Securing a restful API with Spring Security

written by Hong Le Nguyen, last update: 06.2025

The related Code on Github :
https://github.com/hong1234/Spring6Boot3-JWT-Authen-APIh

1) Security Filter Chain

Spring Security Integration into Spring MVC

Terminology

Filter	*all* filters implement the jakarta.servlet.Filter interface
Servlet filters	filters managed by Servlet Container
FilterChain	sequence of Servlet filters
Security filters	filters managed by Spring Container
SecurityFilterChain	a sequence of some Security filters

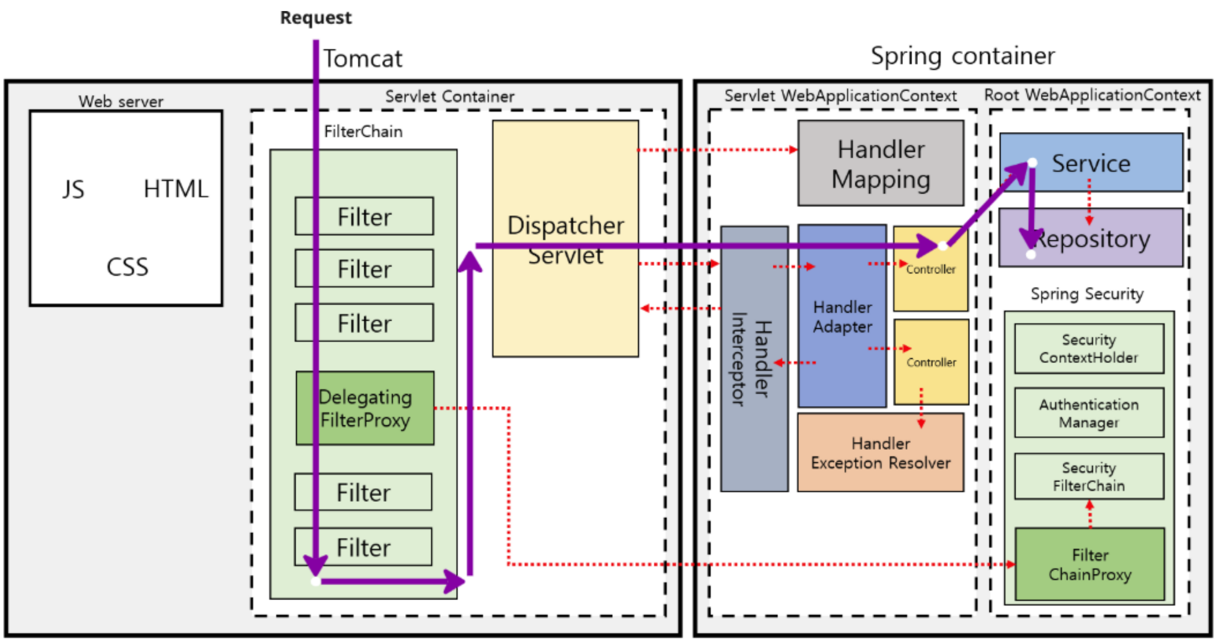


Figure 1a. Injecting Security filters into (Servlet-) FilterChain

Spring Security comes into action by adding Security filters (SecurityFilterChain) in front of the DispatcherServlet. Incoming request will visit these filters one by one before it hit DispatcherServlet and then controllers. This way, I can check Authentication, Authorization states and common attacks. Figure 1a.

DelegatingFilterProxy --

Since the Security filters are not registered via Servlet Container standards (e.g in web.xml file), the Servlet Container is not aware of them. But Security filters should be injected into the FilterChain of Servlet Container.

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 1a.

FilterChainProxy --

The filter bean provided by Spring Security is the entry point to the Security filters. It manages the registered SecurityFilterChains and determines which SecurityFilterChain a given request goes through based on the match between the request-path and the URL-pattern configured in SecurityFilterChains.

