DAW
Desarrollo de Aplicaciones Web
2º Curso

DWES
Desarrollo Web Entorno Servidor

UD 5 Spring Boot

5. Spring SECURITY

IES BALMIS
Dpto Informática
Curso 2025-2026
Versión 2 (01/2026)

UD5.5 – Spring SECURITY

ÍNDICE

5.1. Spring SECURITY
5.2. SECURITY con Thymeleaf
5.3. Backend usando SECURITY con Thymeleaf (NO EXAMEN)
5.4. SECURITY con APIREST

1. Spring SECURITY

Spring Security es un framework de seguridad para aplicaciones Java, especialmente diseñado para proyectos Spring. que se encarga de la autenticación (identificar quién eres) y la autorización (qué puedes hacer) en tu aplicación web o API.

El uso de **Spring Security** nos permite:

- Controlar quién accede a los recursos de tu aplicación
- Gestionar logins, contraseñas y sesiones
- Proteger recursos según roles de usuario (ADMIN, USER, etc.)
- Prevenir ataques comunes (CSRF, inyecciones, etc.)
- Asegurar endpoints (URLs de acceso a métodos) en aplicaciones APIREST

En un proyecto **Spring Security** podremos incluir:

- Redirecciones automáticas
- Login con formulario HTML
- Integración en Thymeleaf mediante etiquetas de seguridad
- Sesiones almacenadas en servidor
- Protección contra CSRF activada por defecto

Para gestionar la seguridad se puede usar una gestión **Stateful** (con almacenamiento en el servidor usando SESSION) o **Stateless** (con almacenamiento en cliente usando token JWT)

	Stateful (Con Estado)	Stateless (Sin Estado)
Tecnología	Uso de **SESSION**	Uso de tokens **JWT**
Cómo funciona	El servidor guarda información de sesión	El servidor NO guarda información
Almacenamiento	En el servidor (memoria o base de datos)	En el cliente (navegador o app móvil)
Cada petición	Cliente envía un ID de sesión	Cliente envía el token completo

Como ejemplo podríamos utilizar el siguiente símil:

- **Stateful (SESSION)**: "Como una discoteca con pulseras: l entrar, te dan un número (SESSION_ID). El portero consulta su lista para ver si puedes pasar."
- **Stateless (JWT)**: "Como un pase de autobús: Tiene toda la información impresa (destino, validez). El conductor solo mira el pase, no consulta nada."

Usaremos SESSION en nuestros proyectos

En principio, es más simple usar SESSION que JWT, por lo que en nuestros ejemplos usaremos **SESSION**.

A nivel profesional usaremos **JWT** si:

- Necesitamos estar preparados para una escalabildad
- Tenemos múltiples servidores
- Si los clientes usan Apps móviles para acceder
- Si se ofrece una servicio de APIREST con muchos usuarios

Activar Spring Security

Para activar **Spring Security** debemos preparar:

- los recursos públicos accesibles sin identificación
- los recursos privados accesibles con identificación
- los usuarios que van a atener acceso con al menos los siguientes datos:
  - **username**: identificador de usuario
  - **password**: contraseña cifrada
  - **role**: rol o roles de usuario que nos permitirán agrupar permisos de acceso

Para activar Spring Security deberemos incluir en el pom.xml las siguientes dependencias:

pom.xml con Spring Security

```
<!-- SECURITY -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.thymeleaf.extras</groupId>
    <artifactId>thymeleaf-extras-springsecurity6</artifactId>
</dependency>
```

Cifrado de las contraseñas

La normativa española prohíbe expresamente que se almacenen las contraseñas en texto plano bajo pena de multa de entre 40.000 y 300.000 euros, de forma que no es solo una "sugerencia de seguridad".

En el **Reglamento de desarrollo de la Ley Orgánica 15/1999**, de 13 de diciembre, de **protección de datos de carácter personal**.
En Real Decreto 1720/2007, de 21 de diciembre
https://www.boe.es/buscar/pdf/2008/BOE-A-2008-979-consolidado.pdf

Artículo 93. Identificación y autenticación.

1. El responsable del fichero o tratamiento deberá adoptar las medidas que garanticen la correcta identificación y autenticación de los usuarios.
2. El responsable del fichero o tratamiento establecerá un mecanismo que permita la identificación de forma inequívoca y personalizada de todo aquel usuario que intente acceder al sistema de información y la verificación de que está autorizado.
3. Cuando el mecanismo de autenticación se base en la existencia de contraseñas existirá un procedimiento de asignación, distribución y almacenamiento que garantice su confidencialidad e integridad.
4. El documento de seguridad establecerá la periodicidad, que en ningún caso será superior a un año, con la que tienen que ser cambiadas las contraseñas que, mientras estén vigentes, **se almacenarán de forma ininteligible**.

