UT3 - Desarrollo de aplicaciones seguras

> **Resultado de aprendizaje 5:** Protege las aplicaciones y los datos definiendo y aplicando criterios de seguridad en el acceso, almacenamiento y transmisión de la información

- **a)** Identificar y aplicar principios y prácticas de programación segura
- **b)** Analizar las principales técnicas y prácticas criptográficas
- **c)** Definir e implantar políticas de seguridad para el acceso de los usuarios
- **d)** Utilizar esquemas de seguridad basados en roles
- **e)** Emplear algoritmos criptográficos para proteger la información almacenada.
- **f)** Identificar métodos para asegurar la información transmitida
- **g)** Desarrollar aplicaciones que utilicen sockets seguros para la transmisión de información
- **h)** Depurar y documentar las aplicaciones desarrolladas

Instrucciones iniciales

Adjunto a la unidad tenemos una **plantilla de proyecto Maven** llamada `ut3-ejercicios-plantilla`.
Esta plantilla ya incluye un servicio rest de /empresas, a continuación debemos **configurar la base de datos y JPA** correctamente.

1. Crear el esquema en MySQL

Este proyecto utiliza un esquema de bd llamado `pspr-bd`.

2. Revisar / Configurar JPA en el proyecto

En el proyecto ya existe la estructura básica.
Tu tarea es **comprobar y ajustar** la configuración de JPA para que funcione correctamente con el esquema `pspr-bd`.

Dependencias necesarias (`pom.xml`)

Abre el archivo `pom.xml` y asegúrate de que están incluidas estas dependencias:

```
<!-- Spring Data JPA -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<!-- Driver MySQL -->
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
```

```
</dependency>
```

Configuración del DataSource (`application.properties`)

Configura la conexión apuntando al esquema `pspr-bd` :

```
spring.datasource.url=jdbc:mysql://localhost:3306/pspr-bd?useSSL=false&serverTimezone=Europe/Madrid
spring.datasource.username=TU_USUARIO
spring.datasource.password=TU_PASSWORD
```

Estrategia de creación / actualización de tablas

Activa **Hibernate DDL** auto-update para que **cree/actualice automaticamente las tablas según las entidades del modelo** añadiendo en `application.properties` :

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Observa la estructura del servicio web REST de empresas

En el proyecto tienes un ejemplo completo de **capas** ya montado para la entidad `Empresa` .
Fíjate en estas clases:

- **Entidad (Model):** `Empresa`
  Representa la tabla `empresa` en la base de datos.
  Aquí se definen los atributos (campos) de la empresa: `id` , `nombre` , etc.
  Es la clase que se mapea con JPA a la BD mediante anotaciones como `@Entity` , `@Id` , etc.
- **Repositorio JPA:** `EmpresaRepository`
  Es una interfaz que extiende de `JpaRepository` (u otra interfaz de Spring Data JPA).
  Se encarga de las operaciones de acceso a datos: guardar, buscar, borrar, listar empresas…
  Tú no programas el SQL: Spring Data genera las consultas básicas automáticamente.
- **Servicio:** `EmpresaService`
  Contiene la **lógica de negocio** relacionada con las empresas.
  - Llama al `EmpresaRepository` para acceder a la BD.
  - Aplica validaciones o reglas antes de guardar o devolver datos.
    Esta capa evita que el controlador trabaje directamente con el repositorio.
- **Controlador:** `EmpresaController`
  Expone una **API REST** (endpoints) para trabajar con empresas.
  - Recibe las peticiones HTTP (GET, POST, PUT, DELETE).
  - Llama a `EmpresaService` para realizar las operaciones.
  - Devuelve las respuestas HTTP (JSON) al cliente.

Spring y anotaciones básicas

En el proyecto vas a ver varias **anotaciones de Spring** que indican a qué capa pertenece cada clase y cómo se gestionan los objetos (beans).

Estereotipos de Spring

- `@Component`
  Marca una clase como **componente genérico** de Spring.
  Spring la detecta en el escaneo de paquetes y crea un **bean** de esa clase.
- `@Service`
  Variante de `@Component` pensada para la **capa de servicio** (lógica de negocio).
  Es lo que usaremos, por ejemplo, en `EmpresaService`.
- `@Configuration`
  Indica que una clase contiene **configuración de Spring**.
  Dentro de ella suele haber métodos anotados con `@Bean`.
- `@Bean`
  Se usa sobre un **método** dentro de una clase `@Configuration`.
  El método devuelve un objeto que Spring registra como **bean** en el contenedor.
  Ejemplo típico: definir un `PasswordEncoder`.

Spring crea los objetos (beans) por ti y te los "inyecta" donde los necesitas, usando anotaciones

Inyección de dependencias

Spring puede **inyectar automáticamente** los beans donde se necesitan.

- `@Autowired`
  Se usa para indicar que una dependencia debe ser inyectada por Spring.
  - sobre un **campo** (atributo de la clase):

```
    @Autowired
    private final EmpresaRepository empresaRepository;
```

- Inyeccion en el **constructor** (recomendada):

```
@Service
 public class EmpresaService {

    private final EmpresaRepository empresaRepository;
    // Inyecta empresaRepository//
    public EmpresaService(EmpresaRepository empresaRepository) {
        this.empresaRepository = empresaRepository;
    }
}
```

Manejo seguro de errores en APIs REST

En esta unidad añadimos un **formato estándar de error** y un **método de manejo dentro del controller**.

Clase `ErrorResponse`

Dentro del paquete **com.pspr.security.dto** creamos una clase que utilizaremos para devolver errores de una manera estandarizada.

DTO(Data Transfer Object) A menudo se identifican estas clases como dto, son clases que no pertenecen al modelo de negocio (model) y se usan para transferir datos en un objeto

```
public class ErrorResponse {
    private int status;
    private String error;
    private String message;
    private String path;

    public ErrorResponse(int status, String error, String message, String path) {
        this.status = status;
        this.error = error;
        this.message = message;
        this.path = path;
    }
    // getters/setters...
}
```

Manejo desde el controller

Desde el controller, si todo va bien → devuelve la entidad normal.

- Si hay un error interno → JSON 500.

```
 // --- MANEJO DE ERRORES ---

    @ExceptionHandler(RuntimeException.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public ErrorResponse generic(RuntimeException ex,
                                 HttpServletRequest req) {
        return new ErrorResponse(500, "Internal Server Error",
                "Error interno. Inténtelo más tarde.", req.getRequestURI());
    }
```

📝 Ejercicio 1: Maneja errores

Puedes implementar tus propias excepciones para manejar errores de la lógica de negocio.

