UNIVERSIDAD TECNOLÓGICA NACIONAL - FACULTAD REGIONAL RESISTENCIA

DESARROLLO DE SOFTWARE
TRABAJO PRÁCTICO INTEGRADOR

Año: 2025
Número de Grupo: 11
Módulo: Stock
Integrantes:

- Arduino, Augusto Horacio Lorenzo - lorenzoarduinoh@gmail.com
- Ayala, Alejo Yerimen - alejoayala79306@gmail.com
- Brocal, Mateo Joaquín - joaquinbrocal25@ca.utn.frre.edu.ar
- Fidanza Roch, Gonzalo - g.fidanza.roch@gmail.com
- Gimenez, Franco Nahuel - franconahuelgimenez199@ca.frre.utn.edu.ar
- Kobluk, Tomás - tomas.kobluk@gmail.com
- Martina, Ignacio Antonio - ignacioamartina@gmail.com
- Mecozzi, Luz - luzmecozzi@gmail.com
- Soto, Hector Aron - soto.aron.official@gmail.com
- Suarez, Facundo Gabriel - facundosuarez200@gmail.com

Índice

1. INTRODUCCIÓN

Este informe detalla el análisis técnico del subsistema "Stock de Bienes y Servicios", desarrollado como parte del Trabajo Práctico Integrador (TPI). El sistema tiene como objetivo permitir a los vendedores gestionar sus productos y stock, así como integrarse con un portal de compras externo para la reserva y liberación de mercadería, coordinando posteriormente con un servicio de logística.

El proyecto está estructurado como un monorepo que contiene tanto el Frontend como el Backend, ambos desarrollados sobre el framework Next.js, utilizando TypeScript para asegurar el tipado estático.

2. DESARROLLO

2.1. Índice Temático de Tecnologías

A continuación, se detallan las herramientas y lenguajes utilizados, clasificados por su ámbito de aplicación.

2.1.1. Frontend

- **Lenguaje:** TypeScript.
- **Framework Principal:** Next.js (App Router).
- **Librería de UI: React**. Se eligió React debido a que su arquitectura basada en componentes nos permite dividir la interfaz en piezas pequeñas y reutilizables (como botones, formularios o tablas), lo que facilita el mantenimiento y asegura una consistencia visual en toda la aplicación. También es clave su capacidad para manejar el estado de manera eficiente, permitiendo que el dashboard de stock se actualice dinámicamente y sin recargas lentas cada vez que los datos cambian. Por último, al ser la base fundamental sobre la que funciona Next.js, nos brinda una integración nativa y fluida para construir experiencias de usuario modernas, rápidas y altamente interactivas.
- **Estilos: Tailwind CSS**. Se eligió Tailwind CSS debido a que nos permite aplicar estilos directamente sobre los componentes HTML, lo que acelera significativamente el desarrollo al evitar tener que escribir y mantener archivos de hojas de estilo separados. También proporciona un sistema de diseño estandarizado con colores y espaciados predefinidos, asegurando que toda la interfaz mantenga una coherencia visual profesional y limpia
- **Componentes UI:** Radix UI. Se eligió Radix UI debido a que proporciona los cimientos funcionales para componentes complejos (como menús desplegables, diálogos o tooltips) resolviendo por nosotros toda la lógica difícil de accesibilidad y comportamiento, sin imponernos un diseño visual fijo. También nos permite mantener un control total sobre la estética, ya que al ser componentes "sin estilo", podemos personalizarlos completamente con Tailwind CSS para que encajen

perfectamente con la identidad visual del proyecto. Por último, garantiza que nuestra aplicación sea inclusiva y usable para todos, cumpliendo automáticamente con los estándares de navegación por teclado y lectores de pantalla.

- **Validación de Formularios: Zod**. Se eligió Zod debido a que nos permite definir reglas estrictas y esquemas para los datos (como formatos de email o rangos de precios) asegurando que la información ingresada por el usuario sea válida antes de siquiera enviarla al servidor.También es fundamental su integración perfecta con TypeScript, ya que es capaz de generar automáticamente los tipos de datos a partir de las reglas de validación, lo que nos evita duplicar código y previene errores de inconsistencia. Por último, facilita enormemente el manejo de errores en los formularios, permitiendo mostrar mensajes claros y precisos al usuario final cuando un campo no cumple con los requisitos esperados.

- **Autenticación:** NextAuth.js (integrado con Keycloak).

- **Gestión de Iconos**: Lucide React / SVGs.

- **Cliente HTTP:** Fetch API (nativo) con wrappers personalizados (`useAuthenticatedApi`).

