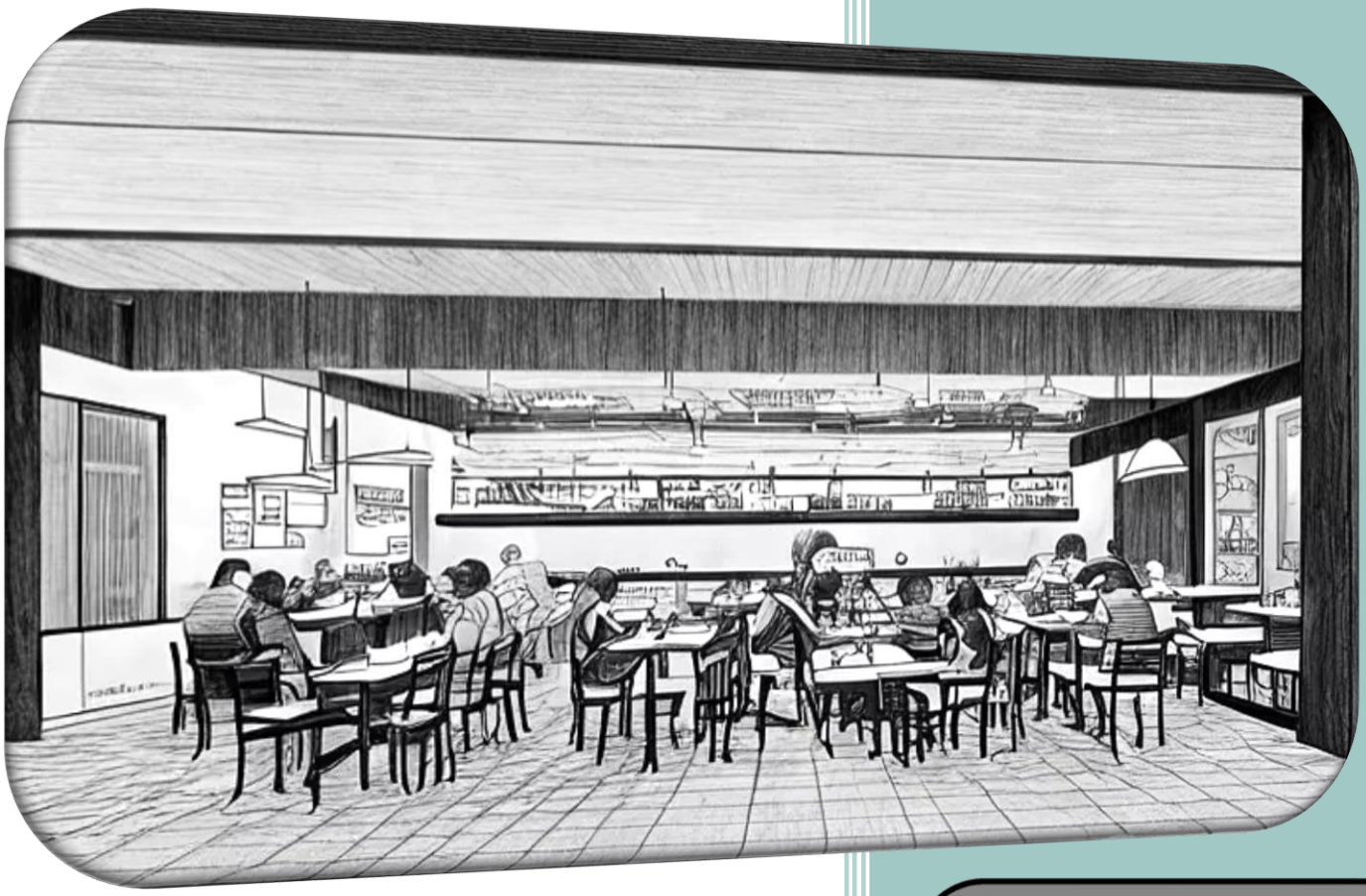
2023

MacJava



Jaime Lozano

Kevin Sánchez

Diego torres

Oscar Encabo

2º DAW

ÍNDICE:

1.Introducción:

Bienvenido a nuestra api creada por Jaime Lozano, Diego Torres, Oscar Encabo y Kelvin Sánchez, esta ofrece una administración segura de base de datos, completa y escalable.

