Bank Core Project — Documentación

Visión, arquitectura, despliegue y batch COBOL

Documentación del repositorio (demo portafolio)

- 1 Índice de documentación — Bank Core Project
  - 1.1 Empieza aquí
  - 1.2 Profundizar en el batch y los datos
  - 1.3 Versión unificada (HTML + PDF)
  - 1.4 README raíz
- 2 Visión del software y arquitectura
  - 2.1 1. Enfoque no técnico
    - 2.1.1 ¿Qué es?
    - 2.1.2 ¿Para qué sirve?
    - 2.1.3 ¿Qué puede hacer un usuario típico?
    - 2.1.4 Limitaciones importantes (honestidad técnica)
  - 2.2 2. Enfoque técnico
    - 2.2.1 2.1 Stack y repositorio
    - 2.2.2 2.2 Flujo principal (`POST /transfer`)
    - 2.2.3 2.3 Módulos NestJS (mapa mental)
    - 2.2.4 2.4 Frontend
    - 2.2.5 2.5 Datos y separación de concerns
    - 2.2.6 2.6 Seguridad (estado demo vs producción)
  - 2.3 3. Lecturas relacionadas
- 3 Guía para desarrolladores: entorno local y despliegue
  - 3.1 1. Prerrequisitos
    - 3.1.1 1.1 Local (fuera de Docker)
    - 3.1.2 1.2 Solo Docker
  - 3.2 2. Obtener el código e instalar dependencias
  - 3.3 3. Primera configuración (obligatoria antes de usar auth)
  - 3.4 4. Ejecutar en local (sin Docker)
    - 3.4.1 4.1 Una terminal (recomendado)
    - 3.4.2 4.2 Dos terminales
    - 3.4.3 4.3 URLs
    - 3.4.4 4.4 Variables de entorno útiles (backend)
    - 3.4.5 4.5 Frontend en desarrollo
  - 3.5 5. Docker Compose (desarrollo con hot reload)
  - 3.6 6. Build de producción (orientación)
    - 3.6.1 6.1 Backend

        - 3.6.2 6.2 Frontend
        - 3.6.3 6.3 Reverse proxy
    - 3.7 7. Checklist rápido antes de compartir el entorno
    - 3.8 8. Manual unificado (HTML / PDF)
    - 3.9 9. Documentación adicional en el repo
    - 3.10 10. Problemas frecuentes
- 4 Flujo del «core bancario» híbrido
    - 4.1 Flujo de información
    - 4.2 Corte / cierre diario
    - 4.3 Interfaz web (`frontend/`)
    - 4.4 Docker Compose (desarrollo con recarga)
    - 4.5 Cómo se simula el mainframe
    - 4.6 Cómo validar que funciona
        - 4.6.1 Prerrequisitos
        - 4.6.2 Arranque del backend
        - 4.6.3 Casos de ejemplo (curl)
        - 4.6.4 Restaurar datos semilla
        - 4.6.5 Variable opcional `BANK_CORE_ROOT`
        - 4.6.6 Tests automatizados de validación
- 5 COBOL y JCL: cómo encajan en la arquitectura
    - 5.1 Rol del JCL
    - 5.2 Rol del programa COBOL (`BANKBATCH.cbl`)
    - 5.3 Qué hace `run-job.sh` como sustituto local
    - 5.4 Bitácora operacional
- 6 Persistencia y bases de datos
    - 6.1 Qué está donde
    - 6.2 Perfil local y perfil productivo
    - 6.3 Cuándo usar PostgreSQL (recomendación profesional)
    - 6.4 Docker
    - 6.5 Tabla actual
    - 6.6 Próximas mejoras

Compilación automática: 2026-05-04T15:13:22-06:00. La fuente editable está en los archivos `.md` bajo `docs/`.

1 Índice de documentación — Bank Core Project

Bienvenida al material de **referencia** del repo: visión del producto, cómo levantarlo y detalle del batch legacy.

1.1 Empieza aquí

Documento	Contenido
VISION_Y_ARQUITECTURA.md	Qué hace el software (lectura **no técnica** y **técnica**), piezas del repo y roles de cada módulo.
GUIA_DESPLIEGUE_DESARROLLADORES.md	Instalación local, Docker Compose, variables de entorno, build de producción y problemas frecuentes.

