  private final HandlerExceptionResolver handlerExceptionResolver;

  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                  FilterChain filterChain) throws ServletException, IOException {
    try {
        ...
        filterChain.doFilter(request, response);
    } catch (Exception exception) {
        handlerExceptionResolver.resolveException(request, response, null, exception);
    }
```

Or a new custom implementation of AccessDeniedHandler:

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

  @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 {
  …
  @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;
}
```

4) Implement JWT authentication in a Spring Boot 3 application

AuthenticationProvider bean ---

```
// src\main\java\com\hong\authapi\configs\ApplicationConfiguration.java
@Configuration
public class ApplicationConfiguration {
   ...

   @Bean
   AuthenticationProvider authenticationProvider() {
      DaoAuthenticationProvider authProvider = new DaoAuthenticationProvider();

      authProvider.setUserDetailsService(userDetailsService());
      authProvider.setPasswordEncoder(passwordEncoder());
      return authProvider;
   }
```

JWT Filter / Authen process ---

```
// src\main\java\com\hong\authapi\configs\JwtAuthenticationFilter.java
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

   final String userEmail = jwtService.extractUsername(jwt);  // jwt == String token
   Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

   if (userEmail != null && authentication == null) {  //  (3)
      UserDetails userDetails = this.userDetailsService.loadUserByUsername(userEmail);

      if (jwtService.isTokenValid(jwt, userDetails)) {  // (4)
         // (5)
         usernamePasswordAuthenticationToken =
         new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
         SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
```

Step.3
Once the username is extracted, we verify if a valid Authentication object i.e. if a logged-in user is available using SecurityContextHolder.getContext().getAuthentication(). If not (authentication == null), we use the Spring Security UserDetailsService to load the UserDetails object. For this example we have created AuthUserDetailsService class which returns us the UserDetails object.

Step.4
Next, the JwtFilter calls the jwtHelper.validateToken() to validate the extracted username and makes sure the jwt token has not expired.

Step.5
Once the token is validated, we create an instance of the Authentication object.
Here, the object UsernamePasswordAuthenticationToken object is created (which is an implementation of the Authentication interface) and set it to SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken). This indicates that the user is now authenticated.

Step.6
Finally, we call filterChain.doFilter(request, response) so that the next filter gets called in the FilterChain.

Parsing JWT Token ---

```
// src\main\java\com\hong\authapi\services\JwtService.java
import io.jsonwebtoken.Jwts;

@Service
public class JwtService {

  private Claims extractAllClaims(String token) {
    return Jwts.parserBuilder().setSigningKey(getSignInKey()).build().parseClaimsJws(token)
.getBody();
  }

  public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
    final Claims claims = extractAllClaims(token);
    return claimsResolver.apply(claims);
  }

  private Date extractExpiration(String token) {
    return extractClaim(token, Claims::getExpiration);
  }

  public String extractUsername(String token) {
    return extractClaim(token, Claims::getSubject);
  }

  private boolean isTokenExpired(String token) {  // token expired ?
    return extractExpiration(token).before(new Date());
  }

  public boolean isTokenValid(String token, UserDetails userDetails) {  // token valid ?
    final String username = extractUsername(token);
    return (username.equals(userDetails.getUsername())) && !isTokenExpired(token);
  }
```

Creating JWT Token (Login) ---

```
// src\main\java\com\hong\authapi\controllers\AuthenticationController.java
@RequestMapping("/auth")
@RestController
public class AuthenticationController {
  …
  @PostMapping("/login")
  public ResponseEntity<LoginResponse> authenticate(@RequestBody LoginUserDto loginUserDto) {
```

```
        User authenticatedUser = authenticationService.authenticate(loginUserDto);
        String jwtToken = jwtService.generateToken(authenticatedUser);
        LoginResponse loginResponse =
        new LoginResponse().setToken(jwtToken).setExpiresIn(jwtService.getExpirationTime());
        return ResponseEntity.ok(loginResponse);
    }
```

SecurityConfiguration ---

```
    // src\main\java\com\hong\authapi\configs\SecurityConfiguration.java
    @Configuration
    @EnableWebSecurity
    public class SecurityConfiguration {

      @Bean
      public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authenticationProvider(authenticationProvider)
            .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)
```

Secure Exception Handling ---

```
    // src\main\java\com\hong\authapi\configs\JwtAuthenticationFilter.java
    @Component
    public class JwtAuthenticationFilter extends OncePerRequestFilter {
      @Override
      protected void doFilterInternal(
        try {  // Throw exceptions
            ...
            if (userEmail != null && authentication == null) {
                …
                if (jwtService.isTokenValid(jwt, userDetails)) {
                  …                                    }
            filterChain.doFilter(request, response);

        } catch (Exception exception) { // pass to ExceptionResolver
            handlerExceptionResolver.resolveException(request, response, null, exception);
        }

    // Handling
    // src\main\java\com\hong\authapi\exceptions\GlobalExceptionHandler.java
    @RestControllerAdvice
    public class GlobalExceptionHandler {

      @ExceptionHandler(Exception.class)
      public ProblemDetail handleSecurityException(Exception exception) {
```

5) 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();
   }
```

LINKS ---

https://docs.spring.io/spring-security/reference/servlet/index.html
https://www.codingshuttle.com/spring-boot-hand-book/introduction-to-spring-boot/
https://www.codingshuttle.com/spring-boot-hand-book/internal-working-of-spring-security-advance/
https://wankhedeshubham.medium.com/spring-boot-security-with-userdetailsservice-and-custom-authentication-provider-3df3a188993f

https://szarpcode.com/spring-security-fundamentals/
https://medium.com/@minadev/authentication-and-authorization-with-spring-security-bf22e985f2cb
https://medium.com/@diluckshan/mastering-exception-handling-in-spring-boot-filters-70be7bb940c7

https://reflectoring.io/spring-security/
https://reflectoring.io/spring-security-jwt/

https://medium.com/@dilankacm/spring-security-architecture-explained-with-jwt-authentication-example-spring-boot-5cc583a9aeac
https://medium.com/@dev.muhammet.ozen/understanding-spring-security-09b25c73581d
https://medium.com/@shobhittulshain/spring-security-deep-dive-into-its-internal-components-and-flow-of-control-130938be9419
https://medium.com/@tericcabrel/implement-jwt-authentication-in-a-spring-boot-3-application-5839e4fd8fac