Atendiendo a la normativa, deberemos almacenar las contraseñas aplicando algún método de **hash** o **cifrado**.

Antes de continuar aclararemos varios conceptos.

La **criptografía** es la técnica que se encarga del cifrado y el codificado de los datos con la finalidad de hacerlos inteligibles a personas y terceros para los que no está destinado el mensaje.

Por lo tanto, la **criptografía** es un método de protección de la información y las comunicaciones mediante el uso de códigos que permite que solo aquellos a quienes está destinada la información puedan leerla y procesarla.

La **encriptación** (encrypt) o **cifrado** (cipher) es un procedimiento que utiliza un algoritmo para transformar un mensaje utilizando una clave, sin atender a su estructura lingüística o significado, de tal forma que sea incomprensible a toda persona que no tenga la clave secreta del algoritmo para descifrarlo.

El **hashing** es el proceso de generar una salida (output) de extensión fija de bits a partir de una entrada (input) de extensión variable, mediante el uso de unas fórmulas matemáticas denominadas **funciones hash** que se implementan/programan como **algoritmos de hashing**.

La principal distinción entre **cifrado** y **hashing** es que el **cifrado es reversible** (se puede descifrar) mientras que el **hashing es irreversible,** por lo que para el **almacenamiento de contraseñas usaremos hash**.

El **hash tradicional** utiliza algoritmos como **SHA-256** o **MD5**, que son genéricas y rápidas de calcular, lo que puede ser un riesgo en ataques de fuerza bruta.

Para mejorar la seguridad del hash existen algoritmos más seguros como **Bcrypt**, **Argon2** o **PBKDF2**. **En Spring Boot usaremos Bcrypt.**

BCrypt es un algoritmo de hashing diseñado específicamente para almacenar contraseñas de manera segura. Se basa en la función de cifrado Blowfish y está diseñado para ser resistente a ataques de fuerza bruta y precomputados.

Bcrypt añade en su algoritmo de hash un valor aleatorio (**Salt**) a la cadena antes de aplicarlo, y realiza este proceso un número de veces, rondas o ciclos (**Work Factor**) para crear un hash seguro. Este enfoque fortalece la seguridad de las contraseñas almacenadas, protegiendo efectivamente contra ataques cibernéticos.

La verificación de una contraseña implica comparar la contraseña proporcionada por el usuario con el hash almacenado en la base de datos.

Este proceso de verificación normalmente se realiza de la siguiente manera:

- **Obtener los Datos**: contraseña recibida y hash almacenado
- **Recrear el Hash**: aplicar el algoritmo a la contraseña recibida
- **Comparar los Hashes**: el almacenado con el recreado

GenerarBcryptMv

```
Proyecto realizado con Maven que permite crear y verificar un HASH para almacenamiento
de contraseñas usando Bcrypt
```

Nuevo contenido en los proyectos

El componente que gestiona la seguridad, generalmente denominamos a esta clase **SecurityConfig,** utiliza las anotaciones:

- **@Configuration**
- **@EnableWebSecurity**

Los métodos de la clase **SecurityConfig:**

- **securityFilterChain**: Configura las reglas de autorización de la aplicación
- **userDetailsService**: Crea usuarios en memoria para autenticación (no usa base de datos)
- **passwordEncoder**: Proporciona un codificador de contraseñas Bcrypt

Si deseamos tener almacenados los usuarios en una base de datos deberemos eliminar el método **userDetailsService** e incorporar una clase **CustomUserDetailsService** que permita la identificación del usuario mediante la búsqueda en la BD.

2. SECURITY con Thymeleaf

Aunque no es nuestro objetivo principal en este módulo usar **Thymeleaf**, con todo lo aprendido en **SPRING MVC** y **SPRING DATA** podemos crear proyectos Web con **Bootstrap** y **Jquery** con toda la funcionalidad de consultas y actualizaciones.

Para poder ver de forma sencilla lo que ofrece el **framework** de **Spring Security**, se ha realizado un proyecto base y se han ido incorporando características paso a paso.

