Introducción

El presente informe describe el desarrollo, la arquitectura, las funcionalidades implementadas y los aspectos técnicos del proyecto **Campus Food Sharing API**, una aplicación backend construida con el framework **NestJS**. Este proyecto tiene como propósito ofrecer una solución robusta, modular y escalable que gestione la compra y venta de alimentos dentro de un entorno universitario, garantizando autenticación segura, control de roles, persistencia confiable de datos y pruebas automatizadas que validen la calidad del código.

Integrantes: David Artunduaga, Rony Ordoñez, Juan De la Pava, Jennifer Castro

2. Objetivos del Proyecto

El objetivo principal del proyecto es desarrollar una **API RESTful** que permita gestionar los procesos de compra y venta de alimentos dentro de un campus universitario. Entre los objetivos específicos se destacan:

- Implementar un sistema de autenticación y autorización seguro mediante JWT y roles.
- Gestionar usuarios, productos y pedidos de forma eficiente y modular.
- Garantizar la persistencia de datos en una base de datos relacional.
- Documentar la API utilizando Swagger (OpenAPI).
- Asegurar la calidad del código mediante pruebas unitarias y de integración.
- Implementar integración y despliegue continuo a través de GitHub Actions.

3. Arquitectura y Diseño

La aplicación **Campus Food Sharing API** está diseñada bajo una arquitectura modular siguiendo los principios del framework **NestJS**, que promueve la inyección de dependencias, la separación de responsabilidades y la reutilización de componentes.

El sistema se organiza en módulos independientes: **auth**, **users**, **products** y **orders**, los cuales se comunican entre sí a través de servicios y controladores. Cada módulo cuenta con sus propios **DTOs (Data Transfer Objects)**, entidades y validaciones, asegurando coherencia y mantenibilidad.

El backend se implementó utilizando **TypeScript**, lo que garantiza tipado estático, mayor seguridad en tiempo de compilación y claridad en el desarrollo. Además, se aplicaron principios de **Clean Architecture** y buenas prácticas en la estructuración del código y la organización de los archivos.

4. Tecnologías Utilizadas

El stack tecnológico seleccionado combina herramientas modernas y ampliamente utilizadas en el desarrollo backend:

- **Framework Backend:** NestJS (TypeScript).
- **Base de Datos:** PostgreSQL.
- **ORM:** TypeORM.
- **Autenticación:** JWT y Passport.
- **Pruebas:** Jest y Supertest.
- **Documentación:** Swagger (OpenAPI).
- **Despliegue:** Render Cloud.
- **CI/CD:** GitHub Actions.

Este conjunto de tecnologías permitió crear una API eficiente, escalable y segura, con un flujo de trabajo automatizado que facilita su mantenimiento y evolución.

5. Persistencia de Datos

La persistencia se implementó mediante **TypeORM**, conectado a una base de datos **PostgreSQL**. El archivo typeorm.config.ts define una configuración asíncrona capaz de adaptarse tanto a entornos locales como a producción, soportando conexiones SSL cuando la aplicación está desplegada en la nube.

Durante el desarrollo, se habilitó la opción synchronize: true, que permite la creación automática de tablas a partir de las entidades, acelerando el proceso de construcción y pruebas.

El modelo de datos contempla las siguientes entidades principales: **User**, **Product**, **Order** y **OrderItem**, con relaciones uno a muchos y muchos a uno entre ellas. Las entidades fueron diseñadas para mantener la integridad referencial, soportar

transacciones y permitir consultas optimizadas mediante los repositorios de TypeORM.

6. Autenticación y Autorización

El sistema de autenticación está basado en **JWT (JSON Web Tokens)** y utiliza el módulo **Passport** para la gestión de estrategias de validación. La API implementa guards y decoradores personalizados como @Auth() y @GetUser() para proteger las rutas y obtener información del usuario autenticado.

Los **roles definidos** dentro de la aplicación son buyer, seller y admin.