1.2 Profundizar en el batch y los datos

Documento	Contenido
FLUJO.md	Secuencia NestJS ↔ archivos ↔ COBOL, Docker, ejemplos `curl`, cierre diario.
COBOL_Y_JCL.md	Relación entre `BANKBATCH.cbl` y `TRANJOB.jcl`.
PERSISTENCIA.md	SQLite (`bank-core.sqlite`), archivos `.DAT`, opción Postgres.

1.3 Versión unificada (HTML + PDF)

En **`docs/pdf/`** tienes copias compiladas del manual:

Archivo	Descripción
Bank-Core-Project-Documentacion.pdf	Todo el contenido de las guías en un solo PDF.
Bank-Core-Project-Documentacion.html	Misma unión en HTML (útil si imprimes desde el navegador).

Para **regenerar** después de editar los `.md` (requiere **pandoc** y **Google Chrome**, **Chromium** o **chromium-browser** en PATH; opcional `CHROME_BIN`):

```
bash scripts/build-docs-pdf.sh
```

Desde la raíz del repo también puedes usar `npm run docs:pdf`. Detalle en pdf/README.md.

1.4 README raíz

La entrada rápida al repo (comandos `npm run dev`, tabla resumida de API) sigue en **README.md**.

2 Visión del software y arquitectura

Este documento resume **qué problema ilustra el proyecto**, **qué piezas lo componen** y **cómo encajan**, en dos niveles: lectura **no técnica** y **técnica**. Para el detalle del flujo batch paso a paso, véase también FLUJO.md.

2.1 1. Enfoque no técnico

2.1.1 ¿Qué es?

Es una **demostración educativa** de cómo un banco (o sistema financiero) puede combinar **canales modernos** (una aplicación web y una API) con **procesos "clásicos"** que históricamente viven en mainframes: programas que leen y escriben **archivos de datos** en lugar de hablar directamente con bases relacionales en cada movimiento.

Piensa en una **oficina híbrida**: la ventanilla nueva habla con un servidor actual (NestJS), pero las **reglas contables críticas** siguen ejecutándose en un **lote** (`batch`) escrito en COBOL, igual que en muchos cores reales.

2.1.2 ¿Para qué sirve?

- **Aprender y mostrar en portafolio** la integración entre mundos distintos (HTTP ↔ archivos ↔ COBOL).
- **Probar transferencias** (débitos y créditos) sobre un maestro de cuentas simulado.
- **Dejar rastro** de lo que pasó (historial y exportaciones) para analizar volumetría, errores o patrones (con matices de "demo", no es un sistema AML/regulatorio real).

2.1.3 ¿Qué puede hacer un usuario típico?

Acción	**Idea simple**
Registrarse / iniciar sesión	Acceso con correo y contraseña; la sesión va en cookies seguras gestionadas por Better Auth.
Ver cuentas y saldos	Refleja el archivo maestro que actualiza el programa COBOL.
Hacer una transferencia	La web pide el movimiento; el sistema lo manda al "batch"; el COBOL decide si es válido y actualiza saldos.
Ver historial	Lista de intentos y resultados guardados para consulta.
Descargar CSV	Exportar datos para Excel u otras herramientas.
Subir un CSV	Varias líneas de movimientos procesadas una tras otra con la misma lógica que una transferencia suelta.
Ver gráficos	Resúmenes sobre el historial (rechazos, volumen, actividad por día): útil como **ejemplo** de tablero de riesgo u operaciones.
Cierre diario (demo)	Genera un archivo-resumen del día para ilustrar cortes operativos.

2.1.4 Limitaciones importantes (honestidad técnica)

- No es un core bancario productivo: **no cumple** normativa, escalabilidad ni seguridad de un entorno real sin trabajo adicional.
- Los datos sensibles, HA, auditoría formal y modelos de fraude **no** están cubiertos como en producción.

