SpringSecurityJwtApplication

This code represents a Spring Boot application with Spring Security and JWT (JSON Web Token) authentication.

1. Application Class (SpringSecurityJwtApplication):

- The class is annotated with @SpringBootApplication, indicating that it is the main entry point for the Spring Boot application.
- It implements the CommandLineRunner interface, which means that the run method will be executed when the application starts. In this method, a sample user with the username "mno" and a hashed password is created and saved to the database.

2. Security Configuration (SecurityFilterChain Bean):

- The @Bean method filterChain configures the security settings using HttpSecurity.
- It disables CSRF protection, configures authorization rules, and sets the session creation policy to be stateless (JWT is used for authentication, and there is no need for sessions).
- The .antMatchers("/home").hasRole("ADMIN") configuration specifies that the "/home" endpoint requires the "ADMIN" role.
- The "/authenticate" endpoint is configured to be accessible without authentication (permitAll()).
- The JwtRequestFilter is added to the security filter chain before the UsernamePasswordAuthenticationFilter. This filter is responsible for validating JWTs and setting up Spring Security authentication based on the token.

3. Authentication Configuration (AuthenticationManager Bean):

- The @Bean method authenticationManager configures the AuthenticationManager bean. It uses AuthenticationConfiguration to obtain the authentication manager.

4. Password Encoder Bean (getEncoder):

- The @Bean method getEncoder creates and returns a BCryptPasswordEncoder. This encoder is used to hash passwords.

5. Run Method (run):

- The run method is part of the CommandLineRunner interface and is executed on application startup.
- It creates a sample user with the username "mno," sets up its password, and assigns it the role "ROLE_ADMIN." Both the user and the authority are then saved to their respective repositories (userRepo and authRepo).

In summary, this Spring Boot application demonstrates a basic setup for JWT authentication with Spring Security. It includes configurations for authorization, a JWT filter, and a sample user creation during application startup. The use of JWT allows stateless authentication, making it suitable for RESTful APIs. The password hashing is handled using BCryptPasswordEncoder, and the application uses rolebased authorization.

`AppController`

AuthenticationManager Injection:

The AuthenticationManager is injected using @Autowired. This is a component responsible for authenticating users based on the provided credentials.

@GetMapping("home") Home Endpoint:

Defines a simple endpoint at "/home" accessible via HTTP GET requests.

Returns the string "home" as the response.

@PostMapping("authenticate") Authentication Endpoint:

Handles HTTP POST requests to the "/authenticate" endpoint.

Accepts an AuthenticationRequest object in the request body, which typically contains a username and password.

Attempts to authenticate the user using the injected AuthenticationManager and the provided credentials.

If authentication is successful:

Retrieves user details from the authenticated Authentication object.

Generates a JWT (JSON Web Token) containing the username, issued and expiration dates, and user roles.

Adds the JWT to the response headers under the "Authorization" field.

Returns a ResponseEntity<Boolean> with true indicating successful authentication, along with the response headers and HTTP status OK (200).

If authentication fails:

Catches any exceptions thrown during the authentication process.

Returns a ResponseEntity<Boolean> with false indicating unsuccessful authentication and HTTP status OK (200).

JWT Generation:

Uses the Auth0 JWT library (com.auth0.jwt) to create a JWT.

Sets the subject (username), issued date, expiration date, and user roles in the JWT payload.

Signs the JWT using the HMAC256 algorithm with the secret key "helloworld".

Response Handling:

Creates an HttpHeaders object to include in the response.

Adds the JWT to the "Authorization" header in the form of "Bearer <token>".

Constructs and returns a ResponseEntity<Boolean> with the appropriate headers and status.

Exception Handling:

Catches any exceptions that may occur during the authentication process.

Returns a ResponseEntity<Boolean> with false indicating unsuccessful authentication and HTTP status OK (200).

In summary, this controller class defines endpoints for basic home access and user authentication. The authentication endpoint uses Spring Security's AuthenticationManager to authenticate users, generates a JWT upon successful authentication, and returns the result along with the JWT in the response headers. Exception handling is included to handle authentication failures.

`JwtRequestFilter`

Filter Class (JwtRequestFilter):

This is a component annotated with @Component, making it a Springmanaged bean.

Filtering Logic (doFilterInternal method):

Overrides the doFilterInternal method from the OncePerRequestFilter class, which ensures that the filter is only executed once per request.

Checks if the request path is "/authenticate." If true, the filter chain proceeds without any JWT processing.

JWT Processing:

Extracts the "Authorization" header from the incoming request.

Checks if the header starts with "Bearer " to identify a JWT token.

If a valid JWT token is found:

Extracts the token from the header.

Uses the Auth0 JWT library (com.auth0.jwt) to verify the token's signature and decode its content.

Retrieves the subject (username) from the decoded JWT.

Uses the UserDetailsService to load user details based on the username from the JWT.

Constructs a UsernamePasswordAuthenticationToken using the loaded user details.

Sets the authentication in the SecurityContextHolder.

Allows the filter chain to proceed with the authenticated user.

Exception Handling:

Catches any exceptions that may occur during JWT verification or user details loading.

Prints the stack trace to the console for debugging purposes.

Continues with the filter chain to ensure that the request proceeds even in case of exceptions.

In summary, this JwtRequestFilter is designed to intercept incoming requests, extract and validate JWT tokens from the "Authorization" header, and authenticate users based on the token's content. It integrates with Spring Security's authentication mechanism, setting the authenticated user in the security context. The filter allows requests to the "/authenticate" endpoint to proceed without JWT processing, assuming that endpoint is responsible for initial authentication.