Introducción **2**
Crear Proyecto en Spring **3**
Creación de la Base de Datos **9**
Entidades de la base de datos con sus relaciones y reglas 10
BackEnd SpringBoot - Servidor **13**
Class Usuario 13
Interface InmuebleDAO 17
InmuebleServiceImpl 20
FavoritoDTO 23
InmuebleRestController 25
DIFERENCIA ENTRE @RequestParam,@PathVariable,@RequestBody 29
Control de errores **32**
ApiExceptionHandler.java **32**
ErrorResponse.java 34
AutorizacionException.java 35
JWTRequest 36
JWTResponse 37
AuthController 39
JWTService 43
JWTUserDetailsService 45
JWTValidationFilter 47
JWTAuthenticationEntryPoint 50
JWTAccessDeniedHandler 52
SecurityConfig 53
Resumen de flujo de seguridad 56
Subida de archivos(Storage) **57**
UtilidadesStorage 57
StorageService 60
Frontend Angular - Lado del cliente **65**
Frontend (Angular) - Repositorio Git 65
Estructura del proyecto Frontend (Angular) 65
Capturas de pantalla de la Aplicación **73**
Conclusión: **81**

Introducción

El objetivo de este proyecto es desarrollar una **inmobiliaria** como proyecto personal, utilizando **Spring Boot para el backend** y **Angular para el frontend.**

Arquitectura

La aplicación sigue una arquitectura basada en API REST con el patrón Modelo–Vista–Controlador (MVC) y Hibernate como ORM:

1. Modelo (Model)
    - Representado por las entidades de la base de datos.
    - Hibernate se encarga de mapear estas entidades a las tablas de MySQL, actuando como traductor entre Java y la base de datos.
2. Vista (View)
    - En esta API REST no se renderiza HTML.
    - El frontend, desarrollado en Angular, recibe JSON para mostrar la información al usuario.
3. Controlador (Controller)
    - Gestiona las peticiones HTTP entrantes y devuelve respuestas en formato JSON.
    - Aquí se implementa la lógica de negocio para el CRUD y otras operaciones.

Comunicación y CRUD

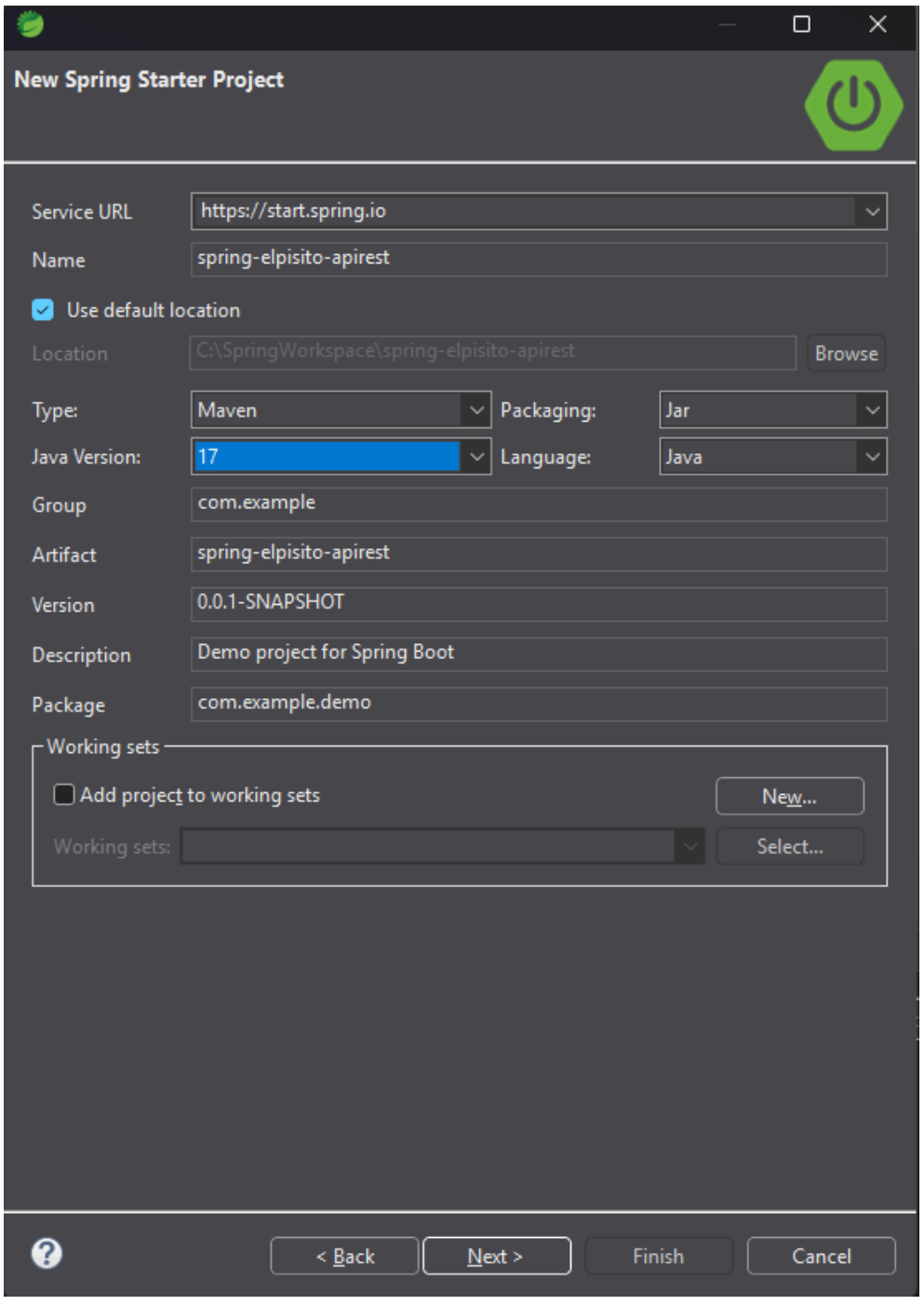
- Las operaciones de Crear, Leer, Actualizar y Borrar (CRUD) se realizan a través de:
    - Angular (HTTP Client)
    - Postman
- Toda la comunicación con el backend se realiza mediante API REST, siguiendo buenas prácticas de desarrollo y separación de capas.

Tecnología destacada

- Backend: Java Spring Boot
- Frontend: Angular
- Base de datos: MySQL
- ORM: Hibernate

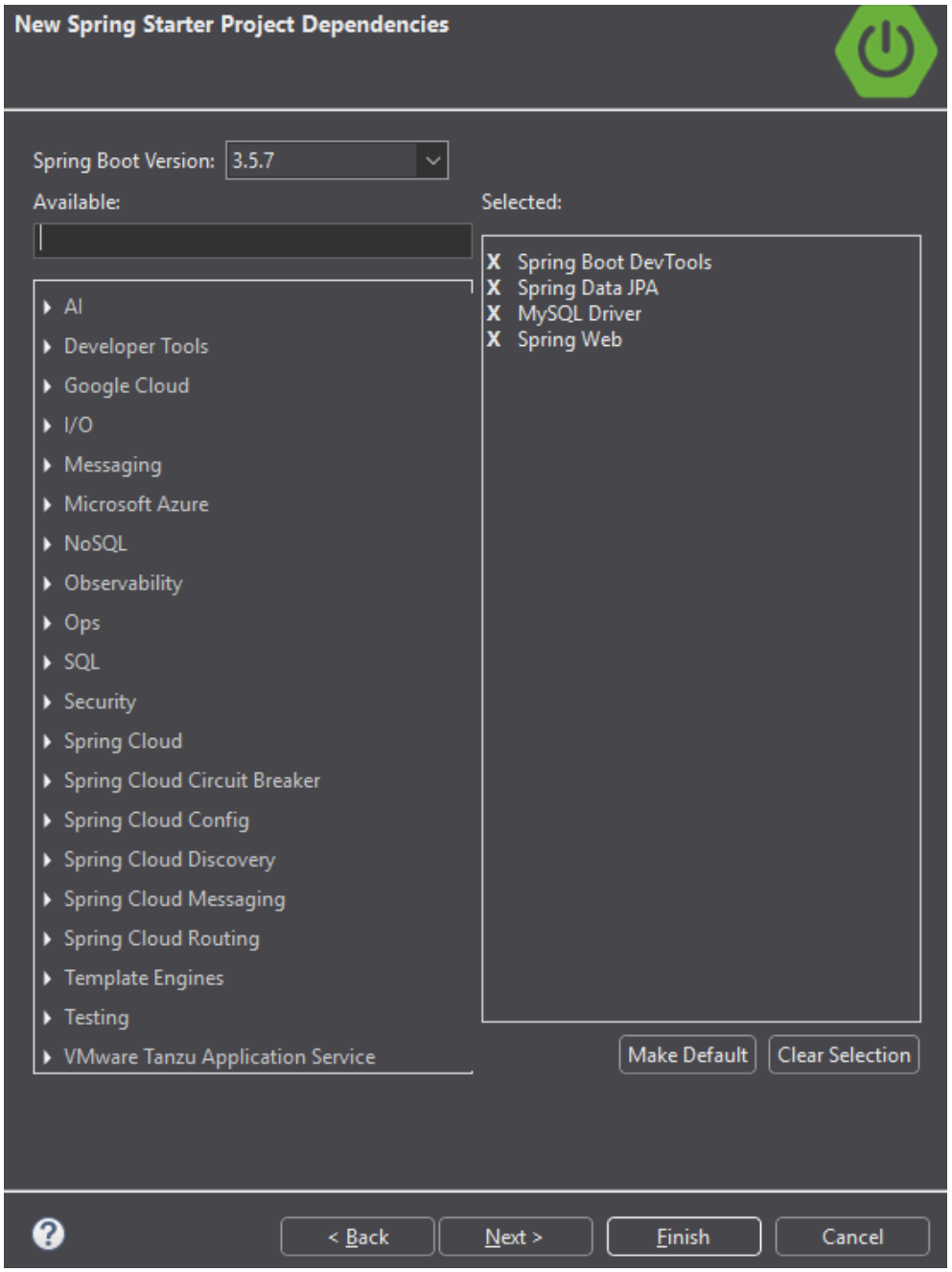
Crear Proyecto en Spring

- Tenemos que elegir un nombre adecuado a nuestro proyecto
- Type: elegimos la herramienta para gestionar el proyecto
- Spring Boot normalmente genera JAR, a no ser que se necesite desplegar en un servidor externo.
- Usaremos Java 17, versión LTS recomendada para Spring Boot moderno.

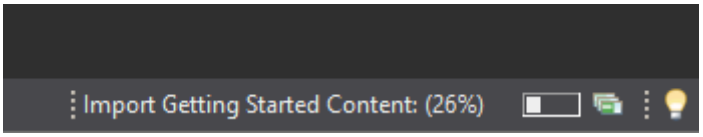


Dependencias a instalar:

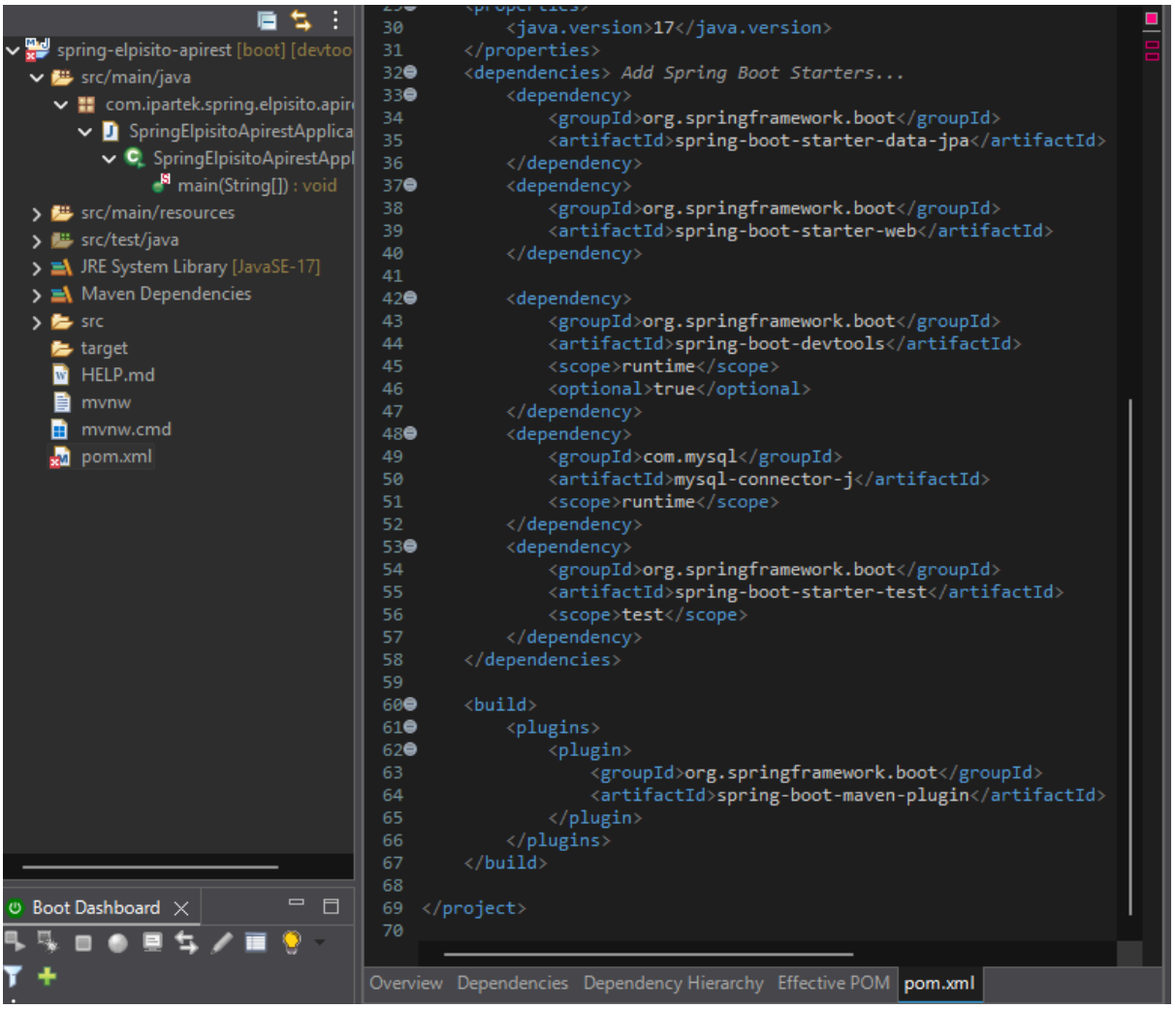
- **Spring Boot DevTools:** facilita el desarrollo recargando la aplicación automáticamente y desactivando cachés para ver cambios al instante.
- **Spring Data JPA:** acceso y gestión de datos en la base de datos sin escribir SQL.
- **MySQL Driver:** permite que la aplicación Java se conecte a MySQL.
- **Spring Web:** crear APIs REST y manejar peticiones HTTP.