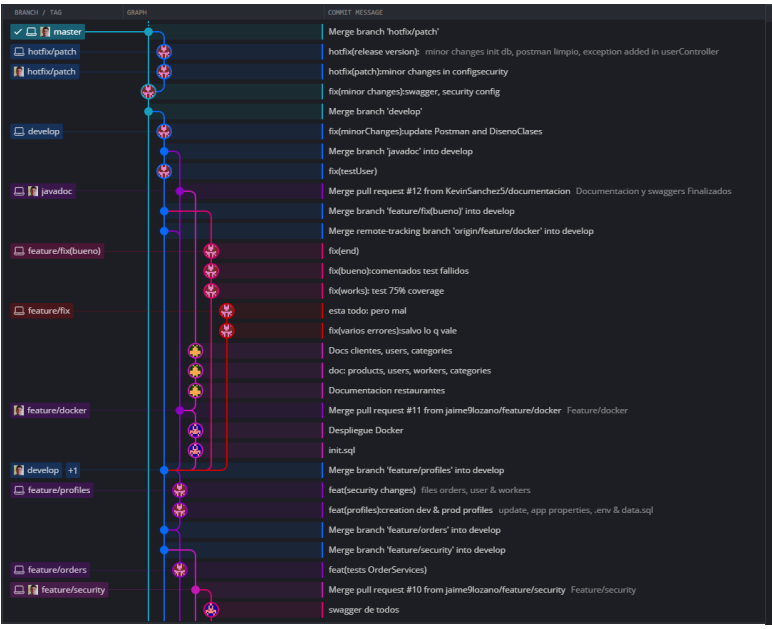
MacJava, nuestra api proporciona un conjunto de endpoints sólidos que facilita la gestión de una tienda online, ofrece operaciones relacionadas con productos, pedidos, trabajadores, restaurantes, posición de los trabajadores y clientes.

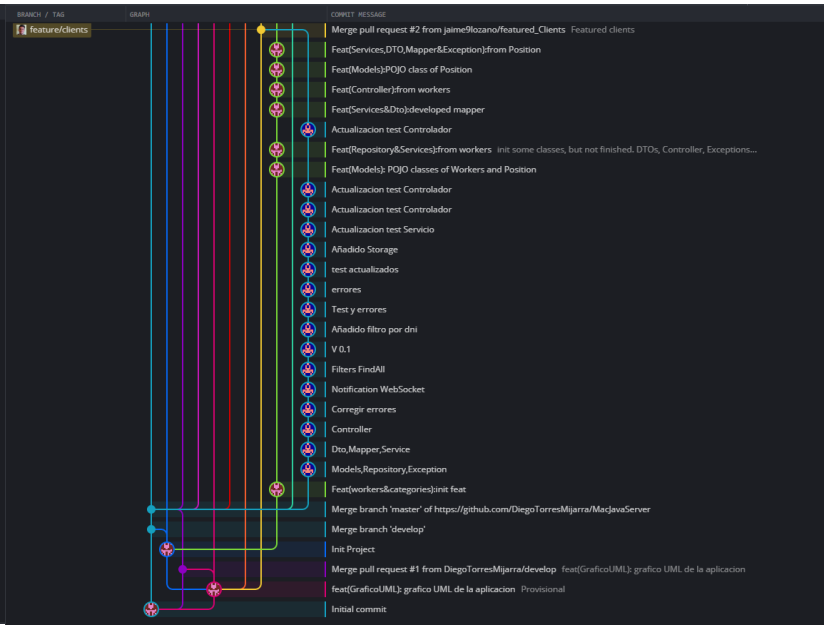
Dos de las características que nos han llevado a elegir Springboot son la facilidad a la hora de configurar el proyecto y su rápido desarrollo. SpringBoot se destaca por su capacidad para simplificar la configuración y acelerar el desarrollo de aplicaciones Java. Utiliza convenciones sobre configuración para proporcionar valores predeterminados sensibles y configuraciones automáticas basadas en el entorno, reduciendo así la necesidad de una configuración extensa. Además, facilita el desarrollo rápido al ofrecer un conjunto de bibliotecas y dependencias preconfiguradas, permitiendo a los desarrolladores iniciar proyectos sin preocuparse por integraciones complejas o configuraciones manuales.

2. GitFlow

Hemos seguido este esquema de organización, creando una rama develop sobre la que íbamos integrando (merge) otras ramas llamadas featured, cada cual tenía una característica del programa, teniendo así dos ramas estables y seguras, protegiendo el código preparado y correcto. Durante el proyecto hemos ido trabajando en paralelo, cada uno desarrollando las feature y una vez estas estaban testeadas, se iban añadiendo y comprobando a develop. Casi al final hubo un problema con el código y distintas clases, pero conseguimos recuperarlo gracias a ramas auxiliares fix. Por último, cuando hemos considerado que la aplicación estaba bien, hemos realizado el merge a master. Aunque nos dimos cuenta de algún error que corregimos en un hotfix.

Estas son algunas capturas de nuestro árbol de git:





3. Dependencias:

Las dependencias usadas en nuestro proyecto son springboot (data JPA, Web, cache, validation, websocket, security), dependencias de bases de datos (PostgreSQL, h2, mongoDB), lombok, swagger, jackson, jwt y finalmente dependencias para testear.

4. Bases de datos

Decidimos utilizar H2 para desarrollar la api debido a que puede funcionar en memoria, esto mejora el rendimiento, acelera el desarrollo facilitando la configuración de los datos y no hace falta administrar un servidor, ofrece un soporte muy amplio para estándares SQL, permitiendo hacer consultas complejas y por último la familiaridad que tenemos con este sistema.

