Informe de Funcionalidades y Arquitectura

- David Artunduaga
- Jennifer Castro
- Rony Ordoñez
- Juan de la Pava

Resumen

Aplicación web construida con Next.js (App Router) y TypeScript que implementa catálogo y gestión de productos, órdenes y perfil de usuario. La autenticación y parte del flujo de seguridad se integran vía GraphQL; el consumo de datos de negocio (productos/órdenes/usuarios) se realiza vía API REST. El estado de la UI se gestiona con Zustand, persistiendo la sesión de forma segura en el cliente.

Funcionalidades Implementadas

- Catálogo de productos:
  - Listado con búsqueda y paginado (`/products`).
  - Detalle de producto, control de stock y acción “Agregar al carrito”.
  - CRUD de productos (crear, editar, eliminar) para el propietario o administrador.
- Carrito y órdenes:
  - Carrito de compras y flujo de creación de órdenes.
  - Listado de órdenes y detalle con estados (pendiente, completada, cancelada).
- Perfil y seguridad:
  - Vista de perfil con datos básicos y estado de 2FA.
  - Pantalla y componente de configuración de 2FA (en progreso).
- Administración (área admin):
  - Listado de usuarios, productos y órdenes con acciones administrativas.
- Infraestructura de UI:
  - Componentes reutilizables (Button, Input, Card, Modal, Pagination, ToastProvider).
  - Layouts seccionados para dashboard, auth y admin.

Autenticación

- Store de sesión: `src/store/authStore.ts` con Zustand y persistencia (localStorage) de `user` y `token` .
- Hook de acceso: `src/lib/hooks/useAuth.ts` expone `user` , `token` , `isAuthenticated` , `isAdmin` , `login` , `logout` y `setUser` .
- Inicio de sesión/registro: `src/lib/api/auth.ts` usa Apollo Client contra el backend GraphQL (`LOGIN_MUTATION` , `SIGNUP_MUTATION`).
- Perfil: `getProfile()` via REST (`/auth/profile`) con `apiClient` .
- Token: se guarda en `localStorage` bajo `TOKEN_KEY` para adjuntarlo en solicitudes HTTP.

Flujo típico de login:

1. Vista de login envía credenciales a GraphQL.
2. Respuesta incluye datos de usuario y token.
3. `useAuthStore.login()` persiste token y actualiza `user` , `isAuthenticated` e `isAdmin` .
4. El `apiClient` agrega el token en `Authorization` en cada request REST.

2FA (en progreso):

- `TwoFactorSetup.tsx` es el componente UI para habilitar/verificar/deshabilitar 2FA.
- Endpoints GraphQL ya definidos en `auth.ts` (`enable2FA` , `verify2FA` , `disable2FA`).
- Faltan wiring y actualización del estado `user.twoFactorEnabled` tras cada operación.

Autorización (RBAC y ownership)

- RBAC básico: `isAdmin` se deriva de `user.role === 'admin'` .
- Rutas protegidas: `components/auth/ProtectedRoute.tsx` verifica `isAuthenticated` y, opcionalmente, `requireAdmin` para impedir acceso y redirigir.
- Ownership en UI: en componentes como `ProductCard` se calcula `isOwner` (comparando `user.id` con `sellerId`) y se condicionan acciones (Editar/Eliminar) a `isOwner || isAdmin` .
- Backend: se asume validación definitiva de permisos en la API/GraphQL; el frontend refuerza reglas para UX y evitar acciones visibles a quien no corresponde.

Gestión de Estado

- Librería: Zustand, stores por dominio.

Stores clave:

- `authStore` :
  - Estado: `user` , `token` , `isAuthenticated` , `isAdmin` , `_hasHydrated` .
  - Acciones: `login` , `logout` (limpia token e intenta notificar backend), `setUser` y `setHasHydrated` .
  - Persistencia: se guardan solo `user` y `token` (derive de ellos `isAuthenticated` / `isAdmin` en `onRehydrateStorage`).
- `productStore` :
  - Estado: `products` , `selectedProduct` , `loading` , `error` , filtros y paginación.
  - Acciones asíncronas: `fetchProducts` , `fetchProductById` , `createProduct` , `updateProduct` , `deleteProduct` que consumen `src/lib/api` (REST).
  - Patrón: las funciones no retornan datos; actualizan el estado centralizado y los componentes leen desde el store.
- `orderStore` (estructura análoga):
  - Maneja carrito, creación de órdenes y listado/estado de órdenes.

Ventajas del enfoque:

- Simplicidad: stores delgados, acciones explícitas, sin boilerplate.
- Persistencia controlada: solo datos necesarios de sesión.
- Hidratación segura: `_hasHydrated` evita parpadeos de UI antes de rehidratar.

Integración con APIs

- GraphQL (Apollo Client): autenticación y 2FA.
- REST (Axios `apiClient`): productos, órdenes y perfil.
- `apiClient` añade el token desde `localStorage` y maneja errores comunes (401 redirige a login, 403/404/5xx loguea y notifica).

Consideraciones de SSR/SSG y variables de entorno

- App Router con componentes client-side (`'use client'`) para vistas protegidas.
- Durante el build, Next.js puede evaluar módulos importados. Se evitó inicializar SDKs sensibles en el tope del módulo para no depender de variables de entorno en tiempo de build.
- Cliente de Supabase: se recomienda patrón de “factory perezosa” para evitar efectos al importar, y asegurar que `NEXT_PUBLIC_*` existan durante el build cuando se requiera.

Decisiones Técnicas Destacadas

- Next.js 16 + TypeScript estricto: el build verifica tipos; se añadieron utilidades para `type-check` en desarrollo.
- Zustand con `persist` para auth: minimiza re-render y facilita control de sesión.
- Separación de dominios: hooks (`useAuth` , `useProducts` , `useOrders`) encapsulan acceso al estado y a las APIs.
- UI desacoplada: componentes atómicos (Button, Input, Modal, Card) y compuestos (ProductCard, OrdersTable, UserTable).