1. Introducción

En un entorno comercial cada vez más dominado por grandes cadenas y plataformas de venta en línea, los pequeños comercios de barrio enfrentan dificultades para competir y atraer clientes. La falta de visibilidad, la incapacidad para comunicar sus ofertas de manera eficiente y la ausencia de una plataforma accesible para mostrar sus productos representan obstáculos significativos que afectan su sostenibilidad y crecimiento.

Mercado Conecta surge como una solución tecnológica a esta problemática. Se trata de una plataforma digital que facilita la conexión entre los negocios locales y los consumidores de su comunidad. A través de esta herramienta, los usuarios pueden descubrir comercios cercanos, consultar productos con precios actualizados, recibir notificaciones de ofertas y promociones, y fomentar el consumo en el comercio de proximidad.

El objetivo principal del proyecto es proporcionar una herramienta accesible y eficiente que potencie la visibilidad de los pequeños comercios, optimizando su comunicación con los clientes y promoviendo un ecosistema comercial más equitativo. Se espera que esta solución no solo ayude a mejorar la competitividad de los negocios locales, sino que también fortalezca el sentido de comunidad y contribuya al desarrollo económico de los barrios.

2. Arquitectura del Proyecto

2.1 Estructura de Carpetas

El proyecto está organizado siguiendo una estructura modular que facilita la escalabilidad y el mantenimiento del código. A continuación, se describe la estructura principal:

```
src/
├── components/        # Componentes reutilizables (Header, LoginModal, etc.)
├── pages/             # Vistas principales del sistema (Home, Dashboard, etc.)
├── lib/               # Lógica de negocio (User.js)
├── providers/         # Contextos globales (authContext.jsx)
├── App.jsx            # Componente raíz con configuración de rutas
└── main.jsx           # Punto de entrada de la aplicación
```

Esta organización permite separar claramente la lógica de presentación, negocio y contexto, siguiendo buenas prácticas de desarrollo en React.

2.2 Tecnologías Utilizadas

El sistema está construido con las siguientes tecnologías:

- **React**: Biblioteca principal para construir la interfaz de usuario.

- **React Router DOM**: Manejo de rutas y navegación entre páginas.
- **Supabase**: Backend como servicio para autenticación y base de datos.
- **Tailwind CSS**: Framework de estilos utilitario para diseño responsivo.
- **React Hot Toast**: Sistema de notificaciones emergentes.
- **Lucide React**: Librería de iconos SVG modernos.

2.3 Flujo General de Componentes

El flujo de navegación y renderizado de componentes sigue esta jerarquía:

```
App.jsx
├── AuthProvider (contexto de autenticación)
└── Router
    ├── Home
    │   ├── Header
    │   ├── Hero
    │   ├── BusinessGrid
    │   ├── LoginModal
    │   └── RegisterModal
    ├── Dashboard
    ├── Maps
    ├── FavoriteCommercers
    └── Otros...
```

Cada página se encarga de una funcionalidad específica, y los componentes hijos permiten una experiencia modular y reutilizable.

3. componentes

Esto se realizará en cascada, empezando por la carpeta **components/** y terminando en el **App**. Estas carpetas se encuentran dentro de la carpeta llamada **src.**

Componente: BusinessGrid.jsx

Descripción General

BusinessGrid es un componente visual que muestra una cuadrícula de comercios destacados. Permite a los usuarios buscar tiendas por nombre, filtrar por tipo y navegar entre páginas de resultados. Es una pieza clave para la exploración de comercios dentro de la plataforma.

Funcionalidades Principales

- **Búsqueda por nombre**: Campo de texto con ícono de búsqueda.
- **Filtro por tipo de tienda**: Selector desplegable (TypeFilterSelect).
- **Paginación**: Navegación entre páginas de resultados (Paginator).
- **Visualización de comercios**: Cada comercio se muestra con CommerceInfoBox.
- **Modal de login**: Se activa al hacer clic en un comercio si el usuario no está autenticado.

Lógica del Componente

- **Estados locales**:
    - searchTerm: término de búsqueda.
    - typeFilter: filtro por tipo de tienda.
    - commerces: lista de comercios obtenidos.
    - currentPage: página actual.
    - totalCount: total de comercios disponibles.
    - loginOpen: controla la visibilidad del modal de login.
- **Carga de datos**:
    - Se ejecuta un useEffect cada vez que cambian searchTerm, typeFilter o currentPage.
    - Llama a Commerce.getTopRated() para obtener los comercios desde el backend.

Componentes Relacionados

- CommerceInfoBox: Muestra la información de cada comercio.
- TypeFilterSelect: Selector de tipo de tienda.
- Paginator: Control de navegación entre páginas.
- LoginModal: Modal para iniciar sesión.

Componente: CommerceDeleteModal.jsx

Descripción

CommerceDeleteModal es un componente modal de confirmación que permite al usuario eliminar un comercio registrado. Se utiliza para evitar eliminaciones accidentales, solicitando una confirmación explícita del usuario antes de proceder.

Props Recibidas

- **commerce**: Objeto que contiene la información del comercio a eliminar (se usa para mostrar el nombre).
- **isOpen**: Booleano que determina si el modal está visible.
- **onClose**: Función que se ejecuta al cerrar el modal sin eliminar.
- **onSave**: Función que se ejecuta cuando el usuario confirma la eliminación.

Lógica del Componente

- Si isOpen es false, el componente no se renderiza (return null).
- Muestra un mensaje de advertencia con el nombre del comercio.
- Ofrece dos opciones:
    - **Confirmar eliminación**: Ejecuta onSave.
    - **Cancelar**: Ejecuta onClose.

Estilo y UX

- Modal centrado con fondo oscuro (bg-black/50).
- Diseño limpio y moderno con clases de Tailwind CSS.
- Botón de acción destacado y botón de cancelación más discreto.

Componente: CommercesInfoBox.jsx

Descripción General

CommercesInfoBox es un componente visual que representa una tarjeta individual de comercio. Se utiliza para mostrar información resumida de una tienda, incluyendo su imagen, nombre, ubicación, calificación promedio y número de comentarios. Es un componente reutilizable dentro de listados como el de BusinessGrid.

Props Recibidas

- **image**: URL de la imagen del comercio.
- **commerceName**: Nombre del comercio.
- **location**: Dirección o ubicación del comercio.
- **calification**: Calificación promedio (de 0 a 5).
- **comment**: Texto con el número de comentarios.
- **onClick**: Función que se ejecuta al hacer clic en la tarjeta (por ejemplo, abrir un modal o redirigir).

Lógica del Componente

- Se genera un arreglo de 5 estrellas.
- Cada estrella se pinta de amarillo si la calificación es mayor o igual a su índice.
- Se muestra la calificación numérica con un decimal (toFixed(1)).

Componente: Container.jsx

Descripción

Container es un componente de presentación que actúa como un contenedor visual para otras secciones o componentes. Proporciona un diseño consistente con márgenes, padding y un fondo gris claro, ideal para estructurar páginas o secciones completas.

Props Recibidas

- **title**: Título principal que se muestra en la parte superior del contenedor.
- **children**: Contenido interno que se renderiza dentro del contenedor.

