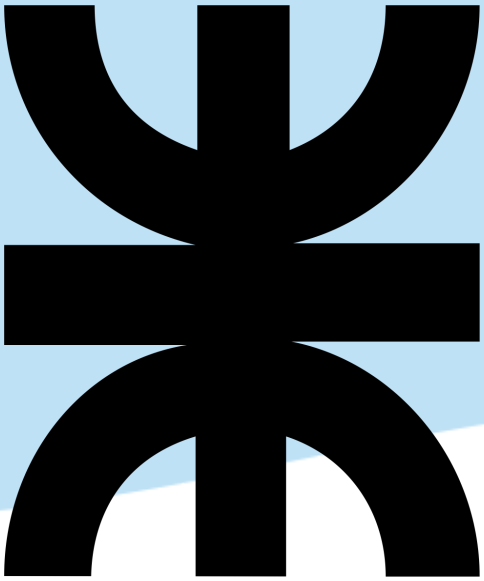


Universidad Tecnológica Nacional FRRe

Ingeniería en Sistemas de Información

Trabajo Práctico Integrador

Desarrollo de Software

Grupo 3 - Big Brain

Profesores:

Ing. Fernandez, Jose A.

Ing. Villaverde, Jorge E.

Integrantes:

Casano, Julieta - julicasano2626@gmail.com - 28740

Duarte, Tomás - tomiduarte2801@gmail.com - 27913

Gorrochategui, Josefina - josefinagorro@gmail.com - 28553

Kozak Riehme, Paula - paukozakr@gmail.com - 28646

Lima, Matias Ezequiel - matiaselima@hotmail.com - 28765

Orban, Gonzalo Tomás - gonzaorban@gmail.com - 28427

Rodriguez Leiva, Juan Ignacio - juanignaciorodriguezleiva5@gmail.com - 28733

Rojas, Alejo Ivan - alejoivanrojas@gmail.com - 28733

Sotelo, Maria Celina - sotelocelina45@gmail.com - 27961

Zapata, Jerónimo - jerozapata@gmail.com - 28386

Ciclo lectivo 2025

Índice

Introducción

En este trabajo presentamos el desarrollo del subsistema de **Transporte, Logística y Seguimiento**, dentro del ecosistema de compras planteado para la materia. Nuestro rol en este proyecto es cerrar el ciclo de la venta, gestionando la entrega física de los productos desde el vendedor hasta el cliente final.

Organización del equipo

Para optimizar el desarrollo del proyecto, se decidió estructurar al equipo en dos subgrupos especializados: **Frontend** y **Backend**. Esta decisión se tomó evaluando las habilidades y preferencias técnicas de cada integrante, asegurando así que cada uno trabajara en el área donde se sentía más cómodo y productivo. Esto nos permitió un desarrollo paralelo, optimizando el tiempo y cumpliendo con los plazos establecidos. A pesar de la división de tareas, mantuvimos una comunicación mutua y constante, lo que garantizó la coherencia del proyecto y facilitó la toma de decisiones conjuntas ante cualquier imprevisto.

Herramientas y Tecnologías

TypeScript: Decidimos utilizar TypeScript porque NestJS y TypeORM son frameworks diseñados nativamente para este lenguaje, lo que nos permite implementar una arquitectura robusta y escalable. Además, dada la naturaleza crítica del cálculo de costos logísticos y la gestión de estados de envíos descrita en el problema, el tipado estático nos previene de errores aritméticos y lógicos graves. Finalmente, al usar también Next.js, compartimos interfaces de datos entre backend y frontend, asegurando que la información de los pedidos viaje íntegra y sin errores de formato.

Frontend

- **Framework Next.js:** Entorno basado en React que permite crear interfaces modernas, rápidas y con renderizado optimizado, mejorando tanto el rendimiento como la experiencia del usuario.
- **Tailwind CSS:** Framework de estilos utilitario que permite construir interfaces con gran rapidez mediante clases predefinidas y altamente personalizables. Facilita mantener un

diseño consistente sin necesidad de escribir hojas de estilo extensas, optimizando el desarrollo frontend y permitiendo una gran flexibilidad visual.

Backend

- **Entorno Node.js:** Plataforma de ejecución basada en JavaScript utilizada para levantar el servidor y manejar operaciones de forma eficiente gracias a su arquitectura orientada a eventos.
- **Framework NestJS:** Framework estructurado que permite construir aplicaciones del lado del servidor con una arquitectura modular, mantenible y escalable, facilitando la organización del código.
- **Gestor ORM TypeORM:** Herramienta que simplifica la interacción con la base de datos mediante el uso de entidades y repositorios, evitando escribir consultas SQL manuales y facilitando la consistencia de los datos.

Infraestructura y despliegue

- **Railway:** Servicio en la nube utilizado para desplegar y administrar la infraestructura del proyecto, permitiendo alojar tanto el backend como la base de datos de manera centralizada y sencilla.
- **Vercel:** Plataforma orientada al despliegue de aplicaciones frontend, especialmente optimizada para proyectos en Next.js, permitiendo actualizaciones rápidas y automatizadas.
- **Docker:** Tecnología que encapsula la aplicación y sus dependencias en contenedores, garantizando que se ejecute de la misma forma en desarrollo y producción.
- **GitHub Actions:** Servicio de CI/CD que permite ejecutar flujos automatizados para compilar, testear y desplegar el proyecto, garantizando mayor calidad y coherencia en cada entrega.

Gestión y colaboración

- **GitHub:** Plataforma utilizada para el control de versiones y la colaboración entre miembros del equipo, permitiendo almacenar el código, gestionar cambios y revisar contribuciones.
    - Se adoptó el estándar **Conventional Commits** para garantizar la trazabilidad y legibilidad del historial de versiones. Mediante el uso de prefijos semánticos (como `feat`, `fix`, etc.), logramos estandarizar los mensajes de confirmación, lo que permite identificar de inmediato el propósito de cada cambio. Esta práctica no solo agiliza las revisiones de código (*code reviews*), sino que también facilita el mantenimiento evolutivo y la generación automática de documentación.
- **Trello:** Herramienta visual de gestión de proyectos que permite organizar tareas mediante tableros, listas y tarjetas, facilitando la planificación y seguimiento del progreso del equipo.

Autenticación

- **Keycloak:** software de código abierto que actúa como un gestor de identidades y accesos (IAM), permitiendo a los usuarios iniciar sesión una sola vez (inicio de sesión único o SSO) para acceder a múltiples aplicaciones y servicios. Simplifica la seguridad al centralizar la autenticación, reduciendo el trabajo de los desarrolladores al evitar que tengan que crear su propio sistema de registro y autenticación desde cero.

Frontend

Decisiones de Diseño

El desarrollo de la interfaz combinó **decisiones de arquitectura** y decisiones de diseño que permitieron construir una aplicación clara y funcional para el personal interno encargado de la gestión logística. Desde el punto de vista arquitectónico, la aplicación se desarrolló utilizando **Next.js** con **TypeScript**, lo que permitió trabajar con una SPA de navegación fluida y componentes reutilizables. Para los estilos se empleó **Tailwind CSS**, que facilitó la consistencia visual y el desarrollo modular.
A nivel de organización interna, el proyecto se estructuró en carpetas bien definidas —como **src** para el código principal, **public** para los recursos estáticos y **cypress** para las pruebas— junto con los archivos de configuración necesarios para el entorno, dependencias y compilación. Esta estructura ordenada favorece el mantenimiento, la escalabilidad y la claridad del proyecto.

En cuanto a las **decisiones de diseño**, la interfaz se construyó pensando en las tareas habituales del personal operativo del sistema. La **Landing Page** funciona como punto de entrada y se diseñó con un enfoque simple, incorporando un hero banner que contextualiza el entorno logístico y tarjetas que destacan las funciones principales.
Las pantallas de **Calcular Costo** y **Crear Envío** se desarrollaron con formularios claros y bien organizados, acompañados de mensajes de validación que ayudan al usuario a corregir errores antes de enviar los datos. En la sección de **Consultar Envío** se priorizó la lectura rápida y la interacción directa: se muestra un listado de envíos con su estado actual y se ofrece la posibilidad de buscar por ID. Al seleccionar un envío, se despliega su información completa y se habilita la opción de **actualizar el estado**, mostrando únicamente las transiciones válidas para mantener coherencia y evitar elecciones incorrectas. Estas validaciones y restricciones forman parte del diseño, orientado a guiar al usuario y asegurar un flujo de trabajo claro.