A su vez el uso de PostgreSQL en producción ofrece robustez escalabilidad y estabilidad, puede manejar grandes volúmenes de datos a nuestro programa, tiene varias características avanzadas de búsqueda, da soporte a tipos de datos complejos y existe una comunidad activa (hay buen soporte en términos de correcciones de errores, recursos y actualizaciones), finalmente tiene una licencia opensource, al no tener restricciones de licencia ahorramos costes.

5. Características comunes de los endpoints:

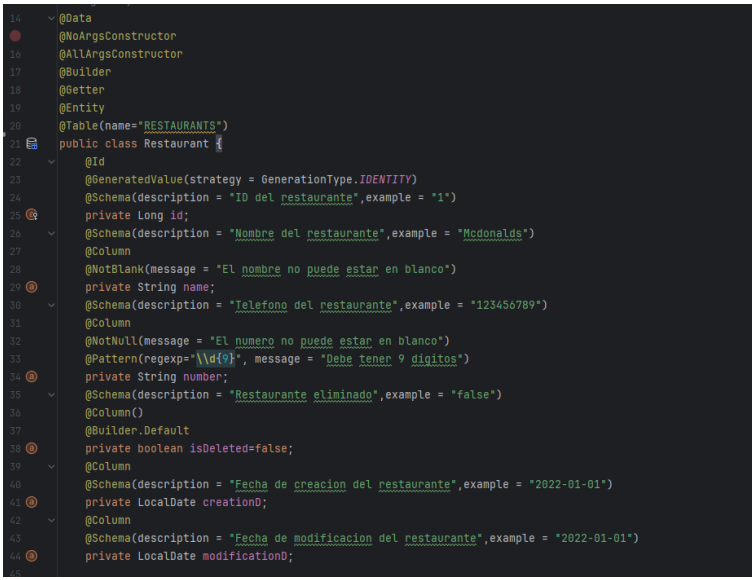
Varios endpoints de nuestro programa siguen una serie de características y métodos comunes que explicamos en los siguientes puntos.

Modelos

Los modelos tienen notaciones de Lombok (data, allargsconstructor) para facilitar la creación de constructores, creación de getters y setters, etc. y notaciones de SpringbootJPA y web (Entity, Table…) para indicar a springboot que las clases deben mapearse a tablas de la base de datos o para personalizar la configuracion de las tablas de la bbdd, y atributos con validaciones las incorporadas.

Los atributos que tienen en común los modelos de datos de nuestros endpoints son los id, algunos son autonumérico y otros uuid.

Los trabajadores y clientes tienen un atributo único, el DNI, la fecha de creación y fecha de modificación y un boolean que representa si está borrado o no, para permitir un borrado lógico de las entidades. En estas clases representan un modelo de datos de restaurante, por ejemplo:



UUIDs y Autonuméricos:

Las razones por las cuales restaurante, categoría y productos utilizan un identificador autonumérico, las colisiones son poco probables al usar Springboot la seguridad y privacidad de estos endpoints no es prioritario, la ordenación secuencial facilita la ordenación e indexación de datos, donde más nos conviene es en productos donde vamos a tener varios almacenados y este tipo de identificadores ocupan menos espacio y ofrecen un rendimiento mejor.

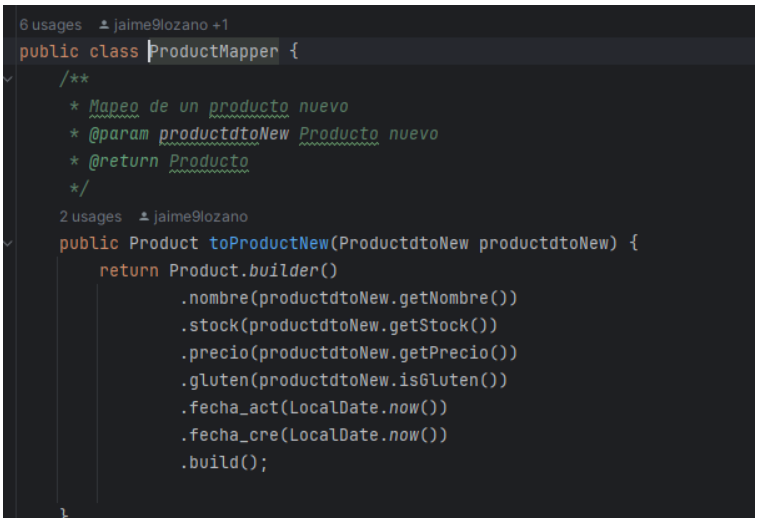
En cuanto a los uuid, usados en trabajadores, clientes y usuarios, decidimos darles este identificador para ofrecerles más seguridad y privacidad, son únicos a nivel mundial y la probabilidad de colisión es ínfima, además son independientes de las bases de datos lo que facilita replicación e integración de datos entre sistemas.

