# Spring Boot Security & Validation

## 1. ✅ Data Validation in Spring Boot

Validation ensures that **input data is correct before processing**. Spring Boot integrates with the **JSR-380 (Bean Validation 2.0)** API, typically using **Hibernate Validator**.

### Common Annotations (from `javax.validation.constraints`)

| Annotation | Usage |
|---|---|
| `@NotNull` | Field must not be null |
| `@NotEmpty` | String/Collection must not be empty |
| `@NotBlank` | String must not be null/empty/whitespace |
| `@Size(min, max)` | Restricts length of a String/Collection |
| `@Min`, `@Max` | Numeric range constraints |
| `@Email` | Must be valid email |
| `@Pattern(regex="…")` | Must match regex |
| `@Past`, `@Future` | Date/time validation |

### Example – DTO Validation

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

public class UserDTO {

    @NotNull(message = "Id cannot be null")
    private Long id;

    @NotBlank(message = "Name is mandatory")
    @Size(min = 3, max = 20, message = "Name must be between 3-20 characters")
    private String name;

    @Email(message = "Invalid email format")
    private String email;

    @Min(value = 18, message = "Age must be at least 18")
```

```
    private int age;

    // getters and setters
}
```

### Controller Example

```
import org.springframework.web.bind.annotation.*;
import org.springframework.validation.annotation.Validated;
import jakarta.validation.Valid;

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

    @PostMapping
    public String createUser(@Valid @RequestBody UserDTO user) {
        return "User created: " + user.getName();
    }
}
```

⚡ Here `@Valid` ensures validation is applied before entering the method. If validation fails, Spring throws `MethodArgumentNotValidException`.

## 2. 🔒 Spring Boot Security Overview

Spring Security is the **de-facto security framework** in the Java ecosystem, used to secure applications by providing:

- **Authentication** → Verifying *who* the user is.
- **Authorization** → Controlling *what* the user can access.
- **Protection** against attacks (CSRF, Session Fixation, Clickjacking, etc.)

### Key Principles

1. **Authentication Manager** – Handles user authentication.
2. **Security Context** – Stores security info (authenticated user).
3. **Filters** – A chain that intercepts requests and applies security.
4. **Role-Based Access Control (RBAC)** – Grants access based on roles (`ROLE_ADMIN`, `ROLE_USER`).
5. **Stateless Authentication with JWT** – Common for REST APIs.

# 3. 🔑 Spring Security – Authentication & Authorization

## Authentication – Who are you?

Spring Security provides multiple ways:

- **In-memory authentication**
- **Database-backed authentication**
- **LDAP / OAuth2 / JWT**

**Example – In-Memory Authentication**

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .requestMatchers("/user/**").hasAnyRole("USER", "ADMIN")
                .anyRequest().authenticated()
            )
            .httpBasic(); // Basic auth for simplicity
        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.withUsername("john")
                .password("{noop}password") // {noop} → no password encoding
                .roles("USER")
                .build();

        UserDetails admin = User.withUsername("admin")
                .password("{noop}admin123")
                .roles("ADMIN")
                .build();

        return new InMemoryUserDetailsManager(user, admin);
    }
}
```

# 4. 🎭 Role-Based Access Control (RBAC)

Access can be restricted based on roles using:

- **Configuration level** (`hasRole("ADMIN")`)
- **Method level** (`@PreAuthorize`, `@Secured`)

**Example – Method Level RBAC**

```
import org.springframework.security.access.prepost.PreAuthorize;

@Service
public class AdminService {

    @PreAuthorize("hasRole('ADMIN')")
    public String getAdminData() {
        return "Top-secret admin data!";
    }
}
```

**Enable with:**

```
@EnableMethodSecurity
@Configuration
public class MethodSecurityConfig {
}
```

# 5. JWT (JSON Web Token) – Stateless Authentication

## Why JWT?

- Traditional session-based auth requires server memory → not scalable for microservices.
- JWT is **stateless** → server only validates token signature.

## JWT Structure

HEADER.PAYLOAD.SIGNATURE

- **Header** → algorithm info (HS256, RS256)
- **Payload** → claims (username, roles, expiration)
- **Signature** → ensures integrity (HMAC + secret key)

## Workflow

1. User logs in with username/password.
2. Server authenticates and generates a JWT.
3. JWT is sent to the client (stored in localStorage/cookie).
4. For every request, client sends JWT in `Authorization: Bearer <token>`.
5. Server validates token before processing request.

**Login → Token Generated → Client Stores → Sends in Header → Server Validates**

## Example – JWT Filter

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Autowired
    private JwtUtil jwtUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
                                    throws ServletException, IOException {

        String header = request.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            String token = header.substring(7);
            String username = jwtUtil.extractUsername(token);

            if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
                if (jwtUtil.validateToken(token)) {
                    UsernamePasswordAuthenticationToken authToken =
                            new UsernamePasswordAuthenticationToken(username, null,
jwtUtil.getAuthorities(token));
                    SecurityContextHolder.getContext().setAuthentication(authToken);
                }
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

### Example – JWT Utility

```
@Component
public class JwtUtil {

    private final String SECRET_KEY = "mysecret123";

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

    public boolean validateToken(String token) {
        try {
            Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token);
            return true;
        } catch (JwtException e) {
            return false;
        }
    }

    public List<SimpleGrantedAuthority> getAuthorities(String token) {
        Claims claims =
Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
        String roles = claims.get("roles", String.class);
        return Arrays.stream(roles.split(","))
                .map(SimpleGrantedAuthority::new)
                .toList();
    }
}
```

# 6. 📝 Key Interview Pointers

- Difference between **Authentication vs Authorization**.
- Why **JWT is preferred** in microservices.
- How `@Valid` works with `@RequestBody`.
- Difference between `@NotNull`, `@NotEmpty`, `@NotBlank`.
- Role of `SecurityFilterChain` in Spring Security 6+.
- Method-level security with `@PreAuthorize`.
- How to handle validation errors (`@ControllerAdvice`).