

Buenas prácticas de seguridad en Java Spring Boot

16/06/2026 · **Fernando de la Torre** · Arquitecto de Software

En este post, realizaremos una breve introducción sobre buenas prácticas Java Spring Boot, aplicables tanto a APIs como a aplicaciones empresariales.

Contenido

- Configuración segura de Spring Security

- Validación estricta de los parámetros de entrada

- Protección frente a CSRF

 - API Rest

 - Aplicaciones con sesión y formularios

- Gestión segura de JWT

 - Usa claves robustas y seguras

- Expiración corta del Access Token

- Evita exponer datos sensibles (DTOs seguros)

- Seguridad en Base de Datos

 - Usar consultas preparadas o repositorios JPA

 - Limitar permisos del usuario de Base de Datos

 - NO almacenar credenciales de BD en application.properties en texto plano

 - Audita y registra accesos sospechosos

- Configuración segura de propiedades

- CORS: solo lo necesario

- Manejo correcto de errores

- HTTPS obligatorio

- Rate Limiting (Muy importante en APIs públicas)

 - Bucket4j — Rate Limiting dentro de Spring Boot

 - Spring Cloud Gateway — Rate Limiting nativo

- Content Security Policy (CSP)

- Auditoría y Logs seguros

- Pruebas de seguridad automáticas

 - OWASP Dependency Check (SCA)

 - SonarQube como herramienta SAST

- Conclusión

Configuración segura de Spring Security

En Spring Boot 3, la manera recomendada de configurar la seguridad es **declarando Beans explícitos**

Esto permite una configuración más **modular, funcional y explícita**, donde tú defines cómo se comporta la seguridad declarando Beans en una clase `@Configuration`.

```
@Configuration
@EnableMethodSecurity // opcional si usarás @PreAuthorize, etc.
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .httpBasic(); // o formLogin()

        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        UserDetails user = User.builder()
            .username("fernando")
            .password(passwordEncoder().encode("1234"))
            .roles("USER")
            .build();

        return new InMemoryUserDetailsManager(user);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Claramente usar la configuración basada en Beans en Spring Boot 3+ te permite:

- Definir explícitamente la seguridad de tu aplicación.
- Construir configuraciones más limpias y modulares.
- Adaptarte a los cambios introducidos por Spring Security 6.
- Preparar fácilmente entornos seguros: REST APIs, JWT, OAuth2, login forms, etc.

Validación estricta de los parámetros de entrada

Spring Boot integra **Bean Validation (Jakarta Validation)**, antes conocido como Hibernate Validator.

La validación es fundamental para:

- Evitar que entren datos corruptos o maliciosos
- Proteger de ataques como injections, desbordes o manipulación
- Garantizar integridad de datos

Supongamos que tenemos este DTO:

```
public class RegisterDTO {

    @NotBlank
    @Size(min = 3, max = 50)
    private String username;

    @NotBlank
    @Email
    private String email;

    @NotBlank
    @Size(min = 8)
    private String password;

    // getters & setters
}
```

Para validar el Dto en nuestro Controller seria usando @Valid en la firma de nuestro método, de la siguiente forma:

```
@PostMapping("/register")
public ResponseEntity<?> register(@Valid @RequestBody RegisterDTO dto) {
    return ResponseEntity.ok("Usuario creado");
}
```

Pero si lo usamos en un servicio (cuando el objeto no llega desde un controller), usaríamos @Validated añadido a nuestro @Valid

```
@Service
@Validated
public class UserService {

    public void process(@Valid RegisterDTO dto) {
        // lógica
    }
}
```

Aunque Spring Security tiene filtros XSS, **no protege automáticamente** todo contenido. Siempre debes sanitizar **entradas que luego se muestran en vistas**, logs o plantillas HTML.

Para asegurarnos que si el usuario escribe “<script>alert('XSS')</script>” no se ejecute, podemos tener algo así:

```
import org.owasp.encoder.Encode;

String safe = Encode.forHtml(userInput);
```

Protección frente a CSRF

CSRF (Cross-Site Request Forgery) es un ataque donde un atacante fuerza al navegador del usuario a enviar peticiones autenticadas sin su consentimiento.

- Afecta **solo** a aplicaciones **stateful** con **cookies de sesión**
- NO afecta APIs **stateless** con JWT, tokens o API Keys

API Rest

En una API REST **no hay sesiones** y **no se usan cookies** para autenticación.

Normalmente usas:

- JWT
- Tokens Bearer
- API Keys

Por lo tanto **CSRF no aplica**, y **debe desactivarse** para evitar errores 403.

