



**Trabajo de Fin de Grado**

# **Tienda Tattoo's**

**Autoría:**

Jorge Campos Rodríguez

**Tutoría:**

Ignacio Lerga Paz

## Manual técnico de la web

Explicación del proyecto.....	3
Puesta a punto: Configuración.....	4
Application.yaml.....	4
spring.datasource:.....	4
spring.jpa.hibernate:.....	4
spring.springdoc:.....	5
spring.logging:.....	5
spring.jwt:.....	5
Desarrollo de la API.....	5
Gestión de dependencias: pom.xml.....	6
Spring Boot Starters:.....	6
Io.jsonwebtoken:.....	6
Documentación OpenApi:.....	6
Conectores y persistencia para la base de datos:.....	6
Utilidades:.....	6
Tests:.....	7
Swagger-codegen-maven-plugin:.....	7
Paquetes generados:.....	7
Base de datos.....	7
Tablas.....	7
Modelo relacional.....	8
Diagrama entidad-relación.....	8
Configuración del proyecto: Backend.....	9
Repositories (usando JPA):.....	9
Services: .....	9
Controllers:.....	10
Entities: .....	10
Mappers (usando MapStruct): .....	11
Config:.....	11
Errors:.....	12
Security:.....	12
Ejemplos de configuración.....	13
SecurityConfig:.....	13
Repositories:.....	14
Services:.....	15
Controllers.....	16
Ejemplo de Api autogenerada:.....	17
Entities.....	17
Ejemplo de modelo autogenerado.....	18
Mappers.....	18
Errors.....	19
Security.....	19
Test Junit / mockito.....	20
Configuración del proyecto: Frontend.....	21
Páginas HTML.....	21
JavaScript.....	22

CSS3.....	25
Conclusiones y posibles mejoras.....	26

## Explicación del proyecto

Este proyecto es un microservicio basado en API RESTful para gestionar los endpoints de una tienda especializada en productos para tatuajes. La aplicación permite administrar el catálogo, gestionar el carrito de compras y procesar pedidos, comunicándose con el frontend mediante peticiones HTTP y autenticación basada en JWT. En el lado del cliente, se utiliza JavaScript para cargar y renderizar dinámicamente la información en páginas HTML.

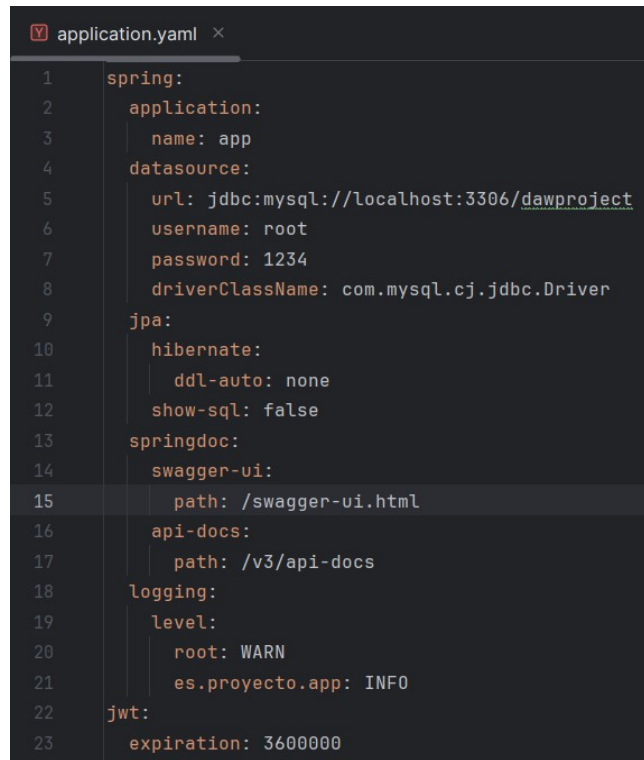
El backend se ha orientado en su mayor parte en lenguaje Java, aunque podría haberse desarrollado perfectamente en PHP. También se eligió Json Web Tokens por encima de Cookies dando que las API RESTful tengo entendido que no tienen estado (*'stateless'*) y según tengo entendido, podría encajar mejor en este tipo de arquitectura, aunque podría ser posible que ambas tecnologías pudiesen coexistir.

Se ha desarrollado todo el proyecto de manera monolítica (todo en el mismo proyecto), dado que es un proyecto pequeño de ejemplo e irreal, donde no se contempla (a priori) demasiada escalabilidad, más allá de un enfoque meramente evolutivo para añadir nuevas funcionalidades o para refactorizar código existente.

## Puesta a punto: Configuración.

### *Application.yaml*

Este archivo define la configuración principal de la aplicación Spring Boot. A continuación se detalla cada bloque:

A screenshot of a code editor showing the content of the application.yaml file. The file is named 'application.yaml' and is located in a directory. The content is as follows:

```
1  spring:
2    application:
3      name: app
4    datasource:
5      url: jdbc:mysql://localhost:3306/dawproject
6      username: root
7      password: 1234
8      driverClassName: com.mysql.cj.jdbc.Driver
9    jpa:
10     hibernate:
11       ddl-auto: none
12       show-sql: false
13     springdoc:
14       swagger-ui:
15         path: /swagger-ui.html
16       api-docs:
17         path: /v3/api-docs
18     logging:
19       level:
20         root: WARN
21         es.proyecto.app: INFO
22   jwt:
23     expiration: 3600000
```

#### **spring.datasource:**

- **url:** Dirección de conexión a la base de datos MySQL. En este caso, se conecta a la base de datos dawproject en localhost (puerto 3306).
- **username / password:** Credenciales del usuario para acceder a la base de datos.
- **driverClassName:** Especifica el driver JDBC para MySQL.

#### **spring.jpa.hibernate:**

- **ddl-auto: none:** Desactiva la creación automática del esquema de base de datos por Hibernate. Ideal para producción cuando ya existe la estructura.
- **show-sql: false:** Está determinado que las consultas SQL ya no aparezcan más en los logs. Antes, con true, servía para comprobar las consultas que se están realizando, útil para depurar.

**spring.springdoc:**

- **swagger-ui.path:** Ruta personalizada donde se puede acceder a la documentación interactiva Swagger.
- **api-docs.path:** Ruta del endpoint que devuelve la documentación OpenAPI en formato JSON.

