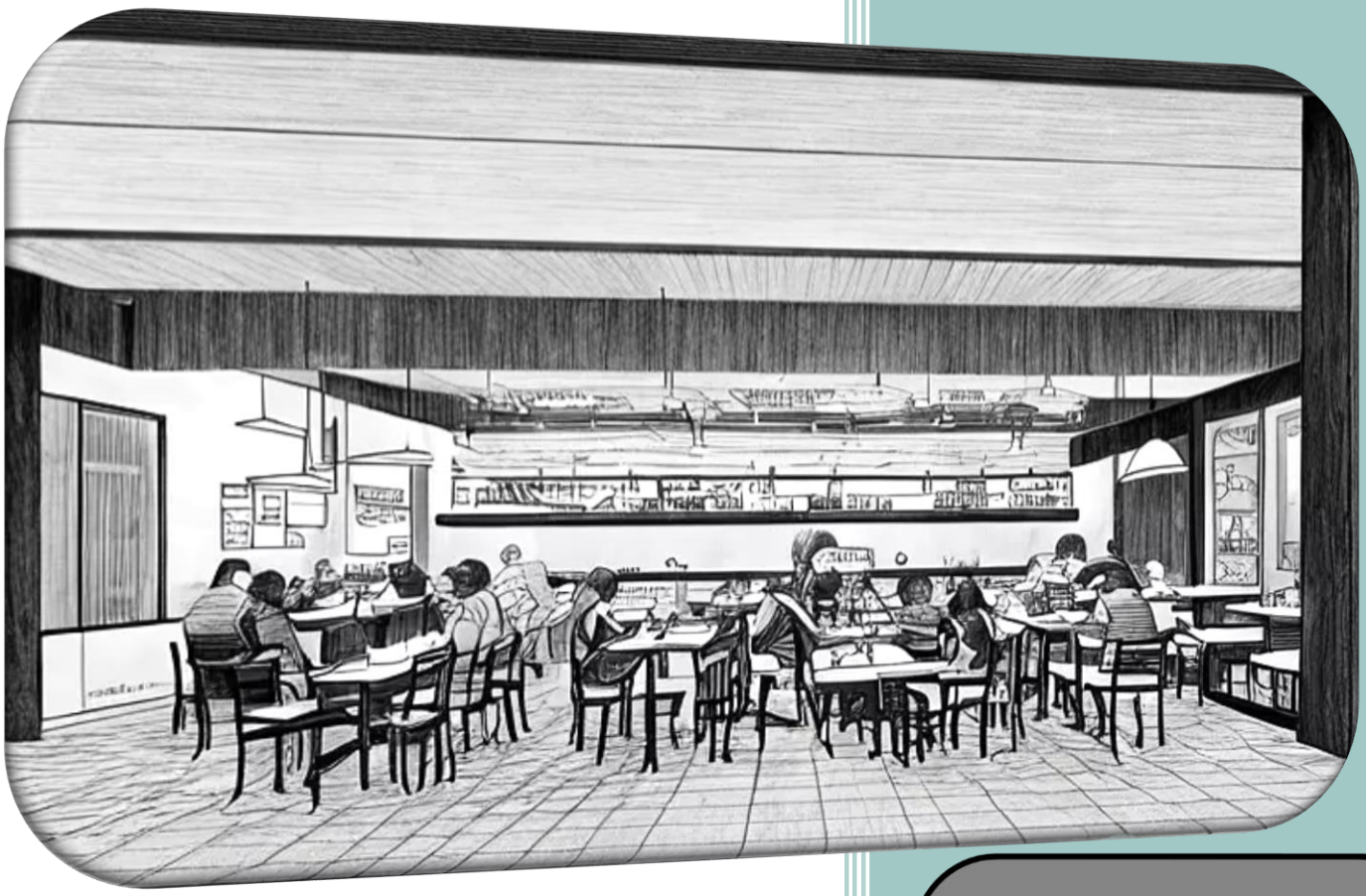
2023

MacJava



Jaime Lozano

Kevin Sánchez

Diego torres

Raúl Rodríguez

Oscar Encabo

2º DAW

ÍNDICE:

1.Introducción:

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

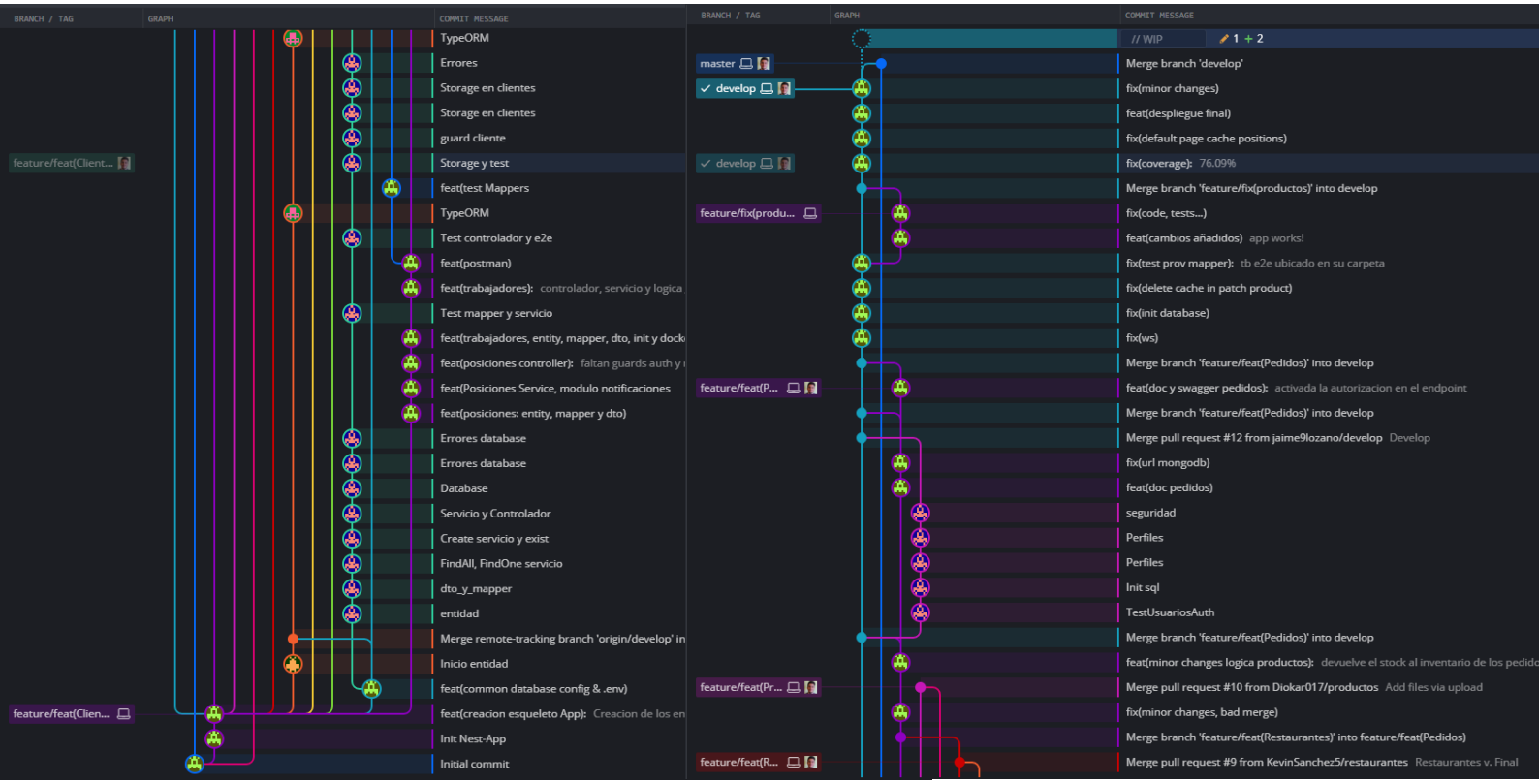
MacJava, nuestra api proporciona un conjunto de endpoints sólidos que facilita la gestión de un restaurante online, ofrece operaciones relacionadas con productos, pedidos, trabajadores, restaurantes, posición de los trabajadores, clientes y proveedores.

