Manual Técnico de la Aplicación

Introducción:

Hemos usado un modelo Vista-controlador que consta de un **Back-end,** desarrollado en SpringBoot y un **Front-end** desarrollado en Angular usando Ionic como framework guía. Hemos escogido SpringBoot, porque nos aporta la solidez de un gran ecosistema, documentación, librerías y comunidad de gran variedad y calidad. Además la curva de aprendizaje de este es bastante asequible al principio pero también nos permite avanzar a la par que se complica el desarrollo de esta. Por otra parte, hemos escogido Ionic porque lo consideramos un 3 por 1, ya que nos aporta muchas facilidades cuando queramos desplegar y consumir la app en IOS y Android. El porqué de Angular, se basa principalmente en su estructura y en TypeScript. El paso desde Java a este nos parece muy sencillo y además nos ofrece distintas funcionalidades extras al JavaScript convencional.

Bases de datos:

Las bases de datos con las que vamos a trabajar son PostgreSQL y MongoDB. Hemos usado PostgreSQL para trabajar con las entidades más sencillas con relaciones entre sí. Y MongoDB para las más complejas o mutables como son los Pedidos.

Entidades:

Lo más relevante, es que usamos borrado lógico en toda nuestra aplicación, para ello hemos generado una columna deletedAt, del tipo fecha que nos permita saber si esta borrada y cuando ha sido. Aportándonos más información, sin deteriorar el rendimiento de nuestra app.

- **Addresses.java**: Fundamental para gestionar la información de las direcciones de los usuarios, esencial para funcionalidades como el envío de productos.
- **User.java y Role.java**: Cruciales para la gestión de usuarios y roles dentro del sistema, soportando la autenticación y autorización.

- **Category.java**: Permite organizar los productos en categorías, mejorando la navegabilidad y búsqueda en la aplicación.
- **Evaluation.java**: Facilita la implementación de un sistema de reseñas, mejorando la experiencia del usuario mediante feedback de otros clientes.
- **Offer.java**: Permite gestionar ofertas y promociones, atrayendo a más usuarios y aumentando las ventas.
- **Order.java y OrderedProduct.java**: Esenciales para gestionar los pedidos de los usuarios, incluyendo detalles específicos de cada producto ordenado y el estado del pedido.
- **Product.java**: Modelo central para la gestión de los productos ofrecidos en la plataforma, incluyendo detalles esenciales como nombre, precio y descripción.
- **Restaurant.java**: Permite gestionar los restaurantes que forman parte de la plataforma, facilitando la asociación de productos con sus respectivos proveedores.

Estructura de la aplicación:

Back-end:

Los controladores hacen la función de enrutador dentro de la app, ellos reciben la llamada y derivan al servicio que se encargue de gestionarla. Los servicios, son los que llevan el peso lógico de la aplicación, insertan buscan, verifican, etc… los datos en los repositorios. Usamos DTOs para comunicarnos con el front y para validar los datos que nos llegan. Para transformar los datos hemos usado la librería de SpringBoot mapstruct, que nos aporta una gran versatilidad a la hora de trabajar con las entidades y los mapers generados.

Hemos intentado enfocar el desarrollo de nuestra app, usando aplicando al máximo posible los principios SOLID. Para ello hemos creado un endpoint común con un CRUD completo. Primero, tenemos un servicio que se encarga de las tareas relacionadas con la verificación de los usuarios. Ya que además del token no todos los usuarios pueden acceder a entidades que no les pertenezcan. A parte, este contiene un servicio de excepciones con los tipos base de nuestra App (EntityNotFound…). De este heredarán los servicios comunes a las entidades relacionadas con MongoDB y otro para

PostgreSQL. Lo hemos dividido de esta forma ya que a cada uno de ellos le pasaremos un tipo repositorio distinto, asociado a cada tipo. Y será este el que implemente el CRUD básico de cada entidad. Más tarde aplicando genéricos, generaremos los servicios específicos para cada entidad. Lo hemos hecho de esta forma ya que hay algunas entidades que necesitan verificar o transformar los distintos datos que se puedan pasar. Pero los métodos que no necesiten estos pasos intermedios no serán implementados en las clases hijo, ya que con las del padre es suficiente.