Por ejemplo, en caso de que al buscar una empresa por Id no se encuentre, se genere la excepción *RecursoNoEncontradoException*

```
// Mi propia Excepcion //

public class RecursoNoEncontradoException extends RuntimeException {
    public RecursoNoEncontradoException(String msg) {
        super(msg);
    }
}
```

```
// Desde el controller ...

 @ExceptionHandler(RecursoNoEncontradoException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ErrorResponse notFound(RecursoNoEncontradoException ex,
                                  HttpServletRequest req) {
        return new ErrorResponse(404, "Not Found", ex.getMessage(), req.getRequestURI());
    }
```

- Si el recurso no existe → JSON 404. Respuesta:

```
{
  "status": 404,
  "error": "Not Found",
  "message": "Empresa 12 no existe",
  "path": "/empresas/12"
}
```

Crear el servicio de Usuarios y Roles para gestionar la seguridad

📌 Criterios trabajados: **a**

Crear las entidades del *model*

Vas a crear las clases del **modelo** que representarán a los usuarios y sus roles en la base de datos.

Entidad `User` (implementa `UserDetails`)

- Crea una clase `User` en el paquete `model`.
- Esta clase representará la **tabla** `user` en la BD.
- Debería estar anotada con `@Entity` y `@Table(name = "user")`.
- Implementa la interfaz `org.springframework.security.core.userdetails.UserDetails`.

Al implementar `UserDetails`, tendrás que sobrescribir:

- `getUsername()`
- `getPassword()`
- `getAuthorities()` → devolverá los roles.
- `isEnabled()`
- `isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentialsNonExpired()` (pueden devolver `true` al principio)

```
@Entity
@Table(name = "user")
public class User implements UserDetails {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String password;

    // metodos UserDetails (nos obliga a implementarlos)
    // Damos una implementacion basica

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return null; // de momento retorna null ...
    }

    @Override
    public String getUsername() { return this.username; }

    @Override
    public String getPassword() { return this.password; }

    @Override
    public boolean isAccountNonExpired() { return true; }

    @Override
    public boolean isAccountNonLocked() { return true; }

    @Override
    public boolean isCredentialsNonExpired() { return true; }

    @Override
    public boolean isEnabled() { return true; }

    // getters / setters
}
```

```
return roles.stream()
        .map(r -> new SimpleGrantedAuthority
(r.getName()))
        .collect(Collectors.toList());
```

Entidad `Role`

- Crea una clase `Role` en el mismo paquete `model`.
- Esta clase representará la **tabla** `role`.
- Serán los roles que pueden tener los usuarios. (por ejemplo: `ROLE_USER`, `ROLE_ADMIN`)

```
@Entity
@Table(name = "role")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(nullable = false, unique = true)
    private String name;   // 'ROLE_ADMIN', 'ROLE_USER', etc.
```

```
    // getters / setters
}
```

Relación ManyToMany: `user ↔ role`

Un usuario puede tener **varios roles** y un mismo rol puede pertenecer a **varios usuarios**.
Esto se modela como una relación `@ManyToMany` con una tabla intermedia.

En la clase `User` :

- Declara un campo:

```
@ManyToMany(fetch = FetchType.EAGER)
@JoinTable(
    name = "user_roles",
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name = "role_id")
)
private Set<Role> roles;
```

- Da implementacion a getAuthorities(). Spring lo usará para comprobar los roles y dar acceso a recursos.

```
@Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        // Recupera los nombres de los roles 'ROLE_ADMIN', 'ROLE_USER'
        // .stream: recorrer colección
        // .map: transformar cada elemento
        // .collect(toList): convertir el resultado en lista
        return roles.stream()
                .map(r -> new SimpleGrantedAuthority(r.getName()))
                .collect(Collectors.toList());
    }
```

Crear los repositorios para User y Role

Una vez que tienes las entidades `User` y `Role` , el siguiente paso es crear los **repositorios** que gestionarán el acceso a la base de datos.

`UserRepository` y `RoleRepository`

- Crea las interfaces en un paquete `repository` .
- **extiende** de `JpaRepository` para heredar las operaciones básicas (findAll, save, delete…).

Ejemplo de estructura:

```
public interface UserRepository extends JpaRepository<User, Long> {
    // Buscar un usuario por su username (lo usará Spring Security)
    Optional<User> findByUsername(String username);
```

```
}
```

Clase `ConfiguraBDInicial` : limpiar la BD y crear usuarios/roles

Vas a crear una clase que se ejecute **automáticamente al arrancar la aplicación** para:

1. **Limpiar** las tablas relacionadas con usuarios y roles.
2. **Crear** unos roles iniciales.
3. **Crear** usuarios de prueba con esos roles.

Crea la clase `ConfiguraBDInicial` en un paquete `com.pspr.config.bd`

- Anótala con `@Configuration` y declara el `@Bean CommandLineRunner` en un metodo propio.

```
@Configuration
public class ConfiguraBDInicial {

    private static final Logger log = LoggerFactory.getLogger(ConfiguraBDInicial.class);

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private RoleRepository roleRepository;

    @Bean
    public CommandLineRunner initDatabase() {
        // Usamos una clase anónima en lugar de lambda
        return new CommandLineRunner() {
            @Override
            public void run(String... args) throws Exception {

                log.info("=== Inicializando base de datos de usuarios y roles ===");

                // 1. Borrar datos existentes
                log.info("Eliminando usuarios y roles existentes...");
                userRepository.deleteAll();
                roleRepository.deleteAll();
                log.info("Tablas limpiadas correctamente.");

                // 2. Crear roles
                log.info("Creando roles iniciales...");

                Role adminRole = new Role();
                adminRole.setName("ROLE_ADMIN");
                roleRepository.save(adminRole);
                log.info("Rol creado: {}", adminRole);

                Role userRole = new Role();
                userRole.setName("ROLE_USER");
                roleRepository.save(userRole);
                log.info("Rol creado: {}", userRole);

                User admin = new User();
                admin.setUsername("admin");

                admin.setPassword(passwordEncoder.encode("
                12345"));

                // roles del admin
                Set<Role> rolesAldmin = new HashSet<>();
                rolesAldmin.add(adminRole);
                rolesAldmin.add(userRole);
                admin.setRoles(rolesAldmin);

                 userRepository.save(admin);
```

```
                // 3. Crear usuario admin con los dos roles
                log.info("Creando usuario 'admin' con roles ADMIN y USER...");
                //TODO: crear y guardar usuario username "admin" con 2 roles

                // 4. Crear un usuario normal con ROLE_USER
                log.info("Creando usuario 'user' con rol USER...");
                // TODO: crear y guardar usuario username "user" con un rol

                log.info("=== Inicialización de BD completada correctamente ===");
            }
        };
    }
}
```