Conexión con el back

La comunicación entre el frontend y el backend se realiza mediante solicitudes HTTP utilizando **fetch**, trabajando en todo momento con datos en formato JSON. Para mantener la organización del código, se implementó una capa de servicios donde se centralizan las operaciones que

interactúan con el backend, como calcular costos, crear envíos, consultar información o actualizar estados.

Cuando una operación lo requiere, el frontend agrega el **token de autenticación** en los headers para acceder a las rutas protegidas. Además, se manejan las respuestas de error del servidor, traduciéndolas en mensajes claros para el usuario. Esta estructura permite mantener las peticiones tipadas, reutilizables y coherentes con los formatos esperados por el backend.

AUTH

Para la gestión de identidad y control de acceso en el frontend, se implementó un sistema de autenticación basado en Keycloak, un proveedor de identidad estandarizado que permite administrar sesiones de usuario mediante los protocolos OAuth2 y OpenID Connect. El frontend integra este servicio a través de un AuthContext desarrollado en React, el cual centraliza toda la lógica asociada a la autenticación, el manejo de tokens y la persistencia de la sesión dentro de la aplicación.

La inicialización del proceso de autenticación ocurre al momento de cargar la aplicación, cuando el AuthContext instancia el cliente de Keycloak utilizando los parámetros de configuración definidos en variables de entorno (URL del servidor, realm y clientId). A partir de esta configuración, se ejecuta el método keycloak.init(), encargado de verificar si el usuario posee una sesión activa mediante la estrategia check-sso. Esta modalidad permite que, si el usuario ya está autenticado en Keycloak, la sesión se recupere automáticamente sin necesidad de redirigir nuevamente al inicio de sesión.

Una vez validada la sesión, el AuthContext almacena el token de acceso provisto por Keycloak y actualiza el estado interno de autenticación de la aplicación, exponiendo esta información a través de su contexto global. Este token será posteriormente utilizado por el frontend para autorizar peticiones hacia el backend, agregándolo en los encabezados HTTP cuando la operación lo requiera. De esta manera, cualquier componente o página puede determinar si el usuario está autenticado mediante la propiedad isAuthenticated, o acceder directamente al token mediante token.

Finalmente, se implementa un mecanismo automático de renovación del token. Keycloak notifica al frontend cuando el token está próximo a expirar, disparando la función updateToken(), que solicita un nuevo token al servidor y actualiza su valor en el AuthContext. Este proceso garantiza la continuidad de la sesión sin interrupciones, evitando que el usuario sea deslogueado durante el uso normal de la aplicación.

El sistema también provee funciones de autenticación explícitas:

- login(): redirige al usuario a la página de inicio de sesión de Keycloak.
- logout(): finaliza la sesión y redirige al usuario al punto de salida configurado.

Toda esta estructura permite mantener un flujo de autenticación seguro, centralizado y transparente, asegurando que las rutas protegidas del frontend solo sean accesibles para usuarios autenticados y que todas las comunicaciones con el backend incluyan el token requerido de manera automática.

Tests

Para garantizar la robustez, confiabilidad y correcta experiencia de usuario en la aplicación web, el equipo implementó una estrategia integral de testing orientada a validar tanto los flujos principales de la interfaz como el funcionamiento individual de componentes específicos. Esta estrategia se dividió en dos niveles fundamentales: **pruebas End-to-End (E2E)** y **pruebas unitarias**, utilizando **Cypress** como framework principal de automatización para los tests E2E y su módulo integrado para pruebas unitarias.

Las pruebas E2E fueron diseñadas con el objetivo de reproducir el comportamiento de un usuario real interactuando con la aplicación. De esta manera, se validan no solo las funcionalidades visibles, sino también la integración entre pantallas, la navegación, las restricciones de campos, la estructura de datos ingresados y la respuesta de la interfaz ante diferentes escenarios. Cada test se ejecuta en un entorno aislado, desacoplado del backend productivo, lo que permite validar únicamente la lógica funcional del frontend sin dependencias externas. Estos tests se integran posteriormente al flujo de CI/CD, asegurando su ejecución automática ante cada cambio relevante en el código.

Como requisito previo, todos los tests comienzan ejecutando el proceso de inicio de sesión con un usuario válido, ya que este paso permite verificar la comunicación entre el frontend y el backend. Durante esta autenticación, Cypress envía las credenciales al servidor, el cual responde con el token o la sesión correspondiente, confirmando que el sistema backend se encuentra funcionando correctamente. Solo una vez obtenida la sesión activa, cada test continúa con el flujo específico que debe validar. Este mecanismo asegura que las pruebas se realicen siempre bajo condiciones reales de uso, reflejando fielmente la experiencia de un usuario autenticado dentro de la aplicación.

Backend

Decisiones de diseño

Fundamentos Tecnológicos y Patrones

Para el desarrollo del microservicio de logística, la elección de NestJS como framework principal fue fundamental debido a su arquitectura modular y su soporte nativo para TypeScript, lo que garantiza un tipado estricto y minimiza errores en tiempo de ejecución. En cuanto a la estructura del sistema, se realizó una transición desde un diseño monolítico hacia una Arquitectura Limpia (Clean Architecture) modificada. Este cambio permitió aislar el núcleo de la lógica de negocios de las capas externas, asegurando que la presentación de los datos sea consistente y que la lógica del dominio permanezca independiente de la infraestructura. La arquitectura resultante se divide en tres capas principales: una Capa de Dominio con entidades puras de TypeScript que encapsulan las reglas de negocio; una Capa de Infraestructura que implementa la persistencia utilizando TypeORM para definir tablas y relaciones SQL; y una Capa de Aplicación donde los servicios orquestan el flujo de datos entre controladores y repositorios.

Desacoplamiento y Persistencia

Para garantizar la flexibilidad y mantenibilidad del sistema, se utilizó la Inyección de Dependencias (DI), desacoplando los servicios de las implementaciones concretas de los repositorios. Esto facilita, por ejemplo, la sustitución del motor de base de datos MySQL en el futuro sin afectar la lógica de negocio, siguiendo el patrón de Repository Interface. Asimismo, la elección de TypeORM como ORM fue estratégica para gestionar relaciones complejas entre entidades, como la relación "uno a muchos" entre envíos y productos, sin necesidad de escribir consultas SQL manuales, lo que agilizó significativamente el proceso de desarrollo.

Calidad de Datos, Seguridad y Robustez

La integridad y seguridad de la información se abordaron mediante múltiples mecanismos. Se implementaron Data Transfer Objects (DTOs) junto con la librería **class-validator** para asegurar que los datos entrantes cumplan con el formato esperado antes de ser procesados por la lógica de negocio. Además, se incorporó un filtro global de excepciones (Global

Exception Filter) que captura y estandariza los errores del sistema, devolviendo respuestas HTTP coherentes y ocultando detalles técnicos sensibles. En términos de seguridad, la protección de los endpoints se delegó a Guards personalizados integrados con Keycloak, garantizando que solo las peticiones con un token válido puedan acceder a los recursos protegidos.

Estandarización de Infraestructura

Para resolver las discrepancias entre los entornos de desarrollo y producción, se implementó una estrategia de dockerización integral. La base de datos se ejecuta en un contenedor aislado, asegurando que todo el equipo trabaje con la misma configuración de MySQL. Además, se configuraron controles de salud (Healthchecks) nativos en Docker Compose para sincronizar el arranque de los servicios, evitando condiciones de carrera y asegurando que la aplicación y los scripts de carga de datos solo se ejecuten cuando la base de datos esté plenamente operativa.

En cuanto al despliegue, se adoptó una estrategia basada en contenedores inmutables mediante un pipeline de CI/CD automatizado con GitHub Actions. Este flujo de trabajo se activa con cada cambio en la rama principal y consta de tres etapas: construcción de la imagen Docker utilizando una estrategia multi-stage para optimizar el tamaño final; publicación de la imagen en el GitHub Container Registry (GHCR) utilizando credenciales seguras; y finalmente, el despliegue automático en Railway mediante un webhook que actualiza el servicio con la nueva versión.

La gestión de la configuración y los secretos sigue una estricta separación de entornos. Las credenciales de tránsito, como tokens de registro, se inyectan en tiempo de compilación mediante GitHub Secrets, mientras que las variables operativas, como las credenciales de base de datos, se configuran en el panel de Railway y se inyectan en el contenedor en tiempo de ejecución, asegurando que ninguna información sensible quede expuesta en el repositorio. Finalmente, la infraestructura en la nube se organiza en una red privada donde el microservicio de logística y la base de datos se comunican internamente, y el acceso público se gestiona a través de un API Gateway (Nginx) que centraliza el tráfico y termina las conexiones SSL, ocultando la topología interna del sistema.