Para paginar las distintas entidades usaremos una clase modelo llamada CommonFilters, que nos aportará el size, shortBy y direccion por defecto. Pudiendo obtenerlos en forma de Pageable que mandar al repo. Las especificaciones particulares de cada entidad se realizarán en los filters de la entidad. Para aplicarlos, solo tenemos que llamar al pageable() de este filter y obtener las especifications() y pasárselos al listAll del repo. Los mappers heredan de uno común que tiene los cambios más típicos y que usamos en nuestra app, de dto a modelo, sus list y viceversa. Además hemos añadido un update usando la anotación @TargetMapping.

El repositorio común, contiene métodos apropiados para buscar las entidades teniendo en cuenta los borrados lógicos. Además del método getTableName() que es necesario para el correcto funcionamiento de nuestra aplicación. Esto se debe a que preferimos que sea el usuario quien se encargue de pasar este dato a nuestra app. Ya que a veces SpringBoot, altera los strings transformándolos de camel case a snake case al comunicarse con las bbdd.

El commonController es una clase abstracta con el crud básico. Ya que nos interesa que sea el usuario quien se encargue de exponer y securizar las distintas rutas de nuestra app.

Front:

Para el front hemos usado una estructura similar a los proyectos de Angular. Por lo tanto, En nuestro proyecto, hemos desarrollado el front-end utilizando Angular, siguiendo una estructura modular que nos permite mantener una separación clara de responsabilidades y garantizar la escalabilidad de la aplicación. A continuación, describimos cómo hemos organizado los diferentes componentes y directorios:

Estructura General

Nuestra aplicación Angular está organizada en varios directorios y archivos que encapsulan funcionalidades específicas. Esta organización nos ayuda a mantener el código limpio, modular y fácil de mantener.

Directorios Clave

- **app**: Este es el núcleo de nuestro front-end. Aquí encontramos los componentes principales, servicios, modelos y configuraciones de rutas. Este directorio es fundamental para la estructura de nuestra aplicación.
- **assets**: Almacenamos aquí todos los archivos estáticos como imágenes, íconos y otros recursos multimedia que utiliza nuestra aplicación.
- **environments**: Este directorio contiene configuraciones específicas para diferentes entornos, como desarrollo y producción. Esto nos permite ajustar parámetros sin necesidad de modificar el código base.
- **theme**: En este directorio incluimos el archivo global.scss, donde definimos los estilos globales para nuestra aplicación, asegurando una apariencia coherente en todas las vistas.

Componentes y Módulos

En el directorio **app**, hemos organizado nuestros componentes, servicios y modelos de la siguiente manera:

- **Componentes Principales**:
  - **body**: Contiene la estructura principal de la aplicación.
  - **header**: Maneja la navegación y el encabezado de nuestra aplicación.
  - **footer**: Gestiona el pie de página.
- **Guardas y Servicios**:
  - **guards**: Aquí hemos definido las guardas de ruta, como role.guard.ts, para manejar la seguridad y restricciones de acceso basadas en roles.
  - **interceptors**: Incluye interceptores HTTP para gestionar errores y solicitudes.
  - **services**: Contiene los servicios que utilizan nuestros componentes para interactuar con la API backend.

- **Modelos y Entidades**:
    - **models**: Definimos aquí las entidades y DTOs como address.entity.ts, user.entity.ts, entre otros. Estas clases nos ayudan a mapear los datos entre el front-end y el back-end.