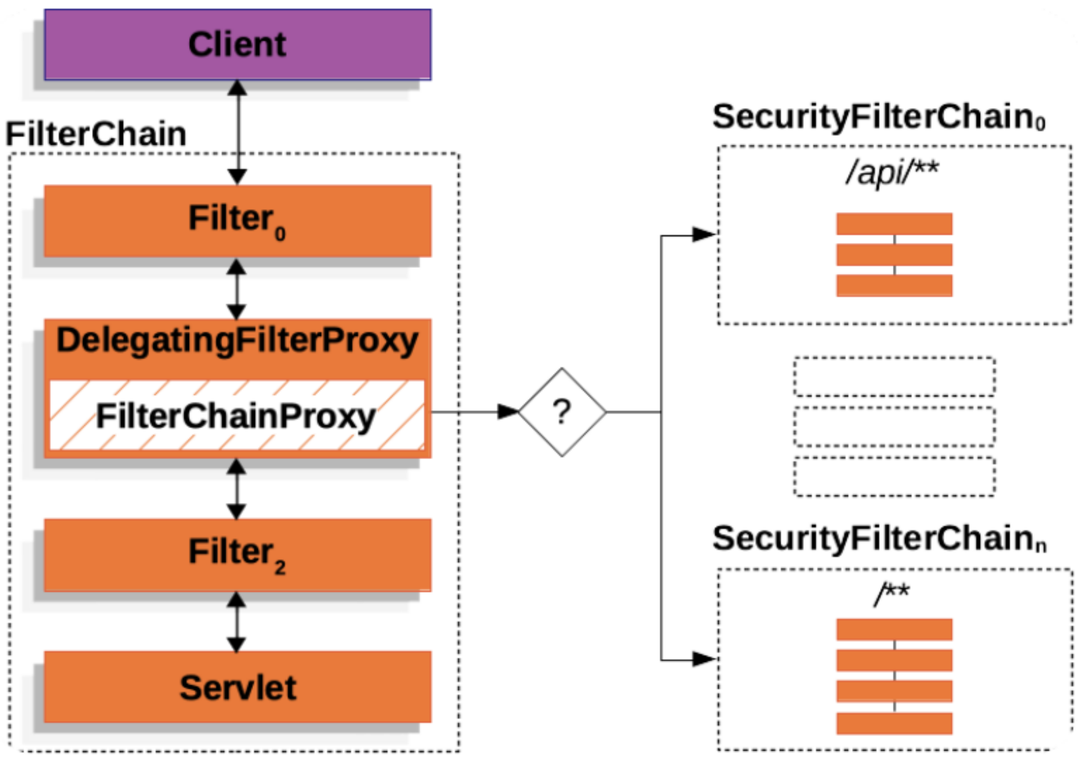


Figure 1b. Delegating the request to the corresponding SecurityFilterChain

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 available whenever we use the @EnableWebSecurity annotation in the security config

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

Security Filter Chain

The filter chain represents a collection of filters *with a defined order* in which they act. The filters form a chain of responsibilities. They work together to perform various security-related tasks.

A filter receives a request, executes its logic, and eventually delegates the request to the next filter in the chain (figure 1c.).

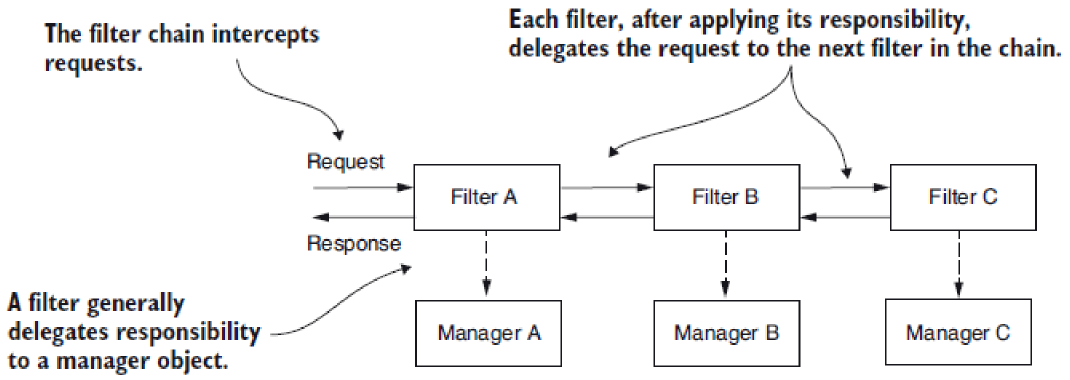

The filter chain intercepts requests.
Each filter, after applying its responsibility, delegates the request to the next filter in the chain.
Request
Response
Filter A
Filter B
Filter C
Manager A
Manager B
Manager C
A filter generally delegates responsibility to a manager object.

