Tienda Sol

Plataforma de Comercio Electrónico

Trabajo Práctico Integrador
Desarrollo de Software
2C 2025 - Grupo 6

Alex Fiorenza
Ian Gabriel Sanna
Facundo Tomasetti
Ignacio Alejo Scarfo
Ignacio Castro Planas

Entrega 1: Implementación del Modelo de Objetos + Configuración del Proyecto

Fecha: 11/09/2025

Entregables

1. ***Implementación*** del Diagrama de Clases provisto, completando cada uno de los métodos mencionados y agregando aquellos que el equipo considere necesarios.
2. ***Implementación*** del endpoint “Health Check”.
3. ***Explicación*** del Git flow definido para utilizar a lo largo del proyecto.

Tecnologías a utilizar

- Backend
  - Lenguaje de programación base: JavaScript
  - Entorno de ejecución: Node.js
  - Framework: Express
- Git como Sistema de Versionado de Código, con repositorios en Github.

Respecto a las notificaciones:

- Cada vez que se realice un pedido, es necesario enviarle una notificación al Vendedor, donde se le indique quién realizó el pedido, qué productos incluye, el total del mismo y dirección de entrega.
- Cada vez que un vendedor marque un pedido como enviado, es necesario notificar al comprador.
- Si un comprador decide cancelar un pedido, es necesario notificar al vendedor.

Justificaciones iniciales

Repositorio de implementación del TP:

https://github.com/ddso-utn/tpa-2025-2c-grupo-6

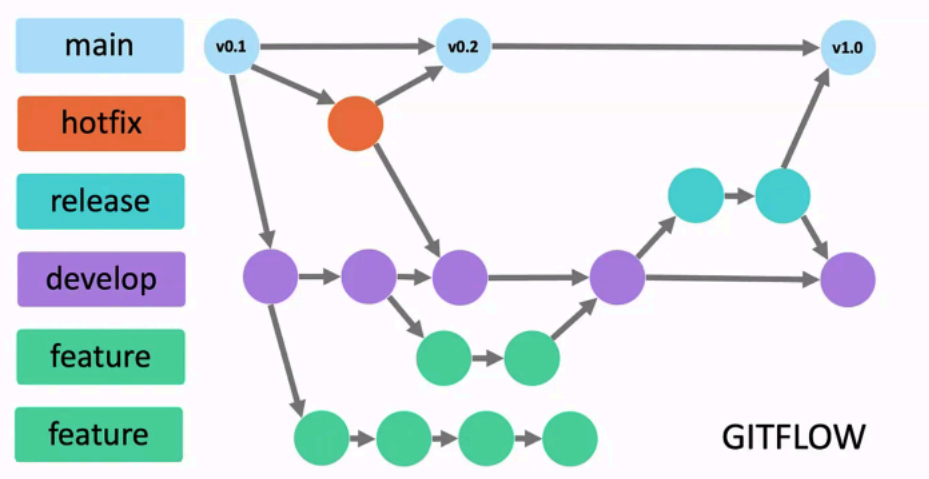
Se trata de un monorepo, que integra una aplicación frontend con Next.js y un backend con Express. La estructura monorepo facilita el desarrollo y la gestión de ambos proyectos en un único entorno.

Se tomó esta decisión, en primer lugar ya que Create React App está deprecado.
https://react.dev/blog/2025/02/14/sunsetting-create-react-app

GitFlow definido

La existencia de sólo dos ramas en GitHub, main y development, se justifica porque permite un flujo de trabajo claro y ordenado: la rama main se reserva exclusivamente para el código listo para entregar, mientras que develop concentra las tareas en curso, integrando nuevas funcionalidades y correcciones antes de ser validadas y fusionadas en main. Esta estrategia minimiza riesgos de romper la versión entregable, facilita el control de cambios y simplifica la gestión del repositorio, evitando la complejidad innecesaria que puede traer manejar múltiples ramas adicionales.

https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow



Entrega 2: Exposición de APIs + Persistencia

Fecha: 09/10/2025

Entregables

1. ***Implementación*** de la API REST del Sistema, que incluya todos los endpoints necesarios para dar solución a los requerimientos listados.
2. ***Implementación*** de la persistencia de las entidades de dominio en una base de datos NoSQL Documental.
3. ***Implementación*** de los Test Unitarios de la capa de servicios y de la capa de dominio.
4. ***Documentación*** de la API REST.

Tecnologías a utilizar

- Todas las mencionadas en la primera iteración.
- MongoDB como Base de Datos Documental NoSQL.
- Jest como framework de Testing Unitario.
- Swagger como Herramienta de Documentación de APIs. Está permitido la utilización de alguna dependencia que genere el contenido de Swagger.

Justificaciones iniciales

1. Introducción

Esta documentación presenta la justificación de diseño para la API REST de la plataforma "Tienda Sol", desarrollada en la segunda iteración del proyecto. La API se enfoca en la gestión de pedidos, la búsqueda y visualización de productos, y la visualización de notificaciones, cumpliendo con los requerimientos funcionales especificados. Todos los datos manipulados son persistentes, asumiendo el uso de una base de datos como MongoDB para almacenar información de usuarios, productos, pedidos y notificaciones de forma no relacional

El diseño sigue el enfoque REST, utilizando métodos HTTP estándar (GET, POST, PUT, PATCH, DELETE) para operaciones CRUD, con énfasis en distintos atributos de calidad como performance y escalabilidad. Se incorpora paginación para consultas eficientes, filtros y ordenamientos para mejorar la usabilidad, y validaciones en los esquemas para garantizar la integridad de los datos. Además, se prevé la implementación de tests unitarios en la capa de servicios para validar la lógica de negocio.

La especificación OpenAPI 3.0.3 define la estructura de la API, incluyendo paths, parámetros, request/response bodies y esquemas, facilitando la generación de documentación automática (con Swagger UI) y la integración con clientes.

2. Enfoque General del Diseño

2.1 Principios REST

- Recursos y URIs: Cada recurso (pedidos, productos, notificaciones) se representa como un endpoint lógico. Por ejemplo, /pedidos para la colección de pedidos y /pedidos/{pedidoId} para instancias específicas. Esto promueve la uniformidad y la reutilización de lógica.
- Métodos HTTP:
    - POST para creación (crear pedido o producto).
    - GET para consultas (listar productos con filtros).
    - PATCH para actualizaciones parciales (cancelar o enviar un pedido, marcar notificación como leída), ya que no siempre se modifica todo el recurso.
    - PUT para actualizaciones completas (actualizar producto).
    - DELETE para eliminación (producto).
- Códigos de Respuesta HTTP: Se usan códigos estándar (200 OK, 201 Created, 400 Bad Request, 404 Not Found) para indicar el resultado de las operaciones, con cuerpos de respuesta en JSON que incluyen mensajes de error o datos relevantes.
- Persistencia: Todos los endpoints interactúan con una base de datos persistente. Por ejemplo, la creación de un pedido valida el stock de productos y actualiza el inventario atómicamente para evitar inconsistencias.
- Formato de Datos: JSON como único formato, con esquemas validados (usando bibliotecas como Zod en el backend).

