

APIs REST y Autenticación con JWT

Arquitectura Stateless, Controladores, MapStruct
y Spring Security

Computación en Red 2 • Semana 11 • Universidad Icesi

Modelos de Comunicación: Stateful vs. Stateless

El Modelo Tradicional con Estado (Stateful)

Gestión en Memoria RAM

El Servidor Recuerda al Cliente

Almacena objetos de sesión en memoria RAM (`HttpSession`) asignados a un ID alfanumérico único devuelto en la cabecera `Set-Cookie` .

Cookie Puntero (JSESSIONID)

El navegador adjunta automáticamente la cookie en cada solicitud HTTP subsiguiente para localizar su bloque de memoria en el servidor web.

Desafíos en Arquitecturas Distribuidas

Escalabilidad Horizontal Limitada

La memoria RAM ocupada en el servidor crece linealmente con cada nuevo usuario conectado. Dificulta el soporte para cientos de miles de clientes.

Dependencia de Sticky Sessions

El balanceador de carga debe enrutar siempre al mismo nodo. Si ese servidor colapsa, todas las sesiones en su RAM se destruyen inmediatamente.

Implementación de Estado con HttpSession en Spring MVC

En aplicaciones MVC clásicas (Thymeleaf), `HttpSession` retiene los datos del usuario entre páginas sucesivas:

```
@Controller
public class UsuarioController {

    @PostMapping("/login")
    public String login(@RequestParam String nombre, HttpSession session) {
        session.setAttribute("usuario", nombre); // Guarda nombre en la RAM de Tomcat
        return "redirect:/perfil";
    }

    @GetMapping("/perfil")
    public String perfil(HttpSession session, Model model) {
        String usuario = (String) session.getAttribute("usuario"); // Busca sesión vía JSESSIONID
        if (usuario == null) return "redirect:/login";
        model.addAttribute("nombre", usuario);
        return "perfil";
    }
}
```

El Modelo Sin Estado (Stateless / REST)

Autonomía por Petición

Cero Memoria de Sesión en el Servidor

El backend no guarda ningún contexto entre llamadas sucesivas. Cada réplica solo valida la petición o su firma criptográfica al vuelo.

Toda la Información en la Solicitud

El cliente almacena y envía todos los parámetros, credenciales o token JWT en cada solicitud HTTP sin depender de almacenamiento previo en RAM.

Beneficios para la Nube y Microservicios

Escalabilidad Horizontal Transparente

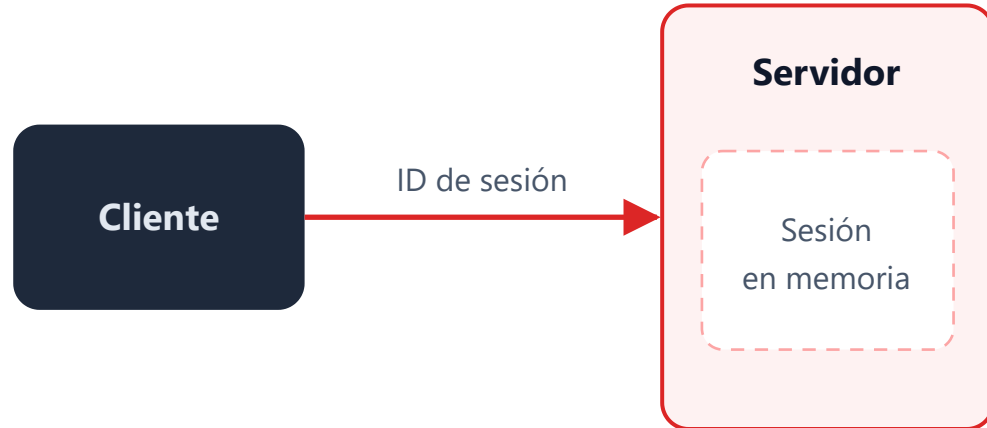
Permite escalar de 1 a 100 réplicas elásticas con balanceo *Round Robin* estándar sin requerir sincronización de memoria ni Sticky Sessions.

Alta Resiliencia ante Caídas

Si un servidor colapsa, cualquier otra réplica disponible atiende la siguiente solicitud del cliente sin pérdida de sesión ni interrupción del servicio.

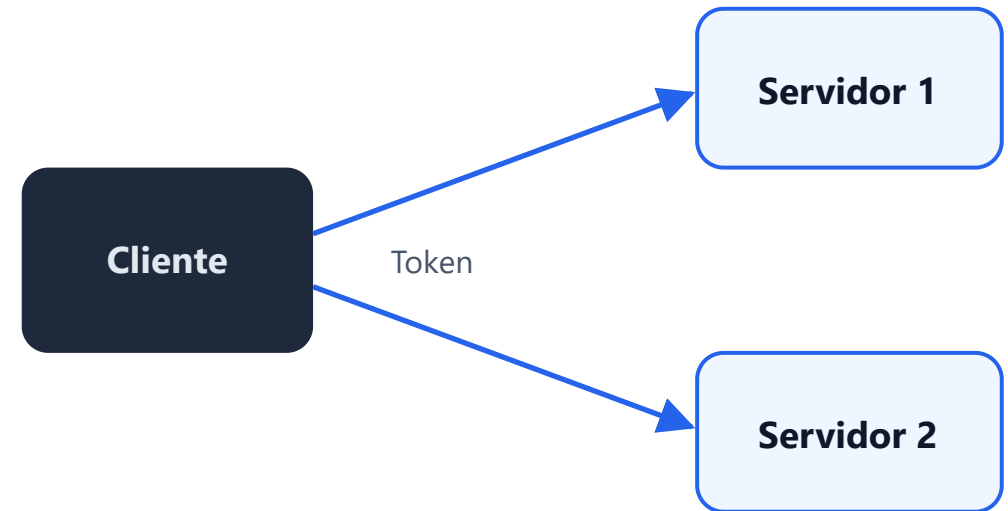
Stateful vs. Stateless en Sistemas Distribuidos