**spring.logging:**

- **root: WARN:** El nivel de log general de la aplicación se establece en WARN para evitar ruido.
- **es.proyecto.app: INFO:** Se activa un nivel de log más detallado (INFO) específicamente para los paquetes propios del proyecto.

**spring.jwt:**

**expiration:** Tiempo de expiración de los tokens JWT en milisegundos. En este caso, equivale a **1 hora (3600000 ms)**.

**Desarrollo de la API.**

Se ha desarrollado un fichero 'api.yaml' que contiene los endpoints y schemas que se utilizarán en la aplicación.

También se configura un esquema de seguridad haciendo uso de bearerAuth para algunos endpoints críticos.

Spring Boot Proyecto DAW 1.0.0 OAS 3.0

/v3/api-docs

Spring Boot Rest API

Servers

http://localhost:8080 - Generated server url

Roles

Users

Subcategories

GET /Subcategory/{idSubcategory} Obtener una subcategoría por ID

PUT /Subcategory/{idSubcategory} Actualizar una subcategoría por ID

DELETE /Subcategory/{idSubcategory} Eliminar una subcategoría por ID

GET /Subcategory Obtener todas las subcategorías

POST /Subcategory Crear una nueva subcategoría

## ***Gestión de dependencias: pom.xml***

A continuación se enumeran las dependencias generales configuradas en el archivo 'pom.xml'.

### **Spring Boot Starters:**

- **spring-boot-starter-web**: Para la creación de aplicaciones web RESTful.
- **spring-boot-starter-security**: Para la gestión de seguridad y autenticación.
- **spring-boot-starter-data-jpa**: Para el acceso a datos usando JPA.
- **spring-boot-starter-validation**: Para validaciones automáticas con anotaciones (@Valid, etc.).

### **io.jsonwebtoken:**

- **jjwt-api, jjwt-impl, jjwt-jackson, y jjwt**: Para la creación, parseo y validación de JSON Web Tokens.

### **Documentación OpenApi:**

- **springdoc-openapi-starter-webmvc-ui**: Para exponer Swagger UI y OpenAPI en endpoints.
- **swagger-annotations**: Para anotar modelos y controladores con metadatos Swagger.

### **Conectores y persistencia para la base de datos:**

- **mysql-connector-j**: Driver JDBC para conectarse a MySQL.
- **jakarta.persistence-api**: API de persistencia compatible con JPA/Hibernate.

### **Utilidades:**

- **lombok**: Para reducir código repetitivo, hace innecesario el uso de getters, setters, constructores, toString, etc mediante etiquetas.
- **mapstruct y mapstruct-processor**: Para el mapeo automático entre objetos Java (entidades y modelos).
- **javax.validation y javax.annotation-api**: Soporte adicional para anotaciones y validaciones estándar.
- **jackson-datatype-threetenbp**: Es un soporte para tipos de fecha/hora que autogeneraba la API en primera instancia, luego se hizo la configuración a utilizar java.time.LocalDateTime (se mantiene por si acaso).
- **jakarta.servlet-api**: API necesaria para trabajar con servlets.

**Tests:**

- **spring-boot-starter-test:** Incluye JUnit, Mockito y otras herramientas para testing de Spring. Se hará uso de algunos test de ejemplo aunque no se llegará al 80% del coverage el cual suele ser el mínimo exigido por empresas reales.

**Swagger-codegen-maven-plugin:**

- **Función:** Genera interfaces y modelos Java a partir del archivo OpenAPI (api.yaml).
- **Directorio fuente:** src/main/resources/contract/api.yaml
- **Salida:** target/generated-sources/

**Paquetes generados:**

- **API:** es.swagger.codegen.api
- **Modelos:** es.swagger.codegen.models
- **Mapeo de fechas personalizado:** Se reemplaza DateTime por java.time.LocalDateTime.

**Base de datos.**

La base de datos está recogida en el fichero 'schema.sql' y los inserts están en 'data.sql'. Esta base de datos anteriormente se sobrescribía cada vez que se iniciase el microservicio habiéndole dado en application.yaml la configuración necesaria para ello.

Ahora mismo ya no es necesario sobrescribir la base de datos y es completamente persistente.

A continuación se enumeran las tablas de la base de datos.

**Tablas**

1. Roles
2. Users
3. Category
4. Subcategory
5. Products
6. Orders
7. Order\_Products (tabla intermedia)
8. Cart

## Modelo relacional.

Las tablas mantienen las siguientes relaciones:

**Users.role\_id → Roles.id\_role**

**Relación:** muchos usuarios pueden tener un mismo rol.

**Subcategory.id\_category → Category.id\_category**

**Relación:** muchas subcategorías por categoría.

**Products.id\_subcategory → Subcategory.id\_subcategory**

**Relación:** muchos productos por subcategoría.

**Orders.id\_user → Users.id\_user**

**Relación:** un usuario puede tener muchos pedidos.

**Order\_Products.id\_order → Orders.id\_order**

**Order\_Products.id\_product → Products.id\_product**

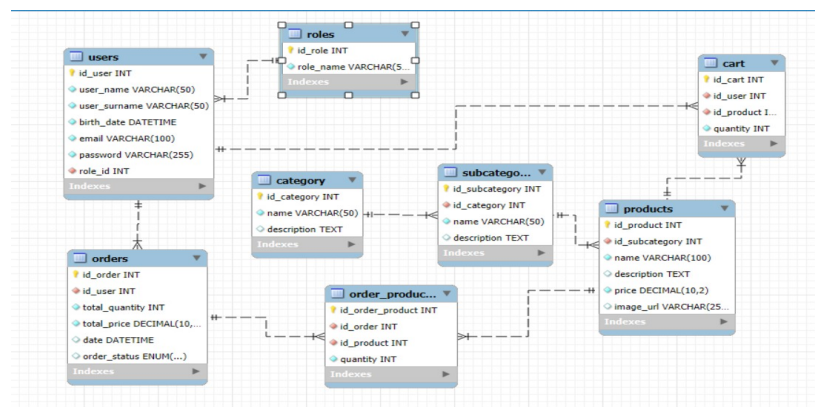
**Relación:** muchos productos en muchos pedidos (relación N:M).