2.2 2. Enfoque técnico

2.2.1 2.1 Stack y repositorio

Carpeta / artefacto	**Rol**
`backend-nest/`	API **NestJS** (HTTP, validación, orquestación, persistencia auxiliar, auth).
`frontend/`	**SPA React + Vite + Tailwind**: UI, proxy en desarrollo, cliente Better Auth.
`mainframe/`	**Datos y programa batch**: `CUENTAS.DAT`, `TRANS.DAT`, `REPORTE.DAT`, `HISTORIAL.DAT`, `CIERRE_*.DAT`, fuente `BANKBATCH.cbl`, referencia `TRANJOB.jcl`.
`scripts/`	`run-job.sh` — simula JOB (compilar/ejecutar COBOL); `dev-local.sh` — arranca API + Vite.
`data/`	**`bank-core.sqlite`** — bitácora TypeORM (`transferencias`, etc.). **`auth.sqlite`** — usuarios/sesiones Better Auth (separado).

Carpeta / artefacto	Rol
`docker/`	Imagen API (Node + GnuCOBOL + toolchain) y entrypoints para Compose.

2.2.2 2.2 Flujo principal (`POST /transfer`)

1. Cliente autenticado llama al endpoint (guard global excepto rutas públicas de Better Auth y salvo tests).
2. **DTO + ValidationPipe** validan entrada.
3. **TransferService** escribe `mainframe/TRANS.DAT`, ejecuta `scripts/run-job.sh`.
4. El script compila si hace falta y ejecuta **GnuCOBOL** contra `BANKBATCH.cbl`; lectura/escritura de `.DAT`.
5. Nest lee `REPORTE.DAT`, mapea códigos a respuesta JSON.
6. Se persiste la operación en **SQLite negocio** y append opcional a **`HISTORIAL.DAT`** (detalle en PERSISTENCIA.md).

2.2.3 2.3 Módulos NestJS (mapa mental)

Módulo	Responsabilidad
DatabaseModule	TypeORM + entidades (p. ej. `TransferenciaEntity`).
Better Auth (`AuthModule`)	Sesiones email/contraseña, rate limit, `trustedOrigins`; BD dedicada `auth.sqlite`.
TransferModule	Tubera principal batch + uso de `TransferService`.
CuentasModule	Lectura/parsing de `CUENTAS.DAT` → JSON.
HistorialModule	Consulta bitácora SQLite; filtros; sincronización hacia archivo si aplica.
ReportsModule	CSV UTF-8 BOM (`cuentas`, `historial`, `combinado`).
BatchModule	`POST /batch/csv` — parse CSV, una llamada equivalente a transferencia por fila.
AnalyticsModule	Agregaciones sobre historial para la UI (demo analítica).
CierreModule	Generación/descarga de archivos de corte diario.

2.2.4 2.4 Frontend

- **Proxy Vite**: `/api/auth` sin rewrite hacia el backend (Better Auth montado en `/api/auth`); resto `/api/*` → rutas Nest sin prefijo `/api`.
- **`credentials: 'include'`** en llamadas API para cookies de sesión.
- Componentes de tablero: transferencias, tablas, exportación, carga CSV, gráficos (**Recharts**), cierre.

2.2.5 2.5 Datos y separación de concerns

- **Ledger “legacy”**: `CUENTAS.DAT` gobernado por COBOL.
- **Trazabilidad canal HTTP**: SQLite `bank-core.sqlite` + opcional `HISTORIAL.DAT`.

- **Identidad**: `auth.sqlite` — no mezclar con tablas de negocio.

2.2.6 2.6 Seguridad (estado demo vs producción)

En **desarrollo** el proyecto privilegia ergonomía (secretos por defecto, SQLite local). En **producción** deben definirse como mínimo:

- `BETTER_AUTH_SECRET` (alta entropía, ≥32 caracteres).
- `BETTER_AUTH_URL` coherente con la URL pública HTTPS del API.
- `FRONTEND_ORIGINS` acotado a los orígenes reales del SPA.
- HTTPS terminado en proxy; cookies `Secure` en entornos HTTPS.

2.3 3. Lecturas relacionadas

- README.md — índice maestro de esta carpeta y enlaces al PDF/HTML unificado.
- FLUJO.md — secuencia batch, Docker, validación con `curl`.
- COBOL_Y_JCL.md — programa y JOB de referencia.
- PERSISTENCIA.md — SQLite vs archivos, Postgres opcional.
- GUIA_DESPLIEGUE_DESARROLLADORES.md — cómo levantar y desplegar.
- Manual compilado: pdf/Bank-Core-Project-Documentacion.pdf (regenerar con `npm run docs:pdf`).