2.2 Paginación, Filtros y Ordenamiento

Para manejar grandes volúmenes de datos (listas de productos o notificaciones), se implementa paginación obligatoria en endpoints de consulta múltiple:

- Parámetros page (default: 1) y limit (default: 10, max: 100) para dividir resultados.
- Respuestas incluyen metadatos como totalColecciones, totalPaginas para navegación frontend. Esto mejora la eficiencia, reduce la carga en el servidor y previene timeouts.

Los filtros y ordenamientos se aplican vía query params, permitiendo búsquedas dinámicas sin endpoints adicionales, alineado con REST.

2.3 Manejo de Errores implementados

- Errores de negocio (stock insuficiente al crear pedido) devuelven 400 con un objeto { error: "mensaje descriptivo" }.
- Errores del servidor (500) incluyen detalles para debugging, pero en producción se ocultan.

- Validaciones en request bodies aseguran datos requeridos y tipos correctos.

2.4 Tests Unitarios

Se implementarán tests unitarios en la capa de servicios (usando Jest) para cubrir:

- Lógica de validación de stock en creación de pedidos.
- Aplicación de filtros y paginación en servicios de productos.
- Transiciones de estado en pedidos (ej: de PENDIENTE a CANCELADO).
- Generación y marcado de notificaciones. Esto garantiza robustez y facilita el mantenimiento.
- Y otras cuestiones que tienen que ver con la implementación de dominio

3. Diseño de Endpoints por Módulo

3.1 Gestión de Pedidos

Los requerimientos exigen el ciclo de vida completo de un pedido: creación (con validación de stock), cancelación (antes de envío), consulta de historial por usuario, y marcado como enviado por el vendedor.

- **Creación (POST /pedidos)**:
  - Justificación: Se encarga de validar el stock disponible por ítem (restando del stock del producto si es suficiente), actualiza totalVendido en productos e inicializa historial de estados con PENDIENTE. Requiere compradorId, items, moneda y direccionEntrega para completitud. Respuesta incluye pedidoId para tracking inmediato.
  - Alineación: Cumple con "Creación de un pedido, validando el stock disponible".
- **Consulta General (GET /pedidos) y por Usuario (GET /pedidos/usuario/{usuarioId})**:
  - Justificación: El endpoint general lo pensamos para admins o vendedores; el GET por usuario para historial personal (resumido para privacidad). Usa PedidoResumido para ocultar detalles sensibles como la dirección completa.
  - Alineación: Cumple con "Consulta del historial de pedidos de un usuario".
- **Cancelación (PATCH /pedidos/{pedidoId}/cancelar)**:
  - Justificación: Sólo se permite si el estado es PENDIENTE o CONFIRMADO (no enviado). Requiere compradorId para autorización. Actualiza historial con motivo y restaura stock. PATCH para cambio parcial de estado.
  - Alineación: Cumple con "Cancelación de un pedido antes de que haya sido enviado".

- **Marcado como Enviado (PATCH /pedidos/{pedidoId}/enviar)**:
    - Justificación: Sólo se permite si no está cancelado. Requiere vendedorId para autorización. Transición a ENVIADO en historial. PATCH para actualización mínima.
    - Alineación: Cumple con "Marcado de un pedido como enviado por parte del vendedor".

Además, para permitir la persistencia del pedido en MongoDB se realizó un esquema Pedido que incluye historialEstados para auditar cambios, y enums para estados estandarizados, asegurando consistencia.

3.2 Búsqueda y Visualización de Productos

Se requiere listar productos de un vendedor con filtros (nombre, categoría, descripción, rango de precios), paginación y ordenamiento (precio asc/desc, más vendido).

- **Creación (POST /productos), Actualización (PUT /productos/{id}) y Eliminación (DELETE /productos/{id})**:
    - Justificación: Están asociados a usuarioId (vendedor). Incluyen stock, totalVendido (inicial 0, actualizado en pedidos) y activo para visibilidad. PUT reemplaza completamente; DELETE soft-delete o delete lógico (set activo: false) para preservar datos históricos.
    - Alineación: Soporte base para gestión de inventario.
- **Listado General (GET /productos) y por Vendedor (GET /productos/vendedor/{vendedorId})**:
    - Justificación: Esta permite los siguientes filtros via query: nombre, descripcion (búsqueda parcial con LIKE o regex), categoría (exacta por ID), precioMin/Max (rango numérico). Ordenamiento: sortParam con enums para SQL/Mongo sort (e.g., {precio: 1} para asc). Paginación con offset/limit. mas_vendidos ordena por totalVendido descendente. Respuesta paginada con ProductoRespuesta (incluye timestamps para auditoría).
    - Alineación: Cumple con "Listar los productos de un vendedor en particular, con filtros (término de búsqueda, rango de precios), paginación y ordenamiento".

Esto permite búsquedas eficientes, indexando campos como título, descripción y precio en la BD.

3.3 Visualización de Notificaciones

Se implementan notificaciones para confirmación/cancelación/envío de pedidos, con endpoints para sin leer, leídas y marcadas.

- **Sin Leer (GET /notificaciones/unread/{usuarioId}) y Leídas (GET /notificaciones/read/{usuarioId})**:
    - Justificación: Son filtradas por usuarioId y leída: false/true. Utilizan paginación para listas largas. tipo enum para categorizar (como "pedido" para confirmaciones).
    - Alineación: Cumple con "Endpoint para obtener la lista de notificaciones sin leer/leídas de un usuario".
- **Marcado como Leída (PATCH /notificaciones/{id}/read)**:
    - Justificación: Actualiza el campo de leída: true y fechaLectura. Se utiliza PATCH para cambio atómico. No requiere body, solo ID.
    - Alineación: Cumple con "Endpoint para marcar una notificación como leída".

El esquema Notificación incluye título, mensaje y timestamps. Se generan automáticamente en eventos

4. Esquemas y Modelos de Datos

Los esquemas de la API se implementan utilizando Mongoose para MongoDB, lo que permite una modelación flexible de documentos NoSQL con soporte para referencias (populate), validaciones integradas y optimizaciones de rendimiento mediante índices. Esta elección se justifica por la naturaleza no relacional de MongoDB, que facilita el manejo de arrays embebidos (como ítems de pedidos, fotos de productos) y subdocumentos (como historial de estados), reduciendo joins complejos y mejorando la velocidad de consultas.

4.1 Modelo de Usuario

El modelo **Usuario** representa a compradores y vendedores, sirviendo como base para referencias en otros modelos (vendedor en productos, comprador en pedidos).

- Estructura Principal:
    - **nombre**: String requerido y trimmeado para limpieza de datos.
    - **email**: String único, requerido, trimmeado, en minúsculas y validado con regex para formato estándar (e.g., evita emails inválidos como "user@invalid").
    - **telefono**: String opcional, trimmeado para normalización.
    - **dirección**: Subobjeto embebido con calle, ciudad y codigoPostal (todos opcionales y trimmeados), permitiendo almacenamiento flexible de datos geográficos sin un modelo separado.
    - **Activo**: Booleano default true, para soft-delete o desactivación de cuentas.
    - **fechaRegistro**: Date default Date.now, para tracking de onboarding.
- Justificación: Este esquema es minimalista pero robusto, enfocándose en datos esenciales para autenticación y perfiles. La validación de email

previene errores comunes, y el subobjeto dirección simplifica el modelo sin sacrificar extensibilidad (puede expandirse para lat/lon en futuras iteraciones).