**Cart.id\_user → Users.id\_user**

**Cart.id\_product → Products.id\_product**

**Relación:** un usuario puede tener varios productos en su carrito.

## Diagrama entidad-relación.



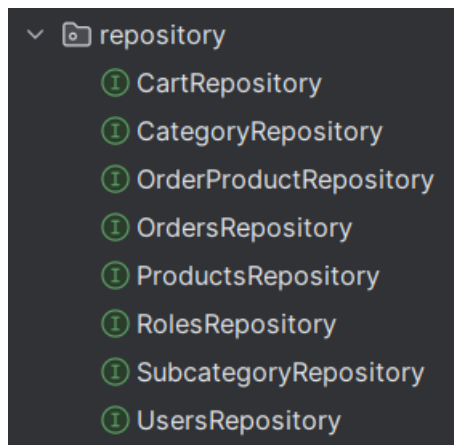


## Configuración del proyecto: Backend.

El backend se ha configurado para tener la estructura básica y funcional de un microservicio mínimo. Las carpetas y sus clases se despliegan de la siguiente manera:

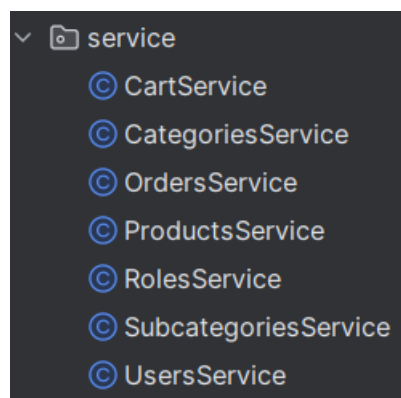
### Repositories (usando JPA):

Son la capa encargada de interactuar con la base de datos. Utilizando JPA (Java Persistence API), puedes realizar operaciones CRUD (crear, leer, actualizar, eliminar) de forma sencilla a través de interfaces que extienden JpaRepository.



### Services:

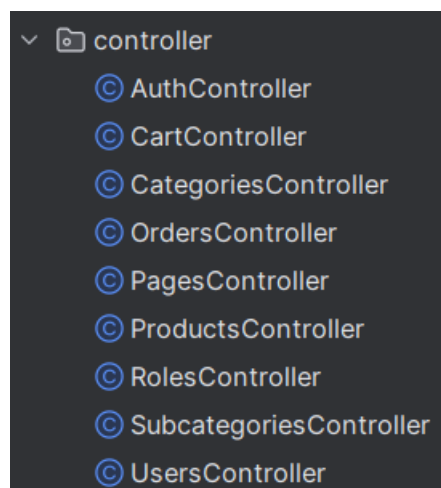
Son la capa de lógica de negocio. Estos services son responsables de mantener la comunicación/coordinación entre repositorios y controladores, asegurando que todo funcione correctamente.



## Controllers:

Son la capa que maneja las solicitudes HTTP (como GET, POST, PUT, DELETE) y decide cómo responder a ellas. Los controllers actúan como intermediarios entre los clientes (como un navegador web) y los servicios.

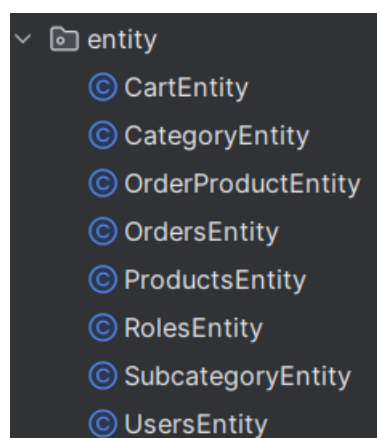
En este caso, implementarán y sobrescribirán los métodos de las interfaces autogeneradas por la compilación del proyecto gracias al plugin swagger-codegen descrito con anterioridad, recogerán la actividad de la aplicación mediante logs, controlará mensajes de error, etc.



## Entities:

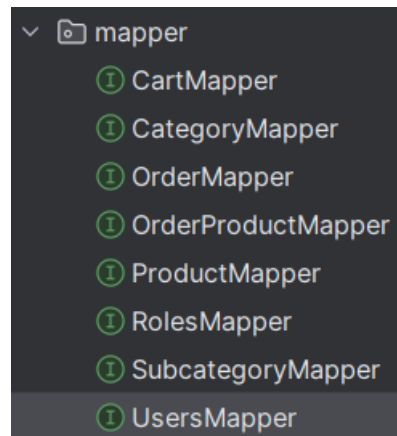
Son las representaciones de las tablas de la base de datos. Se definen como clases anotadas con `@Entity`, y cada objeto corresponde a un registro en una tabla.

Se gestionarán también las relaciones entre entidades con los `joinColumns` y las anotaciones `ManyToOne`, `OneToMany`, etc.



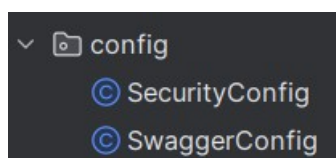
**Mappers (usando MapStruct):**

Se usan para convertir datos entre diferentes capas de la aplicación, como transformar entidades en DTOs (Data Transfer Objects) y viceversa. Se utiliza MapStruct, herramienta que genera automáticamente el código necesario para estas transformaciones basándose en interfaces.

**Config:**

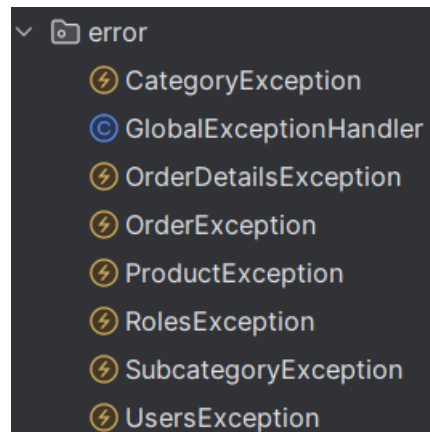
Se usa para configurar la seguridad en la aplicación, en este caso contiene clases especializadas en configuraciones para JWT.

- Integra un filtro personalizado (JwtAuthenticationFilter) que valida los tokens JWT antes de que se procese la autenticación por defecto de Spring Security.
- Define que las peticiones a la URL /cart requieren autenticación, mientras que las demás rutas se permiten sin restricciones.
- Desactiva la protección CSRF, lo cual es común en APIs REST donde se utiliza JWT
- Proporciona un bean de PasswordEncoder que utiliza BCrypt, un algoritmo seguro para el cifrado de contraseñas.

