

Trabajo Práctico

Final

Desarrollo de Software

Profesores:

- Jose Alejandro Fernández
- Jorge Eduardo Villaverde

Grupo N°6 - Integrantes:

- Azula, Tomas - tomas-a21@outlook.com
- Alegre Roth, Facundo Tahiel - tahiel.utn@gmail.com
- Brites, Elisa Alejandra - elisabrites8@gmail.com
- Caneva, Franco Manuel - francomanuc@gmail.com
- Dajruch, Pablo Naim - pablodajruch10@gmail.com
- Gomes, Iara Micaela - miacelagomess333@gmail.com
- Guaglianone, Tobias - tobiasguaglianone@gmail.com
- Orquera, Ariel Agustín - disciplinam.studium@gmail.com
- Urturi, Carla Micaela - carlaurturi2013@gmail.com

Módulo de Transporte, Logística y Seguimiento.....	3
Arquitectura.....	3
Dominio.....	4
Aplicación.....	4
Infraestructura.....	4
Presentación.....	4
Integración con servicios externos.....	5
Escalabilidad.....	5
Base de datos.....	5
ORM.....	6
Reglas de negocio.....	6
En el Backend.....	7
En el Frontend.....	7
Autenticación: Keycloak.....	8
Docker compose.....	8

Módulo de Transporte, Logística y Seguimiento

El subsistema de Transporte, Logística y Seguimiento es el encargado de gestionar todo lo relacionado con el envío de los pedidos generados en el Portal de Compras.

Su principal responsabilidad es mostrar los tipos de transportes posibles, calcular el costo del envío, crear el envío asociado a una compra y mantener actualizado el estado del mismo hasta su entrega.

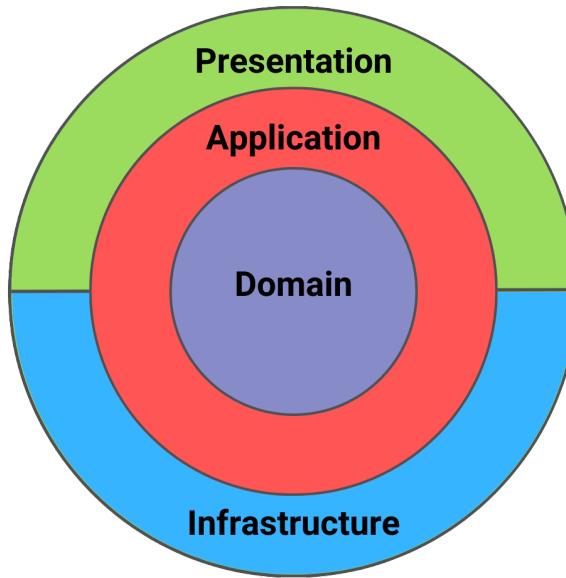
Desde el punto de vista de la integración, este módulo no interactúa directamente con el usuario final, sino que se comunica con los otros módulos/subsistemas mediante servicios REST.

- Recepción de solicitudes desde el portal de compras:
 - Cuando el cliente está cerrando una compra y selecciona la dirección de entrega, el portal envía la dirección de entrega, los productos y el tipo de transporte elegido.
 - Nuestro módulo de transporte calcula el costo de envío (el cual para hacerlo le pide al módulo de stocks que le brinde los datos de la dirección de origen del producto) y la fecha estimada de entrega, y devuelve esta información al portal de compra para que sea mostrada al usuario antes de confirmar la compra.
- Creación del envío cuando la compra se confirma:
 - Una vez que el usuario confirma la compra, el portal de compras registra el pedido y envía a nuestro módulo de transporte el id de pedido y la id del usuario, la dirección de entrega, el medio de transporte seleccionado y el detalle de productos.
 - Nuestro módulo crea un envío (shipment), calcula y guarda el costo final del envío, y devuelve al Portal de Compras el id del pedido. Ese identificador queda asociado a la orden para poder seguir el paquete.
- Consulta y seguimiento del envío:
 - El Portal de Compras, y eventualmente el usuario final, pueden consultar el estado del envío, exponiendo la información principal del envío: direcciones, productos, costo, medio de transporte, estado actual y fecha estimada de entrega.
 - Además, nuestro módulo de transporte permite a través del Dashboard, listar los envíos, ver el detalle y actualizar el estado.