Dos de las características que nos han llevado a elegir NestJs y por lo tanto NodeJS y TypeScript son las siguientes:

- Usa TypeScript como lenguaje, permite usar características moderas de JavaScript y lo más importante existe un tipado estático que proporciona un código más organizado y reduce la producción de errores y el control de estos.
- Arquitectura escalable y modular. Utiliza el patrón de diseño de Inyección de dependencias y el principio de responsabilidad única. Facilita la escalabilidad y el mantenimiento de la aplicación.
- Soporta decoradores, el código es más claro y expresivo gracias a esto. Se pueden definir tanto en controladores como servicios y modelos.
- Permite la integración de muchas bibliotecas, especialmente de bases de datos (PostGreSQL, MongoDB, GraphQl...) al igual que funcionalidades avanzadas como autenticación, validación, control de repositorios, logging...
- Facilita tanto pruebas unitarias como de integración mediante el use de inyección de dependencias. En nuestro caso usamos Jest y Supertest.
- Existe una comunidad activa que puede ayudarnos a implementar nuevas herramientas, al igual que existe una amplia cantidad de documentación de la que inspirarnos para desarrollar nuevas características para la aplicación.

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.



3. Dependencias:

Las dependencias principales usadas en nuestro proyecto de NestJS son las siguientes cache manager, dependencias para controlar las bases de datos TypeORM (PostGreSQL) y mongoose (MongoDB), websockets, swagger, paginate, jwt y finalmente dependencias para testear (Jest y Supertest).

4. Bases de datos

Usamos PostGreSQL para datos estructurados, para la información de los clientes, trabajadores, posición de estos, restaurantes, proveedores y productos.

La integridad referencial de esta BBDD permite la consistencia de datos y evita la inconsistencia entre tablas relacionadas. Otra razón por la que elegimos esta base de datos es la capacidad de realizar consultas complejas que involucran varias tablas. Soporta el lenguaje SQL al completo, da soporte a tipos de datos complejos y existe una comunidad activa, es altamente escalable, puede manejar grandes volúmenes de datos y tiene un rendimiento sólido y eficiente, finalmente tiene una licencia opensource, al no tener restricciones de licencia ahorramos costes.

También usamos MondoDB para almacenar los datos de los pedidos

Esta base de datos nos permite almacenas datos semiestructurados, como son los pedidos, ya que estos varían según el tipo de pedidos, productos, clientes, restaurante...

Es altamente escalable, especialmente de manera horizontal, puede manejas altos volúmenes de datos y ofrece un muy buen rendimiento en cuanto a operaciones rápidas de lectura y escritura, esto es muy beneficioso para los pedidos, ya que son datos que se piden en tiempo real y la velocidad de respuesta es fundamental.

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 TypeORM para indicar que tipos de datos tienen las entidades, su clave primaria, mediante la notación @Column se puede indicar el tipo en

la base de datos, su valor, por ejemplo, en su tiempo de creado, se almacena el día en que fue almacenado por primera vez.

Los atributos que tienen en común los modelos de datos de nuestros endpoints son los id, algunos son autonumérico y otros uuid, también todos tienen una fecha de creación, fecha de modificación y un campo llamado deleted que permite un borrado lógico de ese dato.