4.2 Modelo de Categoría

El modelo **Categoría** es un esquema simple para clasificar productos, permitiendo referencias múltiples.

- Estructura Principal:
    - **nombre**: String requerido y trimmeado, para un identificador único y legible.
- Justificación: Como catálogo estático, se mantiene liviano para facilitar la creación y mantenimiento manual/administrativo. Se usa en arrays de referencias en productos (categorías: [ObjectId ref 'Categoria']), permitiendo productos multi-categorías sin duplicación de datos. Esto soporta filtros por categoría en endpoints de productos, con queries eficientes vía populate o agregaciones.

4.3 Modelo de Producto

El modelo **Producto** gestiona el inventario, con énfasis en visibilidad, ventas y filtros de búsqueda.

- Estructura Principal:
    - **vendedor**: ObjectId ref 'Usuario', requerido, para asociar productos a un vendedor específico.
    - **titulo**: String requerido y trimmeado, para búsquedas por nombre.
    - **descripción**: String opcional y trimmeado, para búsquedas textuales detalladas.
    - **Categorías**: Array de ObjectId ref 'Categoría', para clasificación flexible (un producto puede tener múltiples).
    - **precio**: Number requerido, mínimo 0, para validación económica.
    - **moneda**: String enum ["PESO_ARG", "DOLAR_USA", "REAL"], requerido, para soporte multi-moneda regional (alineado con pedidos).
    - **stock**: Number default 0, mínimo 0, para control de inventario.
    - **totalVendido**: Number default 0, mínimo 0, para métricas de popularidad (actualizado en creación de pedidos).
    - **Fotos**: Array de strings (URLs), para multimedia flexible.
    - **Activo**: Booleano default true, para soft-delete o pausar ventas.
- Justificación: El esquema soporta los requerimientos de búsqueda y visualización: campos como título, descripción y categorías habilitan filtros parciales/exactos; precio y moneda permiten rangos y ordenamientos; stock y totalVendido validan pedidos y ordenan por "más vendido". Arrays para categorías y fotos ofrecen flexibilidad sin normalización excesiva. En OpenAPI, ProductoCrear mapea inputs a este modelo, y ProductoRespuesta

expone outputs enriquecidos con timestamps y totalVendido. Soft-delete via activo preserva historial de ventas.

4.4 Modelo de Pedido

El modelo **Pedido** captura el ciclo de vida completo, con subesquemas para ítems e historial.

- Subesquemas:
    - **ItemPedidoSchema**: Incluye **productoId** (ObjectId ref 'Producto', requerido), **cantidad** (Number requerido) y **precioUnitario** (Number requerido).

      Justificación: Desglosa pedidos en ítems atómicos para calcular totales y validar stock individualmente, con precio fijo al momento de compra (evita inflación).

    - **HistorialEstadoSchema**: Incluye **fecha** (Date default Date.now), **estado** (String enum ["PENDIENTE", "CONFIRMADO", "EN_PREPARACION", "ENVIADO", "ENTREGADO", "CANCELADO"], requerido), **quien** (ObjectId ref 'Usuario', requerido) y **motivo** (String opcional).

      Justificación: Proporciona auditoría inmutable de transiciones, rastreando quién (comprador/vendedor) y por qué (e.g., cancelación por stock bajo), esencial para disputas o analíticas.

- Estructura Principal:
    - **compradorId**: String ref 'Usuario', requerido (nota: usa String en schema, pero idealmente ObjectId para consistencia).
    - **items**: Array de ItemPedidoSchema, para múltiples productos.
    - **moneda**: String enum ["PESO_ARG", "DOLAR_USA", "REAL"], requerido, consistente con productos.
    - **direccionEntrega**: Object requerido (embebido, con campos como calle, ciudad; extensible para lat/lon).
    - **estado**: String enum (mismo que historial), default "PENDIENTE".
    - **fechaCreacion**: Date default Date.now.
    - **historialEstados**: Array de HistorialEstadoSchema, inicializado con el estado inicial.
- Justificación: Este esquema embebido reduce complejidad de relaciones, permitiendo queries atómicas (e.g., obtener pedido completo con items populated). Enums estandarizan estados para lógica de negocio (e.g., solo cancelar si PENDIENTE). El total se calcula dinámicamente en servicios (suma de ítems), no almacenado para evitar inconsistencias.

4.5 Modelo de Notificación

El modelo **Notificacion** maneja alertas push/in-app para eventos clave.

- Estructura Principal:
    - **usuarioDestinoId**: ObjectId ref 'Usuario', requerido, con índice para queries rápidas.
    - **mensaje**: String requerido y trimmeado (nota: en OpenAPI incluye **título** y **tipo** enum; aquí se simplifica a mensaje único, pero puede expandirse).
    - **leída**: Booleano requerido, default false, con índice para filtrado eficiente.
    - **fechaCreacion**: Date default Date.now.
    - **fechaLectura**: Date default null, para tracking de interacción.

5. Conclusión

Este diseño de API REST cumple integralmente con los requerimientos de la iteración, priorizando usabilidad, eficiencia y mantenibilidad. La persistencia asegura durabilidad de datos, mientras que paginación/filtros optimizan performance. La implementación de tests unitarios en servicios validará la lógica crítica. La especificación OpenAPI facilita el desarrollo frontend y la colaboración, posicionando la plataforma para expansiones futuras como pagos o reseñas. Si se requieren ajustes, se pueden iterar basados en feedback.

Entrega 3: UI + Integración con Backend

Fecha: 06/11/2025

Entregables

1. ***Maquetado*** de todas las pantallas que el equipo considere necesarias para dar cumplimiento a todos los requerimientos funcionales de la iteración 2, con la correspondiente navegabilidad entre ellas y ajuste al cumplimiento de los requerimientos no funcionales listados en esta iteración.
2. ***Implementación*** de funcionalidad completa de "*Búsqueda y Visualización de Productos*, incluyendo su integración con el backend.
3. ***Implementación*** del carrito de compras del lado cliente.
4. ***Justificación*** acerca del cumplimiento de los requerimientos no funcionales.

Tecnologías a utilizar

- Todas las mencionadas en las anteriores entregas.
- Next.js como Framework para el desarrollo de nuestro Frontend.
- React como biblioteca de generación de componentes para nuestro Frontend.
- Axios como Cliente HTTP para integrar nuestro Frontend con nuestro Backend.

Justificaciones iniciales

1. Introducción

Esta documentación presenta la justificación de diseño de la interfaz de usuario de la plataforma "Tienda Sol", desarrollada en la tercera iteración del proyecto. La interfaz permite al usuario interactuar con todas las funcionalidades que la tienda ofrece, tales como visualizar productos, filtrar búsquedas, armado de carrito y gestión de pedidos. Todas las pantallas buscan ofrecer una experiencia y usabilidad adecuada, con botones visibles, acciones claras, paleta de colores coherente y tipografía correcta. Se incorpora paginación para consultas eficientes, filtros y ordenamientos para mejorar la usabilidad, y validaciones en los esquemas para garantizar la integridad de los datos. Además, se prevé la implementación de tests e2e para validar la lógica de negocio.