📝 Ejercicio 2: Arranca el servicio web, comprueba usuarios y roles

Arranca el servicio y comprueba:

- que ha creado las tablas en el esquema de Base de datos
- que ha insertado 2 roles y 2 usuarios

Añade al inicio la limpieza de datos de empresas y guarda 2 o 3 empresas de prueba

- con POSTMAN GET /empresas comprueba que devuelve las empresas de la tabla empresas

Cifrado de datos

📌 Criterios trabajados: **e**

Problema: Contraseñas visibles en la base de datos

- Para cifrar/descifrar las contraseñas de los usuarios usaremos BCryp para almacenar un hash de la contraseña

Bcrypt de Spring Security

Dependencia: Si ya usas `spring-boot-starter-security` , **ya tienes BCrypt**.

Define el Bean PasswordEncoder (BCrypt) para poder encriptar/desencriptar datos.

paquete config

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

```
<!--dependencia Spring Security -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

@Autowired
    private PasswordEncoder passwordEncoder;
añadir al constructor

// crear y guardar usuario username "admin" con 2 roles
 User admin = new User();
 admin.setUsername("admin");
            admin.setPassword (passwordEncoder.encode("12345"));
.

📝 Ejercicio 3: Cifra las password

Inyecta el Bean PasswordEncoder en la clase de carga inicial `ConfiguraBDInicial` .

- Cifra las password llamando a `passwordEncoder.encode("password")` .
- Comprueba en BD que se guardan cifradas.

Autenticación: Sesiones vs JWT

📌 Criterios trabajados: **a, b**

Sesiones (modelo clásico)

- El usuario hace **login** → el servidor crea una **sesión**.
- El servidor guarda datos de la sesión en memoria/BD.
- Al usuario se le asigna una **cookie** con el `sessionId` .
- En cada petición, el navegador envía la cookie y el servidor sabe quién es.

Ventajas:

- Fácil de usar en **páginas web clásicas** (HTML, formularios).

Inconvenientes:

- El servidor **guarda estado** (sesiones).
- Escala peor en arquitecturas con muchos servidores.
- Menos cómodo para **APIs REST** y **apps móviles**.

JWT (JSON Web Token, modelo tokens)

- El usuario hace **login** → el servidor genera un **JWT** (token firmado).
- El cliente guarda ese token (en memoria, localStorage, etc.).
- En cada petición protegida, el cliente envía el token en la cabecera:
  - `Authorization: Bearer <token>`
- El servidor **no guarda sesión**: solo valida el token en cada petición.

Ventajas:

- Ideal para **APIs REST**.
- Backend **stateless** (sin sesiones en servidor).
- Fácil de usar desde **apps móviles** y **frontends SPA** (Angular, React…).

Inconvenientes:

- Hay que gestionar bien la **caducidad** y la **seguridad** del token.

JWT en servicios REST

📌 Criterios trabajados: **c**, **f**

¿Qué es JWT?

Un **JWT (JSON Web Token)** es:

> Un **JSON token** es una cadena de texto que contiene datos del usuario en formato JSON y está **firmado criptográficamente** para que no se pueda modificar.

Se usa, sobre todo, para **autenticación en APIs REST**:

- El usuario hace *login* → el servidor genera un JWT.
- El cliente lo envía en cada petición en la cabecera:
  `Authorization: Bearer <token>` .
- El servidor valida la firma y la fecha de expiración del token.

Estructura básica de JWT

JWT tiene **3 partes**: `HEADER.PAYLOAD.SIGNATURE` .

- **HEADER**
  Indica el tipo de token y el algoritmo de firma.
  Ejemplo: `{ "alg": "HS256", "typ": "JWT" }`
- **PAYLOAD**
  Contiene los **datos (claims)** del usuario: quién es, roles, fecha de caducidad/expiración, etc.
  Ejemplo: `{ "sub": "andres", "role": "ADMIN", "exp": 1716220000 }`
- **SIGNATURE**
  Es la **firma** del token, calculada con:
  `header + payload + clave_secreta` (por ejemplo, HS256).
  La **clave secreta compartida** se almacena en el servidor.

HS256 es el algoritmo de firma más usado en JWT y significa:

- **HMAC**: mecanismo de firma que usa una **clave secreta compartida**.
- **SHA-256**: función hash (produce un resumen del JSON en 256 bits).

Validacion del token

Cuando el servidor recibe el token, vuelve a calcular la firma:

- Si coincide y no ha caducado → token **válido**.
- Si no coincide → token **modificado o falso** → deniega el acceso.

Pasos para usar JWT en el servicio

1. Añadir las dependencias JWT
2. Añadir propiedades clave y expiration en `application.properties` .
3. Crear clase JwtUtil
4. Crear DTOs de login (`LoginRequest` y `LoginResponse`).
5. Crear `AuthController` con `/auth/login` que devuelva el token JWT.
6. Proteger un endpoint de `Empresa` leyendo la cabecera `Authorization: Bearer` .

Dependencias JWT

Para implementar JWT en nuestro servicio, añadimos dependencias:

- jjwt-api: la API (interfaces, clases principales).
- jjwt-impl: implementación interna.
- jjwt-jackson: para convertir JSON ↔ objetos dentro del token.

```
<dependency>
     <groupId>io.jsonwebtoken</groupId>
     <artifactId>jjwt-api</artifactId>
     <version>0.11.5</version>
 </dependency>
 <dependency>
     <groupId>io.jsonwebtoken</groupId>
     <artifactId>jjwt-impl</artifactId>
     <version>0.11.5</version>
     <scope>runtime</scope>
 </dependency>
 <dependency>
     <groupId>io.jsonwebtoken</groupId>
     <artifactId>jjwt-jackson</artifactId>
     <version>0.11.5</version>
     <scope>runtime</scope>
 </dependency>
```

Propiedades JWT

En application.properties. Clave de servidor y tiempo de expiracion en milisegundos

```
### Propiedades JWT ###
jwt.secret=clavesupersecretamuylargaparahs2561234567890
jwt.expiration=900000
```

Clase JwtUtil

Necesitamos crear una clase que se encargará de crear y validar tokens JWT en nuestro servicio.

Dentro de un paquete com.pspr.**config.security.jwt**