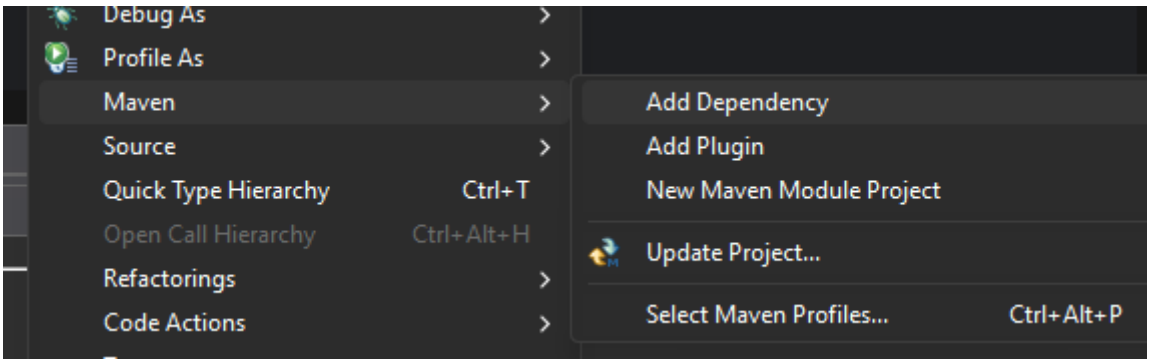
Cuando terminamos de elegir las dependencias y clicamos en finish empezará a descargar e importar archivos en la parte baja izquierda, tendremos que esperar que termine para empezar a realizar el proyecto



En pom.xml podemos ver las dependencias que tenemos instaladas y también podemos instalar más si es necesario(Siempre dentro de dependencies)



Al copiar la dependencia dentro del archivo ya solo quedaría añadirla:
click derecho -> Maven -> Add Dependency



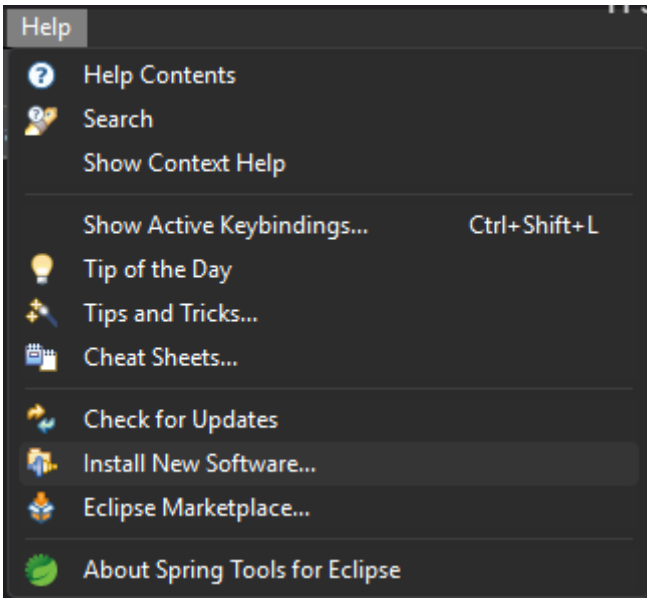
Instalación de Lombok:

Lombok reduce mucho el código repetitivo (getters, setters, constructores…).

Formas de instalación:

1. Desde su página web ejecutando el instalador.

2. Añadiendo un nuevo software en el IDE:



Help -> Install New Software -> Add ->

(Name: vacío, Location: (URL de la dependencia, podemos encontrarla en la página oficial de lombok)

https://mvnrepository.com/artifact/org.projectlombok/lombok

Una vez instalado tenemos que acceder al archivo “pom.xml” y añadir las siguientes líneas dentro de las dependencias “dependencies”

Dependencia en el pom:

```
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<optional>true</optional>
</dependency>
```

Tras esto, podremos usar anotaciones como:

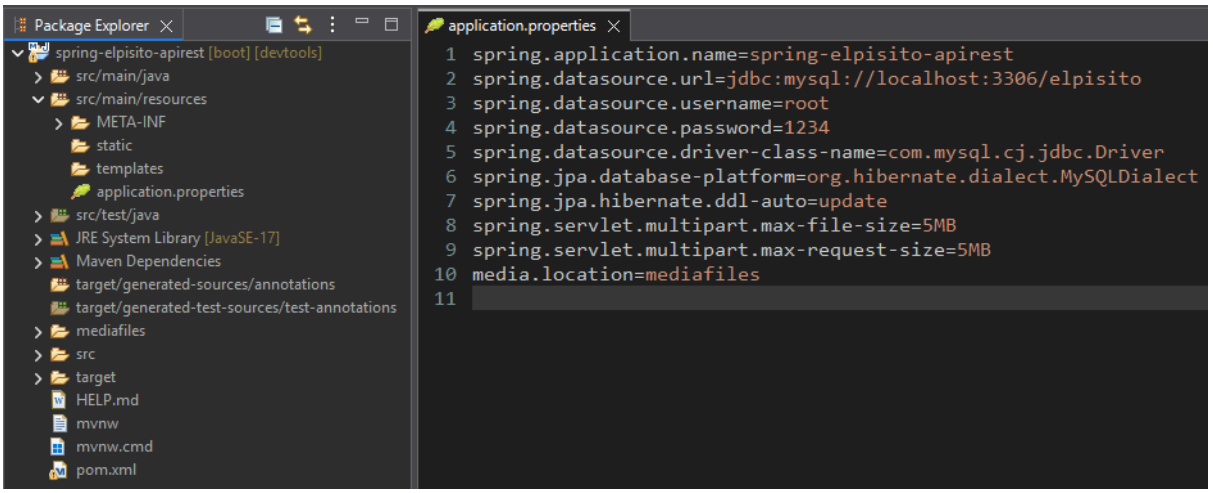
- **@Getter** → genera todos los métodos **getCampo()**
- **@Setter** → genera todos los métodos **setCampo()**
- **@AllArgsConstructor** → genera un constructor con todos los parámetros
- **@NoArgsConstructor** → genera un constructor vacío
- **@ToString** → genera automáticamente el método **toString()**

Application.properties

Es el archivo de configuración principal de Spring Boot donde se definen propiedades de la aplicación (base de datos, servidor, JPA, archivos, etc.).

src/main/resources -> applications.properties

No admite espacios en blanco alrededor del =



spring.application.name: nombre de la aplicación Spring Boot.

spring.datasource.url: URL de conexión a la base de datos MySQL.

spring.datasource.username: usuario de la base de datos.

spring.datasource.password: contraseña de la base de datos.

spring.datasource.driver-class-name: driver JDBC de MySQL.

spring.jpa.database-platform: dialecto de Hibernate para MySQL.

spring.jpa.hibernate.ddl-auto: gestión automática del esquema de la base de datos.

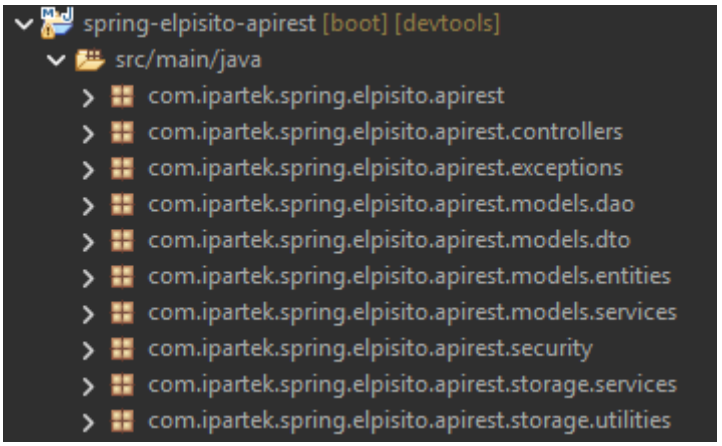
spring.servlet.multipart.max-file-size: tamaño máximo de un archivo subido.

spring.servlet.multipart.max-request-size: tamaño máximo de la petición de subida.

media.location: ruta donde se almacenan los archivos multimedia.

Ya podemos empezar a crear nuestro proyecto, **creando el árbol de carpetas** que organizará el código y los recursos de la aplicación.

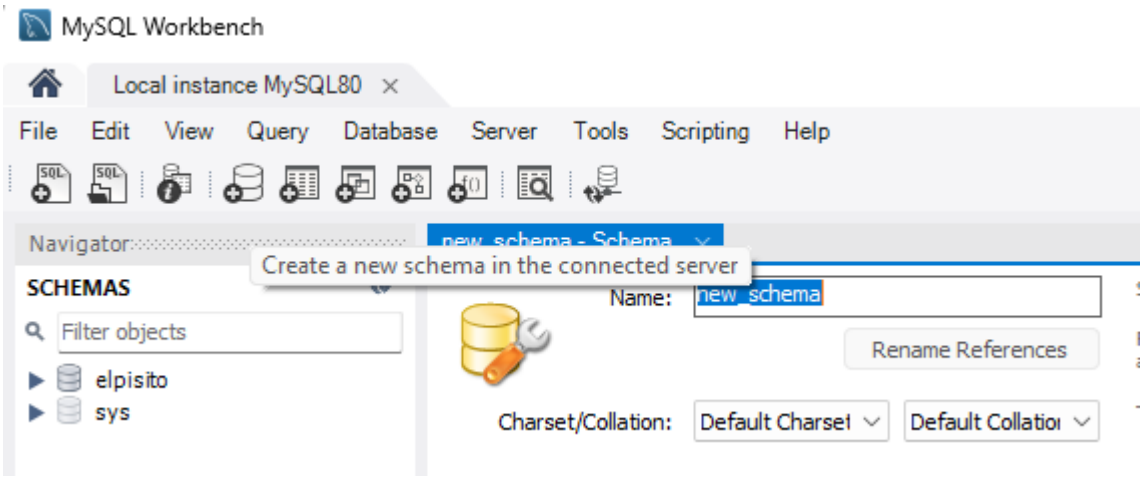
src/main/java ->



Creación de la Base de Datos

Pasos en MySQL Workbench:

1. Crear schema: corresponde al nombre de tu BD.
2. Icono de crear base de datos.
3. Introducir el nombre del schema (ej: elpisito).
4. Clicar en "Apply" para crearlo.



No es necesario crear las tablas manualmente, ya que **Hibernate las genera automáticamente a partir de las entidades** definidas en la aplicación. Esto se explicará más adelante.

Entidades de la base de datos con sus relaciones y reglas

Inmueble (inmuebles)

Representa una propiedad en la inmobiliaria.

Relaciones:

- ManyToOne → tipos
- ManyToOne → poblaciones
- OneToMany → imagenes_inmueble
- ManyToMany → usuarios (mediante usuario_inmueble)
- ManyToOne → operaciones

Reglas:

- Un inmueble debe tener un tipo.
- Un inmueble debe pertenecer a una población.
- Un inmueble puede tener cero o más imágenes.
- Un inmueble puede estar asociado a uno o varios usuarios.
- Un inmueble puede tener varias operaciones a lo largo del tiempo.
- Un inmueble no puede existir sin tipo ni ubicación.

ImagenInmueble (imagenes_inmueble)

Imágenes asociadas a un inmueble.

Relación:

- ManyToOne → inmuebles

Regla:

- Un inmueble puede tener múltiples imágenes.

Usuario (usuarios)

Usuarios del sistema (clientes, agentes, administradores).

Relaciones:

- ManyToMany → inmuebles (tabla intermedia usuario_inmueble)

Regla:

- Un usuario **puede estar asociado a cero o varios inmuebles**.

UsuarioInmueble (usuario_inmueble)

Tabla intermedia para la relación usuarios–inmuebles.

Propósito:

- Asignar inmuebles favoritos a usuarios

Inmobiliaria (inmobiliarias)

Empresas inmobiliarias.

Relación:

- OneToMany → inmuebles

Regla:

- Una inmobiliaria puede gestionar múltiples inmuebles.
- Una inmobiliaria puede existir sin inmuebles asignados.

Operación (operaciones)

Representa operaciones realizadas sobre un inmueble.

Ejemplos:

- Venta, Alquiler

Relaciones:

- ManyToOne → inmuebles

Regla:

- Una operación siempre está asociada a un inmueble.
- Un inmueble puede tener varias operaciones.
- Una operación no puede existir sin un inmueble.

Tipo (tipos)

Clasificación del inmueble.

Ejemplos:

- Piso, Chalet, Lonja vacía

Relación:

- OneToMany → inmuebles

Regla:

- Un inmueble puede tener varias operaciones.
- Una operación no puede existir sin inmueble.

Provincia (provincias)

Provincias geográficas.

Relación:

- OneToMany → poblaciones

Reglas:

- Una provincia puede tener una o varias poblaciones.
- Una población pertenece a una sola provincia.

Población (poblaciones)

Ciudades o municipios.

Relaciones:

- ManyToOne → provincias
- OneToMany → inmuebles

Reglas:

- Una población pertenece a una única provincia.
- Una población puede tener múltiples inmuebles.
- Un inmueble debe pertenecer a una población.

BackEnd SpringBoot - Servidor

Public class Usuario

```
@Getter
@Setter
@AllArgsConstructor
@ToString
@NoArgsConstructor
@Entity
@Table(name="usuarios")
public class Usuario {

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

	@Column(unique=true)
	private String nombre;

	@Column
	private String password;

	@Column(name = "passWord_open")
	private String passwordOpen;

	@Column(unique=true)
	private String email;

	@Column
	private String rol = "ROLE_USER";

	@Column
	private Integer activo = 1;

	@JsonIgnore
	@ManyToMany
	@JoinTable(
			name = "usuario_inmueble",
			joinColumns = {@JoinColumn(name = "usuario_id")},
			inverseJoinColumns =  {@JoinColumn(name = "inmueble_id")}
			)
	private Set<Inmueble> inmueblesFavoritos;
}
```

Ubicación del archivo dentro del proyecto.

package com.ipartek.spring.elpisito.apirest.models.entities;

Sigue la estructura recomendada por Spring Boot:

- models → capa de datos
- entities → clases que representan tablas de la base de datos

Explicación:

- **@Entity:** Indica que la clase es una entidad JPA y corresponde a una tabla en la base de datos.
- **@Table(name="usuarios"):** Define el nombre real de la tabla en MySQL.
- **@Id:** indica la clave primaria de la entidad.
- **@GeneratedValue(strategy = GenerationType.IDENTITY):** genera el valor automáticamente de forma incremental
- **@Column**: mapea el atributo con una columna de la tabla en la base de datos.
    - (name = "passWord_open"): especifica el nombre de la columna en la tabla de la base de datos.
    - (unique = true): indica que el valor de la columna debe ser único en la tabla (no se permiten duplicados).
- **@JsonIgnore**: evita que el campo se incluya en la respuesta JSON (por ejemplo, para ocultar contraseñas).
- **@ManyToMany**: relación muchos a muchos entre entidades.
- **@OneToMany**: relación uno a muchos.
- **@OneToOne**: relación uno a uno.
- **@JoinTable**: especifica la tabla intermedia que gestiona la relación.

Cómo Hibernate ha creado la tabla usuarios

Cuando ejecutas el proyecto Spring Boot, Hibernate revisa todas las clases anotadas con:

@Entity

por ejemplo:

@Entity

@Table(name="usuarios")

public class Usuario { ... }

Hibernate usa esta información para:

1. Generar la tabla en MySQL
2. Crear las columnas correspondientes
3. Asignar tipos de datos (VARCHAR, INT…)
4. Aplicar restricciones como unique = true
5. Configurar claves primarias (@Id)
6. Configurar autoincrementos (GenerationType.IDENTITY)

¿Qué causó que se creara la tabla?

Las siguientes líneas en **application.properties** indican a **Hibernate** que debe crear o actualizar las tablas según tus clases:

spring.jpa.hibernate.ddl-auto=update

El valor update significa:

- Si la tabla no existe, la crea.
- Si la tabla existe pero faltan columnas, las añade.