Figure 1c.

In practice, applications come with various requirements where default configurations no longer work. You should adjust the filter chain. Spring Security provides filter implementations that you can enable or disable ; but you can also define custom filters.

Note that there can be 1-n SecurityFilterChains in the application.

The Security Default configuration / defaultSecurityFilterChain

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

```
// org.springframework.boot.autoconfigure.security.servlet.SpringBootWebSecurityConfiguration
        class SpringBootWebSecurityConfiguration {
          ...
          @Bean
          SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception {
            http.authorizeHttpRequests((requests) -> requests
                .anyRequest().authenticated()
            );
            http.formLogin(withDefaults());
            http.httpBasic(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. By default, Spring Security expects the default username „user“. Each time you run the application, it generates a new password and prints this password in the console.

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 order so that authentication occurs before authorization*.

UsernamePasswordAuthenticationFilter --
// org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter

is a default filter designed for the default form login.

This filter will extract username and password from a POST request body and send them to *ProviderManager* (Authentication Manager). This manager then send them to *DaoAuthenticationProvider* (default Authentication Provider). And this provider will go to *InMemoryUserDetailsManager* (default UserDetailsService), and check if user exists with given credentials.

BasicAuthenticationFilter --
// org.springframework.security.web.authentication.www.BasicAuthenticationFilter

is responsible for processing any request that has a header "Authorization: Basic Username-Password-Token" with Username-Password-Token - a Base64-encoded *username:password*.

For example, to authenticate user "admin" with password "admin", the following header would be presented "Authorization: Basic YWRtaW46YWRtaW4=". We use cURL to call the endpoint

```
curl -H "Authorization: Basic YWRtaW46YWRtaW4=" localhost:8080/hello
```

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.

SecurityFilterChain-related Configuration

To configure security at the web request level, we'll need to declare a SecurityFilterChain bean.

```
@Configuration
public class SecuConfig {
  …
  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
    // here call methods of the HttpSecurity object to configure
    …
    return http.build();
  }
```

The filterChain() method accepts an HttpSecurity object, which acts as a builder that can be used to configure how security is handled at the web level. Once security configuration is set up via the HttpSecurity object, a call to build() will create a SecurityFilterChain.

Adding a custom filter in Security Filter Chain

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

```
import jakarta.servlet.Filter;
public class CustomFilter implements Filter {
  @Override
  public void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
                                                        throws ServletException, IOException { … }
```

To make sure that our Security 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(HttpServletRequest request, HttpServletResponse response,
                                          FilterChain filterChain) throws ServletException, IOException {
    // filter logic
    ...
    // call the next filter
    filterChain.doFilter(request, response);
  }
```

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.

```
@Bean
public SecurityFilterChain filterChain(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)
```

Multiple Filter Chains

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/**")
  …
  .httpBasic(withDefaults());
  return http.build();
}

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

2) Authentication

Authentication is the process of verifying who user is.

In a Spring Boot application, this involves checking user credentials (username, password) or token(*) 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.

(*) In token-based authentication (for example using JSON Web Token), the server doesn't need to store the token. Instead, it accepts the token and *verifies its validity* before granting access to the user.

