INFORME GENERAL DEL SISTEMA ZONA-44 (Versión 1.1)

Fecha de generación: 4 de noviembre de 2025
Versión del documento: 1.1

Importante: Este informe será actualizado al estar disponible una versión actualizada del aplicativo.

---

1. RESUMEN EJECUTIVO

ZONA-44 es una plataforma para la gestión de restaurantes y gastro-bares. El backend está desarrollado en Ruby on Rails (API RESTful) y el frontend es una SPA construida con Angular (versión ~20.2). El frontend y el backend se comunican mediante una API (base: http://localhost:3000/api/v1 en desarrollo). La aplicación gestiona usuarios, productos, pedidos, promociones y reservas; además integra autenticación con Google y persistencia local de carrito.

---

2. ARQUITECTURA GENERAL

- Patrón: MVC en backend (Rails), SPA en frontend (Angular).
- Comunicación: API RESTful (JSON).
- Deploy esperado: Containerización (Docker), servidor web Puma en backend.
- Autenticación: token-based (token devuelto por backend, almacenado en localStorage).
- Internacionalización: @ngx-translate con archivos en.json y es.json.

---

3. STACK Y DEPENDENCIAS PRINCIPALES (Frontend)

- Angular 20.2.x
- TypeScript ~5.9.x
- RxJS ~7.8
- @ngx-translate/core + @ngx-translate/http-loader (i18n)
- Leaflet, Chart.js, ng2-charts **(mapas y gráficos)**
- Google sign-in: integracion mediante la librería oficial (google.accounts.id)
- Testing: Karma + Jasmine

Archivo relevante: package.json en frontend_angular.

---

4. ESTRUCTURA Y COMPONENTES DEL FRONTEND

App raíz: app.ts (standalone component, usa TranslationModule y RouterOutlet).

- **Rutas principales: app.routes.ts**
  - Rutas públicas: / (Home), /menu, /reservas, /seguimiento, /order, /pago, /login, /register, /forgot-password.
  - Rutas protegidas (AuthGuard): /perfil, /admin/* (dashboard y sub-rutas admin: grupos, productos, promociones, pedidos, reservas).
- **Servicios clave:**
  - AuthService (auth.service.ts): maneja login, register, login con Google (/auth/google), carga perfil (/profile), persistencia de token y usuario en localStorage.
    - Base API: http://localhost:3000/api/v1
    - Endpoints usados (dev): /login, /register, /auth/google, /profile
  - GoogleAuthService: renderiza botón Google, captura ID token y envía al backend con AuthService.loginWithGoogle.
  - GlobalCartService: gestión del carrito en memoria/reactiva con BehaviorSubject y persistencia en localStorage (global_cart). Operaciones: add, update, remove, clear, total.
  - PromocionesService: actualmente usa datos mock en frontend (no consulta API).
  - LanguageService: inicializa y cambia idioma con ngx-translate.
- **Componentes principales:**
  - Home (página principal), Menu, Order, Pago, Perfil, Reservas, Seguimiento.
  - Shared: navbar, footer, carrito (componente standalone que define la interfaz CarritoItem), promociones.
  - Admin: carpeta con componentes para gestión (productos, grupos, promociones, pedidos, reservas).
- **Contratos de datos:**
  - CarritoItem (id, name, precio, cantidad, foto_url)
  - PromocionPublica (id, title, description, image, newPrice, oldPrice, isActive...)
  - User y AuthResponse en AuthService
- **Persistencia local:**
  - Token: localStorage['auth_token']
  - Usuario: localStorage['current_user']
  - Carrito: localStorage['global_cart']

5. MAPA DE RELACIÓN ENTRE FRONTEND Y BACKEND

- **Autenticación:**
  - Frontend: `AuthService.login` -> POST /login (dev). Guarda `token` en localStorage.
  - Backend (Rails): Responsable de endpoints /login, /register, /auth/google, /profile, implementación probable en controllers (SessionsController, RegistrationsController, Omniauth/Google endpoint).
- **Usuarios:**
  - Frontend: Perfil usa `AuthService.loadUserProfile` (/profile) y `updateProfile` (PUT /profile) y `deleteProfile` (DELETE /profile).
  - Backend: Modelo `User`, controladores para perfil y gestión de roles.
- **Pedidos y carrito:**
  - Frontend: `GlobalCartService` gestiona carrito local; Order page probablemente crea Order en backend durante checkout (no se encontró el servicio explícito de órdenes en los archivos leídos pero existe orders.service.ts en la estructura).
  - Backend: Modelos Order, OrderItem, controlador CheckoutController/OrdersController para persistir pedidos, generación de facturas (mailers).
- **Productos / Menú / Grupos:**
  - Frontend: `Menu` y admin/productos, admin/grupos.
  - Backend: Controladores ProductosController, GruposController, modelos Producto, Grupo.
- **Promociones:**
  - Frontend: `PromocionesService` mockea promociones (recomendación: pasar a llamadas API).
  - Backend: PromocionesController, mailers relacionados si aplica.
- **Reservas:**
  - Frontend: `ReservasComponent`.
  - Backend: ReservationsController, TableReservationMailer.
- **Notificaciones / Mailers:**
  - Backend: CustomerInvoiceMailer, OrderMailer, TableReservationMailer, PasswordResetMailer.
  - Frontend: solo recibe resultados y muestra notificaciones simples (ej. alert en Home.showNotification).
- **Jobs / background:**
  - Backend: Jobs como CleanupOrdersJob y otros.
  - Frontend: no aplica.

6. OBSERVACIONES TÉCNICAS (problemas detectados y riesgos)

1. Token en `localStorage`:
    - Riesgos: XSS puede exponer token. Ideal usar cookies HttpOnly+Secure o implementar refresh tokens con httpOnly cookies.
2. Cabecera Authorization:
    - `AuthService` envía cabecera 'Authorization': token (sin prefijo Bearer). Esto funciona solo si backend acepta token raw en ese header. Recomiendo estandarizar a Authorization: Bearer <token> o usar un HttpInterceptor que formatee correctamente.
3. Manejo de errores:
    - Hay catchError que "simulan" éxito cuando el backend no está disponible (modo testing). Esto es útil en dev, pero peligroso en producción si no condicionado.
4. Promociones en frontend:
    - Actualmente mock (hard-coded). Falta integración con endpoint real de promociones.
5. Google Auth:
    - El frontend envía `id_token` al backend; es crítico que el backend valide ese token con las librerías de Google (no confiar en frontend).
6. CSRF y CORS:
    - Dado que la app es SPA y backend API, revisar configuración CORS en Rails y CSRF si se usa cookies.
7. Guard y control de rutas:
    - `AuthGuard` solo verifica existencia de token (booleano). Mejor validar token y su vigencia; considerar interceptor que chequee respuestas 401 para redirigir.
8. Seguridad en general:
    - Revisar sanitización de inputs, validaciones tanto en frontend como en backend, rate limiting y protección contra fuerza bruta en endpoints de autenticación.
9. Persistencia del carrito:
    - Cart en localStorage es práctico, pero sincronización con backend (cuando el usuario se autentica) no está explícita: recomendar merge del carrito local con el servidor al loguear.

---

7. CONTRATO MÍNIMO (inputs/outputs y criterios)

- Inputs:
    - Peticiones API JSON (login/register, crear pedido, obtener productos, reservas).
    - Eventos del usuario (añadir al carrito, checkout).
- Outputs:
    - Respuestas JSON con estructura { success: boolean, token?, message?, errors? }.
    - Mailers para facturación y reservas.
- Error modes:

    - 4xx respuestas de validación, 401/403 para auth, 5xx para fallos de servidor.
- Éxito:
    - Token válido devuelto y usuario cargado en frontend; pedido persistido y mail enviado.

---

8. EDGE CASES A TENER EN CUENTA

- Carrito corrupto en localStorage (JSON inválido).
- Token expirado o revocado en backend (frontend debe detectar 401).
- Fallo en verificación de Google id_token si backend no valida.
- Promociones expiradas mostradas por frontend si no hay validación server-side.
- Race conditions al actualizar cantidades del carrito desde múltiples pestañas.
- Usuarios con roles cambiantes (admin/cliente): cache del usuario en localStorage desactualizada.

---

9. RECOMENDACIONES (priorizadas y concretas)

Prioridad alta (seguridad y corrección)

- Implementar un HttpInterceptor en Angular para:
    - Adjuntar cabecera Authorization: Bearer <token> a todas las peticiones.
    - Manejar 401/403 centralmente (redirigir a login, limpiar token).
- Migrar token a mecanismo más seguro:
    - Preferible: cookies HttpOnly+Secure con SameSite=strict o lax + CSRF tokens si se usa sesión.
    - Si se mantiene localStorage, agregar refresh token y expiración corta.
- Validar ID tokens de Google en backend con librería oficial de Google y usar claims (aud, iss, expiry).
- No "simular" éxito por defecto en producción; condicionar modo testing con flag (env var).

Prioridad media (calidad/arquitectura)

- Integrar `PromocionesService` con endpoint real y eliminar mocks.
- Implementar sincronización del carrito al loguear: merge servidor <-> local.
- Versionado de API: exponer /api/v1/... ya existe; documentar y agregar header/accept-version.
- Añadir HttpInterceptors para logging y manejo de errores (user-friendly UI).
- Agregar validaciones UI y manejo de errores detallado (mostrar mensajes traducidos con ngx-translate).

Prioridad baja (mejoras y ops)

- Tests: añadir tests unitarios para `GlobalCartService`, `AuthService` (mocks), y tests E2E para flujo checkout.

- Rendimiento: cachear recursos estáticos (CDN) y usar estrategias de lazy-loading en Angular para módulos admin.
- Monitoring: agregar Sentry/LogRocket y métricas de frontend; en backend, Prometheus/Datadog.
- Websockets: considerar websocket (ActionCable en Rails o similar) para actualizaciones en tiempo real de pedidos.

Pequeñas mejoras inmediatas (rápidas, alto ROI)

- Centralizar `apiUrl` en archivo de environment (ya existe `environment.ts`), usarlo en todos los servicios.
- Reemplazar `alert()` por un componente de notificaciones (toast) consistente y traducible.
- Asegurar que `AuthService.setToken` añade prefijo Bearer o que interceptor lo añada.

---

10. Tareas sugeridas (próximos pasos técnicos)

1. Implementar un HttpInterceptor que:
   - Añada Authorization: Bearer <token>.
   - Maneje 401 globalmente.
2. Cambiar `PromocionesService` para consumir API real (/promociones) y eliminar mock.
3. Añadir test unitarios para `GlobalCartService` (add/update/remove/clear/getTotal).
4. Revisar backend Rails para:
   - Asegurar validación de Google id_token.
   - Aceptar Authorization: Bearer y/o documentar formato.
5. Revisar políticas CORS y despliegue HTTPS.

---

11. Mapeo archivo-a-responsabilidad (resumen de los archivos clave leídos)

- `package.json` — dependencias y scripts del frontend.
- `app.routes.ts` — definición de rutas y protecciones (AuthGuard).
- `app.ts` — componente raíz, arranque de la app.
- `auth.service.ts` — login/register, google login, perfil y manejo de token.
- `google-auth.service.ts` — integración cliente Google sign-in.
- `global-cart.service.ts` — manejo reactivo del carrito con localStorage.
- `promociones.service.ts` — promociones (mock).
- `auth.guard.ts` — guard para rutas protegidas.
- `carrito.ts` — interfaz `CarritoItem` y UI del carrito.
- `language.service.ts` — manejo inicial de idioma con ngx-translate.

12. Conclusión

La integración actual entre frontend Angular y backend Rails es funcional y sensata para un MVP: i18n, integración Google Sign-In, carrito persistente y estructura modular en admin. Sin embargo, hay varios puntos críticos de seguridad y de calidad que conviene arreglar antes de producción (manejo de tokens, validación de Google ID token en backend, corrección del header Authorization, no simular éxitos de backend en producción, y sustituir mocks por endpoints reales). Adoptando las recomendaciones priorizadas se mejora seguridad, mantenibilidad y experiencia de usuario.