Páginas y Vistas

Hemos organizado las diferentes vistas de la aplicación en subdirectorios dentro de **pages**:

- **addresses**: Gestión de direcciones, incluyendo vistas para detalles, nuevas direcciones y actualización de direcciones.
- **cart**: Gestión del carrito de compras.
- **categories**: Manejo de categorías de productos.
- **evaluations**: Gestión de evaluaciones y reseñas.
- **login**: Página de inicio de sesión.
- **me**: Gestión del perfil de usuario.
- **offers**: Gestión de ofertas especiales.
- **orders**: Gestión de pedidos, incluyendo detalles y actualización de pedidos.
- **products**: Gestión de productos, incluyendo vistas para añadir, actualizar y listar productos.
- **register**: Página de registro de nuevos usuarios.
- **restaurants**: Gestión de restaurantes.
- **websocket-orders**: Manejo de pedidos en tiempo real mediante WebSockets.

Cada una de estas páginas tiene su propio conjunto de componentes, estilos y pruebas unitarias, lo que nos permite mantener cada sección de la aplicación modular y fácilmente gestionable.

Estilo y Tematización

En el directorio **theme**, utilizamos el archivo global.scss para definir estilos globales que se aplican en toda la aplicación, garantizando una apariencia consistente y profesional.

En resumen, hemos estructurado nuestro front-end de manera que cada componente, servicio y modelo tenga su propio lugar claramente definido. Esta organización nos

permite trabajar de manera eficiente, facilitando el mantenimiento y la escalabilidad de la aplicación.

Desarrollo:

El trabajo en distintos perfiles es fácil con springboot. Aunque pudiéramos trabajar con bbdd embebidas. Hemos preferido generar los Docker de estas desde el principio. Pudiendo consultarlos en cualquier momento aunque la app no funcionara. Ya sea mediante adminer o la interfaz de IntellIj.

El desarrollo nuestra aplicación ha sido aplicando la filosofía agile. Primero hemos intentado tener una aplicación base que nos permitiera gestionar un endpoint sencillo. Para más adelante generalizar todas las características posibles, usando comodines. Tras esto teníamos un endpoint base del que heredaran todos los siguientes. Avanzando poco a poco en función de los requisitos que tuviera nuestra aplicación. Por ejemplo, tras acabar los endpoints básicos, gestionamos la inserción de valoraciones y la tramitación de los pedidos. Tras tener una API Rest bastante robusta, nos pusimos con el front. Primero generamos una aplicación semilla que consumiera los datos de nuestra app. Tras esto, fuimos creando una malla de rutas y paginas que se parecieran lo máximo posible a una app real.

Dependencias:

Para la gestión de dependencias en el back, hemos usado Maven, en clase habíamos trabajado con Gradle y queríamos aprender este gestor. En el front, hemos usado node. Ya que tanto angular como ionic se basan en él.

Dependencias principales:

Angular

- **@angular/animations**: Animaciones de Angular.
- **@angular/cdk**: Component Development Kit de Angular, herramientas de UI.
- **@angular/common**: Módulos comunes de Angular.

- **@angular/compiler**: Compilador de Angular.
- **@angular/core**: Núcleo de Angular.
- **@angular/forms**: Módulos de formularios de Angular.
- **@angular/platform-browser**: Soporte para el navegador en Angular.
- **@angular/platform-browser-dynamic**: Compilación y ejecución dinámica en el navegador.
- **@angular/router**: Enrutador de Angular para navegación.

Ionic y Capacitor

- **@ionic/angular**: Integración de Ionic con Angular.
- **@capacitor/app**: API de Capacitor para la gestión de la aplicación.
- **@capacitor/camera**: API de Capacitor para el uso de la cámara.
- **@capacitor/clipboard**: API de Capacitor para el portapapeles.
- **@capacitor/core**: Núcleo de Capacitor para la integración con aplicaciones nativas.
- **@capacitor/haptics**: API de Capacitor para retroalimentación háptica.
- **@capacitor/keyboard**: API de Capacitor para la gestión del teclado.
- **@capacitor/status-bar**: API de Capacitor para la barra de estado.

Utilidades Angular

- **@maskito/angular**: Utilidades de Angular para Maskito.
- **@maskito/core**: Núcleo de Maskito, una biblioteca de máscaras de entrada.
- **ionicons**: Conjunto de iconos de Ionicons.
- **jwt-decode**: Biblioteca para decodificar JSON Web Tokens (JWT).
- **rxjs**: Biblioteca de programación reactiva de JavaScript.
- **tslib**: Biblioteca de soporte de TypeScript.
- **zone.js**: Biblioteca para el manejo de zonas en JavaScript, utilizada por Angular.

