Factum — Entregable 2: Parcial WS

Nicolas José Machado Martinez

Docente

Edwar Alfonso Villamizar Vallejo

Unidades tegnologicas de Santander

Web Services

2025

1. Contexto ampliado del caso

Factum es una aplicación web de facturación pensada para pequeñas y medianas empresas (pymes) que necesitan:

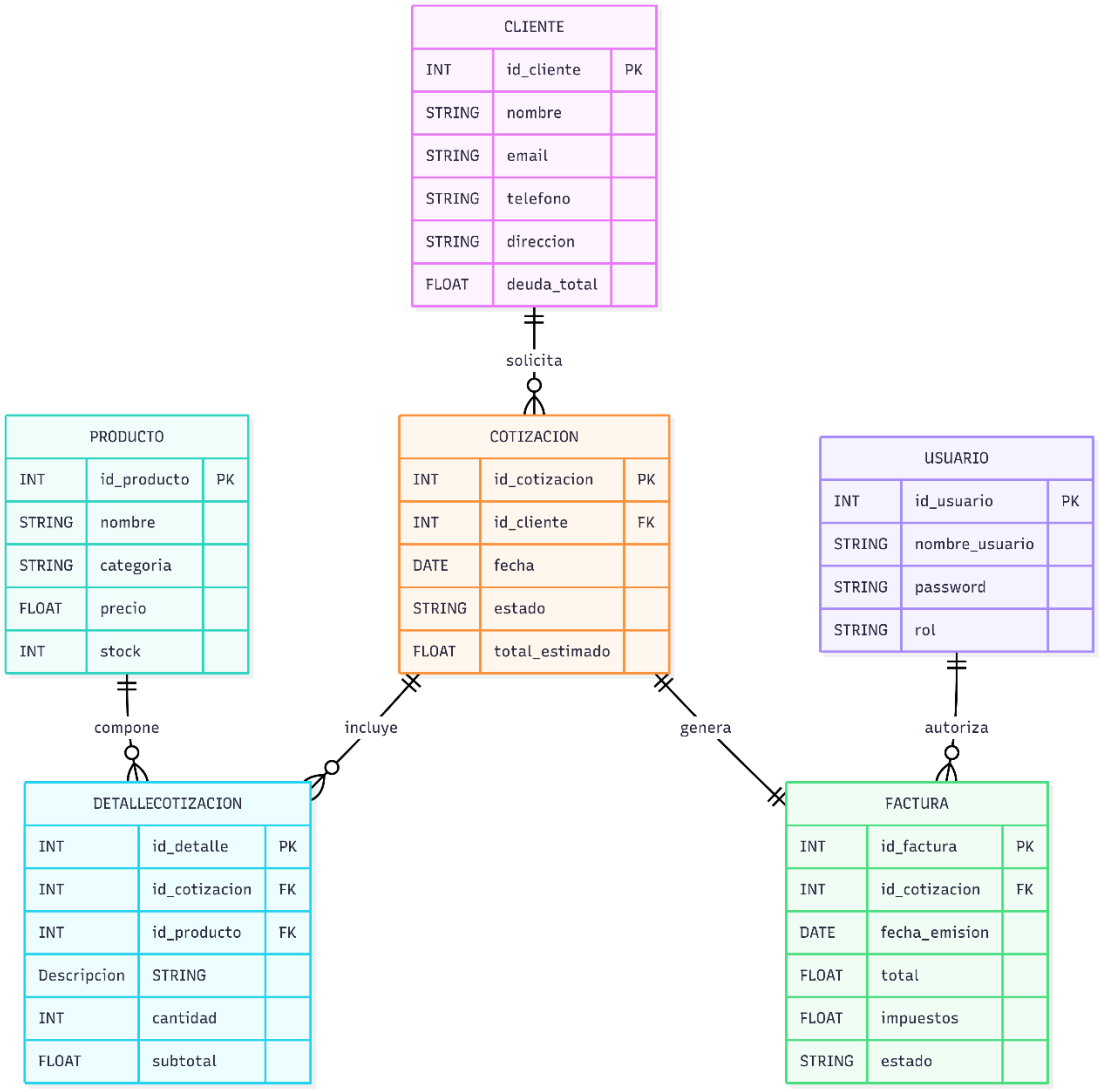
- Gestionar clientes, productos/servicios y facturas.
- Emitir facturas electrónicas o en PDF.
- Automatizar cálculos de impuestos y totales.
- Llevar un control básico de inventario mínimo.
- Mantener registro de deudas de clientes y estado de pagos.
- Usar APIs REST de forma interna para organizar procesos contables.

