AUDITORÍA DE ARQUITECTURA - FINWRK

Reporte de Cumplimiento contra Principios de Ecosistema

Fecha: 25 de enero de 2026
Auditor: Manus AI
Objetivo: Diagnóstico completo del estado del sistema contra principios arquitectónicos definidos

---

1. MAPEO DE MÓDULOS EXISTENTES

Backend (Routers)

- ✅ `routers.ts` - Router principal (auth, clients, reminders, etc.)
- ✅ `routers_finances.ts` - Dashboard financiero
- ✅ `routers_invoices.ts` - Gestión de facturas
- ✅ `routers_notifications.ts` - Sistema de notificaciones
- ✅ `routers_payments.ts` - Registro de pagos
- ✅ `routers_savings.ts` - Metas de ahorro
- ✅ `routers_transactions.ts` - Transacciones manuales

Base de Datos (Tablas Principales)

- `users` - Usuarios y configuración
- `clients` - Clientes
- `invoices` - Facturas
- `invoice_items` - Items de facturas

- `payments` - Pagos registrados
- `transactions` - Transacciones manuales (ingresos/gastos)
- `savings_goals` - Metas de ahorro
- `notifications` - Notificaciones persistentes
- `reminders` - Recordatorios

Frontend (Páginas)

- `Home.tsx` - Dashboard principal
- `Clients.tsx` - Gestión de clientes
- `Invoices.tsx` - Gestión de facturas
- `Finances.tsx` - Dashboard financiero
- `Savings.tsx` - Metas de ahorro
- `Reminders.tsx` - Recordatorios

---

2. ANÁLISIS POR PRINCIPIO

PRINCIPIO 1: RESPONSABILIDAD CLARA

Estado: ⚠ CUMPLIMIENTO PARCIAL

✅ Módulo: Savings (Ahorros)

Cumplimiento: EXCELENTE

El módulo de ahorros cumple perfectamente con el principio de independencia establecido en el documento arquitectónico. No existe ninguna referencia a `invoices`, `transactions`, `payments` o `finances` en todo el router de savings.

Evidencia:

- Router completamente aislado
- Cada meta tiene su propia moneda (no hereda de `primary_currency`)

- No afecta balances financieros
- No interactúa con otros módulos

⚠ Módulo: Invoices (Facturas)

Cumplimiento: PARCIAL - VIOLACIÓN CRÍTICA DETECTADA

PROBLEMA CRÍTICO: El router de invoices permite cambiar el status de una factura a `paid` directamente mediante el endpoint `updateStatus`, violando el principio de fuente única de verdad.

Evidencia del problema:

```
// routers_invoices.ts línea 284-291
await db
  .update(invoices)
  .set({
    status: newStatus,
    updated_at: new Date(),
  })
  .where(eq(invoices.id, input.id));
```

Violación del documento arquitectónico (Sección 2):

> *"El estado de una factura SOLO puede cambiar a paid cuando el módulo de Pagos registra un pago que cubre el total."*

Riesgo:

- Dos caminos para marcar una factura como pagada:
  1. ✅ Correcto: `payments.register` → actualiza invoice status
  2. ❌ Incorrecto: `invoices.updateStatus` → actualiza directamente
- Posibilidad de marcar facturas como pagadas sin registrar pagos
- Inconsistencia en datos financieros

Recomendación:

- Eliminar la transición `sent → paid` del endpoint `updateStatus`

- Forzar que SOLO `payments.register` pueda cambiar status a `paid`

✅ Módulo: Payments (Pagos)

Cumplimiento: EXCELENTE

El módulo de pagos cumple correctamente con su responsabilidad como fuente de verdad para el estado `paid` de las facturas.

Evidencia:

- Actualiza invoice status transaccionalmente
- Valida montos antes de registrar
- Genera notificaciones correctamente
- No permite editar o eliminar pagos (inmutabilidad)

✅ Módulo: Finances (Dashboard Financiero)

Cumplimiento: EXCELENTE

El dashboard financiero es correctamente de solo lectura y consulta múltiples fuentes.

Evidencia:

- No modifica datos de otros módulos
- Consulta `invoices` con status `paid`
- Consulta `transactions` manuales
- Calcula totales en backend (no en frontend)

Nota menor: El frontend calcula `balance = totalIncome - totalExpenses` localmente, pero esto es aceptable ya que son cálculos derivados simples sin lógica de negocio.

✅ Módulo: Transactions (Transacciones Manuales)

Cumplimiento: EXCELENTE

El módulo de transacciones manuales es independiente y no afecta facturas.

Evidencia:

- Tabla separada con su propio ciclo de vida
- No modifica invoices
- Incluido correctamente en cálculos financieros

PRINCIPIO 2: FUENTES DE VERDAD

Estado: ❌ VIOLACIÓN CRÍTICA

Problema identificado:

El documento arquitectónico establece claramente en la Sección 2:

> ***Invoices (Facturas)***
>
> - *Estado: El estado de una factura SOLO puede cambiar a paid cuando el módulo de Pagos registra un pago que cubre el total.*

Violación actual:

Existen **DOS caminos** para cambiar el status de una factura a `paid`:

1. **Camino correcto:** `payments.register` → actualiza invoice status
2. **Camino incorrecto:** `invoices.updateStatus` → permite transición `sent → paid`

Código problemático:

```
// routers_invoices.ts líneas 272-282
const validTransitions: Record<string, string[]> = {
  draft: ["sent", "cancelled"],
  sent: ["paid", "cancelled"], // ❌ ESTO NO DEBERÍA EXISTIR
  paid: [], // Cannot change from paid
  cancelled: [], // Cannot change from cancelled
};
```

Impacto:

- Riesgo de inconsistencia de datos

- Facturas marcadas como pagadas sin pagos registrados
- Dashboard financiero mostrando ingresos sin transacciones reales
- Violación del principio “el sistema NO crea dinero”

PRINCIPIO 3: FLUJOS PRINCIPALES

Estado: ⚠ CUMPLIMIENTO PARCIAL

Flujo 1: Crear y Enviar Factura

Estado: ✅ CORRECTO

```
Cliente → Factura (draft) → Enviar (sent)
```

Flujo 2: Registrar Pago

Estado: ⚠ RIESGO DE BYPASS

Flujo correcto:

```
Factura (sent) → Pago registrado → Factura (paid/partial)
```

Flujo incorrecto (actualmente posible):

```
Factura (sent) → invoices.updateStatus → Factura (paid) ❌
```

Flujo 3: Dashboard Financiero

Estado: ✅ CORRECTO

```
Facturas (paid) + Transacciones manuales → Totales calculados → Dashboard
```

PRINCIPIO 4: SISTEMA DE EVENTOS

Estado: ❌ NO IMPLEMENTADO

El documento arquitectónico define en la Sección 4 un sistema de eventos para desacoplar módulos:

Eventos esperados:

- `invoice.paid`
- `invoice.overdue`
- `savings.goal_completed`
- `payment.registered`

Estado actual:

- ❌ No existe sistema de eventos
- ❌ No hay EventEmitter o bus de eventos
- ⚠ Se usan llamadas directas a `notificationHelpers`

Evidencia:

```
// routers_payments.ts línea 211-218
await notifyPaymentRegistered(
  ctx.user.id,
  input.invoice_id,
  invoiceData.invoice_number,
  input.amount,
  invoiceData.currency,
  newStatus
);
```

Problema:

- Acoplamiento directo entre módulos
- Difícil agregar nuevos listeners
- No hay registro de eventos históricos

Recomendación: Implementar sistema de eventos según Sección 4 del documento:

```
// Ejemplo esperado:
eventBus.emit('payment.registered', {
  userId,
  invoiceId,
  amount,
  newStatus
});
```

PRINCIPIO 5: NOTIFICACIONES

Estado: ✅ CUMPLIMIENTO CORRECTO

El sistema de notificaciones cumple con los principios definidos:

Evidencia:

- ✅ Notificaciones persistentes en base de datos
- ✅ No hay toasts automáticos
- ✅ No hay popups intrusivos
- ✅ Side panel para visualización
- ✅ Prevención de duplicados
- ✅ Validación de contenido

Código correcto:

```
// helpers/notificationHelpers.ts línea 50-68
if (params.source_id) {
  const [existing] = await db
    .select()
    .from(notifications)
    .where(...)
    .limit(1);

  if (existing) {
    console.log(`DISCARDED: Duplicate notification`);
    return false;
  }
}
```

PRINCIPIO 11: ACOPLAMIENTO

Estado: ⚠ ACOPLAMIENTO MODERADO

Acoplamientos Identificados:

1. Payments → Invoices (ACEPTABLE)

```
// routers_payments.ts línea 200-206
await db
  .update(invoices)
  .set({
    status: newStatus,
    updated_at: new Date(),
  })
  .where(eq(invoices.id, input.invoice_id));
```

Justificación: Este acoplamiento es necesario y está documentado en el flujo principal.

2. Payments → NotificationHelpers (MEJORABLE)

```
await notifyPaymentRegistered(...);
```

Problema: Llamada directa en lugar de usar eventos.

3. Invoices → NotificationHelpers (MEJORABLE)

```
// routers_invoices.ts línea 296-304
if (newStatus === 'paid') {
  const { notifyInvoicePaid } = await
import('../helpers/notificationHelpers');
  await notifyInvoicePaid(...);
}
```

Problema: Invoices no debería notificar pagos, ya que no debería poder marcar facturas como pagadas.

---

3. ANÁLISIS DE RIESGOS TÉCNICOS

RIESGO CRÍTICO 1: Doble Fuente de Verdad para Invoice Status

Severidad: 🔴 CRÍTICA
Probabilidad: ALTA

Descripción: Dos endpoints pueden cambiar el status de una factura a `paid`:

- `payments.register` (correcto)
- `invoices.updateStatus` (incorrecto)

Escenario de falla:

1. Usuario marca factura como “paid” desde UI de facturas
2. No se registra pago
3. Dashboard muestra ingreso fantasma
4. Totales financieros incorrectos

Impacto:

- Datos financieros inconsistentes
- Reportes incorrectos

- Pérdida de confianza en el sistema

RIESGO ALTO 2: Ausencia de Sistema de Eventos

Severidad: 🟡 ALTA
Probabilidad: MEDIA

Descripción: Sin un sistema de eventos centralizado, es difícil:

- Agregar nuevos listeners
- Auditar qué pasó en el sistema
- Desacoplar módulos

Escenario de falla:

1. Se necesita agregar nueva funcionalidad que reaccione a pagos
2. Hay que modificar `routers_payments.ts` directamente
3. Riesgo de romper funcionalidad existente

Impacto:

- Código frágil
- Difícil mantenimiento
- Regresiones frecuentes

RIESGO MEDIO 3: Cálculos en Frontend

Severidad: 🟢 MEDIA
Probabilidad: BAJA

Descripción: El frontend calcula `balance = totalIncome - totalExpenses` localmente.

Justificación de baja severidad:

- Es un cálculo simple y derivado
- No modifica datos

- Los valores base vienen del backend

Recomendación:

- Mover el cálculo al backend para consistencia
- Retornar `balance` directamente en `getSummary`

---

4. ENDPOINTS QUE NO DEBERÍAN EXISTIR

❌ `invoices.updateStatus` con transición `sent → paid`

Ubicación: `routers_invoices.ts` línea 246-312

Razón: Viola el principio de fuente única de verdad. Solo `payments.register` debería poder cambiar status a `paid`.

Acción recomendada:

```
const validTransitions: Record<string, string[]> = {
  draft: ["sent", "cancelled"],
  sent: ["cancelled"], // ❌ ELIMINAR "paid"
  paid: [],
  cancelled: [],
};
```

---

5. LÓGICA RESIDUAL Y LISTENERS NO DESEADOS

✅ No se detectó lógica residual significativa

Búsqueda realizada:

- ✅ No hay listeners de eventos no documentados
- ✅ No hay código comentado con lógica activa
- ✅ No hay módulos deshabilitados con efectos secundarios

Nota: El comentario en `routers.ts` línea 16-23 indica que hubo una simplificación previa, pero todos los módulos están correctamente habilitados y funcionando.

---

6. ESTADOS Y DATOS QUE NO DEBERÍAN EXISTIR

✅ Schema de base de datos correcto

Análisis de `schema.ts`:

- ✅ Tabla `invoices` con estados correctos: draft, sent, paid, partial, cancelled
- ✅ Tabla `payments` inmutable (no tiene `deleted_at` o `status`)
- ✅ Tabla `transactions` con status: active, voided
- ✅ Tabla `savings_goals` independiente con su propia moneda

No se detectaron:

- ❌ Campos redundantes
- ❌ Estados inválidos
- ❌ Datos duplicados

---

7. RESUMEN EJECUTIVO

CUMPLIMIENTO GENERAL: ⚠ 75% - BUENO CON VIOLACIÓN CRÍTICA

Principio	Estado	Cumplimiento
1. Responsabilidad Clara	⚠ Parcial	80%
2. Fuentes de Verdad	❌ Violación	50%
3. Flujos Principales	⚠ Parcial	85%
4. Sistema de Eventos	❌ No implementado	0%
5. Notificaciones	✅ Correcto	100%
11. Acoplamiento	⚠ Moderado	70%

MÓDULOS POR CUMPLIMIENTO

Módulo	Cumplimiento	Observaciones
Savings	✅ 100%	Perfectamente aislado
Payments	✅ 95%	Correcto, mejorable con eventos
Finances	✅ 95%	Solo lectura, correcto
Transactions	✅ 100%	Independiente, correcto
Invoices	❌ 60%	Violación crítica en updateStatus
Notifications	✅ 100%	Implementación correcta

PRIORIDADES DE CORRECCIÓN

🔴 CRÍTICO (Implementar INMEDIATAMENTE)

1. **Eliminar transición `sent → paid` en `invoices.updateStatus`**
   - Archivo: `routers_invoices.ts` línea 274

- Cambio: Remover "paid" de `validTransitions.sent`
- Impacto: Elimina riesgo de inconsistencia de datos

🟡 **ALTO (Implementar en siguiente sprint)**

1. **Implementar sistema de eventos**
   - Crear `EventBus` según Sección 4 del documento
   - Migrar llamadas directas a notificaciones
   - Agregar eventos: `invoice.paid`, `payment.registered`, etc.
2. **Mover cálculo de balance al backend**
   - Archivo: `routers_finances.ts`
   - Agregar campo `balance` en respuesta de `getSummary`
   - Eliminar cálculo en `Finances.tsx` línea 196

🟢 **MEDIO (Backlog)**

1. **Documentar acoplamientos necesarios**
   - Crear diagrama de dependencias
   - Documentar por qué `payments` actualiza `invoices`
2. **Agregar tests de integración**
   - Test: No se puede marcar factura como paid sin pago
   - Test: Pago actualiza invoice status correctamente

CONCLUSIÓN

El sistema tiene una **arquitectura sólida en general**, con módulos bien separados y responsabilidades claras. Sin embargo, existe una **violación crítica** en el módulo de Invoices que permite bypass del flujo de pagos.

La implementación del sistema de eventos (Sección 4 del documento) es la mejora más importante después de corregir la violación crítica, ya que permitirá:

- Desacoplar módulos completamente
- Facilitar auditoría y debugging
- Agregar nuevas funcionalidades sin modificar código existente

Recomendación final: Corregir la violación crítica antes de implementar las validaciones del ecosistema (Sección 12), ya que las validaciones asumirán que solo existe una fuente de verdad para cada dato.

Fin del Reporte de Auditoría