Componente: ContainerMaps.jsx

Descripción

ContainerMaps es un componente que renderiza un mapa interactivo utilizando **Leaflet** (a través de react-leaflet). Muestra la ubicación del usuario y la de los comercios registrados, geocodificando sus direcciones si no tienen coordenadas. Es clave para la funcionalidad de exploración geográfica de la plataforma.

Funcionalidades Principales

- **Ubicación del usuario**: Se obtiene mediante la API de geolocalización del navegador.

- **Carga de comercios**: Se obtienen desde Commerce.getAll() y se geocodifican si no tienen lat/lng.
- **Marcadores personalizados**: Cada comercio se representa con un ícono SVG personalizado.
- **Centrado automático del mapa**: El mapa se centra en la ubicación del usuario al cargar.

Lógica del Componente

- **Geocodificación**:
    - Usa la API de Nominatim (OpenStreetMap) para convertir direcciones en coordenadas.
    - Si un comercio ya tiene lat y lng, se omite la geocodificación.
- **Marcadores**:
    - Se renderiza un marcador para el usuario y uno para cada comercio.
    - Cada marcador muestra un Popup con el nombre y dirección del comercio.
- **Centrado del mapa**:
    - El componente CenterMap usa useMap() para mover la vista al centro deseado.

Dependencias

- react-leaflet: Para renderizar el mapa.
- leaflet: Librería base de mapas.
- Commerce (de lib/Commerce.js): Para obtener los datos de los comercios.

Componente: DynamicSidebar.jsx

Descripción

DynamicSidebar es un componente de barra lateral adaptable que muestra información del comercio, botones de acción personalizados y una sección de configuración. También permite a los propietarios calificar su propio comercio mediante un sistema de estrellas.

Props Recibidas

- **userId**: ID del usuario actual.
- **commerceId**: ID del comercio asociado.

- **rating**: Calificación inicial del comercio (por defecto 0).
- **isOwner**: Booleano que indica si el usuario es el dueño del comercio.
- **showEmpresa**: Muestra la sección de información del comercio.
- **commpress**: Nombre o descripción del comercio.
- **listButtons**: Arreglo de botones dinámicos con etiquetas.
- **onButtonClick**: Función que se ejecuta al hacer clic en un botón.
- **showConfig**: Muestra la sección de configuración.
- **onConfig**: Función que se ejecuta al hacer clic en "Configuración".

Lógica del Componente

- **Calificación por estrellas**:
    - Solo visible si isOwner es true.
    - Al hacer clic en una estrella, se actualiza el estado local y se guarda la calificación usando Calification.upsert().
- **Botones dinámicos**:
    - Se generan a partir del arreglo listButtons.
    - Cada botón ejecuta onButtonClick con su etiqueta como argumento.
- **Configuración**:
    - Se muestra al final de la barra lateral si showConfig es true.

Estilo y UX

- Fondo azul con texto blanco para secciones destacadas.
- Botones blancos con hover azul claro.
- Estrellas interactivas para calificación.
- Diseño vertical con separación clara entre secciones.

Componente: FavoriteRemoveModal.jsx

Descripción

FavoriteRemoveModal es un componente modal de confirmación que permite al usuario remover un comercio de su lista de favoritos. Su propósito es evitar eliminaciones accidentales, solicitando una confirmación explícita antes de proceder.

Props Recibidas

- **commerce**: Objeto que contiene la información del comercio (se usa para mostrar su nombre).
- **isOpen**: Booleano que determina si el modal está visible.
- **onClose**: Función que se ejecuta al cerrar el modal sin eliminar.
- **onSave**: Función que se ejecuta cuando el usuario confirma la eliminación del favorito.

Lógica del Componente

- Si isOpen es false, el componente no se renderiza (return null).
- Muestra un mensaje de advertencia con el nombre del comercio.
- Ofrece dos opciones:
    - **Confirmar eliminación**: Ejecuta onSave.
    - **Cancelar**: Ejecuta onClose.

Estilo y UX

- Modal centrado con fondo oscuro (bg-black/50).
- Diseño limpio y moderno con clases de Tailwind CSS.
- Botón de acción destacado y botón de cancelación más discreto.

Componente: FloatingScoreCard.jsx

Descripción

FloatingScoreCard es un componente visual que muestra una tarjeta flotante con la puntuación promedio de un comercio. Se utiliza para destacar la calificación general de forma visible y atractiva, generalmente en una vista de perfil o detalle de comercio.

Props Recibidas

- **score**: Valor numérico de la calificación promedio. Por defecto es 5.0.

Lógica del Componente

- Si no se proporciona una calificación, se muestra 0.0 como valor por defecto.
- El valor se formatea a un decimal con toFixed(1).

Estructura del Renderizado

- **sticky top-20**: Hace que la tarjeta se mantenga visible al hacer scroll.
- **SVG de estrella**: Representa visualmente la calificación.
- **Texto destacado**: Muestra el valor numérico de la puntuación.

Componente: Header.jsx

Descripción

Header es el componente de navegación principal de la aplicación. Se muestra en la parte superior de todas las páginas y proporciona acceso al inicio de sesión, así como un menú adaptable para dispositivos móviles. También incluye el logotipo de la plataforma.

Funcionalidades Principales

- **Logotipo**: Representado por un ícono (Landmark) y el nombre "Mercado Conecta".
- **Inicio de sesión**: Botón visible solo si el usuario no está autenticado.
- **Menú móvil**: Se despliega en pantallas pequeñas mediante un botón tipo hamburguesa.
- **Modal de login**: Se activa al hacer clic en el botón "Iniciar sesión".

Lógica del Componente

- **useState**:
    - mobileOpen: Controla la visibilidad del menú móvil.
    - loginOpen: Controla la visibilidad del modal de inicio de sesión.
- **useAuth**:

- Se utiliza para verificar si hay un usuario autenticado (user).

🎨 Estilo y UX

- Diseño responsivo con Tailwind CSS.
- Modal accesible y centrado.
- Botones con transiciones suaves y colores institucionales.

Componente: Hero.jsx

Descripción General

Hero es el componente de bienvenida que se muestra en la parte superior de la página principal. Está diseñado para captar la atención del usuario con un fondo visual atractivo, un mensaje claro y un llamado a la acción para registrarse. También incluye un texto animado que refuerza los valores del proyecto.

Funcionalidades Principales

- **Fondo visual**: Imagen de fondo con superposición oscura para mejorar la legibilidad.
- **Título destacado**: Presenta el nombre del proyecto con énfasis visual.
- **Texto rotativo**: Palabras clave animadas que refuerzan los beneficios del sistema.
- **Botón de registro**: Abre un modal para que el usuario se registre.
- **Modal de registro**: Componente RegisterModal que se muestra al hacer clic en el botón.

Lógica del Componente

- **useState**:
    - regOpen: Controla la visibilidad del modal de registro.
- **RotatingText**:
    - Muestra palabras como “Optimización”, “Facilidad”, “Crecimiento” y “Éxito” con animaciones de entrada y salida.

- Se actualiza automáticamente cada 2 segundos (rotationInterval).

Componente: ImageUploadField.jsx

Descripción

ImageUploadField es un componente de formulario que permite a los usuarios subir imágenes a través de la API de Cloudinary. Se utiliza principalmente para adjuntar fotos de productos o comercios, y proporciona una vista previa de la imagen cargada.

