📘 Guía del Proyecto: Áurea Virtual Shop (Liliam Boutique)

Documento guía para explicar de forma clara la estructura, los recursos, los aplicativos (frontend y backend) y los aspectos clave del sistema.

- Público objetivo: stakeholders técnicos y no técnicos.
- Alcance: arquitectura, estructura del repo, flujos funcionales, endpoints, modelos, variables, operación y despliegue.

---

1) Resumen Ejecutivo

Áurea Virtual Shop es una aplicación eCommerce moderna compuesta por:

- Frontend SPA en React + Vite + Tailwind CSS
- Backend API REST en Node.js + Express
- Base de datos MongoDB (Atlas en la nube)
- Autenticación JWT, carrito con persistencia local, newsletter y contacto por email

Diagrama general: ver `ARCHITECTURE.md`.

---

2) Estructura del Repositorio

```
AureaGH/
├─ frontend/               # Aplicación React + Vite (SPA)
├─ backend/                # API REST (Node.js + Express)
├─ old-static-version/     # Versión estática original (referencia histórica)
├─ README.md               # Resumen general y arranque
├─ ARCHITECTURE.md         # Diagrama/flujo de arquitectura
├─ GETTING_STARTED.md      # Guía rápida de inicio
├─ DEPLOYMENT.md           # Opciones de despliegue
├─ FAQ.md                  # Preguntas frecuentes
└─ GUIA_PROYECTO.md        # Este documento (guía integral)
```

Componentes principales:

- Frontend: páginas, componentes, Context API, servicios Axios.
- Backend: rutas, controladores, modelos Mongoose, middlewares, seed de datos.
- Recursos: MongoDB Atlas, servicio de email (Nodemailer + Gmail), hosting sugerido (Vercel/Netlify + Railway/Render).

---

3) Tecnologías y Decisiones

- Frontend: React 18, Vite, Tailwind CSS, React Router, Axios.
- Backend: Node.js, Express, Mongoose, JWT, Bcrypt, Express Validator, Nodemailer.
- DB: MongoDB Atlas (NoSQL, alta disponibilidad y fácil gestión).
- Autenticación: JWT stateless, token en localStorage, protección vía middleware.
- Estilos: Tailwind por velocidad, consistencia y responsive.
- Deploy sugerido: Vercel (frontend), Railway (backend), MongoDB Atlas (DB).

---

4) Aplicativos

4.1 Frontend (React + Vite)

- Ubicación: `frontend/`
- Entradas clave: `src/App.jsx`, `src/main.jsx`, `src/index.css`
- Arquitectura UI:
  - **Pages**: Home, Mujer, Hombre, Destacados, Promociones, Accesorios, Contacto, Login, Register, Carrito, ProductDetail, Dashboard.
    - **Home**: Página de inicio con hero, categorías destacadas, productos featured y newsletter.
    - **Mujer/Hombre/Accesorios**: Catálogos filtrados por categoría. Muestran grid de `ProductCard`.
    - **Destacados**: Productos marcados como `featured: true`. Carrusel o grid especial.
    - **Promociones**: Productos `onSale: true`. Muestra precio original tachado y descuento.
    - **Contacto**: Formulario de contacto (nombre, email, mensaje) → POST `/contact/send`.
    - **Login/Register**: Formularios de autenticación. Redirigen a Dashboard tras éxito.
    - **Carrito**: Lista de items, modificar quantity, eliminar, calcular total, botón WhatsApp/Checkout.
    - **ProductDetail**: Vista detallada con galería, descripción, tallas/colores, agregar al carrito, reseñas.
    - **Dashboard**: Página privada. Muestra perfil, órdenes, favoritos, permite editar datos.
  - **Components**: Header, Footer, Layout, ProductCard, CategoryCard, WhatsAppButton, PrivateRoute.
    - **Header**: Navegación principal, logo, links de categorías, buscador, badge de carrito, botón login/logout.
    - **Footer**: Links legales, redes sociales, newsletter, copyright.
    - **Layout**: Wrapper con `<Outlet />` de React Router. Comparte Header/Footer en todas las páginas.
    - **ProductCard**: Tarjeta de producto con imagen, nombre, precio, badge, botón "Agregar al Carrito", ícono de favorito.
    - **CategoryCard**: Tarjeta clickeable para navegar a categorías. Usada en Home.
    - **WhatsAppButton**: Botón flotante que abre chat de WhatsApp con mensaje predefinido.
    - **PrivateRoute**: HOC que verifica `isAuthenticated`. Si no, redirige a `/login`.
  - **Context**: `AuthContext`, `CartContext`, `ProductContext`, `FavoritesContext`.
    - **AuthContext**: Gestiona autenticación y sesión del usuario.
      - Estados: `user` (datos del usuario actual), `loading` (carga inicial), `isAuthenticated` (booleano).
      - Métodos: `login(email, password)`, `register(userData)`, `logout()`.
      - Funcionalidad: Al montar, verifica token en localStorage, decodifica con `jwtDecode`, valida expiración. Si es válido, llama `/auth/me` para hidratar perfil. En login/register, guarda token y actualiza estado. En logout, limpia token y resetea usuario. Provee contexto global a toda la app.
    - **CartContext**: Administra carrito de compras con persistencia local.
      - Estados: `cartItems` (array de productos con quantity).
      - Métodos: `addToCart(product, quantity)`, `removeFromCart(productId)`, `updateQuantity(productId, quantity)`, `clearCart()`,