2.1. Proyecto minimal-security

Para empezar crearemos un proyecto muy simple con Spring Boot que contendrá dos páginas web con Thymeleaf y dos controladores.

Minimal-security-v0 (SIN SECURITY)

Creamos un proyecto Spring Security muy simple:

-Editar variables en el **application.properties** para el nuevo proyecto

-Incluir la dependencia de **Thymeleaf y** los archivos ***.html de templates**

-Añadir **HomeController** para redirigir a **home.html**
http://localhost:8080/minimalsec0

-Añadir **SecureController** para redirigir a **secure.html**
http://localhost:8080/minimalsec0/secure

Al no disponer de seguridad, las dos páginas son **accesibles de forma pública**, es decir, sin identificación. Además, se puede comprobar que no disponemos de página para el **login** ni para el **logout**.

Minimal-security-v1 (CON SECURITY)

-Editar variables en el **application.properties** para el nuevo proyecto

-Añadir dependencias de Spring Security en **pom.xml**

-Añadir clase **SecurityConfig** con **securityFilterChain y userDetailsService**

La página **secure.html** ya no es accesible sin identificación.

http://localhost:8080/minimalsec1

Al añadir las dependencias de Spring Security debemos incorporar la clase **SecurityConfig** para gestionar los accesos.

Spring Security crea dos formularios por defecto asignados a las URL:

- **/login**
- **/logout**

NOTA: Revisar los comentarios en el código de la clase SecurityConfig

Minimal-security-v2

-Editar variables en el **application.properties** para el nuevo proyecto

-Mejorar los archivos ***.html** de templates aprovechando **Thymeleaf**

http://localhost:8080/minimalsec2

En este proyecto se han mejorado las plantillas **html** aprovechando las funcionalidades de **Thymeleaf**:

- incluyendo un **menú** navegable
- botones para **iniciar y cerrar sesión**
- mostrando las **páginas disponibles** del proyecto
- indicando el **estado de la identificación**
- ...

Se puede comprobar también que si realizamos un **POST** a **/logout** en vez de un **GET** desconecta sin preguntar.

Minimal-security-v3

-Editar variables en el **application.properties** para el nuevo proyecto

-Añadir plantilla **login.html** personalizado para no usar el de por defecto de **Thymeleaf**

-Añadir **"/login"** a **SecurityConfig** tanto en **authorizeHttpRequests** en **formLogin**

-Añadir a **HomeController** el método **GET** a **"/login"**

http://localhost:8080/minimalsec3

Como hemos añadido nuestra plantilla **login.html** hemos modificado la clase **SecurityConfig**, añadiéndola como formulario de login (**con .loginPage**) y permitiendo su acceso público (**con .requestMatchers**).