DTOs, Mappers y Excepciones:

Otra característica en común de nuestros endpoints es la existencia de DTOs para transferir datos y mappers para transformar los DTOs a las clases modelo y viceversa.

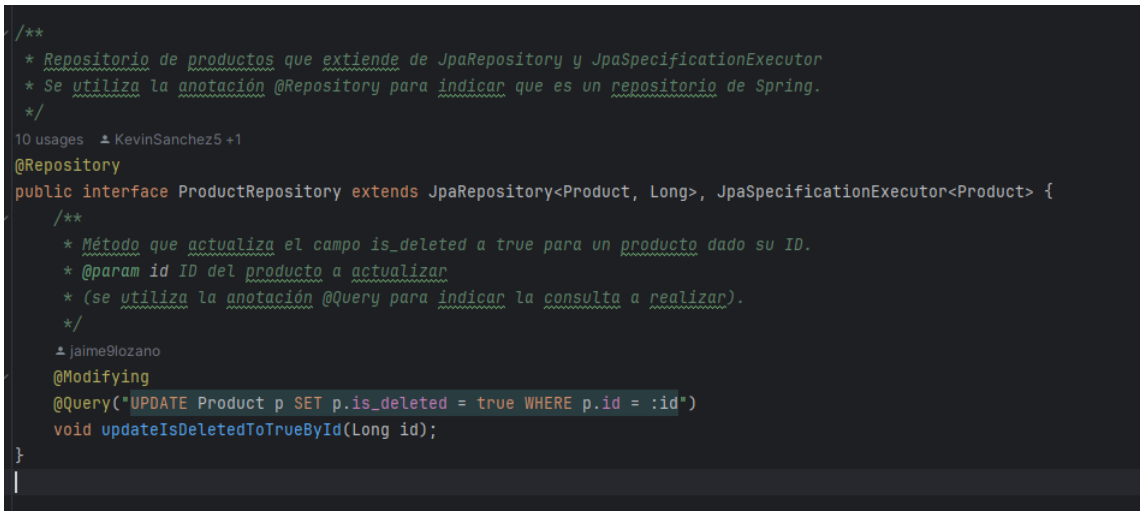
```java
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class ProductdtoUpdate {
    @Schema(description = "Nombre del producto" , example = "Coca Cola")
    @NotBlank(message = "El nombre no puede estar vacío")
    private String nombre;
    @Schema(description = "Precio del producto" , example = "12.0")
    @Positive(message = "El precio no puede ser negativo")
    @NotNull(message = "El precio no puede estar vacío")
    private double precio;
    @Schema(description = "Stock del producto" , example = "10")
    @PositiveOrZero(message = "El stock no puede ser negativo")
    @NotNull(message = "El stock no puede estar vacío")
    private Integer stock;
    @Builder.Default
    @Schema(description = "¿Es Gluten Free?" , example = "true")
    private boolean gluten = true;
    @Builder.Default
    @Schema(description = "¿Esta eliminado?" , example = "true")
    private boolean is_deleted = false;
}
```

Todos los endpoints tienen una serie de excepciones personalizadas para indicar si el objeto no ha sido encontrado y si el usuario que realiza la consulta ha entregado un body incorrecto.



Repositorio:

Al usar SpringbootJPA los endpoints tienen un repositorio básico ya que las consultas básicas están codificadas, aun así, incluimos algunos métodos para realizar las consultas más específicas de manera manual. Todas tienen la siguiente consulta para realizar el borrado lógico:



Servicio:

Los servicios implementan la interfaz de servicios de cada endpoint y se encarga de gestionar las operaciones relacionadas con los productos en una aplicación. Está anotado con @Service para indicar que es un servicio de Spring y utiliza la anotación @CacheConfig para configurar la caché asociada a este servicio.

Inyección de Dependencias:

Se utiliza la inyección de dependencias mediante la anotación @Autowired para proporcionar una instancia del repositorio en el momento de la creación del servicio.

Métodos Principales:

- FindAll: este método recupera una página de productos según los parámetros de búsqueda especificados. Utiliza especificaciones JPA para construir criterios de búsqueda dinámicos.
    - Parámetros: dependen de la clase los que hay en comun son
        - Is_deleted: Indica si el producto está eliminado (opcional).
        - Pageable: Información de paginación.
    - Retorno: Página de los objetos que cumplen con los criterios de búsqueda.
- FindById:
    - Descripción: recupera un producto por su ID.
    - Parámetro: id del producto a buscar.
    - Retorno: Producto con el ID indicado.
    - Excepción: NotFound si no se encuentra.
- Save:
    - Descripción: guarda un nuevo en la base de datos.
    - Parámetros:
    - DtoNew: DTO con la información a guardar.
    - Retorno: objeto guardado.
- Update:
    - Descripción: Actualiza un objeto existente según su ID.
    - Parámetros:
        - id del producto a actualizar.
    - DtoUpdate: DTO con la información a actualizar.
    - Retorno: objeto actualizado.
    - Excepción: NotFound si no se encuentra.