```
@Bean
public SecurityFilterChain apiSecurity(HttpSecurity http) throws Exception {
    http
        .csrf(csrf -> csrf.disable())
        .sessionManagement(sess -> sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/auth/**").permitAll()
            .anyRequest().authenticated()
        )
        .httpBasic(Customizer.withDefaults());

    return http.build();
}
```

Esto hace que tengamos los siguientes beneficios:

- Reduce overhead
- Evita problemas con clientes externos (Postman, Angular, móvil)
- Simplifica la seguridad cuando usas JWT

Aplicaciones con sesión y formularios

Si la app usa:

- Login con formulario (formLogin)
- Cookies de sesión (JSESSIONID)
- Vistas como Thymeleaf, JSP, Freemarker

DEBES mantener CSRF activado.

Spring Security lo habilita por defecto y protege el POST/PUT/DELETE.

La configuración sería:

```
@Bean
public SecurityFilterChain webSecurity(HttpSecurity http) throws Exception {
    http
        .csrf(Customizer.withDefaults()) // habilitado por defecto
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/", "/login", "/public/**").permitAll()
            .anyRequest().authenticated()
        )
        .formLogin(Customizer.withDefaults());

    return http.build();
}
```

Y al formulario añades:

```
<input type="hidden" name="_csrf" value="...token...">
```

Gestión segura de JWT

A partir de Spring Security 6, lo más habitual es implementar JWT manualmente (filtros custom) o mediante Spring Authorization Server.

Usa claves robustas y seguras

La seguridad de un JWT depende **más de la clave** que del algoritmo.

Para **HS256/384/512 (HMAC)**:

- Usa claves de **mínimo 256 bits** (32 bytes)
- Usa claves generadas criptográficamente, NO contraseñas

Para **RS256/PS256/ECDSA**:

- Guarda claves privadas **fuera del repositorio**
- Usa **KeyStore**, AWS KMS, Azure KeyVault, Hashicorp Vault, etc.

```
@Bean
public SecretKey jwtKey() {
    return Keys.secretKeyFor(SignatureAlgorithm.HS256); // 256-bit key
}
```

Expiración corta del Access Token

Los **access tokens deben vivir poco**, porque si un atacante los roba, el daño es limitado.

Por ejemplo:

Access Token: 5–15 minutos

Refresh Token: 7–30 días (según nivel de sensibilidad)

```
String jwt = Jwts.builder()
    .subject(username)
    .issuedAt(new Date())
    .expiration(new Date(System.currentTimeMillis() + 10 * 60 * 1000)) // 10 min
    .signWith(jwtKey())
    .compact();
```

Evita exponer datos sensibles (DTOs seguros)

Nunca devuelvas entidades JPA completas, se deben crear Dtos que limiten los campos expuestos.

```
public record UserResponse(String id,String email,String role) {}
```

Seguridad en Base de Datos

La capa de persistencia es uno de los puntos más críticos de una aplicación. Protegerla implica evitar inyecciones, limitar permisos y asegurar el uso correcto del ORM.

Usar consultas preparadas o repositorios JPA

Spring Data JPA y Hibernate **ya ejecutan consultas preparadas internamente**, lo cual evita ataques de **SQL Injection**.