`getCartTotal()`, `getCartCount()`.
        - Funcionalidad: Carga carrito desde localStorage al inicio. Detecta duplicados por `_id`, suma quantities o agrega nuevos items. Sincroniza cambios a localStorage vía `useEffect`. Calcula total y contador en tiempo real. Usado en header (badge), página de carrito y checkout.
    - **ProductContext**: Centraliza acceso al catálogo de productos.
        - Estados: `products` (array completo), `loading`, `error`.
        - Métodos: `fetchProducts()`, `getProductsByCategory(category)`.
        - Funcionalidad: Al montar, llama GET `/products` y cachea resultado. Páginas de categorías consumen `products` y filtran localmente o llaman método específico. Evita múltiples requests innecesarios. Puede agregar búsqueda/filtros en el futuro.
    - **FavoritesContext**: Sincroniza favoritos del usuario con backend.
        - Estados: `favorites` (Set de productIds), `loaded` (flag de carga inicial).
        - Métodos: `isFavorite(productId)`, `toggleFavorite(productId)`.
        - Funcionalidad: Si usuario está autenticado, carga GET `/auth/favorites` al inicio y puebla Set. `toggleFavorite` envía POST, actualiza estado local optimísticamente y sincroniza con respuesta del servidor. Requiere autenticación; si no, retorna `{ requiresAuth: true }`. UI usa `isFavorite` para mostrar íconos activos/inactivos.
- **Services (Axios)**: `authService`, `productService`, `orderService`, `generalService` (newsletter/contacto). Base URL configurable (`VITE_API_URL`).
    - **api.js**: Instancia base de Axios. Interceptor de request agrega token JWT automáticamente. Interceptor de response captura errores 401 y redirige a login.
    - **authService**: `loginUser`, `registerUser`, `getCurrentUser`, `updateProfile`, `getMyFavorites`.
    - **productService**: `getAllProducts`, `getProductById`, `getProductsByCategory`, `getFeaturedProducts`, `getPromotions`, `searchProducts`, `addReview`, `toggleFavorite`.
    - **orderService**: `createOrder`, `getMyOrders`, `getOrderById`.
    - **generalService**: `subscribeNewsletter`, `sendContactMessage`.

Flujos clave:

- **Autenticación**:
    1. Usuario ingresa email/password en formulario de login.
    2. Frontend envía POST `/auth/login` con credenciales.
    3. Backend valida datos, busca usuario por email, compara hash con `bcrypt.compare()`.
    4. Si es válido, genera token JWT firmado con `JWT_SECRET` (payload: `{id, email, role}`), expira en 30 días.
    5. Backend retorna `{ token, user: {...} }`.
    6. Frontend guarda token en `localStorage.setItem('token', token)`.
    7. `AuthContext` decodifica token, verifica expiración, llama GET `/auth/me` para obtener perfil completo.
    8. Usuario autenticado → `isAuthenticated: true`, acceso a rutas protegidas (`PrivateRoute`).