Authentication Architecture in Spring Security

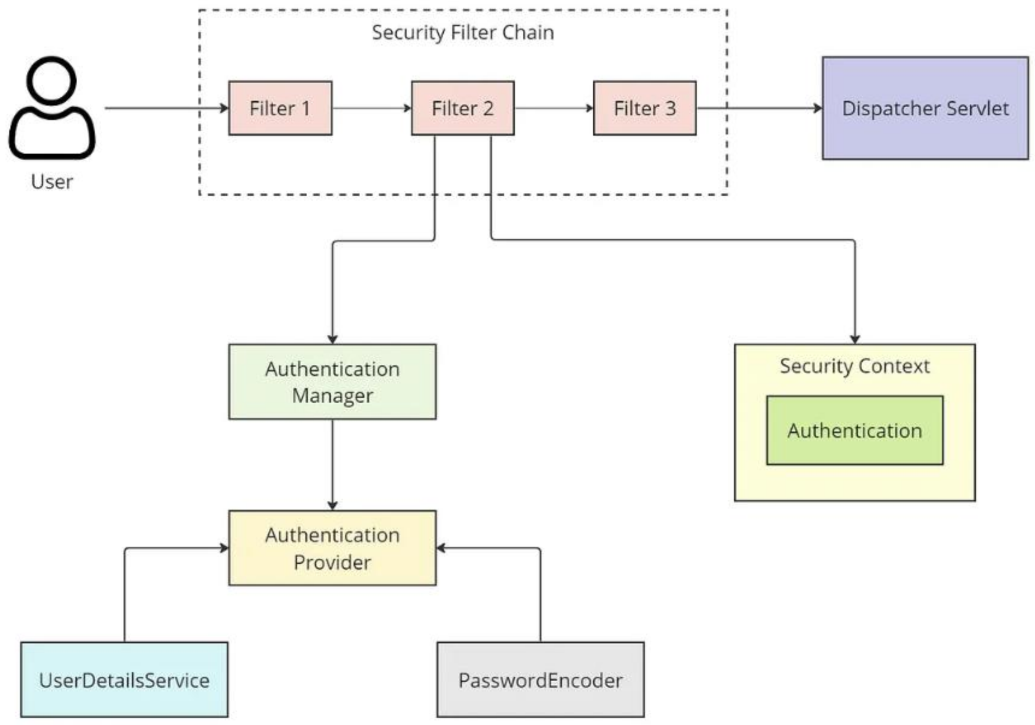


Figure 2a. Authentication Architecture in Spring Security

SecurityContext

The Spring Security framework stores logged-in users in the system. The SecurityContext is responsible for storing and retrieving the Authentication object which includes details of the currently authenticated user.

An Authentication filter sets the Authentication object in the SecurityContext

```
Authentication  populatedAuthentication = … ;
SecurityContext context = SecurityContextHolder.getContext();
context.setAuthentication(populatedAuthentication);
```

The Authorization filter then uses this object to determine access to a specific resource.

```
Authentication authentication = context.getAuthentication();
```

Authentication interface

```
public interface Authentication extends Principal, Serializable {
   Collection<? extends GrantedAuthority> getAuthorities();
   Object getCredentials();
   Object getPrincipal();
   Object getDetails();
   boolean isAuthenticated();
   void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException;
```

UsernamePasswordAuthenticationToken
is a subclass of AbstractAuthenticationToken that implements Authentication, contains information of user.

```
public class UsernamePasswordAuthenticationToken extends AbstractAuthenticationToken {
   private final Object principal;  // the user's ID
   private Object credentials;       // the user's PW

   // Create object before authentication is completed
   public UsernamePasswordAuthenticationToken(Object principal, Object credentials) { … }

   // Create object after authentication is completed
   public UsernamePasswordAuthenticationToken(Object principal, Object credentials,
                                              Collection<? extends GrantedAuthority> authorities) { … }
}
```

An object created by the first constructor is used as input for authentication.

```
Authentication authen = new UsernamePasswordAuthenticationToken(username, password);
//  authen.isAuthenticated() returns false.
```

The result of the authentication is an object created by the second constructor. The security-relevant user data and authorities are stored in the object.

```
Authentication populatedAuthen = authenticationManager.authenticate(authen);
// populatedAuthen.isAuthenticated() returns true.
```

Authentication Manager

The type AuthenticationManager responsible for authenticating a user.

```
public interface AuthenticationManager {
   Authentication authenticate(Authentication authentication) throws AuthenticationException;
}
```

The authenticate(…) method is passed an Authentication object with user credentials (username, password), an Authentication object is returned—at least in the best case.

The *ProviderManager* is the default implementation. It manages multiple AuthenticationProviders to handle different authentication mechanisms.

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

```
public class ProviderManager implements AuthenticationManager {
   …
   public List<AuthenticationProvider> getProviders() { return providers; }

   public Authentication authenticate(Authentication authentication) {
      Authentication result = null;

      for (AuthenticationProvider provider : getProviders()) {
         result = provider.authenticate(authentication);
         if (result != null) { …; break; }
      …
      return result;
   }
```

If any of the providers successfully authenticates the request, the manager returns the Authentication object to the caller (filter). If none of the providers can authenticate the request, the authentication manager will throw an *AuthenticationException*.

