# Language and Application Overview

* **Language**: Java
* **Framework**: Spring Boot
* **Application Type**: RESTful Web API for user authentication and data retrieval

# Security Vulnerability Analysis

|  |  |
| --- | --- |
| **Vulnerability** | **Description** |
| **SQL Injection** | Injection attacks are possible when dynamic SQL is used with user input. |
| **Plaintext Password Validation** | Raw password comparison is unsafe and increases the possibility of credential theft. |
| **No Authentication Token Mechanism** | JWT and session tokens are not used; this goes against best practices. |
| **No Input Validation** | No format or length checks are performed on user input. |
| **Error Handling** | System internals could be revealed if exception management is neglected. |
| **Insecure Communication** | Secure headers and HTTPS are not enforced. |

# Secure Coding Recommendations

**3.1 Use Parameterized Queries (Prepared Statements)**

**Fix**:

String sql = "SELECT \* FROM users WHERE username = ? AND password = ?";

List<Map<String, Object>> users = jdbcTemplate.queryForList(sql, username, password);

**Recommended**: To further abstract and secure database operations, use Hibernate or Spring Data JPA.

**3.2 Hash and Salt Passwords**

**Fix**:  
Use a password encoder such as BCryptPasswordEncoder.

@Autowired

private PasswordEncoder passwordEncoder;

public boolean verifyPassword(String rawPassword, String storedHash) {

return passwordEncoder.matches(rawPassword, storedHash);

}

* Store only hashed passwords.
* Use bcrypt, scrypt, or argon2.

**3.3 Implement Token-Based Authentication (JWT)**

* After a successful login, issue JWT tokens.
* Check protected endpoints for JWTs.
* Make use of OAuth2-compatible libraries like Spring Security or JJWt.

**3.4 Validate Inputs**

* Add validation annotations:

@Size(min = 3, max = 20)

@NotNull

private String username;

* Or manually sanitize input for APIs.

**3.5 Secure HTTP Headers & HTTPS Enforcement**

* Enforce HTTPS in production (via reverse proxy or Spring Security).
* Use headers like:

http.headers()

.xssProtection().and()

.contentSecurityPolicy("script-src 'self'");

**4.6 Graceful Error Handling**

* Steer clear of SQL problems and stack trace leaks.
* Return generic error messages.

@ExceptionHandler(Exception.class)

public ResponseEntity<String> handleException(Exception e) {

return new ResponseEntity<>("Internal Server Error", HttpStatus.INTERNAL\_SERVER\_ERROR);

}

# Static Code Analysis Tools

|  |  |
| --- | --- |
| **Tool** | **Description** |
| **SpotBugs + FindSecBugs** | looks for common vulnerabilities in Java bytecode. |
| **SonarQube** | finds bugs, security holes, and code smells. connects to CI/CD. |
| **PMD** | Java source code analyzer for unused code, bad practices. |
| **Checkmarx / Fortify** *(Enterprise)* | SAST tools of commercial quality for extensive security audits. |

**Example: Run FindSecBugs**

# Build the project first (e.g., with Maven)

mvn clean compile

# Run SpotBugs with FindSecBugs plugin

spotbugs -pluginList findsecbugs-plugin.jar -textui -effort:max -high .

# Summary

The application is seriously at danger from SQL Injection, password theft, and insecure session management due to the examined code. Developers can drastically lower the attack surface by incorporating safe coding techniques like parameterised queries, robust password hashing, JWT-based authentication, and appropriate error handling. To automate security reviews and preserve code cleanliness, static analysis tools such as FindSecBugs and SonarQube ought to be incorporated into the development lifecycle (CI/CD).