```
@Component
public class JwtUtil {

    // Clave secreta para firmar el token
    @Value("${jwt.secret}")
    private String secretKey;

    // Tiempo de expiración del token en milisegundos (ej: 900000 = 15 min)
    @Value("${jwt.expiration}")
    private long expirationTime;

    public String generateToken(String username) {
        // Fecha de creación
        Date issuedAt = new Date();
        // Fecha de expiración = ahora + tiempo configurado
        Date expiration = new Date(System.currentTimeMillis() + expirationTime);

        return Jwts.builder()
                .setSubject(username)               // "sub": usuario
                .setIssuedAt(issuedAt)             // "iat": fecha creación
                .setExpiration(expiration)         // "exp": fecha caducidad
                .signWith(SignatureAlgorithm.HS256 // Algoritmo de firma HS256
                        , secretKey)               // Usa la clave secreta
                .compact();                        // Construye el token (String)
    }

    public String extractUsername(String token) {
        // Devuelve el "subject" (usuario) guardado en el token
        return getClaims(token).getSubject();
    }

    public boolean validateToken(String token, String username) {
        // El token es válido si:
        //   - el usuario dentro del token coincide
        //   - el token no ha caducado
        String usernameFromToken = extractUsername(token);
        boolean notExpired = !isTokenExpired(token);
        return username.equals(usernameFromToken) && notExpired;
    }

    private Claims getClaims(String token) {
        // Parsea el token y devuelve el payload (claims)
        return Jwts.parser()
                .setSigningKey(secretKey)   // misma clave usada para firmar
                .parseClaimsJws(token)      // valida firma y formato
```

```
                .getBody();                 // payload
    }

    private boolean isTokenExpired(String token) {
        // Compara la fecha de expiración con la fecha actual
        Date expiration = getClaims(token).getExpiration();
        return expiration.before(new Date());
    }
}
```

UserService

"Esta clase (UserService) será la encargada de decirle a Spring cómo buscar usuarios en la base de datos cuando alguien se quiera loguear."

Por eso debe implementar la interfaz *UserDetailsService* en concreto el metodo *loadUserByUsername(username)*.

Además podemos añadir nuestros propios métodos segun las necesidades.

```
@Service
public class UserService implements UserDetailsService {


    private final UserRepository userRepository;

    // Inicializar el Repositorio
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // Metodo requerido por SpringSecurity //
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        return // TODO buscar usuario
    }

    // Buscar por Nombre
    public Optional<User> buscarPorNombre(String username) {
        return // TODO buscar usuario

    }

    // Dado un nombre y una pass sin codificar valida si existe en BD //
    public boolean validaUsuarioPassword(String nombreUsuario, String rawPassword){
        // rawPassword: password sin cifrar
        //TODO validacion de usuario y pass
    }
}
```

return userRepository.findByUsername (username).get();

return userRepository.findByUsername (username);

//buscar el usuario por el nombre
Optional<User> opt = userRepository.findByUsername(nombreUsuario);
if (opt.isPresent()) {
//El usuario existe
User usuario = opt.get();
//Si existe compruebo la password con passwordEncoder.match
if(passwordEncoder.matches (rawPassword,usuario.getPassword())){
return true;}else{
//Si la password no coincide retorna falso
return false;}
}else{
//Si el usuario no existe retorna falso
return false;

📝 Ejercicio 4: Implementa la validacion de usuario y password

Implementa `validaUsuarioPassword(String nombreUsuario, String rawPassword)` :

- Si el usuario no existe retorna falso.
- Si la password no coincide retorna falso.
- La password de base de datos está cifrada. Por tanto, utiliza el metodo `matches(CharSequence rawPassword, String encodedPassword)` de PasswordEncoder para comprobar si las password coinciden.

AuthController (peticiones de token)

Añade la clase **AuthController** en el paquete **controllers** que va a gestionar las peticiones de comprobacion de la autenticacion y retornar el token JWT.

Añade unas clase DTOs para la peticion/respuesta del token

- Dentro de un paquete com.pspr.**config.security.dto** crea las clases (`LoginRequest` y `LoginResponse`) van a utilizarse para :
  - **Recibir** los datos de login que envía el cliente en el cuerpo de la petición (`LoginRequest` : `username` y `password`).
  - **Devolver** al cliente la información de autenticación tras un login correcto (`LoginResponse` : el `token` JWT generado).

DTO(Data Transfer Object) A menudo se identifican estas clases como dto, son clases que no pertenecen al modelo de negocio (model).

```
/**
 * En la peticion de login recibiremos username y  password
 * */
public class LoginRequest {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

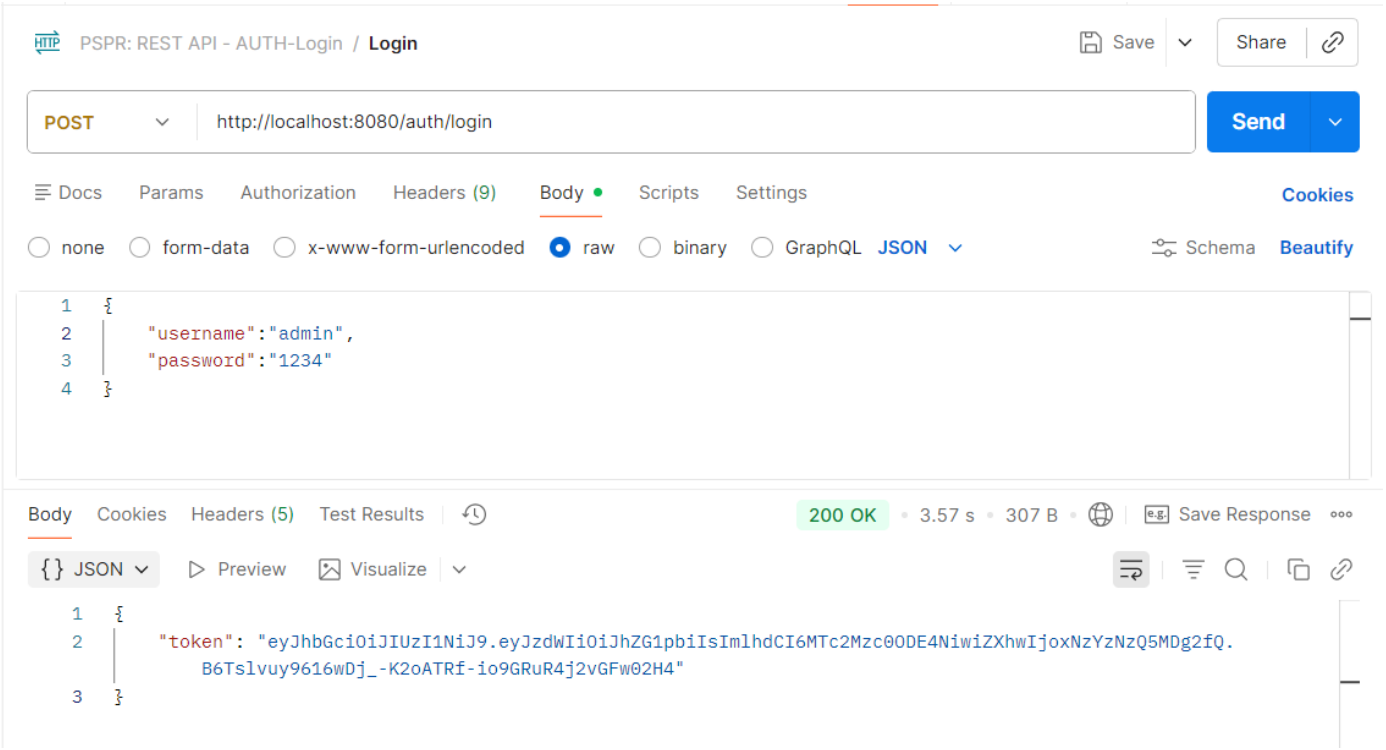
```
/**
 * En la respuesta devolveremos el token
 * */
```

```java
public class LoginResponse {
    private String token;

    public LoginResponse(String token) {
        this.token = token;
    }

    public String getToken() {
        return token;
    }
}
```