2.1.2. Backend

- **Lenguaje:** TypeScript. Se eligió TypeScript debido a que al tratarse de un sistema de stock y reservas, se manejan datos sensibles como precios, cantidades y estados, por lo que necesitamos un lenguaje que prevenga errores comunes antes de que el código se ejecute, como lo hace TypeScript. También al proyecto usar Prisma ORM para la base de datos, TypeScript permite que Prisma genere automáticamente los tipos de datos basados en el esquema, por lo que si se cambia un campo en la base de datos se mostrará en el código donde se debe actualizar, para garantizar una consistencia robusta entre el backend y la base de datos. Por último al tener el frontend y el backend en TypeScript, facilita compartir interfaces entre ambos

- **Framework:** Next.js. Se eligió Next.js como Framework para trabajar debido a que nos permite tener todo el código en un solo lugar. El mismo servidor de Next que entrega la página web (Front), también procesa las peticiones de datos (back) lo que simplifica mucho el despliegue y desarrollo. Como todo está modularizado hace que el código sea fácil de entender y de mantener. Además se integra directamente con la base de datos, las rutas se se ejecutan en el lado del servidor (Node.js), lo que permite usar herramientas como Prisma ORM (como se mencionó anteriormente) de forma segura para conectar directamente con PostgreSQL, ocultando las credenciales y la lógica de la base de datos al usuario final.

- **Base de Datos:** PostgreSQL. Se eligió PostgreSQL porque es un motor de base de datos muy confiable que garantiza la integridad de los datos, asegurando que el control de stock y las transacciones sean precisas y no tengan errores. También fue clave su capacidad para manejar datos flexibles tipo JSON, lo que nos permitió

guardar detalles complejos del producto (como dimensiones o fotos) de forma sencilla dentro de la misma estructura. Por último, maneja los cálculos numéricos con total exactitud, algo fundamental para los precios, y se integra perfectamente con las herramientas de desarrollo que utilizamos. (Como Prisma ORM)

- **Autenticación/Seguridad:** `jose` (para manejo y validación de JWT), integración con Keycloak: Se eligió la integración con Keycloak junto a la librería `jose` debido a que permite delegar toda la gestión de usuarios y contraseñas a un servicio externo dedicado, evitando que tengamos que almacenar credenciales sensibles en nuestra propia base de datos. También se optó por jose por ser una herramienta ligera y moderna para validar los tokens JWT entrantes, lo que asegura que cada petición sea verificada de forma rápida y segura sin sobrecargar el servidor. Por último, esta combinación facilita el manejo de roles y permisos, garantizando que solo los usuarios con la autorización correcta puedan modificar el stock o acceder a información protegida.

- **ORM (Object-Relational Mapping):** Se eligió Prisma debido a que simplifica enormemente la interacción con la base de datos, permitiéndonos escribir consultas en código TypeScript limpio e intuitivo en lugar de sentencias SQL complejas y propensas a errores. También es fundamental su sistema de tipado automático, que detecta problemas en tiempo de desarrollo, si cambiamos algo en la base de datos, Prisma nos avisa al instante en qué partes del código debemos hacer ajustes. Por último, facilita la gestión de las migraciones, permitiendo evolucionar y modificar la estructura de la base de datos de manera segura y ordenada a medida que el proyecto crece.

- **Documentación de API:** OpenAPI 3. (`openapi.yaml`): Se utilizó debido a que fue proporcionado por la cátedra. Fue de gran ayuda debido a que nos permitió que los equipos externos (como Compras y Logística) entiendan exactamente cómo integrarse con nuestro sistema sin necesidad de leer nuestro código fuente. También es fundamental porque define con precisión la estructura de los datos que entran y salen, lo que evita errores de interpretación y acelera significativamente el proceso de conexión entre los distintos servicios.

- **Contenedores:** Docker & Docker Compose. Se eligió Docker junto con Docker Compose debido a que nos permite empaquetar toda la aplicación y sus dependencias en contenedores aislados, garantizando que el sistema funcione exactamente igual en la computadora de cualquier desarrollador que en el servidor de producción. También simplifica enormemente la puesta en marcha del proyecto, ya que con un solo comando se pueden levantar simultáneamente la base de datos, el sistema de autenticación y la aplicación web, sin necesidad de instalaciones manuales complejas. Por último, esta tecnología facilita la escalabilidad futura, permitiendo desplegar o replicar servicios de manera rápida y estandarizada si la demanda del sistema aumenta.

2.1.3. Herramientas Comunes / Devops

- **Control de Versiones:** Git. El sistema de control de versiones es fundamental en un proyecto de desarrollo de software, debido a su arquitectura distribuida y eficiencia.

  Arquitectura Distribuida

  Cada integrante posee una copia completa del historial del proyecto en su máquina local. Si el servidor central (GitHub) falla, el repositorio completo puede ser restaurado desde la computadora de cualquiera de los integrantes del grupo.

  Gestión Eficiente de Ramas (Branching y Merging)

  El modelo de ramas de git es su mayor ventaja, ya que permite crear, eliminar y fusionar ramas de manera extremadamente rápida y sencilla. Esto permite a cualquier persona crear una nueva rama donde podrá trabajar y realizar cambios sin que afecte a la rama principal (main).