- El rol **buyer** puede realizar compras y crear órdenes.
- El rol **seller** puede gestionar sus productos y los pedidos relacionados.
- El rol **admin** tiene acceso total al sistema, incluyendo la administración de usuarios y pedidos.

Las rutas están protegidas por guards de autenticación y autorización, garantizando que solo los usuarios con los permisos correspondientes puedan ejecutar las acciones permitidas.

7. Módulos Implementados

El sistema está compuesto por cuatro módulos principales, cada uno con responsabilidades bien definidas:

Auth: Gestiona el registro, inicio de sesión y autenticación mediante JWT. Incluye validación de credenciales, generación de tokens y protección de rutas.

Users: Permite realizar operaciones CRUD sobre los usuarios. Integra la funcionalidad de perfil público de vendedores y el manejo de roles.

Products: Gestiona los productos ofrecidos en el sistema. Los usuarios con rol de vendedor o administrador pueden crear, editar o eliminar productos. Los productos son visibles para cualquier usuario autenticado o público.

Orders: Administra las órdenes de compra. Este módulo contiene la lógica más compleja del sistema, al implementar validaciones de stock, prevención de autoconsumo, cálculo automático del total y manejo de estados. Se utilizan

transacciones con **QueryRunner** para garantizar la integridad de los datos al modificar el inventario o el estado de las órdenes.

8. Funcionalidades Principales

Entre las principales funcionalidades desarrolladas se destacan:

- Registro e inicio de sesión de usuarios con validación de credenciales.
- Creación y administración de productos por parte de los vendedores.
- Creación de pedidos por parte de compradores, con validaciones de stock y propiedad.
- Control de estados de los pedidos, con flujos definidos: pending, accepted, delivered, canceled.
- Cálculo automático del valor total de las órdenes.
- Transacciones atómicas en operaciones críticas.
- Visualización de pedidos por compradores, vendedores y administradores según permisos.

9. Documentación de la API

La API está completamente documentada utilizando **Swagger (OpenAPI)**, lo que facilita su comprensión, uso y prueba desde el navegador. La documentación describe cada endpoint, los parámetros requeridos, las respuestas esperadas y los posibles códigos de error.

En el entorno de despliegue, la documentación está disponible en: https://cfs-api.onrender.com/api-docs.

Esta interfaz permite probar los endpoints de manera interactiva, revisar los modelos de datos y comprender el comportamiento de cada módulo.

10. Pruebas e Integración Continua

El proyecto cuenta con una suite de pruebas implementada con **Jest** y **Supertest**, que evalúa tanto la lógica de negocio como los controladores de cada módulo. Las

pruebas abarcan escenarios positivos y negativos, asegurando que las validaciones y transacciones funcionen correctamente.

La cobertura actual supera el **80%** del código fuente. Adicionalmente, se configuró un flujo de **integración continua con GitHub Actions**, que ejecuta automáticamente las pruebas cada vez que se realiza un push al repositorio. Esta configuración permite detectar errores tempranamente y garantiza que el código desplegado sea estable y funcional.

11. Despliegue

El despliegue se realizó en la plataforma **Render Cloud**, utilizando una base de datos **PostgreSQL** alojada en la nube. Se configuraron las variables de entorno necesarias para la conexión segura a la base de datos y la generación de tokens JWT.

El pipeline de CI/CD ejecuta las pruebas antes de cada despliegue, lo que asegura la integridad del sistema. En caso de que alguna prueba falle, el despliegue no se realiza hasta corregir el error, reforzando así la confiabilidad del proceso.

12. Consideraciones Técnicas

El sistema está desarrollado bajo principios de modularidad, escalabilidad y mantenibilidad. Se emplearon DTOs validados con class-validator para asegurar la integridad de los datos de entrada y se documentaron todos los endpoints con anotaciones de Swagger.

Las operaciones críticas que involucran múltiples entidades, como la creación o cancelación de órdenes, se ejecutan mediante transacciones para evitar inconsistencias. Además, se implementó un endpoint de carga inicial (seed) que permite insertar un usuario administrador y productos de ejemplo para pruebas y demostraciones.