9. En cada petición subsecuente, Axios interceptor agrega header `Authorization: Bearer <token>`.
10. Backend middleware `protect` verifica token, decodifica, adjunta `req.user`, permite acceso o retorna 401.

- **Catálogo**:

1. Página (ej. Mujer) monta componente.
2. `useEffect` dispara `productService.getProductsByCategory('mujer')`.
3. Axios envía GET `/products/category/mujer`.
4. Backend busca en DB: `Product.find({ category: 'mujer', active: true })`.
5. Retorna JSON con array de productos.
6. Frontend actualiza estado local (`useState`) o Context (`ProductContext`).
7. Componentes `ProductCard` renderizan cada producto con imagen, nombre, precio, botón "Agregar al Carrito".
8. Usuario puede filtrar, buscar o navegar entre categorías sin reload (SPA).

- **Carrito**:

1. Usuario hace clic en "Agregar al Carrito" en `ProductCard`.
2. `CartContext.addToCart(product, quantity)` se ejecuta.
3. Context verifica si producto ya existe en `cartItems` (por `_id`).
4. Si existe, incrementa `quantity`; si no, agrega nuevo item `{ ...product, quantity }`.
5. `useEffect` detecta cambio en `cartItems`, guarda en `localStorage.setItem('cart', JSON.stringify(cartItems))`.
6. UI actualiza badge de carrito con `getCartCount()` (suma de quantities).
7. Usuario navega a `/carrito`, ve lista de items, puede modificar quantity o eliminar.
8. Clic en "Ordenar por WhatsApp" genera mensaje pre-formateado con lista de productos y total.
9. Abre WhatsApp Web/App con `window.open()` y número de tienda.
10. (Opcional) Crear orden real: POST `/orders` con `cartItems`, dirección, método de pago → guarda en DB → limpia carrito.

- **Favoritos**:

1. Usuario autenticado hace clic en ícono de corazón en `ProductCard`.
2. `FavoritesContext.toggleFavorite(productId)` se ejecuta.
3. Envía POST `/products/:id/favorite` con token en header.
4. Backend verifica token, obtiene `user._id` de `req.user`.
5. Busca producto en `user.favorites[]`. Si existe, lo quita; si no, lo agrega.
6. Guarda usuario actualizado: `user.save()`.
7. Retorna `{ favorited: true/false, favorites: [...] }`.
8. Frontend actualiza estado de `favorites` en Context.
9. UI cambia ícono (relleno vs outline) según `isFavorite(productId)`.
10. Usuario puede ver lista de favoritos en Dashboard o página dedicada.

Variables frontend (`frontend/.env`):

- `VITE_API_URL=http://localhost:3001/api` (o URL de producción)

Scripts frontend:

- `npm run dev` | `npm run build` | `npm run preview`

4.2 Backend (Node.js + Express)

- Ubicación: `backend/`
- Entrada: `src/server.js`
- Capas:
  - `routes/` → define endpoints y protecciones.
  - `controllers/` → lógica de negocio.
  - `models/` → esquemas Mongoose (User, Product, Order, Newsletter).
  - `middleware/` → `auth.middleware` (protect/admin), `validation.middleware`.
  - `config/database.js` → conexión MongoDB.
  - `seeds/seedProducts.js` → datos de ejemplo.

Variables backend (`backend/.env`):

- `PORT=3001`
- `MONGODB_URI=...` (Atlas o local)
- `JWT_SECRET=...`
- `JWT_EXPIRE=30d`
- `EMAIL_USER=...` (Gmail)
- `EMAIL_PASS=...` (contraseña de app)
- `FRONTEND_URL=http://localhost:5173`
- `NODE_ENV=development|production`

Scripts backend:

- `npm run dev` | `npm start` | `npm run seed`

CORS: restringido por `FRONTEND_URL`.

