

AUDITORÍA FORENSE TÉCNICA — CONTROL GASTRONÓMICO V2

Fecha: 2026-01-18

Auditor: Lead Software Architect

Clasificación: ANÁLISIS PROFUNDO - CÓDIGO EXISTENTE

RESUMEN EJECUTIVO

[!IMPORTANT]

Este análisis es **exclusivamente sobre código existente**. No se evalúan funcionalidades faltantes ni comparaciones con roadmaps. Se evalúa la calidad intrínseca de lo implementado.

Veredicto Rápido: El codebase demuestra una madurez técnica **superior a la media** para proyectos de su escala. No es perfecto, pero es **defendible para producción enterprise** con refactorizaciones menores.

1. ARQUITECTURA Y DISEÑO DEL SISTEMA

1.1 Integridad Frontend-Backend

Aspecto	Evaluación	Detalles
Separación de Concerns	<input checked="" type="checkbox"/> SÓLIDA	El frontend no contiene lógica de negocio. Toda la validación crítica ocurre en el backend.
State Management	<input checked="" type="checkbox"/> CORRECTO	Zustand para estado global (auth, pos, cash, kitchen) con persistencia. Sin Redux overhead innecesario.
API Integration	<input checked="" type="checkbox"/> ROBUSTO	Axios con interceptors, auto-logout en 401, user-friendly messages.

Fortalezas detectadas:

- El frontend implementa **offline-first** con IndexedDB (offlineDb.ts, syncManager.ts)
- Conexión WebSocket vía Socket.IO para real-time (KDS)
- API versionada (/api/v1/) — correcto para evolución futura

Debilidad menor:

```
// frontend/src/services/orderService.ts:156
updateItemStatus: async (itemId: number, status: string): Promise<any> => {
```

Uso de any como retorno. Debería tiparse explícitamente.

1.2 Autopsia del Schema de Base de Datos

El schema Prisma (schema.prisma, 665 líneas, ~35 modelos) está **sorprendentemente bien diseñado**.

Indexación

Tabla	Índices	Evaluación
Order	tableId, clientId, serverId, driverId, businessDate, createdAt, externalId	<input checked="" type="checkbox"/> Excelente cobertura
OrderItem	orderId, productId	<input checked="" type="checkbox"/> Correcto
Payment	orderId, shiftId	<input checked="" type="checkbox"/> Correcto
AuditLog	userId, (entity, entityId), action, createdAt	<input checked="" type="checkbox"/> Excelente para queries de auditoría

No se detectan **N+1 query issues** obvios en el servicio principal — los includes están bien configurados.

Relaciones y Constraints

```
model Order {
  @@index([tableId])
  @@index([clientId])
  @@index([businessDate]) // ← Inteligente para partición por día operativo
}
```

Decisión arquitectónica positiva: Uso de `businessDate` separado de `createdAt` para reportes. Esto evita problemas de timezone en cierres nocturnos.

Potencial Bottleneck Identificado

```
model OrderSequence {
  id          Int @id @default(1)
  lastNumber  Int @default(0)
}
```

El servicio `orderNumber.service.ts` implementa correctamente atomic increment con `upsert` + fallback a raw SQL. **Bien manejado.**

```
// Atomic increment, no locks en Order table
const sequence = await tx.orderSequence.upsert({
  where: { id: 1 },
  update: { lastNumber: { increment: 1 } },
  create: { id: 1, lastNumber: 1 },
});
```

1.3 Modularidad

Estructura Backend:

```
backend/src/
├─ controllers/    (26 archivos) - Thin controllers, correcto
├─ services/       (35 archivos) - Business logic concentrada
├─ middleware/     (4 archivos) - Auth, error, rate limit
├─ integrations/  (16 archivos) - Delivery adapters, webhooks
├─ lib/            (queue, prisma, socket)
├─ routes/        (25 archivos)
└─ utils/         (errors, logger, response)
```

Evaluación: Modular Monolith bien estructurado, no Big Ball of Mud.

El `order.service.ts` (444 líneas) fue identificado previamente como "God Object", pero se observa **delegación activa** a servicios especializados:

```
class OrderService {
  // @delegates orderDeliveryService.assignDriver
  async assignDriver(...) { return orderDeliveryService.assignDriver(...); }

  // @delegates orderKitchenService.getActiveOrders
  async getActiveOrders(...) { return orderKitchenService.getActiveOrders(); }
}
```

Esto es **Facade Pattern**, no God Object. El patrón es correcto.

2. CALIDAD DE CÓDIGO Y PATRONES

2.1 Principios SOLID y Clean Code

Principio	Cumplimiento	Evidencia
S (SRP)	<input checked="" type="checkbox"/> Alto	Servicios separados por dominio
O (OCP)	<input checked="" type="checkbox"/> Medio-Alto	Adapter pattern en delivery integrations
L (LSP)	<input checked="" type="checkbox"/> Aplicable	AbstractDeliveryAdapter correctamente heredado
I (ISP)	<input checked="" type="checkbox"/> Medio	Interfaces de queue bien definidas
D (DIP)	<input checked="" type="checkbox"/> Alto	Inyección via factory (AdapterFactory)

Patrón destacable — Abstract Factory:

```
// AbstractDeliveryAdapter.ts
export abstract class AbstractDeliveryAdapter {
  protected abstract get platformCode(): DeliveryPlatformCode;
  protected abstract getDefaultBaseUrl(): string;
  abstract validateWebhookSignature(
    signature: string,
    rawBody: Buffer,
  ): boolean;
  abstract parseWebhookPayload(rawPayload: unknown): ProcessedWebhook;
  // ...
}
```

Implementado correctamente en RappiAdapter.ts (14KB). Extensible para Glovo, PedidosYa, UberEats.

2.2 Anti-Patterns Detectados

Uso de any

Se detectaron ~50 instancias de any. Clasificación:

Contexto	Cantidad	Severidad
Tests (mocks)	~15	○ Aceptable
Catch errors (catch (err: any))	~10	⦿ Menor
Servicios (data: any)	~8	● Problemático
JSON fields (theme: any)	~5	⦿ Menor

Ejemplos problemáticos:

```
// product.service.ts:61
export const createProduct = async (data: any) => { ... }

// product.service.ts:104
export const updateProduct = async (id: number, data: any) => { ... }
```

[!WARNING]

Estos servicios deberían usar schemas Zod tipados. La validación está en el controller, pero el servicio es vulnerable a errores de tipo internos.

Funciones Largas

OrderService.createOrder() tiene 175 líneas. Está estructurada con comentarios de sección, pero debería dividirse en submétodos privados.

2.3 Error Handling y Logging

Error Handling: EXCELENTE

Jerarquía de errores bien definida (utils/errors.ts):

```
ApiError (base)
├─ ValidationError (400)
├─ BadRequestError (400)
├─ UnauthorizedError (401)
├─ ForbiddenError (403)
├─ NotFoundError (404)
├─ ConflictError (409)
├─ RateLimitError (429)
├─ InternalError (500)
├─ InsufficientStockError (400) — Domain-specific
└─ ServiceUnavailableError (503)
```

Global error handler en middleware/error.ts:

```
// Maneja correctamente:
☑ ApiError (custom)
☑ ZodError (validation)
☑ Prisma errors (P2002, P2025, P2003)
☑ SyntaxError (invalid JSON)
☑ Fallback genérico (500)
```

Logging: BIEN ESTRUCTURADO

```
// utils/logger.ts
class Logger {
  private readonly minLevel: LogLevel;
  log(level, message, meta): void {
    // JSON structured output
    const entry = { timestamp, level, message, ...meta };
    console.log(JSON.stringify(entry));
  }
  child(defaultMeta): ChildLogger { ... } // Context propagation
}
```

[TIP]

Migrar a Pino para mejor performance en producción (ya sugerido en comentarios del código).

3. AUDITORÍA DE SEGURIDAD (OWASP TOP 10)

3.1 Resumen de Hallazgos

OWASP	Vulnerabilidad	Estado	Notas
A01	Broken Access Control	☑ Mitigado	RBAC con <code>requirePermission()</code>
A02	Cryptographic Failures	☑ Correcto	<code>bcrypt</code> (10 rounds), JWT con <code>secret</code> validado
A03	Injection	☑ Seguro	Prisma ORM, no raw SQL vulnerable
A04	Insecure Design	⚠ Menor	Ver observaciones
A05	Security Misconfiguration	⚠ Menor	CORS warning en producción
A06	Vulnerable Components	☑ Actual	Dependencies actualizadas
A07	Auth Failures	☑ Robusto	Account lockout, timing-safe compare
A08	Data Integrity	☑ Correcto	HMAC validation en webhooks
A09	Logging Failures	☑ Implementado	Structured logging sin PII
A10	SSRF	☑ N/A	No hay fetching de URLs externas dinámicas

3.2 Autenticación y Autorización

Excelente implementación:

```
// auth.service.ts - Validación de JWT_SECRET
if (!JWT_SECRET) {
  throw new Error('CRITICAL: JWT_SECRET not defined. Server cannot start.');
```

Account Lockout:

```
const MAX_FAILED_ATTEMPTS = 5;
const LOCKOUT_DURATION_MINUTES = 15;
// Implementado correctamente con failedLoginAttempts + lockedUntil
```

RBAC Middleware:

```
export const requirePermission = (resource: string, action: string) => {
  return (req, res, next) => {
    if (req.user.role === 'ADMIN') return next(); // Bypass
    const permissions = req.user.permissions;
    if (!permissions[resource]?.includes(action)) {
      return sendError(res, 'AUTH_FORBIDDEN', ...);
    }
    next();
  };
};
```

[CAUTION]

El bypass de ADMIN es correcto para operaciones normales, pero considerar implementar "super-admin confirmation" para acciones destructivas críticas (borrado masivo, etc).

3.3 Inyección SQL

SEGURO. Todo el acceso a datos usa Prisma ORM.

Único uso de raw SQL:

```
// orderNumber.service.ts:66
const result = await tx.$executeRaw`
  INSERT INTO OrderSequence (id, lastNumber) VALUES (1, 1)
  ON DUPLICATE KEY UPDATE lastNumber = lastNumber + 1
`;
```

Esto usa tagged template literals, que Prisma parametriza automáticamente. No vulnerable.

3.4 Webhook Security

```
// hmac.middleware.ts
export function validateHmac(platformCode: string) {
  return async (req, res, next) => {
    const signature = req.headers[headerName];
    const rawBody = req.body as Buffer; // express.raw()
    const adapter = await AdapterFactory.getByPlatformCode(platformCode);
    const isValid = adapter.validateWebhookSignature(signature, rawBody);
    // ...
  };
}
```

Protección correcta:

- ☒ Raw body para HMAC (no parseado)
- ☒ Timing-safe compare en adapter base
- ☒ Logging de intentos fallidos

4. INTEGRACIONES EXTERNAS (Delivery/Payments)

4.1 Arquitectura de Integración

integrations/delivery/

├─ adapters/

├─ AbstractDeliveryAdapter.ts (8.5KB) ─ Base class

├─ AdapterFactory.ts (6KB) ─ Factory pattern

├─ RappiAdapter.ts (14KB) ─ Implementation

├─ webhooks/

├─ hmac.middleware.ts ─ HMAC validation

├─ webhook.controller.ts ─ Async processing

├─ webhook.routes.ts

├─ jobs/ ─ BullMQ workers

└─ sync/ ─ Menu/Stock sync

4.2 Patrones de Resiliencia

Patrón	Implementado	Notas
Queue-based Processing	<input checked="" type="checkbox"/>	BullMQ con retry config
Async Webhook Handling	<input checked="" type="checkbox"/>	Responde 200 OK inmediatamente
Timeout Handling	<input checked="" type="checkbox"/>	Configurable en AdapterConfig
Circuit Breaker	<input type="checkbox"/> ⚠ No explícito	Podría añadirse en AdapterFactory
Retry with Backoff	<input checked="" type="checkbox"/>	Configurado en queue

Manejo de Webhooks:

```
// webhook.controller.ts
async handleWebhook(req, res) {
  const jobData = { platform, eventType, payload, ... };

  // Encolar para procesamiento asíncrono
  const jobId = await queueService.enqueue(
    QUEUE_NAMES.DELIVERY_WEBHOOKS,
    jobData,
    { jobId: `${platformCode}_${externalOrderId}` }
  );

  // Responder inmediatamente (< 100ms)
  return res.status(200).json({ success: true, jobId });
}
```

[!NOTE]

Incluso en caso de error, responde 200 para evitar reintentos de la plataforma. El error se maneja internamente. **Patrón correcto para webhooks.**

5. VEREDICTO FINAL

5.1 Score de Madurez Técnica

Categoría	Puntuación	Peso
Arquitectura	8/10	25%
Calidad de Código	7/10	20%
Seguridad	8.5/10	30%
Integraciones	8/10	15%
Testing Infrastructure	6/10	10%

SCORE FINAL: 7.7/10

5.2 Lista de Refactorizaciones Obligatorias

[!CAUTION]

Los siguientes puntos **deben** abordarse antes de escalar a producción enterprise:

Prioridad	Item	Esfuerzo
● CRÍTICO	Eliminar <code>any</code> en servicios core (<code>product.service.ts</code> , <code>stockMovement.service.ts</code>)	2-4h
● CRÍTICO	Configurar <code>CORS_ORIGINS</code> explícitamente en producción	30min
● ALTO	Añadir tipos explícitos a retornos de frontend services	2-3h
● ALTO	Dividir <code>OrderService.createOrder()</code> en submétodos	2h
● MEDIO	Implementar Circuit Breaker en <code>AdapterFactory</code>	3-4h
● MEDIO	Migrar logger a Pino para producción	1-2h
○ BAJO	Aumentar test coverage (actualmente parece bajo)	Variable

5.3 Respuesta Definitiva

¿Es este codebase digno de producción Enterprise?

SÍ, CONDICIONALMENTE.

El código demuestra:

- ☒ Arquitectura profesional (modular monolith bien estructurado)
- ☒ Patrones de diseño correctos (Adapter, Factory, Facade)
- ☒ Seguridad robusta (RBAC, HMAC, account lockout, JWT validation)
- ☒ Error handling enterprise-grade
- ☒ Infraestructura de queue para resiliencia

No requiere reescritura. Las debilidades identificadas son **deuda técnica menor** que puede resolverse incrementalmente sin riesgo arquitectónico.

El equipo que escribió este código **sabe lo que hace**. La base es sólida y escalable.

ANEXO: Archivos Clave Revisados

Archivo	Líneas	Propósito
schema.prisma	665	Database schema
auth.service.ts	309	Authentication logic
order.service.ts	444	Order facade
error.ts (middleware)	102	Global error handler
auth.ts (middleware)	105	RBAC implementation
webhook.controller.ts	178	Webhook processing
AbstractDeliveryAdapter.ts	295	Integration base class
hmac.middleware.ts	205	Webhook security
queue/index.ts	71	Queue abstraction
logger.ts	154	Structured logging

Fin del Reporte — Clasificación: Técnico-Confidencial