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

    @Autowired
    private UserService usuarioService;

    private final JwtUtil jwtUtil;

    public AuthController(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil;
    }

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody LoginRequest request) {

        // consulta a servicio para comprobar usuario y pass //
        boolean usuarioValido = usuarioService.validaUsuarioPassword(request.getUsername(), request.getPassword

        if (usuarioValido) {
            Optional<User> optUser = usuarioService.buscarPorNombre(request.getUsername());
            String token = // TODO generar token
            return ResponseEntity.ok(new LoginResponse(token));
        }
        // Error 401 (Unathorized)
        return ResponseEntity.status(401).body("Credenciales incorrectas");
    }
}
```

📝 Ejercicio 5: AuthController

Comprueba que genera el token realizando la prueba desde POSTMAN.



¿Como proteger un Endpoint?

Ahora que tenemos JWT implementando veremos como proteger un enpoint para validar el token en cada peticion.

Para proteger nuestros endoint (*/empresas, /alumnos...*) utilizaremos un filtro **Spring Security** que veremos a continuación.

2. Spring Security

📌 Criterios trabajados: **a**, **c**, **d**, **e**, **h**

¿Qué es Spring Security?

Spring Security es un **framework de seguridad para aplicaciones Spring Boot** que se encarga de:

- **Autorizar** el acceso a recursos, en base a la **Autenticación** de usuarios y roles.

Funciona configurando filtros que **interceptan las peticiones HTTP** y comprobando si el usuario está autenticado (tiene token jwt) y tiene permiso para acceder al endpoint.

Características principales:

- **Autenticación y autorización** por usuarios y roles.
- Integración con **JWT** para APIs REST.
- **Protección frente a ataques comunes** (accesos no autorizados, etc.).

- Posibilidad de integrarse con **OAuth2, LDAP u otros sistemas externos** de autenticación.

Dependencia Spring Security

```
<!--dependencia Spring Security -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

`SecurityFilterChain` en Spring Security

`SecurityFilterChain` es el **objeto principal de configuración de seguridad web** en Spring Security.

> Gestiona una **cadena ordenada de filtros de seguridad** que se aplican a las peticiones HTTP para decidir:
>
> - si el usuario está autenticado,
> - si tiene permisos para acceder a un recurso,
> - qué hacer cuando no cumple los requisitos (errores 401/403, etc.).

Crear un filtro que compruebe el token

Debemos implementar nuestro propio filtro que comprobará que el token viaja en la cabecera de las peticiones HTTP.

- El token se manda en una cabecera HTTP llamada Authorization.
- El valor va precedido por la palabra estandar 'Bearer' que indica el tipo de token y después de un espacio va el token real.
- El filtro debe extraer el token y revisar si es válido utilizando los metodos de JwtUtil extractUsername() y validateToken()
- Añade la autenticacion en el contexto de Spring

Dentro del paquete config.security.jwt

```
/**
 * Filtro que:
 *  - Lee el token del header Authorization.
 *  - Usa JwtUtil para extraer el username y validar el token.
 *  - Si es válido, marca al usuario como autenticado.
 */
@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    private final JwtUtil jwtUtil;

    private final UserDetailsService userDetailsService;
```

```java
// Inyectamos JwtUtil y UserService por constructor
public JwtAuthFilter(JwtUtil jwtUtil, UserDetailsService userDetailsService) {
    this.jwtUtil = jwtUtil;
    this.userDetailsService = userDetailsService;
}

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

    // 0) Si es el login, no exigimos token: dejamos pasar y salimos del filtro
    String path = request.getServletPath();
    if (path.equals("/auth/login")) {
        // continua la cadena de filtros
        filterChain.doFilter(request, response);
        return;
    }

    // 1) Leer la cabecera Authorization
    String authHeader = request.getHeader("Authorization");

    // Si no hay cabecera o no empieza por "Bearer ", seguimos sin autenticación
    if (authHeader == null || !authHeader.startsWith("Bearer ")) {
        // Respuesta de error //
        generarErrorResponse(response, "No autorizado: cabecera");
        return;
    }

    // 2) Sacar el token (sin "Bearer ")
    String token = authHeader.substring(7);

    // 3) Pedir a JwtUtil el username que hay dentro del token
    String username = //TODO

    // 4) Si tenemos username y aún no hay nadie autenticado en este hilo
    if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {

        // 5) Validar el token con JwtUtil
        if (jwtUtil.validateToken(token, username)) {

            // Recuperamos los datos del usuario //
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);

            // 6) Crear la autenticación para Spring Security
            UsernamePasswordAuthenticationToken authToken =
                    new UsernamePasswordAuthenticationToken(
                            userDetails,
                            null, // sin credentials (usamos token ya generado, la password no la necesitam
                            userDetails.getAuthorities() // Roles
                    );
            // añade información extra de la petición (como IP, sessionID, etc.) al objeto de autenticación
            authToken.setDetails(
                    new WebAuthenticationDetailsSource().buildDetails(request)
            );

            // 7) Guardar la autenticación en el contexto
            SecurityContextHolder.getContext().setAuthentication(authToken);
        }else {
            generarErrorResponse(response, "No autorizado: token no valido");
```

```
            }
        }

        // 8) Continuar la cadena de filtros
        filterChain.doFilter(request, response);
    }

    private static void generarErrorResponse(HttpServletResponse response, String mensajeError) throws IOExcept
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // 401
        response.setContentType("application/json;charset=UTF-8");
        response.getWriter().write("""
        {
          "error": """ + mensajeError +"""
        }
        """);
    }
}
```