Authentication Provider

The authentication provider is the component where *the logic for authentication* resides.

```
public interface AuthenticationProvider {
   Authentication authenticate(Authentication authentication) throws AuthenticationException;
   boolean supports(Class<?> authentication);
}
```

The method authenticate() accepts an Authentication object containing user credentials and validates it based on the defined logic.

If the user is successfully authenticated, an another Authentication object with information and authorities of the user is returned. If authentication fails, an *AuthenticationException* is thrown.

The supports method of the authentication provider checks whether a particular type of Authentication object can be validated by the authentication provider. This is useful in applications with multiple authentication mechanisms, such as user credentials, access tokens, or OTPs. For each authentication method, individual authentication providers can be defined, ensuring that only requests of the corresponding type are validated by the appropriate provider.

DaoAuthenticationProvider

The AuthenticationProvider interface has around 20 different implementations, each with its own specific functionality. One such implementation is the DaoAuthenticationProvider.

DaoAuthenticationProvider checks whether the provided username exists via *UserDetailsService* and retrieves the user's details, such as the encoded password and any associated roles or permissions. It then uses *PasswordEncoder* to verify that the provided password matches the stored password. If authentication is successful, it returns a fully populated Authentication object with the user's details and granted authorities.

UserDetailsService

This is responsible for retrieving user information from the application's database or other (usually) persistent storage.

```
public interface UserDetailsService {
   UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

UserDetailsService Implementation
Solution #1 : Custom Implementation of method : UserDetails loadUserByUsername(String username)
Solution #2 : Using (build-in) Classes JdbcUserDetailsManager, InMemoryUserDetailsManager

PasswordEncoder

This helps to match the provided username and password with the stored ones.
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.

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 {
```

The way to register our own CustomAuthenticationProvider in the ProviderManager described above can be done in SecurityConfig.

```
@Configuration
public class AppConfiguration {
  …

 @Bean
  AuthenticationProvider authenticationProvider() {
    CustomAuthenProvider authProvider = new CustomAuthenProvider();
    return authProvider;
  }

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

Authentication Filter

Authentication filter is the place where the *authentication process* should be implemented.

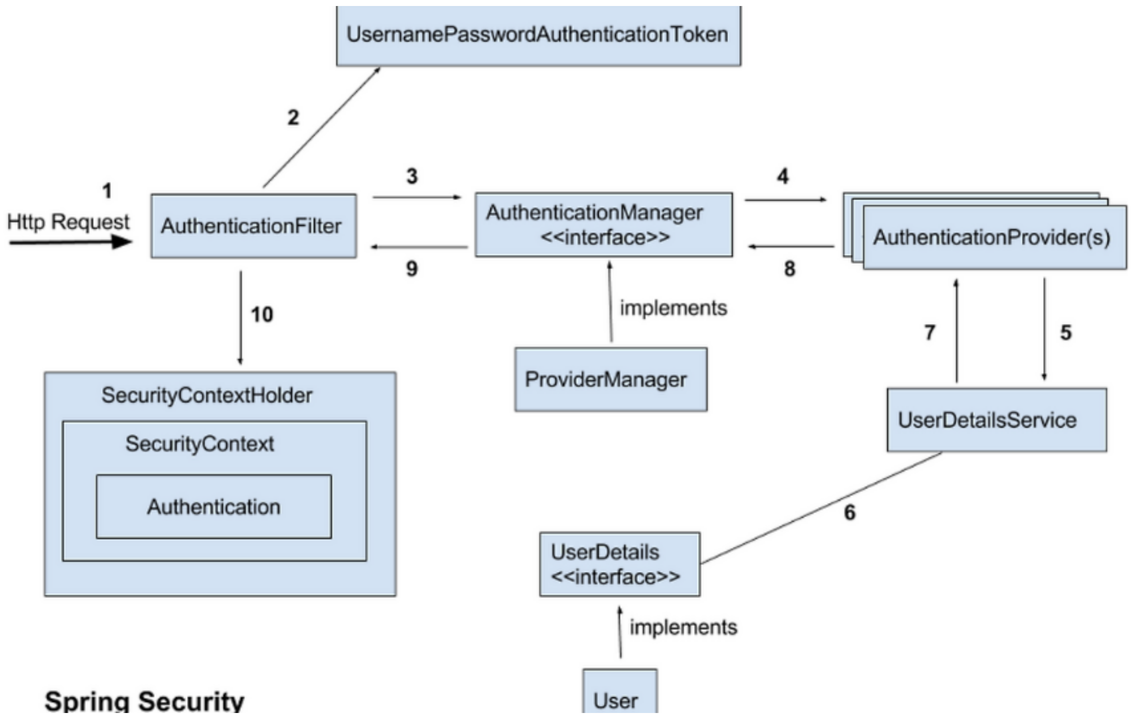


Figure 2b. Authentication process

The filter intercepts the request and *checks whether it is meant to process that particular type of request*.

It then creates an Authentication object with the username and password extracted from the request and passes object to the authentication manager.

The authentication manager iterates over the list of authentication providers.

Each authentication provider uses the UserDetailsService to retrieve user information and the password Encoder to validate the password; authenticating the request based on the provided logic.

If the Authentication Manager throws an AuthenticationException, the request is filtered out.

On successful authentication, a fully authenticated Authentication object is returned.

The filter then makes the Authentication object available throughout the lifecycle of the request by feeding it into the SecurityContext.

At any point during the processing of the request, we can retrieve the currently logged-in user's details from this Authentication object by using: SecurityContextHolder.getContext().getAuthentication();