Traducción de la clase Usuario a tabla SQL:

Clase usuario:

@Column(unique=true)

private String nombre;

Hibernate lo traduce a:

nombre VARCHAR(255) UNIQUE

Clase usuario:

@Id

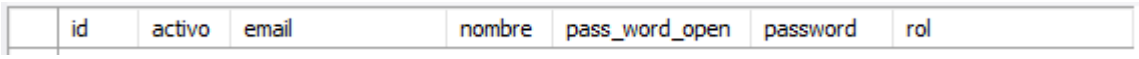
@GeneratedValue(strategy=GenerationType.IDENTITY)

private Long id;

Hibernate lo traduce a:

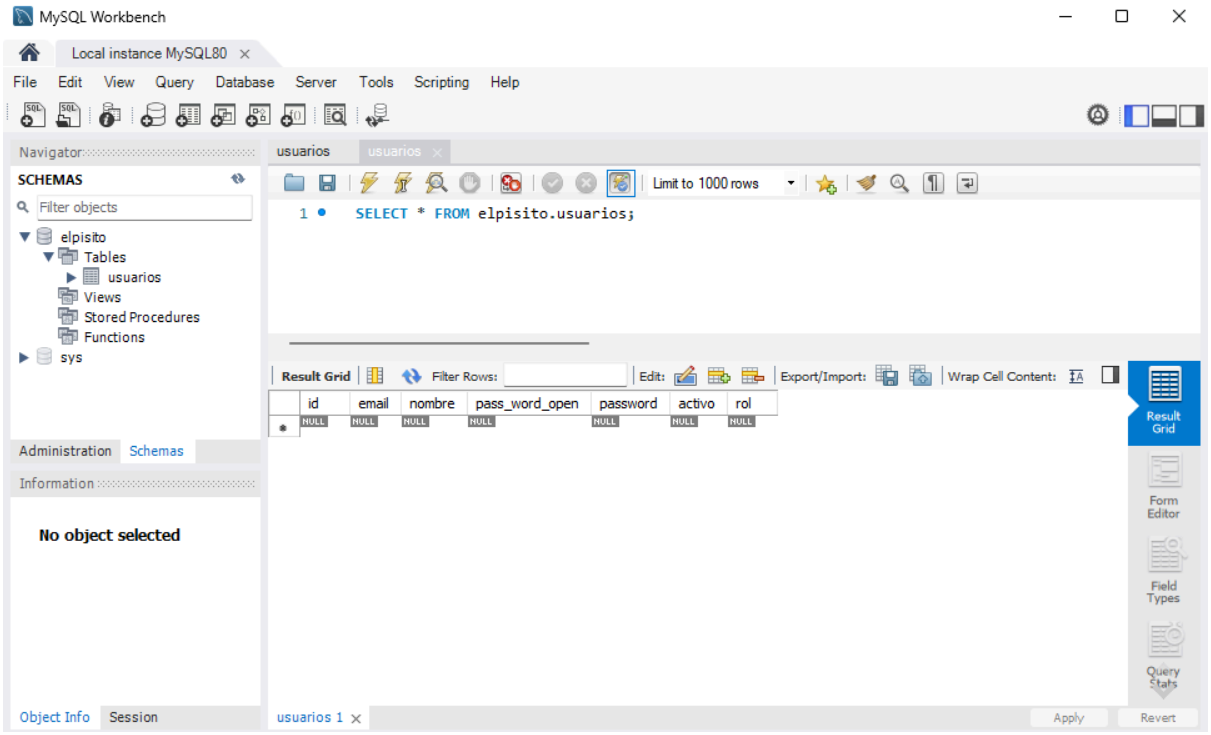
id BIGINT AUTO_INCREMENT PRIMARY KEY

Resultado:



	id	activo	email	nombre	pass_word_open	password	rol

Todo ha sido creado automáticamente por Hibernate usando la entidad.



Resumen

La clase **Usuario**:

- Es una entidad JPA que representa la tabla **usuarios**.
- Hibernate la usa para crear automáticamente la tabla en la base de datos.
- Define campos como columnas SQL, incluyendo:
    - restricciones unique
    - autoincremento
    - valores por defecto
    - roles y estados del usuario

Interface InmuebleDAO

1. Propósito del archivo

InmuebleDAO es una interfaz de acceso a datos (DAO) que define cómo interactuar con la base de datos para la entidad **Inmueble**.
Al extender de **JpaRepository**, hereda métodos estándar de CRUD (**save, findById, findAll, delete**, etc.) y además define consultas personalizadas específicas para la aplicación.

Este archivo es clave para separar la lógica de acceso a datos de la lógica de negocio o de los controladores de la API, siguiendo el patrón Repository de Spring Data JPA.

2. Paquete y anotaciones

package com.ipartek.spring.elpisito.apirest.models.dao;

```
@Repository
public interface InmuebleDAO extends JpaRepository<Inmueble, Long> {
```

- **@Repository** Indica a Spring que esta interfaz es un componente de acceso a datos.

3. Extensión de JpaRepository

```
public interface InmuebleDAO extends JpaRepository<Inmueble, Long> {
```

- **Inmueble** → la entidad sobre la que se realizan operaciones.
- **Long** → tipo del ID de la entidad.
- Al extender **JpaRepository** se obtienen automáticamente métodos como:
    - **findAll()** → obtener todos los registros.
    - **findById(Long id)** → buscar por ID.
    - **save(Inmueble entity)** → insertar o actualizar un registro.
    - **deleteById(Long id)** → eliminar por ID.

Esto reduce mucho el código repetitivo en la API.

4. Consultas personalizadas

4.1 findByActivo

```
        List<Inmueble> findByActivo(Integer activo);
```

- Busca todos los inmuebles filtrando por el campo activo.
    - Ejemplo de uso:

List<Inmueble> activos = inmuebleDAO.findByActivo(1);

4.2 findByActivoAndPortada

```
List<Inmueble> findByActivoAndPortada(Integer activo, Integer portada);
```

- Filtra inmuebles que estén activos y que tengan la bandera de portada.
    - Ejemplo de uso:

```
List<Inmueble> destacados = inmuebleDAO.findByActivoAndPortada(1, 1);
```

4.3 findByTipoAndPoblacionAndOperacionAndActivo

```
List<Inmueble> findByTipoAndPoblacionAndOperacionAndActivo(Tipo tipo,
Poblacion poblacion, Operacion operacion, Integer activo);
```

- Filtra inmuebles por múltiples criterios: tipo, población, operación y estado activo.
- Muy útil para filtros avanzados en el front-end.
    - Ejemplo de uso:

```
List<Inmueble> resultados =

inmuebleDAO.findByTipoAndPoblacionAndOperacionAndActivo(

  tipo, poblacion, operacion, 1);
```

4.4 findByInmobiliariaAndActivo

```
List<Inmueble> findByInmobiliariaAndActivo(Inmobiliaria inmobiliaria, Integer activo);
```

- Obtiene todos los inmuebles activos de una inmobiliaria específica.
    - Ejemplo de uso:

```
List<Inmueble> misInmuebles = inmuebleDAO.findByInmobiliariaAndActivo(miInmobiliaria,
1);
```

5. Notas y buenas prácticas

1. Nombrado de métodos: Spring Data JPA interpreta los nombres de los métodos y genera las consultas automáticamente. Mantener nombres claros y coherentes es crucial.

2. Filtros combinados: Puedes crear consultas combinando varios campos como findByCampo1AndCampo2...

3. Evitar lógica de negocio en DAO. **DAO solo debe acceder a datos**, **cualquier cálculo o validación debe ir en el servicio.**

4. Optional: Para búsquedas que devuelven un solo objeto, se puede usar Optional<Inmueble> para manejar nulos de forma elegante.

InmuebleServiceImpl

1. Propósito del archivo

InmuebleServiceImpl es la implementación de la capa de servicio para la entidad **Inmueble**. Su función es gestionar la lógica de negocio relacionada con inmuebles, utilizando los DAO para acceder a los datos.

Esta clase actúa como intermediaria entre los controladores (que reciben las peticiones HTTP) y los DAO (que acceden a la base de datos).

2. Anotación y dependencias

```
@Service
public class InmuebleServiceImpl implements GeneralService<Inmueble> {
```

- Marca esta clase como un componente de servicio gestionado por Spring.
- Permite la inyección automática en los controladores o en otras clases.

```
    @Autowired
    private InmuebleDAO inmuebleDAO;

    @Autowired
    private TipoDAO tipoDAO;

    @Autowired
    private PoblacionDAO poblacionDAO;

    @Autowired
    private OperacionDAO operacionDAO;

    @Autowired
    private InmobiliariaDAO inmobiliariaDAO;
```

- Se inyectan los DAO necesarios para realizar operaciones sobre las entidades relacionadas.
- Esto facilita la reutilización de métodos y garantiza el principio de inversión de dependencias.

3. Implementación de GeneralService<Inmueble>

Esta clase implementa la interfaz genérica GeneralService para manejar operaciones comunes de la entidad Inmueble.

Métodos sobreescritos:

```
	@Override
	public List<Inmueble> findAll() {
		return inmuebleDAO.findAll();
	}
```

- Devuelve una lista con todos los inmuebles de la base de datos.
- Utiliza el método estándar findAll de Spring Data JPA.

```
	@Override
	public List<Inmueble> findAllActivo() {
		return inmuebleDAO.findByActivo(1);
	}
```

- Obtiene todos los inmuebles cuyo campo activo está en 1 (activos).
- Usa la consulta personalizada del DAO.

```
	@Override
	public Inmueble save(Inmueble i) {
		return inmuebleDAO.save(i);
	}
```

- Inserta o actualiza un inmueble en la base de datos.
- Usa el método estándar save.

```
	@Override
	public Inmueble findById(Long id) {
		return inmuebleDAO.findById(id).orElseThrow(()-> new
RecursoNoEncontradoException("El inmueble con id: " + id + " no existe en la BBDD"));
	}
```

- Busca un inmueble por su ID.
- Si no lo encuentra, lanza una excepción personalizada RecursoNoEncontradoException para notificar que el recurso no existe.

4. Métodos adicionales específicos

```
	public List<Inmueble>findAllPortada(){
		return inmuebleDAO.findByActivoAndPortada(1, 1);
	}
```

- Obtiene los inmuebles activos que además están marcados como portada (destacados).

```
	public List<Inmueble> finder(Long idTipo, Long idPoblacion, Long idOperacion){
		Tipo tipo = tipoDAO.findById(idTipo).orElseThrow(()-> new
RecursoNoEncontradoException("El tipo con id: " + idTipo + " que estás intentado utilizar en el finder
no existe en la BBDD"));
		Poblacion poblacion = poblacionDAO.findById(idPoblacion).orElseThrow(()-> new
RecursoNoEncontradoException("La población con id: " + idPoblacion + " que estás intentado utilizar
en el finder no existe en la BBDD"));
		Operacion  operacion = operacionDAO.findById(idOperacion).orElseThrow(()-> new
RecursoNoEncontradoException("La operación con id: " + idOperacion + " que estás intentado utilizar
en el finder no existe en la BBDD"));

		return inmuebleDAO.findByTipoAndPoblacionAndOperacionAndActivo(tipo,
poblacion, operacion, 1);
	}
```

- Método complejo que permite buscar inmuebles filtrando por tipo, población y operación, asegurándose que todos los parámetros existan.
- Si alguno no existe, lanza una excepción específica con mensaje claro.
- Retorna solo inmuebles activos.

```
	public List<Inmueble> findAllInmueblesInmobiliaria(Long idInmobiliaria) {
		Inmobiliaria inmobiliaria = inmobiliariaDAO.findById(idInmobiliaria).orElseThrow(()->
new RecursoNoEncontradoException("El inmobiliairia con id: " + idInmobiliaria + " que estás intentado
utilizar en el finder no existe en la BBDD"));
		return inmuebleDAO.findByInmobiliariaAndActivo(inmobiliaria,1);
	}
```

- Obtiene todos los inmuebles activos que pertenecen a una inmobiliaria específica.
- Controla que la inmobiliaria exista para evitar errores en la consulta.

5. Manejo de excepciones

- En varios métodos, ante la ausencia de datos relevantes (tipo, poblacion, operacion, inmobiliaria o inmueble), se lanza la excepción personalizada RecursoNoEncontradoException.

- Esto ayuda a que el controlador devuelva un error HTTP 404 con mensaje claro al cliente.

6. Buenas prácticas y recomendaciones

- Validaciones tempranas: Se asegura que los objetos relacionados existan antes de realizar consultas, previniendo errores.

- Separación clara de responsabilidades: La capa servicio no accede directamente a la base de datos, sino mediante DAO.
- Reutilización: Métodos genéricos y específicos para diferentes casos de uso.
- Excepciones personalizadas: Facilitan la gestión centralizada de errores en la API.

FavoritoDTO

1. Propósito del archivo

FavoritoDTO es un Data Transfer Object (DTO) que representa una versión simplificada y específica de los datos relacionados con un inmueble favorito.

Los DTOs se usan para transferir datos entre capas (por ejemplo, entre servicio y controlador o entre servidor y cliente) sin exponer directamente las entidades completas, que pueden tener más campos o información sensible.

En este caso, FavoritoDTO agrupa datos importantes para mostrar o procesar un inmueble favorito en la aplicación, pero evita exponer la entidad completa Inmueble.

¿Para qué sirve un DTO en Spring?

1. **SEPARACIÓN DE RESPONSABILIDADES:**
   Permite que las distintas capas de una aplicación tengan responsabilidades claras y no dependan directamente unas de otras.

2. **FLEXIBILIDAD Y CONTROL:**
   Un DTO puede contener solo los datos necesarios para una operación específica, evitando exponer información sensible o irrelevante de las entidades de la base de datos.

3. **TRANSFERENCIA DE DATOS SIMPLIFICADA:**
   Facilita la transferencia de datos entre las distintas partes de la aplicación, haciendo este proceso más ligero.

4. **MEJORA EL RENDIMIENTO:**
   Al enviar menos datos en una respuesta, se puede reducir el tráfico en la red y mejorar el rendimiento de la aplicación.

5. **FACILITA LA REUTILIZACIÓN DE CÓDIGO:**
   Los DTOs pueden ser reutilizados en diferentes partes de la aplicación que requieran los mismos datos.

6. **PROTECCIÓN DE ENTIDADES:**
   Los DTOs ayudan a proteger las entidades de la base de datos de cambios no deseados o de su exposición a capas externa

3. Atributos

Atributo	Tipo	Descripción
idInmueble	Long	Identificador único del inmueble.
nombreTipo	String	Nombre del tipo de inmueble (p. ej., “Apartamento”).
nombrePoblacion	String	Nombre de la población donde se encuentra el inmueble.
nombreProvincia	String	Nombre de la provincia.
nombreOperacion	String	Tipo de operación (venta, alquiler, etc.).
precio	Double	Precio del inmueble.
nombreInmobiliaria	String	Nombre de la inmobiliaria que gestiona el inmueble.

4. Uso típico

- Se utiliza para enviar datos al cliente con la información relevante y legible del inmueble favorito.
- Puede ser construido a partir de una entidad Inmueble y sus relaciones para transformar los datos antes de enviarlos.
- Facilita desacoplar la API de las entidades internas, permitiendo modificar la base de datos sin afectar las respuestas.

5. Ejemplo de creación y uso

```
FavoritoDTO favorito = new FavoritoDTO(
        inmueble.getId(),
        inmueble.getTipo().getNombre(),
        inmueble.getPoblacion().getNombre(),
        inmueble.getPoblacion().getProvincia().getNombre(),
        inmueble.getOperacion().getNombre(),
        inmueble.getPrecio(),
        inmueble.getInmobiliaria().getNombre()
    );
```

Este DTO podría ser devuelto en un endpoint que consulta los inmuebles favoritos del usuario, mostrando solo los campos esenciales y fáciles de entender.

6. Beneficios del uso de DTOs en general

- Mejor control sobre qué datos se exponen en la API.
- Facilita la validación y transformación de datos.
- Reduce el volumen de información enviada (no se envían campos innecesarios).
- Mejora la seguridad evitando exponer campos sensibles.
- Facilita la evolución del API sin romper contratos con clientes.

InmuebleRestController

1. Propósito del archivo

InmuebleRestController es la capa de controlador REST encargada de gestionar las solicitudes HTTP relacionadas con los inmuebles.

Esta clase recibe peticiones del cliente (por ejemplo, navegador o app móvil), invoca los servicios correspondientes y devuelve respuestas con el formato adecuado (generalmente JSON).

2. Anotaciones principales

@RestController

@RequestMapping("/api")

- @RestController indica que esta clase es un controlador REST y que todos los métodos devolverán respuestas en formato JSON (o similar).
- @RequestMapping("/api") define el prefijo común para todos los endpoints, que en este caso es /api.