---

5) API (Resumen)

Base: `/api`

5.1 Autenticación (`/auth`)

Gestiona el ciclo de vida de usuarios y sesiones mediante JWT.

- **POST** `/register` (público): Crea un nuevo usuario. Valida email único, hash de contraseña con bcrypt. Retorna token JWT y datos del usuario.
- **POST** `/login` (público): Autentica credenciales (email/password). Compara hash, genera token JWT válido por 30 días. Retorna token y perfil.
- **GET** `/me` (protegido): Obtiene perfil del usuario autenticado. Decodifica token del header `Authorization: Bearer <token>` y retorna datos actuales desde DB.
- **PUT** `/profile` (protegido): Actualiza nombre, teléfono o dirección del usuario. Valida y persiste cambios.
- **GET** `/favorites` (protegido): Lista los productos marcados como favoritos por el usuario actual. Retorna array de IDs o productos completos.

5.2 Productos (`/products`)

CRUD y búsqueda de catálogo.

- **GET** `/` (público): Lista todos los productos activos. Soporta paginación y filtros básicos.
- **GET** `/featured` (público): Productos destacados (`featured: true`). Para carruseles o secciones especiales en Home.

- **GET** `/promotions` (público): Productos en oferta (`onSale: true`). Muestra descuentos y precios originales.
- **GET** `/category/:category` (público): Filtra por categoría (mujer/hombre/accesorios). Esencial para páginas de catálogo.
- **GET** `/search?q=...` (público): Búsqueda full-text por nombre y descripción. Usa índices de texto en MongoDB.
- **GET** `/:id` (público): Detalle completo de un producto (nombre, precio, stock, imágenes, reseñas, tallas, colores).
- **POST** `/` (admin): Crea producto nuevo. Valida campos requeridos (name, price, category). Auto-genera SKU si se omite.
- **PUT** `/:id` (admin): Actualiza producto existente. Permite cambiar stock, precios, estado (active/inactive).
- **DELETE** `/:id` (admin): Marca producto como inactivo o lo elimina (soft/hard delete según lógica).
- **POST** `/:id/reviews` (usuario): Agrega/actualiza reseña y rating (1-5 estrellas). Calcula rating promedio del producto.
- **POST** `/:id/favorite` (usuario): Toggle favorito. Si ya existe, lo quita; si no, lo agrega al array `favorites` del user.

5.3 Órdenes (`/orders`)

Gestión de pedidos y seguimiento.

- **POST** `/` (usuario): Crea pedido. Valida items, dirección de envío, método de pago. Guarda snapshot de precios para auditoría. Retorna orden con estado `pendiente` .
- **GET** `/my-orders` (usuario): Lista órdenes del usuario autenticado. Filtra por `user._id` . Incluye estados y totales.
- **GET** `/:id` (usuario): Detalle de orden específica. Solo si pertenece al usuario autenticado (verificación de ownership).
- **GET** `/` (admin): Lista todas las órdenes del sistema. Para dashboard administrativo. Soporta filtros por estado, fecha, usuario.
- **PUT** `/:id/status` (admin): Actualiza estado de orden (pendiente → procesando → enviado → entregado → cancelado). Puede marcar `isPaid` y `isDelivered` con timestamps.

5.4 Newsletter (`/newsletter`)

Suscripciones a boletín.

- **POST** `/subscribe` (público): Agrega email a lista. Valida formato, previene duplicados (unique index). Marca `active: true` .
- **POST** `/unsubscribe` (público): Marca email como inactivo (`active: false`) o lo elimina. Permite opt-out GDPR-compliant.
- **GET** `/subscribers` (admin): Lista todos los suscriptores activos. Para campañas de email marketing.

5.5 Contacto (`/contact`)

Formulario de contacto.

- **POST** `/send` (público): Recibe mensaje de contacto (nombre, email, asunto, mensaje). Envía email a admin usando Nodemailer. Valida campos y previene spam básico.

Ver detalles ampliados (request/response schemas, códigos de error, ejemplos curl) en `backend/README.md` .