En estas class representa un modelo de datos de restaurante, por ejemplo:

```typescript
@Entity( name: 'restaurantes')
export class Restaurante {
  public readonly CLASS_NAME : "Restaurante"  = 'Restaurante'

  @PrimaryGeneratedColumn( options: { type: 'bigint', name: 'id' })
  id: number

  @Column( options: {
    type: 'varchar',
    length: 100,
    name: 'nombre',
    nullable: false,
    unique: true,
  })
  nombre: string

  @Column( options: { type: 'varchar', length: 255, name: 'calle' })
  calle: string

  @Column( options: { type: 'varchar', length: 100, name: 'localidad' })
  localidad: string

  @Column( options: { type: 'int', name: 'capacidad' })
  capacidad: number

  @Column( options: { type: 'boolean', name: 'borrado', default: false })
  borrado: boolean = false
```

UUIDs y Autonuméricos:

Las razones por las cuales restaurante, categoría, productos y proveedores utilizan un identificador autonumérico, las colisiones son poco probables al usar Nest 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 y Mappers :

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.

Nuestros DTOs tienen notaciones de la librería class-validator para filtrar e impedir datos incorrectos que nos quieran insertar en la base de datos. En caso de que ocurriera eso se envía un mensaje al cliente mostrando que campo está mal y por qué.

```
1+ usages   Diego Torres Mijarra
export class CreateProductoDto {
  @ApiProperty({example: 'Solomillo'...})
  @IsString()
  @IsNotEmpty()
  @Length( min: 3,  max: 100,  validationOptions: { message: 'El nombre debe tener entre 3 y 100 caracteres' })
  nombre: string

  @ApiProperty({example: 'Solomillo de ternera'...})
  @ApiProperty({description: 'El precio del producto'...})
  @IsNumber()
  @Min( minValue: 0,  validationOptions: { message: 'El precio debe ser mayor que 0' })
  precio: number

  @ApiProperty( options: {
    example: 10,
    description: 'El stock del producto',
    minimum: 0,
  })
  @IsNumber()
  @Min( minValue: 0,  validationOptions: { message: 'El stock debe ser mayor que 0' })
  stock: number

  @ApiProperty({example: 'https://example.com/imagen.jpg'...})
  @IsOptional()
  @IsString()
  imagen?: string
}
```

Repositorio (TypeORM):

Para controlar las bases de datos en PostGreSQL utilizamos TypeOrm, esta nos permite la interacción con la BBDD. Como mencionamos en el punto de modelos nos permite definir las clases y los datos de esta que se insertaran en la tabla en TypeScript.

Simplifica el trabajo con bases de datos relacionales, proporcionando un conjunto de herramientas y una API intuitiva para realizar operaciones CRUD, gestionar relaciones, realizar consultas avanzadas, aplicar migraciones de base de datos. Esto hace que sea más fácil y rápido desarrollar aplicaciones que interactúan con bases de datos PostgreSQL y otras bases de datos relacionales.

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 @Injectable para indicar que es un servicio de Nest y puede ser inyectado en otras dependencias, en nuestro caso todos los servicios son inyectados en el controlador de su respectiva entidad.

Métodos Principales:

- FindAll: este método recupera una página de productos según los parámetros de búsqueda especificados. Utiliza especificaciones de la librería paginate para construir criterios de búsqueda dinámicos. Algunos endpoints tienen opción de enviar los datos paginados y sin paginar.
    - Parámetros: dependen de la clase los que hay en común son
        - 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.
    - 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 el dato a actualizar.

- 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.

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

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

Este servicio 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.

Gestión de Caché:

El servicio implementa la gestión de caché para mejorar el rendimiento y reducir la carga en la base de datos. Se utiliza el módulo de caché de NestJS para almacenar en caché los resultados de ciertas consultas y evitar consultas innecesarias a la base de datos.

El método findAllPaginated utiliza la caché para almacenar los resultados de las consultas paginadas. Antes de realizar la consulta a la base de datos, se comprueba si ya existe un resultado en la caché para los parámetros de búsqueda especificados. Si se encuentra en caché, se devuelve el resultado almacenado en la caché sin necesidad de consultar la base de datos nuevamente. Si no se encuentra en caché, se realiza la consulta a la base de datos y se almacena el resultado en la caché para consultas futuras.