2. Maquetado de interfaces

2.1 Home

La pantalla principal de Tienda Sol tiene como finalidad ofrecer al usuario una primera experiencia atractiva, clara y funcional al ingresar a la plataforma. Desde ella se promueve la exploración de productos, se destaca la navegación hacia el catálogo completo y se refuerza la identidad de marca con un diseño visual coherente.

Diseño de Interfaz

La pantalla se organiza mediante una estructura de tipo layout con componentes reutilizables (Navbar, Footer y ProductCard), asegurando consistencia visual y modularidad. Se emplea Container y Box de Material UI para mantener alineación, márgenes y espaciado uniforme.

- Elementos visuales:
    - Se utiliza una paleta de colores neutra y coherente (grises y azul Oxford) que transmite profesionalismo y legibilidad.
    - La tipografía mantiene jerarquías visuales claras mediante Typography con tamaños adaptativos (xs, sm, md) que garantizan la correcta visualización en distintos dispositivos.
    - Los botones cuentan con labels descriptivos y atributos de accesibilidad (aria-label) para favorecer la inclusión y la comprensión de las acciones.

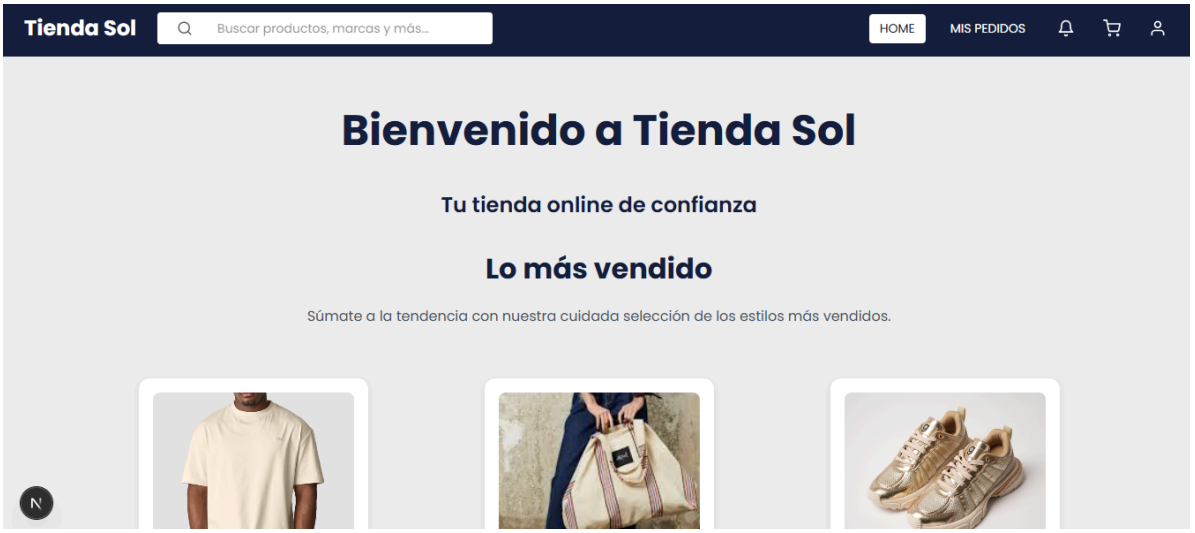
- Usabilidad y accesibilidad:
    - Se aplican roles y etiquetas ARIA (role="region", aria-label) para permitir la navegación asistida por lectores de pantalla.
    - El diseño es responsive, adaptando grillas y carruseles según el ancho del dispositivo gracias a las propiedades responsive en el componente Slider.
    - Se minimiza la sobrecarga visual mediante una disposición limpia y uso de espacio en blanco.

Integración con el Backend

- Se utiliza Axios como cliente HTTP para integrar el frontend con el backend de productos. El hook *useEffect* ejecuta una llamada GET a /productos, lo que permite visualizar dinámicamente los productos más vendidos en la Home.

- El estado local (*useState*) almacena los datos recibidos y controla los mensajes de carga (Cargando productos...) y error (No se pudieron cargar los productos.), asegurando feedback inmediato al usuario.

- Cada producto se renderiza mediante el componente ProductCard, garantizando una visualización homogénea y una arquitectura de componentes desacoplada.



2.2 Products (comprador)

La pantalla Productos permite al usuario explorar el catálogo completo de Tienda Sol, con herramientas de búsqueda, filtrado y ordenamiento que optimizan la experiencia de compra. Su diseño busca equilibrar claridad visual, eficiencia en la carga y fácil interacción con los productos.

Diseño de Interfaz

La vista combina un panel lateral de filtros y un grid central de productos, priorizando la organización y legibilidad. Se emplean componentes de Material UI (Container, Box, Select, Typography) junto con clases utilitarias de Tailwind para garantizar un diseño adaptable y limpio. La interfaz es responsive, ocultando el panel lateral en pantallas pequeñas y reorganizando los productos mediante un grid flexible.

Filtros y ordenamientos: Incluye controles para filtrar por precio, categoría y orden de resultados, además de un buscador integrado en el Navbar. Los filtros están acompañados de etiquetas y placeholders claros, manteniendo coherencia visual con el resto del sistema.

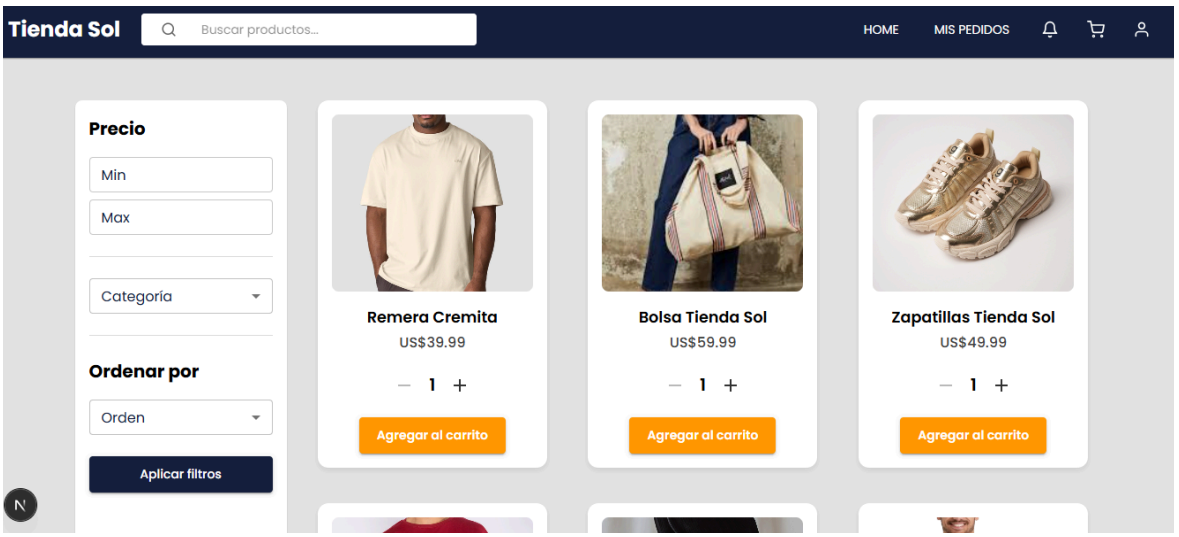
Integración con el Backend

Se implementa la comunicación con el servidor mediante Axios, consumiendo los endpoints:

- GET /productos para obtener productos con soporte de paginación, búsqueda y filtros dinámicos.
- GET /productos/categorías para cargar categorías disponibles desde el backend.