3. Inyección de dependencia

```
    @Autowired
    private InmuebleServiceImpl inmuebleService;
```

- Se inyecta la implementación del servicio de inmuebles para poder llamar a la lógica de negocio.

4. Endpoints y métodos

4.1 GET /api/inmuebles

```
    @GetMapping("/inmuebles")
    public ResponseEntity<List<Inmueble>> findAll(){
            return ResponseEntity.ok(inmuebleService.findAll());//200
    }
```

- Obtiene todos los inmuebles de la base de datos (activos o no).
- Devuelve un código HTTP 200 (OK) con la lista de inmuebles.

4.2 GET /api/inmuebles-activos

```
	@GetMapping("/inmuebles-activos")
	public ResponseEntity<List<Inmueble>> findAllActivos(){
		return ResponseEntity.ok(inmuebleService.findAllActivo());//200
	}
```

- Obtiene solo los inmuebles que están activos (activo = 1).
- Código HTTP 200 (OK) con la lista filtrada.

4.3 GET /api/inmueble/{id}

```
	@GetMapping("/inmueble/{id}")
	public ResponseEntity<Inmueble> findById(
			@PathVariable Long id
			){
		return ResponseEntity.ok(inmuebleService.findById(id));//200
	}
```

- Busca un inmueble por su ID.
- Devuelve un único inmueble con HTTP 200 si existe.

4.4 POST /api/inmueble

```
	@PostMapping("/inmueble")
	public ResponseEntity<Inmueble> create(@RequestBody Inmueble inmueble){
		return
ResponseEntity.status(HttpStatus.CREATED).body(inmuebleService.save(inmueble));//201
	}
```

- Crea un nuevo inmueble recibiendo un JSON con los datos en el cuerpo de la petición.
- Devuelve HTTP 201 (Created) con el inmueble creado, que incluirá el ID generado.

4.5 PUT /api/inmueble

```
	@PutMapping("/inmueble")
	public ResponseEntity<Inmueble> update(@RequestBody Inmueble inmueble){
		return ResponseEntity.ok(inmuebleService.save(inmueble));//200
	}
```

- Actualiza un inmueble existente.
- Recibe un JSON con los datos y el ID del inmueble a actualizar.
- Devuelve HTTP 200 con el inmueble actualizado.

4.6 GET /api/inmuebles-portada

```
	@GetMapping("/inmuebles-portada")
	public ResponseEntity<List<Inmueble>> findAllPortada(){
		return ResponseEntity.ok(inmuebleService.findAllPortada()); //200

	}
```

- Devuelve los inmuebles activos que están marcados como portada.
- Código HTTP 200 con la lista.

4.7 GET /api/inmuebles/{idTipo}/{idPoblacion}/{idOperacion}

```
	@GetMapping("/inmuebles/{idTipo}/{idPoblacion}/{idOperacion}")
	public ResponseEntity<List<Inmueble>> finder(
			@PathVariable Long idTipo,
			@PathVariable Long idPoblacion,
			@PathVariable Long idOperacion
			){
		return
ResponseEntity.ok(inmuebleService.finder(idTipo,idPoblacion,idOperacion)); //200

	}
```

- Búsqueda avanzada que filtra inmuebles por tipo, población y operación, todos activos.
  Parámetros en la URL.
- HTTP 200 con resultados.

4.8 GET /api/inmuebles-inmobiliaria/{idInmobiliaria}

```
	@GetMapping("/inmuebles-inmobiliaria/{idInmobiliaria}")
	public ResponseEntity<List<Inmueble>> findAllInmueblesInmobiliaria(
			@PathVariable Long idInmobiliaria
			){
		return
ResponseEntity.ok(inmuebleService.findAllInmueblesInmobiliaria(idInmobiliaria));
	}
```

- Devuelve todos los inmuebles activos que pertenecen a una inmobiliaria específica.
- Parámetro en la URL.
- HTTP 200 con la lista.

5. Uso de ResponseEntity

- Todos los métodos devuelven ResponseEntity<T>, que permite controlar el código HTTP y el cuerpo de la respuesta.
- Ejemplos:
    - ResponseEntity.ok(data) → 200 OK
    - ResponseEntity.status(HttpStatus.CREATED).body(data) → 201 Created

6. Manejo de errores

- El controlador asume que las excepciones como RecursoNoEncontradoException lanzadas por el servicio serán manejadas por un manejador global de excepciones, (ApiExceptionHandler) devolviendo respuestas adecuadas (por ejemplo, 404).
- Esto evita tener que poner lógica de manejo de errores en cada método.
- En el controlador no se hacen try catch, el manejo de errores se hace en el servicio

7. Buenas prácticas reflejadas

- Uso claro de verbos HTTP adecuados (GET, POST, PUT).
- Diseño RESTful con URLs semánticas.
- Separación clara de responsabilidades: el controlador solo recibe, llama al servicio y devuelve la respuesta.
- Uso consistente de ResponseEntity para control de estado HTTP.
- Inyección automática del servicio para fácil mantenimiento y pruebas.

Resumen general

Este controlador define la interfaz pública para la gestión de inmuebles en la API REST. Implementa todos los endpoints necesarios para crear, leer, actualizar y buscar inmuebles, utilizando la capa de servicios para toda la lógica.

DIFERENCIA ENTRE
@RequestParam,@PathVariable,@RequestBody

1. @RequestParam

- Se usa para recibir parámetros de la URL en forma de query string (parte después del ? en la URL).
- Es opcional por defecto si se indica required = false.
- Útil para filtros, búsquedas, opciones opcionales, etc.

Ejemplo:

```
	@PostMapping("/usuario-inmueble")
	public ResponseEntity<FavoritoDTO> addFavorito(
			@RequestParam("usuid") Long usuarioId,
			@RequestParam("inmid") Long inmuebleId
			) {

			return
ResponseEntity.status(HttpStatus.CREATED).body(favoritoService.addFavorito(usuarioId,
inmuebleId));//201

	}
```

Llamada HTTP:

POST /usuario-inmueble?usuid=1&inmid=2

2. @PathVariable

(a veces llamado @PathParam en JAX-RS, pero en Spring se usa @PathVariable)

- Se usa para extraer valores directamente de la ruta (path) de la URL.
- Ideal para identificadores o recursos específicos.
- Es obligatorio por defecto, aunque se puede hacer opcional con parámetros required.

Ejemplo:

```
	@GetMapping("/inmuebles/{idTipo}/{idPoblacion}/{idOperacion}")
	public ResponseEntity<List<Inmueble>> finder(
				@PathVariable Long idTipo,
				@PathVariable Long idPoblacion,
```

```
                        @PathVariable Long idOperacion
                        ){
                return
ResponseEntity.ok(inmuebleService.finder(idTipo,idPoblacion,idOperacion)); //200

        }
```

Llamada HTTP:

GET /inmuebles/1/2/3

3. @RequestBody

- Se usa para recibir el cuerpo de la petición HTTP, generalmente en formato JSON o XML.
- Es obligatorio por defecto (pero se puede hacer opcional con required = false).
- Ideal para crear o actualizar recursos, cuando envías objetos complejos con varias propiedades.

Ejemplo:

```
        @PostMapping("/inmueble")
        public ResponseEntity<Inmueble> create(@RequestBody Inmueble inmueble){
                return
ResponseEntity.status(HttpStatus.CREATED).body(inmuebleService.save(inmueble));//201
        }
```

Llamada HTTP:

POST /inmueble

Content-Type: application/json

{

"nombre": "Apartamento céntrico",

"precio": 120000,

"tipo": "Apartamento"

}

Spring convierte automáticamente el JSON en un objeto Inmueble.

4. Tip práctico:

- Usa @RequestParam para parámetros opcionales o filtros.
- Usa @PathVariable para identificadores obligatorios en la ruta.
- Usa @RequestBody para enviar objetos completos en JSON o XML.

Control de errores

ApiExceptionHandler.java

1. Propósito del archivo

ApiExceptionHandler es un controlador global de excepciones para la API REST en Spring Boot:

- Anotado con @RestControllerAdvice, intercepta automáticamente cualquier excepción lanzada desde los servicios. Devuelve un objeto uniforme (ErrorResponse) con información del error al cliente. Permite centralizar el manejo de errores, evitando duplicar código en cada controlador.

- Una aplicación puede tener varios @RestControllerAdvice: la respuesta es sí. No solo puede tener varios, sino que además es recomendado. Es buena práctica tener un @RestControllerAdvice general y después tener otros más especializados, por ejemplo uno especializado en recoger y tratar errores de seguridad.

- Si tenemos varios, ¿cuál se llama primero, en qué orden? Para ello existe una anotación complementaria llamada @Order en la que establecemos el orden de llamada, por ejemplo @Order(1).

- ¿Podríamos tener una excepción duplicada tratada en varios @RestControllerAdvice? En teoría sí, pero existe una norma de buena práctica que nos dice que no debemos duplicar handlers para tratar un mismo error salvo que esté debidamente justificado.

- Spring Security no envía excepciones al @RestControllerAdvice porque son errores internos. Sin embargo, las excepciones que se producen en nuestro código interno, por ejemplo en un controller, sí son enviadas.

2. Flujo de ejecución

Cuando ocurre una excepción en un endpoint:

Cliente ---> Controlador ---> Servicio ---> (Excepción) ---> @RestControllerAdvice ---> Cliente

Si todo va bien, el flujo normal es:

Cliente ---> Controlador ---> Servicio ---> Repositorio ---> Servicio ---> Controlador ---> Cliente

3. Método de utilidad build

```
        private ResponseEntity<ErrorResponse> build(HttpStatus status, Exception ex,
HttpServletRequest req ){
                ErrorResponse error = new ErrorResponse(
                                status.value(),
                                status.getReasonPhrase(),
                                ex.getMessage(),
                                req.getRequestURI()

                                );
                return ResponseEntity.status(status).body(error);
        }
```

- Construye un ErrorResponse estándar.
- Recibe:
    - status: código HTTP.
    - ex: excepción lanzada.
    - req: request para capturar la URI del endpoint.

- Retorna ResponseEntity<ErrorResponse> listo para enviar al cliente.

4. Beneficios de esta estructura

1. Uniformidad: todas las respuestas de error tienen el mismo formato (ErrorResponse).

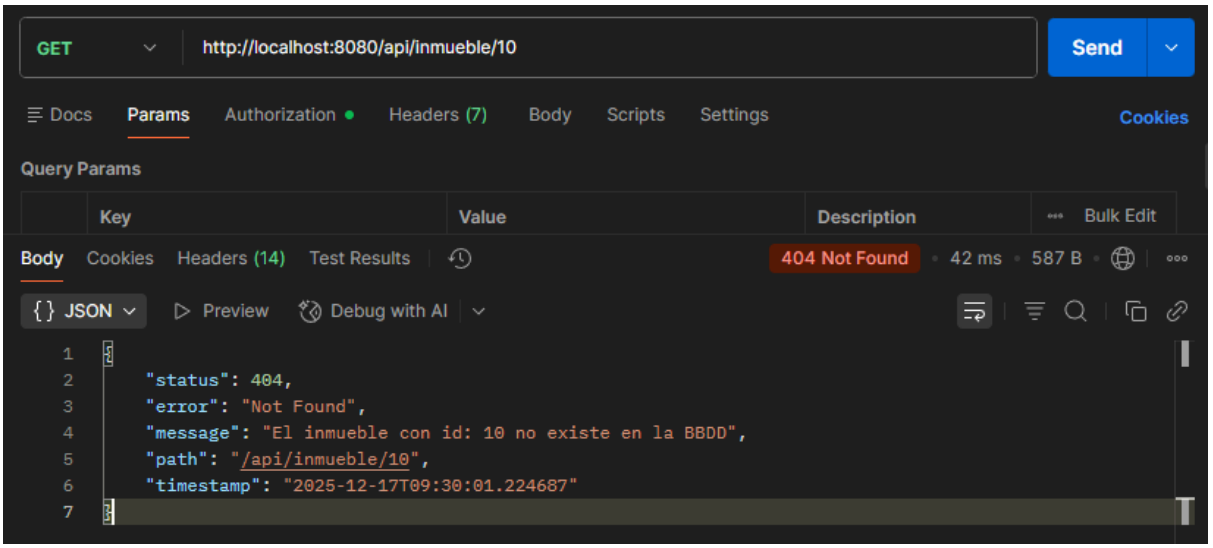
2. Centralización: evita duplicar el manejo de errores en servicios.

3. Claridad para clientes: código HTTP + mensaje + ruta + timestamp.

4. Flexibilidad: puedes agregar manejadores específicos según tipo de excepción.

5. Extensibilidad: fácil de combinar con Swagger/OpenAPI para documentar errores.

5. Ejemplo de respuesta JSON estándar



ErrorResponse.java

1. Propósito del archivo

ErrorResponse es un DTO (Data Transfer Object) utilizado para enviar información sobre errores al cliente de manera consistente y estructurada.

- No tiene relación con la base de datos (no es una entidad JPA).
- Su objetivo es uniformar la respuesta de error en toda la API REST, haciendo más fácil para el cliente entender qué ocurrió.

2. Constructor

```
    public ErrorResponse(int status, String error, String message, String path) {
            this.timestamp = LocalDateTime.now();
            this.status = status;
            this.error = error;
            this.message = message;
            this.path = path;
    }
```

- Inicializa todos los campos necesarios para enviar la respuesta de error al cliente.
- timestamp se asigna automáticamente al momento de crear la instancia, para reflejar la hora exacta del error.

AutorizacionException.java

1. Propósito del archivo

AutorizacionException es una excepción personalizada que representa un error de autorización en la API REST.

- Se lanza cuando un usuario intenta acceder a un recurso o realizar una acción para la que no tiene permisos.
- Extiende RuntimeException, lo que significa que es una excepción no verificada.
- Permite centralizar y manejar errores de autorización de forma clara y consistente.

2. Serialización

```
    @Serial
    private static final long serialVersionUID = -6618213959357055284L;
```

- Se define serialVersionUID para asegurar la compatibilidad en la serialización de la excepción, útil si se envía a través de streams o se guarda en logs.
- Anotación @Serial es opcional pero recomendada en Java 14+ para indicar que este campo está relacionado con serialización.

3. Constructor

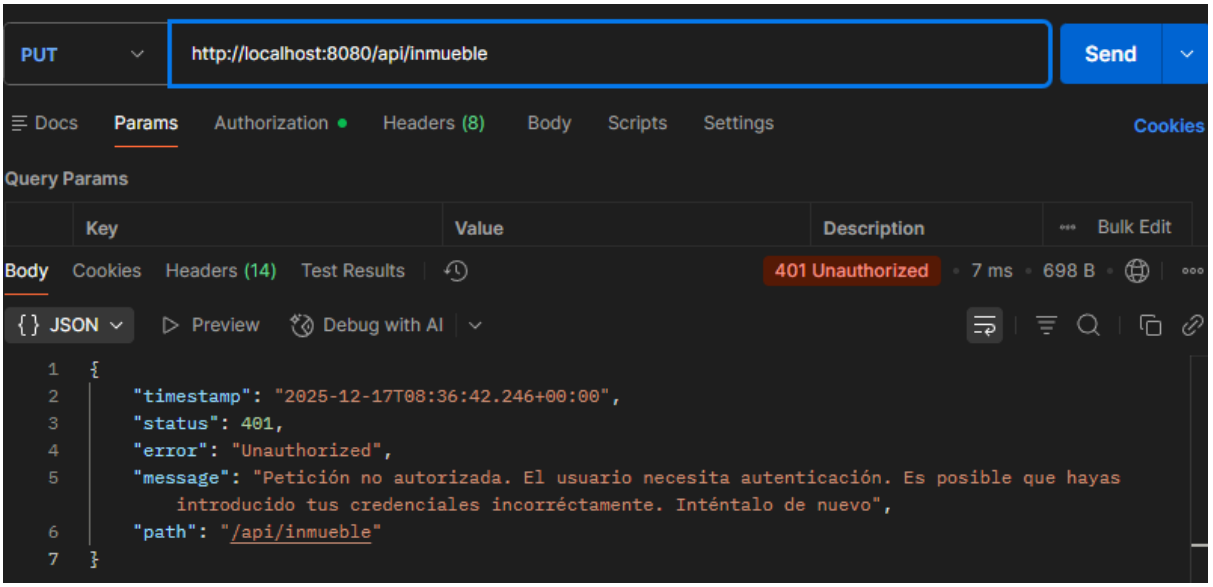
```
    public AutenticacionException(String msg) {
            super(msg);
    }
```

- Permite crear la excepción con un mensaje personalizado que describe el motivo de la falta de autorización.
- Este mensaje puede luego ser incluido en la respuesta JSON enviada al cliente.