Stateful



El servidor recuerda al cliente

Stateless



Cada petición trae todo lo necesario

Principios Fundamentales de la Arquitectura REST

1. INTERFAZ UNIFORME

Identificación única de recursos mediante URIs estables y uso estándar de verbos HTTP.

2. STATELESS

El servidor no guarda estado del cliente. Cada petición HTTP contiene todo el contexto.

3. CACHEABLE

Las respuestas declaran explícitamente si son almacenables en caché (Cache-Control).

4. CLIENTE-SERVIDOR

Separación total entre la UI cliente (React, Móvil) y la lógica/persistencia del backend.

Métodos HTTP y Controladores REST en Spring Boot

Semántica de Métodos HTTP: Idempotencia y Seguridad

Propiedades Operativas Clave

01 Método Seguro (Read-Only)

No altera el estado del servidor ni produce efectos colaterales en la persistencia.

Verbos: GET • HEAD • OPTIONS

02 Método Idempotente

Ejecutar la misma petición N veces produce exactamente el mismo estado en el backend que ejecutarla 1 vez.

Verbos: GET • PUT • DELETE

Reglas de Oro del Diseño REST

03 Sustantivos en URIs, NUNCA Verbos

La URI representa el recurso; la acción la determina el método HTTP.

✓ **Correcto:** /api/usuarios

✗ **Incorrecto:** /api/obtenerUsuarios

04 Mutación de Recursos: PUT vs. PATCH

PUT : Reemplazo completo del recurso (idempotente).

PATCH : Modificación parcial atómica de atributos (no idempotente).

Matriz Semántica de Métodos y Códigos de Estado

MÉTODO	ACCIÓN SEMÁNTICA	IDEMPOTENTE	SEGURO
GET	Consulta y lectura de recursos	SÍ	SÍ
POST	Creación de nuevos recursos	NO	NO
PUT	Reemplazo total / Creación idemp.	SÍ	NO
PATCH	Actualización parcial de campos	NO	NO
DEL	Eliminación por URI de recurso	SÍ	NO

STATUS CODES CLAVE	
200 OK	GET / PUT exitosos
201	POST creación (+Location)
204	DELETE / sin cuerpo
400	Validación @Valid fallida
401	Sin token / Token inválido
403	Rol / Autoridad insuficiente
404	ID de recurso no existe

Mapeo de la Petición HTTP en Spring Boot

Petición HTTP Entrante

`/api/v1/productos/42`
Identificador primario en la URI

`?nombre=laptop&limite=10`
Parámetros opcionales / filtros

Authorization: **Bearer eyJ...**
Metadatos, tokens o cliente

`{ "nombre": "Monitor", "precio": 350.0 }`
Payload serializado en JSON

Anotaciones del Controlador REST

`@PathVariable("id") Long id`
Inyecta variable del path directamente en Java

`@RequestParam(defaultValue="20") int limite`
Parámetros de consulta con default o requeridos

`@RequestHeader("Authorization") String token`
Lectura de cabeceras de seguridad y cliente

`@RequestBody ProductoCreateDTO dto`
Jackson deserializa JSON → Objeto Java tipado

@RestController y @RequestMapping

El Contrato del Controlador REST

@RestController = @Controller + @ResponseBody

Instruye a Spring a serializar los valores de retorno directamente en el cuerpo HTTP en formato JSON mediante Jackson, sin buscar plantillas HTML como Thymeleaf.

Prefijo Global con @RequestMapping

Define la raíz institucional para todos los endpoints expuestos en la clase (por ejemplo, `/api/v1/productos`).

Código en Spring Boot

```
@RestController
@RequestMapping("/api/v1/productos")
public class ProductoController {

    @GetMapping
    public List<String> listar() {
        // Jackson convierte la lista de Java a JSON:
        // ["Monitor Curvo", "Teclado RGB"]
        return List.of("Monitor Curvo", "Teclado RGB");
    }
}
```

@PathVariable vs. @RequestParam

@PathVariable (Variables de Ruta)

Indispensable para identificar el recurso en la jerarquía URI (ej: `/productos/42`):

```
@GetMapping("/{id}")
public String obtener(@PathVariable Long id){
    return "Producto con ID: " + id;
}
```

@RequestParam (Query String)

Filtros opcionales, búsqueda y paginación (ej: `/productos/buscar?nombre=teclado&limite=10`):

```
@GetMapping("/buscar")
public String buscar(
    @RequestParam String nombre,
    @RequestParam(required = false) String cat,
    @RequestParam(defaultValue = "20") int limite){
    return "Filtro: " + nombre + ", lim: " + limite;
}
```