Los parámetros enviados (paginación, categoría, precio, orden) se construyen dinámicamente en la consulta, garantizando una búsqueda precisa y optimizada.

Los datos recibidos se almacenan en estado local y actualizan el DOM de forma reactiva, sin recargar la página.



2.2 Products (vendedor)

La pantalla Productos del Vendedor permite al usuario con rol de vendedor visualizar, editar y eliminar los productos que tiene publicados en Tienda Sol. Su propósito es ofrecer una interfaz sencilla para la gestión del catálogo propio, facilitando la administración del stock, precios y descripciones.

Diseño de Interfaz

- Los botones de acción (editar y eliminar) están claramente diferenciados, reforzando la usabilidad y evitando errores.

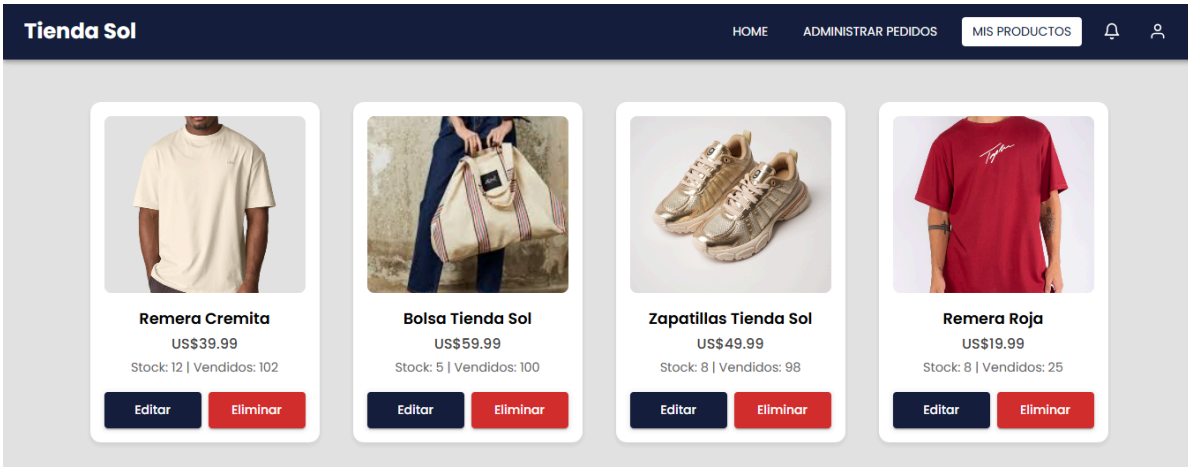
- Diálogo de edición: El formulario de modificación se implementa con un Dialog de Material UI, que permite editar en contexto sin abandonar la pantalla. Incluye campos validados de texto, número y descripción, con un diseño consistente y etiquetas descriptivas.

Interacción y Navegación

- El vendedor puede editar un producto haciendo clic en el botón correspondiente, lo que abre un modal donde se modifican los campos de manera controlada.

- La opción eliminar y editar producto simulan la acción con confirmación en consola, pensadas para integrarse posteriormente con la API real.

- Se incluye paginación para mejorar la legibilidad cuando la lista de productos es extensa, evitando sobrecarga visual.



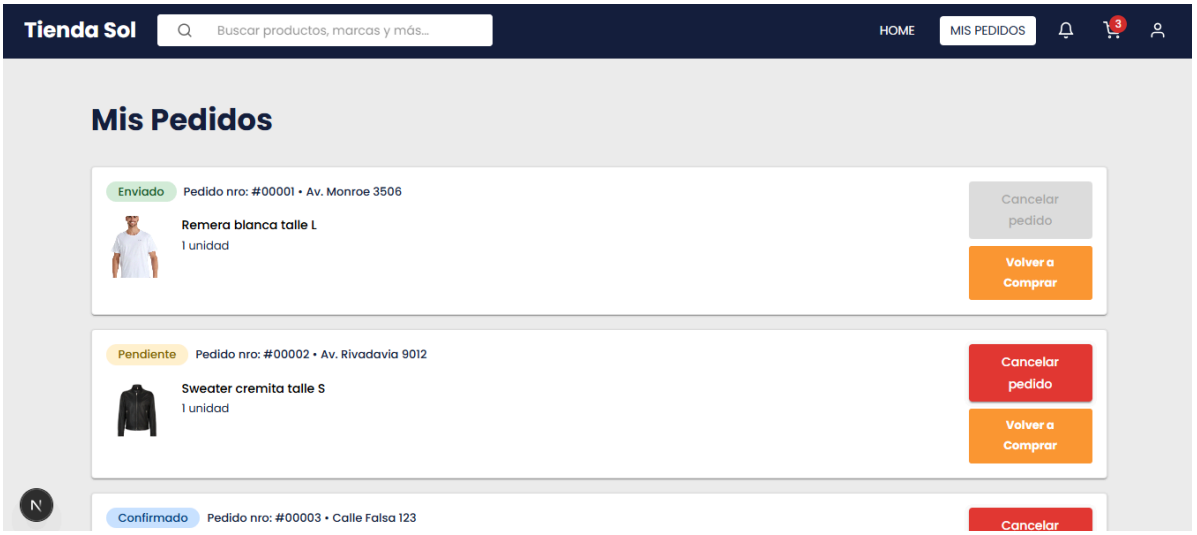
2.3 Pedidos (comprador)

La pantalla Pedidos del Comprador permite al usuario visualizar el historial y estado de sus compras dentro de Tienda Sol. Desde esta sección, puede consultar los detalles de cada pedido, su dirección de entrega y los productos adquiridos.

El objetivo principal es ofrecer **transparencia y trazabilidad** del proceso de compra, mostrando de forma clara el estado de cada pedido (pendiente, confirmado, enviado, cancelado, etc.).

Visualización de pedidos: Cada pedido se muestra como un bloque con número de orden, dirección de entrega, estado actual y los productos asociados. El diseño mantiene una estética minimalista y limpia, priorizando la legibilidad de la información y la coherencia visual.

Adaptabilidad: El diseño es totalmente responsive, adaptándose a distintos tamaños de pantalla sin perder legibilidad ni funcionalidad.