👉 Problema de negocio que resuelve:

- Evita la gestión manual de facturas (hojas de cálculo, papeles).
- Centraliza información de clientes y productos.
- Calcula impuestos y totales automáticamente.
- Permite a los dueños/empleados manejar la contabilidad sin errores.

Importante: Los clientes externos no usan Factum directamente; solo reciben las cotizaciones o facturas en PDF o por medios externos.

2. Entidades y Relaciones



3. Requisitos no funcionales

- Seguridad → uso de JWT para autenticar usuarios internos.
- Escalabilidad → arquitectura modular.
- Rendimiento → respuesta < 2 segundos.
- Disponibilidad → 99,5% uptime.
- Usabilidad → interfaz clara para empleados (web interna).
- Interoperabilidad → conexión futura con sistemas contables o de la DIAN.

4. Diseño de capas

Dominio:

- Cliente: con deuda_total que se actualiza cada vez que se genera una factura.
- Factura: solo se crea si la cotización está aprobada.
- Cotización: estado = pendiente, aprobada, rechazada.

Repository:

- Persistencia en MySQL.
- Manejo de tablas relacionales para clientes, cotizaciones, facturas, usuarios.

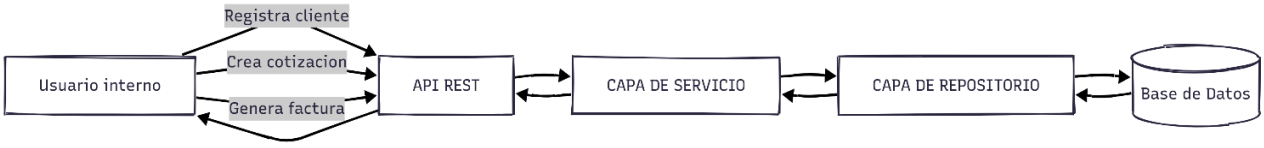
Service (reglas de negocio):

- Una factura solo se crea a partir de una cotización aprobada.
- Solo usuarios internos con rol autorizado pueden facturar.
- La deuda_total del cliente se actualiza automáticamente al generar facturas en estado pendiente.

API (endpoints internos):

- POST /clientes → crear cliente.
- GET /clientes/{id} → consultar cliente (incluye deuda_total).
- POST /cotizaciones → crear cotización.
- PUT /cotizaciones/{id}/aprobar → aprobar cotización.
- POST /facturas/generar → generar factura desde cotización.

4. Diagrama de Flujos de Datos



Explicación del flujo

1. Usuario interno
   - Representa al empleado de la empresa (admin, contador o vendedor).

- Es la persona que usa el sistema Factum.
- Puede:
  - Registrar un cliente.
  - Crear una cotización.
  - Generar una factura.

2. API REST

- Es la puerta de entrada al sistema.
- Cuando el usuario hace clic en la interfaz (ejemplo: “crear cliente”), esa acción se traduce en una llamada a la API (ejemplo: POST /clientes).
- La API no guarda datos, solo los recibe y los envía a la siguiente capa.

3. Capa de Servicios

- Aquí viven las reglas de negocio.
- Ejemplos:
  - Verificar que el cliente no exista antes de crearlo.
  - Calcular el total de la cotización.
  - No permitir generar factura si la cotización no está aprobada.
  - Actualizar el campo deuda_total de un cliente cuando se emite una factura pendiente.

4. Capa de Repositorio

- Es la encargada de hablar con la Base de Datos.
- Recibe órdenes de la capa Service y ejecuta: insertar, actualizar, consultar, eliminar.
- Ejemplo: “guardar el nuevo cliente en la tabla clientes”.

5. Base de Datos (DB)

- Aquí se almacena todo: clientes, productos, cotizaciones, facturas, usuarios.
- Es el corazón del sistema porque guarda la información de manera persistente.

Flujo de retorno

- La DB responde al Repository → el Repository responde al Service → el Service responde al API → y finalmente el API le devuelve una respuesta JSON al Usuario.

- Ejemplo:

    - El usuario crea un cliente.
    - La API pasa los datos al Service.
    - El Service valida y pasa al Repository.
    - El Repository inserta en la DB.
    - La DB confirma la inserción.
    - La confirmación viaja de regreso hasta el Usuario en formato JSON con mensaje “Cliente creado con éxito”.

6. Esquema JSON estándar

```
{
  "success": true,
  "message": "Operacion realizada correctamente",
  "data": {},
  "error_code": null,
  "details": {}
}
```

7. Casos de uso con ejemplos JSON

7.1 Creación de cliente

Request

```
POST /clientes
{
  "nombre": "Empresa XYZ",
  "email": "contacto@xyz.com",
  "telefono": "3001234567",
  "direccion": "Calle 10 #20-30"
}
```

Response

```
{
  "success": true,
  "message": "Cliente creado con exito",
  "data": { "id_cliente": 101, "deuda_total": 0 },
  "error_code": null,
  "details": {}
}
```

7.2 Consulta de cliente con deuda

Request

```
GET /clientes/101
```

Response

```
{
  "success": true,
  "message": "Informacion del cliente",
  "data": {
    "id_cliente": 101,
    "nombre": "Empresa XYZ",
    "email": "contacto@xyz.com",
    "deuda_total": 350000
  },
  "error_code": null,
  "details": {}
}
```

7.3 Error de validacion

Request

```
POST /clientes
{
  "nombre": "Empresa ABC",
  "email": "correo_invalido",
```

```
  "telefono": "300abc"
}
```


Response


```
{
  "success": false,
  "message": "Error de validacion",
  "data": {},
  "error_code": "ERR_VAL_001",
  "details": {
    "email": "Formato invalido",
    "telefono": "Debe ser numerico"
  }
}
```


7.4 Error por interoperabilidad (conexion fallida DIAN)

Response


```
{
  "success": false,
  "message": "No se pudo conectar con el servicio de facturacion electronica",
  "data": {},
  "error_code": "ERR_CONN_503",
  "details": {
    "timestamp": "2025-09-14T10:00:00Z"
  }
}
```

8. Justificacion REST vs SOAP

Criterio	REST	SOAP
Formato de datos	Usa **JSON ligero** (menos peso, fácil de procesar).	Usa **XML** (más pesado y verboso).
Rendimiento	Más eficiente: menor latencia y menos consumo de red gracias a JSON y HTTP estándar.	Menos eficiente: parseo de XML, validaciones y sobrecarga de WS-* aumentan la latencia.
Protocolos	Basado en **HTTP/HTTPS** con métodos estándar (GET, POST, PUT, DELETE).	Puede usar varios protocolos (HTTP, SMTP), pero con mayor complejidad.
Seguridad	Soporta HTTPS, autenticación con **JWT**, OAuth, etc.	Incluye estándares avanzados como **WS-Security** (firmas digitales, cifrado a nivel de mensaje).
Interoperabilidad	Ideal para **web y apps móviles**, microservicios y sistemas modernos.	Muy usado en entornos **legacy corporativos y bancarios**.
Simplicidad	Fácil de implementar e integrar.	Más complejo: requiere WSDL, contratos estrictos y más configuración.

.