Minimal-security-v3 - SecurityConfig

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
  http
    .authorizeHttpRequests(auth -> auth
    .requestMatchers("/").permitAll()       // Acceso público a "/"
    .requestMatchers("/login").permitAll()  // Acceso público a "/login"
    .anyRequest().authenticated()           // Acceso identificado a "/secure"
  )
  .formLogin(form -> form
    .loginPage("/login")                    // Usar la plantilla /login.html
    .defaultSuccessUrl("/secure", true)     // Si se indentifica
  )
  .logout(logout -> logout     // Configuración del logout
    .logoutUrl("/logout")      // URL que dispara el logout (POST por defecto "/logout")
    .logoutSuccessUrl("/")     // Redirige a la página principal después del logout
    .invalidateHttpSession(true)  // Invalida la sesión HTTP
    .clearAuthentication(true)    // Limpia la autenticación
    .deleteCookies("JSESSIONID")  // Elimina la cookie de sesión (opcional-recomendable)
  );
  return http.build();
}
```

Minimal-security-v4

-Editar variables en el **application.properties** para el nuevo proyecto

-Eliminar **userDetailsService** de **SecurityConfig** y
crear clase **CustomUserDetailsService** para personalizar la **gestión de usuarios**

http://localhost:8080/minimalsec4

Después de eliminar **userDetailsService** en **SecurityConfig**, hemos añadido nuestro propio **userDetailsService** denominado **customUserDetailsService.**

La clase tiene **customUserDetailsService** los mismos usuarios que antes, pero como diferencia, comprueba que el usuario devuelto no sea null, pues en este caso devolvería una excepción que el **login.html** recibiría como **param.error,** igual que antes con el **userDetailsService** por defecto.

Minimal-security-v5

En este proyecto deseamos configurar la gestión de usuarios para Security a través de una Base de Datos en H2:

-Editar variables en el **application.properties** para el nuevo proyecto

-Añadir dependencias para H2
-Configurar **application.properties** para JPA y H2

-Incluir **schema.sql** y **data.sql**
-Añadir **"/h2-console"** a **SecurityConfig** en **authorizeHttpRequests**
-Añadir **csrf y headers** a **SecurityConfig** para el buen funcionamiento de **h2-console**

-Añadir clases de datos **Usuario**
-Añadir **UsuarioRepository**
-Modificar **CustomUserDeta**
ilsService para que la **gestión de usuarios acceda a la BD** a la hora de validar el usuario

http://localhost:8080/minimalsec5

http://localhost:8080/minimalsec5/h2-console

Este es nuestro último proyecto de ejemplo en el que incorporamos una base de datos H2 con los usuarios disponibles.

Para gestionarlo le hemos incluido al anterior proyecto:

- El modelo **Usuario** y el repositorio **UsuarioRepository** que nos permitirá realizar los accesos básicos a la tabla **users_security** como vimos en **Spring DATA**.
- Para devolver el **UserDetails** correcto, hemos modificado en **CustomUserDetailsService** el método **loadUserByUsername** que ahora busca el usuario en la BD con **userRepository.findByUsername(username).**
- **IMPORTANTE**: No es necesario comprobar en **CustomUserDetailsService** si la contraseña es correcta al buscar en la BD ya que **Spring Boot** lo realiza posteriormente comparando los datos introducidos en el formulario con el objeto **UserDetails** que le hemos devuelto.

IMPORTANTE: Si usamos **Spring Security NO** debemos crear clases de datos con el nombre **User** ni **Role** porque ya existen y podrían crear confusión al usarlas.

Para poder acceder a la consola de H2 se ha añadido al **SecurityConfig** una nueva configuración de **csrf, sessionManagement** y **headers**.

Minimal-security-v5 - SecurityConfig

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
  http
    .authorizeHttpRequests(auth -> auth
    .requestMatchers("/").permitAll()                // Acceso público a "/"
    .requestMatchers("/login").permitAll()           // Acceso público a "/login"
    .requestMatchers("/h2-console/**").permitAll()   // Acceso público a "/h2.console"
    .anyRequest().authenticated()                    // Acceso identificado a "/secure"
  )
  // Inicio de Configuraciones adicionales para H2
  .csrf(csrf -> csrf
    .ignoringRequestMatchers("/h2-console/**")       // Desactiva CSRF para H2
  )
  .sessionManagement(session -> session
    .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
                                 // Crea sesión solo si es necesario (por defecto)
  )
  .headers(headers -> headers
    .frameOptions(frame -> frame
      .sameOrigin()              // Permite iframes del mismo origen (necesario para H2)
    )
  )
  // Fin de Configuraciones adicionales para H2
  .formLogin(form -> form
    .loginPage("/login")                    // Usar la plantilla /login.html
    .defaultSuccessUrl("/secure", true)     // Si se indentifica
  )
  .logout(logout -> logout     // Configuración del logout
    .logoutUrl("/logout")      // URL que dispara el logout (POST por defecto "/logout")
    .logoutSuccessUrl("/")     // Redirige a la página principal después del logout
    .invalidateHttpSession(true)  // Invalida la sesión HTTP
    .clearAuthentication(true)    // Limpia la autenticación
    .deleteCookies("JSESSIONID")  // Elimina la cookie de sesión (opcional-recomendable)
  );
  return http.build();
}
```

CSRF (Cross-Site Request Forgery) es una medida de seguridad que protege contra ataques donde un atacante engaña a un usuario autenticado para que realice acciones no deseadas en una aplicación web.

Por ejemplo, imagina que estás logueado en tu banco online (banco.com) y en otra pestaña visitas un sitio malicioso. Si ese sitio tiene un formulario oculto que envía una transferencia bancaria a banco.com como ya estás autenticado en el banco, la transferencia se ejecuta sin tu consentimiento.

Por defecto, el control de **CSRF en Spring Boot está activado**.
En nuestro caso, lo desactivamos/ifnoramos para **h2-console** ya que se abren ventanas que también deben acceder.

sessionManagement es un componente de Spring Security que gestiona el ciclo de vida, seguridad y comportamiento de las sesiones HTTP en una aplicación web, controlando cómo se crean, mantienen, protegen y destruyen las sesiones de usuario durante la interacción con el sistema.