```
public class MyAuthenticationFilter extends OncePerRequestFilter {
  …
  @Override
  protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                   FilterChain chain) throws IOException, ServletException {
    …
    Authentication authRequest = this.authenticationConverter.convert(request);

    if (authRequest == null) { chain.doFilter(request, response); return; }

    Authentication existingAuthen = SecurityContextHolder.getContext().getAuthentication();
    // Do not attempt to authenticate if client is already authenticated
    if (existingAuthen != null && existingAuthen.isAuthenticated()) {
        chain.doFilter(request, response); return; }

    try {

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

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

       // Pass the control to the next filter
      chain.doFilter(request, response);

    }  catch (Exception e) {
       handlerExceptionResolver.resolveException(request, response, null, exception);
    }
  }
```

3) Authorization

Authorization: The process of determining whether an authenticated user can access a requested resource.

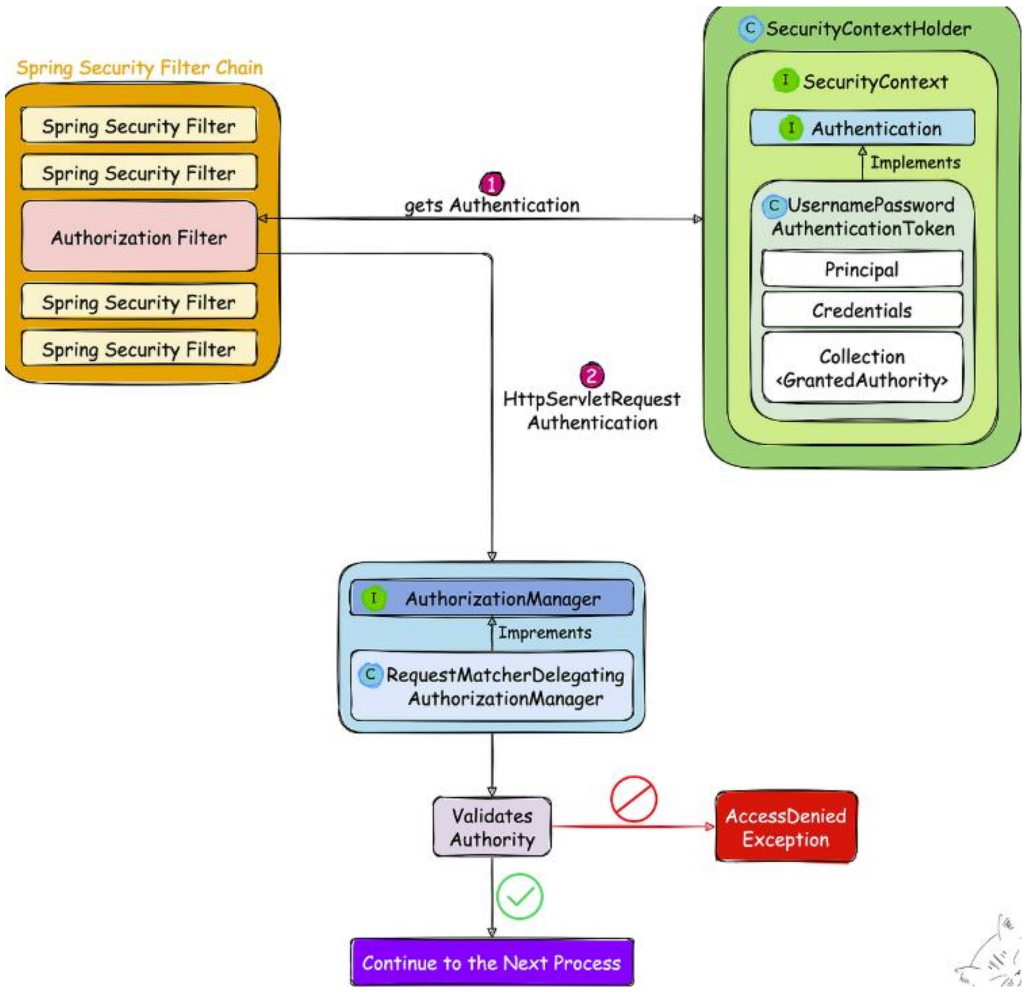


Figure 3a. The Authorization Architecture

AuthorizationFilter

The last filter in the Security filter chain has the function to restrict user access through URL. The decision to allow or deny a request is based on user information (an authentication object stored in the SecurityContext) and endpoint authorization rules (declared in the configuration).