Spring Boot Starters

- **spring-boot-starter-web**: Esencial para construir aplicaciones web y RESTful APIs, proporcionando soporte para controladores, validación de datos, y manejo de excepciones.

- **spring-boot-starter-security**: Fundamental para añadir capas de seguridad a la aplicación, gestionando la autenticación y autorización de usuarios.
- **spring-boot-starter-data-jpa**: Simplifica la interacción con bases de datos relacionales, permitiendo operaciones CRUD y transacciones de manera sencilla y eficiente.

Bases de Datos

- **org.postgresql**: Necesario para conectar y ejecutar operaciones en una base de datos PostgreSQL, crucial si tu aplicación utiliza esta base de datos para almacenamiento persistente.

Utilidades

- **org.projectlombok**: Ayuda a reducir la cantidad de código repetitivo, haciendo el código más limpio y fácil de mantener.
- **org.mapstruct**: Facilita la conversión entre diferentes representaciones de datos (como entre DTOs y entidades JPA), mejorando la mantenibilidad y limpieza del código.

Seguridad

- **com.auth0**: Proporciona una manera segura y eficiente de manejar la autenticación basada en tokens, especialmente útil en aplicaciones con autenticación JWT.

Jackson

- **jackson-datatype-jsr310**: Facilita el trabajo con tipos de datos de fecha y hora en JSON, asegurando que las serializaciones y deserializaciones sean correctas y compatibles.

Apache POI

- **org.apache.poi y poi-ooxml**: Necesarias para generar y manipular archivos Excel, útiles para funciones como la exportación de datos y la generación de reportes.

Seguridad:

Nuestra app implementa una configuración de seguridad robusta utilizando varias características avanzadas proporcionadas por Spring Security. Aquí tienes un resumen de los componentes y su configuración:

Configuración CORS (Cross-Origin Resource Sharing)

- **CorsConfig.java**:
  - Configura CORS para permitir solicitudes desde http://localhost:4200.
  - Permite métodos HTTP específicos: GET, POST, PUT, DELETE, PATCH.
  - Define un tiempo máximo de 3600 segundos para las respuestas CORS.

Configuración de Seguridad

- **SecurityConfig.java**:
  - **Anotaciones**:
    - @Configuration, @EnableWebSecurity, @EnableMethodSecurity: Habilitan la configuración de seguridad y el uso de anotaciones de seguridad a nivel de métodos.
  - **SecurityFilterChain**:
    - Deshabilita CSRF.
    - Establece la política de creación de sesiones a STATELESS.
    - Configura una lista blanca de URLs que no requieren autenticación, como /auth/**, /products/**, /categories/**, etc.
    - Añade un filtro de autenticación JWT antes del filtro UsernamePasswordAuthenticationFilter.
  - **Autenticación y Autorización**:
    - Define un PasswordEncoder utilizando BCryptPasswordEncoder.
    - Configura un DaoAuthenticationProvider que utiliza userService para cargar los detalles del usuario y passwordEncoder para codificar las contraseñas.

        - Proporciona un AuthenticationManager a partir de la configuración de autenticación.

Filtro de Autenticación JWT

- **JwtAuthenticationFilter.java**:
    - Extiende OncePerRequestFilter para asegurarse de que cada solicitud sea filtrada una vez.
    - En el método doFilterInternal, se extrae y valida el token JWT de la cabecera Authorization.
    - Si el token es válido, se autentica al usuario y se añade la autenticación al SecurityContextHolder.

Métodos de Utilidad en Servicios

- **Métodos de Servicio**:
    - **getLoggedUser()**: Obtiene el usuario autenticado actual desde el SecurityContextHolder.
    - **getLoggedUserId()**: Obtiene el ID del usuario autenticado actual.
    - **verifyLogguedSameUser(User user)**: Verifica si el usuario autenticado es el mismo que el usuario proporcionado, lanza una excepción si no lo es.
    - **verifyLogguedSameUser(UUID userId)**: Verifica si el ID del usuario autenticado coincide con el ID proporcionado, lanza una excepción si no lo es.
    - **verifyWorker()**: Verifica si el usuario autenticado tiene los roles ADMIN o WORKER.