El tiempo de vida de la caché se establece en 60 segundos para asegurar que los datos en caché se actualicen periódicamente y reflejen los cambios más recientes en la base de datos.

Esta estrategia de caché ayuda a mejorar significativamente el rendimiento de la aplicación al reducir la carga en la base de datos y minimizar el tiempo de respuesta de las consultas paginadas.

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 está 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, clientes y proveedores:

Estos dos endpoints tienen un comportamiento similar, en un principio solo los trabajadores tienen un rol de control del programa (User), así 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.

En cuanto al endpoint de proveedores la seguridad y autorización es igual que Trabajadores, las funciones solo pueden ser realizadas por un administrador protegiendo los datos de los proveedores.

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 de los 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 y tiene un guarda para comprobar que el rol existe al crear un usuario.

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),también 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 Bcrypt, una librería que permite que las contraseñas no se pasen a simple vista, las codifican 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 JSON que permite representar pedidos.

Existen dos clases en este endpoint Pedido y ProductosPedidos:

La clase ProductosPedidos 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 Pedido agrupa todos estos ProductosPedidos(productos de un mismo tipo pedidos por el cliente) en una lista.

Los pedidos (clase Pedido) 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.

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 Pedido hay un atributo, precioTotal, 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 , pagado, indicando si está pagado el pedido o no.

Cuando un Producto pasa una lista de ProductosPedidos la almacena y calcula la cantidad total de los productos y el precio total de estos.

Servicio tiene el repositorio 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 environment común. En este podremos variar las características de nuestro programa como los nombres y puertos de las bases de datos, las contraseñas, versiones, etc.

El documento docker-compose-db, es el que se encarga de generar los Dockers de nuestras bases de datos, PostgreSQL y MongoDB, junto con una herramienta de administración (Adminer) 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 bases de datos.