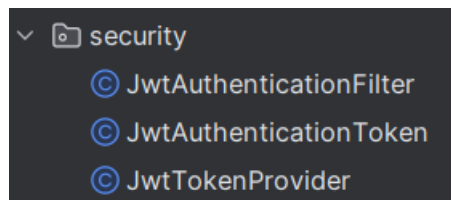


**Errors:**

Contiene errores personalizados que utilizará el controller. Estas clases definen y centralizan los errores específicos relacionados con operaciones.

**Security:**

Protege el acceso a la aplicación usando tokens JWT, permitiendo que solo los usuarios con un token válido puedan acceder a los recursos privados.



## Ejemplos de configuración.

A continuación se enumeran y explican algunos ejemplos de código de las clases contenidas en esta estructura de carpetas:

### **SecurityConfig:**

Contiene un bean que nos permite el cifrado de contraseñas y un método `securityFilterChain` el cual desactiva CSRF, un tipo de ataque que no es usual en aplicaciones que utilicen JWT (pero si sería imperativo hacerlo de usar Cookies).

Configura que el carrito de compra esté protegido por autenticación, lo que se traduce en que este carrito, personal para cada usuario, solo podrá ser visto en caso de que el usuario se haya logeado correctamente.

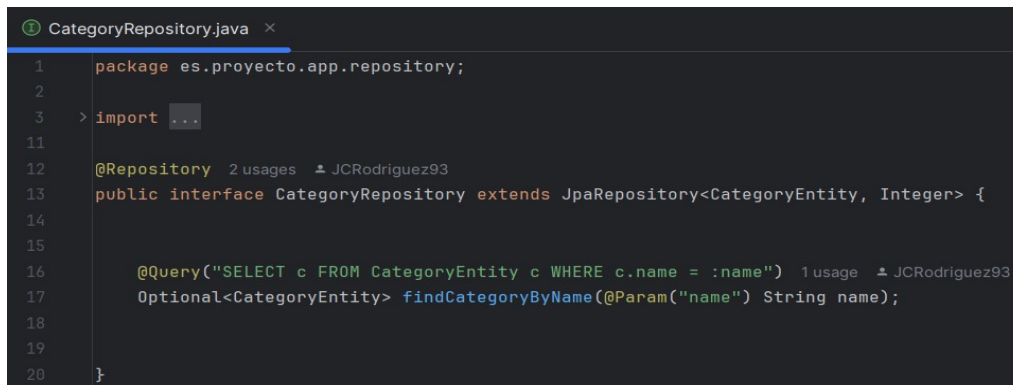
En aplicaciones que emplean autenticación basada en JWT, no se utiliza el mecanismo tradicional de sesiones y formularios. En su lugar, se interceptan las solicitudes mediante un filtro (en este caso, `JwtAuthenticationFilter`) que valida los tokens JWT. Esto significa que la autenticación se maneja mediante tokens enviados en cada petición. Desactivar el login por defecto evita presentar una página innecesaria y simplifica la seguridad.

```
17 public class SecurityConfig {
18
19     @Bean no usages JCRodriguez93
20     public PasswordEncoder passwordEncoder() {
21         return new BCryptPasswordEncoder(); // BCrypt para el cifrado de contraseñas
22     }
23
24     @Bean no usages JCRodriguez93
25     @Autowired
26     public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
27         http
28             .csrf( CsrfConfigurer<HttpSecurity> csrf -> csrf.disable()) // Desactiva CSRF
29             .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
30                 .requestMatchers( ...patterns: "/cart").authenticated() // Proteges la API
31                 .anyRequest().permitAll()
32             )
33             .formLogin( FormLoginConfigurer<HttpSecurity> form -> form.disable()) // Desactiva el login predeterminado
34             .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);
35         // Agregado el filtro JWT antes del filtro de autenticación por defecto
36
37         return http.build();
38     }
39 }
```

## ***Repositories:***

Son interfaces que extienden de `JpaRepository` como ya se ha comentado, las cuales le pasaríamos una entidad y un `Integer`.

Estas interfaces deben ser anotadas como `@Repository` y permiten la creación de `@Query` personalizadas en caso de ser necesarias.



```
1 package es.proyecto.app.repository;
2
3 > import ...
11
12 @Repository 2 usages  ⚡ JCRodriguez93
13 public interface CategoryRepository extends JpaRepository<CategoryEntity, Integer> {
14
15
16     @Query("SELECT c FROM CategoryEntity c WHERE c.name = :name") 1 usage  ⚡ JCRodriguez93
17     Optional<CategoryEntity> findCategoryByName(@Param("name") String name);
18
19
20 }
```

## Services:

Aquí un ejemplo de una clase Service. Se aprovecha la inyección de dependencias al anotar el objeto mapper y repository con `@Autowired`, simplificando el código y eliminando la necesidad de instanciar manualmente los objetos.

Posteriormente, se desarrollan métodos para el CRUD (crear, leer, actualizar y borrar) para que estas clases hagan de intermediarias entre la base de datos y los controllers.

```
CategoriesService.java x
19 @Validated 4 usages JCRodriguez93
20 @Transactional
21 @Service
22 public class CategoriesService {
23
24     @Autowired 4 usages
25     private CategoryMapper mapper; //sin el INSTANCE ya aparecen las subcategorias en vez de null.
26
27     @Autowired 8 usages
28     private CategoryRepository categoryRepository;
29
30     public List<Category> getAllCategories() { 1 usage JCRodriguez93
31         List<CategoryEntity> categoryEntities = categoryRepository.findAll();
32         return mapper.toApiDomain(categoryEntities); // MapStruct para convertir entidades a modelos
33     }
34
35
36     public void createCategory(Category idCategory) { 1 usage JCRodriguez93
37         CategoryEntity entity = mapper.toEntity(idCategory);
38         categoryRepository.save(entity);
39     }
40
41     public Category getCategoryById(Integer idCategory) { 7 usages JCRodriguez93
42         Optional<CategoryEntity> optionalCategoryEntity = categoryRepository.findById(idCategory);
43         return optionalCategoryEntity.map(mapper::toApiDomain).orElse(other: null);
44     }
45 > public boolean getCategoryByName(String name) { return categoryRepository.findCategoryByName(name).isPresent(); }
46
47
48
49     public void deleteCategory(Integer idCategory) { 1 usage JCRodriguez93
50         if (categoryRepository.existsById(idCategory)) {
51             categoryRepository.deleteById(idCategory);
52         }
53     }
54 }
```