Arquitectura

La arquitectura empleada en el sistema es un tipo de **arquitectura en capas**. Elegimos arquitectura en capas porque fue una de las más sencillas y entendibles conceptualmente, y pensamos que sería relativamente sencilla de aplicar.

En específico, usamos una **arquitectura Clean**, que es un tipo específico de arquitectura en capas que resulta ser un poco más estricta y organizada. En esta arquitectura, las capas se clasifican de más internas a más externas. Una capa puede depender de todas las capas más internas en la arquitectura.



Dominio

La capa **Domain** representa el núcleo de la aplicación, contiene la lógica de negocio pura. Contiene las Entidades, Value Objects, Interfaces y Repositorios. Tiene la característica de no depender de ninguna otra capa, ya que es la capa más interna.

Aplicación

La capa **Application** cumple el rol de coordinar los casos de uso, orquestando la lógica del dominio y accediendo a la infraestructura a través de interfaces. Contiene los Servicios (casos de uso), DTOs (*Data Transfer Objects*) e interfaces implementadas por la capa de Infraestructura. Esta capa depende de Domain, ya que utiliza las entidades y reglas del negocio.

Infraestructura

La capa de **Infrastructure** representa el cómo se hace. Cumple el rol de implementar las tecnologías que nuestro sistema necesita para funcionar. Contiene los Repositorios concretos (implementaciones de las interfaces de Domain para bases de datos), Servicios externos (integración con las demás APIs y la Base de Datos). Depende de Application y Domain a través de interfaces, y contiene la **implementación concreta**, pero no la lógica del negocio.

Tener la Infraestructura en una capa aislada facilita cambiar de tecnologías sin tocar las reglas del negocio.

Presentación

La capa de **Presentation** (o API) expone los endpoints del sistema a otros módulos, permitiendo recibir solicitudes y devolver respuestas.

Contiene los Controladores y Endpoints, que reciben peticiones HTTP y llaman a los servicios de Application. También incluye la lógica de autenticación de usuarios.

Esta capa se utiliza como entry point, y se encarga de validar y transmitir datos, pero no contiene reglas de negocio.

Integración con servicios externos

La integración con servicios externos se da con la **Base de Datos de Logística**, de la cual hablaremos con más detalle más abajo.

También es importante la integración con el **Servicio de Stock**, el cual contiene información de las dimensiones y el peso de los productos, que resulta de gran importancia para el área de Logística para poder calcular el costo de los envíos correctamente.

Escalabilidad

En rasgos generales, el sistema completo presenta un diseño que facilita la escalabilidad al estar dividido en módulos (Compras, Stock y Logística), cada uno con su propia API, lo que permite escalar módulos individualmente según sea necesario.

En cuanto al módulo de Logística, al estar desarrollado como una API *stateless*, es decir, que no guarda estados en memoria entre peticiones, permite que en el futuro se pudieran levantar más instancias del mismo servicio detrás de un balanceador sin necesidad de realizar grandes modificaciones al código. Además, a pesar de no contar con cantidades masivas de datos, el sistema está preparado para situaciones más exigentes al implementar estrategias de paginación para transmitir la información por bloques.

Por último, al haber configurado el servicio dentro de un contenedor de Docker, sería más fácil levantar más copias del servicio en otros servidores si la exigencia sobre este crece en el futuro.

Base de datos

En cuanto a la Base de Datos, elegimos como motor a MySQL, debido a que es el motor con el que más experiencia contamos en el grupo gracias a las materias de Bases de Datos y Sistemas de Gestión de Bases de Datos. Pensamos que, conociendo las características del motor, sería más que suficiente para lo que necesitábamos desarrollar.

La Base de Datos del módulo de Logística es utilizada para almacenar información completa de los envíos, viajes, medios de transporte y centros de distribución. También se incluyen tablas con información geográfica de las localidades del país para poder calcular las distancias internamente.