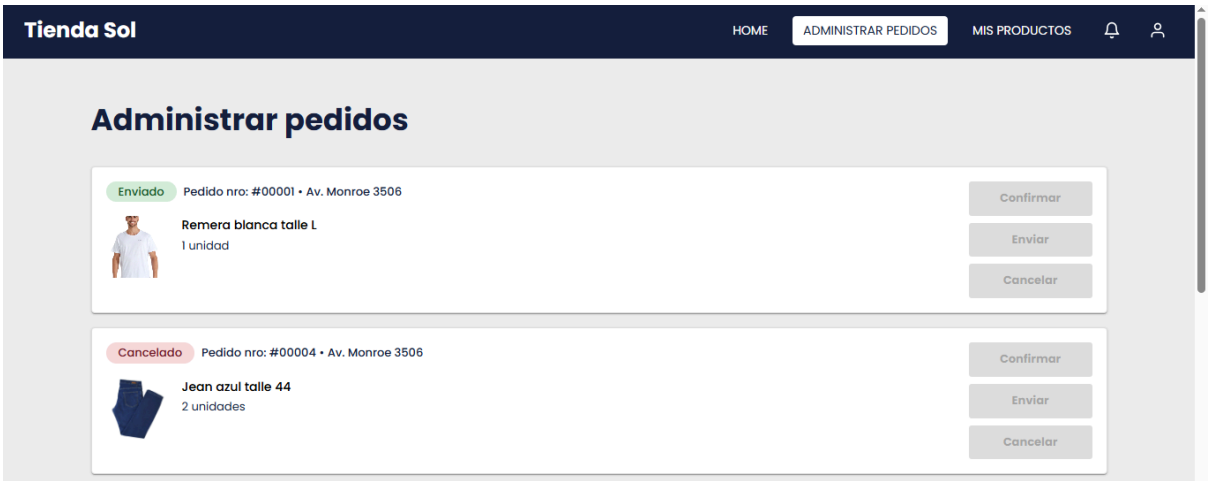


Pantalla del Vendedor

La pantalla del Vendedor comparte la misma base estructural y lógica de presentación (OrderRow y Pagination), pero se diferencia de las acciones disponibles:

- El vendedor gestiona los pedidos (confirmar, enviar, cancelar).
- El comprador consulta o cancela sus propios pedidos.

De este modo, ambas pantallas mantienen consistencia visual y funcional, adaptando las acciones según el rol autenticado en la sesión.



2.4 Notificaciones

La pantalla Notificaciones permite al usuario visualizar los avisos relacionados con su actividad en la plataforma, como actualizaciones de pedidos, confirmaciones, envíos o cancelaciones. Su propósito es mantener una comunicación clara y centralizada entre el sistema y el usuario, evitando que deba revisar manualmente cada pedido o acción.

Diseño de Interfaz

La vista mantiene la estructura general del sitio con Navbar, Footer y contenido principal centrado (Container). Cada notificación se muestra mediante el componente NotificationCard, con un diseño limpio, espaciado y jerarquía visual clara. El contador dinámico de “sin leer” resalta la información más relevante, favoreciendo la atención del usuario.

Interacción y Navegación

- El usuario puede marcar notificaciones como leídas o no leídas mediante interacción directa.
- Se utiliza paginación para navegar cómodamente entre múltiples avisos.
- La información se actualiza en tiempo real en la interfaz, sin recargar la página.
- Las notificaciones se obtendrán según la sesión, mostrando solo los avisos correspondientes a cada usuario.

2.5 Carrito

La pantalla de Carrito permite al usuario administrar los productos seleccionados antes de avanzar al checkout. Su implementación se basa en un estado global mediante **Context API (useContext)**, lo que evita el paso excesivo de props y permite acceder al carrito desde cualquier parte de la aplicación (Navbar, ProductCard, Checkout, etc.).

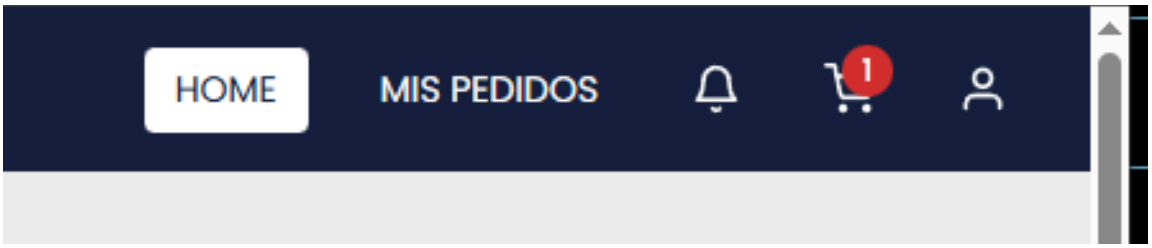
Implementación del Estado (CartContext)

El carrito se gestiona con un **CartContext**, que centraliza todas las operaciones:

- addToCart(product, quantity)
- updateQuantity(productId, qty)
- removeFromCart(productId)
- clearCart()

El contexto almacena los ítems en un arreglo de objetos que extienden Product con el campo cantidad.
Para persistencia local se utiliza **localStorage**, cargando el carrito al iniciar la app y guardando los cambios en cada actualización del state.

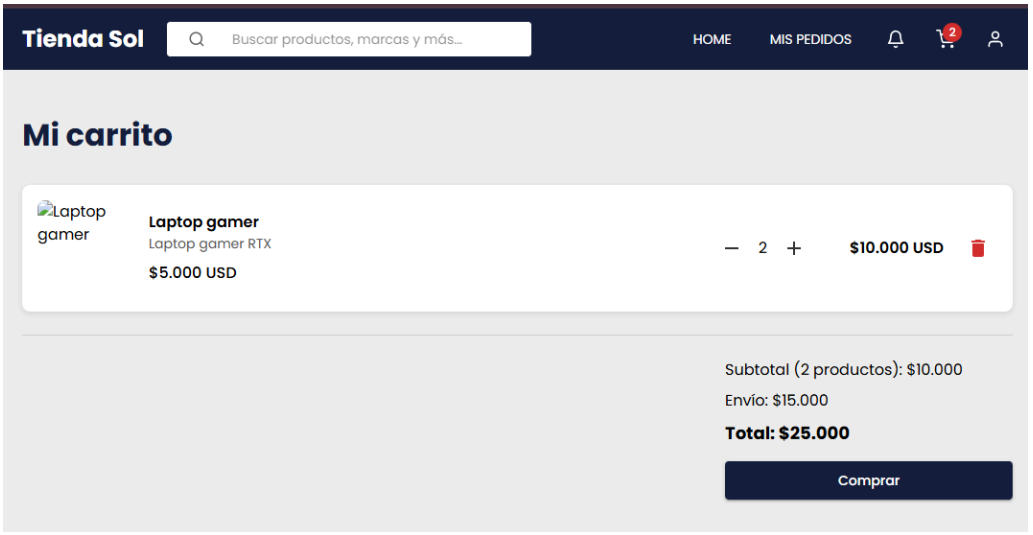
Esto permite mantener el carrito aunque el usuario recargue la página o cambie de vista.