```
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
}

@Query(value = "SELECT * FROM users WHERE email = :email", nativeQuery = true)
User findByEmail(@Param("email") String email);

String sql = "SELECT * FROM users WHERE email = ?";
User user = jdbcTemplate.queryForObject(sql, new Object[]{email}, rowMapper);
```

Limitar permisos del usuario de Base de Datos

Este es uno de los puntos más ignorados y a la vez más importantes.

El usuario de BD NO debería tener privilegios sobre toda la base.

Usa el principio de **menor privilegio**:

- NO usar root, postgres, sa, admin, etc.
- Crear un usuario exclusivo para la app.
- Asignar solo los permisos estrictamente necesarios:
 - SELECT, INSERT, UPDATE, DELETE
- Evitar:
 - ALTER
 - DROP
 - CREATE
 - GRANT
 - SUPERUSER
 - Cualquier permiso DDL fuera de tiempo de migraciones.

NO almacenar credenciales de BD en application.properties en texto plano

Definir las credenciales en texto plano es un error, por ello es recomendable usar opciones más seguras:

- Variables de entorno
- Vault (Azure, AWS,...)
- Kubernetes secrets
- Spring Cloud Config + cifrado

```
spring.datasource.password=${DB_PASSWORD}
```

Audita y registra accesos sospechosos

Puedes activar auditoría en Hibernate (Activar solo en Desarrollo):

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

Configuración segura de propiedades

En Spring Boot, la gestión de propiedades es un punto crítico de seguridad: contraseñas, claves JWT, tokens de API, cadenas de conexión, etc.

Nunca deben quedar expuestas en ficheros `application.properties` o `application.yml`, en Repositorios Git y Logs. La mejor estrategia es extraer esa información fuera del código.

```
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USER}
spring.datasource.password=${DB_PASSWORD}

app.jwt.secret=${JWT_SECRET}
```

CORS: solo lo necesario

CORS **no es un mecanismo de autenticación**, sino un mecanismo de **seguridad del navegador** que evita que sitios maliciosos hagan peticiones AJAX a tu backend usando las credenciales del usuario.

Por eso, una mala configuración puede abrir la puerta a ataques como:

- **Credential Theft**
- **Session Hijacking**
- **CSRF combinados con APIs permisivas**
- **Robo de JWT almacenados en cookies HttpOnly**
- **Exfiltración de datos del backend hacia un dominio atacante**

```
cors.allowedOrigins("*");
```

Esto significa:

- Cualquier web puede hacer peticiones a tu backend
- No importa si es un sitio malicioso
- Los navegadores considerarán válido ese CORS
- Pueden usar las mismas cookies/headers (si configuras `.allowCredentials(true)`)

Que debemos hacer:

```
.allowedOrigins("https://tu-frontend.com")
```

Esto hace que **solo ese dominio específico** pueda llamar a tu API desde navegadores.

Ventajas:

- Evita que otros dominios accedan a tu API
- Aísla a posibles atacantes
- Cumple buenas prácticas de seguridad web
- Permite seguridad con cookies, JWT y sesiones

Para realizar una configuración segura de CORS en Spring Boot 3, podemos hacer lo siguiente:

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(List.of("https://tu-frontend.com"));
}
```

```

config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE"));
config.setAllowedHeaders(List.of("Authorization", "Content-Type"));
config.setAllowCredentials(true); // solo si usas cookies o JWT en cookies
config.setMaxAge(3600L); // cacheo

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
source.registerCorsConfiguration("/**", config);
return source;
}

```

Manejo correcto de errores

Un manejo de errores mal implementado puede **filtrar información sensible**, revelar detalles internos del sistema o dejar puertas abiertas para ataques.

Por eso, en seguridad **NUNCA debemos mostrar**:

- Stack traces completos
- Excepciones de bases de datos
- Mensajes internos de Spring, Hibernate o del servidor
- Información sobre rutas, clases o nombres de tablas

La solución es implementar un **Global Exception Handler**, que centraliza y controla cómo respondemos a errores.

```

@RestControllerAdvice
public class GlobalExceptionHandler {

    private static final Logger logger =
        LoggerFactory.getLogger(GlobalExceptionHandler.class);

    // Manejo de validaciones @Valid (seguro)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponse>
        handleValidation(MethodArgumentNotValidException ex) {

        List<String> errors = ex.getBindingResult()
            .getFieldErrors()
            .stream()
            .map(err -> err.getField() + ": " + err.getDefaultMessage())
            .toList();

        // Log interno con más detalle
        logger.warn("Error de validación: {}", errors);

        return ResponseEntity.badRequest().body(
            new ErrorResponse("Datos inválidos", errors)
        );
    }

    // Manejo de excepciones genéricas (sin filtrar detalles)
    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGeneralError(Exception ex) {

        // Log amplio SOLO en servidor
        logger.error("Error inesperado", ex);

        // Respuesta segura para el cliente
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(new ErrorResponse("Ha ocurrido un error inesperado"));
    }
}