3 Guía para desarrolladores: entorno local y despliegue

Objetivo: que cualquier desarrollador pueda **instalar**, **ejecutar** y **publicar** el proyecto con pasos claros y variables documentadas.

3.1 1. Prerrequisitos

3.1.1 1.1 Local (fuera de Docker)

Requisito	**Uso**
Node.js 18+ (recomendado 22 LTS)	NestJS y Vite.
npm	Gestión de dependencias.
Bash	Scripts `run-job.sh`, `dev-local.sh`.
GnuCOBOL (`cobc`)	Compilar y ejecutar `BANKBATCH.cbl`.
Toolchain nativa (`python3`, `make`, `g++` en Linux)	Compilar el addon **`better-sqlite3`**.

3.1.2 1.2 Solo Docker

- **Docker** + **Docker Compose** v2.
- En la imagen del API ya vienen GnuCOBOL y la toolchain; no hace falta instalar `cobc` en el host.

3.2 2. Obtener el código e instalar dependencias

```
git clone <url-del-repo>
cd bank-core-project
```

Instalar dependencias de backend y frontend (el `package.json` de la raíz solo agrupa scripts):

```
npm run install:all
```

Equivalente manual:

```
(cd backend-nest && npm install)
(cd frontend && npm install)
```

3.3 3. Primera configuración (obligatoria antes de usar auth)

Better Auth guarda usuarios en **`data/auth.sqlite`**. Crea el esquema una vez:

```
cd backend-nest
npm run auth:migrate
```

Si necesitas **eliminar un usuario** para reutilizar un correo:

```
cd backend-nest
npm run auth:delete-user -- correo@ejemplo.com
# o vaciar toda la auth en desarrollo:
npm run auth:delete-user -- --all
```

3.4 4. Ejecutar en local (sin Docker)

3.4.1 4.1 Una terminal (recomendado)

Desde la **raíz del repo**:

```
npm run dev
```

Esto ejecuta `scripts/dev-local.sh`: levanta Nest en **3000**, espera el puerto y arranca Vite en **5173**. Al cerrar (Ctrl+C) intenta detener el proceso del API.

3.4.2 4.2 Dos terminales

```
# Terminal 1
cd backend-nest && npm run start:dev

# Terminal 2
cd frontend && npm run dev
```

Importante: si solo corres Vite sin el API en `:3000`, las peticiones al proxy fallarán (`ECONNREFUSED`).

3.4.3 4.3 URLs

- UI: `http://localhost:5173`
- API: `http://localhost:3000`

3.4.4 4.4 Variables de entorno útiles (backend)

Defínelas en el shell o en un `.env` cargado por Nest si lo configuráis (el proyecto puede leer `process.env` estándar).

Variable	Descripción
`PORT`	Puerto HTTP (default **3000**).
`BANK_CORE_ROOT`	Raíz del repo si el `cwd` del proceso no es `backend-nest/` (Docker usa `/workspace`).
`FRONTEND_ORIGINS`	Orígenes CORS separados por comas (default incluye `localhost:5173`).
`BETTER_AUTH_SECRET`	Secreto firmado sesiones (≥**32 caracteres**). En `NODE_ENV=production` es **obligatorio**.
`BETTER_AUTH_URL`	URL pública base del API (ej. `http://localhost:3000` en local).
`SQLITE_DB_PATH`	Ruta alternativa a `data/bank-core.sqlite`.
`E2E_NO_AUTH`	`true` solo para tests e2e (desactiva guard global simulado).

3.4.5 4.5 Frontend en desarrollo

No suele hacer falta `.env`: el proxy está en `frontend/vite.config.ts`.

Opcional: `API_PROXY_TARGET` si el API no está en `http://localhost:3000` (por ejemplo otro host/puerto).

Plantilla para **build estático** / **preview**: copiar `frontend/.env.example` → `frontend/.env` y ajustar `VITE_API_URL`.