Recepción de Payloads JSON (@RequestBody) y Headers

@RequestBody y Deserialización

Jackson mapea el cuerpo JSON entrante a un DTO de Java:

```
@PostMapping
public ResponseEntity<String> crear(
    @RequestBody ProductoCreatedDTO dto) {
    // Jackson deserializa las claves JSON
    return ResponseEntity
        .status(HttpStatus.CREATED)
        .body("Creado: " + dto.getNombre());
}
```

@RequestHeader (Metadatos)

Captura cabeceras estándar o personalizadas de auditoría:

```
@GetMapping("/auditoria")
public String auditoria(
    @RequestHeader("User-Agent") String agent,
    @RequestHeader(value = "X-Canal",
        defaultValue = "WEB") String canal) {
    return "Cliente: " + agent + ", Canal: " + canal;
}
```

Control Total de Respuestas con ResponseEntity

Los 3 Componentes de una Respuesta HTTP

01 Código de Estado HTTP

200 OK, 201 Created, 204 No Content, 400 Bad Request, 404 Not Found.

02 Cabeceras HTTP de Respuesta

`Location` para nuevos recursos creados, `Cache-Control`, firmas de auditoría.

03 Cuerpo de la Respuesta (Payload)

DTO o colección serializada en formato JSON hacia el cliente.

API Fluida en Java

```
@GetMapping("/{id}")
public ResponseEntity<ProductoDTO> detalle(
    @PathVariable Long id) {
    if (id <= 0) {
        return ResponseEntity.badRequest().build(); // 400
    }
    ProductoDTO prod = servicio.buscarPorId(id);

    return ResponseEntity.ok() // 200 OK
        .header(HttpHeaders.CACHE_CONTROL, "max-age=60")
        .body(prod);
}
```

El Patrón DTO y Aislamiento de Entidades

¿Por qué NO Exponer la @Entity en el Controlador?

Zona de Persistencia: Riesgos en @Entity

```
@Entity
public class Usuario {
    @Id private Long id; // [Riesgo 3: Mass Assignment]
    private String nombre;
    private String email;
    private String passwordHash; // [Riesgo 1: Fuga de Secretos]

    @OneToMany
    private List<Pedido> pedidos; // [Riesgo 2: Bucle Jackson]
}
```

- 1. Fuga de Secretos:** La serialización directa expone hashes en JSON.
- 2. Bucle Jackson:** Usuario → Pedidos → Usuario causa *StackOverflowError*.
- 3. Mass Assignment:** Cliente malicioso envía `{"rol": "ADMIN"}`.

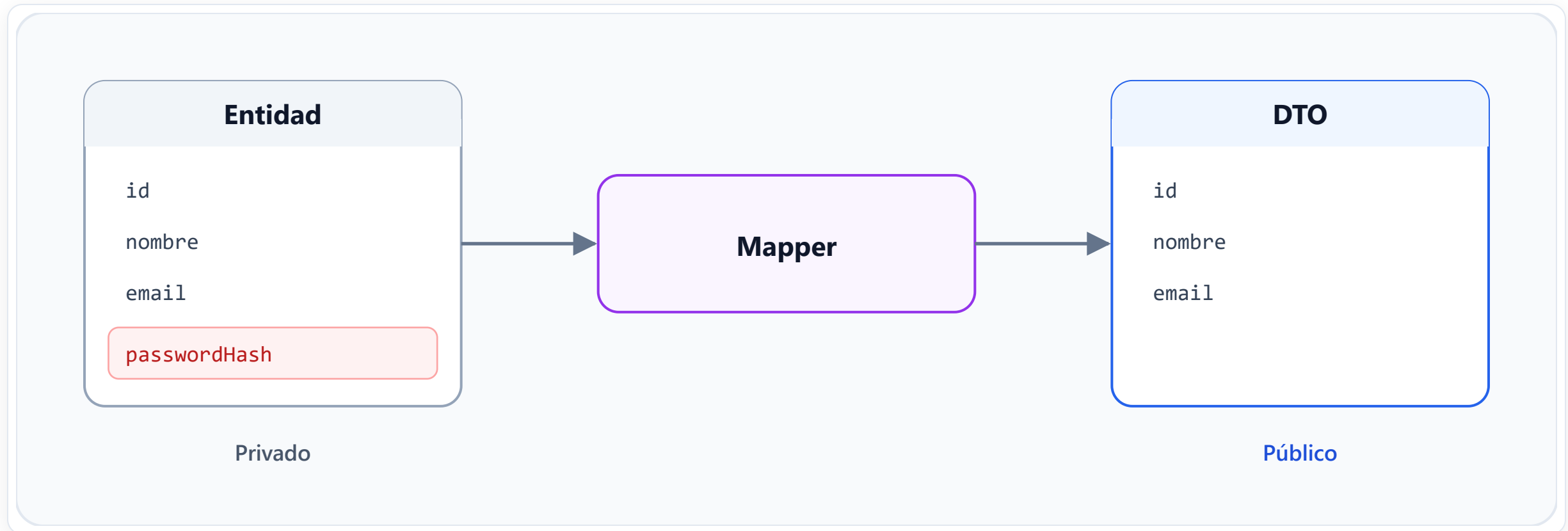
Zona Pública REST: La Solución con DTO

```
public class UsuarioResponseDTO {
    private Long id;
    private String nombre;
    private String email;
    // CERO contraseñas, CERO relaciones circulares
}
```

Garantías de la Barrera de Aislamiento:

- **Proyección Limpia:** Solo expone datos explícitamente autorizados.
- **Serialización Segura:** Jackson procesa campos planos sin bucles.
- **Contrato Estable:** Cambios en tablas SQL no rompen la API.