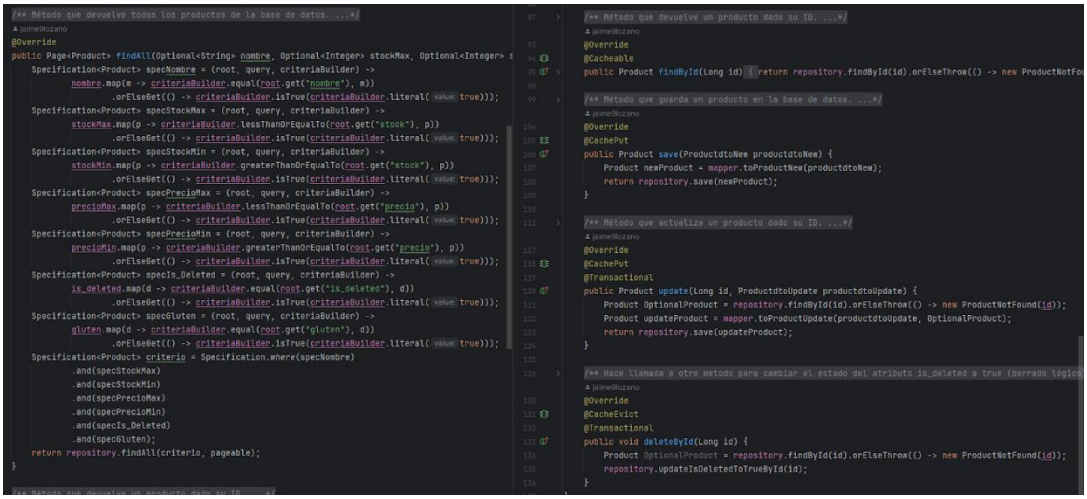
- DeleteById:
    - Descripción: realiza un borrado lógico cambiando el estado is_deleted a true para un producto dado su ID.
    - Parámetro: id del producto a eliminar.
- Anotaciones de Caché:
    - @Cacheable: Indica que el resultado del método se debe almacenar en caché para consultas futuras.
    - @CachePut: Indica que el resultado del método se debe almacenar en caché y también se debe invocar el método para actualizar la caché.
    - @CacheEvict: Indica que la entrada en caché asociada al método debe ser eliminada.

Transacciones: se utiliza la anotación @Transactional para indicar que ciertos métodos deben ejecutarse dentro de una transacción, asegurando la consistencia de la base de datos.

Mapper: se utiliza un Mapper para realizar la conversión entre DTOs y entidades Product.

Eliminación de Cache: el método deleteById utiliza @CacheEvict para eliminar la entrada relacionada en la caché después de realizar el borrado lógico.

Este servicios proporciona una interfaz entre la capa de negocio y el repositorio, gestionando operaciones CRUD, búsquedas avanzadas y mantenimiento de la caché para mejorar el rendimiento.



Controladores:

Varios controladores tienen características similares, todos con sus peticiones básicas (findAll, findById, delete, save, update), posteriormente nos enfatizamos en la explicación de los controladores más avanzados.

6. Categorías:

El endpoints de categorías, es sencillo, además de las características comunes del resto de endpoints (updateAt, createdAt e isDeleted), incluimos un id del tipo Long como clave primaria. Un nombre que esta limitado a los puestos: 'MANAGER', 'COOKER', 'CLEANER', 'WAITER', 'NOT_ASSIGNED'. Por lo que podríamos hacer que fuera de tipo unique llegado un momento y que solo sean creadas en circunstancias muy específicas. Por último, hemos añadido el atributo salary, con el objetivo de poder poner un sueldo base a los empleados que sean de esa categoría.

7. Trabajadores y clientes:

Estos dos endpoints tienen un comportamiento similar, en un principio solo los trabajadores tienen un rol de control del programa (User), asi que son los únicos que pueden realizar cambios en las bases de datos,a parte de los administradores. El programa está pensado para dar cobertura a locales de manera presencial y local, en un futuro se podrá añadir otro rol a los clientes para que puedan realizar sus compras online.

Las funciones del endpoint de Trabajadores sólo pueden ser usadas por un usuario Admin. Este endpoint tiene las peticiones básicas y un método diferente al resto que busca a los trabajadores dependiendo de la variable isDeleted para poder mostrar también los trabajadores borrados.

Al igual que Trabajadores, los clientes y las categorías (posición de los trabajadores) solo pueden ser modificados por un usuario Admin.

8.Restaurantes y productos:

Estos dos endpoints tienen un control mixto, hay algunas acciones que pueden ser controladas por todo el mundo, no hace falta tener un usuario y otras acciones solo pueden ser realizadas por los administradores.

