1. Introduction to Spring Security

1. What is Spring Security?

Spring Security is a comprehensive framework for providing authentication, authorization, and protection against attacks in Java-based applications. It is a crucial component for securing web applications and APIs in enterprise environments. Spring Security is highly configurable and extensible, allowing it to be tailored to the specific security needs of each application.

2. Key Features:

- Authentication and Authorization: Spring Security handles identity verification (authentication) and access control (authorization) to different parts of an application.
- Protection Against Attacks: It offers mechanisms to protect against common threats such as Cross-Site Request Forgery (CSRF), Cross-Site Scripting (XSS), and Session Fixation.
- Integration with Identity Providers: Supports authentication through multiple mechanisms like databases, LDAP, OAuth2, and OpenID Connect.
- Configuration and Extensibility: Provides flexible configuration via configuration files or code, and can be extended with custom components.

3. Spring Security Architecture:

- Security Filter Chain: The backbone of security in Spring is the security filter, which intercepts all requests and applies the configured security before they reach the application controllers.
- Authentication Manager: Handles user authentication. The AuthenticationManager verifies user credentials and provides an `Authentication` object if they are valid.
- UserDetailsService: Loads user details, such as username and credentials, from a data source to be used in the authentication process.
- SecurityContext: Maintains the security information for the current session, such as the `Authentication` object, which contains the details of the authenticated user.

4. Basic Configuration:

In Spring Security 6, configuration is done through the `SecurityConfiguration` class and functional configuration methods. Configuration can be set up using the `SecurityConfiguration` class that employs a functional programming approach to define security rules.

Example Configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSe
curity;
import
org.springframework.security.config.annotation.web.builders.WebSec
urity;
import
org.springframework.security.config.annotation.web.configuration.E
nableWebSecurity;
import
org.springframework.security.config.annotation.web.configuration.W
ebSecurityConfigurerAdapter;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAu
thenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .requestMatchers("/public/**").permitAll()
                    .anyRequest().authenticated()
            )
            .formLogin(withDefaults());
        return http.build();
    }
}
```

5. Benefits of Using Spring Security:

- Comprehensive Security: Provides a complete and proven security solution that addresses various security needs.
- Easy Integration: Seamlessly integrates with other parts of the Spring ecosystem, such as Spring Boot and Spring Data JPA.
- Flexible Configuration: Allows detailed and customizable configuration to meet the specific requirements of each application.

- Active Community: Spring Security is maintained by an active community and is regularly updated to address new threats and vulnerabilities.

6. Common Use Cases:

- Web Application Protection: Implement user authentication and access control to different sections of a web application.
- API Security: Secure RESTful service endpoints and protect sensitive data.
- Third-Party Authentication: Integrate with external authentication systems like LDAP or OAuth2 providers.

2. Configuring Authentication and Authorization

1. Authentication vs. Authorization:

- Authentication is the process of verifying the identity of a user. It ensures that users are who they claim to be.
- Authorization determines what an authenticated user is allowed to do. It manages access control to resources based on user roles or permissions.

2. Authentication Configuration:

In Spring Security, authentication can be configured to support various mechanisms, including:

- Form-Based Authentication: Uses a web form for users to enter their credentials. Spring Security handles the login process and user authentication.
- Basic Authentication: Transmits credentials (username and password) with each request, usually in HTTP headers.
- JWT (JSON Web Tokens): Uses tokens to authenticate users in stateless applications, where tokens are passed with each request.

Example of Form-Based Authentication Configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSe
curity;
import
org.springframework.security.config.annotation.web.configuration.E
nableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

```
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .requestMatchers("/login").permitAll() //
Allow access to login page
                    .anyRequest().authenticated() // Require
authentication for all other requests
            )
            .formLogin(formLogin ->
                formLogin
                    .loginPage("/login") // Specify custom login
page URL
                    .permitAll() // Allow all users to access the
login page
            );

        return http.build();
    }
}
```

3. Authorization Configuration:

Authorization in Spring Security can be configured using role-based or permission-based approaches:

- Role-Based Authorization: Grants access based on user roles (e.g., USER, ADMIN).
- Permission-Based Authorization: Grants access based on specific permissions or actions.

Example of Role-Based Authorization:

```
http.authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .requestMatchers("/admin/**").hasRole("ADMIN")
// Admin access only
                    .requestMatchers("/user/**").hasRole("USER")
// User access only
                    .anyRequest().authenticated() // Require
authentication for all other requests
            )
```

4. Custom Authentication Providers:

You can create custom authentication providers if the built-in providers do not meet your needs. This is done by implementing the `AuthenticationProvider` interface and registering it in the security configuration.

Example:

```
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.AuthenticationProvider;
import
org.springframework.security.core.userdetails.UsernameNotFoundExce
ption;

public class CustomAuthenticationProvider implements
AuthenticationProvider {

    private final UserDetailsService userDetailsService;

    public CustomAuthenticationProvider(UserDetailsService
userDetailsService) {
        this.userDetailsService = userDetailsService;
    }

    @Override
    public Authentication authenticate(Authentication
authentication) throws AuthenticationException {
        // Custom authentication logic
        String username = authentication.getName();
        UserDetails userDetails =
userDetailsService.loadUserByUsername(username);
        if (userDetails != null) {
            // Perform additional authentication checks if needed
            return new
UsernamePasswordAuthenticationToken(userDetails,
authentication.getCredentials(), userDetails.getAuthorities());
        }
        throw new UsernameNotFoundException("User not found");
    }

    @Override
    public boolean supports(Class<?> authentication) {
```

```
        return
UsernamePasswordAuthenticationToken.class.isAssignableFrom(authent
ication);
    }
}
```

5. Best Practices:

- Least Privilege Principle: Grant users only the permissions they need to perform their job functions.
- Secure Password Storage: Use hashing algorithms (e.g., bcrypt) to store passwords securely.
- HTTPS: Use HTTPS to secure communications between clients and servers.
- Regular Updates: Keep Spring Security and other dependencies up-to-date to mitigate vulnerabilities.

6. Troubleshooting and Common Issues:

- Access Denied Errors: Check if the user has the correct roles or permissions and if the URL patterns are correctly configured.
- Login Issues: Ensure that the login form and credentials are correctly configured and that users are being authenticated correctly.

3. JWT en Spring Security

What is JWT?

JSON Web Tokens (JWT) is an open standard (RFC 7519) that defines a compact and self-contained method for securely transmitting information between parties as a JSON object. JWTs are commonly used in authentication and authorization systems to maintain session information and enable secure access to resources. They are ideal for handling token-based authentication and provide an efficient way to validate user identity and authorize access to resources.

Structure of a JWT

A JWT is composed of three parts encoded in Base64:

Header:

- The header typically consists of the token type, which is JWT, and the signing algorithm used, such as HMAC SHA256 or RSA.

Example of a header in JSON format:

```
{
  "alg": "HS256",
  "typ": "JWT"
```

```
}
```

-

Payload:

- The payload contains the claims. These are statements about an entity (typically the user) and additional data.
- Claims are divided into three types:
  - Registered claims: A set of predefined claims in the JWT standard, such as `sub` (subject), `exp` (expiration), `iss` (issuer), and others.
  - Public claims: Claims defined by users that should be registered in the IANA JSON Web Token Registry or be private use.
  - Private claims: Custom claims used to share information between parties who have agreed to use these claims.

Example of a payload in JSON format:

```
{
  "sub": "user123",
  "name": "John Doe",
  "iat": 1516239022
}
```

-

Signature:

  - To create the signature, the header and payload are encoded in Base64Url, and then signed using the specified algorithm and a secret key.
  - The signature ensures that the token has not been altered after being issued.

Example of how to create the signature:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

-

Dependencies for Working with JWT in Spring

To work with JWT in a Spring Boot project, you need to add the following dependencies to your `pom.xml` file if you are using Maven:

1. JJWT API:

Provides the core API for creating and handling JWTs.

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
```

```
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
```

- 2. JJWT Implementation:

Concrete implementation of the JJWT API.

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
</dependency>
```

- 3. JJWT Extensions - Jackson:

Extensions for handling JWT serialization and deserialization with Jackson.

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
</dependency>
```

These dependencies will allow you to generate and validate JWTs in your Spring Boot application. Ensure to include them in your `pom.xml` file so Maven can download and add them to your project's classpath.