4. Uso típico

Se utiliza en la capa de servicio o controlador para detectar accesos no autorizados:

```
if (!usuarioTienePermiso(usuario, recurso)) {
    throw new AutorizacionException("No tienes permisos para acceder a este recurso");
}
```



5. Beneficios

- Permite identificar errores de autorización de manera clara.
- Facilita que los clientes y el frontend manejen correctamente los accesos restringidos.
- Funciona junto con ErrorResponse para respuestas estandarizadas en toda la API.
- Mejora la mantenibilidad y la seguridad del proyecto.

JWTRequest

1. Propósito del archivo

JWTRequest es un DTO (Data Transfer Object) que representa la estructura de la solicitud de autenticación enviada por el cliente al servidor. Contiene los datos necesarios para que el servidor valide al usuario y genere un token JWT.

2. Detalles

- Anotada con @Data de Lombok, que genera automáticamente:
    - Getters y Setters para los atributos.
    - Métodos hashCode(), equals() y toString() usados internamente por Spring Security y otras utilidades.

- Contiene dos atributos:
    - username: nombre de usuario.
    - password: contraseña asociada al usuario.

3.Uso en la API

Cuando un cliente realiza una petición para autenticarse, debe enviar un objeto JSON con esta estructura en el cuerpo de la petición (request body):

```
{
  "username": "ejemploUsuario",
  "password": "miPasswordSecreto"
}
```

Este objeto será mapeado automáticamente a una instancia de JWTRequest por Spring, facilitando la validación y generación del token.

4. Por qué usar un DTO

Separar la estructura de los datos entrantes en un DTO específico permite:

- Validar los datos recibidos con mayor claridad.
- Mantener la seguridad al no exponer directamente las entidades del sistema.
- Facilitar el mantenimiento y evolución de la API, pudiendo modificar el DTO sin afectar el modelo interno.

JWTResponse

1. Propósito del archivo

JWTResponse es un DTO (Data Transfer Object) que representa la respuesta que el servidor envía al cliente tras una autenticación exitosa. Contiene el token JWT generado y otros mensajes o información adicional que se desee comunicar.

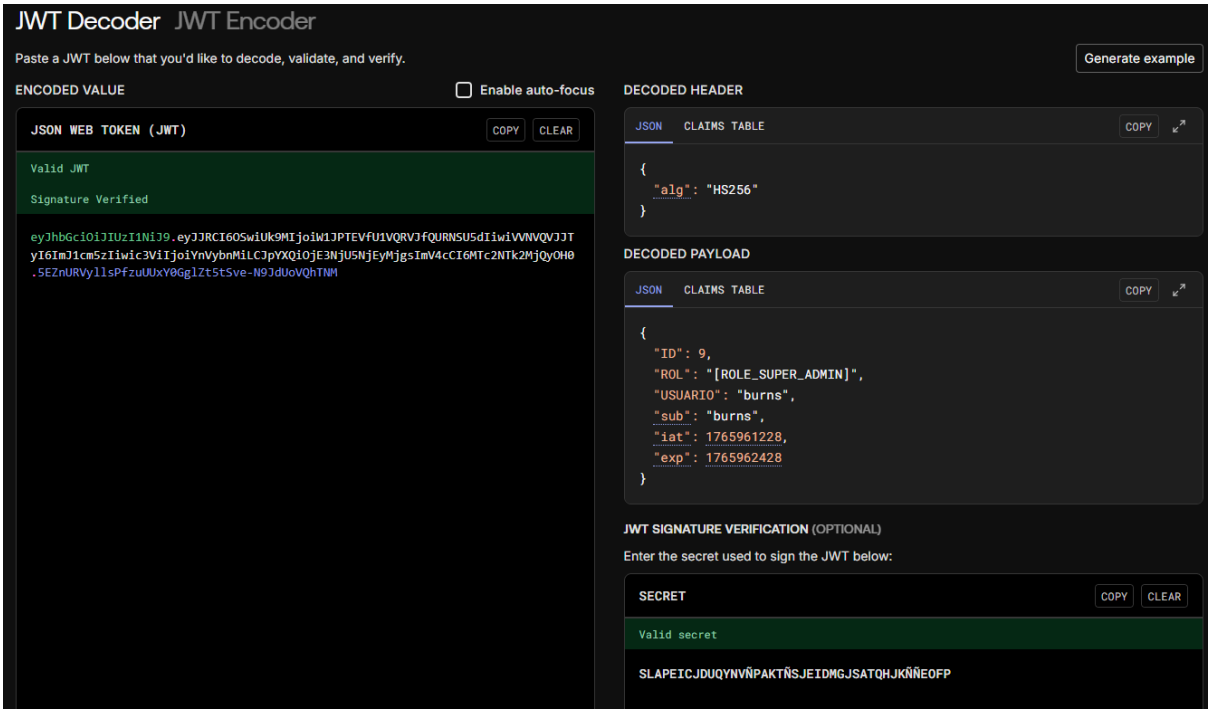
2. Detalles

- Anotada con @Data y @AllArgsConstructor de Lombok:
    - @Data: Genera getters, setters, métodos hashCode(), equals(), toString().
    - @AllArgsConstructor: Genera un constructor con todos los atributos.
- Contiene dos atributos principales:
    - jwt: un String que almacena el token JWT generado para el cliente.
    - mensajes: un Map<String, Object> opcional que puede contener mensajes adicionales, como notificaciones, roles u otra información relevante para el cliente.

3. Uso en la API

Después de que el cliente envía sus credenciales en un JWTRequest y éstas son validadas correctamente, el servidor devuelve un objeto JSON con la siguiente estructura:


```
{
  "ID": 9,
  "ROL": "[ROLE_SUPER_ADMIN]",
  "USUARIO": "burns",
  "sub": "burns",
  "iat": 1765961228,
  "exp": 1765962428
}
```



- El cliente debe almacenar este token (jwt) y enviarlo en el header Authorization en las siguientes peticiones para acceder a recursos protegidos.
- El campo mensajes es opcional y puede usarse para enviar información adicional útil para la interfaz de usuario o lógica cliente.

4. Importancia

Usar un DTO dedicado para la respuesta permite controlar qué datos se exponen al cliente y mantener la flexibilidad para agregar información extra sin alterar la estructura básica del token.

AuthController

1. Propósito del archivo

AuthController es el controlador REST encargado de manejar la autenticación de usuarios en la API mediante JWT. Proporciona un endpoint para que los clientes envíen sus credenciales y reciban un token JWT en caso de autenticación exitosa.

2. Dependencias

- JWTUserDetailsService: Servicio que carga los detalles del usuario (username, password, roles) desde la base de datos.
- JWTService: Servicio que genera y valida los tokens JWT.
- AuthenticationManager: Componente de Spring Security que autentica las credenciales del usuario.

3. Endpoint principal

POST /authenticate

Request

- Tipo: application/json

Cuerpo: un objeto JWTRequest que contiene:

```
 {
  "username": "usuario1",
  "password": "miPasswordSecreto"
}
```

- Proceso de autenticación

1. Autenticación de credenciales
   Se crea un UsernamePasswordAuthenticationToken con el username y password del cliente y se pasa al AuthenticationManager para verificar su validez.

2. Carga de detalles del usuario
   Se obtienen los detalles del usuario (UserDetails) usando JWTUserDetailsService.

3. Generación del token JWT
   Se genera un token JWT mediante JWTService.getToken(userDetails).

4. Respuesta al cliente

    - Se devuelve un JWTResponse que incluye:

        - jwt: el token generado.

        - mensajes: un Map con información adicional (mensaje de éxito, status, roles opcionales, etc.).

    - Se añade el token también en el header Authorization con el prefijo Bearer (importante un espacio después del Bearer).

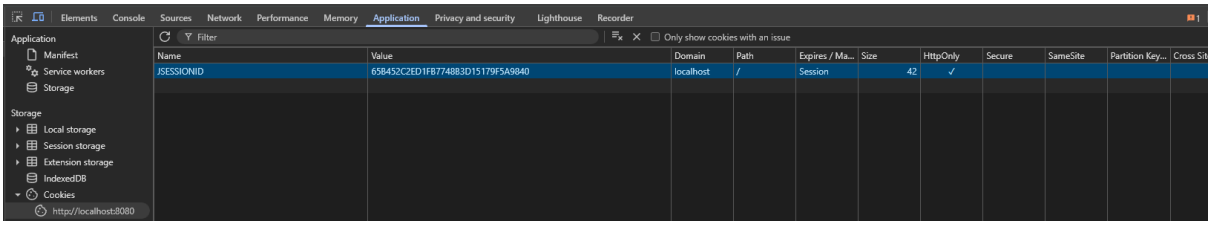
Response

- Tipo: application/json

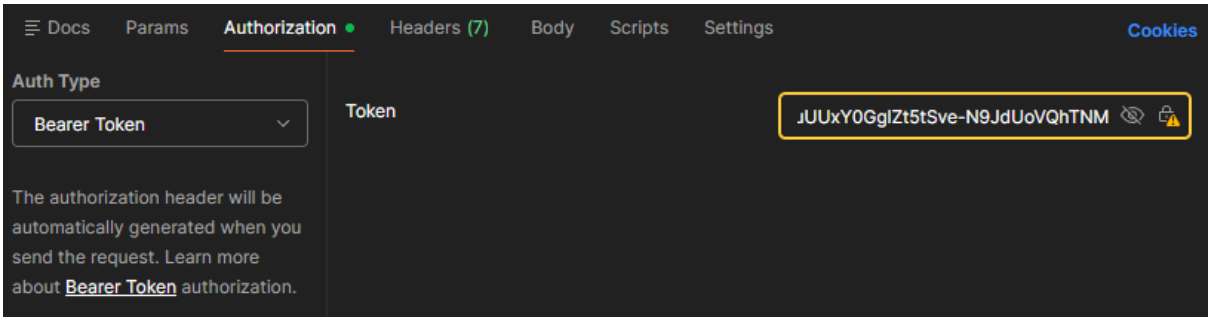
Ejemplo de respuesta exitosa:

En f12 -> Application -> LocalStorage ->token

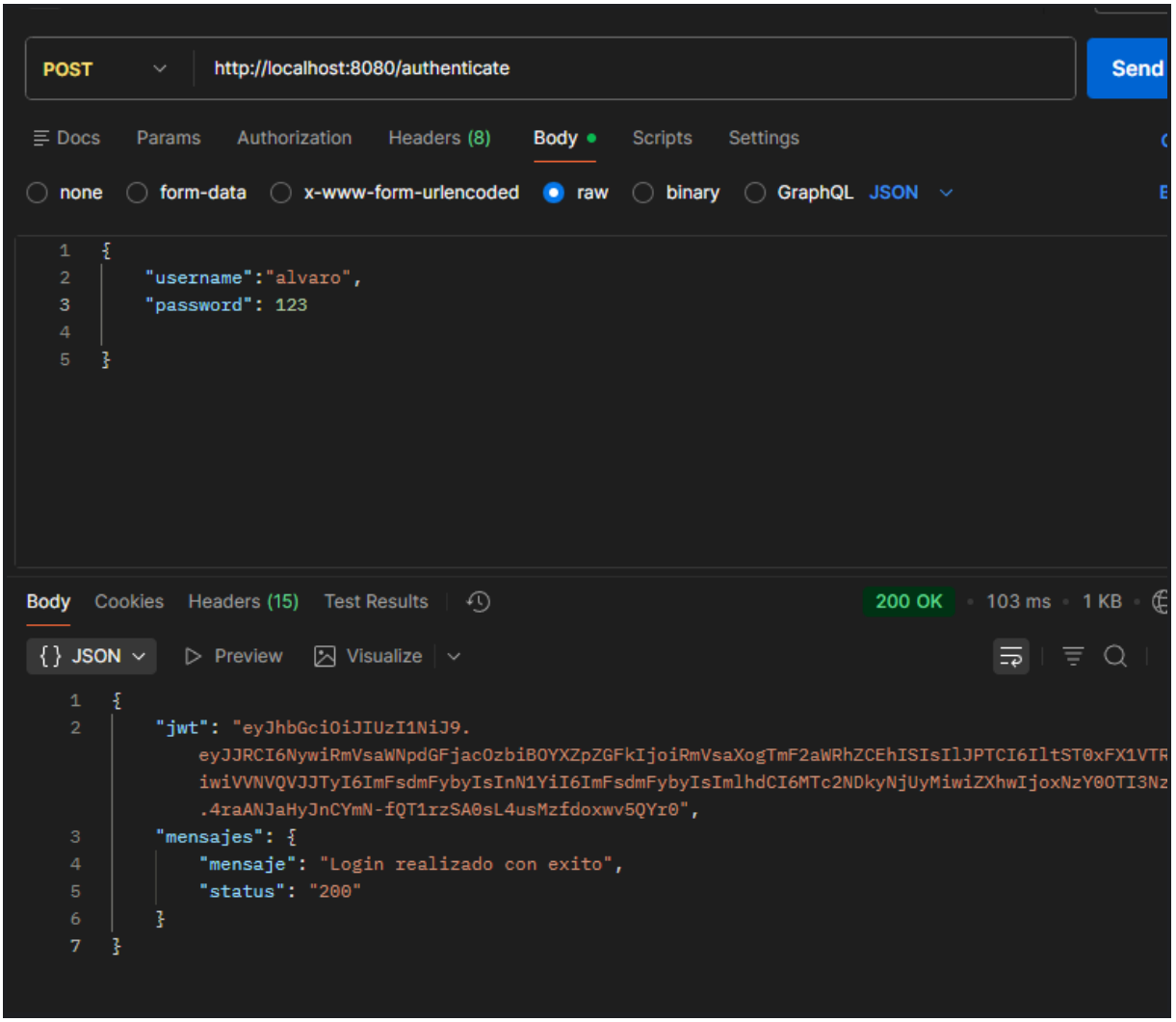
Podemos encontrar el token al haber hecho login en la app



Para hacer la prueba en postman, primero tenemos que hacernos con el token y añadirlo en authorization



Luego añadimos username y password he intentamos hacer login



Header:

Authorization: Bearer <token>

4. Manejo de errores

- Los errores de seguridad los maneja el propio spring (SpringSecurity)

5. Importante

- Centraliza la lógica de autenticación en un solo endpoint.
- Devuelve tanto el token como información útil para el cliente.
- Se integra con Spring Security mediante AuthenticationManager y filtros JWT posteriores.

JWTService

1. Propósito del archivo

JWTService es el servicio encargado de manejar toda la lógica relacionada con JWT en la aplicación. Incluye métodos para:

- Generar tokens.
- Validar tokens.
- Extraer información del token (claims).
- Comprobar su expiración.

Se integra con Spring Security y el flujo de autenticación de la API.

2. Constantes importantes

```
        public static final long JWT_TOKEN_VALIDITY = 1200; //20 minutos
        public static final String JWT_SECRET =
"SLAPEICJDUQYNVÑPAKTÑSJEIDMGJSATQHJKÑÑEOFP";
```

- **JWT_TOKEN_VALIDITY:** tiempo de vida del token en segundos.
- **JWT_SECRET:** clave secreta larga(40 letras) y compleja utilizada para firmar y validar los tokens JWT.