Patrón DTO: Zona Privada de Datos vs. API Pública



Especialización: Request DTOs vs. Response DTOs

Request DTO (Datos de Entrada)

Captura datos del cliente e incluye anotaciones de validación:

```
public class UsuarioCreateDTO {  
    @NotBlank(message = "Nombre obligatorio")  
    private String nombre;  
  
    @Email(message = "Email no válido")  
    private String email;  
  
    @Size(min = 8, message = "Mínimo 8 caracteres")  
    private String password;  
}
```

Response DTO (Datos de Salida)

Proyección limpia y segura: **cero contraseñas o datos internos**:

```
public class UsuarioResponseDTO {  
    private Long id;  
    private String nombre;  
    private String email;  
    private List<String> roles;  
  
    // Constructor, getters y setters  
}
```

MapStruct: Compilación Estática vs. Reflexión en Runtime

CRITERIO ARQUITECTÓNICO	MODELMAPPER / REFLEXIÓN EN RUNTIME	MAPSTRUCT / PROCESADOR DE ANOTACIONES
Momento de Mapeo	En tiempo de ejecución (en cada solicitud HTTP)	Durante la compilación (javac con Maven)
Rendimiento	Lento: inspección reflexiva en cada llamada	Velocidad nativa pura: getters y setters directos
Detección de Errores	Tardía: falla en producción con RuntimeExceptions	Temprana: fallo inmediato en build si un campo no coincide
Impacto en Producción	Sobrecarga de memoria y CPU por introspección	Cero dependencias pesadas en el JAR ejecutable

Pipeline de Generación de Código con MapStruct



Definición de Mappers e Inyección en Servicios

1. Interfaz del Mapper

```
@Mapper(componentModel = "spring")
public interface UsuarioMapper {

    // Copia id, nombre, email de forma automática
    UsuarioResponseDTO toDTO(Usuario entidad);

    // Mapeo avanzado con renombrado e ignore
    @Mapping(source = "titulo", target = "nombreProyecto")
    @Mapping(target = "passwordSecreta", ignore = true)
    ProyectoDTO toProyectoDTO(Proyecto p);
}
```

2. Inyección en el Servicio

```
@Service
public class UsuarioService {

    @Autowired
    private UsuarioRepository usuarioRepo;

    @Autowired
    private UsuarioMapper usuarioMapper; // Inyecta Impl

    public UsuarioResponseDTO buscar(Long id) {
        Usuario u = usuarioRepo.findById(id)
            .orElseThrow();
        return usuarioMapper.toDTO(u); // Seguro
    }
}
```

Fundamentos y Arquitectura de JSON Web Tokens (JWT)

¿Qué es JWT y por qué se utiliza en REST?

El Estándar RFC 7519

JSON Web Token (JWT)

Estándar abierto que define una forma compacta y autónoma (*self-contained*) para transmitir información estructurada de forma segura entre partes como un objeto JSON.

Transporte Compacto

Por su formato de texto delimitado por puntos (.), viaja en la cabecera estándar `Authorization: Bearer <token>` sin sobrecargar el ancho de banda.

Propiedades Arquitectónicas

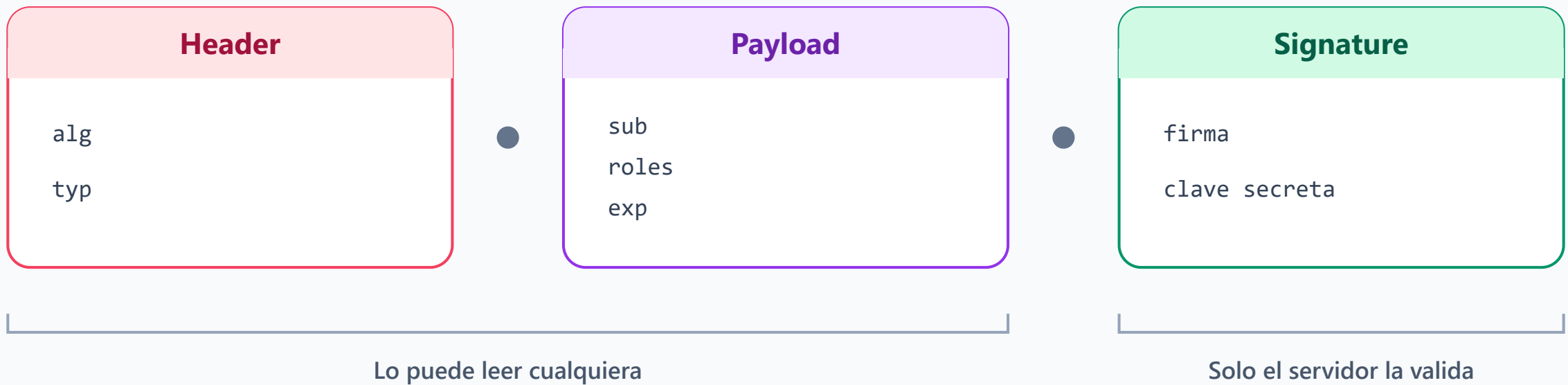
Autónomo (Self-Contained)

Transporta la identidad (`sub`), roles y caducidad (`exp`). El backend no necesita consultar RAM ni tablas de sesión para identificar al cliente.