---

6) Modelos de Datos (Mongoose)

6.1 User (Usuarios)

Campos:

- `nombre` (String, requerido): Nombre completo del usuario.
- `email` (String, único, requerido): Email para autenticación. Se normaliza a lowercase.
- `password` (String, requerido, select: false): Hash bcrypt (10 rounds). Nunca se expone en queries por defecto.
- `telefono` (String, opcional): Número de contacto para notificaciones o envío.
- `role` (String, enum: user/admin, default: user): Define permisos. Admin puede gestionar productos y órdenes.
- `direccion` (Object, opcional): Subdocumento con calle, ciudad, departamento, codigoPostal. Usado como dirección por defecto en checkout.
- `favorites` (Array de ObjectId → Product): IDs de productos favoritos. Permite listas de deseos.
- `timestamps` : createdAt/updatedAt automáticos.

Métodos:

- `matchPassword(enteredPassword)` : Compara password en texto plano con hash almacenado. Usado en login.

Hooks:

- Pre-save: Hashea `password` solo si cambió (evita re-hash en updates no relacionados).

Relaciones:

- 1 User → N Orders (un usuario puede tener múltiples pedidos).
- 1 User → N Product (favoritos, many-to-many via array).

6.2 Product (Productos)

Campos principales:

- `name` (String, requerido): Nombre del producto (ej. "Vestido Rojo Elegante").
- `description` (String): Descripción larga, materiales, cuidados.
- `price` (Number, requerido, min: 0): Precio actual de venta (en centavos o pesos según lógica).
- `originalPrice` (Number, opcional): Precio original (para calcular % descuento). Si `onSale: true` .
- `category` (String, enum: mujer/hombre/accesorios, requerido): Categoría principal. Usado para filtros.
- `image` (String): URL o ruta de imagen principal (default placeholder).
- `images` (Array String): Galería de imágenes adicionales (vistas secundarias).
- `stock` (Number, default: 0, min: 0): Cantidad disponible. Control de inventario.
- `sku` (String, único, sparse): Código único de producto. Usado para integraciones de inventario.
- `rating` (Number, 0-5): Promedio de calificaciones. Calculado a partir de `reviews` .
- `reviewsCount` (Number): Total de reseñas. Optimización para evitar contar array cada vez.
- `reviews` (Array): Subdocumentos con `user` , `rating` , `comment` , `createdAt` .
- `badge` (String, enum: Nuevo/Trending/Oferta/''): Etiqueta visual para destacar en UI.
- `featured` (Boolean, default: false): Si aparece en sección destacados.
- `onSale` (Boolean, default: false): Indica si tiene descuento activo.
- `sizes` (Array String, enum: XS/S/M/L/XL/XXL): Tallas disponibles.
- `colors` (Array String): Colores disponibles (ej. ["Rojo", "Negro"]).
- `active` (Boolean, default: true): Si está visible en catálogo. Soft delete.
- `timestamps` : createdAt/updatedAt.

Índices:

- Text index en `name` y `description` (búsqueda full-text).
- Compound index `{ category: 1, featured: 1 }` (optimiza queries de catálogo).
- Index `{ onSale: 1 }` (páginas de promociones).

Relaciones:

- 1 Product → N Reviews (anidados como subdocumentos).
- N Users → N Products (favoritos, relación many-to-many).

6.3 Order (Pedidos)

Campos:

- `user` (ObjectId → User, requerido): Usuario que realizó el pedido. Populate para obtener nombre/email.
- `orderItems` (Array de subdocumentos):
  - `product` (ObjectId → Product, requerido): Referencia al producto.
  - `name` (String): Snapshot del nombre (por si el producto se elimina/edita después).
  - `quantity` (Number, min: 1): Cantidad pedida.
  - `price` (Number): Precio unitario en momento de compra (snapshot para auditoría).
- `shippingAddress` (Object, requerido):
  - `calle` , `ciudad` , `departamento` , `codigoPostal` : Dirección de envío.
- `paymentMethod` (String, enum: whatsapp/transferencia/contraentrega, default: whatsapp): Método elegido. Define flujo de confirmación.
- `itemsPrice` (Number, requerido): Subtotal de productos (sin envío).
- `shippingPrice` (Number, default: 0): Costo de envío (puede ser calculado por zona).
- `totalPrice` (Number, requerido): Total final (itemsPrice + shippingPrice).
- `status` (String, enum: pendiente/procesando/enviado/entregado/cancelado, default: pendiente): Estado del pedido. Define acciones disponibles.
- `isPaid` (Boolean, default: false): Si se confirmó el pago.
- `paidAt` (Date): Timestamp de confirmación de pago.
- `isDelivered` (Boolean, default: false): Si se entregó.
- `deliveredAt` (Date): Timestamp de entrega.
- `timestamps` : createdAt/updatedAt.