Todo el mundo tiene acceso a los findAll y los getById de estos dos endpoints, al ser información básica que debe administrar a los clientes.

Las acciones solo permitidas a los administradores son los guardados, modificaciones y borrado.

Se tienen en cuenta las respuestas que puede realizar, por ejemplo, en el modificar de la clase restaurante: Código 200 si es correcto y tiene que devolver algo, 404 si no es encontrado el restaurante a modificar, 400 si los parámetros dados a modificar son incorrectos y finalmente 401 si el usuario no está autorizado a realizar esa acción.

9. Authentication:

En el endpoint de auth lo que hacemos es poder registrarse e iniciar sesión en nuestra tienda para según qué usuario seamos tendremos permisos para hacer unas cosas u otras. Tenemos 3 servicios dentro:

- Uno es el propio de auth en el que tendremos los métodos del controlador y que son registrarse y loguearse.
- El siguiente es de jwt el cual va a estar jugando con el token para sacar el username y ver si es correcto, generar el token y ver si el token es válido para estar siempre con seguridad en nuestro endpoint.
- Y el último que es auth de usuarios en el cual vamos a sacar los datos del usuarios buscando por username.

En el controlador como ya he dicho tendremos los métodos registrar e iniciar sesión con sus respectivos dto, validaciones y sus excepciones.

10.Usuarios:

En el endpoint de usuarios vamos a poder llevar toda la información de los usuarios tanto a nivel admin que podrá ver todo como a nivel user que podrá ver las cosas propias del usuario.

Trabaja con uuid e implementa UserDetails en el modelo y con ello tiene unos métodos incluidos para gestionar los usuarios.

El controlador va a trabajar tanto con el servicio de usuarios como con el de orders, y aquí aparte de los métodos del crud normal que son los que podrá ver el admin (sacar todos los usuarios,modificar alguno,crear nuevo y borrar alguno),tambien existiran los que podrá ver el user y son propios de estar logueados y el propio perfil como son: Ver tu perfil, actualizarlo y eliminarlo o ver tus pedidos, ver un pedido tuyo buscado por id, crear o actualizar un pedido tuyo y borrar un pedido tuyo(todos estos métodos se revisan para que solo los pueda hacer si es igual al usuario que está conectado).

Como último apunte en el servicio se utiliza passwordEncoder para que las contraseñas no se pasen a simple vista sino que van codificadas y a la hora de usarlas se descodifican y luego se vuelven a codificar.

11. Pedidos:

Este es el endpoint más complejo del programa, administra los pedidos de los restaurantes, clientes y trabajadores.

Para almacenar los pedidos usamos MongoDB, elegimos esta base de datos por su gran flexibilidad, su optimización en operaciones de lectura y escritura, la capacidad de realizar cambios en el esquema (por ejemplo, agregar o eliminar campos) de manera dinámica y también por su modelo de documentos BSON que permite representar pedidos.

Existen dos clases en este endpoint OrderProduct y Order:

La clase OrderedProduct es un objeto de producto que ha sido pedido por un cliente, del producto se obtiene su id, una cantidad y un precio total. La cantidad es pasada por el cliente y el precio es calculado automáticamente.

La clase Order agrupa todos estos OrderedProduct (productos pedidos) en una lista.

Los pedidos (clase Order) usa uuid como id y en el objeto se incluyen 3 referencias, el trabajador, el cliente y el restaurante. Cada una de estas referencias señala a la clave primaria de cada entidad, por ejemplo en el cliente su uuid. También tiene un booleano isPaid que indica si los pedidos han sido pagados.

En el servicio se incluyen estos métodos que realizan el cálculo del precio o la suma de todos las cantidades, productos pedidos y precio total.

Estos datos son pasados por el usuario la cantidad el precio se calcula automáticamente

En el order hay un totalpriced que hace referencia a la suma del precio de todos los productos pedidos. Cantidad total de productos, la suma de todas las cantidades de productos pedidos. Tiene un boolean isPaid indicando si está pagado el pedido o no.

Cuando set ordered product pasa una lista de productos pedidos la almacena y calcula la cantidad total de los productos y el precio de estos total.

Servicio tiene el repositorios de todos los endpoints relacionados para validar las entidades de los otros repositorios, inyecta los singeltons de los otros repositorios.

12. Despliegue:

Para el despliegue de la aplicación hemos elegido Docker, de cara a poder subirlo de forma sencilla en gracias a servidores como netlify. Nuestra construcción de los Docker se divide en 4 documentos que dependen de un enviroment común. En este podremos variar las características de nuestro programa como los nombres y puertos de las base de datos, las contraseñas, versiones, etc.

El documento Dockerfile presenta dos etapas claramente definidas: una de compilación y otra de ejecución. En la primera etapa, se utiliza una imagen de Docker específica para

compilar un proyecto Java con Gradle. Se copian los archivos de configuración y el código fuente, se realiza la compilación y se genera el archivo JAR del proyecto. En la segunda etapa, se utiliza una imagen de Java mas ligera (Alpine). Luego se copia el Jar generado y se expone en el puerto 3000. Además se le pasa el perfil que queremos que se use al iniciar la aplicación, en este caso “prod”.