3. Dependencias

- UsuarioDAO: para obtener información del usuario desde la base de datos.
- UserDetails (de Spring Security): para obtener datos del usuario que serán usados en la generación o validación del token.

4. Funcionalidades principales

1. Extracción de información de un token

- getAllClaimsFromToken(String token): obtiene todos los claims (información interna) de un token JWT.
- getClaimsFromToken(String token, Function<Claims, T> claimsResolver): extrae un claim específico usando una función lambda.
- getTokenUsername(String token): obtiene el username del token.
- getTokenExpirationDate(String token): obtiene la fecha de expiración del token.

2. Validación del token

- isTokenExpirated(String token): comprueba si el token ha expirado.
- isTokenValid(String token, UserDetails userDetails): verifica si el token es válido comparando el username del token con el del UserDetails y comprobando que no esté expirado.

3. Generación de tokens

- generateToken(Map<String, Object> claims, String subject): crea internamente un token JWT.
    - claims: información opcional que se puede guardar en el token (roles, ID, nombre de usuario, etc.).
    - subject: generalmente el username.
    - Firma el token usando JWT_SECRET y lo convierte en un String.
- getToken(UserDetails userDetails): crea un token JWT para un usuario específico.
    - Extrae el usuario de la base de datos usando UsuarioDAO.
    - Crea los claims con información no confidencial, como ID, nombre y roles.
    - Llama a generateToken y devuelve el token generado.

5. Ejemplo de token

El token generado es un JWT estándar con tres partes:

header.payload.signature

- Header: información sobre el algoritmo de firma.
- Payload: información que decidimos guardar (claims, username, roles, ID…).
- Signature: firma criptográfica que garantiza la integridad del token.

eyJhbGciOiJIUzI1NiJ9.eyJJRCI6OSwiUk9MIjoiW1JPTEVfU1VQRVJfQURNSU5dIiwiVVNVQVJJTyI6ImJ1cm5zIiwic3ViIjoiYnVybnMiLCJpYXQiOjE3NjU5NjEyMjgsImV4cCI6MTc2NTk2MjQyOH0.5EZnURVyllsPfzuUUxY0GglZt5tSve-N9JdUoVQhTNM

{

"ID": 9,

"ROL": "[ROLE_SUPER_ADMIN]",

"USUARIO": "burns",

"sub": "burns",

"iat": 1765961228,

"exp": 1765962428

}

6. Consideraciones de seguridad

- Nunca incluir datos sensibles como contraseñas dentro del token.
- La clave JWT_SECRET debe ser larga y compleja (40 letras).
- El token tiene un tiempo de expiración corto (20 minutos) para mejorar la seguridad.
- Se recomienda regenerar el token tras la renovación de sesión o reautenticación.

JWTUserDetailsService

1. Propósito del archivo

JWTUserDetailsService implementa la interfaz de Spring Security UserDetailsService.
Su función principal es cargar los datos del usuario desde la base de datos para que Spring Security pueda autenticarlo y autorizarlo.

- Se utiliza en el flujo de autenticación con JWT.
- Permite que Spring Security conozca:
    - Username
    - Password
    - Roles o authorities del usuario

2. Dependencias

- UsuarioDAO: repositorio para acceder a la tabla de usuarios en la base de datos.
- UserDetails y User de Spring Security: representan internamente al usuario que se está autenticando.
- GrantedAuthority: representa los roles del usuario.

3. Método principal

```
@Override
	public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
```

- Recibe el username desde la request del cliente.
- Busca en la base de datos (usuarioDAO.findOneByNombre(username)).
- Si el usuario existe:

```
		return usuarioDAO.findOneByNombre(username)
				.map(u -> {
					List<GrantedAuthority> authorities = List.of(new SimpleGrantedAuthority(u.getRol()));
					return new User(
							u.getNombre(),
							 u.getPassword(),
							authorities
					 );
				}).orElseThrow(()-> new UsernameNotFoundException("Usuario no encontrado en la BBDD en el proceso de logueo"));
	}
```

        - Crea un listado de GrantedAuthority con el rol del usuario.
        - Devuelve un objeto User de Spring Security que implementa UserDetails:
            - username
            - password
            - authorities

- Si no existe:
    - Lanza UsernameNotFoundException.

4. Ejemplo de flujo

1. El cliente envía usuario y contraseña a /authenticate.
2. Spring Security llama a loadUserByUsername(username).
3. El método busca el usuario en la base de datos y devuelve un objeto UserDetails.
4. Spring Security valida la contraseña y roles usando el AuthenticationManager.

5. Notas adicionales

- Actualmente se asigna un único rol (u.getRol()) a cada usuario.

En aplicaciones más complejas, se puede usar muchos a muchos entre usuarios y roles:

```
 Set<Rol> roles = u.getRoles();

List<GrantedAuthority> authorities = roles.stream()

   .map(rol -> new SimpleGrantedAuthority(rol.getName()))

   .toList();
```

- La clase está anotada con @Service para que Spring la registre como Bean y pueda inyectarse en otros componentes.

JWTValidationFilter

1. Propósito del archivo

JWTValidationFilter es un filtro de Spring Security que valida todos los tokens JWT entrantes en cada petición HTTP.
Hereda de OncePerRequestFilter, por lo que se ejecuta una única vez por petición.

- Garantiza que solo usuarios autenticados con un token válido puedan acceder a los endpoints protegidos.
- Extrae el usuario del token y lo registra en el SecurityContext de Spring Security.

2. Dependencias

- JWTService: para leer, validar y extraer información del token.
- JWTUserDetailsService: para cargar los detalles del usuario desde la base de datos.
- SecurityContextHolder: contexto de seguridad de Spring, donde se registra la autenticación activa.
- AutorizacionException: se lanza cuando el token es inválido o no se puede procesar.

3. Constantes

```
public static final String AUTHORIZATION_HEADER = "Authorization";
public static final String AUTHORIZATION_HEADER_BEARER = "Bearer ";
```

- AUTHORIZATION_HEADER: nombre del header HTTP donde se envía el token.

- AUTHORIZATION_HEADER_BEARER: prefijo obligatorio para los tokens JWT (Bearer con espacio al final).

4. Método principal

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

Flujo de ejecución:

1. Obtener token JWT:
   Se lee el header Authorization y se verifica que comience con Bearer .

```
                // Obtenemos el header Authorization de la petición
                String requestTokenHeader = request.getHeader(AUTHORIZATION_HEADER);
                String userName = null;
                String jwt = null;

                // Validamos que el header no sea null y que comience con "Bearer "
                if(Objects.nonNull(requestTokenHeader) &&
requestTokenHeader.startsWith(AUTHORIZATION_HEADER_BEARER)) {
                        jwt = requestTokenHeader.substring(7); // Obtenemos el token sin el
prefijo "Bearer "
```

2. Extraer username del token:
   - Llama a jwtService.getTokenUsername(jwt).
   - Si falla, lanza AutorizacionException.

```
                        try {
                           userName = jwtService.getTokenUsername(jwt);
                        } catch (Exception e) {
                           throw new AutorizacionException("Error al intentar leer el usuario
del token");
                        }
```

3. Autenticación en Spring Security:
   - Solo se procede si userName es distinto de null y no existe autenticación previa.
   - Se cargan los detalles del usuario desde la base de datos (jwtUserDetailsService.loadUserByUsername).

- Se valida que el token sea válido (jwtService.isTokenValid(jwt, userDetails)).

```
                    if(Objects.nonNull(userName) &&
Objects.isNull(SecurityContextHolder.getContext().getAuthentication())) {
                        UserDetails userDetails =
jwtUserDetailsService.loadUserByUsername(userName);
                        if(jwtService.isTokenValid(jwt, userDetails)) {
```

4. Registrar autenticación en el SecurityContext:

    - Se crea un UsernamePasswordAuthenticationToken con los datos del usuario.
    - Se añaden detalles del request (IP, sessionId) al token.
    - Se establece el token en SecurityContextHolder.

```
                        UsernamePasswordAuthenticationToken
authenticationToken =
                                new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
        // Asociamos detalles del request (IP, sessionId, etc.) al token de autenticación
                        authenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

            // Establecemos la autenticación en el contexto de seguridad de Spring
            SecurityContextHolder.getContext().setAuthentication(authenticationToken);

                    }
                }
            }
```

5. Continuar con la cadena de filtros:

    - filterChain.doFilter(request, response) garantiza que la petición siga su flujo.

```
            // Continuamos con la cadena de filtros
            filterChain.doFilter(request, response);
        }
```

5. Consideraciones

- Si el token no existe o no es válido, la petición sigue pero sin autenticación, lo que hará que Spring Security bloquee el acceso a endpoints protegidos.
- OncePerRequestFilter asegura que este filtro solo se ejecute una vez por petición, evitando problemas de duplicación.
- Este filtro se debe registrar en la configuración de Spring Security (SecurityConfig) para que forme parte de la cadena de filtros de autenticación.

JWTAuthenticationEntryPoint

1. Propósito del archivo

JWTAuthenticationEntryPoint es un componente de Spring Security que gestiona las solicitudes no autenticadas a endpoints protegidos en una API REST.

- Se activa cuando un usuario intenta acceder a un recurso protegido sin un token válido.
- Devuelve un HTTP 401 Unauthorized con un mensaje explicativo.
- No redirige a páginas de login porque estamos en una API REST, no en una aplicación web frontend.

2. Implementación

```
@Component
public class JWTAuthenticationEntryPoint implements AuthenticationEntryPoint{
```

- Anotada con @Component para que Spring la registre como un bean y pueda ser utilizada en la configuración de seguridad.

- Implementa AuthenticationEntryPoint, que es la interfaz que Spring Security utiliza para interceptar accesos no autorizados.

3. Método principal

```
    @Override
    public void commence(
                    HttpServletRequest request,
                    HttpServletResponse response,
                    AuthenticationException authException) throws IOException,
ServletException {
```

Flujo:

1. Se invoca automáticamente cuando:
    - Un usuario no autenticado intenta acceder a una URI protegida.
    - No se proporciona un token válido en la petición (caso JWT).
2. Se envía una respuesta HTTP 401 Unauthorized:

```
            response.sendError(
                        HttpServletResponse.SC_UNAUTHORIZED,
                        "Petición no autorizada. Usuario no autenticado."
                        );
    }
```

3. Esto interrumpe la ejecución normal de la cadena de filtros y evita que el usuario acceda al recurso.

4. Consideraciones

- Es el primer punto de control para accesos no autenticados.
- Para aplicaciones front-end (como Angular), este punto devuelve información clara para que el cliente maneje la sesión o muestre un mensaje de login.
- No maneja validaciones de token: esa tarea la realiza JWTValidationFilter.

JWTAccessDeniedHandler

1. Propósito del archivo

JWTAccessDeniedHandler es un componente de Spring Security que gestiona las solicitudes de acceso a recursos que requieren permisos específicos.
Se activa cuando un usuario autenticado no tiene los permisos adecuados para acceder al recurso solicitado, devolviendo una respuesta HTTP 403 Forbidden.

2. Implementación:

```
@Component
public class JWTAccessDeniedHandler implements AccessDeniedHandler {
```

- **@Component:** Esta clase está anotada con @Component, lo que la registra como un bean en el contexto de la aplicación, permitiendo que Spring la utilice en la configuración de seguridad.

- **Implementación de AccessDeniedHandler:** Implementa la interfaz AccessDeniedHandler de Spring Security, que maneja situaciones donde un usuario autenticado intenta acceder a un recurso para el que no tiene permisos.

3. Método Principal:

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

Flujo:

- **Invocación:** Este método se invoca cuando un usuario autenticado, aunque correctamente identificado, no tiene los permisos suficientes para acceder a un recurso.

- **Acción:** En caso de que los permisos sean insuficientes, se devuelve una respuesta HTTP 403 Forbidden con el siguiente mensaje:

```
response.sendError(HttpServletResponse.SC_FORBIDDEN, "Petición no autorizada. No tienes
permisos para utilizar el recurso");
```

Esto indica que el usuario no está autorizado para acceder al recurso, a pesar de estar autenticado.

4. Resumen General:

1. Autenticación:
    - JWTAuthenticationEntryPoint maneja las solicitudes no autenticadas, devolviendo un error 401 Unauthorized si no se proporciona un token válido en la solicitud.

2. Autorización:
    - Una vez autenticado el usuario, si no tiene permisos suficientes, el JWTAccessDeniedHandler devuelve un error 403 Forbidden.

3. Flujo de seguridad:
    - Primero se comprueba si el usuario está autenticado.
    - Luego se verifica si tiene los permisos necesarios para acceder al recurso. Si no los tiene, se devuelve un error 403. Si no está autenticado, se devuelve un error 401.

SecurityConfig

1. Propósito del archivo

SecurityConfig centraliza la configuración de Spring Security en la aplicación:

- Define qué endpoints están protegidos y cuáles son públicos.
- Configura JWT, CORS y CSRF.
- Gestiona la política de sesiones (stateless).
- Proporciona PasswordEncoder y AuthenticationManager para autenticación.

Esta clase reemplaza la configuración por XML y permite personalizar la seguridad de manera programática.

2. Anotación principal

@Configuration

- Indica que la clase contiene beans de configuración de Spring.
- Spring detecta esta clase y registra los beans declarados con @Bean.

3. Método: securityFilterChain

```
        @Bean
        SecurityFilterChain securityFilterChain(HttpSecurity http, JWTValidationFilter
jwtValidationFilter)throws Exception {
```

Flujo de configuración:

Política de sesiones:

```
                http.sessionManagement(s ->
s.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
```

- Las APIs REST no usan sesiones; cada petición debe autenticarse con un token JWT.

Autorización de endpoints:

```
                http.authorizeHttpRequests(auth -> {
```

- Se definen reglas de acceso basadas en roles (SUPER_ADMIN, ADMIN, USER).

Ejemplo:

```
                    auth.requestMatchers(HttpMethod.POST,"/api/inmueble")
                    .hasAnyRole("SUPER_ADMIN","ADMIN");
```

```
                    auth.anyRequest().permitAll(); // Todo lo no restringido se permite
```

- **El orden de las reglas importa, las más específicas primero.**

4. Gestión de Errores:

```
            http.exceptionHandling(ex -> ex
                        .authenticationEntryPoint(authenticationEntryPoint)//401
                        .accessDeniedHandler(accessDeniedHandler) //403
                        );
```

Gestionamos los errores de seguridad que se puedan dar para que los gestione SpringSecurity

5. Configuraciones

Filtro JWT:

```
http.addFilterAfter(jwtValidationFilter, BasicAuthenticationFilter.class);
```

- Valida el token JWT en cada petición.
- Se ejecuta después del filtro de autenticación básica.

CORS:

```
http.cors(cors -> cors.configurationSource(corsConfigurationSource()));
```

- Permite solicitudes desde orígenes distintos (front-end Angular u otros clientes).
- Configuración flexible mediante CorsConfigurationSource.

Desactivación CSRF:

```
http.csrf( csrf -> csrf.disable());
```

- CSRF no es necesario en APIs REST que usan JWT.

6. Beans auxiliares

PasswordEncoder

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

- Necesario para codificar y validar contraseñas en la base de datos.
- Solo puede existir un PasswordEncoder en la aplicación.

AuthenticationManager

```
	@Bean
	AuthenticationManager authenticationManager(AuthenticationConfiguration configuration)throws Exception{

		return configuration.getAuthenticationManager();
	}
```

- Bean que gestiona la autenticación usando UserDetailsService y PasswordEncoder.