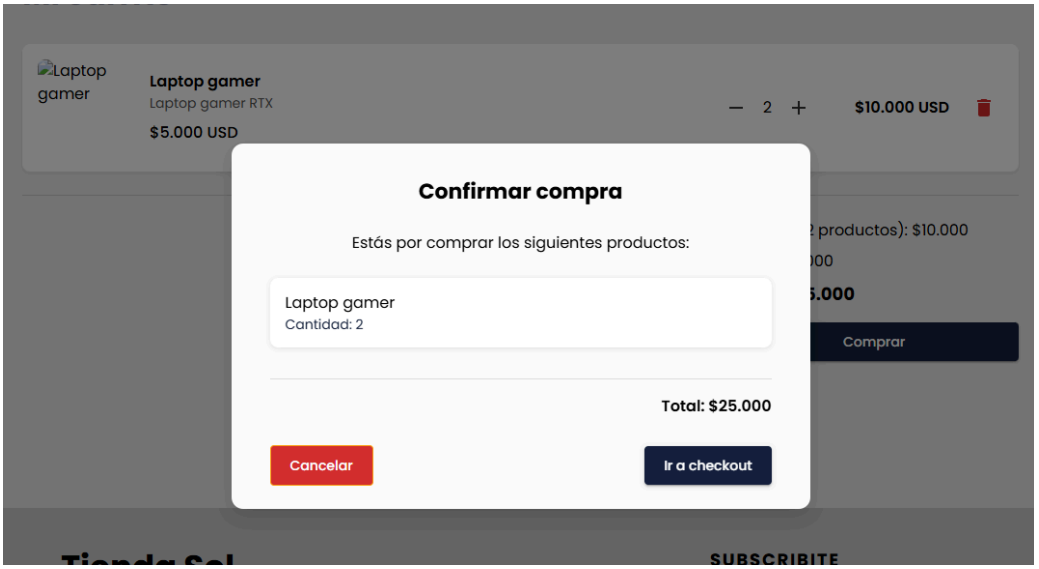
Pantalla del Carrito (CarritoPage)

La vista consume el contexto mediante useCart() y ofrece las funcionalidades principales:

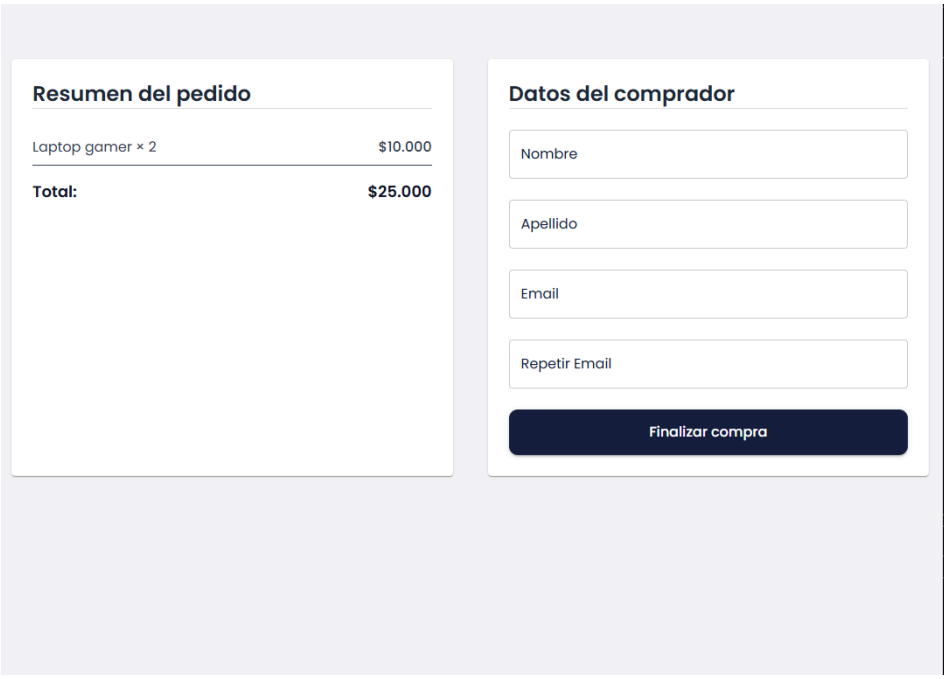
- **Incrementar y decrementar cantidades**, con validaciones según stock disponible.
- **Eliminar productos** con un botón accesible.
- **Cálculo automático de subtotal, envío y total** directamente desde el estado del contexto.
- **Modal de confirmación**, para revisar los productos antes de pasar al checkout.
- **Redirección a /checkout** mediante router.



Luego de tocar comprar, sale un modal para confirmar la misma que permite asegurar que el usuario realmente quiere realizar esa acción



Al confirmar la compra, te envía a la pantalla de checkout, que muestra un resumen del pedido y al enviar los datos personales realiza un POST a pedidos



2.6 Navbar

El sistema utiliza un componente de navegación adaptable según el tipo de usuario (**buyer** o **seller**) y según el dispositivo (**mobile** o **desktop**). Está construido con **Material UI**, **Next.js** y componentes reutilizables.

BuyerNavbar

Es la barra de navegación para compradores. Sus funciones principales:

- **Búsqueda integrada**, con formulario que redirige a /products?search=....

- **Badge del carrito**, conectado al **CartContext**, mostrando en tiempo real el total de productos.
- **Menú de usuario** (UsuarioMenu), accesible desde cualquier dispositivo.
- **Notificaciones** con indicador visual si estás en la ruta correspondiente.
- **Responsividad completa**:
    - En mobile se usa un **Drawer lateral** para los enlaces.
    - En desktop se muestran los enlaces directamente en la barra.
- **Enlaces dinámicos** y estilos que resaltan el link activo usando usePathname().

SellerNavbar

Versión simplificada enfocada en el vendedor:

- No incluye buscador ni carrito.
- Contiene los enlaces del vendedor (HOME, ADMINISTRAR PEDIDOS, MIS PRODUCTOS).
- También alterna entre vista compacta en mobile con Drawer y vista completa en desktop.
- Incluye notificaciones y menú de usuario.

Navbar (wrapper)

Es un componente unificador que decide **qué Navbar renderizar** según el tipo de usuario:

- Si userType === 'buyer' → muestra BuyerNavbar.
- Si userType === 'seller' → muestra SellerNavbar.
- Soporta un modo minimal usado en pantallas especiales (login, registro, etc.), donde solo se muestra el logo.

3. Manejo de sesiones (TODO próxima entrega)

El backend utiliza **Clerk** para gestionar autenticación y control de acceso. El middleware global (clerkMiddleware) se configura en el servidor y expone la información de sesión en cada request.

Para rutas que requieren login se usa requireAuth(), que valida automáticamente la sesión y devuelve 401 si no existe.