```
@Bean
SecurityFilterChain web(HttpSecurity http) throws Exception {
   http.authorizeHttpRequests(authorize -> authorize
         .requestMatchers("/login").permitAll()
         .requestMatchers("/admin/**").hasRole("ADMIN")
         .anyRequest().authenticated()
   );
   return http.build();
}
```

If authorization is denied, an AuthorizationDeniedEvent is published, and an *AccessDeniedException* is thrown. In this case the ExceptionTranslationFilter handles the AccessDeniedException.

If access is granted, an AuthorizationGrantedEvent is published and AuthorizationFilter continues with the FilterChain which allows the application to process normally.

The AuthorizationFilter doesn't contain much authorization-related logic itself. It merely delegates the authorization request to an AuthorizationManager.

```
public class AuthorizationFilter extends OncePerRequestFilter {
  private final AuthorizationManager<HttpServletRequest> authorizationManager;
  …
  protected void doFilterInternal(HttpServletRequest request, ...) {
    AuthorizationDecision decision =
          this.authorizationManager.check(this::getAuthentication, request);
  }
```

AuthorizationManager

In the context of Spring Web MVC, this is per default a *RequestMatcherDelegatingAuthorizationManager*. It receives a Supplier<Authentication> and the HttpServletRequest that needs to be authorized. It also contains a List<RequestMatcherEntry>, which is used to determine if it is responsible for the authorization of the given request. If one of the RequestMatcherEntrys matches, its corresponding AuthorizationManager is used to do the actual authorization of the request.

The authorization flow

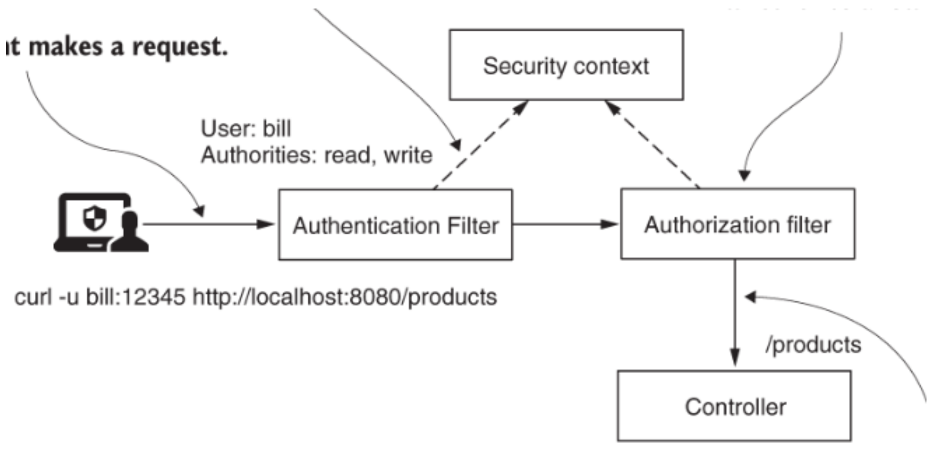
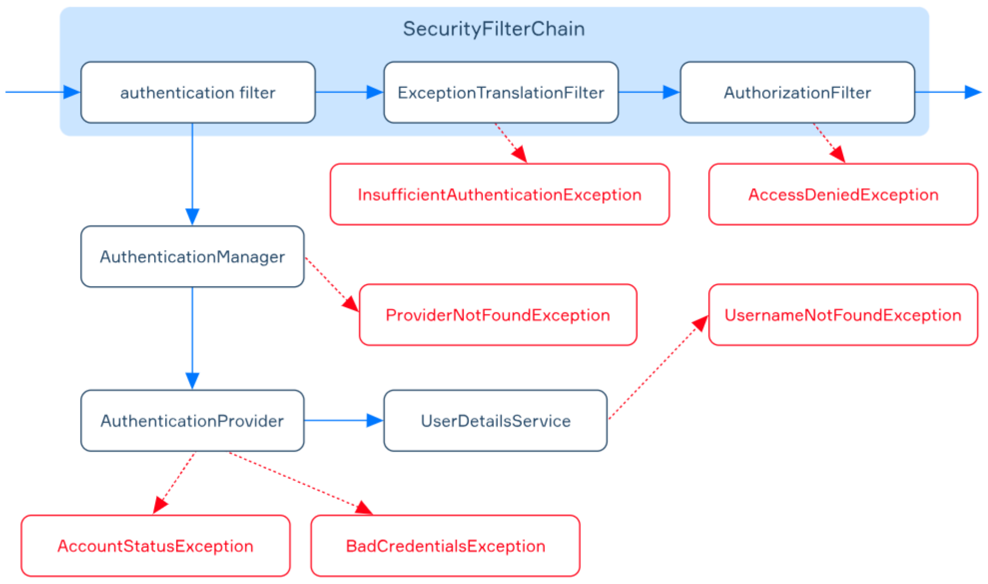


Figure 3b. The Authorization flow