Los valores posibles para gestionar la creación de sesiones son:

- **SessionCreationPolicy.IF_REQUIRED:** Crea sesión solo si es necesario (es el valor por defecto)
- **SessionCreationPolicy.ALWAYS**: Siempre crea sesión
- **SessionCreationPolicy.NEVER**: Nunca crea, pero usa si existe
- **SessionCreationPolicy.STATELESS**: Nunca crea ni usa sesiones (suele usarse con el uso de JWT)

Cuando usando Java (o Spring Security) se crea una sesión entre el navegador cliente y el servidor, se asocian mediante el valor de una **cookie** denominada **JSESSIONID**.

Al no usar JWT, toda la información necesaria entre cliente y servidor es almacenada en el servidor asociada a la cookie y la utiliza cuando este recibe una petición con la cookie **JSESSIONID**.

Con el **sessionManagement**, además del **sessionCreationPolicy**, también podemos controlar:

- **MaximumSessions**: número de sesiones concurrentes del mismo usuario
- **InvalidSessionStrategy**: comportamiento cuando se detecta una sesión inválida o corrupta (por ejemplo, redirigir a una página específica)
- **SessionFixation**: protección contra ataques de fijación de sesión donde un atacante fuerza a una víctima a usar un ID de sesión conocido
- **SessionAuthenticationStrategy**: estrategias personalizadas para manejar eventos de autenticación como logins concurrentes o expiración de sesión (para casos avanzados)

Por ejemplo:

```
.sessionManagement(session -> session
  // Política de creación de sesión: solo crea sesión cuando es necesario (por defecto)
  .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
  // Máximo de sesiones concurrentes: solo 1 sesión activa por usuario
  .maximumSessions(1)
  // Comportamiento al alcanzar el máximo: bloquea nuevos logins (más seguro)
  .maxSessionsPreventsLogin(true)
  // Redirección cuando la sesión expira por tiempo o concurrencia
  .expiredUrl("/login?expired")
   // Redirección cuando la sesión es inválida o corrupta
  .invalidSessionUrl("/login?invalid")
  // Protección contra Session Fixation Attack: crea nueva sesión vacía al autenticar
  .sessionFixation().newSession()
)
```

A continuación mostramos las características incluidas en cada proyecto:

	PROYECTOS				
	min-v1	**min-v2**	**min-v3**	**min-v4**	**min-v5**
Thymeleaf	✅	✅	✅	✅	✅
Lombok	✅	✅	✅	✅	✅
Controlador	✅	✅	✅	✅	✅
Service con Repository	✅	✅	✅	✅	✅
GlobalException Handler	✅	✅	✅	✅	✅
Acceso a Datos					✅ **JpaRepository**
Base de Datos					✅ **H2 Database**
Persistencia					✅ **MEM**
Carga de Datos inicial					✅ **shema.sql + data.sql**
pom.xml SecurityConfig	✅	✅	✅	✅	✅
HTML	❌ HTML	✅ Thymeleaf (Bootstrap)	✅ Thymeleaf (Bootstrap)	✅ Thymeleaf (Bootstrap)	✅
login.html HomeController	❌ Automático	❌ Automático	✅ Personalizado	✅ Personalizado	✅ Personalizado
CustomUser DetailsService				❌ Usuarios Fijos	✅ Usuarios BD

3. Backend usando SECURITY con Thymeleaf

SOLO SE MUESTRA A MODO DE EJEMPLO POR SI ALGÚN ALUMNO QUIERE IMPLEMENTARLO EN EL PROYECTO

NO ENTRA EN EL EXAMEN
NO ES EVALUABLE

3.1. Proyecto mvc-user

El siguiente proyecto muestra un **CRUD** para crear, leer, actualizar y eliminar **usuarios** con la estructura de la tabla **users_security** usada en los ejemplos anteriores de Spring Security**.**

mvc-user

-Editamos variables en el **application.properties** para el nuevo proyecto

-Incluimos la dependencia de **Thymeleaf y** los archivos **.html de templates**

http://localhost:8080/mvcuser

http://localhost:8080/mvcuser/h2-console

3.2. Proyecto mvc-prod

Para poder disponer de un **CRUD** de datos hemos creado otro proyecto que muestra un CRUD para crear, leer, actualizar y eliminar **productos** con la estructura de la tabla **productos** básica con **id**, **descrip** y **precio.**

mvc-prod

-Editamos variables en el **application.properties** para el nuevo proyecto

-Incluimos la dependencia de **Thymeleaf y** los archivos **.html de templates**

http://localhost:8080/mvcprod

http://localhost:8080/mvcprod/h2-console

3.3. Proyecto mvc-merge

En esta versión hemos unido los dos proyectos, creando un menú navegable para acceder a los CRUD.