3.5 5. Docker Compose (desarrollo con hot reload)

Desde la raíz:

```
docker compose up --build
```

- Servicio **`api`**: `npm install` + `npm run start:dev`; monta `backend-nest/`, `mainframe/`, `scripts/`, `data/`.
- Servicio **`web`**: `npm install` + `vite dev --host 0.0.0.0`; variable **`API_PROXY_TARGET=http://api:3000`**.

Volúmenes `*_node_modules` evitan pisar dependencias del contenedor.

Entorno típico expuesto en `docker-compose.yml`:

- `BANK_CORE_ROOT=/workspace`
- `BETTER_AUTH_URL=http://localhost:3000` (coherente con el navegador en el host)

Tras el primer arranque, ejecutar migración de auth **dentro del contenedor** si hace falta:

```
docker compose exec api bash -lc 'cd /workspace/backend-nest && npm run auth:migrate'
```

3.6 6. Build de producción (orientación)

Este repo está pensado como **demo/portafolio**. Un despliegue mínimo razonable:

3.6.1 6.1 Backend

```
cd backend-nest
npm ci
npm run build
npm run start:prod
```

Requisitos en el servidor: **Node**, **GnuCOBOL**, **Bash**, toolchain para `better-sqlite3` si compiláis en máquina (o usáis binarios prebuild compatibles).

Variables **imprescindibles** en producción:

- `NODE_ENV=production`
- `BETTER_AUTH_SECRET` (fuerte, rotación según política)
- `BETTER_AUTH_URL` (URL HTTPS pública del API)
- `FRONTEND_ORIGINS` (solo dominios del SPA)

Persistencia: montar volumen o disco para **`data/`** (ambos SQLite) y **`mainframe/`** (`.DAT` y binarios compilados).

3.6.2 6.2 Frontend

```
cd frontend
cp .env.example .env   # editar VITE_API_URL → URL base del API en producción
npm ci
npm run build
```

Servir `frontend/dist/` con nginx, S3+CloudFront, etc.

CORS y cookies: el SPA debe estar en un origen declarado en `FRONTEND_ORIGINS`; las peticiones deben usar `credentials: 'include'` (ya en `src/lib/api.ts`). Si el front y el API están en **dominios distintos**, revisá políticas SameSite y la configuración del cliente Better Auth (puede requerir `baseURL` explícito hacia el API — consultad la documentación de Better Auth para SPA + API cross-site).

3.6.3 6.3 Reverse proxy

Buenas prácticas: TLS en nginx/Caddy/Traefik, cabeceras de seguridad, límites de tamaño para uploads (`/batch/csv`), y timeouts acordes a la ejecución del batch COBOL.

3.7 7. Checklist rápido antes de compartir el entorno

- [ ] `npm run install:all` sin errores.
- [ ] `cobc --version` en local (o Docker construido correctamente).
- [ ] `npm run auth:migrate` ejecutado al menos una vez.
- [ ] `npm run dev` o Compose: UI abre y permite registro/login.

- [ ] Una transferencia de prueba actualiza cuentas o devuelve error coherente.

3.8 8. Manual unificado (HTML / PDF)

Para empaquetar **todos** los `.md` de `docs/` en un solo archivo:

```
npm run docs:pdf
```

(o `bash scripts/build-docs-pdf.sh`). Requiere **Pandoc** y **Chrome** o **Chromium** en PATH (variable opcional `CHROME_BIN`). Salida en `docs/pdf/` — véase pdf/README.md.

3.9 9. Documentación adicional en el repo

Documento	**Contenido**
README.md	Resumen, tabla de rutas API, requisitos.
docs/README.md	Índice maestro de la carpeta `docs/`.
VISION_Y_ARQUITECTURA.md	Qué es el software (no técnico + técnico).
FLUJO.md	Diagrama y pasos del batch.
COBOL_Y_JCL.md	Programa y JOB.
PERSISTENCIA.md	SQLite y archivos.

3.10 10. Problemas frecuentes

Síntoma	**Qué revisar**
`ECONNREFUSED` en el navegador	Que Nest esté en marcha en el puerto esperado y que el proxy apunte bien (`API_PROXY_TARGET` en Docker).
Error al compilar `better-sqlite3`	Toolchain C++ / usar Node compatible con los prebuilds o compilar en Docker.
Login no mantiene sesión	Mismo sitio/proxy, `credentials: 'include'`, `FRONTEND_ORIGINS` y cookies bloqueadas por el navegador.
Better Auth 500 en producción	`BETTER_AUTH_SECRET` y `BETTER_AUTH_URL` definidos y HTTPS coherente.
Batch no corre	Permisos de `scripts/run-job.sh`, presencia de `cobc`, rutas `BANK_CORE_ROOT`.