Props Recibidas

- **onUpload**: Función que se ejecuta cuando la imagen se ha subido exitosamente. Recibe la URL de la imagen.
- **label** *(opcional)*: Texto que se muestra como etiqueta del campo. Por defecto: "Adjuntar foto del producto".
- **initialUrl** *(opcional)*: URL de una imagen previamente cargada (útil para edición).
- **commerce** *(opcional)*: Booleano que modifica el comportamiento visual si se trata de una imagen de comercio.

Lógica del Componente

- **Carga inicial**: Si se proporciona initialUrl, se muestra como imagen cargada.
- **Subida de imagen**:
    - Se crea un FormData con el archivo y un upload_preset.
    - Se envía a Cloudinary mediante fetch.
    - Si la respuesta contiene secure_url, se actualiza el estado y se llama a onUpload.
- **Manejo de errores**: Si ocurre un error durante la subida, se muestra un mensaje de error.

🎨 Estilo y UX

- Estilo moderno con Tailwind CSS.
- Ícono de imagen (ImagePlus) para mejorar la accesibilidad visual.
- Indicador de carga (Subiendo...) y confirmación visual (Imagen cargada).
- Vista previa de la imagen cargada.

Componente: LoginModal.jsx

Descripción

LoginModal es un componente modal que permite a los usuarios iniciar sesión en la plataforma. Se activa desde la interfaz principal y gestiona tanto la autenticación como la verificación del usuario en la base de datos de Supabase.

Props Recibidas

- **isOpen**: Booleano que determina si el modal está visible.
- **onClose**: Función que se ejecuta para cerrar el modal.

Lógica del Componente

- **Estados locales**:
    - email: Correo electrónico ingresado por el usuario.
    - password: Contraseña ingresada.
    - errorMessage: Mensaje de error mostrado si ocurre un fallo.
- **Autenticación**:
    - Se llama a login() desde el contexto de autenticación (useAuth).
    - Si es exitosa, se consulta la tabla usuarios en Supabase para verificar la existencia del usuario.
    - Si todo es correcto, se cierra el modal y se redirige al dashboard.
- **Errores**:
    - Si falla la autenticación o la verificación, se muestra un mensaje de error en pantalla.

Componente: NavBar.jsx

Descripción

El componente NavBar representa la barra de navegación principal de la aplicación. Proporciona acceso a secciones clave como el perfil del usuario, el

dashboard, el mapa, tiendas favoritas, un buscador, notificaciones y la opción de cerrar sesión.

Dependencias

- **React**: Para la creación del componente y manejo del estado.
- **Lucide-react**: Íconos visuales (LogOut, Bell, Search, User).
- **React Router DOM**: Navegación programática (useNavigate).
- **Contexto de Autenticación**: useAuth para obtener el usuario actual y cerrar sesión.
- **Modelo de Usuario**: UserModel.getUserNameByEmail para obtener el nombre del usuario a partir del correo electrónico.

Funcionalidades Principales

1. Visualización del Nombre del Usuario

- Se obtiene el nombre del usuario autenticado mediante su correo electrónico.
- Se muestra junto a un ícono de usuario en la parte izquierda de la barra.

2. Navegación

- Enlaces a:
    - /dashboard (Inicio)
    - /maps (Mapa)
    - /favoriteCommercers (Tiendas favoritas)
    - /profile (Perfil del usuario)

3. Buscador

- Campo de entrada para búsqueda.
- Al hacer clic en el botón de búsqueda, redirige a /search-resut?name=valor.

4. Notificaciones y Cierre de Sesión

- Ícono de campana para notificaciones (sin funcionalidad implementada).
- Botón de cierre de sesión que:
    - Llama a logout().
    - Redirige al usuario a la página de inicio (/).

Hooks Utilizados

- useState: Para manejar el nombre del usuario y el valor del buscador.
- useEffect: Para cargar el nombre del usuario al montar el componente.
- useNavigate: Para redireccionar al usuario tras acciones como búsqueda o logout.

Estilos

- Utiliza clases de Tailwind CSS para diseño responsivo, espaciado, colores y efectos hover.
- Diseño adaptado para pantallas medianas en adelante (md:flex).

Componente: Paginator.jsx

Descripción General

El componente Paginator es un componente de navegación que permite al usuario moverse entre páginas de contenido. Muestra el número de página actual, el total de páginas disponibles y botones para avanzar o retroceder.

Props Recibidas

Propiedad	Tipo	Descripción
currentPage	number	Número de la página actual.
totalPages	number	Total de páginas disponibles.
onPrevious	func	Función que se ejecuta al hacer clic en el botón **Anterior**.
onNext	func	Función que se ejecuta al hacer

Propiedad	Tipo	Descripción
		clic en el botón **Siguiente**.

Lógica del Componente

- **Botón “Anterior”**:
  - Ejecuta onPrevious al hacer clic.
  - Se desactiva si currentPage === 1.
- **Botón “Siguiente”**:
  - Ejecuta onNext al hacer clic.
  - Se desactiva si currentPage === totalPages.
- **Texto Central**:
  - Muestra el estado actual de la paginación: "Página X de Y".

Estilos

- Utiliza **Tailwind CSS** para diseño y estilos:
  - Contenedor con espaciado, fondo gris claro y bordes redondeados.
  - Botones con fondo azul, texto blanco y opacidad reducida cuando están deshabilitados.
  - Texto central con fuente destacada en azul oscuro.

Componente: PersonalData.jsx

Descripción

El componente PersonalData muestra información básica del usuario, como su nombre y correo electrónico, y proporciona un botón para editar el perfil. Es un componente visual simple y reutilizable, ideal para secciones de perfil de usuario.

Props Recibidas

Propiedad	Tipo	Descripción
name	string	Nombre del usuario.
email	string	Correo electrónico del usuario.
onClick	func	Función que se ejecuta al hacer clic en el botón **Editar perfil**.

Lógica del Componente

- Muestra un encabezado con el título **"Información general"**.
- Incluye un botón **"Editar perfil"** que ejecuta la función onClick al ser presionado.
- Presenta dos secciones:
    - **Nombre**: Muestra el valor de name o un texto por defecto si no se proporciona.
    - **Correo electrónico**: Muestra el valor de email o un texto por defecto si no se proporciona.

Estilos

- Utiliza **Tailwind CSS** para el diseño:
    - Contenedor con fondo blanco, padding, bordes redondeados y sombra.
    - Tipografía jerárquica para títulos y subtítulos.
    - Botón con colores azul y blanco, con efecto hover.

Componente: ProductDeleteModal.jsx

Descripción

ProductDeleteModal es un componente modal que permite al usuario confirmar la eliminación de un producto del catálogo. Muestra el nombre del producto y ofrece opciones para confirmar o cancelar la acción.

Props Recibidas

Propiedad	Tipo	Descripción
productId	string	ID del producto que se desea eliminar.
isOpen	boolean	Controla la visibilidad del modal.
onClose	func	Función que se ejecuta al cerrar el modal.
onSave	func	Función que se ejecuta después de eliminar el producto exitosamente.