Flujo típico:

1. Usuario crea orden (POST /orders) → `status: pendiente` .
2. Admin confirma pago → `isPaid: true` , `paidAt` , `status: procesando` .
3. Admin despacha → `status: enviado` .
4. Admin confirma recepción → `status: entregado` , `isDelivered: true` , `deliveredAt` .

Relaciones:

- N Orders → 1 User (un usuario puede tener muchos pedidos).
- N OrderItems → N Products (snapshot, no depende de cambios posteriores en Product).

6.4 Newsletter (Suscriptores)

Campos:

- `email` (String, único, requerido, lowercase): Email del suscriptor. Índice único previene duplicados.

- `active` (Boolean, default: true): Si está activo (permite unsubscribe sin borrar registro).
- `subscribedAt` (Date, default: now): Fecha de suscripción. Para análisis de crecimiento.
- `timestamps` : createdAt/updatedAt.

Uso:

- Campañas de email marketing (ej. nuevos productos, ofertas).
- Exportable para servicios como Mailchimp, SendGrid.
- GDPR: permite desactivar (`active: false`) sin eliminar historial.

---

7) Seguridad y Buenas Prácticas

- Autenticación JWT con middleware `protect` y roles con `admin` .
- Bcrypt para password hashing.
- Validación con `express-validator` y respuestas de error consistentes.
- CORS limitado a `FRONTEND_URL` en producción.
- Variables de entorno nunca en el repo. Generar `JWT_SECRET` robusto.
- Sanitizar entradas y manejar errores globalmente.

---

8) Entornos, Variables y Recursos

- Desarrollo: localhost (5173 frontend, 3001 backend).
- Producción (sugerido): Vercel/Netlify (frontend), Railway/Render (API), MongoDB Atlas (DB).
- Email: Gmail con contraseña de aplicación (o SendGrid/Mailgun si se desea).

Checklist pre-producción: ver `DEPLOYMENT.md` (CORS, JWT_SECRET, IPs Atlas, variables, build test).

---

9) Operación, Mantenimiento y Soporte

- Logs: dashboards de hosting (Vercel/Railway/Render). Capturar errores relevantes.
- Semillas/datos: `npm run seed` para poblar productos demo.
- Usuarios admin: promover manualmente cambiando `role` en colección `users` .
- Backups: habilitar backups/exports regulares en Atlas.
- Monitoreo: Lighthouse + PageSpeed para frontend; métricas de hosting para backend.
- Troubleshooting: ver `FAQ.md` .

---

10) Roadmap (Sugerido)

- Integración de pagos (Stripe/PayPal/Mercado Pago)
- Panel admin completo para CRUD de productos y gestión de órdenes
- Subida y CDN de imágenes (Cloudinary/S3)
- Tests automáticos (Vitest/Jest + Supertest)
- Internacionalización (i18n) y accesibilidad (a11y)
- Cache y optimizaciones (React Query/RTK Query, ETags en API)

---

11) Glosario

- SPA: Single Page Application
- JWT: JSON Web Token
- ODM: Object Document Mapper (Mongoose)

- CORS: Cross-Origin Resource Sharing
- Seed: Script para poblar datos de ejemplo

---

12) Referencias Rápidas

- Guía de inicio: `GETTING_STARTED.md`
- Despliegue: `DEPLOYMENT.md`
- Arquitectura: `ARCHITECTURE.md`
- Backend/API: `backend/README.md`
- Frontend: `frontend/README.md`
- FAQ: `FAQ.md`

—

Documento mantenido por: Fernando