Using JWT with Spring Security

Using JWT with Spring Security enables implementing a token-based authentication system where the JWT acts as an access token that the server can validate to authenticate requests. This eliminates the need to manage sessions on the server since the JWT is self-contained and contains all the information necessary for authentication and authorization.

3. JWT en Spring Security

3.1. Utility for JWT Handling

The `JwtUtil` class is responsible for creating and validating JWTs. It provides methods for generating tokens, extracting information from them, and validating their authenticity.

Example Implementation of JwtUtil:

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
```

```java
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import javax.crypto.SecretKey;
import java.util.Date;

@Component
public class JwtUtils {

    private final SecretKey secretKey;

    @Value("${jwt.expiration}")
    private long expiration;

    public JwtUtils(@Value("${jwt.secret}") String secret) {
        this.secretKey =
Keys.hmacShaKeyFor(Decoders.BASE64.decode(secret));
    }

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis()
+ expiration))
                .signWith(secretKey, SignatureAlgorithm.HS256)
                .compact();
    }

    public String extractUsername(String token) {
        return parseClaims(token).getSubject();
    }

    public boolean validateToken(String token, String username) {
        final String tokenUsername = extractUsername(token);
        return (tokenUsername.equals(username) &&
!isTokenExpired(token));
    }

    private Claims parseClaims(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(secretKey)
                .build()
                .parseClaimsJws(token)
                .getBody();
    }
```

```
    private boolean isTokenExpired(String token) {
        return parseClaims(token).getExpiration().before(new
Date());
    }
}
```

Explanation of Methods:

1. `generateToken(String username)`:
    - Purpose: Generates a new JWT based on the provided username.
    - How It Works: Creates a token with the username as the subject, sets the issuance date, defines an expiration date, and signs the token with the HMAC SHA-256 algorithm using a secret key.
2. `extractUsername(String token)`:
    - Purpose: Extracts the username from the provided JWT.
    - How It Works: Decodes the JWT and retrieves the subject of the token, which is the username.
3. `validateToken(String token, String username)`:
    - Purpose: Validates the JWT to ensure it is valid and not expired.
    - How It Works: Extracts the username from the token and compares it with the provided username. It also checks if the token has expired using the `isTokenExpired` method.
4. `parseClaims(String token)`:
    - Purpose: Parses the JWT to obtain the claims contained within it.
    - How It Works: Uses the secret key to verify and decode the JWT, then extracts the claims from the token.
5. `isTokenExpired(String token)`:
    - Purpose: Checks if the JWT has expired.
    - How It Works: Compares the expiration date of the JWT with the current date. If the token has expired, it returns `true`; otherwise, it returns `false`.

Configuration in `application.properties`

To configure the secret key and the expiration time for the JWT, you can use the following properties in your `application.properties` file:

```
jwt.secret=${SECRET_KEY}
jwt.expiration=${EXPIRATION_TIME}
```

Ensure that the environment variables `SECRET_KEY` and `EXPIRATION_TIME` are defined in your system.

Warning: The `jwt.secret` environment variable must meet the following requirements:

1. Base64 Format: The value must be encoded in Base64.
2. Minimum Length: It must have a minimum length of 32 bytes once decoded.

Ensure that the provided value meets these requirements to ensure proper functioning and security of the application.

3. JWT en Spring Security

3.2. JWT Authentication Filter

The `JwtAuthenticationFilter` class is a filter that intercepts HTTP requests to extract and validate the JWT token. If the token is valid, it authenticates the user within the Spring security context. This filter ensures that requests to protected endpoints are processed only if the user is properly authenticated.

Example Implementation of JwtAuthenticationFilter:

```
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthen
ticationToken;
import
org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.web.authentication.WebAuthenticationD
etailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

import java.io.IOException;

@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter
{

    @Autowired
    private JwtUtils jwtUtils;

    @Autowired
    private UserDetailsService userDetailsService;
```

```
    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain chain)
            throws ServletException, IOException {
        String requestURI = request.getRequestURI();

        // Allow access to public endpoints without authentication
        if (requestURI.startsWith("/public/")) {
            chain.doFilter(request, response);
            return;
        }