AUTH (Keycloack)

La cátedra decidió delegar la gestión de identidades a **Keycloak** en lugar de implementar un sistema de autenticación propio (JWT manual). Esta decisión de diseño se fundamenta en:

- Estandarización: Keycloak implementa protocolos estándar de la industria como OAuth2 y OpenID Connect (OIDC).
- Seguridad: Reduce la superficie de ataque al no gestionar contraseñas ni sesiones directamente en nuestra base de datos operativa.
- Escalabilidad: Permite la implementación de Single Sign-On (SSO) entre los distintos microservicios del ecosistema (Logística, Compras, Stock) sin duplicar lógica de usuarios."

API

Funcionamiento de la API

La API de logística está desarrollada con NestJS y TypeScript, implementando una arquitectura modular y escalable. El controlador principal **ShippingController** expone los endpoints REST necesarios para la gestión integral de envíos.

Arquitectura de Servicios

El controlador actúa como punto de entrada, delegando la lógica de negocio a **ShippingService**, que coordina servicios auxiliares como *CostCalculatorService* para cálculos tarifarios, **StockApiService** para integración con servicios externos de inventario, y repositorios personalizados que manejan la persistencia de datos.

Endpoints y Funcionalidades

Gestión de Envíos:

- **POST /shipping**: Crea nuevas órdenes de envío validando *CreateShippmentRequestDto*. El flujo incluye: validación de datos del destinatario (nombre, dirección, CPA), consulta automática al servicio de stock externo mediante

token JWT para obtener información detallada de productos (peso, dimensiones, ubicación en almacén), generación automática de número de tracking único, y asignación inicial del estado "En preparación". El sistema valida que los productos existan en stock antes de procesar el envío.

- **GET /shipping:** Lista todos los envíos con soporte de paginación mediante *PaginationInDto* (parámetros page e items_per_page). Retorna *ShippingListResponseDto* con metadata de paginación (total de páginas, elementos totales) y un array de envíos con información resumida.

- **GET /shipping/:id:** Recupera detalles completos de un envío específico, retornando *ShippingDetailsResponseDto* que incluye: información del destinatario, productos incluidos, historial completo de estados con timestamps, método de transporte seleccionado, costos detallados, y número de tracking. Lanza *ShippingIdNotFoundException* si el ID no existe.

Cálculo de Costos:

- **POST /shipping/cost:** Endpoint crítico que calcula estimaciones de costo antes de crear el envío. Recibe *CostCalculationRequestDto* con lista de productos, código postal origen y destino. Lo envía a *CostCalculatorService* implementa un algoritmo sofisticado que retorna *CostCalculationResponseDto* con desglose completo: costo primer tramo, segundo tramo, peso total, peso volumétrico, y costo total final.

Gestión de Estados:

- **PATCH /shipping/:id/status**: Actualiza el estado de un envío existente recibiendo *ShippingStatementLogsRequestDto* que contiene el nuevo estado y observaciones opcionales. Registra cada cambio en una tabla de logs con timestamp automático, permitiendo trazabilidad completa del envío.

- **POST /shipping/:id/cancel**: Endpoint especializado para cancelación de envíos. Valida que el envío esté en un estado cancelable. Retorna *CancelShippingResponseDto* confirmando la cancelación y actualizando automáticamente el estado a "Cancelado" con timestamp.

Métodos de Transporte:

- **GET /shipping/transport-methods**: Endpoint público (@Public()) que retorna *TransportMethodsResponseDto* con todos los métodos de transporte disponibles en el sistema. No requiere autenticación porque consideramos que el método no maneja información delicada que necesite ser protegida.

Seguridad y Control de Acceso

La API implementa una capa de seguridad robusta mediante integración con Keycloak como Identity Provider. El módulo **KeycloakModule** gestiona toda la autenticación OAuth2.0 y validación de tokens JWT.

Autenticación y Autorización:

Cada endpoint protegido utiliza el decorador @Scopes() que verifica permisos específicos en el token JWT:

- @Scopes('envios:read'): Para consultas (GET)
- @Scopes('envios:write'): Para modificaciones (POST, PATCH)

El sistema extrae automáticamente el token del header Authorization y lo propaga a servicios externos cuando es necesario (como llamadas a la API de Stock). Los guards *KeycloakAuthGuard* y *KeycloakResourceGuard* interceptan cada request validando la firma del token, verificando expiración, y comprobando que el usuario posea los scopes necesarios.

Validación y Manejo de Errores

Se implementó **ContextValidationPipe**, un pipe personalizado que transforma errores de validación de class-validator en excepciones de negocio específicas. Cada endpoint declara su excepción contextual esperada, como por ejemplo:

- InvalidShippingOrderException: Datos inválidos al crear envío
- InvalidCostCalculationException: Error en cálculo de costos
- ShippingIdNotFoundException: ID inexistente
- ShippingIdNonCancellableException: Intento de cancelar envío no cancelable

El **GlobalExceptionFilter** captura todas las excepciones y las normaliza en respuestas HTTP consistentes con formato JSON estandarizado, facilitando el consumo desde clientes.

API GATEWAY

Funcionamiento del API Gateway

El API Gateway está implementado con **NGINX**, funcionando como punto de entrada único y centralizado para todas las peticiones del sistema. Actúa como proxy inverso que enruta el tráfico hacia los microservicios correspondientes, proporcionando una capa de abstracción entre clientes y servicios backend.

Arquitectura y Configuración

El gateway está containerizado mediante **Docker** utilizando la imagen oficial de NGINX. La configuración personalizada (*nginx.conf*) define el comportamiento del servidor, estableciendo workers con escalabilidad automática según recursos disponibles.

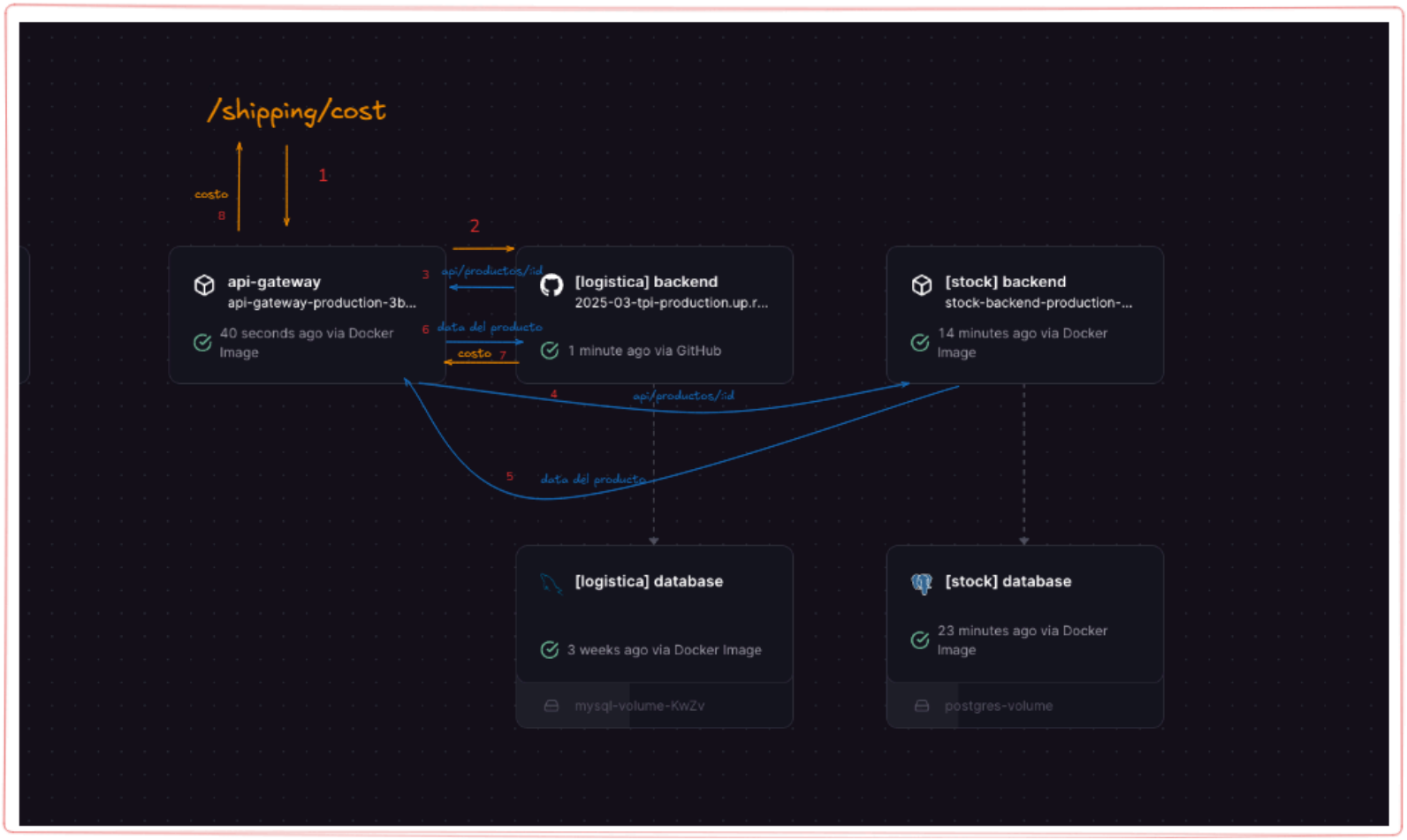
Upstreams y Enrutamiento

El sistema define tres upstream groups que representan los microservicios del ecosistema:

- compras_service: Gestiona operaciones de órdenes de compra y proveedores (puerto 8081)
- stock_service: Maneja inventario, consultas de productos y proporciona datos de dimensiones/pesos (puerto 3000)
- logistica_service: Administra el ciclo completo de envíos desarrollado por el equipo (puerto 3010)

Cada servicio es accesible mediante rutas específicas (*/compras, /api/, /shipping*) que el gateway mapea automáticamente a los contenedores backend correspondientes.

Flujo de las request con API GATEWAY



Configuración de Proxy

El gateway implementa configuraciones críticas de forwarding que preservan información del cliente original:

- Headers personalizados (*Host, X-Real-IP, X-Forwarded-For*) que mantienen trazabilidad completa de las peticiones, permitiendo a los servicios backend identificar el origen real de cada request para auditoría y logs detallados.
- Timeouts configurados con 5 segundos para establecer conexión y 30 segundos para recibir respuesta, balanceando responsividad del sistema con tolerancia a operaciones complejas que requieren procesamiento.

Ventajas de la Arquitectura

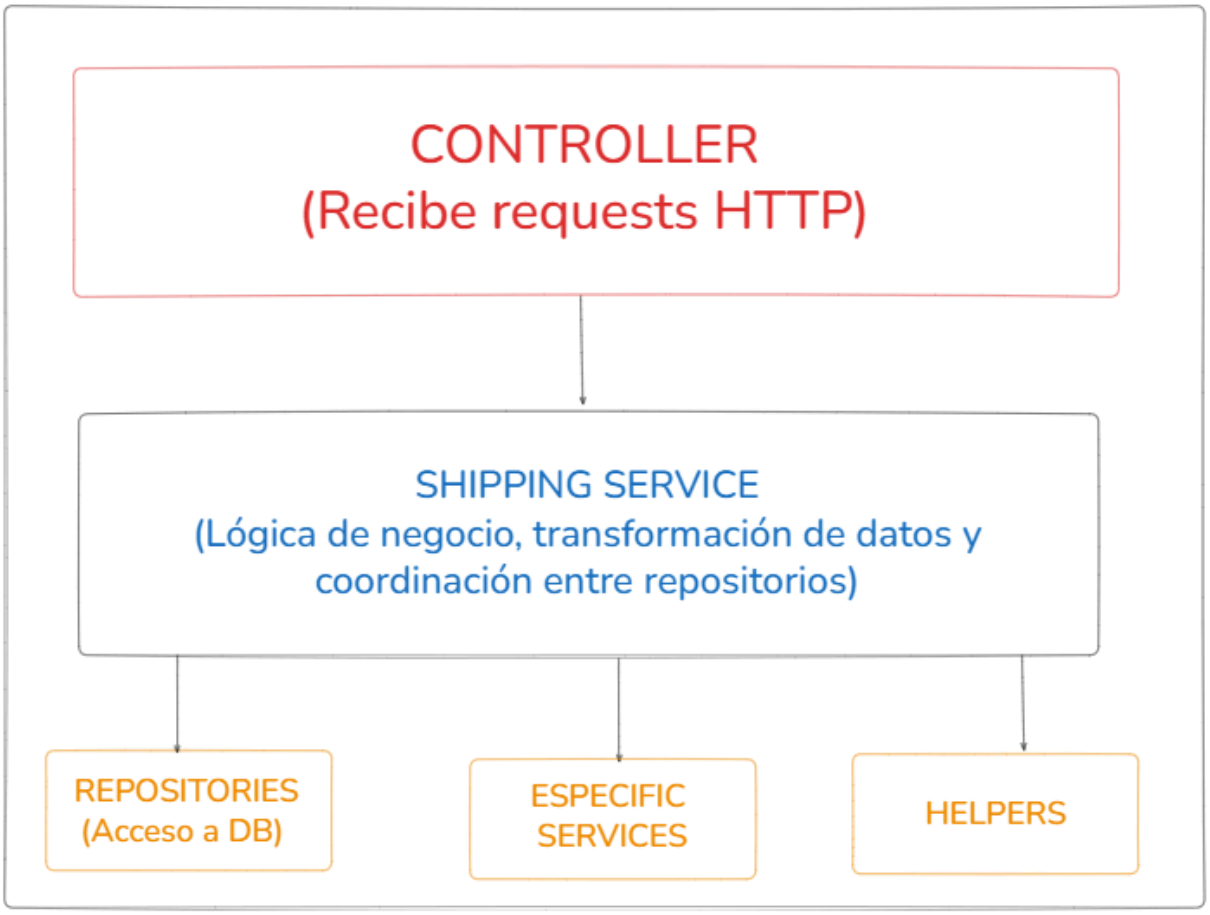
- Centralización: Único punto de entrada (puerto 80) simplificando configuración de clientes y permitiendo cambios en infraestructura backend sin afectar consumidores.
- Seguridad: Oculta topología interna, los servicios no son accesibles directamente desde el exterior.
- Escalabilidad: Soporta fácilmente balanceo de carga agregando múltiples instancias por servicio sin modificar clientes. Cada microservicio puede escalar independientemente según demanda (No está implementado en nuestro trabajo pero la API Gateway nos da esta posibilidad).

Service

¿Qué es la capa de Services?

La capa de Services en NestJS representa el núcleo de la lógica de negocio de la aplicación. Es donde se implementan las reglas, validaciones y procesos que dan sentido a los datos y operaciones del sistema. Los servicios actúan como intermediarios entre los controllers (que manejan las peticiones HTTP) y los repositories (que acceden a la base de datos).

Aplicar esta organización permite cumplir principios de diseño importantes en la realización de un proyecto como el de Responsabilidad Única e Inyección de Dependencias (mediante el sistema de DI de NestJS)



Services Implementados

El servicio principal es **Shipping Service** cuya responsabilidad principal es gestionar el ciclo de vida de todos los envíos. En el se implementan las funcionalidades que justamente controlan los flujos de este ciclo de vida, entre ellas, las más resaltables son:

a) Creación de envíos

Esta funcionalidad es una de las más fundamentales porque contiene todos los datos de la instancia del objeto sobre el que gira después todas las demás articulaciones. A destacar entonces en torno a esto, es que la persistencia de datos que almacena se realiza en cascada, asegurando que todas las entidades relacionadas se guarden correctamente. El servicio crea el registro principal del envío y luego procesa cada producto incluido en la orden. Para cada producto, se verifica si ya existe en el catálogo; si no existe, se crea un nuevo registro. Posteriormente, se establecen las relaciones entre el envío y los productos a través de una tabla intermedia que almacena también la cantidad de cada ítem.

Finalmente, el sistema genera un log inicial que registra la creación del envío con estado CREATED. Este log marca el inicio del historial de trazabilidad del envío y queda almacenado con un timestamp que indica el momento exacto de creación.

Importancia arquitectónica:

Esta funcionalidad demuestra el valor de la capa de servicios al manejar una transacción compleja que involucra múltiples entidades (usuario, direcciones, envío, productos, logs). El servicio garantiza que todas las operaciones se completen exitosamente o, en caso de error, ninguna persista (principio de atomicidad).