Verificable Criptográficamente

Cuenta con una firma digital generada con una clave secreta que reside únicamente en el servidor. Garantiza que los datos no han sido alterados en tránsito.

Anatomía Criptográfica de un JSON Web Token



Desglose Interno: Header, Payload y Signature

1. Header

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Metadatos: algoritmo criptográfico (HMAC con SHA-256) y tipo de credencial.

2. Payload (Claims)

```
{  
  "sub": "juan",  
  "roles": ["ROLE_USER"],  
  "iat": 1773517905,  
  "exp": 1773604305  
}
```

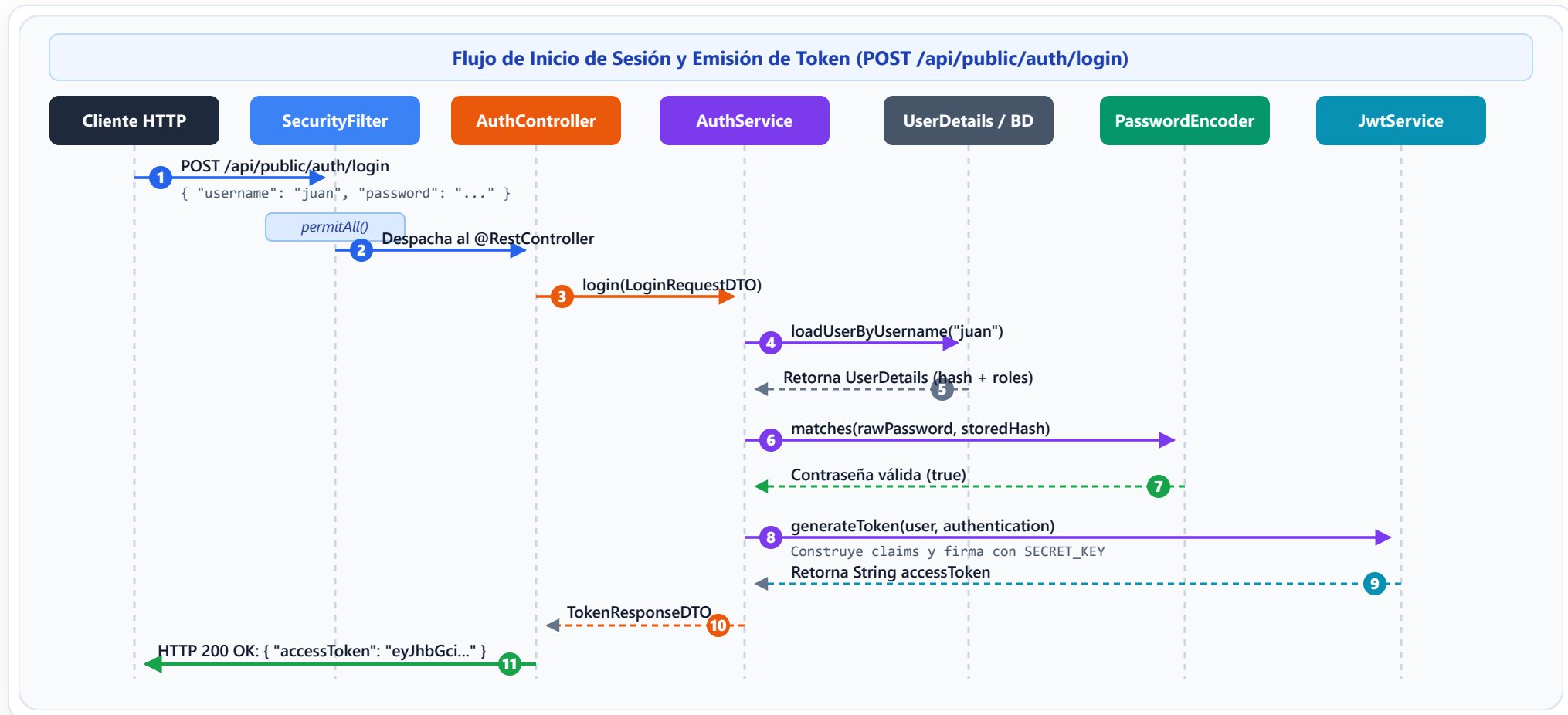
¡No está cifrado! Solo codificado en Base64Url. Jamás guardar contraseñas aquí.