## Controllers

En este apartado se puede comprobar una clase marcada como `@RestController` la cual implementa la interfaz que el plugin swagger generó de manera automática. Inyectaría la dependencia de un objeto service y sobrescribiría los métodos para las solicitudes HTTP.

```
UsersController.java x
19  @Slf4j  86 usages  JCRodriguez93
20  @RestController
21  public class UsersController implements UsersApi {
22
23      @Autowired  8 usages
24      private UsersService usersService;
25
26      private static final Logger logger = LoggerFactory.getLogger(UsersController.class);  20 usages
27
28      @Override  2 usages  JCRodriguez93
29  public ResponseEntity<UserCreatedResponse> createUser(User body) {
30      if (body == null) {
31          logger.error("Null body provided");
32          throw UsersException.NULL_BODY_EXCEPTION;
33      }
34
35      if (body.getUserName() == null || body.getUserName().isEmpty()) {
36          logger.error("Invalid user name when creating user");
37          throw UsersException.MISSING_USER_NAME_EXCEPTION;
38      }
39
40      if (usersService.existsByEmail(body.getEmail())) {
41          logger.error("Duplicate email in database");
42          return new ResponseEntity<>(HttpStatus.CONFLICT); // Retorna 409 Conflict
43      }
44
45      usersService.createUser(body);
46      logger.info("User created successfully: {}", body.getIdUser());
47
48      return new ResponseEntity<>(HttpStatus.CREATED);
49  }
```



## Ejemplo de Api autogenerada:

```
UsersApi.java x
Generated source files should not be edited. The changes will be lost when sources are regenerated.
1 > /.../
6 package es.swagger.codegen.api;
7
8 > import ...
38
39 @javax.annotation.Generated(value = "io.swagger.codegen.v3.generators.java.SpringCodegen", date = "2025-03-19T23:30:00.17
40 @Validated
41 public interface UsersApi {
42
43     @Operation(summary = "Crear un nuevo usuario", description = "Permite a cualquier persona crear una nueva cuenta de u
44     @SecurityRequirement(name = "bearerAuth")    }, tags={ "Users" })
45     @ApiResponse(value = {
46         @ApiResponse(responseCode = "201", description = "Usuario creado exitosamente", content = @Content(mediaType = "a
47
48         @ApiResponse(responseCode = "400", description = "Solicitud incorrecta (Bad Request)", content = @Content(mediaTy
49
50         @ApiResponse(responseCode = "500", description = "Error interno del servidor", content = @Content(mediaType = "ap
51     @RequestMapping(value = "/Users",
52         produces = { "application/json" },
53         consumes = { "application/json" },
54         method = RequestMethod.POST)
55     ResponseEntity<UserCreatedResponse> createUser(@Parameter(in = ParameterIn.DEFAULT, description = "Datos del nuevo us
56
```

## Entities

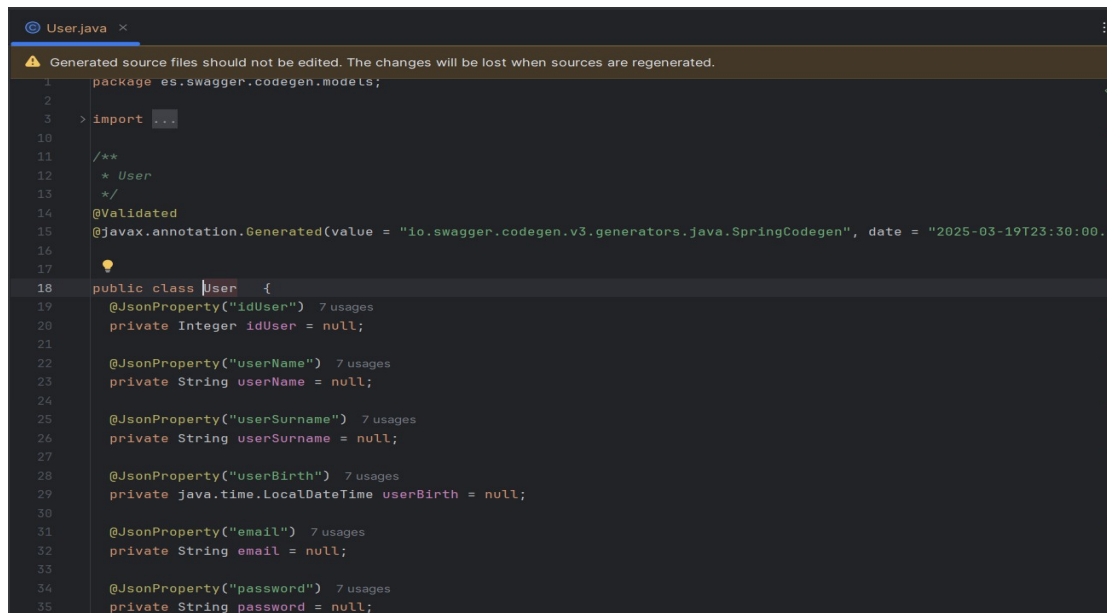
Podemos comprobar efectivamente que es una referencia en este caso a la tabla 'Users' de la base de datos y sus respectivas columnas.

Con la etiqueta `@JoinColumn` y el tipo de relación (`@ManyToOne` en este caso) debemos indicar la relación de clave foránea con otras tablas. A parte de eso, el atributo `@Id` indicaría la primary key de esta tabla.

```
UsersEntity.java x
17 @Table(name = "Users")
18 public class UsersEntity {
19
20     @Id
21     @GeneratedValue(strategy = GenerationType.IDENTITY)
22     @Column(name = "id_user")
23     private Integer idUser;
24
25     @Column(name = "user_name", nullable = false)
26     private String userName;
27
28     @Column(name = "user_surname", nullable = false)
29     private String userSurname;
30
31     @Column(name = "birth_date", nullable = false)
32     private LocalDateTime birthDate;
33
34     @Column(name = "email", unique = true, nullable = false)
35     private String email;
36
37     @Column(name = "password", nullable = false)
38     private String password;
39
40     @ManyToOne
41     @JoinColumn(name = "role_id", referencedColumnName = "id_role", nullable = false)
42     private RolesEntity roleId;
43
```