        String header = request.getHeader("Authorization");
        String token = null;
        String username = null;

        if (header != null && header.startsWith("Bearer ")) {
            token = header.substring(7);
            try {
                username = jwtUtils.extractUsername(token);
            } catch (Exception e) {
                logger.error("Error extracting username from
token", e);
            }
        }

        if (username != null &&
SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails =
userDetailsService.loadUserByUsername(username);
            if (jwtUtils.validateToken(token,
userDetails.getUsername())) {
                UsernamePasswordAuthenticationToken authentication
= new UsernamePasswordAuthenticationToken(
                        userDetails, null,
userDetails.getAuthorities());
                authentication.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authenticatio
n);
            }
        }
        chain.doFilter(request, response);
    }
```

```
}
```

Class Description:

1. Purpose: The `JwtAuthenticationFilter` is used to intercept and process HTTP requests. Its primary function is to check for the presence and validity of the JWT token in incoming requests and, if valid, set the user authentication in the Spring security context.
2. Process:
    - Public Endpoint Access: The filter first allows access to public endpoints without requiring authentication. This is done by checking if the request URL starts with "/public/".
    - Token Extraction and Validation: If the endpoint is not public, the filter looks for the `Authorization` header. If the header is present and starts with "Bearer ", the JWT token is extracted.
    - User Authentication: The filter uses `JwtUtils` to extract the username from the token and validate the token. If the token is valid and the user is not yet authenticated, user information is loaded using `UserDetailsService`, and the authentication is set in the Spring security context.
    - Filter Continuation: Finally, the filter allows the request to continue through the filter chain, processing the request as it normally would.

3. JWT en Spring Security

3.3. CustomUserDetailsService

The `CustomUserDetailsService` class implements `UserDetailsService` and is used to load user details from a data source (such as a database) based on the username. It is essential for Spring Security to retrieve user information and handle authentication.

Example Implementation of CustomUserDetailsService:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import
org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundExce
ption;
import org.springframework.stereotype.Service;

@Service
public class CustomUserDetailsService implements
UserDetailsService {
```

```
    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        UserEntity userEntity =
userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("User
not found with username: " + username));
        return new User(userEntity.getEmail(),
userEntity.getPassword(), new ArrayList<>());
    }
}
```

Class Description:

1. Purpose: The `CustomUserDetailsService` class is responsible for loading user-specific data. It retrieves user details from the database based on the username provided.
2. Process:
    - User Retrieval: The `loadUserByUsername` method queries the `UserRepository` to find a user with the specified username. If no user is found, it throws a `UsernameNotFoundException`.
    - UserDetails Creation: If the user is found, the method returns a `UserDetails` object from Spring Security, which includes the username, password, and authorities (in this case, an empty list of authorities). This `UserDetails` object is used by Spring Security to perform authentication and authorization.
3. Dependency: The class relies on `UserRepository` to access user data from the database, and it uses Spring Security's `User` class for representing user information.

3. JWT en Spring Security

3.4. SecurityConfig

The `SecurityConfig` class configures Spring Security, defines URL access rules, and applies the JWT filter for authentication. It ensures that all incoming requests are properly authenticated and authorized based on the defined security rules.

Example Implementation of SecurityConfig:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import
org.springframework.security.config.annotation.web.builders.HttpSe
curity;
import
org.springframework.security.config.annotation.web.builders.WebSec
urity;
import
org.springframework.security.config.annotation.web.configuration.E
nableWebSecurity;
import
org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import
org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAu
thenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(); // Define the password
encoder bean
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        http
            .csrf(csrf -> csrf.disable()) // Disable CSRF for REST
APIs
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .requestMatchers("/public/**").permitAll() //
Allow public access to specific endpoints
                    .anyRequest().authenticated() // All other
requests must be authenticated
            )
            .addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class); // Apply JWT filter
```