3. Signature

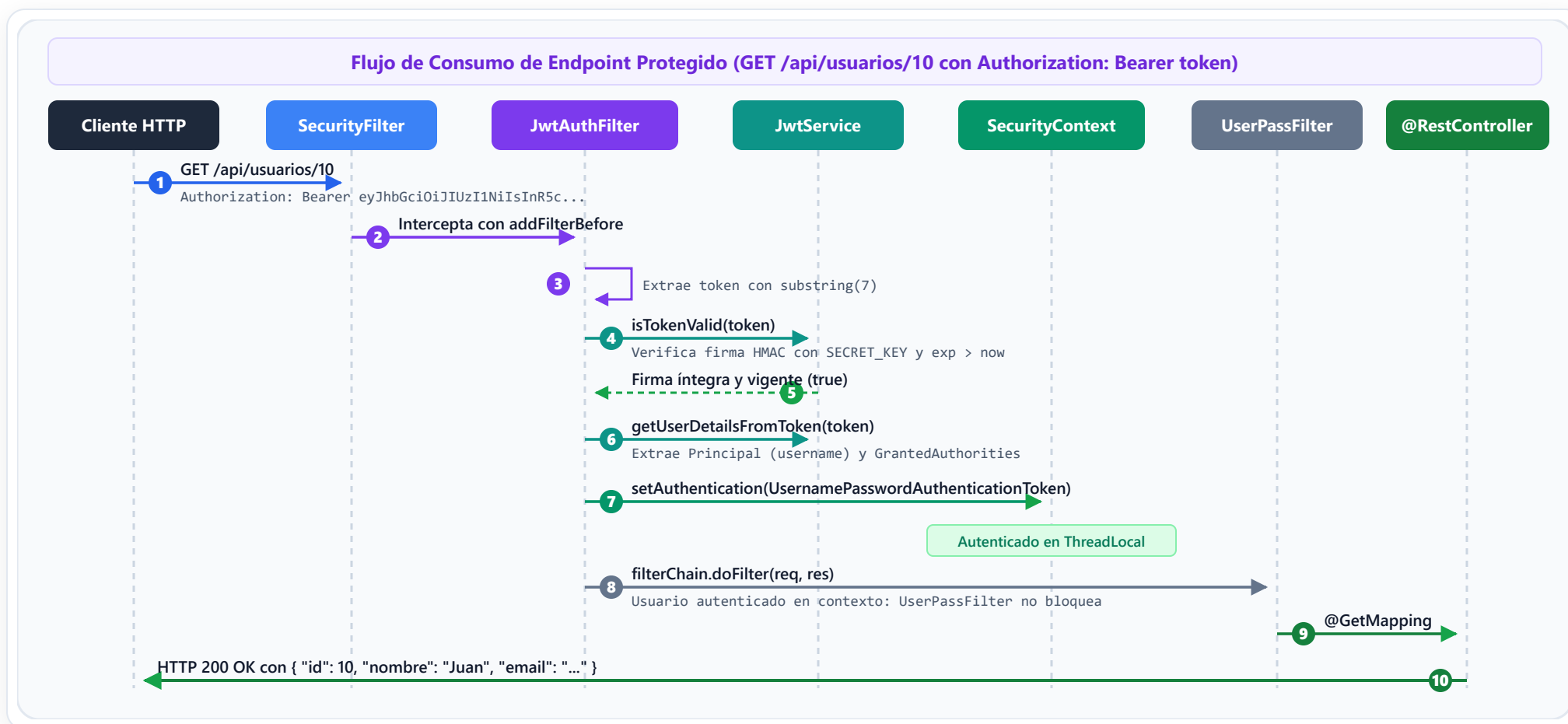
```
HMACSHA256(  
  base64Url(header) +  
  "." +  
  base64Url(payload),  
  SECRET_KEY  
)
```

Garantiza integridad. Si se altera un solo bit del payload, la firma deja de coincidir.

Flujo 1: Inicio de Sesión y Emisión de Token (Login)



Flujo 2: Consumo de Endpoints Protegidos (Bearer Token)



Síntesis: Emisión en Login vs. Validación al Vuelo

Fase 1: Emisión (Login)

Verificación con Base de Datos

- `UserDetailsService` carga la entidad desde la base de datos.
- `PasswordEncoder.matches()` valida la contraseña contra el hash BCrypt.
- `JwtService` genera el token firmado con la clave secreta.
- **Resultado:** Retorna HTTP 200 OK con `{ "accessToken": "..."} .`

Fase 2: Validación (Por Petición)

Validación Cero Memoria / Cero DB

- `JwtAuthenticationFilter` extrae el token del encabezado Bearer.
- Recalcula la firma HMAC con `SECRET_KEY` y comprueba `exp > now`.
- Registra al usuario autenticado en `SecurityContextHolder` (ThreadLocal).
- **Resultado:** Pasa directo a `@RestController` sin tocar la BD.

Arquitectura de Filtros: ¿Por qué addFilterBefore?

La Configuración en Spring Security

```
http.addFilterBefore(  
    jwtAuthenticationFilter(),  
    UsernamePasswordAuthenticationFilter.class  
);
```

Filtro por Defecto de Spring

`UsernamePasswordAuthenticationFilter` está diseñado para autenticación tradicional basada en formularios HTML (POST /login) y sesiones con cookies.

Motivo Técnico Crucial

Interceptar ANTES del Bloqueo

Queremos que nuestro filtro especializado en JWT capture el encabezado `Bearer <token>` y pueble el `SecurityContextHolder` **antes** de que cualquier filtro intente redirigir o considerar anónima la petición.