  Colaboración y Resolución de Conflictos

  Git facilita el trabajo en equipo simultáneo. Múltiples personas pueden editar los mismos archivos al mismo tiempo. Al momento de integrar los cambios, Git fusiona automáticamente la mayoría de las modificaciones y, en casos de conflictos, ofrece herramientas para que los desarrolladores los resuelvan manualmente de manera ordenada.

  En conclusión, el mayor beneficio que ofrece Git es que permite que las personas se equivoquen y aprendan rápidamente, sin miedo a que los errores rompan algo que ya estaba funcionando. Lo mejor es que todo se hace desde la terminal de comandos; no es necesario descargar y eliminar archivos manualmente, todo lo puedes hacer desde la consola.

- **Entorno de Ejecución:** Node.js.
- **Orquestación de Contenedores:** Docker Compose (para levantar DB, Keycloak y la App).
- **Linter/Formatter:** ESLint, Prettier.

2.2. Descripción de Decisiones de Diseño y Soluciones Implementadas

Para cumplir con la consigna del subsistema de Stock, se tomaron ciertas decisiones arquitectónicas y funcionales. A continuación se detalla cada una de ellas.

2.2.1. Gestión de Stock y Productos

Se implementó un ABM (Alta, Baja, Modificación) completo para productos y categorías.

- **Modelo de Datos:** Se definieron las entidades `Producto`, `Categoria` y la relación `ProductoCategoria` en el esquema de Prisma (`schema.prisma`).

- **Atributos Clave:** El modelo `Producto` incluye campos específicos solicitados o inferidos para logística, como `pesoKg`, `dimensiones` (JSON) y `ubicacion` (JSON), permitiendo que el servicio de transporte recupere esta información crucial.

2.2.2. Lógica de Reserva (Integración con Compras)

La consigna requiere que el "Portal de Compras" reserve stock.

- **Solución:** Se expuso el endpoint `POST /api/stock/reservar`.
- **Flujo:**
  1. Recibe un `idCompra` externo y una lista de productos.
  2. Verifica la existencia del usuario y stock suficiente.
  3. Genera un registro en la tabla `Reserva` con estado `pendiente` o `confirmado`.
  4. Descuenta el stock disponible de los productos involucrados.
- **Identidad Compartida:** Se utiliza el `usuarioId` en las reservas para vincularlas con los usuarios del portal de compras, permitiendo auditoría y control, aunque la autenticación se delega a Keycloak.

2.2.3. Liberación de Stock y Expiración

Para manejar el requisito de "liberar el stock comprometido" si pasa el tiempo de reserva o se cancela:

- **Solución:** Endpoint `POST /api/stock/liberar` (o `DELETE /api/reservas/{id}`).
- **Lógica:** Permite reponer el stock de los productos asociados a una reserva específica y marca la reserva como `cancelada`.
- **Automatización (Deuda Técnica):** Actualmente, la liberación depende de que el servicio externo (Logística o Compras) invoque el endpoint de liberación o cancelación.

2.2.4. Roles y Seguridad

- **Integración con Keycloak:** Se utiliza `authMiddleware.ts` y la clase `KeycloakAuth` para validar los tokens JWT entrantes.
- **Roles:** El sistema verifica los roles presentes en `realm_access.roles` del token JWT para proteger rutas sensibles (ej. solo vendedores pueden crear productos).

3. CONCLUSIÓN

El desarrollo del presente Trabajo Práctico Integrador ha permitido consolidar una solución de software robusta y escalable para la gestión de stock, cumpliendo satisfactoriamente con los requisitos funcionales y no funcionales establecidos.

En cuanto a la tecnología seleccionada, la combinación de Next.js en el frontend y backend, junto con Tailwind CSS para el diseño de interfaces y Prisma como ORM, ha demostrado ser una elección acertada. Esta stack tecnológico no solo ha facilitado un desarrollo ágil y tipado gracias a TypeScript, sino que también garantiza un rendimiento optimizado y una experiencia de usuario fluida, alineándose con las tendencias actuales del desarrollo web.

Desde una perspectiva arquitectónica, el sistema sigue los principios de una arquitectura multicapa, desacoplando efectivamente la lógica de presentación, la lógica de negocio y el acceso a datos. Asimismo, la integración de Keycloak como proveedor de identidad externo delega la complejidad de la autenticación y autorización (OAuth2/OIDC) a una herramienta especializada y robusta.

La contenedorización de los servicios críticos, como la base de datos PostgreSQL y el servidor de Keycloak, asegura la paridad entre los entornos de desarrollo y producción, eliminando inconsistencias y simplificando el despliegue. Además, el uso de Git como sistema de control de versiones ha permitido un flujo de trabajo colaborativo eficiente y ordenado.

En síntesis, el proyecto no solo entrega un producto funcional, sino que establece una base arquitectónica sólida y profesional, demostrando la capacidad del equipo para integrar tecnologías complejas en una solución coherente y efectiva.