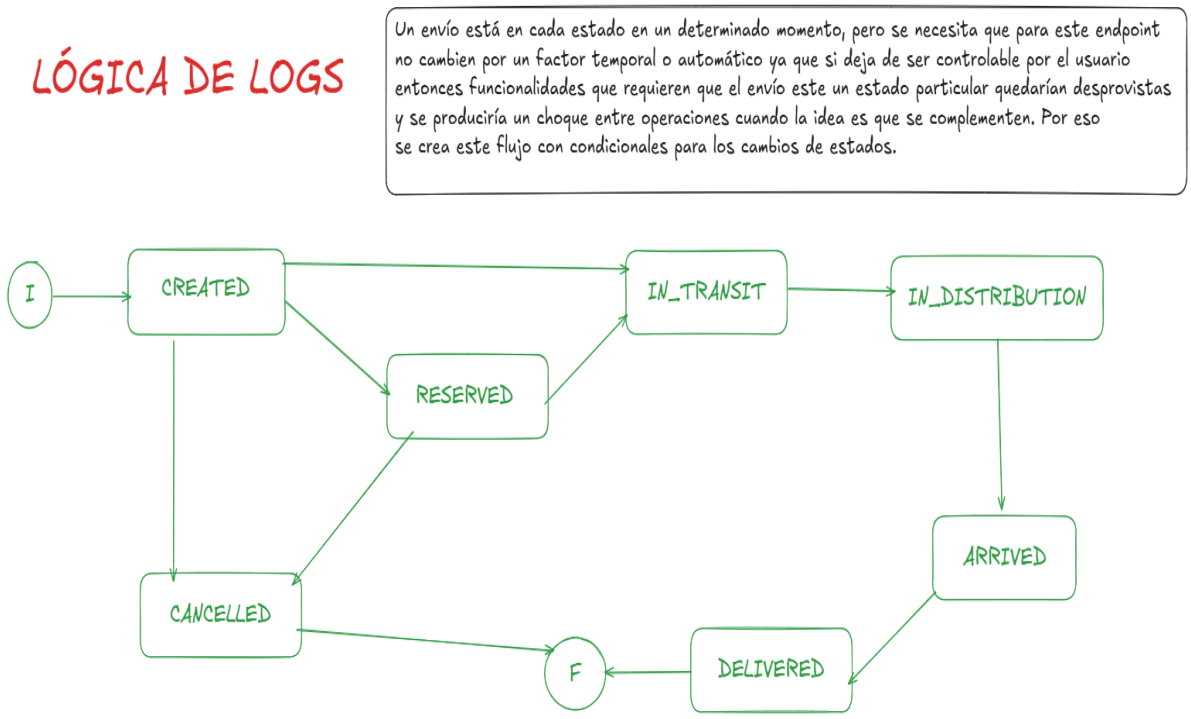
b) Actualización de estado con sistema de logs

El seguimiento del estado de un envío es una de las funcionalidades más críticas del sistema, ya que permite conocer en qué punto del proceso logístico se encuentra cada paquete. Implementamos un sistema robusto que combina validaciones de negocio con un historial completo de cambios.

Proceso de actualización:

Cuando se solicita actualizar el estado de un envío, el primer paso es verificar que el envío exista en el sistema, de ahí la relevancia de la funcionalidad anterior.

El corazón de esta funcionalidad es la validación de transiciones de estado. Implementamos un sistema que define explícitamente qué cambios de estado son válidos según el flujo logístico normal. Para ello se armó un diagrama y se justificó porque el cambio manual y con condicionales era la opción más óptima a diferencia de factores temporales.



Para gestionar estas reglas, utilizamos un componente auxiliar llamado ShippingStatusTransitionHelper que centraliza toda la lógica de transiciones. Este helper define un mapa de estados permitidos que refleja el flujo real de un envío. Cualquier intento de saltar estados o retroceder en el flujo es rechazado automáticamente.

Cuando una transición es válida, el servicio ejecuta dos operaciones de manera automática: actualiza el estado del envío en la base de datos y crea un nuevo registro en el historial de logs. Este registro incluye el timestamp exacto del cambio, el nuevo estado alcanzado y un mensaje descriptivo (que puede ser personalizado o generado automáticamente).

La validación estricta de transiciones evita inconsistencias en los datos que podrían confundir tanto al personal logístico como a los clientes. Y la información de estados permitidos mejora la interfaz de usuario, mostrando solo acciones realmente ejecutables.

c) Cancelación de envíos

Lógica de negocio aplicada:

Implementamos reglas claras sobre cuándo un envío puede ser cancelado. Solo permitimos la cancelación cuando el envío está en estado CREATED o RESERVED. La razón es que en estos estados aún no se han iniciado procesos físicos de movilización de mercancía.

Diferenciación con actualización normal de estado:

Es importante destacar que la cancelación no se maneja como una simple actualización de estado. Aunque técnicamente podríamos considerar CANCELLED como otro estado más, lo tratamos como una acción de negocio independiente por varias razones:

Primero, tiene validaciones específicas que no aplican a otras transiciones. Segundo, puede tener efectos secundarios especiales como notificaciones al cliente, reversión de reservas de stock, o ajustes financieros. Tercero, separarla permite tener un control más granular de permisos (tal vez algunos usuarios pueden actualizar estados pero no cancelar envíos). Y finalmente, mantiene el endpoint de actualización de estados más simple y enfocado en el flujo normal.

d) Cálculo de costos de envío

El cálculo de costos es quizás la funcionalidad más compleja del servicio, ya que requiere integración con servicios externos y aplica un modelo logístico sofisticado.

Integración con el servicio de Stock:

El proceso comienza obteniendo información detallada de cada producto incluido en el envío. Esta información no reside en nuestra base de datos, sino que se consulta dinámicamente al servicio de Stock mediante peticiones HTTP autenticadas.

Implementamos un enfoque eficiente para esta integración: en lugar de consultar los productos uno por uno secuencialmente, lanzamos todas las peticiones en paralelo utilizando Promise.all. Esto reduce significativamente el tiempo de respuesta, especialmente cuando el envío incluye múltiples productos.

Delegación al servicio especializado:

Una vez que tenemos todos los datos necesarios obtenidos de la respuesta de stock, el ShippingService delega el cálculo real al módulo de CostCalculatorService.

CostCalculatorService (Servicio especializado)

Su responsabilidad es calcular costos de envío basados en la zonificación geográfica y características físicas de la carga.

Este servicio implementa un modelo logístico de dos tramos que refleja cómo opera realmente una red de distribución.

Modelo logístico de dos tramos

El sistema reconoce que en la mayoría de los casos, los productos no se envían directamente desde el almacén al cliente final. En su lugar, pasan por un centro de distribución intermedio.

Primer tramo - Consolidación:

Los productos pueden estar almacenados en diferentes ubicaciones geográficas. Imaginemos un escenario donde un cliente ordena tres productos: uno está en un almacén en Mendoza, otro en Córdoba, y un tercero en Rosario. Todos estos productos deben primero trasladarse al centro de distribución principal (ubicado en CABA en nuestro modelo).

Este primer tramo se llama "consolidación" porque es donde se reúnen todos los productos de un pedido en un mismo lugar. El costo de este tramo se calcula para cada producto individualmente, considerando la distancia desde su almacén de origen hasta el centro de distribución.

Segundo tramo - Distribución final:

Una vez consolidado el pedido completo en el centro de distribución, comienza el segundo tramo: desde CABA hasta la dirección del cliente. Este es el tramo que el cliente percibe directamente.

El costo de este tramo se calcula considerando el pedido como un todo, no producto por producto. Se suma el peso facturable total y se aplican las tarifas estándar completas, ya que este es el servicio principal que estamos prestando al cliente.

Sistema de zonificación geográfica

Para calcular costos de envío, implementamos un sistema de zonificación basado en los códigos postales argentinos (CPA), que tienen una estructura específica donde la primera letra indica la provincia.