```
        return http.build();
    }
}
```

Explanation:

1. CSRF Configuration:
    - `csrf -> csrf.disable()`: Disables CSRF protection as it is generally not needed for stateless REST APIs where tokens are used for authentication.
2. Authorization Requests:
    - `authorizeRequests(authorizeRequests -> authorizeRequests.requestMatchers("/public/**").permitAll().anyRequest().authenticated())`: Allows public access to routes that match `/public/**` and requires authentication for all other requests.
3. Custom JWT Filter:
    - `addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class)`: Adds the custom JWT filter to the filter chain before the default `UsernamePasswordAuthenticationFilter` to handle JWT authentication.

3. JWT en Spring Security

3.5. User Authentication: Login and Registration

When building a secure application, user authentication is a critical aspect. It involves validating user credentials to grant access to the application. In a typical setup, there are two main functionalities: user registration and user login.

User Registration

User registration is the process by which a new user creates an account in the application. The user provides necessary information such as username, password, and other optional details. This information is then stored securely in the database.

Steps for User Registration:

1. User Input Validation: Ensure the user-provided data is valid (e.g., proper email format, password strength).
2. Password Encoding: Encode the password before storing it to ensure security.
3. Storing User Details: Save the user details in the database.
4. Error Handling: Handle any errors such as duplicate usernames or database issues.

Example Code for User Registration:

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import
org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private PasswordEncoder passwordEncoder;

    public void registerUser(UserRegistrationDto registrationDto)
{
        if
(userRepository.existsByUsername(registrationDto.getUsername())) {
            throw new RuntimeException("Username already exists");
        }

        User user = new User();
        user.setUsername(registrationDto.getUsername());

user.setPassword(passwordEncoder.encode(registrationDto.getPasswor
d()));
        userRepository.save(user);
    }
}

@RestController
@RequestMapping("/api/auth")
public class AuthController {

    @Autowired
    private UserService userService;

    @PostMapping("/register")
    public ResponseEntity<String> registerUser(@RequestBody
UserRegistrationDto registrationDto) {
        userService.registerUser(registrationDto);
        return ResponseEntity.ok("User registered successfully");
    }
}

public class UserRegistrationDto {
    private String username;
```

```
    private String password;

    // getters and setters
}
```

Explanation:

- UserService: This service handles the registration logic. It validates the user data, encodes the password, and saves the user.
- AuthController: This controller provides an endpoint for user registration.
- UserRegistrationDto: A DTO (Data Transfer Object) for capturing registration data.

User Login

User login is the process where an existing user provides their credentials to access the application. Upon successful authentication, the server generates a token (e.g., JWT) and returns it to the client for subsequent requests.

Steps for User Login:

1. User Authentication: Validate the username and password.
2. Token Generation: Generate a JWT token upon successful authentication.
3. Response: Return the token to the client.

Example Code for User Login:

```
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthen
ticationToken;
import org.springframework.security.core.Authentication;
import
org.springframework.security.core.context.SecurityContextHolder;
import
org.springframework.security.web.authentication.WebAuthenticationD
etailsSource;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/auth")
public class AuthController {
```

```
    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtUtil jwtUtil;

    @PostMapping("/login")
    public ResponseEntity<String> authenticateUser(@RequestBody
LoginRequest loginRequest) {
        Authentication authentication =
authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                loginRequest.getUsername(),
                loginRequest.getPassword()
            )
        );


SecurityContextHolder.getContext().setAuthentication(authenticatio
n);
        String jwt = jwtUtil.generateToken(authentication);
        return ResponseEntity.ok(jwt);
    }
}

public class LoginRequest {
    private String username;
    private String password;

    // getters and setters
}
```

Explanation:

- AuthController: This controller provides an endpoint for user login. It authenticates the user and generates a JWT token.
- LoginRequest: A DTO for capturing login data.
- AuthenticationManager: Used to authenticate the user credentials.
- JwtUtil: A utility class for generating JWT tokens.

Security Considerations

When implementing login and registration, consider the following security best practices:

1. Password Encoding: Always encode passwords before storing them in the database. Use a strong encoder like `BCryptPasswordEncoder`.
2. Input Validation: Validate all user inputs to prevent SQL injection and other attacks.
3. Secure Communication: Ensure all communication is done over HTTPS to protect sensitive data.
4. Token Expiry: Implement token expiry and refresh mechanisms to enhance security.
5. Error Handling: Provide appropriate error messages without exposing sensitive information.