```

Nuestro DTO para errores sería:

```
public class ErrorResponse {
    private String message;
    private List<String> details;

    public ErrorResponse(String message) {
        this.message = message;
    }

    public ErrorResponse(String message, List<String> details) {
        this.message = message;
        this.details = details;
    }

    // getters/setters
}
```

HTTPS obligatorio

Forzar HTTPS deja de ser opcional en aplicaciones modernas.

En Spring Boot 3, HTTPS es absolutamente obligatorio para proteger:

- credenciales
- JWT
- cookies (JSESSIONID, refresh tokens)
- datos sensibles en tránsito
- información personal o financiera
- APIs públicas y privada

Para activar el https se debe añadir en el `application.yml`

```
server.ssl.enabled=true
```

Rate Limiting (Muy importante en APIs públicas)

El **Rate Limiting** evita que tu API sea saturada por:

- Bots
- Ataques de fuerza bruta
- Scrapers
- Denial-of-Service (DoS) de baja intensidad
- Clientes abusivos

En APIs públicas, **es una medida de seguridad obligatoria**.

Aquí tienes una explicación sencilla, profesional y práctica de las dos soluciones más sólidas en el ecosistema Spring:

- Bucket4j (local o distribuido)
- Spring Cloud Gateway Rate Limit (Redis basado)

Bucket4j — Rate Limiting dentro de Spring Boot

Bucket4j es una de las librerías más potentes de rate limit que existen, con:

- Bucket Token algorithm (leaky bucket)
- Límite por IP, usuario, API key o endpoint
- Compatible con Redis, Hazelcast, PostgreSQL, etc.
- Uso directo en filtros de Spring Boot
- Alto rendimiento

Por ejemplo, configuremos un Rate Limit por IP en un filtro, con 100 peticiones por minuto, y lo rechaza con un HTTP 429 (Too Many Requests).

```
@Configuration
public class RateLimitConfig {

    private final Map<String, Bucket> cache = new ConcurrentHashMap<>();

    private Bucket createBucket() {
        return Bucket4j.builder()
            .addLimit(Bandwidth.classic(100, Refill.greedy(100, Duration.ofMinutes(1))))
            .build();
    }

    @Bean
    public Filter rateLimitFilter() {
        return (request, response, chain) -> {
            String ip = request.getRemoteAddr();
            Bucket bucket = cache.computeIfAbsent(ip, k -> createBucket());

            if (bucket.tryConsume(1)) {
                chain.doFilter(request, response);
            } else {
                response.setStatus(HttpStatus.TOO_MANY_REQUESTS.value());
                response.getWriter().write("Rate limit exceeded");
            }
        };
    }
}
```

Spring Cloud Gateway — Rate Limiting nativo

Si tu arquitectura usa **Spring Cloud Gateway**, puedes aplicar Rate Limit **en el edge**, antes de que el tráfico llegue a tus microservicios.

Esto además protege:

- Carga
- Seguridad
- Infraestructuras internas

Gateway usa **Redis** para almacenar los contadores.

Nuestro `application.yml` será así:

```
spring:
  cloud:
    gateway:
      routes:
        - id: public-api
          uri: http://localhost:8080
```

```

predicates:
- Path=/api/public/**
filters:
- name: RequestRateLimiter
  args:
    redis-rate-limiter.replenishRate: 50      # 50 req/segundo
    redis-rate-limiter.burstCapacity: 100    # Pico máximo
    key-resolver: "#{@ipResolver}"          # Por IP

```

Y definiremos un Bean:

```

@Bean
public KeyResolver ipResolver() {
    return exchange -> Mono.just(
        exchange.getRequest().getRemoteAddress().getAddress().getHostAddress()
    );
}

```

Content Security Policy (CSP)

CSP (Content Security Policy) es uno de los mecanismos **más potentes** para prevenir ataques XSS, inyección de scripts, clickjacking y carga de recursos maliciosos.