CORS Configuration Source

```
	@Bean
	CorsConfigurationSource corsConfigurationSource() {
		CorsConfiguration config = new CorsConfiguration();
		config.setAllowedOrigins(List.of("*")); //Permitimos todos los orígenes
		config.setAllowedMethods(List.of("*")); //Permitimos todos los métodos de http: POST, GET, PUT, DELETE...
		config.setAllowedHeaders(List.of("*")); //Permitimos todos los headers (es muy importante porque en los headers llegan los tokens)

		UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

		source.registerCorsConfiguration("/**", config);

		return source;
	}
```

- Permite solicitudes desde cualquier origen y método HTTP ().
- Configura headers permitidos para que los tokens JWT puedan enviarse correctamente.

Resumen de flujo de seguridad

Spring aplica las reglas de acceso definidas en securityFilterChain según el rol del usuario.

1. Primero se explica el flujo básico de autenticación: qué pide el cliente (JWTRequest), qué recibe (JWTResponse), y qué controlador maneja esto (AuthController).

2. Luego, el servicio que genera y valida el token (JWTService).

3. Después, cómo se cargan los usuarios (JWTUserDetailsService) para validar credenciales.

4. Sigue el filtro que procesa todas las peticiones validando el token (JWTValidationFilter).

5. Luego, el punto global para manejar errores de acceso no autorizado (JWTAuthenticationEntryPoint).

6. Por último, la configuración global de seguridad que conecta todo y define las reglas (SecurityConfig).

Subida de archivos(Storage)

UtilidadesStorage

1. Propósito del archivo

UtilidadesStorage es una clase de utilidades estáticas que centraliza la lógica relacionada con:

- Detección real del tipo MIME de archivos subidos.
- Normalización de tipos MIME detectados por Apache Tika.
- Validación de formatos permitidos.
- Obtención de la extensión de archivo a partir de su tipo MIME.

Su objetivo principal es reforzar la seguridad y coherencia en la subida de archivos, evitando confiar únicamente en la extensión proporcionada por el cliente.

2. Dependencias

- **Apache Tika (Tika, MimeTypes, MimeType)**
  Librería utilizada para detectar el tipo MIME real del contenido del archivo.

- **MultipartFile**
  Representa el archivo subido por el cliente.

3. Métodos principales

```
compruebaConcordanciaTipoMIME(MultipartFile file, List<String> tiposPermitidos) {
```

Propósito

Comprueba que el archivo recibido:

- Tiene un tipo MIME real válido.
- Coincide con alguno de los tipos permitidos.
- No ha sido manipulado (por ejemplo, extensión falsa).

Proceso

1. **Obtención del nombre original del archivo**
    - Se extrae getOriginalFilename() y se normaliza a minúsculas.
    - Si ocurre un NullPointerException, se lanza MultipartTratamientoException.
2. **Detección del tipo MIME real**
    - Se utiliza Apache Tika para analizar el contenido del archivo.
    - Si falla la detección, se lanza TratamientoTipoMimeException.
3. **Normalización del tipo MIME**

    - El tipo detectado se pasa al método normalizaMIMEParaTIKA.
    - Se eliminan anotaciones adicionales (por ejemplo: charset=UTF-8).
    - Se corrigen tipos genéricos o ambiguos detectados por Tika.

4. **Validación contra tipos permitidos**

    - Se comprueba si el tipo normalizado está dentro de la lista tiposPermitidos.
    - Si no coincide, se lanza FormatoNoSoportadoException.

Valor devuelto

- Devuelve el tipo MIME real y normalizado del archivo (por ejemplo, image/jpeg).

`DevuelveExtensionTipoMIME(String tipo)`

Propósito

Obtiene la extensión de archivo real a partir de un tipo MIME válido.

Proceso

1. Se obtienen todos los tipos MIME registrados mediante MimeTypes.
2. Se busca el tipo MIME correspondiente.
3. Se devuelve la extensión asociada (por ejemplo, .jpg).

Manejo de errores

- Si el tipo MIME no es válido o no puede resolverse, se lanza TratamientoTipoMimeException.

Valor devuelto

- Extensión del archivo incluyendo el punto (.jpg, .png, etc.).

`normalizaMIMEParaTIKA(String tipoDetectado, String filename)`

Propósito

Normaliza tipos MIME que Apache Tika detecta de forma genérica o ambigua, para hacerlos compatibles con validaciones posteriores.

Casos tratados

- Eliminación de información adicional (; charset=...).

- Corrección de tipos genéricos application/x-tika-ooxml según la extensión:

    - **.docx** → application/vnd.openxmlformats-officedocument.wordprocessingml.document
    - **.xlsx** → application/vnd.openxmlformats-officedocument.spreadsheetml.sheet
    - **.pptx** → application/vnd.openxmlformats-officedocument.presentationml.presentation

- Normalización de archivos .rar detectados como:

    - application/x-rar-compressed
    - application/vnd.rar
    - application/octet-stream

Valor devuelto

- Devuelve el tipo MIME corregido y normalizado.

4. Manejo de errores

- Todos los errores se encapsulan en excepciones personalizadas (ApiExceptionHandler).
- Permiten un tratamiento centralizado mediante @RestControllerAdvice.
- Evitan exponer errores internos o mensajes ambiguos al cliente.

5. Importante

- Nunca se confía únicamente en la extensión del archivo.
- El tipo MIME se obtiene analizando el contenido real.
- Se evita la subida de archivos manipulados o maliciosos.
- La clase es reutilizable para cualquier tipo de subida (imágenes, documentos, etc.).

- Todo el control de formatos está centralizado en un único punto.

StorageService

1. Propósito del archivo

StorageService es el servicio encargado de gestionar el almacenamiento de archivos multimedia (imágenes de inmuebles) en la API.
Se responsabiliza de:

- Validar archivos subidos por el cliente.
- Almacenar físicamente los archivos en el sistema de ficheros del servidor.
- Registrar y mantener la consistencia de dichos archivos con la base de datos.
- Proporcionar acceso a los recursos almacenados mediante URI HTTP.
- Eliminar archivos tanto de la base de datos como del sistema de archivos.

2. Dependencias

- ImagenInmuebleDAO
  Acceso a la tabla de imágenes asociadas a inmuebles en la base de datos.
- InmuebleDAO
  Permite validar la existencia de un inmueble antes de asociarle imágenes.
- HttpServletRequest
  Utilizado para construir URIs completas basadas en el host actual.
- @Value("${media.location}")
  Define el directorio raíz donde se almacenan los archivos físicos (por ejemplo, mediafiles).
- UtilidadesStorage
  Clase auxiliar para validaciones relacionadas con tipos MIME y extensiones de archivo.

3. Inicialización del servicio

Método init()

- Anotado con @PostConstruct, se ejecuta automáticamente al instanciar el servicio.
- Inicializa la ruta raíz de almacenamiento (rootLocation) usando la propiedad media.location.
- Verifica si el directorio existe y lo crea en caso contrario.
- Si ocurre un error en este proceso, lanza una ErrorInternoServidorException.

4. Métodos principales

store(MultipartFile file, Long idInmueble, String descripcion)

Propósito

Gestiona todo el proceso de subida de una imagen, asegurando consistencia entre sistema de archivos y base de datos.

Proceso general

El método se divide en tres fases claramente diferenciadas:

FASE 0: Validaciones previas

1. Validación de archivo vacío
    - Si el MultipartFile está vacío, se lanza una MultipartException.
2. Validación del tipo MIME
    - Solo se permiten imágenes image/jpeg y image/png.
    - Se valida la concordancia real entre contenido y tipo MIME.
    - Se obtiene la extensión real del archivo a partir del MIME.
3. Validación del inmueble
    - Se comprueba que el inmueble con idInmueble exista en la base de datos.
    - Si no existe, se lanza RecursoNoEncontradoException.
    - Esta validación se realiza antes de subir el archivo físico para evitar archivos huérfanos.

FASE 1: Subida física del archivo

4. Generación del nombre del archivo

    - Se sustituye el nombre original por un timestamp único (System.currentTimeMillis).
    - Se añade la extensión correspondiente (.jpg o .png).

5. Construcción de la ruta completa

    - Se resuelve la ruta final combinando rootLocation y el nombre generado.

6. Almacenamiento físico

    - Se copia el archivo usando Files.copy.
    - Se utiliza StandardCopyOption.REPLACE_EXISTING.
    - Se emplea un try-with-resources para asegurar el cierre del InputStream.
    - Cualquier error lanza una SubidaArchivoException.

FASE 2: Registro en la base de datos

7. Creación del registro ImagenInmueble

    - Se guarda el nombre del archivo.
    - Se asocia al inmueble validado.
    - Se almacena la descripción (alt de la imagen).

8. Persistencia en BBDD

    - Si el guardado falla:
        - Se elimina el archivo físico previamente creado.
        - Se lanza una SubidaArchivoException.
    - Se garantiza así la consistencia entre BBDD y sistema de archivos.

Valor devuelto

- Devuelve el nombre del archivo generado si todo el proceso es exitoso.

`getContentType(Resource resource)`

Propósito

Obtiene el tipo MIME real de un recurso almacenado.

- Usa Files.probeContentType.
- Si no se puede determinar, devuelve application/octet-stream.
- En caso de error, lanza PeticionMalFormadaException.

`getResource(String filename)`

Propósito

Obtiene un recurso físico como objeto Resource.

- Resuelve la ruta del archivo a partir de rootLocation.
- Devuelve un UrlResource.
- Verifica que el archivo exista y sea legible.
- Maneja errores de rutas mal formadas o inexistentes mediante PeticionMalFormadaException.

`getURICompletaFile(String filename)`

Propósito

Genera la URI HTTP completa para acceder públicamente a un archivo.

Proceso

- Obtiene el host dinámicamente desde HttpServletRequest.
- Construye la URI con ServletUriComponentsBuilder.
- Devuelve una ruta del tipo:

http://localhost:8080/media/imagen/{filename}

Ideal para servir imágenes directamente al cliente.

`deleteFileFisico(String filename)`

Propósito

Elimina un archivo físico del sistema de archivos.

- Resuelve y normaliza la ruta del archivo.
- Verifica que exista antes de eliminarlo.
- Si la eliminación falla, lanza BorradoArchivoException.

`deleteFileCompleto(Long idImagen)`

Propósito

Elimina completamente una imagen, tanto de la base de datos como del sistema de archivos.

Proceso

1. Obtiene la entidad ImagenInmueble por ID.
2. Elimina el registro de la base de datos.
3. Elimina el archivo físico asociado.
4. Maneja errores tanto de BBDD como de sistema de archivos con BorradoArchivoException.

5. Manejo de errores

- Todas las excepciones lanzadas son excepciones personalizadas.
- Son gestionadas centralizadamente mediante @RestControllerAdvice.
- Permiten devolver respuestas HTTP claras y coherentes al cliente.

6. Importante

- Se valida siempre la existencia del inmueble antes de subir archivos.
- Se evita la creación de archivos huérfanos en el sistema.
- Se mantiene consistencia total entre base de datos y almacenamiento físico.
- El servicio está desacoplado del controlador y puede reutilizarse fácilmente.

- Soporta extensiones y validaciones estrictas de seguridad (MIME real).

Frontend Angular - Lado del cliente

Frontend (Angular) - Repositorio Git

https://github.com/alvaromt6/ElPisitoV9.git

Estructura del proyecto Frontend (Angular)

Árbol de carpetas

```
src/
└── app/
    ├── core/
    │   ├── environments/
    │   ├── guards/
    │   ├── interceptors/
    │   ├── models/
    │   └── services/
    ├── pages/
    │   ├── admin/
    │   ├── auth/
    │   └── content/
    └── shared/
        ├── components/
        ├── directives/
        └── pipes/
```

Explicación de la arquitectura

La aplicación sigue una arquitectura modular, separando responsabilidades para mejorar la mantenibilidad, reutilización de código y escalabilidad.

core/

Contiene la lógica central y transversal de la aplicación.

- **environments/**
  Configuración de variables de entorno (URLs de la API, configuración según entorno).

- **guards/**
  Guards de Angular para proteger rutas (por ejemplo, acceso solo a usuarios autenticados o administradores).

- **interceptors/**
  Interceptores HTTP para:
  - Añadir tokens JWT a las peticiones
  - Manejar errores globales
  - Centralizar la comunicación con la API

- **models/**
  Interfaces y modelos de datos usados en toda la aplicación (Inmueble, Usuario, Operación, etc.).

- **services/**
  Servicios responsables de la comunicación con la API REST y la lógica de negocio.

pages/

Contiene las vistas principales de la aplicación, organizadas por funcionalidad.

pages/admin/

Zona de administración del sistema (CRUD completo).

Incluye páginas para:

- Alta, edición y listado de:
  - Inmuebles
  - Inmobiliarias
  - Operaciones
  - Provincias
  - Poblaciones
  - Tipos
  - Usuarios

Cada carpeta representa una pantalla independiente (add / edit / list).

Protegida por los guards.

pages/auth/

- **login/**
  Pantalla de autenticación de usuarios.

pages/content/

Zona pública de la aplicación, accesible para cualquier usuario.

Incluye páginas como:

- Home
- Detalle de inmueble
- Detalle de inmobiliaria
- Buscador (finder)

- Favoritos
- Registro de usuario
- Contacto
- Mapa web
- Páginas informativas

Representa el flujo normal de navegación del usuario final.

shared/

Contiene elementos reutilizables en toda la aplicación.

shared/components/

Componentes visuales reutilizables como:

- Header y footer
- Carruseles
- Fichas de inmuebles
- Buscador (finder)
- Menús
- Preloader
- Botones específicos de administración

Estos componentes se usan en múltiples páginas.

shared/directives/

Directivas personalizadas para modificar el comportamiento del DOM.

shared/pipes/

Pipes personalizados para transformación de datos en las vistas.

Esta estructura permite separar claramente la lógica de negocio (core), las vistas principales (pages) y los elementos reutilizables (shared), siguiendo buenas prácticas recomendadas en aplicaciones Angular de tamaño medio-grande.

Archivos Interesantes del FrontEnd

src/app/core/environments/globals.ts

Centralizar estas variables permite:

- Evitar duplicar URLs en distintos servicios
- Facilitar cambios de entorno (desarrollo, producción)
- Mejorar el mantenimiento del código

```
export const URL_BASE: string = "http://localhost:8080/";
```

```
export const URL_API: string = URL_BASE + "api/";

export const URL_MEDIA: string = URL_BASE + "media/";

export const URL_IMAGEN_INMUEBLE: string = URL_MEDIA + "imagen/";
```

src/app/core/guards/super.guard.ts

Este guard se encarga de proteger las rutas exclusivas del super administrador, permitiendo el acceso únicamente a usuarios autenticados que posean el rol **ROLE_SUPER_ADMIN**.

Se implementa mediante el tipo **CanActivateFn**.

```
export const superGuard: CanActivateFn = (route, state) => {

  const _router:Router = inject(Router);

  const _authService:AuthService = inject(AuthService);

  const _comunicationService:ComunicacionService =
inject(ComunicacionService);

  if (_authService.estaLogueado() && _authService.getRolFromToken() ==
'[ROLE_SUPER_ADMIN]') {

    return true;

  } else {

    _comunicationService.cambioLogin(false);

    _router.navigate(['/auth/login']);

    return false;

  }

};
```

Funcionamiento

1. Se inyectan los servicios necesarios:

2. El guard verifica:
    - Que el usuario esté logueado.

- Que el rol extraído del token sea **ROLE_SUPER_ADMIN**.

3. Si ambas condiciones se cumplen, se permite el acceso a la ruta.

4. En caso contrario:

    - Se actualiza el estado de login.
    - Se redirige al usuario a la pantalla de inicio de sesión.
    - Se bloquea el acceso a la ruta.

src/app/core/interceptors/token.interceptor.ts

Este interceptor se encarga de añadir automáticamente el token JWT a todas las peticiones HTTP salientes de la aplicación.