Uso de Anotaciones de Seguridad

- **@PreAuthorize**: Utilizas esta anotación para proteger métodos específicos a nivel de servicios, controladores, etc. Esto permite definir permisos de acceso basados en roles o expresiones SpEL.

Flujo de Seguridad

1. **CORS Configuration**: Permite solicitudes CORS desde dominios específicos, configurando qué métodos HTTP son permitidos.
2. **Security Filter Chain**: Define las reglas de autorización y deshabilita CSRF. Las rutas en la lista blanca son accesibles sin autenticación, mientras que todas las demás requieren autenticación.
3. **JWT Authentication**: El filtro JWT intercepta cada solicitud, valida el token y autentica al usuario si el token es válido.
4. **User Details Service**: Carga los detalles del usuario para autenticación utilizando UserDetailsService.
5. **Custom Methods**: Proporcionan funcionalidades adicionales para obtener el usuario autenticado y verificar roles y permisos.
6. **Method Security**: Las anotaciones como @PreAuthorize aseguran que los métodos sólo sean accesibles a usuarios con los permisos adecuados.

Producción:

El perfil de producción no tiene muchas cosas diferentes, desactivamos el swagger y herramientas de debug como las consultas etc. Y modificamos las rutas a las bbdd de producción. De cara al correcto despliegue en Docker.

Despliegue del proyecto:

El proyecto está estructurado en 3 carpetas principales: **Back-end**, **Front-end** y **database**. En root, se proporciona un archivo Docker-compose para facilitar el despliegue. Este archivo llama al Docker File del back-end, donde se ejecutará la aplicación back-end en el puerto 3000, y al Docker File del front-end, que lanzará el front-end de Ionic y Angular en el puerto 4200. También se incluyen imágenes y scripts para bases de datos PostgreSQL y MongoDB con datos predefinidos en la carpeta database.

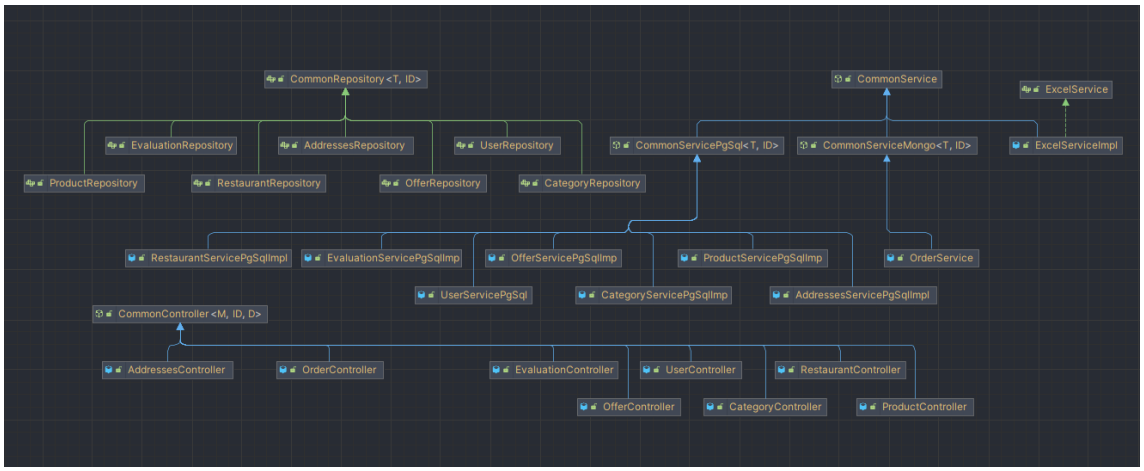
Acceso a la Aplicación

Una vez desplegada la aplicación, se puede acceder a ella a través de la URL http://localhost:4200/.