## Ejemplo de modelo autogenerado

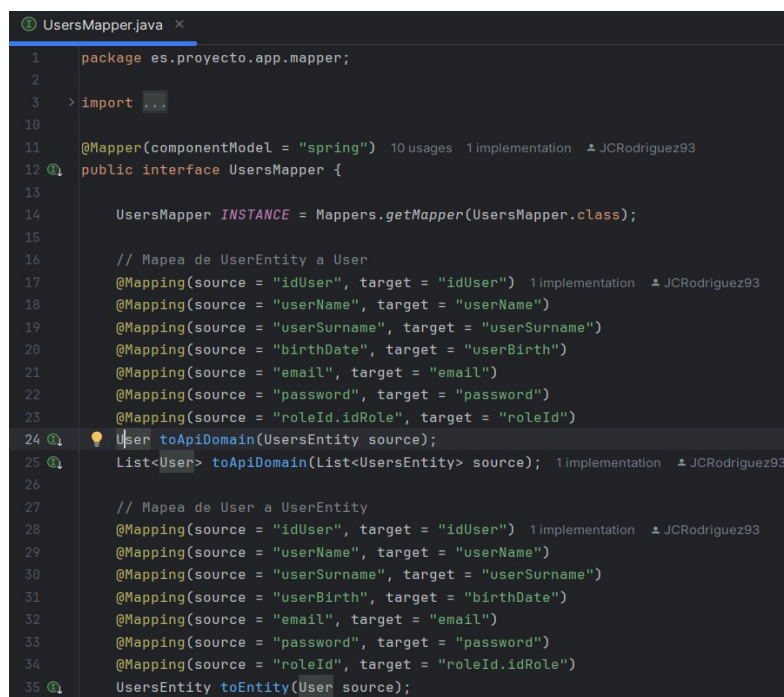
Este tipo de clases representarían las estructuras de los datos de la Api, en este caso el cuerpo de la solicitud y respuesta de los endpoints descritos en el fichero 'api.yaml' descrito con anterioridad.



```
1 package es.swagger.codegen.models;
2
3 > import ...
4
5
6
7
8
9
10
11 /**
12  * User
13  */
14 @Validated
15 @javax.annotation.Generated(value = "io.swagger.codegen.v3.generators.java.SpringCodegen", date = "2025-03-19T23:30:00.1")
16
17
18 public class User {
19     @JsonProperty("idUser") 7 usages
20     private Integer idUser = null;
21
22     @JsonProperty("userName") 7 usages
23     private String userName = null;
24
25     @JsonProperty("userSurname") 7 usages
26     private String userSurname = null;
27
28     @JsonProperty("userBirth") 7 usages
29     private java.time.LocalDateTime userBirth = null;
30
31     @JsonProperty("email") 7 usages
32     private String email = null;
33
34     @JsonProperty("password") 7 usages
35     private String password = null;
```

## Mappers

Los mappers permiten que los datos viajen entre la base de datos y el cliente de manera fluida, manteniendo la separación de responsabilidades entre las capas y asegurando que la estructura sea correcta en cada paso. Aquí se puede comprobar como se mapea, haciendo uso de mapstruct, los modelos a entidades y viceversa.



```
1 package es.proyecto.app.mapper;
2
3 > import ...
4
5
6
7
8
9
10
11 @Mapper(componentModel = "spring") 10 usages 1 implementation JCRodriguez93
12 public interface UsersMapper {
13
14     UsersMapper INSTANCE = Mappers.getMapper(UsersMapper.class);
15
16     // Mapea de UserEntity a User
17     @Mapping(source = "idUser", target = "idUser") 1 implementation JCRodriguez93
18     @Mapping(source = "userName", target = "userName")
19     @Mapping(source = "userSurname", target = "userSurname")
20     @Mapping(source = "birthDate", target = "userBirth")
21     @Mapping(source = "email", target = "email")
22     @Mapping(source = "password", target = "password")
23     @Mapping(source = "roleId.idRole", target = "roleId")
24     User toApiDomain(UserEntity source);
25     List<User> toApiDomain(List<UsersEntity> source); 1 implementation JCRodriguez93
26
27     // Mapea de User a UserEntity
28     @Mapping(source = "idUser", target = "idUser") 1 implementation JCRodriguez93
29     @Mapping(source = "userName", target = "userName")
30     @Mapping(source = "userSurname", target = "userSurname")
31     @Mapping(source = "userBirth", target = "birthDate")
32     @Mapping(source = "email", target = "email")
33     @Mapping(source = "password", target = "password")
34     @Mapping(source = "roleId", target = "roleId.idRole")
35     UsersEntity toEntity(User source);
```

## Errors

Este tipo de clases extienden de `RuntimeException` y son útiles en caso de querer personalizar los mensajes de error que se lanzarían en caso de que el código devolviese una excepción.

Se comprueba la creación de constantes que mostrarían un mensaje acorde a la excepción ocurrida.

```
ProductException.java x
1 package es.proyecto.app.error;
2
3 public class ProductException extends RuntimeException { 31 usages  JCRodriguez93
4
5     public static final ProductException NO_PRODUCT_FOUND_EXCEPTION = new ProductException("No product found with the s
6     public static final ProductException INVALID_PRODUCT_ID_EXCEPTION = new ProductException("Invalid product ID provid
7     public static final ProductException NULL_BODY_EXCEPTION = new ProductException("Null body provided for the product
8     public static final ProductException MISSING_PRODUCT_NAME_EXCEPTION = new ProductException("Product name is require
9     public static final ProductException MISSING_PRODUCT_ID_EXCEPTION = new ProductException("Product ID is required");
10    public static final ProductException ERROR_UPDATING_PRODUCT_EXCEPTION = new ProductException("Error updating produc
11
12    public ProductException(String message) { 8 usages  JCRodriguez93
13        super(message);
14    }
15 }
```

## Security

Esta clase crea, valida y extrae datos de tokens JWT (como el email o los roles). También puede invalidar tokens guardándolos en memoria (como si los “anulara”).

```
JwtAuthenticationFilter.java  JwtTokenProvider.java x
15 @Component 8 usages  JCRodriguez93
16 public class JwtTokenProvider {
17
18     private final SecretKey jwtSecret = Keys.secretKeyFor(SignatureAlgorithm.HS512); 4 usages
19
20     @Value("${jwt.expiration}") 1 usage
21     private long jwtExpiration;
22
23     // Simulamos un almacenamiento en memoria para tokens invalidados
24     private Set<String> invalidatedTokens = new HashSet<>(); 1 usage
25
26     public String generateToken(String email) { 2 usages  JCRodriguez93
27         return Jwts.builder()
28             .setSubject(email)
29             .setIssuedAt(new Date())
30             .setExpiration(new Date(new Date().getTime() + jwtExpiration))
31             .signWith(SignatureAlgorithm.HS512, jwtSecret)
32             .compact();
33     }
34
35     public boolean validateToken(String token) { 1 usage  JCRodriguez93
36         try {
37             Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(token);
38             return true;
39         } catch (JwtException | IllegalArgumentException e) {
40             return false;
41         }
42     }
43
44     public String getEmailFromToken(String token) { 1 usage  JCRodriguez93
45         return Jwts.parserBuilder().setSigningKey(jwtSecret).build().parse(token).getSubject();
46     }
```