Lógica del Componente

- **Carga del producto**:
    - Al abrir el modal (isOpen === true), se obtiene el nombre del producto mediante Product.getById(productId) y se guarda en el estado local.
- **Eliminación del producto**:
    - Al hacer clic en **Guardar**, se ejecuta Product.delete(productId).
    - Si la operación es exitosa, se llama a onSave().
    - Si ocurre un error, se muestra un mensaje de error.
- **Cierre del modal**:
    - Puede cerrarse haciendo clic en el botón ✕ o en el texto **"No, rechazar"**.

Estilos

- Modal centrado con fondo semitransparente (bg-black/50).

- Contenedor blanco con bordes redondeados y sombra.
- Botón principal estilizado con colores azules y efecto hover.
- Tipografía clara y jerárquica.
- Mensajes de error en rojo.

Componente: ProductModal.jsx

Descripción

ProductModal es un componente modal reutilizable que permite **crear o editar productos** dentro de un comercio. Incluye campos para nombre, descripción, precio, tipo y una imagen del producto. Se conecta con la API para obtener, actualizar o crear productos.

Props Recibidas

Propiedad	Tipo	Descripción
commerceId	string	ID del comercio al que pertenece el producto.
productId	string	ID del producto a editar (vacío si se está creando uno nuevo).
isOpen	boolean	Controla la visibilidad del modal.
onClose	func	Función que se ejecuta al cerrar el modal.
onSave	func	Función que se ejecuta después

Propiedad	Tipo	Descripción
		de guardar exitosamente.

Lógica del Componente

1. Carga de datos

- Si productId está definido y el modal se abre, se obtiene la información del producto desde la API (Product.getById) y se precargan los campos.

2. Guardado de datos

- Valida que los campos obligatorios (name, price, type) estén completos.
- Si productId existe, actualiza el producto con Product.update.
- Si no existe, crea uno nuevo con Product.create.
- Llama a onSave() y limpia los campos.

3. Limpieza de campos

- La función cleanValue() reinicia todos los campos del formulario.

Campos del Formulario

Campo	Tipo de entrada	Descripción
Nombre	input	Nombre del producto.
Descripción	textarea	Descripción del producto.
Precio	input	Precio del producto.

Campo	Tipo de entrada	Descripción
Tipo	TypeFilterSelect	Selector de categoría del producto.
Imagen	ImageUploadField	Componente para subir imagen del producto.

Componente: ProfileModifyModal.jsx

Descripción

ProfileModifyModal es un componente modal que permite al usuario **editar su información personal**, específicamente su nombre y correo electrónico. Es un formulario simple con validación básica y control de errores.

Props Recibidas

Propiedad	Tipo	Descripción
profile	object	Objeto con los datos actuales del perfil del usuario (name, email).
isOpen	boolean	Controla la visibilidad del modal.

Propiedad	Tipo	Descripción
onClose	func	Función que se ejecuta al cerrar el modal.
onSave	func	Función que se ejecuta al guardar los cambios, recibe { name, email }.

Lógica del Componente

- **Carga de datos**:
  - Al abrir el modal (isOpen === true), se precargan los valores de name y email desde el objeto profile.
- **Validación y guardado**:
  - La función validateSave() verifica que ambos campos estén completos.
  - Si la validación es exitosa, se llama a onSave() con los nuevos datos.
- **Cierre del modal**:
  - Se puede cerrar haciendo clic en el botón ✕ o desde el componente padre mediante onClose.

Campos del Formulario

Campo	Tipo de entrada	Descripción
Nombre	input	Campo editable para el nombre.
Email	(no visible)	Aunque se carga, no se muestra.

Componente: PublicationInfoBox.jsx

Descripción

PublicationInfoBox es un componente que muestra información de una publicación, incluyendo el nombre del autor, la descripción y botones de reacción (me gusta / no me gusta). Permite a los usuarios interactuar con la publicación y registrar sus reacciones.

Props Recibidas

Propiedad	Tipo	Descripción
userId	string	ID del usuario que está reaccionando.
publicationId	string	ID de la publicación mostrada.
name	string	Nombre del autor o comercio.
description	string	Descripción del contenido de la publicación.
like	number	Número inicial de reacciones positivas.
dislike	number	Número inicial de reacciones negativas.
initialReaction	string	Reacción inicial del usuario

Propiedad	Tipo	Descripción
		('like', 'dislike' o null).
onCommerce	func	Función que se ejecuta al hacer clic en el nombre del autor/comercio.

Lógica del Componente

- **Estado local**:
  - likes y dislikes: Contadores de reacciones.
  - userReaction: Reacción actual del usuario.
- **Reacción del usuario**:
  - Si el usuario hace clic en una reacción que ya había seleccionado, se elimina.
  - Si cambia de reacción, se actualizan ambos contadores.
  - Se utiliza PublicationReactions.setReaction() para registrar la reacción en el backend.

Estilos

- Contenedor blanco con sombra y bordes redondeados.
- Nombre del autor con ícono de usuario, estilo interactivo (hover:text-blue-800).
- Descripción en texto base.
- Botones de reacción con íconos (ThumbsUp, ThumbsDown) y estilos dinámicos:
  - Azul para "me gusta".
  - Rojo para "no me gusta".
  - Negrita y color activo si el usuario ya reaccionó.

Componente: PublicationModal.jsx

Descripción

PublicationModal es un componente modal que permite a los usuarios **crear una nueva publicación** asociada a un comercio. Incluye un campo de descripción y maneja la validación básica antes de enviar los datos a la API.

Props Recibidas

Propiedad	Tipo	Descripción
commereceId	string	ID del comercio al que se asociará la publicación.
userId	string	ID del usuario que crea la publicación.
isOpen	boolean	Controla la visibilidad del modal.
onClose	func	Función que se ejecuta al cerrar el modal.
onSave	func	Función que se ejecuta después de guardar exitosamente.

Lógica del Componente

- **Estado local**:
    - description: Contenido de la publicación.

- errorMessage: Mensaje de error si el campo está vacío.

- **Guardado de publicación**:
    - Si el campo de descripción está lleno, se llama a Publications.create() con los datos necesarios.
    - Si la operación es exitosa, se ejecuta onSave().
    - Si el campo está vacío, se muestra un mensaje de error.
- **Cierre del modal**:
    - Se puede cerrar haciendo clic en el botón ✕ o desde el componente padre mediante onClose.

Campos del Formulario

Campo	Tipo de entrada	Descripción
Descripción	textarea	Campo obligatorio para el contenido textual.

Estilos

- Modal centrado con fondo oscuro semitransparente.
- Contenedor blanco con bordes redondeados y sombra.
- Campo de texto con bordes suaves y colores azules.
- Botón principal estilizado con animación hover.

Componente: RegisterModal.jsx

Descripción

RegisterModal es un componente modal que permite a los usuarios **crear una nueva cuenta**. Incluye un formulario con campos para nombre, correo electrónico y contraseña, y se conecta con el contexto de autenticación para registrar al usuario.

Props Recibidas

Propiedad	Tipo	Descripción
isOpen	boolean	Controla la visibilidad del modal.
onClose	func	Función que se ejecuta al cerrar el modal.