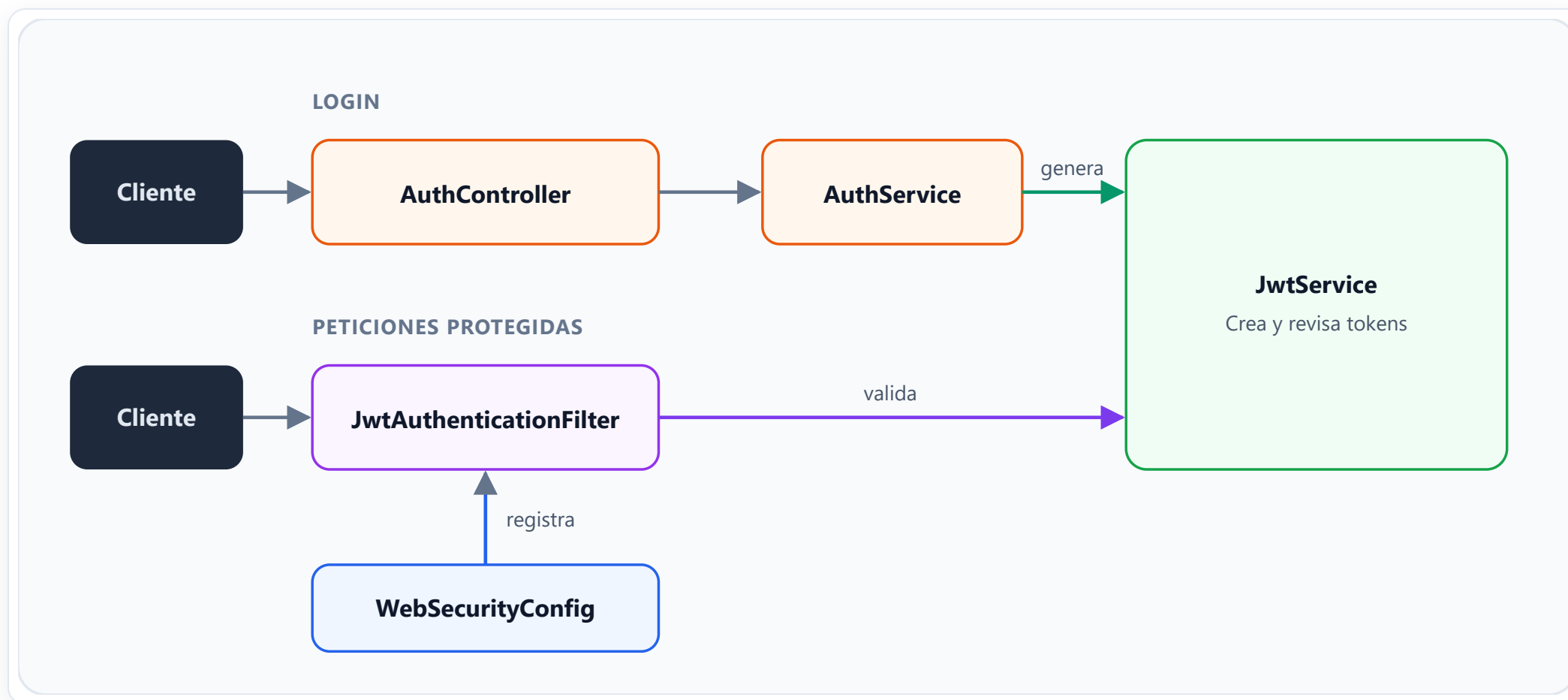
Autorización Inmediata

Al ejecutarse antes, los filtros posteriores encuentran al usuario y sus roles listos en memoria para evaluar permisos de acceso en las rutas.

Implementación de Seguridad Stateless en Spring Boot

3

Arquitectura de Clases e Inyección de Dependencias



Pasos 1 y 2: Dependencias JWT y Clave Secreta

1. Dependencias en pom.xml (JWT 0.12+)

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.12.3</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.12.6</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.12.6</version>
  <scope>runtime</scope>
</dependency>
```

2. application.properties

```
# Clave secreta simétrica HMAC (mínimo 256 bits)
app.security.jwt.secret-key=dGhpc0lzQVZlcnlTZWN1cmVTZWNYZXRLZXIGb3JKV1RBdXRoZW50aWNhdGlvbkluU3ByaW5nQm9vdDlwMjU=

# Tiempo de caducidad en ms (86400000 = 24h)
app.security.jwt.expiration-time=86400000
```

En Producción:

Injectar siempre como variable de entorno: `${JWT_SECRET}` . Jamás commitar secretos en texto plano al repositorio Git.

Paso 3: Servicio Criptográfico JwtServiceImpl

Generación del Token

```
@Override
public String generateToken(User user, Authentication auth){
    List<String> roles = auth.getAuthorities().stream()
        .map(GrantedAuthority::getAuthority).toList();

    return Jwts.builder()
        .id(user.getId().toString())
        .claims(Map.of("email", user.getEmail(),
            "authorities", roles))
        .subject(user.getUsername())
        .issuedAt(new Date())
        .expiration(new Date(System.currentTimeMillis() + exp))
        .signWith(getSignInKey()) // HS256
        .compact();
}
```

Validación y Verificación de Firma

```
@Override
public boolean isValidToken(String token){
    try{
        Jwts.parser()
            .verifyWith(getSignInKey()) // Valida firma
            .build()
            .parseSignedClaims(token);

        return !isTokenExpired(token);
    } catch (Exception e){
        return false; // Firma inválida o expirado
    }
}
```

Paso 4: Filtro Interceptor JwtAuthenticationFilter

Extracción y Verificación