Aunque Spring Security no genera una CSP por defecto, **te permite configurarla fácilmente**.

Como activarlo en Spring Security:

```

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .headers(headers -> headers
            .contentSecurityPolicy(csp -> csp
                .policyDirectives("default-src 'self'; script-src 'self'; object-src 'none';")
            )
        );
    return http.build();
}

```

Para un frontend moderno como Angular, React o Vue deberemos crear una directiva parecida a este ejemplo:

```

.policyDirectives("""
    default-src 'self';
    script-src 'self';
    style-src 'self' 'unsafe-inline';
    img-src 'self' data:;
    font-src 'self';
    connect-src 'self' https://tu-api.com;
    frame-ancestors 'none';
    object-src 'none';
""")

```

Auditoría y Logs seguros

Debemos tener claro que datos podemos o no mostrar en nuestro logs. Nunca debemos mostrar contraseñas, Tokens, datos sensibles, ... ; en su lugar, podemos mostrar el usuario/sistema, la acción, resultado, el origen (IP, RequestId), etc.

Podemos crear un código para que todas nuestras peticiones se registren de la misma manera, con la misma información, así lo tenemos centralizado para toda la aplicación.

Primero crearemos nuestra clase:

```
@Aspect
@Component
public class AuditAspect {

    private final AuditService auditService;

    @Around("@annotation(Audited)")
    public Object auditMethod(ProceedingJoinPoint pjp) throws Throwable {
        try {
            Object result = pjp.proceed();
            auditService.audit(
                pjp.getSignature().getName(),
                "SUCCESS",
                "ok"
            );
            return result;
        } catch (Exception ex) {
            auditService.audit(
                pjp.getSignature().getName(),
                "ERROR",
                ex.getClass().getSimpleName()
            );
            throw ex;
        }
    }
}
```

Y luego su interfaz.

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Audited {}
```

Finalmente en el método que queramos registrar, le añadiremos la etiqueta @Audited y registrará automáticamente todas las peticiones

Pruebas de seguridad automáticas

La seguridad no debe ser manual ni tardía. En Spring Boot, una buena práctica es automatizar la detección de vulnerabilidades desde el código hasta las dependencias, integrándola en la pipeline de CI/CD.

Principalmente tenemos 3 tipos de pruebas: OWASP Dependency Check, SonarQube como herramienta SAST (Static Application Security Testing).

OWASP Dependency Check (SCA)

OWASP Dependency Check analiza las librerías usadas en tu proyecto Spring Boot y las compara con bases de datos de vulnerabilidades conocidas (NVD / CVE).

Para integrarlo en nuestro proyecto, nos iremos al pom.xml y añadimos este plugin:

```
<plugin>
  <groupId>org.owasp</groupId>
  <artifactId>dependency-check-maven</artifactId>
  <version>9.0.8</version>
```

```
<executions>
  <execution>
    <goals>
      <goal>check</goal>
    </goals>
  </execution>
</executions>
</plugin>
```

Para ejecutarlo, usaremos: mvn verify

SonarQube como herramienta SAST

SonarQube actúa como herramienta SAST, analizando el código fuente sin ejecutarlo para encontrar vulnerabilidades de seguridad, bugs, code smells,...

Algunas vulnerabilidades que detecta SonarQube a nivel de seguridad son:

- SQL Injection
- Path Traversal
- Configuración Insegura
- Uso de criptografía débil
- Hardcoded secrets
- Deserialización insegura
- Autorización incorrecta

Conclusión

La seguridad en Spring Boot debe aplicarse en todas las capas y de forma continua. Incluye buena configuración, control de accesos, validación de datos y protección frente a ataques. También requiere prácticas seguras en base de datos, uso de HTTPS, control de errores y limitación de peticiones. Finalmente, auditorías y herramientas automáticas ayudan a detectar fallos, logrando aplicaciones seguras alineadas con estándares.