Configuracion de la seguridad de la API

A continuación veremos cómo crear la clase `SecurityConfig` que protegerá nuestros endpoints. Esta clase debe:

1. **Registrar la cadena de filtros de seguridad**
   Definir un `@Bean SecurityFilterChain` para indicar a Spring cómo debe comportarse la seguridad.
2. **Desactivar CSRF**
   Deshabilitar CSRF porque trabajamos con una API REST que usa JWT (sin formularios ni cookies de sesión).
3. **Configurar la API como "sin estado" (STATELESS)**
   Indicar que el servidor no guarda sesión y que cada petición debe traer su propio token.
4. **Definir qué URLs necesitan autenticación**
   - `/auth/login` → `permitAll()` (no necesita token, sirve para obtenerlo).
   - `/empresas/**` → `hasRole(...)` (requiere un rol especifico).
   - `anyRequest()` → `authenticated()` (resto de peticiones, requiere autenticacion token válido).
5. **Registrar el filtro `JwtAuthFilter` en la cadena**
   Añadir nuestro filtro antes del filtro de login estándar para que lea y valide el token en cada petición.

Dentro del paquete `config.security`

- CustomAccessDeniedHandler: Manejador de errores de acceso.
- SecurityConfig: carga el bean que gestiona la cadena de filtros.

```
@Component
public class CustomAccessDeniedHandler implements AccessDeniedHandler {

    @Override
    public void handle(HttpServletRequest request,
                       HttpServletResponse response,
                       AccessDeniedException accessDeniedException)
            throws IOException, ServletException {

        response.setStatus(HttpServletResponse.SC_FORBIDDEN); // 403 (Forbidden), se quien eres, pero no tienes
        response.setContentType("application/json;charset=UTF-8");
```

```
        String body = """
        {
          "error": "Acceso denegado: no tienes permisos suficientes",
          "path": "%s"
        }
        """.formatted(request.getRequestURI());

        response.getWriter().write(body);
    }
}
```

```
/**
 * Clase de configuración de seguridad para la API
 * - Protege endpoints Ej: empresas/**
 * - Usa JWT en la cabecera Authorization
 **/
public class SecurityConfig {

    private final JwtAuthFilter jwtAuthFilter;
    private final CustomAccessDeniedHandler accessDeniedHandler; // Para manejar el error de rol requerido //
    public SecurityConfig(JwtAuthFilter jwtAuthFilter, CustomAccessDeniedHandler accessDeniedHandler) {

        this.jwtAuthFilter = jwtAuthFilter;
        this.accessDeniedHandler = accessDeniedHandler;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http
                // 1) Desactivar CSRF con la sintaxis nueva de funciones flecha (no deprecated)
                .csrf(csrf -> {
                    csrf.disable();
                })

                // 2) API sin sesiones (stateless): cada petición trae su token
                .sessionManagement(sess -> {
                    sess.sessionCreationPolicy(SessionCreationPolicy.STATELESS);
                })

                // 3) Reglas de autorización sobre peticiones a los endpoint
                .authorizeHttpRequests(auth -> {
                    auth.requestMatchers("/auth/login").permitAll() // el endpoint login es público
                        .requestMatchers("/empresas/**").hasRole("ADMIN") // endpoints autenticacion (token) co
                        .anyRequest().authenticated(); // el resto de peticiones requiere autenticacion (token)
                })

                // Manejo de errores customizado:
                // Si el usuario tiene token valido pero no tiene el rol requerido devuelve 403 Forbidden//
                .exceptionHandling(ex -> ex
                        .accessDeniedHandler(accessDeniedHandler)  // <<--- AQUÍ
                )

                // 4) Añadir nuestro filtro que valida el token a la cadena de filtros
                // antes del filtro estandar UsernamePasswordAuthenticationFilter.class
                .addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);
```

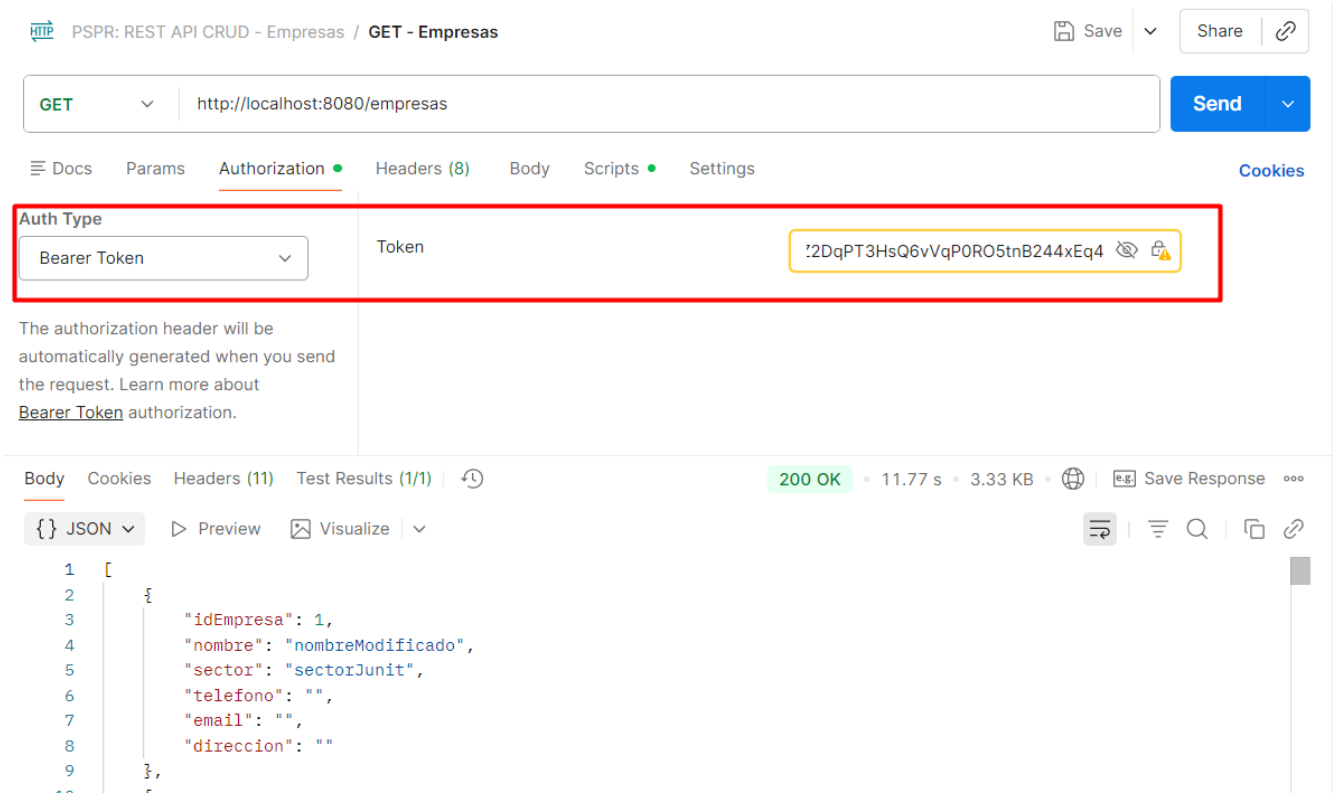
```
            return http.build();
        }

    }
```