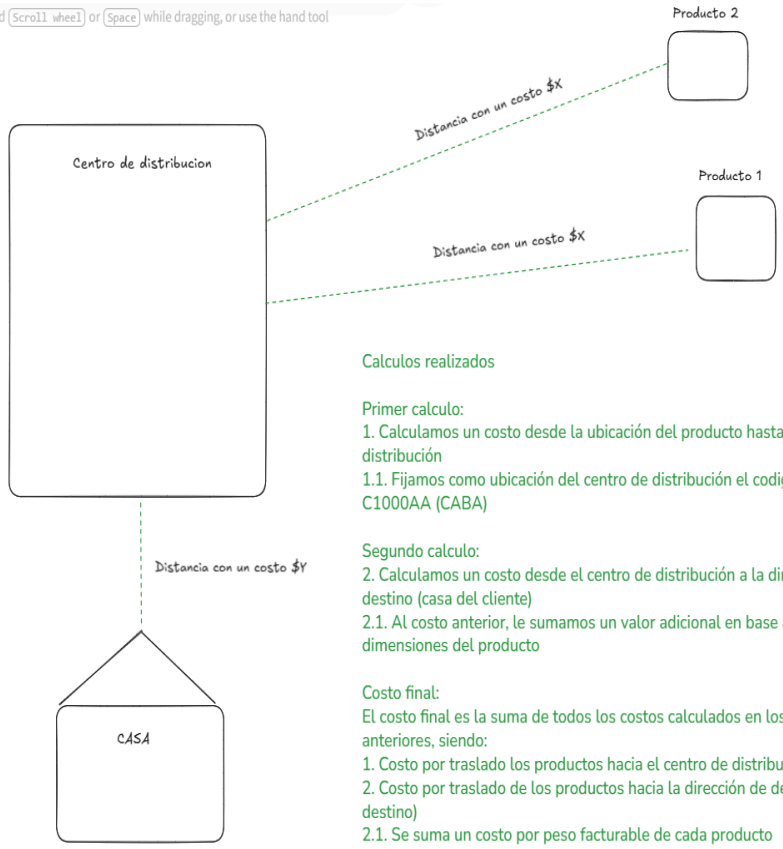
Mapeo provincia-región:

Agrupamos las provincias en regiones geográficas más amplias. Esta agrupación regional sirve para determinar costos intermedios. Si origen y destino están en la misma provincia, el envío es "local" y tiene el costo más bajo. Si están en provincias diferentes pero dentro de la misma región, es un envío "regional" con costo intermedio. Si están en regiones diferentes, es un envío "nacional" con el costo más alto. Todo esto fue ideado de manera visual a través del siguiente esquema:

Como calculamos los costos en base a las distancias?

Hicimos la siguiente simplificación:
- Si el envio es dentro de la misma provincia -> tiene un costo fijo de 1500
- Si el envio es dentro de la misma region (NEA, AMBA, NOA, etc) -> tiene un costo fijo de 3000
- Si el envio es hacia una region diferente (de NEA a AMBA) -> tiene un costo fijo de 4500

Esto a modo de evitar tener que calcular distancias a mano o en base a direcciones



Como calculamos el costo adicional en base al peso?

Muy simple, calculamos el volumen y lo multplicamos por un valor fijo -> en este caso definimos ese valor fijo como 250

Ejemplo:

Volumen del producto -> 15
Costo fijo por volumen -> 250
Calculo del costo -> 15 * 250

Calculos realizados

Primer calculo:
1. Calculamos un costo desde la ubicación del producto hasta el centro de distribución
1.1. Fijamos como ubicación del centro de distribución el codigo postal C1000AA (CABA)

Segundo calculo:
2. Calculamos un costo desde el centro de distribución a la dirección de destino (casa del cliente)
2.1. Al costo anterior, le sumamos un valor adicional en base a las dimensiones del producto

Costo final:
El costo final es la suma de todos los costos calculados en los puntos anteriores, siendo:
1. Costo por traslado los productos hacia el centro de distribución
2. Costo por traslado de los productos hacia la dirección de destino (casa de destino)
2.1. Se suma un costo por peso facturable de cada producto

En síntesis, la capa de servicios implementada representa el núcleo inteligente de nuestro backend. Es donde las necesidades del negocio se traducen en código ejecutable, mantenible y robusto.

En última instancia, una arquitectura sólida de servicios no es un lujo técnico, sino una necesidad práctica para construir sistemas que perduren, evolucionen y escalen conforme las necesidades del negocio crecen y cambian.

Capa de Datos

En lo que respecta a la **Gestión y Persistencia de Datos**, el backend se diseñó bajo el principio de separación de responsabilidades. Para garantizar la integridad y flexibilidad del sistema, la arquitectura de datos se desacopló del resto de la aplicación (como los controladores o la lógica de orquestación).

Tecnologías de persistencia para el motor de base de datos, el equipo decidió utilizar **MySQL**. Esta elección se fundamenta en la familiaridad técnica y la experiencia adquirida durante la cursada, habiendo trabajado con esta tecnología en la asignatura "Base de Datos" durante el primer cuatrimestre y profundizando su uso en "Sistema de Gestión de Base de Datos" en el segundo cuatrimestre. Esto nos permitió agilizar el desarrollo al trabajar sobre un entorno conocido.

Asimismo, se optó por implementar **TypeORM** como herramienta de mapeo objeto-relacional. Su incorporación aporta beneficios clave al proyecto, tales como la integración nativa con TypeScript, la definición clara de esquemas y relaciones mediante decoradores, y la abstracción de consultas complejas, lo que mejora la legibilidad y el mantenimiento del código.

A continuación, detallamos la **estructura de datos** y cómo transformamos la información desde reglas de negocio puras hasta registros de base de datos a través de cuatro niveles de abstracción:

1. Domain (Dominio) - Capa de Negocio

Los archivos en domain representan las entidades de negocio puras, sin dependencias de frameworks o bases de datos.

Características:

- Representan conceptos del negocio.
- No tienen decoradores de frameworks.
- Son independientes de la infraestructura.
- Usadas en toda la lógica de negocio.

2. Entity (Entidades) - Capa de Persistencia

Los archivos en entities son entidades de TypeORM que mapean directamente a tablas de base de datos.

Características:

- Decoradores de TypeORM (@Entity, @Column, etc.).
- Mapean a tablas SQL.
- Definen relaciones entre tablas.
- Solo se usan en la capa de repositorios.

3. Repository Abstracto - Capa de Abstracción

Los archivos como UserRepository definen contratos/interfaces para acceso a datos.

Características:

- Define qué operaciones se pueden hacer.
- No define cómo se implementan.
- Trabaja con objetos de Domain, no Entities.
- Permite cambiar la implementación sin afectar la lógica de negocio.

4. Repository MySQL - Capa de Implementación

Los archivos en mysql implementan los repositorios abstractos usando TypeORM.

Características:

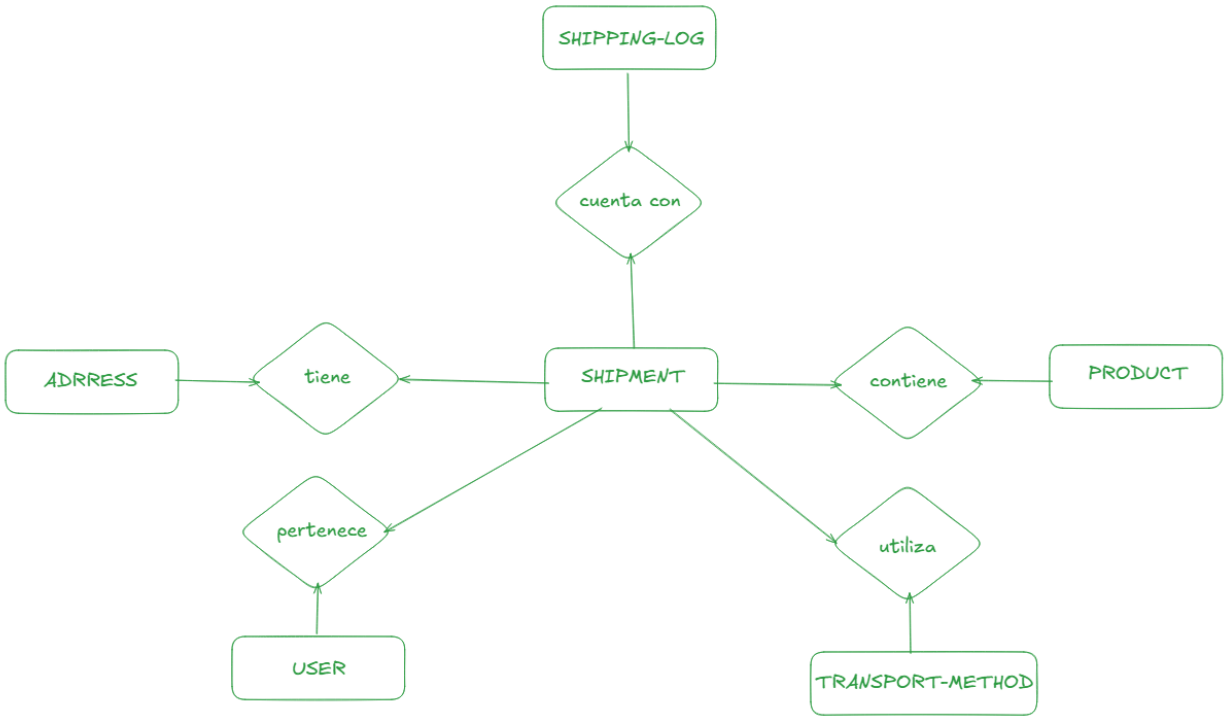
- Implementa el repositorio abstracto.
- Usa TypeORM y las Entities.
- Contiene queries SQL específicos.