4 Flujo del «core bancario» híbrido

Este documento resume cómo circula la información entre NestJS y el lote COBOL, cómo se ejecuta localmente el flujo batch y cómo comprobar que los archivos operativos y la bitácora quedan sincronizados.

4.1 Flujo de información

1. **Cliente HTTP** envía `POST /transfer` al backend NestJS con `cuenta`, `tipo` (`DEBITO` | `CREDITO`) y `monto`.
2. **Validación**: `class-validator` garantiza reglas de negocio básicas antes de tocar el sistema legacy (monto positivo, tipo permitido, cuenta no vacía).
3. **Puente de archivos**: el servicio escribe una única línea en `mainframe/TRANS.DAT` con el formato `cuenta|tipo|monto` (dos decimales). Este archivo es el analog del dataset de entrada que en MVS referenciaría el DD `TRANS` del JCL.
4. **Orquestación**: se ejecuta `scripts/run-job.sh`, que modela un JOB:
   - paso de compilación condicional del programa COBOL (`BANKBATCH.cbl` → `mainframe/bin/BANKBATCH`);
   - pausa con `sleep` para simular espera en cola;
   - ejecución del binario con directorio de trabajo `mainframe/`, de modo que los `ASSIGN TO "TRANS.DAT"` resuelvan igual que en un entorno batch real.
5. **Procesamiento legacy**: `BANKBATCH` lee `CUENTAS.DAT`, valida existencia y fondos (en débitos), aplica la regla contable y reescribe `CUENTAS.DAT` solo si la operación es válida.
6. **Resultado**: se genera `REPORTE.DAT` (`cuenta|status|saldo_final`). NestJS lee ese archivo y traduce `status` y saldo a una respuesta JSON con `status`, `saldoActualizado` y `mensaje` legible.
7. **Bitácora operacional**: NestJS persiste cada intento en la tabla `transferencias` con marca de tiempo, datos solicitados y resultado (`OK`, errores COBOL o códigos como `BATCH_EXEC_ERROR`). Además hace **append** de la misma operación en `mainframe/HISTORIAL.DAT` (histórico en PS, acumulativo). El COBOL **no** toca la base de datos; separa el ledger legacy (`.DAT`) de la trazabilidad del canal HTTP. Detalle en PERSISTENCIA.md.

```
sequenceDiagram
    participant C as Cliente
    participant N as NestJS
    participant F as Archivos .DAT
    participant J as run-job.sh
    participant B as BANKBATCH COBOL

    C->>N: POST /transfer
    N->>N: Validación DTO
    N->>F: escribe TRANS.DAT
    N->>J: exec script bash
    J->>B: compila si aplica + ejecuta
    B->>F: lee TRANS / CUENTAS
    B->>F: escribe CUENTAS / REPORTE
    J-->>N: RC 0
    N->>F: lee REPORTE.DAT
    N->>N: INSERT bitácora + append HISTORIAL.DAT
    N-->>C: JSON resultado
```

4.2 Corte / cierre diario

- **`POST /cierre`** (body opcional `{ "fecha": "YYYY-MM-DD" }`, por defecto el día **UTC** actual) genera `mainframe/CIERRE_YYYYMMDD.DAT`: resumen de movimientos de bitácora en ese día, totales por `resultadoBatch`, suma de saldos del maestro al instante y listado detallado. Útil para conciliación y archivo operativo tipo corte de fin de día (el rango horario es **UTC**; ajústalo en operación real si tu banco usa otra zona).
- **`GET /cierre/archivo?fecha=YYYY-MM-DD`** descarga ese dataset si ya fue generado.
- En la UI, la sección **“Corte / cierre diario”** (panel de exportación) enlaza ambas acciones.

4.3 Interfaz web (`frontend/`)

La aplicación React permite **ver el maestro** (`GET /cuentas`), **lanzar transferencias** y leer el resultado sin usar `curl`.