After successful authentication, the authentication filter stores the userdetails in the Security Context and forwards the request to the authorization filter. The *Authorization Filter* gets the userdetails from the Security Context and decides whether the call is permitted.

4) 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;
  }
```

5) Implement JWT-token-based 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) {
```

LINKS ----

// good architect //
https://velog.io/@gwon477/Spring-Security-JWT

// good architect //
https://jschmitz.dev/posts/exploring_spring_securitys_authentication_mechanism_in_web_applications/
https://jschmitz.dev/posts/exploring_spring_securitys_authorization_mechanisms/

// good Authentication //
https://medium.com/@sallu-salman/understanding-the-key-components-in-spring-security-authentication-245297a40b93
https://devsyw.tistory.com/11
// deeper //
https://backendstory.com/spring-security-authentication-architecture-explained-in-depth/

// good Authorizing //
https://gnidinger.tistory.com/entry/SpringSpring-Security-%EA%B6%8C%ED%95%9C-%EB%B6%80%EC%97%ACAuthorization-%EA%B5%AC%EC%A1%B0
https://szarpcode.com/spring-security-fundamentals/

// good book Spring Secu in Action //
https://fizalihsan.github.io/technology/spring-security.html

https://www.codejava.net/frameworks/spring/handle-exceptions-in-a-filter
https://medium.com/@tharinduanupa05/custom-exception-handling-for-spring-security-related-exceptions-dcc93fcb187a

// good JWT Authentication Implement //
https://medium.com/@tericcabrel/implement-jwt-authentication-in-a-spring-boot-3-application-5839e4fd8fac

// Implement //
https://www.codingshuttle.com/blogs/spring-security-is-really-not-that-hard-internal-working-of-spring-security/

https://scrutinybykhimaanshu.blogspot.com/2020/02/spring-security-part-3-spring-security.html#google_vignette