📝 Ejercicio 6: Proteger el endpoint de /empresas

Añade las clases para proteger endpoint de `Empresa` leyendo la cabecera `Authorization: Bearer .`

1. Desde POSTMAN:

- Realiza el login para obtener el token
- Realiza la peticion a empresas indicando Bearer Token



2. Mejora la gestión del error:

- Crea un metodo en JwtAuthFilter para generar una respuesta diferenciada de error en JSON para los casos de cabecera sin token y token no valido
  - generarErrorResponse(response, "No autorizado: cabecera sin token")
  - generarErrorResponse(response, "No autorizado: token no valido");

3. Crea el servicio UsuarioService > UsuarioDAO > Tabla Usuario (MySQL), para consultar y validar datos de usuarios almacenados en BD.

Activacion de HTTPS

📌 Criterios trabajados: **f, g**

¿Qué es HTTPS?

- HTTPS = **HTTP + TLS (Transport Layer Security, trabaja sobre sockets TCP)** (capa de cifrado).
- El cliente se conecta a `https://servidor` .
- El servidor envía su **certificado** (con su clave pública).
- Cliente y servidor negocian una **clave simétrica** para cifrar los datos.
- Todo lo que viaja después (usuario, password, JSON…) va **cifrado**.

¿Qué es un certificado autofirmado?

- Certificado **firmado por el propio servidor**, **no** por una Autoridad de Certificación (CA).
- **Sí cifra** la conexión, pero:
  - Navegador / Postman **no lo consideran de confianza** → aviso de seguridad.
- Útil para **desarrollo y pruebas**.
- En producción se usan certificados emitidos por una **CA reconocida** (p. ej. Let's Encrypt).

HTTPS en Spring Boot:

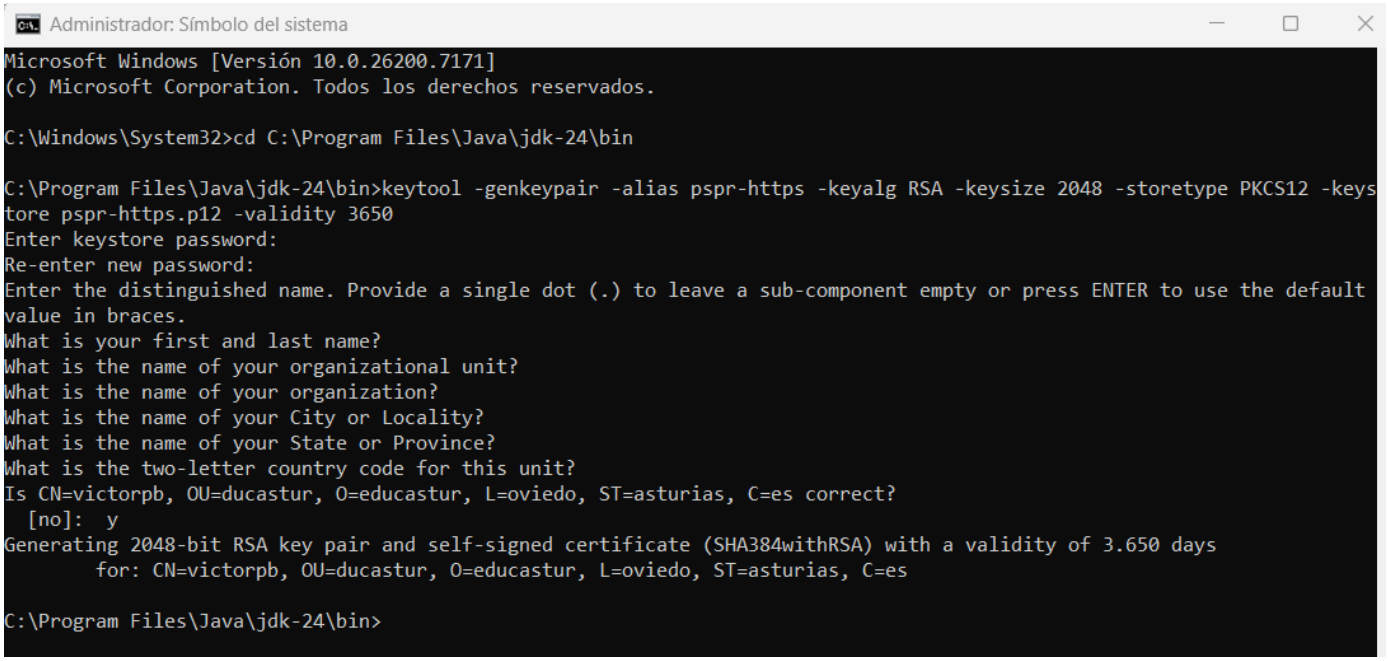
Necesitamos:

- Un certificado + clave privada → en un keystore (.p12 o .jks)
- Configurar Spring Boot con ese keystore.
- Para desarrollo basta con un certificado autofirmado.