```
@Component
public class JwtAuthenticationFilter
    extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(
        HttpServletRequest req, HttpServletResponse res,
        FilterChain chain) throws Exception {

        String header = req.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            String token = header.substring(7);
            if (jwtService.isValid(token)) {
                UserDetails ud = jwtService.getUserDetailsFromToken(token);
                // Crea Authentication y registra en contexto
                ...
            }
        }
        chain.doFilter(req, res);
    }
}
```

Inyección en SecurityContextHolder

```
//Reconstrucción del token de autenticación
UsernamePasswordAuthenticationToken auth =
    new UsernamePasswordAuthenticationToken(
        ud, null, ud.getAuthorities()
    );

auth.setDetails(
    new WebAuthenticationDetailsSource().buildDetails(req)
);

// Usuario autenticado en el hilo actual (ThreadLocal)
SecurityContextHolder.getContext().setAuthentication(auth);
```

Aislado por Hilo:

`SecurityContextHolder` garantiza que cada solicitud HTTP procese únicamente los claims de su cliente en un entorno multi-hilo.

Paso 5: Configuración de la Cadena WebSecurityConfig

Configuración Stateless (/api/**)

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class WebSecurityConfig {

    @Bean
    public SecurityFilterChain restFilterChain(HttpSecurity http)
        throws Exception {
        return http
            .securityMatcher("/api/**")
            .csrf(csrf -> csrf.disable()) // Stateless API
            .cors(cors -> cors.configurationSource(...))
            .authorizeHttpRequests(authz -> authz
                .requestMatchers("/api/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .addFilterBefore(jwtFilter(),
                UsernamePasswordAuthenticationFilter.class)
            .sessionManagement(s -> s.sessionCreationPolicy(
                SessionCreationPolicy.STATELESS))
            .build();
    }
}
```

Elementos Críticos

`csrf().disable()`

Innecesario en APIs REST stateless que no utilizan cookies de sesión vulnerables a ataques de falsificación de peticiones en sitios cruzados.

`SessionCreationPolicy.STATELESS`

Instruye a Spring a jamás crear ni utilizar un objeto `HttpSession` en la memoria RAM del contenedor de Servlets.

Paso 6: DTOs, AuthServiceImpl y AuthController

AuthServiceImpl

```
@Override
public TokenResponseDTO login(LoginRequestDTO req) {
    UserDetails ud = userDetailsService
        .loadUserByUsername(req.getUsername());

    if (!passwordEncoder.matches(req.getPassword(),
        ud.getPassword())) {
        throw new RuntimeException("Credenciales inválidas");
    }

    User user = ((CustomUserDetails) ud).getUser();
    Authentication auth = new UsernamePasswordAuthenticationToken(
        ud, "", ud.getAuthorities());

    String token = jwtService.generateToken(user, auth);
    return new TokenResponseDTO(token);
}
```

AuthController (@RestController)

```
@RestController
@RequestMapping("/api/public/auth")
public class AuthController {

    @Autowired
    private IAuthService authService;

    @PostMapping("/login")
    public ResponseEntity<?> login(
        @RequestBody LoginRequestDTO req) {
        try {
            TokenResponseDTO token = authService.login(req);
            return ResponseEntity.ok(token); // 200 OK
        } catch (RuntimeException e) {
            return ResponseEntity
                .status(HttpStatus.UNAUTHORIZED) // 401
                .body(e.getMessage());
        }
    }
}
```

Los 3 Escenarios Críticos de Seguridad en Producción

AMENAZA 1 Token Robado

Interceptación en red (MITM)
o robo vía XSS en el navegador del cliente.

Contramedidas en Producción:

- Cifra el canal.
- Reduce ventana.
- Revocables.
- Inmunes a JavaScript.

AMENAZA 2 Falsificación

Atacante altera el Payload
cambiando "ROLE_USER" por "ROLE_ADMIN".

Garantía Criptográfica:

- Cada bit importa.
- Imposible recalcular.
- JWT rechaza.
- Bloqueo inmediato.

AMENAZA 3 Stale Claims

Rol revocado en BD a las 10:05
pero token sigue vigente hasta las 11:00.

Estrategias Arquitectónicas:

- Desfase de 10-15 min.
- Valida en BD en filtro.
- Invalidación iat.
- Trade-off.

Estrategias Arquitectónicas para Mitigar Stale Claims

ESTRATEGIA	MECANISMO TÉCNICO	LATENCIA EXTRA	RECOMENDACIÓN PARA PRODUCCIÓN
A: Expiración Corta (15 min)	Token de acceso expira rápido. Renovación vía Refresh Token.	0 ms (Validación en memoria)	Estándar para APIs y Microservicios de alto tráfico. Desfase temporal despreciable.
B: Verificación Ligera en Filtro	<code>loadUserByUsername()</code> en cada request dentro del filtro JWT.	+3 a 8 ms (Consulta SQL a BD)	Recomendado para monolitos con requerimiento de revocación en tiempo real estricto.
C: Lista Negra en Redis	Almacena timestamp de revocación. Rechaza tokens con <code>iat < ts</code> .	+0.8 ms (Lectura RAM Redis)	Arquitectura de Grado Bancario / Fintech. Consistencia inmediata sin saturar SQL.