Ventajas de esta Arquitectura

- Testabilidad: Puedes mockear repositorios fácilmente
- Flexibilidad: Cambiar de base de datos es trivial
- Separación de responsabilidades: Cada capa tiene un propósito claro.
- Mantenibilidad: Cambios en la BD no afectan la lógica de negocio
- Desacoplamiento**:** Aísla la lógica de negocio de los detalles de infraestructura.

Estrategia de Datos Iniciales (Seeding) Adicionalmente, implementamos un `SeedService` automático que se ejecuta al inicio de la aplicación (`OnModuleInit`). En lugar de utilizar scripts SQL manuales, optamos por usar código TypeScript para poblar la base de datos. Esta decisión nos permite reutilizar los mismos Repositorios y Entidades del negocio para crear los datos, asegurando que los datos de prueba siempre sean válidos y consistentes con las reglas actuales del sistema.

Diagrama Entidad-Relación de la Base de Datos



Para garantizar la integridad y coherencia de la información logística, se diseñó un modelo de datos relacional normalizado cuyo diagrama adjunto ilustra la interacción entre las entidades principales del sistema. El corazón de este modelo es la entidad **SHIPMENT** (Envío), la cual representa la orden de envío y actúa como el nodo central que conecta todas las demás piezas del sistema.

Tests

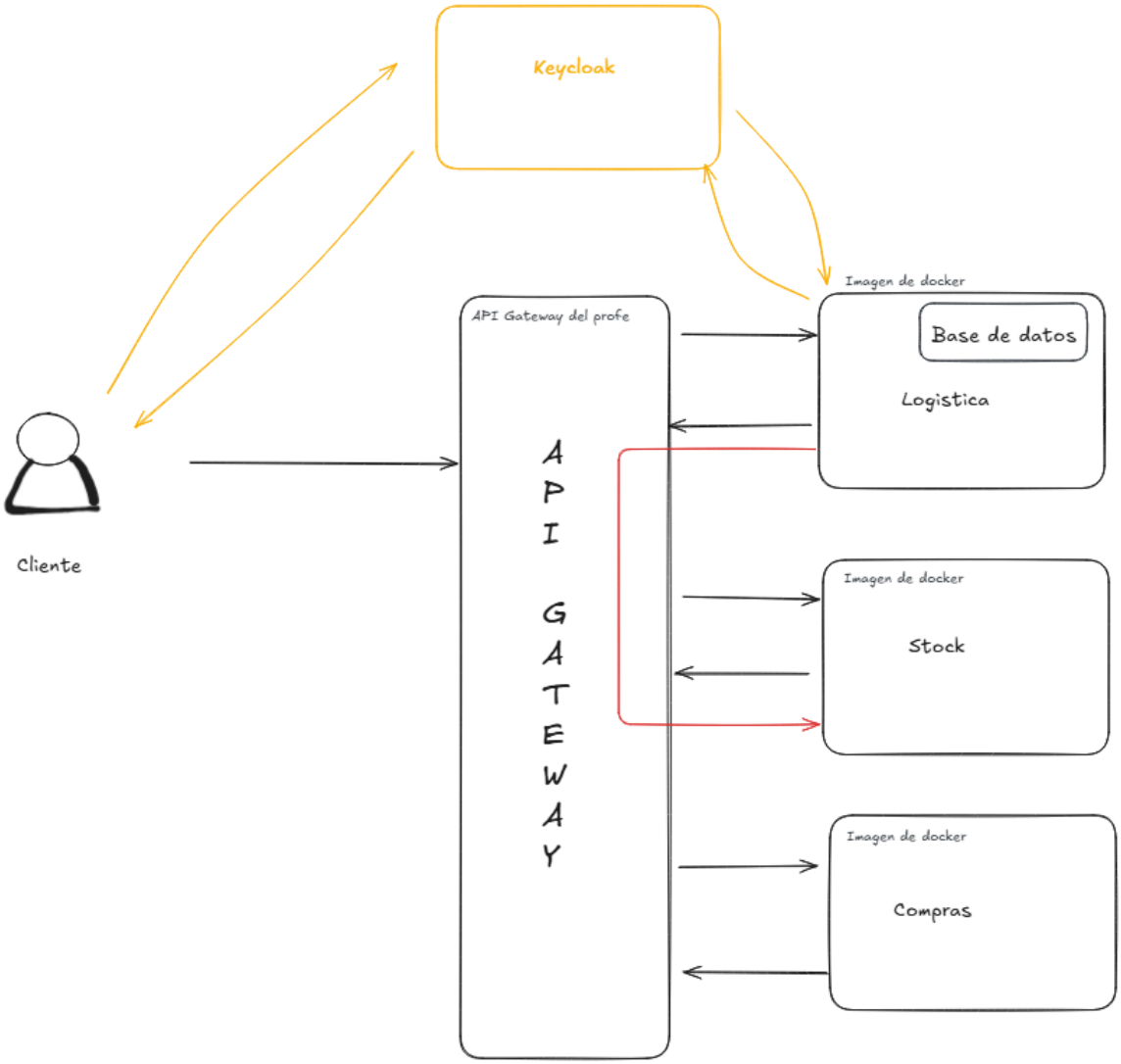
Para garantizar la robustez y confiabilidad del backend, el equipo implementó una serie de pruebas de integración (E2E) y su posterior inclusión al CI/CD. Esta estructura permite validar la interacción del sistema en su conjunto y el despliegue automático en caso de su correcto funcionamiento.

Se desarrolló un conjunto de tests **End-to-End** (E2E) diseñados para evaluar la API desde la perspectiva de un cliente real, verificando el comportamiento de todos los endpoints expuestos por el controlador. Para su implementación, se utilizaron las herramientas del paquete de testing de NestJS para instanciar un módulo de aplicación aislado, junto con la librería Supertest para simular peticiones HTTP reales. Con el objetivo de centrar las pruebas en la capa de controladores y evitar dependencias externas durante la ejecución, como la conexión a la base de datos real, se optó por utilizar datos simulados (mocks) para el servicio de logística. Esto aísla eficazmente la lógica de negocio, garantizando una ejecución rápida y predecible. La cobertura de estos tests valida los códigos de estado HTTP y la estructura de datos de respuesta para operaciones críticas, tales como la creación y listado de envíos, el cálculo de costos, la cancelación de pedidos y la actualización de estados, además de la consulta de datos maestros como los métodos de transporte.

La ejecución de todas estas pruebas está orquestada mediante Jest y ts-jest, integrándose en el pipeline de Integración Continua (CI/CD) para asegurar que cualquier cambio en el código sea validado automáticamente antes de su despliegue. La principal ventaja de este enfoque híbrido, basado en el uso de mocks en lugar de una base de datos viva para las pruebas de paso, radica en la velocidad de ejecución y el aislamiento total de dependencias, lo que facilita el mantenimiento y ofrece un feedback inmediato al equipo de desarrollo sobre la integridad del sistema.

Integración

La integración de nuestro sistema se diseñó bajo un enfoque modular y distribuido, separando servicios críticos en diferentes entornos para mantener aislamiento, escalabilidad y facilidad de mantenimiento. En la plataforma de hosting (Railway), armamos dos entornos principales: uno dedicado exclusivamente al servicio de autenticación (via Keycloak) y otro que contiene los servicios centrales de la aplicación: la API Gateway, el backend de logística con su base de datos, y el backend de stock con su propia base de datos. Cada servicio dispone de sus variables de entorno definidas en Railway, lo que permite mantener configuraciones separadas (credenciales, URLs, parámetros específicos) sin interferencias, y simplifica la gestión de secretos y configuración de producción.



