Spring Security

Integrating Spring Security into a Spring Boot project enhances the application's security and ensures the privacy of users' data. This integration allows users to access their specific dashboards only if their provided credentials are valid and, depending on their roles, grants access to their respective areas.

The GitHub URL contains an application that combines React and Spring Boot, featuring authentication and authorization functionalities.

BE - <https://github.com/sethuvekram/Spring-Security---BE.git>

FE- <https://github.com/sethuvekram/Spring-Security---FE.git>

During the registration process, new users can create an account. The details they provide during registration are securely stored in the MySQL Database. Passwords are hashed using the BCrypt method, which encodes them in a way that prevents unauthorized access. This ensures that passwords remain unreadable by external sources. The application's logic assigns the "User" role to newly registered users. Administrators can modify these roles in the MySQL Database, designating them as "Admin," "Buyer," or "Seller."

Upon entering credentials on the login page, a token specific to the user's email is generated. This token is responsible for authentication purposes. With the correct credentials, users can access the various sections of the website tailored to their roles.

In addition to the existing ecommerce application, I've introduced a "config" folder containing files that enhance the project's configuration. These changes aim to further refine and secure the application's functionality.

Top of Form

**Application Config:**

1. **Package and Imports**: The class is in the package **com.App.config**, and it imports several necessary Spring and custom classes.
2. **Annotations and Dependencies**:
   * **@Configuration**: This annotation indicates that this class provides Spring configuration.
   * **@RequiredArgsConstructor**: This Lombok annotation generates a constructor with all the required fields that are marked with the **final** keyword. It's often used for dependency injection.
3. **Constructor Injection**:

This field is injected through the constructor due to the **@RequiredArgsConstructor** annotation. It seems like the **UserRepository** is a Spring Data repository for managing user data.

1. **PasswordEncoder Bean**:

This method defines a Spring bean that returns a **BCryptPasswordEncoder**. This encoder is used to hash and verify passwords securely.

1. **UserDetailsService Bean**:

This method defines a **UserDetailsService** bean. It uses a lambda expression to fetch user details from the repository based on the provided email (username). If the user is not found, it throws a **UsernameNotFoundException**.

1. **AuthenticationProvider Bean**:

This method defines an **AuthenticationProvider** bean. It creates an instance of **DaoAuthenticationProvider**, sets the **UserDetailsService** and **PasswordEncoder**, and returns the provider.

1. **AuthenticationManager Bean**:

This method defines an **AuthenticationManager** bean. It receives an **AuthenticationConfiguration** object as a parameter and retrieves the authentication manager from it. The authentication manager is a core Spring Security component responsible for authentication.

Overall, this configuration class sets up a secure authentication mechanism using Spring Security. It provides beans for password encoding, user details fetching, and authentication management. The authentication provider connects the user details service and the password encoder, while the authentication manager orchestrates the authentication process in the application.

**JWTAuthFilter:**

1. **Package and Imports**: The class is in the package **com.App.config**. It imports various necessary classes from different libraries, including Spring Security and Lombok.
2. **Annotations and Dependencies**:
   * **@Configuration**: This annotation indicates that this class provides Spring configuration.
   * **@RequiredArgsConstructor**: This Lombok annotation generates a constructor with all the required fields that are marked with the **final** keyword. It's often used for dependency injection.
3. **Extending OncePerRequestFilter**:

This class extends to **OncePerRequestFilter**, which is a base class for filters that ensures they are only executed once per request.

1. **Constructor Injection**:

These fields are injected through the constructor due to the **@RequiredArgsConstructor** annotation. It appears that the **UserDetailsService** and **JwtUtils** classes are dependencies required by this filter.

1. **Overriding doFilterInternal Method**:

This method is an overridden implementation of the **doFilterInternal** method from **OncePerRequestFilter**. It's responsible for the actual filtering logic.

1. **JWT Authentication Logic**:

This section performs the JWT authentication logic:

* + It retrieves the **Authorization** header from the HTTP request.
  + If the header is missing or doesn't start with **"Bearer "**, the request is allowed to continue down the filter chain without further processing.
  + If the header is valid and contains a JWT, it extracts the token and the associated username.
  + If the token is valid and the user's authentication context is not already set, it loads user details from the provided **UserDetailsService** based on the username.
  + If the token is valid and the user details are fetched successfully, an **UsernamePasswordAuthenticationToken** is created and populated with user details and authorities.
  + The token details are also set using **WebAuthenticationDetailsSource** to provide information about the authentication request.
  + The created authentication token is then set in the **SecurityContextHolder**.

1. **Continuing the Filter Chain**:

After the JWT authentication logic is performed (or skipped), the filter allows the request to proceed down the filter chain.

In summary, this class represents a filter that intercepts incoming requests, extracts JWTs from the **Authorization** header, validates them, and sets up the user's authentication context using Spring Security's **SecurityContextHolder**. This is a key component in implementing JWT-based authentication in a Spring Boot application.

**JWTUtils:**

1. **Package and Imports**: The class is in the package **com.App.config**. It imports various classes from the **io.jsonwebtoken** package, which is part of the JSON Web Token library.
2. **Service Annotation**:

This annotation indicates that this class is a Spring service bean and can be injected into other components.

1. **Constants**:

This is a static string representing the secret key used for JWT signing and validation. The key is in hexadecimal format.

1. **Token Extraction Methods**:These methods are used to extract information (claims) from a JWT. **extractUsername** extracts the subject (username) from the token, while **extractClaim** is a generic method that can be used to extract any claim by providing a function that resolves the claim.
2. **Token Generation Methods**:

These methods are used to generate JWT tokens. The first method generates a token without any extra claims. The second method allows additional custom claims to be added to the token. The token's subject (username), issued time, and expiration time are set, and the token is signed using the secret key.

1. **Token Validation Methods**:

These methods are used to validate JWT tokens. **isTokenValid** checks if the username in the token matches the provided user details and if the token is not expired. **isTokenExpired** checks if the token's expiration time is before the current date. **extractExpiration** extracts the expiration time claim from the token.

1. **Token Parsing Method**:

This method parses the JWT token to retrieve all its claims. It uses the JWT library's parser and sets the signing key to verify the token's authenticity.

1. **Key Retrieval Method**:

This method is used to convert the secret key from its hexadecimal representation to binary format and then create an HMAC-based signing key using the converted bytes.

Overall, this class provides a comprehensive set of methods for handling JWTs in a Spring Boot application. It allows for token generation, extraction of claims, validation of tokens, and other essential JWT-related operations.

**Security Config:**

1. **Package and Imports**: The class is in the package **com.App.config**. It imports various Spring Security and other classes that are necessary for configuring security.
2. **Annotations and Dependencies**:
   * **@Configuration**: This annotation indicates that this class provides Spring configuration.
   * **@RequiredArgsConstructor**: This Lombok annotation generates a constructor with all the required fields that are marked with the **final** keyword. It's often used for dependency injection.
   * **@EnableWebSecurity**: This annotation enables Spring Security's web security features.
   * **@EnableMethodSecurity**: This annotation enables Spring Security's method-level security features.
3. **Constructor Injection**:These fields are injected through the constructor due to the **@RequiredArgsConstructor** annotation. The **AuthenticationProvider** and **JwtAuthFilter** are dependencies required for the security configuration.
4. **SecurityFilterChain Bean**:

This method defines a **SecurityFilterChain** bean that configures various security-related settings:

* + **cors()** enables Cross-Origin Resource Sharing and is configured to allow cross-origin requests.
  + **csrf()** disables Cross-Site Request Forgery protection.
  + **authorizeHttpRequests()** configures authorization rules for different URL patterns:
    - Requests to **/User/login** and **/User/register** are allowed without authentication (public access).
    - Requests to URLs starting with **/Admin/** require the user to have the "ADMIN" authority.
    - Requests to URLs starting with **/User/** require the user to have the "USER" authority.
    - Any other request requires authentication.
  + **sessionManagement()** configures session management settings:
    - **SessionCreationPolicy.STATELESS** ensures that Spring Security will not create or use sessions for authentication.
  + **authenticationProvider(authenticationProvider)** sets the provided **AuthenticationProvider** for authentication.
  + **addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class)** adds the custom **JwtAuthFilter** before the default **UsernamePasswordAuthenticationFilter**.