El documento docker-compose-db, es el que se encarga de generar los Dockers de nuestras base de datos, PostgreSQL y MongoDB, junto con herramientas de administración como Adminer y Mongo Express para facilitar la conexión y visualización de datos durante el desarrollo. Como hemos mencionado, obtienen algunos valores de nuestro archivo env. También establece la red en la que se comunicaran nuestros distintos Docker. En producción solo serán necesarios las base de datos.

En el Docker-compose-app construimos el Docker de nuestra api-rest. Para ello utilizamos el archivo Dockerfile, que hemos mencionado antes. Definimos el nombre y la gestión de los volúmenes y establecemos el puerto de nuestra app, y la red de comunicación común.

Por último, Docker-compose.yml es el archivo que genera los Docker finales para ser distribuida. Es decir, genera las bases de datos y la app. E inicia los Docker, para esto espera a que las bases estén desplegadas, para iniciar la app.

13. Tests:

En los tests, hemos creado pruebas de caja negra y caja blanca para verificar el correcto comportamiento de nuestra app. Además para simular los comportamientos de algunas clases y métodos de nuestro código hemos usado Mockito. Para comprobar hemos usado los asserts de Junit5 y en algún caso hemos asegurado las pruebas, con el metodo verify.

En los tests del servicio mockeamos todos los repositorios que use ese servicio, para que cuando realicemos las consultas, devolver los valores o excepciones deseados. En los métodos de prueba, se evalúa la recuperación de órdenes, tanto de manera general

como con paginación. Además, se verifica la correcta manipulación de datos al realizar las distintas operaciones, como salvar, eliminar y actualizar. Tambien se evalúan otros métodos como si los datos buscados existen en el repositorio. Estos nos ayudan a realizar otras operaciones, como borrados, etc. Cada método, se centra en una operación concreta y evaluamos tanto casos positivos, como las excepciones que puedan lanzar. Cuando hemos considerado, tambien hemos establecido métodos setUp que se realicen antes de cada prueba por si hemos modificado algún objeto de prueba. Un ejemplo de los test de servicio es el siguiente:

```java
@ExtendWith(MockitoExtension.class)
class OrdersServiceImplTest {

    @Mock
    private OrdersCrudRepository ordersCrudRepository;

    @Mock
    private ClientsRepository clientsRepository;

    @Mock
    private ProductRepository productRepository;

    @Mock
    private RestaurantRepository restaurantRepository;

    @Mock
    private WorkersCrudRepository workersCrudRepository;

    @InjectMocks
    private OrdersServiceImpl ordersService;

    private final Product product= Product.builder()
            .id(1L)
            .stock(100)
            .precio(10.00)
            .build();
```

```java
    @BeforeEach
    void setUp(){
        order1.setOrderedProducts(validList);
        orderList = List.of(order1,order2);
    }
    @Test
    void testFindAll() {
        // Mock data
        when(ordersCrudRepository.findAll()).thenReturn(orderList);

        // Test the method
        List<Order> result = ordersService.findAll();

        // Verify the result
        assertAll("testFindAll",
                () -> assertNotNull(result),
                () -> assertEquals(2, result.size())
        );

        // Verify that the repository findAll method is called
        verify(ordersCrudRepository).findAll();
    }

    @Test
    void testFindAllPageable_ShouldReturnAll_whenNoParamsPassed() {
        Pageable pageable = PageRequest.of(0, 10, Sort.by("id").ascending());
        Page<Order> expectedPage = new PageImpl<>(orderList);
        when(ordersCrudRepository.findAll(any(Pageable.class))).thenReturn(expectedPage);

        Page<Order> result = ordersService.findAll(pageable);

        verify(ordersCrudRepository).findAll(pageable);

        assertAll("testFindAllPageable_ShouldReturnAll_whenNoParamsPassed",
                () -> assertNotNull(result),
                ()-> assertFalse(result.isEmpty()),
                () -> assertEquals(2, result.getTotalElements())
        );
    }
```