- En desarrollo, Vite proxifica `GET /api/cuentas`, `GET /api/historial` y `POST /api/transfer` hacia NestJS (véase `frontend/vite.config.ts` y la variable `API_PROXY_TARGET` para Docker).
- Para un build estático (`npm run build`) sirviendo solo los archivos `dist/`, define `VITE_API_URL` apuntando al backend (plantilla en `frontend/.env.example`).
- Tras cada intento de transferencia que llega al backend (éxito o error batch), el panel de historial puede refrescar `GET /historial` y la tabla de cuentas refleja cambios en `CUENTAS.DAT` cuando COBOL aplicó el movimiento.

4.4 Docker Compose (desarrollo con recarga)

En la raíz del repo, `docker-compose.yml` levanta:

- **`api`**: imagen con Node + **GnuCOBOL** + Bash + toolchain (`python3`, `make`, `g++`) para compilar `better-sqlite3`; monta `backend-nest/`, `mainframe/`, `scripts/` y **`./data`** (persistencia de `bank-core.sqlite`). Ejecuta `docker/entrypoint-api.sh` (`npm install` + `npm run start:dev`). Los cambios en TypeScript o en `BANKBATCH.cbl` se ven al guardar: Nest recarga el proceso y el siguiente batch recompila COBOL si el `.cbl` cambió.
- **`web`**: `node:bookworm-slim` con `frontend/` montado; ejecuta `docker/entrypoint-web.sh` (`npm install` + `vite dev`). Variable **`API_PROXY_TARGET=http://api:3000`** para que el proxy `/api` alcance el servicio por DNS interno.

Volúmenes nombrados (`*_node_modules`) evitan que el `node_modules` vacío del host pise las dependencias instaladas dentro del contenedor. **Cada arranque** corre `npm install` para recoger cambios en `package.json` / lockfile (trade-off a favor de mantener el entorno coherente tras pulls).

```
docker compose up --build
```

UI: `http://localhost:5173` · API: `http://localhost:3000`.

Relación entre **COBOL** y **JCL**: **docs/COBOL_Y_JCL.md**.

4.5 Cómo se simula el mainframe

En un IBM Z real, el **JCL** (`mainframe/TRANJOB.jcl`) describiría pasos `EXEC PGM=BANKBATCH` y los **DD** apuntarían a datasets catalogados. Aquí:

- **Hercules / MVS** no es obligatorio para demostrar la integración: los datasets se representan como **archivos secuenciales** en `mainframe/`.
- El script **reproduce la semántica del JOB**: preparación, compilación del programa y ejecución con ficheros locales.
- El programa fuente COBOL es el mismo tipo de artefacto que subirías al mainframe; solo cambia el compilador (GnuCOBOL en Linux frente al compilador IBM en z/OS).

Para acercarte a un entorno real, puedes trasladar `TRANJOB.jcl` a Hercules sustituyendo los nombres `DSN=` por tus datasets PS y compilando `BANKBATCH` en una `LOADLIB`.

4.6 Cómo validar que funciona

4.6.1 Prerrequisitos

- Node.js y `npm` en `backend-nest/`.
- `cobc` (GnuCOBOL) y `bash` disponibles en el PATH.

4.6.2 Arranque del backend

```
cd backend-nest
npm install
npm run start:dev
```

4.6.3 Casos de ejemplo (curl)

Saldo suficiente (débito)

```
curl -s -X POST http://localhost:3000/transfer \
  -H 'Content-Type: application/json' \
  -d '{"cuenta":"0012345678","tipo":"DEBITO","monto":100}'
```

Esperado: `status` `OK` y `saldoActualizado` coherente con el débito sobre `CUENTAS.DAT`.

Saldo insuficiente

```
curl -s -X POST http://localhost:3000/transfer \
  -H 'Content-Type: application/json' \
  -d '{"cuenta":"0098765432","tipo":"DEBITO","monto":500}'
```

Esperado: `ERROR_INSUFICIENTE`, saldo igual al actual sin mutar el maestro.

Cuenta inexistente

```
curl -s -X POST http://localhost:3000/transfer \
  -H 'Content-Type: application/json' \
```

```
  -d '{"cuenta":"9999999999","tipo":"CREDITO","monto":10}'
```

Esperado: `ERROR_NO_EXISTE`, saldo `0` en la respuesta numérica.