El flujo representado en la imagen corresponde a la interacción entre el cliente, la API Gateway, los microservicios y Keycloak dentro de nuestra arquitectura. El proceso inicia cuando el cliente solicita un token de autenticación a Keycloak. Una vez obtenido, utiliza ese token para realizar las peticiones a la API Gateway, que actúa como punto de entrada del sistema. La API Gateway recibe cada request y la redirige al microservicio que corresponda, enviando también el token original proporcionado por el cliente. Cada servicio, en lugar de confiar únicamente en una validación previa, verifica el token directamente contra Keycloak, lo que añade una capa adicional de seguridad al permitir que cada componente valide por sí mismo las credenciales y permisos asociados.

Cuando un microservicio necesita consultar información o ejecutar acciones en otro servicio, no se comunica directamente, sino que envía la petición nuevamente a la API Gateway utilizando el mismo token del cliente que originó la solicitud inicial. Esto asegura trazabilidad, coherencia en los permisos y una ruta de comunicación uniforme, manteniendo una arquitectura ordenada, segura y desacoplada. Este flujo integrado garantiza que todos los componentes operen de manera coordinada dentro de un entorno distribuido, preservando la autenticación centralizada y el control de acceso descentralizado que caracteriza a nuestra arquitectura.

Además del flujo de comunicación entre servicios, nuestra infraestructura integra un mecanismo de despliegue continuo que garantiza que cada componente esté siempre actualizado. Cuando se actualizan los paquetes de alguno de estos servicios (mediante commits en el repositorio de la organización de GitHub FRRe-DS), Railway detecta los cambios y dispara el nuevo despliegue automáticamente, asegurando que la versión en producción esté siempre alineada con la versión de código. Este modelo favorece un flujo de Continuous Delivery, reduciendo la carga operativa y minimizando errores derivados de despliegues manuales.

Publicamos la interfaz de usuario (frontend) utilizando Vercel como plataforma de despliegue. Esta decisión aporta beneficios importantes: cada vez que se realiza un cambio en el código (por ejemplo, un merge en la rama principal o un pull request), La herramienta nos permite visualizar cambios en tiempo real mediante despliegues automáticos por cada actualización de código, facilitando la revisión continua sin afectar el entorno productivo. Además, Vercel ofrece optimizaciones automáticas de recursos (como imágenes y assets) y pre-renderizado de páginas cuando corresponde, lo que mejora el rendimiento, reduce la carga en los servidores backend y proporciona una mejor experiencia al usuario final.

Dificultades y problemas que surgieron

1. Complicaciones mínimas

1.1 Falta de organización al inicio

Al comenzar el proyecto, nos costó bastante mantener un orden claro entre tareas, responsables y avances. No teníamos una metodología definida y eso generaba confusiones sobre qué debía hacerse primero.
Para resolverlo, decidimos organizar todo en **Trello**, lo que nos permitió distribuir tareas, priorizar y tener visibilidad del progreso. A partir de ahí la coordinación del equipo mejoró.

1.2 Desconocimiento de las tecnologías

Otra complicación inicial fue que varias herramientas y frameworks que usamos eran nuevas para nosotros. Esto generaba dudas constantes sobre cómo implementarlas correctamente.
Para superar esa barrera, recurrimos bastante a la **IA como apoyo**, acelerando el aprendizaje, validando conceptos y resolviendo dudas puntuales mientras avanzábamos.

2. Complicaciones mayores (nivel técnico)

Estas son las dificultades reales del proyecto, las que implicaron más tiempo, frustración y aprendizaje.

2.1 Integrar Keycloak al servicio

Uno de los mayores retos fue la integración de **Keycloak** para manejar la autenticación.
El problema no fue sólo entender cómo funcionaba Keycloak, sino hacer que el servicio se comunicará correctamente con él dentro de un entorno con **Docker Compose**.

Tuvimos varios inconvenientes:

- Los **puertos** no coincidían entre los contenedores.
- Las **URL internas** que usaba Keycloak para redirigir al login no eran las mismas que las externas.
- El servicio esperaba endpoints que no estaban expuestos como pensábamos.

En resumen, todo funcionaba fuera del contenedor, pero al meterlo en Docker, no se resolvían las direcciones correctamente.
Probamos diversas soluciones para este problema, las cuales fueron principalmente dirigidas al manejo de redes internas de los contenedores de Docker. Unos ejemplos son:

- Agrupar todas las imágenes de los servicios en un mismo contenedor de Docker. Esta solución nos permitía comunicarnos correctamente entre los servicios utilizando la red interna del contenedor. El problema que teníamos era que la redirección de keycloak nos generaba problemas con el login a ese servicio.
- Separar a Keycloak del contenedor de todos los servicios. Este formato nos arreglaba el problema del login anterior, pero nos generaba inconvenientes con las validaciones de tokens en un servicio A, que fueron pedidos por un servicio B. Es decir que, si el token lo pedía un servicio este si lo podía validar por el mismo, pero si lo enviaba a un servicio distinto ese segundo no realizaba la validación de manera correcta.

Para solucionar estos problemas decidimos **desplegar el servicio de AUTH**, de modo que nuestro Keycloak quedará como un servicio externo y pudiéramos usar su URL pública tanto en nuestro proyecto como en los de los demás grupos con los que trabajamos. El despliegue de Keycloak fue de gran ayuda porque nos evitó los problemas anteriores, ya que todos los servicios validan y solicitan contra la misma imágen de la herramienta de autenticación. Lo implementamos usando Railway, aprovechando una plantilla ya preparada para Keycloak.

2.2 Deploy del proyecto y API Gateway

Cuando logramos que Keycloak funcionara, pasamos a otro nivel de problemas: el **deploy general**, y acá apareció el API Gateway como punto crítico.

Los principales desafíos fueron:

- Las **direcciones internas** del API Gateway.
- Algunos servicios estaban expuestos en puertos distintos a los que esperábamos.
- El Gateway no resolvía correctamente los nombres de host de los contenedores, lo que causaba respuestas 404 o timeouts.

Y mientras intentábamos arreglar eso, surgió otro obstáculo:

Problemas de CORS entre servicios

En el entorno real, los navegadores bloqueaban la comunicación entre módulos por políticas de CORS mal configuradas.

Tuvimos que ajustar:

- Headers permitidos.
- Métodos.
- Orígenes válidos.
- Uso de HTTPS / HTTP según el entorno.

Fue necesario ir probando cada servicio individualmente para validar que la comunicación realmente estuviera permitida.

Resultado final

Aunque fueron dificultades bastante grandes, nos ayudaron a entender cómo se conectan los servicios en una arquitectura modular real, cómo resolver problemas de integración entre contenedores y cómo manejar despliegues más complejos.

Conclusión

El desarrollo del microservicio de logística ha representado un desafío técnico significativo que nos permitió aplicar patrones de diseño avanzados y estándares de la industria en un escenario práctico. La transición hacia una arquitectura limpia, junto con la dockerización integral y la automatización mediante CI/CD, no solo resolvió los problemas iniciales de inconsistencia entre entornos, sino que estableció una base sólida, mantenible y escalable para el futuro del sistema.

Desde la perspectiva del backend, el mayor logro ha sido desacoplar la lógica de negocio de la infraestructura tecnológica. Gracias a la implementación de contratos abstractos y la inyección de dependencias, el sistema es ahora resiliente a cambios en la base de datos y fácilmente testeable. La incorporación de mecanismos de seguridad con Keycloak y la gestión automatizada de datos de prueba (seeds) garantizan que el servicio sea robusto y seguro desde el primer despliegue.

En cuanto al frontend, el equipo se enfocó en construir una interfaz de usuario moderna, intuitiva y responsiva, implementando validaciones de datos en tiempo real y una gestión eficiente de los estados de carga y error en los formularios de cotización y seguimiento, logrando así una integración fluida con la API desarrollada.

Este Trabajo Práctico Integrador representó un hito fundamental para el equipo, constituyendo para la mayoría nuestra primera experiencia inmersiva en un proyecto de esta envergadura y complejidad arquitectónica. A pesar de partir con una base de conocimientos heterogénea respecto a las tecnologías implementadas, logramos transformar esa curva de aprendizaje en una oportunidad de crecimiento. Este proceso no solo nos permitió dominar nuevas herramientas técnicas, sino que nos brindó la invaluable experiencia de enfrentar la incertidumbre, investigar soluciones y colaborar para llevar un sistema desde su concepción hasta un despliegue productivo funcional.

En definitiva, este proceso nos ha permitido consolidar conocimientos teóricos sobre arquitectura de software, DevOps y gestión de datos, demostrando en la práctica que la calidad del código y la automatización de procesos son tan importantes como la funcionalidad misma del producto final.