mvc-merge

-Editamos variables en el **application.properties** para el nuevo proyecto

http://localhost:8080/mvcmerge

http://localhost:8080/mvcmerge/h2-console

3.4. Proyecto prod-security

Hemos creado este proyecto a partir del anterior añadiendo **Spring Security** para:

- Permitir el acceso público sin identificación a:
  - **/**
  - **/acerca**
  - **/catalogo** => Nueva plantilla de Thymeleaf par consultar productos
- Permitir el acceso privado con identificación y rol de "USER"
  - **/productos** => CRUD de Productos
- Permitir el acceso privado con identificación y rol de "ADMIN"
  - **/productos** => CRUD de Productos
  - **/usuarios** => CRUD de Usuarios
  - **/h2-console** => Consola de H2

mvc-merge-sec

-Editamos variables en el **application.properties** para el nuevo proyecto

-Las opciones de menú se cargarán solo si tenemos acceso

http://localhost:8080/mvcmergesec

http://localhost:8080/mvcmergesec/h2-console

4. SECURITY con APIREST

Este es el objetivo de nuestro módulo y de la UD5, crear un APIREST que permita realizar CRUD a las tablas de una BD.

A continuación crearemos proyectos para realizar un **APIREST** como se ha visto en el apartado de **Spring APIREST** pero añadiendo **Spring SECURITY**.

4.1. Proyecto apirest-user y apirest-prod

Estos proyectos proyectos de APIREST gestionan las tablas de usuarios y productos respectivamente e incluyen:

- Todos los métodos básicos
- Persistencia con **H2 Database**
- Documentación con **Swagger**

4.2. Proyecto apirest-merge

Hemos creado otro proyecto que incluya los dos APIREST juntos con las mismas características.

4.3. Proyecto apirest-merge-sec

Partiendo del proyecto **apirest-merge**, se ha creado este incluyendo **Spring Security**.

Para que tenga toda la funcionalidad, y dado que no tenemos la parte visual de Thymeleaf, se ha tenido que crear otro controlador para gestionar la identificación de usuarios.

Al añadir Security en un proyecto Spring APIREST creamos/modificamos lo siguiente:

- **CustomUserDetailsService**
  - usamos el mismo que con Thymeleaf para validar con una BD
- **SecurityConfig**
- **AuthController**
- **GlobalExceptionHandler**