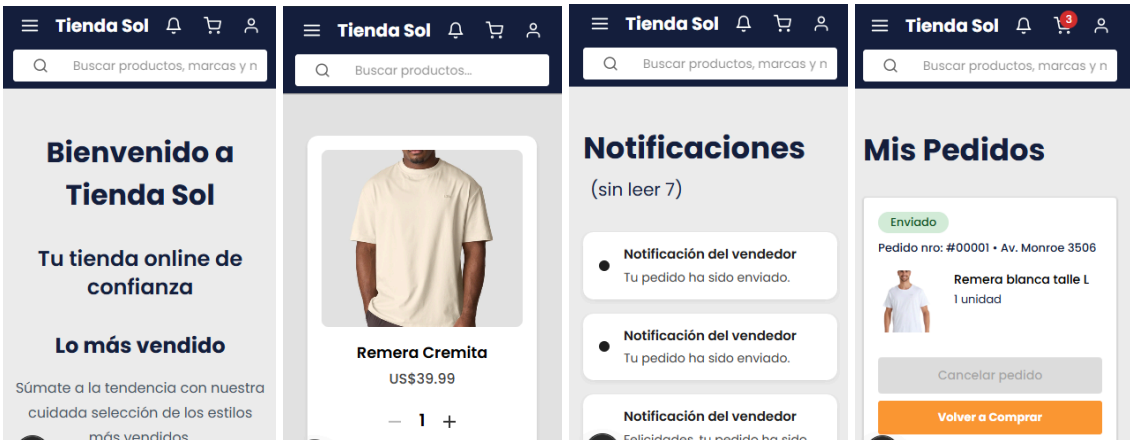
Algunas rutas permiten autenticación opcional, leyendo req.auth?.userId para adaptar la respuesta según el usuario.

Además, se implementó un middleware propio requireRole, que verifica los roles asignados al usuario (buyer, seller, etc.) y restringe el acceso según corresponda. De esta forma se pueden crear endpoints exclusivos para vendedores, compradores o ambos.

Esta estructura asegura rutas públicas, rutas protegidas por sesión y rutas protegidas por rol, manteniendo una arquitectura clara y escalable.

4. Responsive

Vistas de la interfaz desde un dispositivo móvil



5. Conclusión

Requerimientos no funcionales cubiertos en la entrega:

- Rendimiento: Se limita la carga inicial de productos (limit=3) y se usa paginación para evitar sobrecargar el renderizado.
- Usabilidad: Interfaz intuitiva, elementos reconocibles y botones visibles.
- Accesibilidad: Uso de etiquetas ARIA y roles semánticos en los elementos principales.
- Reutilización de componentes: Navbar, Footer y ProductCard se emplean en múltiples pantallas, promoviendo la modularidad.
- Escalabilidad: La estructura basada en componentes y hooks facilita la ampliación del proyecto.

Entrega 4: Entrega final

Fecha: 27/11/2025

Entregables

1. ***Implementación*** de todas las funcionalidades completas, con la correspondiente integración con el Backend.
2. ***Despliegue*** del aplicativo en la nube (Backend + Frontend).
3. ***Documentación*** acerca del despliegue del aplicativo que contenga el detalle de los pasos a seguir para desplegar el aplicativo por primera vez y por cada vez que se quiere subir a producción una nueva release.
4. ***Tests de integración*** en la Capa de Controladores del Backend: solamente es necesario realizar 1 test, cuyo endpoint a testear queda a criterio del equipo.
5. ***Tests E2E*** en el Frontend: solamente es necesario realizar 1 test, cuya funcionalidad a testear queda a criterio del equipo.

Tecnologías a utilizar

- Todas las mencionadas en las anteriores entregas.
- Render como Cloud Application Platform para el despliegue de nuestro Backend (se pueden utilizar algunas otras alternativas, a criterio del equipo y en común acuerdo con el equipo docente).
- Netlify como plataforma para despliegue de nuestro Frontend (se pueden utilizar algunas otras alternativas, a criterio del equipo y en común acuerdo con el equipo docente).
- Cypress como herramienta de Testing E2E. Está permitido buscar y utilizar alguna otra herramienta alternativa (como Jest, por ejemplo).
- Jest como framework para Testing de Integración

1. Despliegue del Frontend

Para el frontend utilizamos Vercel debido a que la plataforma está optimizada para aplicaciones basadas en Next.js y ofrecen integración nativa con GitHub. Implementamos un GitHub Action que, en cada push a main, ejecuta la CLI de Vercel y dispara automáticamente el deploy. Solo fue necesario generar una API_KEY desde la plataforma y configurarla como variable de entorno en GitHub.

https://vercel.com/kb/guide/how-can-i-use-github-actions-with-vercel

Esto garantiza despliegues rápidos y reproducibles sin intervención manual.
Ventajas:

- Deploy automático por CI/CD sin pasos manuales.
- Soporte nativo para Next.js (renderizado híbrido SSR/ISR).
- Rollbacks instantáneos.

- Manejo fácil de variables de entorno.

Desventajas:
- Límites en planes gratuitos (tiempos de build).
- Dependencia de una plataforma propietaria.

2. Despliegue del Backend

El backend se desplegó en Render, utilizando nuestra imagen Docker. Render permite subir un servicio basado en un contenedor propio sin modificar el código, lo cual simplifica el pipeline. Render nos ofrece 512 MB de RAM, lo cual es justo para nuestro backend y por eso elegimos delegar la autenticación en Clerk.

3. Base de Datos

La base de datos es MongoDB Atlas, porque:
- Tiene alta disponibilidad por defecto.
- Fácil conexión desde Render mediante IP allowlist.
- No requiere mantenimiento local y escala automáticamente.

4. Autenticación con Clerk (SSO en la nube)

Elegimos Clerk para autenticación y manejo de sesiones.
Se justificó principalmente porque:
- Al estar 100% gestionado en la nube, evita que tengamos que almacenar sesiones en memoria local.
- Esto reduce el consumo de RAM del backend, algo crítico porque Render solo da 512 MB en el plan gratuito.
- Clerk maneja tokens, sesiones, refresh, MFA y roles sin procesamiento local.

Ventajas:
- Ahorro de RAM (evitamos almacenar sesiones/server state).
- Seguridad y escalabilidad con mínima configuración.
- SDK simple para frontend y backend.

Desventajas:
- Servicio externo (dependencia).
- Requiere configurar claves públicas y privadas en ambos entornos.

5. CI/CD con GitHub Actions

Implementamos un flujo CI/CD simple:
- Cada vez que mergeamos a main, GitHub Actions:
    - construye el proyecto,
    - valida que compile
    - ejecuta la CLI de Vercel
    - despliega automáticamente la nueva versión.

Esto garantiza que el deploy siempre refleje el estado real del repositorio y evita divergencias.

6. Test E2E con Cypress (Frontend)

Para el test E2E implementamos un flujo completo en Cypress ya que Clerk posee soporte para el mismo. Dicha prueba valida la interacción real de un vendedor con la plataforma ejecutada de manera local

Flujo real del test E2E (Cypress)

1. Logueo del vendedor mediante Clerk.
2. Creación de un producto desde la UI.
3. Agregar ese producto al carrito desde el catálogo.
4. Realizar la compra con el usuario correspondiente.
5. Validar que el stock se reduce correctamente después del pedido.
6. Cancelar el pedido desde la vista del vendedor.
7. Verificar que el stock se repone luego de la cancelación.
8. Eliminar el producto para finalizar el ciclo.

Este test asegura la consistencia del flujo completo: creación → venta → actualización de stock → cancelación → reposición → limpieza final.