Lógica del Componente

- **Estado local**:
    - formData: Objeto que contiene los valores de los campos del formulario (nombre, email, password).
- **Manejo de cambios**:
    - handleChange: Actualiza el estado formData al escribir en los campos del formulario.
- **Registro de usuario**:
    - handleRegister: Llama a la función register del contexto de autenticación con los datos del formulario.
    - Si el registro es exitoso:
        - Se cierra el modal.
        - Se redirige al usuario al dashboard (/dashboard).
    - Si hay error, se muestra en consola (aunque el contexto ya maneja notificaciones).

Campos del Formulario

Campo	Tipo de entrada	Descripción
Nombres	input	Nombre del usuario.
Correo	input	Correo electrónico del usuario.
Contraseña	input	Contraseña para la cuenta.

Componente: StoreList.jsx

Descripción

StoreList es un componente que muestra una lista de tiendas. Cada tienda puede ser seleccionada y, si está marcada como favorita, se muestra con un ícono de corazón. Es ideal para vistas donde se desea listar comercios con interacción rápida.

Props Recibidas

Propiedad	Tipo	Descripción
stores	array	Lista de objetos tienda. Cada objeto debe tener al menos una propiedad name y opcionalmente isFavorite.
onStoreClick	func	Función que se ejecuta al hacer

Propiedad	Tipo	Descripción
		clic sobre una tienda.
onToggleFavorite	func	Función que se ejecuta al hacer clic en el ícono de favorito.

Lógica del Componente

- Si la lista stores está vacía, se muestra un mensaje: **"No hay tiendas relacionadas"**.
- Si hay tiendas:
    - Se renderiza cada tienda con su nombre y un ícono de carrito.
    - Si la tienda está marcada como favorita (isFavorite === true), se muestra un ícono de corazón azul relleno.
    - Al hacer clic en el nombre de la tienda, se ejecuta onStoreClick(store).
    - Al hacer clic en el ícono de favorito, se ejecuta onToggleFavorite(store).

Componente: TypeFilterSelect.jsx

Descripción

TypeFilterSelect es un componente de selección desplegable que permite filtrar elementos por tipo. Los tipos se obtienen dinámicamente desde una fuente externa según el ámbito (scope) especificado.

Props Recibidas

Propiedad	Tipo	Descripción
scope	string	Define el contexto de los tipos a cargar ('commerce' por

Propiedad	Tipo	Descripción
		defecto o 'product').
value	number	ID del tipo actualmente seleccionado.
onChange	func	Función que se ejecuta al seleccionar un nuevo tipo.

Lógica del Componente

- **Carga de tipos**:
  - Al montar el componente o cambiar el scope, se llama a Types.getAll(scope) para obtener los tipos disponibles.
  - Los tipos se almacenan en el estado local types.
- **Selección de tipo**:
  - Se renderiza un <select> con una opción por cada tipo.
  - Al cambiar la selección, se ejecuta onChange con el ID del tipo seleccionado.

Componente: Calification.js

Descripción

El módulo Calification gestiona la lógica relacionada con las **calificaciones de comercios** por parte de los usuarios. Permite insertar, actualizar o eliminar una calificación en la base de datos, utilizando Supabase como backend.

Clase: Calification

Método Estático: upsert({ userId, commerceId, score })

Este método realiza una operación de **upsert** (insertar o actualizar) sobre la tabla califications.

Parámetros:

Parámetro	Tipo	Descripción
userId	string	ID del usuario que realiza la calificación.
commerceId	string	ID del comercio que está siendo calificado.
score	number	Valor de la calificación (0 para eliminar).

Lógica del Método

1. **Eliminar calificación**:
    - Si score === 0, se elimina la calificación existente para ese usuario y comercio.
2. **Actualizar calificación existente**:
    - Si ya existe una calificación, se actualiza con el nuevo score.
3. **Insertar nueva calificación**:
    - Si no existe una calificación previa, se crea una nueva entrada en la tabla.

Manejo de Errores

- Se lanza un Error con el mensaje correspondiente si ocurre un fallo en cualquiera de las operaciones (delete, select, update, insert).
- Se ignora el error PGRST116 (registro no encontrado) al verificar si ya existe una calificación.

Componente Commerce.js

Descripción

El módulo Commerce encapsula todas las operaciones relacionadas con la gestión de comercios en la base de datos. Utiliza Supabase como backend para realizar operaciones CRUD, obtener comercios destacados, y consultar información personalizada para usuarios autenticados.

Clase: Commerce

Métodos Estáticos

create({ name, location, owner_id, type_id })

- **Descripción**: Crea un nuevo comercio con los datos proporcionados.
- **Retorna**: El comercio recién creado.
- **Errores**: Lanza error si la inserción falla.

update(id, updates)

- **Descripción**: Actualiza un comercio existente con los datos proporcionados.
- **Parámetros**:
  - id: ID del comercio a actualizar.
  - updates: Objeto con los campos a modificar.
- **Retorna**: El comercio actualizado.
- **Errores**: Lanza error si la actualización falla.

delete(id)

- **Descripción**: Elimina un comercio por su ID.
- **Retorna**: true si la operación fue exitosa.
- **Errores**: Lanza error si la eliminación falla.

getAll()

- **Descripción**: Obtiene todos los comercios registrados.
- **Retorna**: Lista de comercios.
- **Errores**: Lanza error si la consulta falla.

getTopRated(limit = 10, offset = 0, filters = {})

- **Descripción**: Obtiene los comercios mejor calificados desde la vista commerces_top_rated.
- **Parámetros**:
  - limit: Número máximo de resultados.
  - offset: Desplazamiento para paginación.
  - filters: Objeto con filtros opcionales (searchTerm, typeId, score).
- **Retorna**: Objeto con data (comercios) y count (total de resultados).
- **Errores**: Lanza error si la consulta falla.

getById(id)

- **Descripción**: Obtiene un comercio por su ID desde la vista vista_comercio_con_tipo.
- **Retorna**: Objeto con los datos del comercio.
- **Errores**: Lanza error si la consulta falla.

getUserInsight(commerceId, userIdAuth)

- **Descripción**: Obtiene información personalizada del usuario sobre un comercio desde la vista v_commerce_user_insight.
- **Retorna**: Datos de interacción del usuario con el comercio.
- **Errores**: Lanza error si la consulta falla.

getCommerceOwner(userIdAuth)

- **Descripción**: Obtiene los comercios donde el usuario autenticado es propietario.
- **Retorna**: Lista de comercios del usuario.

- **Errores**: Lanza error si la consulta falla.

Componente: **FavoriteCommerces.js**

Descripción

El módulo FavoriteCommerces gestiona las operaciones relacionadas con los **comercios favoritos** de los usuarios. Permite consultar, agregar y eliminar favoritos, utilizando Supabase como backend y vistas optimizadas para la lectura.