apirest-merge-sec - SecurityConfig

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
  http
    .authorizeHttpRequests(auth -> auth
      .requestMatchers("/").permitAll()                 // Acceso público a "/"
      .requestMatchers("/swagger-ui/**",                // Acceso público a SWAGGER
                       "/swagger-ui.html",
                       "/swagger-ui/index.html",
                       "/api-docs/**",
                       "/webjars/swagger-ui/**").permitAll()
      .requestMatchers("/api/auth/**").permitAll()     // Acceso público a Identificación
      .requestMatchers("/productos/**").hasAnyRole("ADMIN","USER")
                                                        // Acceso identificado productos
      .requestMatchers("/usuarios/**").hasRole("ADMIN")  // Acceso identificado usuarios
      .requestMatchers("/h2-console/**").hasRole("ADMIN")
                                                        // Acceso identificado H2 console
      .anyRequest().denyAll()                           // Acceso DENEGADO al resto
    )
    // Inicio de Configuraciones adicionales para H2
    .csrf(csrf -> csrf
      .ignoringRequestMatchers("/h2-console/**")      // Desactiva CSRF para H2
      .disable()                                        // Desactiva para APIREST
    )
    .sessionManagement(session -> session
      .sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
                              // Crea sesión solo si es necesario (por defecto)
    )
    .headers(headers -> headers
      .frameOptions(frame -> frame
        .sameOrigin()            // Permite iframes del mismo origen (necesario para H2)
      )
    )
    // Fin de Configuraciones adicionales para H2
    .formLogin(form -> form
      .disable()                 // Desactivar login por defecto para APIREST
    )
    .httpBasic(httpBasic -> Customizer
      .withDefaults())           // ??????????
    );
  return http.build();
}
```

Veamos los cambios introducidos:

Permisos de acceso

- **permisos públicos**
  - **/:** acceso a página inicial, que en nuestro caso redirigimos a Swagger
  - **Swagger**: incluimos todas las páginas de swagger
  - **/api/auth**: incluimos todas las páginas del controlador de identificación
- **permisos para el role "USER"**
  - **/productos**: incluimos todas las páginas del controlador de productos
- **permisos para el role "ADMIN"**
  - **/productos**: incluimos todas las páginas del controlador de productos
  - **/productos**: incluimos todas las páginas del controlador de usuarios
  - **/h2-console**: incluimos la consola de H2

CSRF disable

Por defecto es **enable** pero lo deshabilitamos porque:

- El API no usa formularios HTML
- Los clientes de API no pueden manejar tokens CSRF
- Si se usara JWT es necesario que esté disable

Si no lo deshabilitamos, al enviar la petición Spring Security verificaría si hay token CSRF y devolvería "403 Acceso Prohibido"

Formulario de Login disable

Desactivamos el formulario de login por defecto para APIREST ya que la identificación se realiza mediante POST al nuevo controlador **AuthController**.

Httpbasic

Añadimos **.httpBasic(Customizer.withDefaults())** en **APIREST** porque es el mecanismo de autenticación más simple y estándar cuando no se usa **JWT**, **OAuth2**, o tokens personalizados para la autentificación.

Especialmente útil para desarrollo, pruebas y APIs internas.

Al usar SESSION necesitamos que el cliente envíe las credenciales en cada petición para mantener la sesión.

Spring Security con httpBasic() implementa el esquema de autenticación HTTP Basic definido en RFC 7617. En cada petición, el cliente incluye las credenciales codificadas en base64 en el header Authorization. Spring Security decodifica y valida estas credenciales contra el UserDetailsService, y mantiene el estado de autenticación en una sesión HTTP server-side. Esto permite autenticación stateless-from-client-perspective pero stateful-en-el-servidor.

Métodos authProvider y authenticationManager

Estos dos beans son el CORAZÓN de la autenticación en Spring Security.

- **AuthenticationManager** es el coordinador central que decide CÓMO se autentican los usuarios.

DaoAuthenticationProvider es el verificador de credenciales

Spring Boot los crea automáticamente si tenemos un **UserDetailsService** bean y un **PasswordEncoder** bean. pero como nostros definimos nuestro CustomUserDetailsService, necesitamos crearlos.

AUTHCONTROLLER

Este nuevo controlador nos permite la identificación usando APIREST.

Dispone de 3 métodos:

- **GET /api/auth/user** : informa de los datos del usuario identificado
- **POST /api/auth/login**: envía las credenciales de un usuario para la identificación y por lo tanto iniciar sesión
- **POST /api/auth/logout**: cierra la sesión del usuario identificado

Una vez nos hayamos identificado, dependiendo del role que tengamos podremos acceder a unos recursos o a otros, según lo especificado en **SecurityConfig**.

Este controlador, junto a la tabla **users_security** de la BD en los archivos .sql permitirá el acceso a los usuarios existentes.

Si deseamos crear nuevas credenciales para los **INSERT** de **data.sql** podemos usar el proyecto **GenerarBcryptMv**.

Podemos usar este controlador **AuthController** tal cual en nuestros nuevos proyectos, cambiando si lo deseamos la URL de acceso.

GlobalExceptionHandler – Controlar excepción de acceso prohibido

Añadimos al gestor de excepciones la nueva excepción **BadCredentialsException** que se producirá si no es correcta la identificación o autentificación devolviendo el status **"401 - UNAUTHORIZED"**.

Apirest-merge-sec - GlobalExceptionHandler

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(BadCredentialsException.class)
    public ResponseEntity<?> serializacionError(
            BadCredentialsException ex, HttpServletRequest request) {

        Map<String, Object> response = new HashMap<>();
        response.put("timestamp", java.time.LocalDateTime.now());
        response.put("status", 400);
        response.put("error", "BadCredentialsException: Error en usuario o contraseña");
        response.put("message", ex.getMessage());
        response.put("path", request.getRequestURI());
        return new ResponseEntity<>(response, HttpStatus.UNAUTHORIZED);
    }

    ...
```

Si además quisiéramos capturar las excepciones de **UNAUTHORIZED** y **FORBIDDEN**, y mostrar un JSON personalizado podríamos añadirlo en **SecurityConfig** con **exceptionHandling**.