Se implementa como un interceptor funcional **(HttpInterceptorFn)**.

```
export const tokenInterceptor: HttpInterceptorFn = (req, next) => {

  const _authService = inject(AuthService);
```

```
  const token = _authService.getTokenFromLocalStorage();

  if(token){

    const cloneReq = req.clone({

      setHeaders: {

        Authorization: `Bearer ${token}`

      }

    });

    return next(cloneReq);

  }

  return next(req);

};
```

Funcionamiento

1. El interceptor se ejecuta automáticamente en todas las peticiones HTTP realizadas por la aplicación.

2. Obtiene el token JWT almacenado en el navegador mediante AuthService.

3. Si el token existe:
    - Se clona la petición original (los objetos HttpRequest son inmutables).
    - Se añade el encabezado Authorization con el formato Bearer <token>.

4. La petición modificada continúa su flujo normal.

5. Si no existe token, la petición se envía sin modificaciones.

src/app/core/services/auth-service.ts

Este servicio se encarga de gestionar la autenticación de usuarios en la aplicación frontend, utilizando tokens JWT para controlar el acceso y la sesión del usuario.

Centraliza toda la lógica relacionada con:

- Inicio y cierre de sesión
- Gestión del token en localStorage
- Decodificación del token JWT
- Obtención de información del usuario (rol, id, nombre)

- Verificación de la validez y expiración del token

```
//Obtiene el nombre de usuario desde el token.

getUsuarioFromToken(): string {

  this.token = this.getTokenFromLocalStorage();

  if (this.token) {

    try {

      const decoded: CustomJwtPayload = jwtDecode(this.token);

      return decoded.USUARIO;

    } catch (error) {

      return 'Desconocido';

    }

  }

  return 'El usuario no está logueado';

}
```

Funcionamiento

1. Obtención del token

this.token = this.getTokenFromLocalStorage();

Se recupera el token JWT almacenado en localStorage.

2. Comprobación de existencia

if (this.token) {

Si no existe token, el método asume que el usuario no está autenticado.

3. Decodificación del token

const decoded: CustomJwtPayload = jwtDecode(this.token);

Se decodifica el token JWT para acceder a su payload, donde se encuentra la información del usuario.

4. Extracción del nombre de usuario

return decoded.USUARIO;

Se devuelve el nombre de usuario incluido en el token.

5. Control de errores

```
catch (error) {
  return 'Desconocido';
}
```

Si el token es inválido, está corrupto o ha sido manipulado, se devuelve un valor por defecto.

6. Caso sin sesión iniciada

return 'El usuario no está logueado';

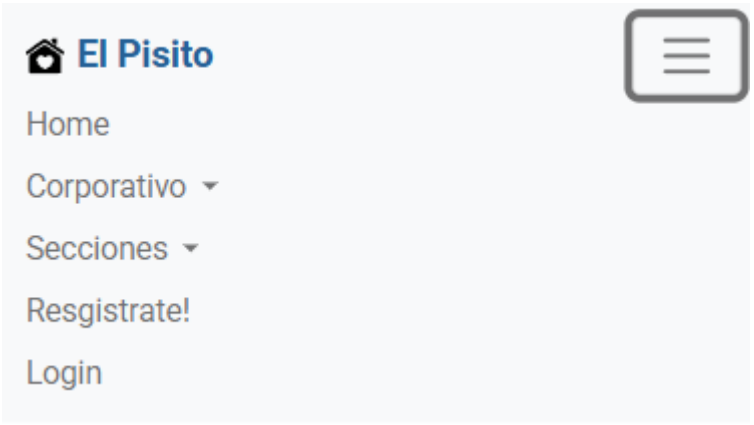
Si no hay token almacenado, el método indica que no existe sesión activa.

Capturas de pantalla de la Aplicación

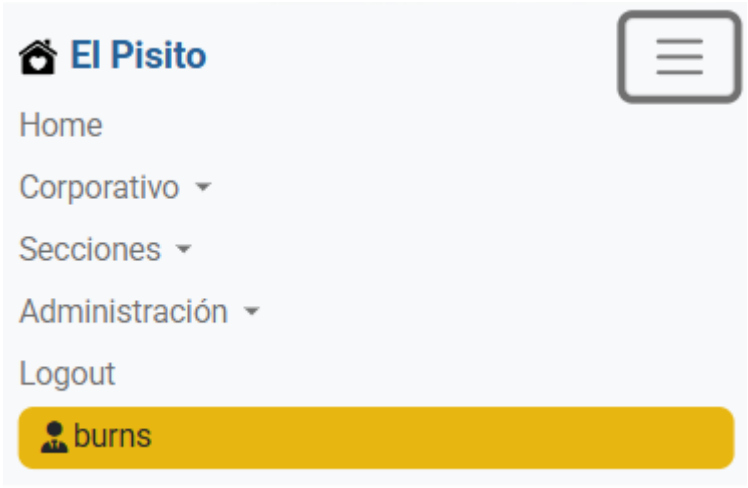
Página Home de la aplicación:

http://localhost:4200/home
http://localhost:4200

Menú desplegable para usuario que NO ha iniciado sesión:

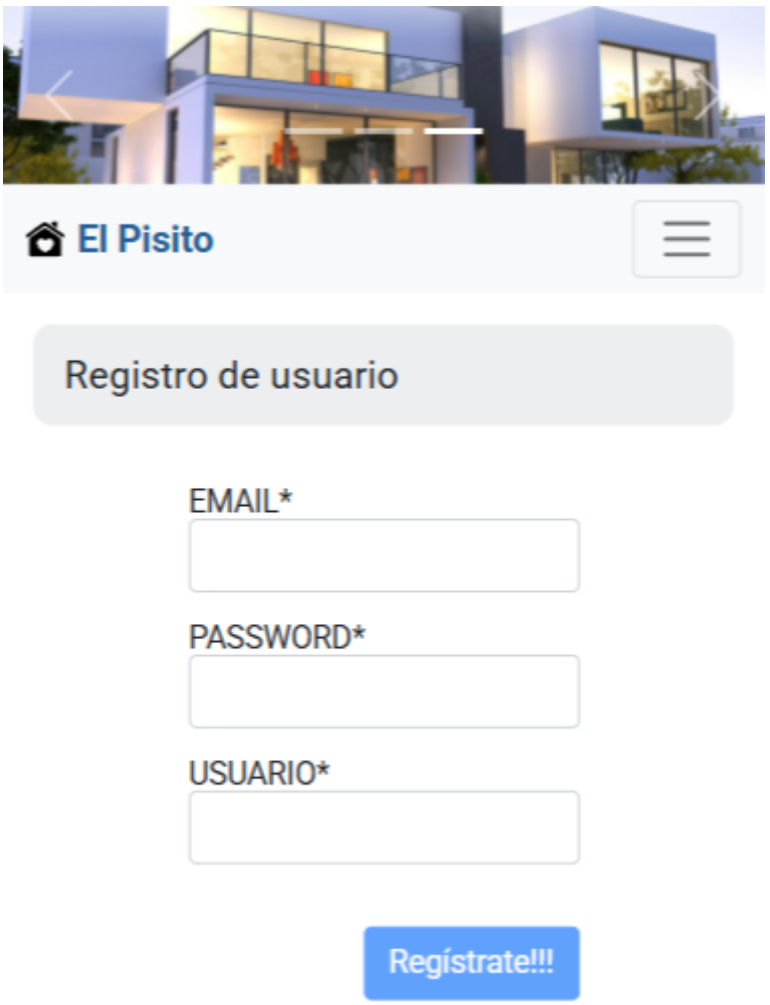


Menú desplegable para usuario que ha iniciado sesión:



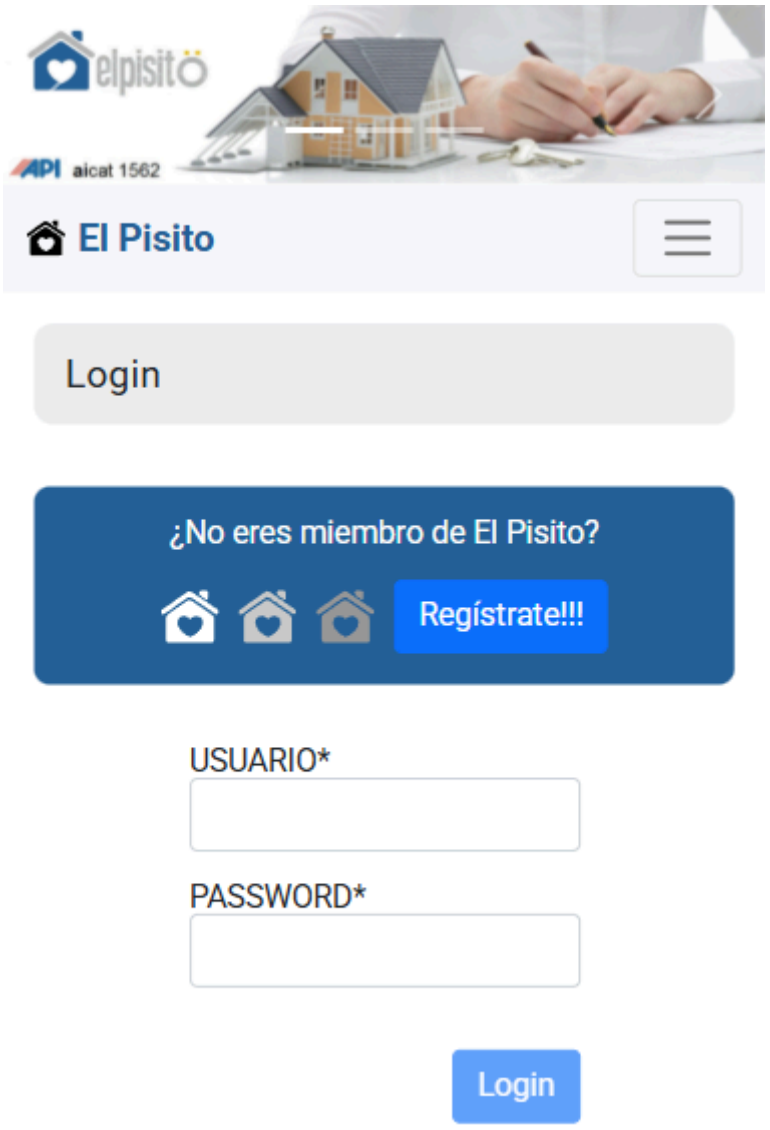
Formulario de registro para los usuarios:

http://localhost:4200/registro-usuario



Formulario de login:

http://localhost:4200/auth/login



Error de inicio de sesión al introducir credenciales erróneas:

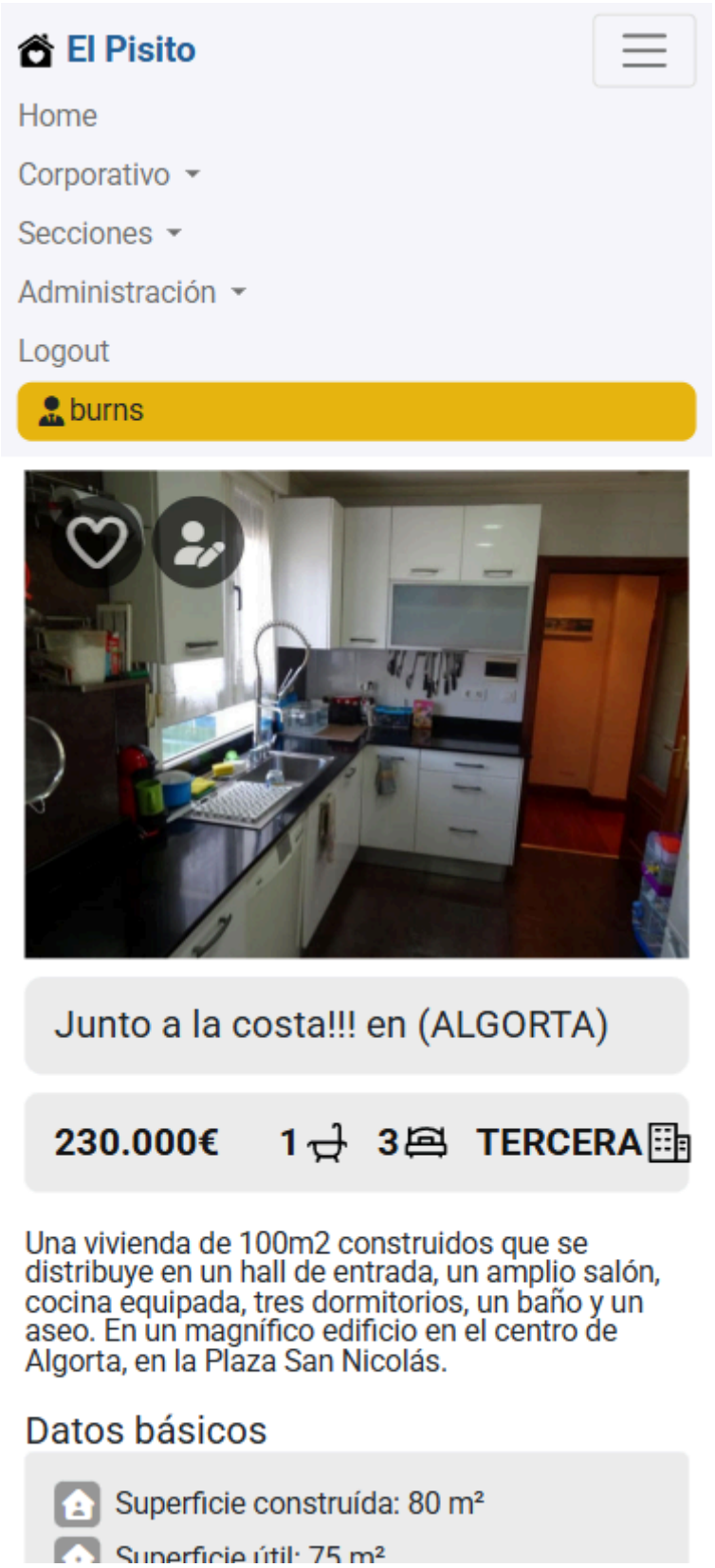
http://localhost:4200/error

Vista de un inmueble con un usuario [ROLE_USER]:

http://localhost:4200/detail-inmueble/2

Vista de un inmueble con un SuperAdmin [ROLE_SUPER_ADMIN]:

http://localhost:4200/detail-inmueble/3


El Pisito
Home
Corporativo
Secciones
Administración
Logout
burns
Junto a la costa!!! en (ALGORTA)
230.000€
1
3
TERCERA
Una vivienda de 100m2 construidos que se distribuye en un hall de entrada, un amplio salón, cocina equipada, tres dormitorios, un baño y un aseo. En un magnífico edificio en el centro de Algorta, en la Plaza San Nicolás.
Datos básicos
Superficie construída: 80 m²

Conclusión:

Este proyecto desarrollado con **Spring Boot** y **Angular** me ha permitido poner en práctica de manera integral los conceptos de **desarrollo full-stack** y arquitectura **API REST**.

Se han alcanzado los objetivos planteados:

- Implementación de un **backend robusto** con Spring Boot y Hibernate, siguiendo el patrón **MVC** y gestionando la persistencia de datos en MySQL.
- Desarrollo de un **frontend dinámico** con Angular, consumiendo la API REST y mostrando información en tiempo real al usuario.
- Gestión segura de la **autenticación y autorización** mediante JWT, interceptores e guards, asegurando que solo usuarios con los permisos adecuados accedan a determinadas rutas.
- Centralización de la lógica de negocio y reutilización de componentes, servicios y directivas, siguiendo buenas prácticas de desarrollo.

Además, el proyecto ha permitido consolidar competencias clave:

- **Diseño y documentación de APIs REST**
- **Integración entre frontend y backend**
- **Manejo de autenticación y control de acceso**
- **Uso de herramientas modernas** de desarrollo web (Postman, Angular CLI, Spring Boot DevTools)

En resumen, este proyecto no solo cumple su función como inmobiliaria funcional, sino que también constituye un **ejemplo completo de aplicación full-stack**, escalable y mantenible, demostrando la comprensión de buenas prácticas y patrones de arquitectura modernos.