Clase: FavoriteCommerces

Métodos Estáticos

fetchFavorites(user_id, limit = 10, offset = 0)

- **Descripción**: Obtiene una lista paginada de comercios favoritos de un usuario desde la vista v_favorite_commerces.
- **Parámetros**:
  - user_id: ID del usuario.
  - limit: Número máximo de resultados (por defecto 10).
  - offset: Desplazamiento para paginación.
- **Retorna**: Objeto { data, count } con los resultados y el total.
- **Errores**: Devuelve un array vacío y muestra error en consola si falla.

getFavoritesByUser(userId)

- **Descripción**: Obtiene todos los comercios favoritos de un usuario sin paginación.
- **Retorna**: Lista de comercios favoritos.
- **Errores**: Devuelve un array vacío y muestra error en consola si falla.

addFavorite(userId, commerceId)

- **Descripción**: Agrega un comercio a la lista de favoritos del usuario.
- **Retorna**: true si fue exitoso, false si hubo error.
- **Errores**: Muestra error en consola si falla.

removeFavorite(userId, commerceId)

- **Descripción**: Elimina un comercio de la lista de favoritos del usuario.
- **Retorna**: true si fue exitoso, false si hubo error.
- **Errores**: Muestra error en consola si falla.

Componente Notification.js

Descripción

El módulo Notification gestiona el envío y la configuración de notificaciones dentro del sistema. Utiliza Supabase como backend para insertar y actualizar registros en la tabla notifications.

Clase: Notification

Constructor

- **Parámetros**:
  - id: ID de la notificación (opcional para creación).
  - name: Nombre o título de la notificación.
  - message: Contenido del mensaje.
  - location: Ubicación o contexto relacionado con la notificación.

Método de instancia: sendNotification()

- **Descripción**: Inserta una nueva notificación en la base de datos.
- **Retorna**: Nada explícito; lanza error si falla.
- **Errores**: Lanza Error si ocurre un fallo al insertar.

Método estático: configureNotification(notificationId, newData)

- **Descripción**: Actualiza una notificación existente con nuevos datos.
- **Parámetros**:
  - notificationId: ID de la notificación a actualizar.

- newData: Objeto con los campos a modificar.

- **Errores**: Lanza Error si ocurre un fallo al actualizar.

Componente Product.js

Descripción

El módulo Product centraliza la lógica de gestión de productos en la plataforma. Permite realizar operaciones CRUD (crear, leer, actualizar, eliminar) y consultar productos por comercio o por ID, utilizando Supabase como backend.

Clase: Product

Métodos Estáticos

getProductByCommerce(commerceId = null, productId = null, limit = 10, offset = 0)

- **Descripción**: Obtiene productos desde la vista v_products_with_type_name, filtrando por comercio o por ID de producto.
- **Parámetros**:
  - commerceId: ID del comercio (opcional).
  - productId: ID del producto (opcional).
  - limit: Número máximo de resultados (por defecto 10).
  - offset: Desplazamiento para paginación.
- **Retorna**: Objeto { data, count } con los productos y el total.
- **Errores**: Lanza error si la consulta falla.

getById(productId)

- **Descripción**: Obtiene un producto específico por su ID desde la tabla products.
- **Retorna**: Objeto { data } con los datos del producto.
- **Errores**: Lanza error si la consulta falla.

create({ name, description, price, categoryId, commerceId, imageUrl })

- **Descripción**: Crea un nuevo producto con los datos proporcionados.

- **Retorna**: El producto recién creado.
- **Errores**: Lanza error si la inserción falla.

update(productId, updates)

- **Descripción**: Actualiza un producto existente con los datos proporcionados.
- **Parámetros**:
  - productId: ID del producto a actualizar.
  - updates: Objeto con los campos a modificar.
- **Retorna**: El producto actualizado.
- **Errores**: Lanza error si la actualización falla.

delete(productId)

- **Descripción**: Elimina un producto por su ID.
- **Retorna**: true si la operación fue exitosa.
- **Errores**: Lanza error si la eliminación falla.

Buenas Prácticas

- Utiliza vistas (v_products_with_type_name) para enriquecer la información de productos.
- Maneja errores de forma explícita, lo que facilita el control desde la interfaz.
- Métodos bien estructurados y reutilizables, ideales para integrarse con formularios y listados de productos.

Componente PublicationReactions.js

Descripción

El módulo PublicationReactions gestiona las reacciones de los usuarios (como "me gusta" o "no me gusta") a publicaciones dentro de la plataforma. Permite consultar, establecer, actualizar o eliminar una reacción, utilizando Supabase como backend.

Clase: PublicationReactions

Métodos Estáticos

getUserReaction(userId, publicationId)

- **Descripción**: Obtiene la reacción actual de un usuario sobre una publicación específica.
- **Retorna**: El tipo de reacción ('like', 'dislike', etc.) o null si no existe.
- **Errores**: Lanza error si ocurre un fallo distinto a "sin resultados" (PGRST116).

setReaction(userId, publicationId, reactionType)

- **Descripción**: Establece o actualiza la reacción de un usuario a una publicación.
- **Lógica**:
    - Si la reacción ya existe y es igual a la nueva, se elimina.
    - Si existe pero es diferente, se actualiza.
    - Si no existe, se inserta una nueva.
- **Retorna**:
    - null si se eliminó la reacción.
    - reactionType si se insertó o actualizó.
- **Errores**: Lanza error si ocurre un fallo en cualquiera de las operaciones.

Buenas Prácticas

- El método setReaction encapsula toda la lógica de inserción, actualización y eliminación, lo que simplifica su uso desde la interfaz.
- Se maneja adecuadamente el caso de "no hay reacción previa" sin lanzar errores innecesarios.
- Ideal para integrarse con interfaces de publicaciones que permiten interacción del usuario.

Componente Publications.js

Descripción

El módulo Publications gestiona las publicaciones realizadas por los comercios. Permite crear publicaciones, obtenerlas con reacciones de usuarios, y actualizar contadores de reacciones (likes/dislikes). Utiliza Supabase como backend y vistas optimizadas para enriquecer los datos.

Clase: Publications

Métodos Estáticos

getWithUserReactions(userId, commerceId = null, limit = 10)

- **Descripción**: Obtiene publicaciones desde la vista publicaciones_con_reacciones_usuario, incluyendo información sobre reacciones del usuario.
- **Parámetros**:
  - userId: ID del usuario autenticado.
  - commerceId: (opcional) ID del comercio para filtrar publicaciones.
  - limit: Número máximo de publicaciones a retornar.
- **Retorna**: Lista de publicaciones con campos enriquecidos (likes, dislikes, userReaction).
- **Errores**: Lanza error si la consulta falla.

create({ name, content, commerce_id, owner_id})

- **Descripción**: Crea una nueva publicación con los datos proporcionados.
- **Campos iniciales**: likes y dislikes se inicializan en 0.
- **Retorna**: La publicación recién creada.
- **Errores**: Lanza error si la inserción falla.

incrementReaction(id, type = 'likes')

- **Descripción**: Incrementa el contador de reacciones (likes o dislikes) de una publicación.
- **Parámetros**:
  - id: ID de la publicación.
  - type: Tipo de reacción a incrementar ('likes' o 'dislikes').
- **Retorna**: Publicación actualizada.
- **Errores**: Lanza error si la operación falla.

decrementReaction(id, type = 'likes')

- **Descripción**: Decrementa el contador de reacciones (likes o dislikes) de una publicación, sin permitir valores negativos.
- **Parámetros**:

    - id: ID de la publicación.
    - type: Tipo de reacción a decrementar.
- **Retorna**: Publicación actualizada.
- **Errores**: Lanza error si la operación falla.