Por otro lado , tenemos un filtro que se ejecuta con cada petición. Comprueba si hay un token en la cabecera, lo valida y, si es correcto, autentica al usuario en Spring usando el email y los roles que hay en el token.

```
@Component 4 usages  JCRodriguez93
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtTokenProvider jwtTokenProvider; 4 usages

    public JwtAuthenticationFilter(JwtTokenProvider jwtTokenProvider) { this.jwtTokenProvider = jwtTokenProvider; }

    @Override  JCRodriguez93
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) th

    private String getTokenFromRequest(HttpServletRequest request) {...}
}
```

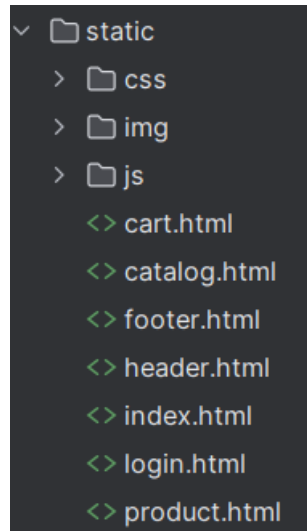
## Test Junit / mockito

Estas clases están destinadas al testing de las diversas respuestas del microservicio. Son indispensables antes de pasar a producción. Se utiliza Junit5 para definir y ejecutar los test mientras que con mockito lo que se hace es simular el comportamiento de los diversos servicios de la aplicación sin tener que hacer uso de datos reales.

```
UsersControllerTest.java x
27 public class UsersControllerTest {
51
52     //hecho
53     @Test  JCRodriguez93
54     @DisplayName("Crear usuario con datos validos")
55     public void createUserWithValidDataAndThenReturnCreated() {
56         // Configuramos el mock para que el correo del usuario NO exista
57         when(usersService.existsByEmail(user.getEmail())).thenReturn(value: false);
58
59         // Llamamos al método del controlador para crear el usuario
60         ResponseEntity<UserCreatedResponse> response = usersController.createUser(user);
61
62         // Verificamos que el status HTTP devuelto sea CREATED (201)
63         assertEquals(HttpStatus.CREATED, response.getStatusCode());
64         // Verificamos que el servicio de usuarios se llame con el usuario correcto
65         verify(usersService).createUser(user);
66     }
```

## Configuración del proyecto: Frontend.

En los recursos estáticos de la aplicación se situarán los archivos CSS, HTML5, JavaScript e imágenes de la aplicación. A continuación se muestran las páginas HTML5 del microservicio.



### Páginas HTML

Uso HTML como plantilla base y con JavaScript cargo todo dinámicamente desde mi microservicio. Así separo el frontend de la lógica de negocio, y todo se monta en tiempo real al abrir la página.

Estas páginas utilizan librerías como bootstrap para crear una interfaz responsive adaptable a los diversos tamaños de pantalla, además de sweet alert 2 para los mensajes pop-up de la interfaz.

```
<> product.html x
2  <html lang="es">
15
16  <body style="...">
17
18    <!-- FETCH EN JAVASCRIPT VALE PARA QUE EL MENÚ Y EL FOOTER ESTÉN VISIBLES EN LAS PÁGINAS
19
20    <header>
21      <a id="top"></a> <!-- ancla para redirigir el index-->
22
23      <div id="header-container"></div>
24
25      <!-- Heading -->
26
27    </header>
28
29
30
31
32    <!-- Contenedor para los detalles del producto CORREGIDO -->
33    <div id="product-details"></div>
34
```

## JavaScript

En uno de los ejemplos que se mostraría a continuación, se utilizan métodos `async`. Esto significa que se pueda ejecutar el código sin bloquear el resto del programa.

`URLSearchParams` nos permite capturar parámetros pasados en la URL, útil para mostrar contenido dinámico.

Por otro lado, `await` pausa la ejecución hasta que una promesa (en este caso, `fetch`) se resuelve o rechaza (en el siguiente pantallazo, buscamos un producto por su id). De este modo, podemos acceder a recursos externos como es este caso con la API que tarda tiempo y podemos preveer que se congele la pantalla.

```
/* ===== CARGA DE DETALLES DEL PRODUCTO (PRODUCT.HTML) ===== */
async function loadProductDetails() {
  const urlParams = new URLSearchParams(window.location.search);
  const idProduct = urlParams.get('id');

  const productDetails = document.getElementById('product-details');
  if (!productDetails) {
    console.error("El contenedor de detalles del producto no se encuentra en el DOM.");
    return;
  }
  try {
    const productResponse = await fetch(`http://localhost:8080/Products/${idProduct}`);
    if (!productResponse.ok) throw new Error("Error en la respuesta de la API del producto");
    const product = await productResponse.json();
  }
}
```



En cuanto a la manipulación del DOM, podemos destacar la utilización de innerHTML para modificar el contenido de un nodo e insertar el contenido de manera dinámica como en el siguiente ejemplo

```
try {
  const response = await fetch(`http://localhost:8080/Products?subcategory=${idSubcategory}`);
  if (!response.ok) {
    throw new Error("Error en la respuesta de la API");
  }

  let responseData = await response.json();
  // Si la respuesta tiene la propiedad 'products', extraemos ese array.
  let similarProducts = responseData.products ? responseData.products : responseData;

  if (!Array.isArray(similarProducts)) {
    similarItemsContainer.innerHTML = "<p>No se encontraron productos similares.</p>";
    return;
  }

  // Opcional: barajar el array (similar a lo que hice en ProductosDestacados)
  similarProducts = similarProducts.sort(() => 0.5 - Math.random());
  // Limitar a 4 productos
  const productsToShow = similarProducts.slice(0, 4);

  similarItemsContainer.innerHTML = "";
  productsToShow.forEach(product => {
    similarItemsContainer.innerHTML += `
      <div class="d-flex mb-3">
        <a href="product.html?id=${product.idProduct}" class="me-3">
          
        </a>
        <div class="info">
          <a href="product.html?id=${product.idProduct}" class="nav-link mb-1">${product.name}</a>
          <strong class="text-dark">${product.price}€</strong>
        </div>
      </div>
    `;
  });
}
```

También se haría necesario el uso de eventos como DOMContentLoaded que quedaría a la espera de que el DOM esté completamente cargado.