Genera certificado autofirmado de servidor

Usando la herramienta **keytool** de la JDK/bin. Ejecuta en un una linea desde cmd con permisos de administrador

```
keytool -genkeypair -alias pspr-https -keyalg RSA -keysize 2048
 -storetype PKCS12 -keystore pspr-https.p12 -validity 3650
```



Copia pspr-https.p12 a src/main/resources/ de tu proyecto.

Properties

En src/main/resources/application.properties

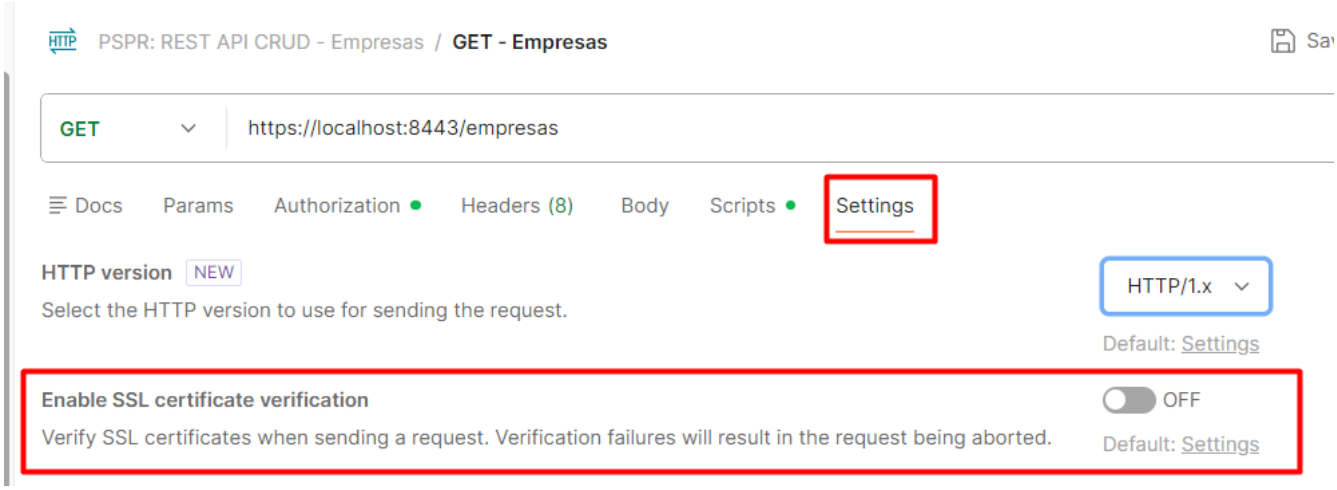
```
## CONFIGURACION HTTPS ##
# Puerto HTTPS
server.port=8443
# Activar SSL
server.ssl.enabled=true
# Ruta al keystore (está en resources)
server.ssl.key-store=classpath:pspr-https.p12
# Password del keystore (la que pusiste en keytool)
server.ssl.key-store-password=123456
# Tipo de keystore
server.ssl.key-store-type=PKCS12
# Alias de la clave (el que pusiste en keytool: pspr-https)
server.ssl.key-alias=pspr-https
```

Con esta configuracion el servidor Tomcat arranca en el puerto seguro htttps

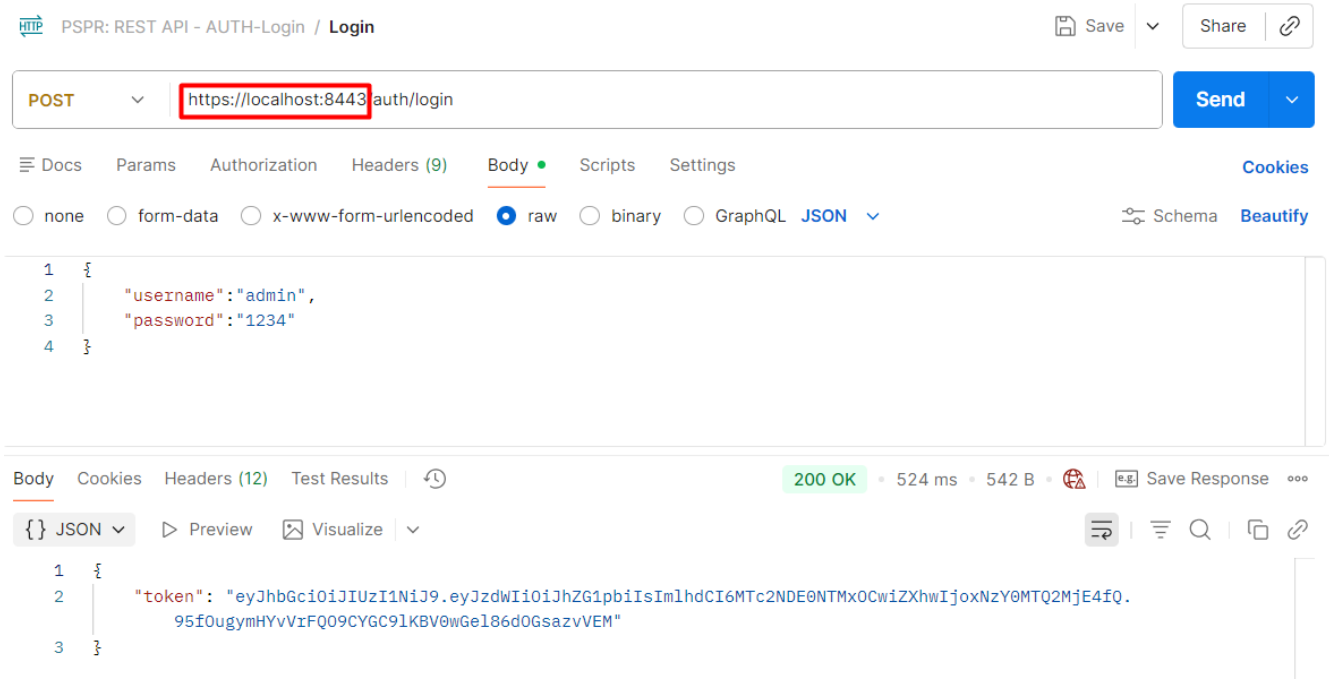
📝 Ejercicio 7: Prueba de cifrado HTTPS

Comprueba desde POSTMAN el funcionamiento de HTTPS.

- Deshabilita la conmprobacion de verificacion del certificado (ya que nos fallará al ser autofirmado).



- Haz una consulta por https, por ejemplo a /auth/login
- Con esta configuración los datos de usuario y password viajarán cifrados, lo que nos protege ante ataques de intermediario.



- Comprueba el certificado de servidor consultando un recurso desde navegador: