Spring Boot Security and Validation

1. Introduction

Spring Boot Security and Validation are two of the most critical aspects of modern enterprise application development. Validation ensures that incoming data is clean, reliable, and adheres to business rules, while Security ensures that applications are protected from unauthorized access, data breaches, and malicious activities. In this document, we will explore detailed concepts of JSR-380 Bean Validation, Spring Security fundamentals, authentication and authorization mechanisms, role-based access control, and JSON Web Token (JWT) for stateless API security

- **Data Validation** ensures that only correct, meaningful, and expected data enters the system.
- **Application Security** ensures that only authenticated and authorized users can access the resources.

Spring Boot provides built-in support for both through:

1. **JSR-380 Bean Validation API (Hibernate Validator is the reference implementation).**
2. **Spring Security Framework** (Authentication, Authorization, Role-based Access, JWT).

2. Data Validation using JSR-380 (Bean Validation API)

Data validation is the process of ensuring that user inputs meet the expected constraints before processing them. Spring Boot integrates with JSR-380 (Jakarta Bean Validation 2.0) API, which provides a standard set of annotations to validate Java objects. Hibernate Validator is the reference implementation commonly used with Spring Boot.

2.1 What is Bean Validation?

- Bean Validation is a standard defined under **JSR-380** (also called Bean Validation 2.0).
- In Spring Boot, it is supported through the **Hibernate Validator** dependency.
- It works by **annotating fields of Java classes** (mostly DTOs or Entities) with validation rules.

2.2 Common Annotations in Bean Validation

Annotation	**Purpose**
`@NotNull`	Field cannot be null.
`@NotEmpty`	Field cannot be empty (String/Collection).
`@NotBlank`	Field must contain non-whitespace characters.
`@Size(min, max)`	Restricts the length of string/collection.
`@Email`	Must be a valid email format.
`@Min`, `@Max`	Restrict numerical values.
`@Pattern`	Validates string with regex.

2.3 Example: Validating User Registration Data

```
import jakarta.validation.constraints.*;

public class UserDTO {

    @NotNull(message = "Username cannot be null")
    @Size(min = 3, max = 20, message = "Username must be between 3 and 20 characters")
    private String username;

    @NotBlank(message = "Password cannot be empty")
    private String password;

    @Email(message = "Email should be valid")
    private String email;

    @Min(value = 18, message = "Age must be at least 18")
    private int age;

    // getters and setters
}
```

2.4 Controller with Validation

```
import org.springframework.web.bind.annotation.*;
import org.springframework.validation.annotation.Validated;
import org.springframework.http.ResponseEntity;

@RestController
@RequestMapping("/users")
public class UserController {

    @PostMapping("/register")
    public ResponseEntity<String> registerUser(@Validated @RequestBody UserDTO user) {
        return ResponseEntity.ok("User registered successfully: " + user.getUsername());
    }
}
```

- The `@Validated` annotation tells Spring to check the constraints before processing the request.
- If validation fails, Spring automatically returns a **400 Bad Request** with validation error details.

2.5 Real-time Use Case

- Prevent saving a user with invalid email, blank password, or age < 18.
- Saves debugging and ensures **data integrity at entry point**.

3. Spring Boot Security

Spring Security is a comprehensive framework for authentication, authorization, and protection against common attacks. It integrates seamlessly with Spring Boot and provides both declarative and programmatic security features.

Key Features of Spring Security:

- Authentication: Verifying user identity.
- Authorization: Determining user permissions and roles.
- Password Encoding: Secure password storage using BCryptPasswordEncoder.
- Protection against CSRF, session fixation, and clickjacking.

3.1 What is Spring Security?

- A **powerful and highly customizable authentication and access-control framework**.
- Provides:
    - Authentication (Who are you?)
    - Authorization (What are you allowed to do?)
    - Protection against CSRF, session fixation, clickjacking.

3.2 Authentication vs Authorization

Authentication is the process of confirming user identity, typically using username/password or tokens. Authorization decides what resources a user can access based on their roles and authorities

- **Authentication** → Verifying user identity (login using username/password).
- **Authorization** → Granting access based on roles/permissions (Admin vs User).

3.3 Role-Based Access Control (RBAC) Example

RBAC is a method of restricting access based on user roles. In Spring Security, roles are typically stored in the database or in-memory configuration and are used to define what endpoints can be accessed by which users.

Security Configuration

```
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf().disable()
            .authorizeHttpRequests()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
            .anyRequest().authenticated()
            .and()
            .httpBasic(); // Basic Authentication
        return http.build();
    }
}
```

Controller

```
@RestController
public class DemoController {

    @GetMapping("/admin/dashboard")
    public String adminDashboard() {
        return "Welcome Admin!";
    }

    @GetMapping("/user/profile")
    public String userProfile() {
        return "Welcome User!";
    }
}
```

- Only users with role **ADMIN** can access `/admin/dashboard`.
- Both **USER** and **ADMIN** can access `/user/profile`.

4. JWT (JSON Web Token) Authentication in Spring Boot

In modern applications, especially REST APIs, stateless authentication is preferred over traditional session-based authentication. JWT (JSON Web Token) is a compact and secure way to transmit user identity and claims between client and server. JWTs eliminate the need to store session state on the server.

4.1 Why JWT?

- By default, Spring Security uses **session-based authentication**.
- For REST APIs, **stateless authentication** is preferred.
- **JWT** solves this by sending a signed token with each request.

4.2 JWT Workflow

1. User logs in with username and password.
2. Server authenticates user and returns a **JWT token**.
3. For each request, client sends token in the `Authorization: Bearer <token>` header.
4. Server validates token and grants/denies access.

4.3 JWT Implementation in Spring Boot

1. Add Dependency (pom.xml)

```
<dependency>
```

```
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

2. JWT Utility Class

```
import io.jsonwebtoken.*;
import java.util.Date;

public class JwtUtil {
    private String secretKey = "mysecretkey";
    private long expiration = 1000 * 60 * 60; // 1 hour

    public String generateToken(String username) {
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + expiration))
                .signWith(SignatureAlgorithm.HS512, secretKey)
                .compact();
    }

    public String extractUsername(String token) {
        return Jwts.parser().setSigningKey(secretKey)
                .parseClaimsJws(token).getBody().getSubject();
    }

    public boolean validateToken(String token) {
        return !extractUsername(token).isEmpty();
    }
}
```

3. Login Controller

```
@RestController
public class AuthController {
    private JwtUtil jwtUtil = new JwtUtil();

    @PostMapping("/login")
    public String login(@RequestParam String username, @RequestParam String password) {
        // In real projects, validate username/password from DB
        if ("admin".equals(username) && "password".equals(password)) {
            return jwtUtil.generateToken(username);
        }
        return "Invalid credentials";
    }
}
```

5. Real-Time Case Study

Consider a banking application where validation ensures input correctness (e.g., account number format, minimum balance), while Spring Security ensures that only authenticated users with appropriate roles (ADMIN, CUSTOMER) can perform operations like money transfer, account creation, or balance inquiry. JWT tokens ensure scalability and statelessness for mobile and web clients.

An **E-commerce application**:

- Validate product data (`@NotNull`, `@Size`) before saving.
- Protect endpoints like `/admin/addProduct` only for admins.
- Use JWT for securing mobile app API calls.

6. Summary

- **Validation** ensures data integrity at the boundary of the system.
- **Spring Security** enforces strong authentication and authorization.
- **JWT** enables scalable, stateless authentication for REST APIs.
- Together, they form the backbone of **secure enterprise-grade applications** built with Spring Boot.