Para los controladores hemos usado las mismas estrategias. Lo más diferente es que en estos hemos verificado también el acceso de los usuarios. Para ello hemos usado la anotación @WithMockUser y @WithAnonymousUser, para que nos generarán usuarios con los roles que quisiéramos o la ausencia de ellos. Para cargar las pruebas hemos usado un entorno de pruebas generado por @SpringBootTest. El MockMvc, es el que se encargara de generar las consultas a los endpoints, con los datos que queramos.

```java
@Test
void testSaveOrder_ShouldThrowForbiddenException() throws Exception {
    String localEndPoint=myEndpoint+"/orders";
    MockHttpServletResponse response = mockMvc.perform(
                    post(localEndPoint)
                            .accept(MediaType.APPLICATION_JSON)
                            .contentType(MediaType.APPLICATION_JSON)
                            .content(mapper.writeValueAsString(orderSaveDto)))
            .andReturn().getResponse();
    assertAll("saveOrder_ShouldThrowForbiddenException",
            () -> assertEquals(403, response.getStatus())
    );
}
@Test
@WithMockUser(username = "admin", password = "u", roles = {"ADMIN","USER"})
void testSaveOrderAdmin() throws Exception {
    String localEndPoint=myEndpoint+"/ordersAdmin";
    when(ordersService.save(any(OrderSaveDto.class))).thenReturn(order1);
    MockHttpServletResponse response = mockMvc.perform(
                    post(localEndPoint)
                            .accept(MediaType.APPLICATION_JSON)
                            .contentType(MediaType.APPLICATION_JSON)
                            .content(mapper.writeValueAsString(orderSaveDto)))
            .andReturn().getResponse();

    Order result = mapper.readValue(response.getContentAsString(), new TypeReference<>() {});
    assertAll("saveOrderAdmin",
            () -> assertEquals(201, response.getStatus()),
            ()-> assertEquals(order1,result)
    );
}
```

```java
@SpringBootTest
@AutoConfigureMockMvc
@ExtendWith(MockitoExtension.class)
@WithMockUser(username = "user", password = "u", roles = "USER")
@Import(SecurityConfig.class)
class OrdersRestControllerTest {
    private final String myEndpoint="http://localhost:8080/v1/orders";
    private final User mockUser=User.builder()
            .username("User")
            .roles(Set.of(Role.USER))
            .id(UUID.fromString("550e8400-e29b-41d4-a716-446655440000"))
            .build();
    private final Product product= Product.builder()
            .id(1L)
            .stock(100)
            .precio(10.00)
            .build();

    private final List<OrderedProduct> validList=List.of(
            OrderedProduct.builder()
                    .quantity(10)
                    .productId(1L)
                    .productPrice(10.00)
                    .totalPrice(100.00)
                    .build(),
            OrderedProduct.builder()
                    .quantity(50)
                    .productId(1L)
                    .productPrice(10.00)
                    .totalPrice(500.00)
                    .build()
    );
```

```java
@Test
@WithAnonymousUser
void NotAuthenticated() throws Exception {
    // Localpoint
    MockHttpServletResponse response = mockMvc.perform(
                    get(myEndpoint)
                            .accept(MediaType.APPLICATION_JSON)
                            .contentType(MediaType.APPLICATION_JSON))
            .andReturn().getResponse();

    assertEquals(403, response.getStatus());
}
@Test
@WithMockUser(username = "admin", password = "u", roles = {"ADMIN","USER"})
void testFindAll() throws Exception {
    String localEndPoint=myEndpoint+"/listAll";
    when(ordersService.findAll()).thenReturn(orderList);
    MockHttpServletResponse response = mockMvc.perform(
                    get(localEndPoint)
                            .accept(MediaType.APPLICATION_JSON))
            .andReturn().getResponse();
    List<Order> result = mapper.readValue(response.getContentAsString(), new TypeReference<>() {});
    assertAll("findAll",
            () -> assertEquals(200, response.getStatus()),
            ()-> assertEquals(orderList,result)
    );
}
@Test
@WithMockUser(username = "admin", password = "u", roles = {"ADMIN","USER"})
void testFindAllPaged() throws Exception {
    String localEndPoint=myEndpoint;
    var res= new PageImpl<>(orderList);
    when(ordersService.findAll(any(Pageable.class))).thenReturn(res);
    MockHttpServletResponse response = mockMvc.perform(
                    get(localEndPoint)
                            .accept(MediaType.APPLICATION_JSON))
            .andReturn().getResponse();
    PageResponse<Order> result = mapper.readValue(response.getContentAsString(), new TypeReference<>() {});
    assertAll("findAllPaged",
            () -> assertEquals(200, response.getStatus()),
            ()-> assertEquals(orderList,result.content())
    );
}
```

14. Presupuesto:

Costo por hora de los programadores: Digamos que el costo por hora de cada programador es de 7,5€.

Horas de trabajo: Estimemos cuántas horas se necesitarán para completar el proyecto. Por ejemplo, si estimas que cada programador trabajará 160 horas al mes y el proyecto tomará 6 meses, tendríamos 4 programadores * 160 horas/mes * 6 meses.

Total: Multiplica el costo por hora por el número total de horas estimadas.

Entonces, si el costo por hora de cada programador es de 7,5 € y el proyecto toma 2 meses con 4 programadores que trabajan 160 horas al mes, el cálculo sería:

Costo por hora por programador = 7,5 €

Horas de trabajo = 4 programadores * 160 horas/mes * 2 meses = 1280 horas

Presupuesto total = 7,5 €/hora * 1280 horas = 9600 €