En el siguiente ejemplo, si se encuentra un elemento con id 'product-details' llamaría a la función loadProductDetails(); y si no, mandaría un warning por consola.

```
// escucha para esperar que el DOM esté cargado
document.addEventListener("DOMContentLoaded", () => {
  // Verificar si estamos en products.html antes de ejecutar
  if (document.getElementById("product-details")) {
    loadProductDetails(); // Llamar a la función cuando el DOM e
  } else {
    console.warn("No se encontró el contenedor 'product-details'");
  }
});
```

La función de todos los scripts, en terminos generales, sería hacer uso de los fetch sobre los endpoints del microservicio, dibujar dinámicamente contenido en los nodos HTML, obtener y enviar datos.

Otros scripts se han desarrollado como clases, y estos a su vez son extendidos y sus métodos sobrescritos para cada una de las secciones de la interfaz principal.

```
ProductSection.js x
1  /* ===== CLASE BASE PARA SECCIONES DE PRODUCTOS ===== */
2  class ProductSection {
3      constructor(apiUrl, containerSelector) {
4          this.apiUrl = apiUrl;
5          this.container = document.querySelector(containerSelector);
6          this.products = [];
7          if (!this.container) {
8              console.error(`El contenedor ${containerSelector} no se encuentra en el DOM.`);
9          }
10     }
11
12     async loadProducts() {
13         try {
14             const response = await fetch(this.apiUrl);
15             const data = await response.json();
16             this.products = data.products;
17             this.displayProducts();
18         } catch (error) {
19             console.error("Error al cargar productos:", error);
20         }
21     }
22
23     /* Método abstracto para mostrar productos: se sobrescribe en cada subclase */
24     displayProducts() {
25         console.error("displayProducts() debe implementarse en la subclase.");
26     }
27 }
```

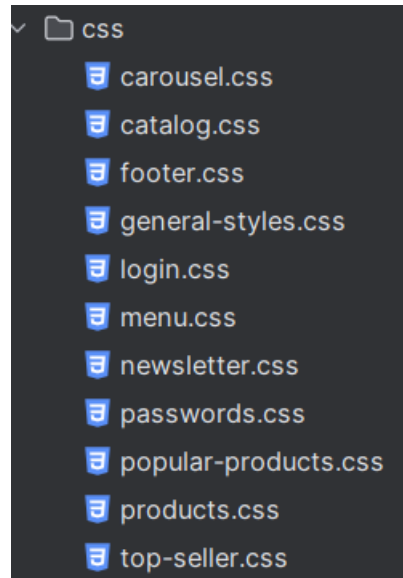
En este caso se puede comprobar como `displayProducts()` está siendo sobrescrito en una de las divisiones que tiene la interfaz principal, concretamente en `top vendidos`.

```
1  /* ===== TOP SELLERS ===== */
2  class TopSellers extends ProductSection {
3      constructor(apiUrl, containerSelector) {
4          super(apiUrl, containerSelector);
5      }
6
7      generateStars(rating) {
8          return Array(5).fill(0).map((_, i) =>
9              `<span class="star">${i < rating ? "★" : "☆"}</span>`
10             ).join('');
11     }
12
13     displayProducts() {
14         if (!this.container) return;
15         this.container.innerHTML = "";
16         const shuffledProducts = this.shuffleArray(this.products);
17         const selectedProducts = shuffledProducts.slice(0, 4);
18
19         selectedProducts.forEach(product => {
20             const productElement = document.createElement("div");
21             productElement.classList.add("product-item", "top-seller");
22             const randomRating = Math.floor(Math.random() * 5) + 1;
23             productElement.innerHTML = `
24                 
25                 <h3 class="product-name">${product.name}</h3>
26                 <div class="product-rating">${this.generateStars(randomRating)}</div>
27             `;
28             this.container.appendChild(productElement);
29         });
30     }
31 }
```



## CSS3

Quedan definidos ficheros de estilo, algunos generales y otros concretos para diversas regiones de la interfaz.



Estos estilos le darán un enfoque más personalizado a la interfaz según los requerimientos de esta lo demanden para crear una experiencia de usuario fluida y agradable.

```
general-styles.css x
1  /*
2   * -----
3   * ESTILOS GENERALES PARA LA WEB COMPLETA
4   * -----
5   */
6  .newsletter p,
7  .newsletter h2,
8  .products h2,
9  .top-sellers-description p,
10 .top-sellers-description h2,
11 .popular-section-title,
12 .popular-left-text,
13 .social-hashtag,
14 h1,
15 button{
16     color: white;
17 }
18
19 h1,
20 .products h2{
21     padding: 50px;
22     text-align: center;
23 }
24
25 #header {
26     transition: top 0.3s;
27     position: fixed;
28     width: 100%;
29     z-index: 1000;
30 }
31 .hide-logo img {
32     display: none;
```

## Conclusiones y posibles mejoras

En cuanto a tecnologías, si hubiese tenido más 'experiencia de campo' y tiempo, hubiese optado por añadir algunas como Angular, Vue.js, Tailwind, etc... haber creado configuraciones alternativas para bases de datos como por ejemplo postgresSQL, la utilización de variables de entorno por ejemplo en usuario y contraseña de la BBDD, mejoras en las respuestas de la API, añadir un control de IP's o algo que se suele ver mucho en los procesos de autenticación como lo es Oauth. Y sobretodo, haber dedicado tiempo a estudiar alternativas de IDE, ya que solamente este proyecto sería funcional en IntelliJ IDEA y daría problemas con la ubicación de las clases api y models autogeneradas, además que no tiene soporte para javascript, al menos en la versión gratuita, así que se va a ciegas.

Por el lado del cliente, opino que temas de seguridad como es el no-acceso al carrito hasta estar el usuario logeado no sería correcto, sino que se debería de haber gestionado de otra manera, por ejemplo que el carrito pudiese ser visible, pero una vez se pulse 'pagar' si sea necesario tener un token almacenado en el navegador por haberse logeado el usuario.