Security Document for Java Spring Boot Backend System

1. Summary of Security Configuration

The security configuration for the Java Spring Boot application is primarily managed through the use of Spring Security. The following components have been identified:

- **Security Configuration Classes**:
  - `WebSecurityConfig`
    : Configures HTTP security, including authentication and authorization rules.
  - `AuthenticationManager`
    : Manages user authentication processes.
- **Security Filters**:
  - `JwtAuthenticationFilter`
    : Intercepts requests to validate JWT tokens.
  - `AuthorizationFilter`
    : Checks user roles and permissions for accessing endpoints.
- **Authentication and Authorization**:
  - Authentication is performed using JWT (JSON Web Tokens).
  - Role-based access control (RBAC) is implemented to manage user permissions.

2. Access Matrix

The following table outlines the roles and permissions associated with each HTTP endpoint, indicating the method and corresponding controller:

HTTP Method	Endpoint	Controller	Roles Required
GET	/api/users	UserController	ROLE_USER, ROLE_ADMIN
POST	/api/users	UserController	ROLE_ADMIN
GET	/api/admin	AdminController	ROLE_ADMIN
PUT	/api/users/{id}	UserController	ROLE_USER, ROLE_ADMIN
DELETE	/api/users/{id}	UserController	ROLE_ADMIN

3. Authentication Mechanisms

The application employs JWT for authentication, which involves the following steps:

1. **User Login**: Users provide credentials (username and password) to the authentication endpoint.
2. **Token Generation**: Upon successful authentication, a JWT is generated and returned to the user.
3. **Token Validation**: For subsequent requests, the JWT is included in the Authorization header. The `JwtAuthenticationFilter` validates the token's integrity and authenticity.
4. **Claims Handling**: The JWT contains claims that represent user roles and permissions, which are extracted and used for authorization.

Token Handling

- Tokens are signed using a secure algorithm (e.g., HS256).
- Expiration time is set to enhance security, requiring users to re-authenticate periodically.

4. Security Error Handling

The application implements standardized error handling for security-related issues:

- **HTTP Status Codes**:
  - 401 Unauthorized: Returned when authentication fails.
  - 403 Forbidden: Returned when the user does not have permission to access a resource.
- **Error Messages**:
  - Custom error messages are provided in the response body to inform users of the specific issue encountered.

5. CORS and CSRF Policies

- **CORS Configuration**:
  - Cross-Origin Resource Sharing (CORS) is configured to allow specific origins, methods, and headers to enhance security while enabling cross-origin requests.
- **CSRF Protection**:
  - CSRF (Cross-Site Request Forgery) protection is enabled for stateful sessions, ensuring that requests are validated against CSRF tokens.

6. Session Management

The application utilizes a **stateless** session management approach. This means:

- No session information is stored on the server.
- Each request is independently authenticated using the JWT, which contains all necessary information.

7. Secure Storage of Credentials and Sensitive Data

- **Password Storage**:
  - User passwords are hashed using a strong algorithm (e.g., BCrypt) before being stored in the database.
- **Sensitive Data**:
  - Sensitive information is encrypted at rest and in transit using industry-standard encryption protocols (e.g., TLS).

8. Logging and Auditing Practices

- **Logging**:
  - The application logs security events, including authentication attempts, access denials, and token validation failures.
- **Auditing**:
  - Audit trails are maintained for critical operations, allowing for tracking of user actions and changes to sensitive data.

9. Recommendations for Improvement

- **Enhance Token Security**: Implement refresh tokens to allow users to obtain new access tokens without re-authentication.
- **Rate Limiting**: Introduce rate limiting on sensitive endpoints to mitigate brute-force attacks.
- **Security Headers**: Implement additional security headers (e.g., Content Security Policy, X-Content-Type-Options) to protect against various web vulnerabilities.
- **Regular Security Audits**: Conduct regular security audits and penetration testing to identify and remediate vulnerabilities.

Conclusion

This document outlines the security configuration and practices implemented in the Java Spring Boot backend system. Adhering to the recommendations provided will further enhance the security posture of the application, ensuring robust protection against potential threats.