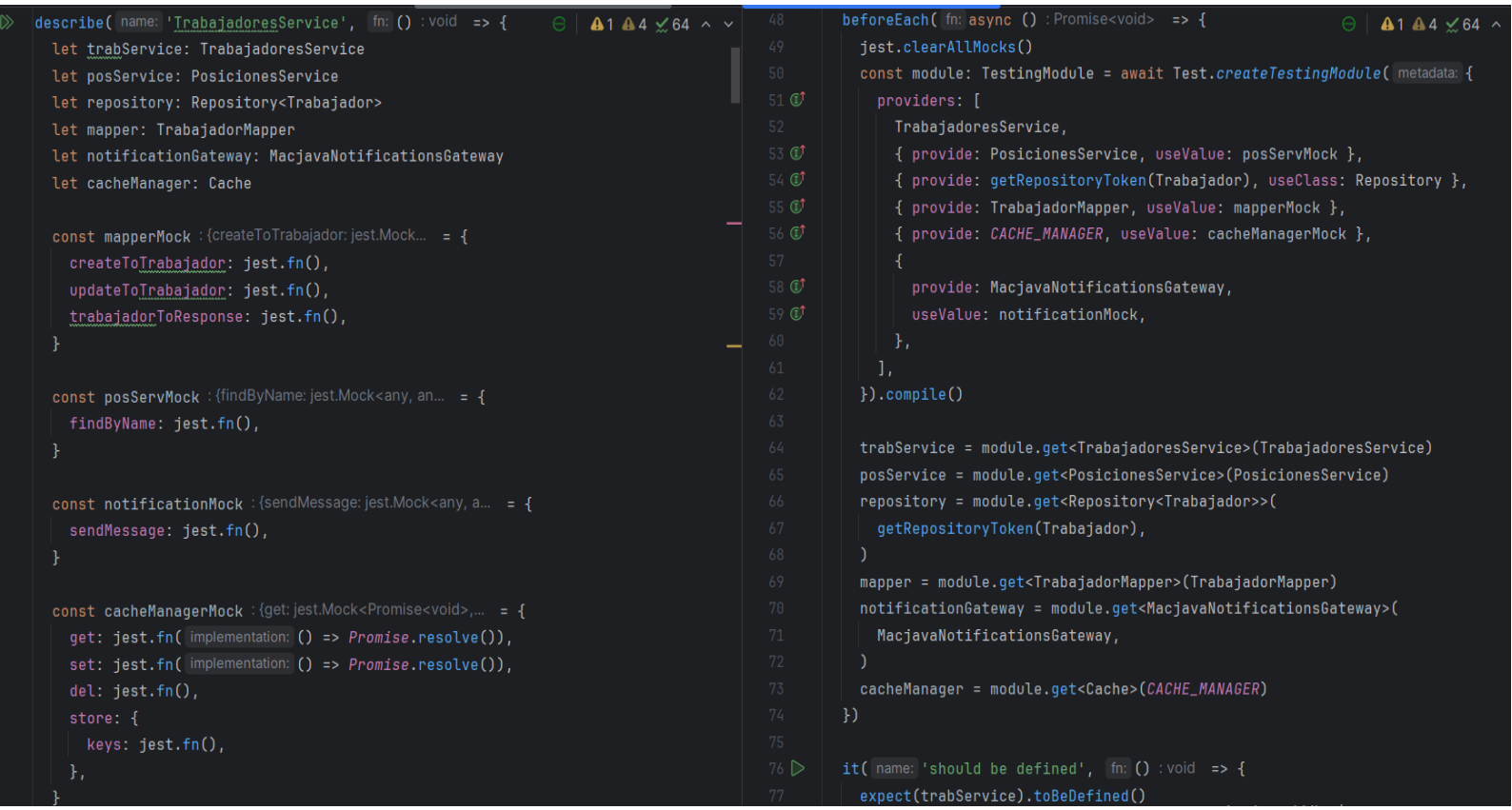
En el docker-compose-app construimos el Docker de nuestra api-rest. Para ello utilizamos el archivo Dockerfile. Definimos el nombre y la gestión de los volúmenes y establecemos el puerto de nuestra app, y la red de comunicación común.

Por último, docker-compose.yml es el archivo que genera los Docker finales para ser distribuida. 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 mockeado estos. Para comprobar hemos usado las funcionalidades de Jest y SuperTest.

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. También 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, también 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:



```
describe( name: 'TrabajadoresService', fn: () : void => {
  let trabService: TrabajadoresService
  let posService: PosicionesService
  let repository: Repository<Trabajador>
  let mapper: TrabajadorMapper
  let notificationGateway: MacjavaNotificationsGateway
  let cacheManager: Cache

  const mapperMock : {createToTrabajador: jest.Mock... = {
    createToTrabajador: jest.fn(),
    updateToTrabajador: jest.fn(),
    trabajadorToResponse: jest.fn(),
  }

  const posServMock : {findByName: jest.Mock<any, an... = {
    findByName: jest.fn(),
  }

  const notificationMock : {sendMessage: jest.Mock<any, a... = {
    sendMessage: jest.fn(),
  }

  const cacheManagerMock : {get: jest.Mock<Promise<void>,... = {
    get: jest.fn( implementation: () => Promise.resolve()),
    set: jest.fn( implementation: () => Promise.resolve()),
    del: jest.fn(),
    store: {
      keys: jest.fn(),
    },
  }
```

```
  beforeEach( fn: async () : Promise<void> => {
    jest.clearAllMocks()
    const module: TestingModule = await Test.createTestingModule( metadata: {
      providers: [
        TrabajadoresService,
        { provide: PosicionesService, useValue: posServMock },
        { provide: getRepositoryToken(Trabajador), useClass: Repository },
        { provide: TrabajadorMapper, useValue: mapperMock },
        { provide: CACHE_MANAGER, useValue: cacheManagerMock },
        {
          provide: MacjavaNotificationsGateway,
          useValue: notificationMock,
        },
      ],
    }).compile()

    trabService = module.get<TrabajadoresService>(TrabajadoresService)
    posService = module.get<PosicionesService>(PosicionesService)
    repository = module.get<Repository<Trabajador>>(
      getRepositoryToken(Trabajador),
    )
    mapper = module.get<TrabajadorMapper>(TrabajadorMapper)
    notificationGateway = module.get<MacjavaNotificationsGateway>(
      MacjavaNotificationsGateway,
    )
    cacheManager = module.get<Cache>(CACHE_MANAGER)
  })

  it( name: 'should be defined', fn: () : void => {
    expect(trabService).toBeDefined()
```