ORM

Elegimos Entity Framework porque es la herramienta que mejor se integra con .NET y nos facilita mucho el trabajo. Básicamente, funciona como un puente que nos deja manejar nuestra base de datos MySQL escribiendo código en C# en lugar de tener que armar consultas manuales, lo que nos ahorra tiempo y hace que conectar todo el backend sea mucho más sencillo.

Reglas de negocio

El cálculo del costo de un envío se realiza considerando las características del envío (cantidad y tipo de productos, peso y dimensiones cuando están disponibles), la dirección de origen, la dirección de destino y el método de transporte elegido. Cada tipo de transporte tiene un multiplicador de costo asociado a su velocidad: a mayor velocidad, mayor precio.

Para calcular la distancia entre origen y destino se utilizan las latitudes y longitudes de las localidades involucradas. Cuando se registra un envío, el centro de distribución de origen se selecciona como el centro más cercano a la dirección de destino (cliente). Los productos pueden provenir de distintas localidades o provincias, y para el cálculo de costo se toma la distancia entre la localidad de origen de cada producto y la localidad de destino del cliente.

Cuando se solicita un cálculo de costo básico, es decir, sin que el usuario haya seleccionado todavía un medio de transporte, el sistema utiliza por defecto el transporte de tipo road (camión), por ser el método más común. En caso de no encontrar una localidad en la base de datos, se recurre a un diccionario por provincia, utilizando latitudes y longitudes generales asociadas a la letra de la provincia correspondiente.

La creación del envío se realiza principalmente a partir de una compra confirmada en el Portal de Compras. Los datos provienen de ese subsistema (orden, usuario, dirección de entrega y productos) siguiendo el contrato definido en la OAS entre los grupos. Para simplificar las pruebas, también permitimos crear envíos desde nuestro Frontend de Logística. En ambos casos, el precio se calcula reutilizando el mismo servicio que se usa para el endpoint de cálculo de costo, con la única diferencia de que, en la creación del envío, ya se conoce y se envía el medio de transporte definitivo elegido por el cliente.

Para consultar envíos, el módulo ofrece dos modalidades:

- Búsqueda por ID: devuelve toda la información detallada del envío (direcciones, productos, costo, estado, fechas, etc.), tal como se declara en la OAS.
- Listado con filtros: permite filtrar por ID, estado, fechas o destino y devuelve una lista paginada con información resumida, incluyendo el shipping_id para poder acceder luego al detalle completo.

En cuanto a la cancelación de envíos, la lógica de negocio establece que un envío no puede ser cancelado si ya fue entregado o si se encuentra previamente cancelado. Este endpoint puede ser utilizado tanto por el módulo de Stock como por el de Compras para reaccionar ante incidencias.

Además de los endpoints expuestos a otros módulos, se implementaron endpoints de uso interno para el Frontend de Logística:

- Listado de localidades.
- Dashboard de envíos (listado paginado y filtrado).
- Cambio de estado de un envío.

El endpoint de cambio de estado es más flexible que el de cancelación, pero respeta ciertas restricciones de negocio:

- No se puede modificar el estado de un envío que ya esté en delivered (entregado) o cancelled (cancelado).
- No se permite volver al estado inicial created una vez que el envío avanzó en el flujo.
- Cada cambio de estado puede ir acompañado de un comentario, que se registra en el historial de estados (logs) para mantener trazabilidad.

Desde nuestro Frontend (dashboard) se pueden crear envíos (limitados a los datos existentes en nuestra base), cambiar el estado de los envíos respetando las restricciones anteriores, filtrar la lista de envíos por distintos criterios y ver el detalle completo de un envío, incluyendo los productos involucrados y la información básica de los métodos de transporte disponibles.

En el Backend

Lenguaje de programación usado: **C#**

Elegimos C# principalmente por dos motivos:

Elegimos C# principalmente porque es un lenguaje que nadie en el grupo había utilizado antes y teníamos curiosidad por aprender, además varios ejemplos y explicaciones que se nos dieron en clase fueron utilizando este lenguaje, tanto en esta como en otras materias.

Además, según lo que leímos, es un lenguaje que **facilita armar APIs para el backend**. Investigando vimos que con C# y .NET es bastante simple crear una API web: podés definir controladores, endpoints, devolver y recibir JSON y organizar bien las capas del proyecto. Eso encaja justo con lo que necesitábamos para nuestro módulo de Transporte y Logística, que tiene que exponer servicios al resto de los subsistemas y al frontend.

En resumen, usamos C# porque queríamos aprovechar la oportunidad para aprender a programar en este lenguaje, y porque se integra muy bien con .NET para armar APIs REST y nos da cierta “seguridad” al momento de implementar y mantener la lógica de negocio del módulo de Transporte y Logística.

En el Frontend

Lenguajes de programación usados: **Svelte / TypeScript**

Elegimos Svelte + TypeScript por los siguientes motivos:

- **Porque nos permitía hacer un frontend bastante prolífico sin meternos en algo demasiado pesado.** La idea era poder armar pantallas y componentes rápido, sin pelearnos tanto con la herramienta. Además, como el renderizado de Svelte se hace del lado del servidor y consideramos que el tráfico en nuestro frontend no sería tan exigente, lo vimos como una buena opción.
- Por **curiosidad y aprendizaje.** Al igual que con el backend, elegimos tecnologías con las que no llegamos a trabajar anteriormente, y aprovechamos la oportunidad para aprender a utilizarlas con un proyecto no tan complejo.
- **TypeScript lo usamos más que nada para tener un poco más de orden.** Nos obliga a decir qué datos esperamos y eso ayuda a evitar errores tontos cuando hablamos con el backend y manejamos las respuestas de las APIs.

Autenticación: Keycloak

Para la autenticación se usó Keycloak en lugar de inventar nuestro propio sistema de usuarios y contraseñas. Keycloak es el proveedor de identidad de todo el proyecto: ahí se encuentran los usuarios, las contraseñas y los roles que garantiza la seguridad de acceso a cada endpoint según sus políticas de acceso.

En este caso levantamos Keycloak con Docker usando la configuración que se nos dio desde la cátedra la cual contiene: realm ds-2025-realm, clientes por grupo y roles como compras-be, stock-be, logistica-be, etc.

Al iniciar sesión en el frontend, el usuario está mandando sus credenciales hacia Keycloak, si estas se encuentran en la base de datos del mismo, Keycloak devuelve un token que luego el frontend utilizará para cada pedido que se le haga al backend.

Desde el backend utilizaremos este token cada vez que queramos autorizar alguna operación, para esto, los controladores tienen el atributo [Authorize] que hace que, a menos que se utilice el token, no se realice la operación y se devuelva el error 401 (error por falta de autenticación). Por ejemplo:

```
[Authorize(Roles = "compras-be, stock-be, logistica-be")]
```

De esa forma solo pueden utilizar nuestros endpoints los módulos que tengan el rol correcto en Keycloak.

Docker compose

El archivo docker-compose que utilizamos, se encarga de levantar 3 servicios distintos: La API de Logística, la Base de Datos y el Frontend.

La base de datos MySQL cuenta con un directorio para persistencia, donde se almacenan los datos de la misma permitiendo que sobrevivan a un **docker compose down**. La base de datos se carga con

registros de prueba almacenados en archivos .csv, por lo que en el archivo docker compose es necesario habilitar explícitamente la carga de archivos locales. También se configura el usuario root con permisos totales sobre la base de datos, así como un usuario ApiUser que tiene acceso a la base de datos de Logística.

La base de datos también tiene un mecanismo de *healthcheck*, que cada cinco segundos intenta responder a un ping interno con un tiempo de espera máximo de tres segundos y hasta 10 reintentos.

El servicio **api**, que es el backend en .NET 8, se construye a partir del Dockerfile, que usa una estrategia de dos etapas: primero compila la aplicación con el SDK .NET 8 y luego genera una imagen liviana solo con el runtime para ejecutarla.

El backend está configurado para iniciar recién cuando la base de datos responda correctamente al healthcheck, evitando errores de conexión en frío.

Por último, el servicio del **frontend**, se construye desde la carpeta frontend-dashboard usando su propio Dockerfile, ya que este se encuentra totalmente desacoplado del backend al momento de la build. El código del frontend se monta en tiempo real desde el sistema hacia el contenedor, lo que permite **desarrollo con recarga en caliente**.