Note: To enable authentication, you need to create a Bean for `AuthenticationManager` in your `SecurityConfig` class. The Bean should be configured as follows:

```
@Bean
public AuthenticationManager
authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
    return authenticationConfiguration.getAuthenticationManager();
}
```

By following these guidelines, you can create a secure authentication system that protects user data and enhances the overall security of your application.

4. SecurityFilterChain

`SecurityFilterChain` is the interface used in Spring Security 5 and later versions to define security configurations based on filters. Starting with Spring Security 6, it is recommended to use `SecurityFilterChain` instead of the older `WebSecurityConfigurerAdapter`.

Basic Configuration with `SecurityFilterChain`: To define security configuration using `SecurityFilterChain`, create a `@Bean` that returns an instance of `SecurityFilterChain`. In this bean, you can configure application security using an `HttpSecurity` object.

Example Configuration with `SecurityFilterChain`:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSe
curity;
import org.springframework.security.web.SecurityFilterChain;
```

```
import
org.springframework.security.web.authentication.UsernamePasswordAu
thenticationFilter;

@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        http
            .csrf(csrf -> csrf.disable()) // Disable CSRF for REST
APIs
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .requestMatchers("/public/**").permitAll() //
Allow access without authentication to public routes
                    .anyRequest().authenticated() // Require
authentication for all other requests
            )
            .addFilterBefore(new JwtAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class) // Add JWT filter
before standard authentication filter
            .cors(cors -> cors.disable()); // Disable CORS by
default (can be customized)

        return http.build();
    }
}
```

HttpSecurity

`HttpSecurity` provides a fluent API for configuring web security in Spring applications. It allows configuration of aspects such as request authorization, authentication, CSRF handling, security headers, and more.

Common Configurations of `HttpSecurity`:

1. Disabling CSRF:
    - Description: CSRF (Cross-Site Request Forgery) is a protection mechanism against attacks where an authenticated user performs unwanted actions on a web application. In REST APIs, where authentication is done via tokens and not through web forms, it is common to disable CSRF.

Configuration:

```
http.csrf(csrf -> csrf.disable());
```

2. Header Policies:
    - Description: Security headers help protect the application against common attacks such as Clickjacking and XSS.
    - Configuration of `X-Frame-Options`:

Allows configuring whether the application can be embedded in an iframe. In environments using H2 Console (a tool for interacting with H2 databases), this option needs to be allowed.

```
http.headers(headers ->
    headers.frameOptions(frameOptions ->
frameOptions.sameOrigin())
);
```

    - Configuration of `Content Security Policy (CSP)`:

Defines which resources can be loaded by the web application, helping to prevent XSS attacks.

```
http.headers(headers ->
    headers.contentSecurityPolicy("script-src 'self'")
);
```

3. Request Authorization:
    - Description: Configures which routes are available to authenticated and unauthenticated users.

Configuration:

```
http.authorizeRequests(authorizeRequests ->
    authorizeRequests
        .requestMatchers("/public/**").permitAll() // Allow access
without authentication
        .anyRequest().authenticated() // Require authentication
for all other requests
);
```

4. Filter Configuration:
    - Description: Allows adding custom filters in the security filter chain, such as JWT authentication filters.

Configuration:

```
http.addFilterBefore(new JwtAuthenticationFilter(),
UsernamePasswordAuthenticationFilter.class);
```

5. Session Management:

- Description: Configures session handling to control session behavior in the application.

Configuration:

```
http.sessionManagement(sessionManagement ->

sessionManagement.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
);
```

6. Authentication and Authorization:
   - Description: Configures authentication and authorization details, such as credentials and role handling.
   - Configuration with `formLogin`:

Note: The `formLogin` configuration is relevant when using form-based authentication (e.g., for HTTP Basic authentication). In an application using JWT-based authentication, controllers handle authentication requests, and `formLogin` is not used. Instead, `formLogin` is disabled to avoid conflicts with JWT-based authentication.

```
http.formLogin(formLogin ->
    formLogin.disable() // Disable form-based login when using JWT authentication
);
```

7. CORS Configuration:
   - Description: CORS (Cross-Origin Resource Sharing) is a mechanism that allows or restricts resources on a web server to be requested from another domain. In a REST API, proper CORS configuration is essential to ensure that the API can be accessed by clients from different origins.
   - Configuration:

Default Disable: By default, CORS is disabled. This setting can be customized based on the application's needs.