Para los controladores hemos estrategias usado las mismas estrategias:

```
describe( name: 'TrabajadoresController',  fn: () : void  => {
  let controller: TrabajadoresController
  let service: TrabajadoresService

  const serviceMock :  = {
    findAll: jest.fn(),
    findAllPaginated: jest.fn(),
    findById: jest.fn(),
    create: jest.fn(),
    updateById: jest.fn(),
    removeById: jest.fn(),
    softRemoveById: jest.fn(),
  }

  beforeEach( fn: async () : Promise<void>  => {
    const module: TestingModule = await Test.createTestingModule( metadata: {
      imports: [CacheModule.register()],
      controllers: [TrabajadoresController],
      providers: [
        TrabajadoresService,
        {
          provide: TrabajadoresService,
          useValue: serviceMock,
        },
      ],
    }).compile()

    controller = module.get<TrabajadoresController>(TrabajadoresController)
    service = module.get<TrabajadoresService>(TrabajadoresService)
```

```
describe( name: 'findById',  fn: () : void  => {
  it( name: 'should return a category',  fn: async () : Promise<void>  => {
    jest.spyOn(service,  method: 'findById').mockResolvedValue(original)

    const category : Trabajador  = await controller.findById(original.id)

    expect(category).toEqual(original)
  })
  it( name: 'should throw an exception',  fn: async () : Promise<void>  => {
    jest
      .spyOn(service,  method: 'findById')
      .mockRejectedValue(new NotFoundException())

    await expect( actual: () => controller.findById(original.id)).rejects.toThrow(
      NotFoundException,
    )
  })
})
```

14. Conclusión y presupuesto:

Presupuesto del Proyecto MacJava:

-Dominio:

Registro del dominio por un año: 10 €

- Publicidad en Instagram:

Costo estimado de una campaña publicitaria en Instagram: 300 €

-Amazon Web Services (AWS):

Costo estimado por el uso de AWS durante 2 meses: 200 €

-Internet:

Costo mensual del servicio de internet para la oficina: 50 €/mes * 2 meses = 100 €

-Salarios de Trabajadores:

Costo por hora de cada programador: 7,5 €/hora

Horas de trabajo: 5 programadores * 160 horas/mes * 2 meses = 1600 horas

Presupuesto total para los salarios de los programadores: 7,5 €/hora * 1600 horas = 12000 €

-Presupuesto Total:

Sumando todos estos elementos, el presupuesto total para el proyecto MacJava durante 2 meses sería: 10 € (Dominio) + 300 € (Publicidad en Instagram) + 200 € (AWS) + 100 € (Internet) + 12000 € (Salarios de Trabajadores) = 12610 €

Conclusión:

En conclusión, nuestro proyecto MacJava-Nest representa una inversión razonable en términos de recursos financieros y humanos. Con la implementación de una estrategia publicitaria en Instagram, el registro de un dominio, el uso de servicios en la nube de AWS y el trabajo de desarrolladores junior, se lograron los objetivos del proyecto en 2 meses.

La inversión en recursos técnicos y de marketing refleja el compromiso y motivación de nuestro equipo con el potencial y calidad del proyecto.

Nuestro proyecto MacJava tiene el potencial de ser exitoso, proporcionar un buen servicio y generar un retorno positivo de la inversión realizada.