- Utiliza vistas para enriquecer los datos con reacciones del usuario.
- Controla los valores mínimos en los contadores de reacciones.
- Maneja errores de forma explícita, facilitando el control desde la interfaz.
- Ideal para integrarse con interfaces de publicaciones y sistemas de interacción social.

Componente Report.js

Descripción

El módulo Report permite la creación de reportes dentro del sistema. Está diseñado para insertar registros en la tabla reports de Supabase, encapsulando la lógica de generación de reportes de forma sencilla y reutilizable.

Clase: Report

Constructor

- **Parámetros**:
    - id: ID del reporte (opcional, útil para instancias existentes).
    - type: Tipo de reporte (por ejemplo, ventas, actividad, etc.).
    - content: Contenido del reporte (puede ser texto, JSON, etc.).

Método Estático: generateReport({ type, content })

- **Descripción**: Inserta un nuevo reporte en la base de datos.
- **Parámetros**:
    - type: Tipo de reporte.
    - content: Contenido del reporte.
- **Retorna**: El reporte recién creado.

- **Errores**: Lanza un Error si ocurre un fallo durante la inserción.

- El método generateReport es simple y directo, ideal para ser llamado desde interfaces administrativas o procesos automáticos.
- Se recomienda validar el contenido del reporte antes de enviarlo a la base de datos.
- Puede extenderse fácilmente para incluir filtros, fechas o exportaciones.

Componente Types.js

Descripción

El módulo Types proporciona acceso a los diferentes tipos definidos en la base de datos, como categorías de productos o comercios. Utiliza Supabase para consultar la tabla types, permitiendo filtrar por ámbito (scope).

Clase: Types

Método Estático: getAll(scope = null)

- **Descripción**: Obtiene todos los registros de la tabla types. Si se proporciona un scope, filtra los resultados por ese valor.
- **Parámetros**:
    - scope *(opcional)*: Valor que define el contexto del tipo (por ejemplo, 'commerce' o 'product').
- **Retorna**: Lista de tipos disponibles según el filtro aplicado.
- **Errores**: Lanza un Error si ocurre un fallo en la consulta.

- El método es simple, reutilizable y flexible gracias al parámetro opcional scope.
- Ideal para alimentar selectores dinámicos en formularios (como en TypeFilterSelect).
- Se recomienda manejar errores visualmente en la interfaz para mejorar la experiencia del usuario.

Componente User.js

Descripción

El módulo User centraliza la lógica relacionada con la autenticación y gestión de usuarios. Permite registrar, iniciar sesión, recuperar contraseñas y consultar o actualizar información del usuario en la base de datos. Utiliza Supabase como backend.

Clase: User

Constructor

- Crea una instancia de usuario con sus propiedades básicas.

Autenticación

register({ name, email, password })

- **Descripción**: Registra un nuevo usuario en Supabase Auth.
- **Retorna**: Objeto user creado.
- **Errores**: Lanza error si el registro falla.

login({ email, password })

- **Descripción**: Inicia sesión con correo y contraseña.
- **Retorna**: Objeto user autenticado.
- **Errores**: Lanza error si las credenciales son incorrectas.

forgetPassword(email)

- **Descripción**: Envía un correo para restablecer la contraseña.
- **Retorna**: true si el correo fue enviado correctamente.
- **Errores**: Lanza error si la operación falla.

Consultas de Usuario

getUserNameByEmail(email)

- **Descripción**: Obtiene solo el nombre del usuario desde la tabla usuarios.
- **Retorna**: nombreusuarios.

Componente Commerce.jsx

Este componente muestra la página de un comercio donde los usuarios pueden ver:

- **Publicaciones** del comercio.
- **Productos** que ofrece.
- Marcar el comercio como **favorito**.
- Si el usuario es el dueño, puede **crear, editar o eliminar** productos y publicaciones.

Funciones principales

1. **Autenticación y navegación**
    - Usa useAuth para obtener el usuario actual.
    - Si no hay usuario, redirige a la página principal (/).
2. **Carga de datos**
    - Obtiene los datos del comercio, productos y publicaciones desde servicios externos (Commerce, Product, Publications).
    - Usa paginación para los productos.
3. **Componentes de interfaz**
    - Header, NavBar, DynamicSidebar, FloatingScoreCard para la estructura.
    - PublicationInfoBox y ProductCard para mostrar publicaciones y productos.
    - PublicationModal, ProductModal, ProductDeleteModal para crear, editar o eliminar contenido.
4. **Gestión de estado**
    - Controla qué vista se muestra (publicaciones o productos).
    - Maneja la apertura/cierre de modales.
    - Controla la paginación y los datos cargados.
5. **Favoritos**
    - Permite a los usuarios marcar o desmarcar un comercio como favorito.
6. **Renderizado condicional**
    - Muestra diferentes botones y opciones si el usuario es el dueño del comercio.

Componente Dashboard.jsx

El componente Dashboard muestra una lista de **publicaciones** relacionadas con comercios, incluyendo las reacciones del usuario (me gusta / no me gusta). Es la vista principal que ve un usuario al iniciar sesión.

Funciones principales

1. **Autenticación**
    - Usa useAuth para obtener el usuario actual.
    - Si no hay usuario, redirige a la página principal (/).
2. **Carga de publicaciones**
    - Llama a Publications.getWithUserReactions(user.id) para obtener publicaciones con las reacciones del usuario.
    - Guarda los datos en el estado publications.
3. **Navegación**
    - Al hacer clic en una publicación, redirige a la página del comercio correspondiente usando navigate(/commerce?id=${id}).
4. **Interfaz**
    - Usa componentes como Header, NavBar, Container y PublicationInfoBox para estructurar y mostrar la información.

Componente FavoriteCommercers.jsx

Este componente muestra una lista paginada de los comercios que el usuario ha marcado como favoritos. Permite navegar a la página de cada comercio.

Funciones principales

1. **Autenticación**
    - Usa useAuth para obtener el usuario actual.
    - Si no hay usuario, redirige a la página principal (/).
2. **Carga de comercios favoritos**
    - Llama a favoriteCommercersService.fetchFavorites para obtener los comercios favoritos del usuario.

- Usa paginación con itemsPerPage = 6.

3. **Interfaz**
    - Usa componentes como Header, NavBar, Container, CommerceInfoBox y Paginator.
    - Muestra cada comercio con su imagen, nombre, ubicación, calificación promedio y número de comentarios.
4. **Navegación**
    - Al hacer clic en un comercio, redirige a su página de detalle (/commerce?id=...).

.

Componente Home.jsx

Este componente representa la **página de inicio** de la aplicación. Muestra una interfaz pública con un encabezado, una sección principal (hero), una cuadrícula de negocios y modales para iniciar sesión o registrarse. Si el usuario ya está autenticado, lo redirige automáticamente al **dashboard**.

Funciones principales

1. **Autenticación y redirección**
    - Usa useAuth para obtener el usuario actual.
    - Si el usuario está autenticado, lo redirige automáticamente a /dashboard.
2. **Interfaz de usuario**
    - Muestra los siguientes componentes:
        - Header: Encabezado de la página.
        - Hero: Sección principal destacada.
        - BusinessGrid: Cuadrícula de negocios visibles públicamente.
        - LoginModal y RegisterModal: Modales para iniciar sesión o registrarse.
3. **Estado local**
    - Controla la visibilidad de los modales de login y registro con useState.