```
http.cors(cors -> cors.disable());
```

Custom CORS Configuration: If you need to enable CORS with specific settings, you can define a `CorsConfigurationSource` bean.

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();

configuration.setAllowedOrigins(Arrays.asList("http://example.com"));
    configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE"));
```

```
    configuration.setAllowedHeaders(Arrays.asList("Authorization",
"Content-Type"));
    UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```

■

Summary

- `SecurityFilterChain`: Defines security configuration based on filters for the application.
- `HttpSecurity`: Provides a fluent API for configuring detailed security aspects in web applications.
- Common configurations: Include disabling CSRF, header policies, request authorization, custom filter configuration, session management, authentication, and authorization.
- CORS: Essential for allowing or restricting cross-origin requests, with customizable configurations based on application requirements.

This modular and flexible approach to security configuration allows tailoring the application's protection to different scenarios and security needs.

5. Configuring Security in SpringDoc

When integrating SpringDoc with Spring Security, you need to ensure that the API documentation (Swagger UI) is accessible while protecting your API endpoints using JWT authentication. Below are the key steps and configurations required:

1. Allowing Access to Documentation Endpoints

Update the `SecurityConfig` to allow access to SpringDoc API documentation endpoints such as `/v3/api-docs` and `/swagger-ui/**`. This configuration permits unauthenticated access to these endpoints while requiring authentication for other requests.

Example Configuration:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builders.HttpSe
curity;
import
org.springframework.security.config.annotation.web.configuration.E
nableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;
```

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity
http) throws Exception {
        http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests
                    .requestMatchers("/v3/api-docs/**",
"/swagger-ui/**", "/swagger-ui.html").permitAll() // Allow public
access to API documentation
                    .anyRequest().authenticated() // Require
authentication for all other requests
            )
            .csrf(csrf -> csrf.disable()) // Disable CSRF for API
endpoints
            .formLogin(withDefaults()) // Enable form login
            .httpBasic(withDefaults()); // Enable basic
authentication

        return http.build();
    }
}
```

2. Configuring OpenAPI for JWT Authentication

To integrate JWT authentication with SpringDoc, configure security schemes in the OpenAPI definition and apply them to specific endpoints.

Example Configuration:

```
import io.swagger.v3.oas.annotations.OpenAPIDefinition;
import io.swagger.v3.oas.annotations.info.Info;
import io.swagger.v3.oas.annotations.security.SecurityRequirement;
import io.swagger.v3.oas.annotations.security.SecurityScheme;
import io.swagger.v3.oas.annotations.enums.SecuritySchemeType;
import org.springframework.context.annotation.Configuration;

@Configuration
@OpenAPIDefinition(
    info = @Info(title = "My API", version = "v1"),
    security = @SecurityRequirement(name = "bearerAuth")
)
@SecurityScheme(
```

```
    name = "bearerAuth",
    type = SecuritySchemeType.HTTP,
    scheme = "bearer",
    bearerFormat = "JWT"
)
public class OpenApiConfig {
}
```

3. Applying Security to Specific Endpoints

Specify security requirements for individual endpoints using the `@SecurityRequirement` annotation at the method level.

Example Controller with Endpoint Security:

```
import io.swagger.v3.oas.annotations.security.SecurityRequirement;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class ApiController {

    @GetMapping("/protected")
    @SecurityRequirement(name = "bearerAuth") // Apply JWT
authentication to this endpoint
    public String protectedEndpoint() {
        return "This is a protected endpoint";
    }

    @GetMapping("/public")
    public String publicEndpoint() {
        return "This is a public endpoint";
    }
}
```

Summary

To configure security for SpringDoc API Starter:

1. Allow public access to the SpringDoc-related documentation endpoints by updating the security configuration.
2. Configure OpenAPI to use JWT authentication with the `@SecurityScheme` and `@OpenAPIDefinition` annotations.

3. Apply security requirements to specific endpoints using the `@SecurityRequirement` annotation.

This approach ensures that the API documentation is accessible while protecting sensitive API endpoints with JWT authentication.