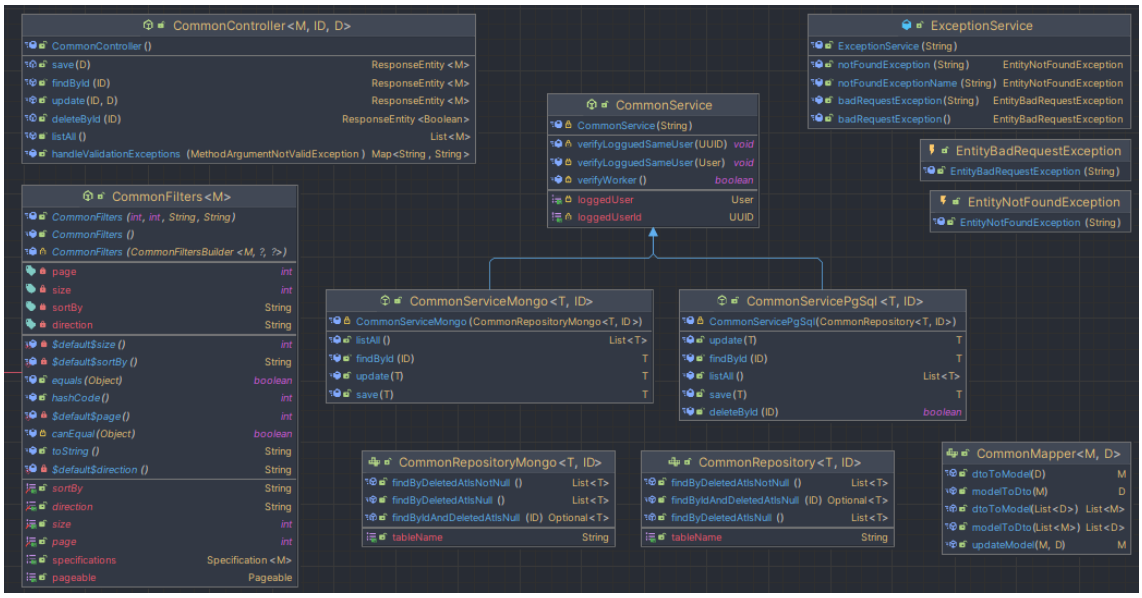
Por su parte las consultas a la Api Rest se harán a http://localhost:3000/v1/. Ya sea a través de Postman o alguna herramienta similar.

ANEXOS:

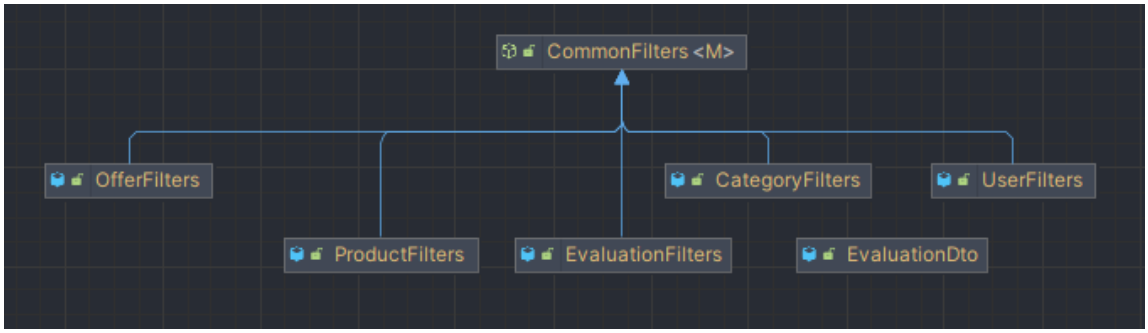
IMG_1: Esquema estructural de la herencia entre los repositorios, contoladores y servicios de nuestra app.



IMG_2: Esquema estructural de la composición del common.



IMG_3: Esquema estructural de la herencia entre los filters.



IMG_4: Esquema estructural de la herencia entre los mappers de nuestra app.