Apirest-merge-sec - SecurityConfig

```
@RestControllerAdvice
public class GlobalExceptionHandler {

  http
    .authorizeHttpRequests((auth -> auth
        // ... tus configuraciones
    )
    .exceptionHandling(exceptions -> exceptions

        // Para 401 UNAUTHORIZED (errores de autenticación)
        .authenticationEntryPoint((request, response, authException) -> {
           Map<String, Object> responseBody = new HashMap<>();
           responseBody.put("timestamp", java.time.LocalDateTime.now());
           responseBody.put("status", 401);
           responseBody.put("error", "Unauthorized");
           responseBody.put("message",
                            getAuthMessage(authException));
           responseBody.put("path", request.getRequestURI());

           response.setStatus(HttpStatus.UNAUTHORIZED.value());
           response.setContentType(MediaType.APPLICATION_JSON_VALUE);
           response.getWriter().write(
                 new ObjectMapper().writeValueAsString(responseBody)
           );
         })

         // Para 403 FORBIDDEN (errores de autorización)
         .accessDeniedHandler((request, response, accessDeniedException) -> {
            Map<String, Object> responseBody = new HashMap<>();
            responseBody.put("timestamp", java.time.LocalDateTime.now());
            responseBody.put("status", 403);
            responseBody.put("error", "Forbidden");
            responseBody.put("message",
                             "No tiene permisos para acceder a este recurso");
            responseBody.put("path", request.getRequestURI());

            response.setStatus(HttpStatus.FORBIDDEN.value());
            response.setContentType(MediaType.APPLICATION_JSON_VALUE);
            response.getWriter().write(
                  new ObjectMapper().writeValueAsString(responseBody)
            );
          })
    )
    ...
```

4.4. Control avanzado con requestMatchers

El método **requestMatchers** y como último método **.anyRequest()** nos permite en **securityFilterChain** aplicar reglas para controlar los accesos.

Los patrones de archivos a indicar en **requestMatchers** permiten controlar el **contenido**:

.requestMatchers("/files/*.txt") **.requestMatchers("/images/*.*")**	URL que acabe con archivos del patrón ✅ http://localhost:8080/app/files/leeme.txt ✅ http://localhost:8080/app/images/foto.jpg

Además, también podemos con un patrón del **path** de la URL que deseamos gestionar:

.requestMatchers("/path")	URL que acabe en **/path** ✅ http://localhost:8080/app/path ❌ http://localhost:8080/app/path/3 ❌ http://localhost:8080/app/path/like/es
.requestMatchers("/path/*")	URL que acabe en **/path/** y posibilidad de algo más que no contenga / ✅ http://localhost:8080/app/path ✅ http://localhost:8080/app/path/3 ❌ http://localhost:8080/app/path/like/es
.requestMatchers("/path/")**	URL que acabe en **/path/** y posibilidad de algo más que contenga o no el carácter / ✅ http://localhost:8080/app/path ✅ http://localhost:8080/app/path/3 ✅ http://localhost:8080/app/path/like/es

Una vez que tengamos capturado nuestro recurso con **requestMatchers** indicaremos como se tiene acceso. De forma general tenemos:

.permitAll()	Acceso público, es decir no necesita estar identificado
.authenticated()	Acceso privado, es decir, necesita estar identificado
.denyAll()	Acceso prohibido, estando identificado o no
.anonymous()	Acceso público, no permitido si ya se está identificado

Por otra parte también podemos dar acceso dependiendo del role que tenga asignado un usuario identificado:

.hasAuthority("juan")	Acceso privado solo al usuario con **username** "juan"
.hasRole("ADMIN")	Acceso privado solo con **role** "ADMIN"
.hasAnyRole("ADMIN","USER")	Acceso privado solo con **role** "ADMIN" o "USER"

Y por último, también podemos diferenciar un mismo **path** pero solo con un método específico:

.requestMatchers(HttpMethod.GET, "/path/")**	
✅ **GET**	http://localhost:8080/app/path
❌ **POST**	http://localhost:8080/app/path

.requestMatchers(HttpMethod.POST, "/path/")**	
❌ **GET**	http://localhost:8080/app/path
✅ **POST**	http://localhost:8080/app/path

.requestMatchers("/path/")**	
✅ **GET**	http://localhost:8080/app/path
✅ **POST**	http://localhost:8080/app/path