Componente maps.jsx

Este componente muestra una vista de mapa (probablemente con ubicaciones de comercios u otros elementos) dentro de un contenedor estructurado. Solo es accesible si el usuario ha iniciado sesión.

Funciones principales

1. **Autenticación**
    - Usa useAuth para verificar si el usuario está autenticado.
    - Si no hay usuario, redirige automáticamente a la página principal (/).
2. **Interfaz de usuario**
    - Usa los siguientes componentes:
        - Header: Encabezado de la aplicación.
        - NavBar: Barra de navegación.
        - Container: Contenedor con el título "Mapa".
        - ContainerMaps: Componente que probablemente renderiza el mapa.

Componente: Profile.jsx

Descripción

Este componente representa la **página de perfil del usuario**. Permite visualizar y modificar datos personales, gestionar tiendas propias y administrar tiendas favoritas.

Componentes utilizados

- Header, NavBar: Encabezado y barra de navegación.
- DynamicSidebar: Barra lateral con botones para cambiar de vista.
- PersonalData: Muestra los datos personales del usuario.
- StoreList: Lista de tiendas (propias o favoritas).
- FavoriteRemoveModal: Modal para confirmar la eliminación de una tienda favorita.
- ProfileModifyModal: Modal para editar el perfil del usuario.

Estados locales

- nombre: Nombre del usuario.
- type: Vista activa ("Infomacion general" o "Tiendas favoritas").
- commerces: Tiendas propias del usuario.
- favorites: Tiendas marcadas como favoritas.
- isOpenFavorite, isOpenProfile: Controlan la visibilidad de los modales.
- commerceId: ID de la tienda seleccionada para eliminar de favoritos.

Funciones clave

- fetchNombre(): Obtiene el nombre del usuario por su correo.
- fetchCommerceOwner(): Carga las tiendas propias del usuario.
- fetchFavorites(): Carga las tiendas favoritas del usuario.
- fetchRemoveFavorite(): Elimina una tienda de favoritos.
- fetchModifyUser(profile): Actualiza los datos del perfil del usuario.
- onStoreClick(store): Navega al perfil de una tienda.
- onAddCommerce(): Redirige a la vista para agregar una tienda.
- handleSidebarButtonClick(label): Cambia entre vistas del perfil.

Flujo de uso

1. Al cargar el componente, se verifica si el usuario está autenticado.
2. Se cargan los datos del usuario, sus tiendas y favoritos.
3. El usuario puede:
    - Ver y editar su información personal.
    - Agregar nuevas tiendas.
    - Ver y eliminar tiendas favoritas.

Componente: ProfileCommerce.jsx

Descripción g

Este componente representa la **vista de perfil de una tienda específica**. Permite al propietario de la tienda ver su información, actualizar la imagen del comercio, editar los datos o eliminar la tienda.

Componentes utilizados

- Header, NavBar: Encabezado y barra de navegación.
- ImageUploadField: Componente para subir o cambiar la imagen del comercio.
- CommerceDeleteModal: Modal de confirmación para eliminar la tienda.

Estados locales

- commerce: Objeto con los datos del comercio.
- isOpen: Controla la visibilidad del modal de eliminación.

Funciones clave

- fetchCommerce(): Obtiene los datos del comercio por su ID.
- setImage(url): Actualiza la imagen del comercio.
- onEdit(): Redirige a la vista de edición de la tienda.
- onDelete(): Abre el modal de confirmación para eliminar.
- deleteCommerce(): Elimina el comercio y redirige al perfil del usuario.

Flujo de uso

1. Al cargar el componente, se obtiene el ID del comercio desde la URL.
2. Se cargan los datos del comercio desde el backend.
3. El usuario puede:
    - Cambiar la imagen del comercio.
    - Ver información como nombre, categoría y dirección.
    - Editar o eliminar la tienda.

Componente: SearchResult.jsx

Descripción

Este componente muestra los **resultados de búsqueda de comercios**, permitiendo aplicar filtros por tipo de comercio y calificación (estrellas). También incluye paginación para navegar entre los resultados.

Componentes utilizados

- Header, NavBar: Encabezado y barra de navegación.
- Container: Contenedor general de la vista.
- TypeFilterSelect: Selector de tipo de comercio.
- StoreList: Lista de comercios encontrados.
- Paginator: Componente de paginación.
- Star (de lucide-react): Íconos de estrellas para filtrar por calificación.

Funciones clave

- useQuery(): Extrae parámetros de la URL (como name).
- fetchCommerce(): Llama a Commerce.getTopRated() para obtener comercios filtrados por nombre, tipo y calificación.
- handleStarClick(index): Aplica el filtro de calificación.
- onStoreClick(store): Redirige a la página del comercio seleccionado.

Flujo de uso

1. El componente obtiene el parámetro name desde la URL.
2. Se cargan los comercios desde el backend aplicando los filtros seleccionados.
3. El usuario puede:
    - Filtrar por tipo de comercio.
    - Filtrar por calificación (estrellas).

- Navegar entre páginas de resultados.
- Hacer clic en un comercio para ver su perfil.

Componente: YourCommerce.jsx

Descripción

Este componente permite a los usuarios **crear o editar una tienda**. Incluye un formulario detallado para ingresar el nombre, categoría y dirección del comercio, con validaciones básicas y generación automática del campo de dirección.

Estados locales

- Información general: nombreTienda, categoria
- Dirección (dividida en partes): tipoVia, numero1, letra1, bis1, cardinalidad1, numero2, letra2, bis2, cardinalidad2, numero3, datosAdicionales
- address: Dirección completa generada automáticamente.
- errorMessage: Mensaje de error si faltan campos obligatorios.

Funciones clave

- fetchCreateOrUpdate(): Crea o actualiza una tienda según si existe un id en la URL.
- parseAddress(address): Parsea una dirección en formato colombiano y llena los campos del formulario.
- fetchCommerce(): Si hay un id, carga los datos de la tienda para edición.
- useEffect():
    - Uno genera la dirección completa a partir de los campos individuales.
    - Otra carga los datos de la tienda y verifica si el usuario está autenticado.

Flujo de uso

1. Si hay un id en la URL, se cargan los datos de la tienda para edición.
2. El usuario completa el formulario con:

- Nombre de la tienda
- Categoría
- Dirección dividida en partes

3. Al hacer clic en **Guardar**, se valida el formulario y se envía la información al backend.
4. Si se está creando una tienda nueva, redirige al perfil del usuario; si se está editando, redirige al perfil de la tienda.

Componente: App.jsx

Descripción

Este archivo define la estructura principal de la aplicación React utilizando **React Router** para la navegación entre páginas. También incluye el proveedor de autenticación y un sistema de notificaciones.

Componentes y librerías utilizadas

- react-router-dom: Para definir rutas y navegación.
- react-hot-toast: Para mostrar notificaciones emergentes.
- AuthProvider: Contexto de autenticación global.
- Páginas importadas:
    - Home, Dashboard, Maps, FavoriteCommercers, Commerce, Profile, YourCommerce , ProfileCommerce, SearchResult.