4.6.4 Restaurar datos semilla

Los tests manuales modifican `CUENTAS.DAT`. Para volver al estado inicial del repo, recupera la copia versionada (`git checkout -- mainframe/CUENTAS.DAT`) o conserva una copia local del fichero semilla.

4.6.5 Variable opcional `BANK_CORE_ROOT`

Si ejecutas el backend con un `cwd` distinto de `backend-nest/`, exporta la ruta absoluta del repo:

```
export BANK_CORE_ROOT=/ruta/a/bank-core-project
```

4.6.6 Tests automatizados de validación

```
cd backend-nest
npm run test:e2e
```

Las pruebas e2e levantan un entorno temporal, copian los archivos `.DAT`, ejecutan transferencias reales contra el batch y verifican que `TRANS.DAT`, `REPORTE.DAT`, `CUENTAS.DAT` y la bitácora queden consistentes sin alterar los datos locales del desarrollador.

5 COBOL y JCL: cómo encajan en la arquitectura

En un **IBM z/OS** real, el **JCL** (*Job Control Language*) es el "guión" que le dice al sistema operativo **qué programa ejecutar**, con **qué parámetros**, **qué librerías de carga** usar y **qué ficheros (datasets)** abrir en cada paso. El programa **COBOL** compilado es la "lógica de negocio" batch que corre dentro de ese trabajo.

5.1 Rol del JCL

Un JOB típico declara:

1. **JOB** — nombre del trabajo, clase, notificaciones.
2. **EXEC PGM=…** — programa a ejecutar (equivalente conceptual a `BANKBATCH` aquí).
3. **DD** (*Data Definition*) — **asignación de ficheros**: cada `DD` tiene un nombre lógico (`TRANS`, `CUENTAS`, `REPORT`) que el programa COBOL referencia, y el JCL lo enlaza a un dataset físico en disco o cinta.

En este entorno local, el fichero `mainframe/TRANJOB.jcl` es una **plantilla de referencia**: muestra cómo se verían esos `DD` en MVS/Hercules. **No se ejecuta literalmente en Linux**; su semántica la cumple `scripts/run-job.sh` (preparar entorno, compilar si hace falta, lanzar el programa con ficheros correctos).

5.2 Rol del programa COBOL (`BANKBATCH.cbl`)

El fuente COBOL define:

- **Archivos lógicos** (`SELECT … ASSIGN TO "TRANS.DAT"`) que corresponderían a los `DD` del JCL en mainframe.
- **Reglas de negocio**: validar transacción, buscar cuenta en el maestro, aplicar débito/crédito, impedir saldos negativos, escribir salida.

Es decir: **JCL = orquestación y cableado de datos**; **COBOL = procesamiento**. Ambos "se entienden" porque los **nombres de ficheros y el orden de ejecución** son coherentes: el job ejecuta el `PGM` que espera esos datasets.

5.3 Qué hace `run-job.sh` como sustituto local

1. Simula pasos de JOB (mensajes tipo consola, espera breve).
2. Compila `BANKBATCH.cbl` con **GnuCOBOL** si el fuente cambió (equivalente al paso de compilación/linkedit que harías en z/OS con otros herramientas).
3. Ejecuta el binario con `cwd` en `mainframe/` para que `TRANS.DAT`, `CUENTAS.DAT` y `REPORTE.DAT` coincidan con los `ASSIGN TO` del programa.

Si montás el mismo programa en **Hercules**, el JCL real dispararía el `EXEC PGM=BANKBATCH` y los `DD` apuntarían a tus datasets en lugar de los `.DAT` locales.

5.4 Bitácora operacional

El COBOL **no** escribe la bitácora: una capa **NestJS + TypeORM** persiste cada intento después de invocar el batch y de interpretar `REPORTE.DAT`. Mantiene separados el **ledger batch** (`.DAT`) y la **auditoría del API** (consultable con `GET /historial`). Ver PERSISTENCIA.md.

6 Persistencia y bases de datos

La plataforma combina **ficheros planos tipo mainframe** (`.DAT` procesados por COBOL) con una **base relacional moderna** para datos que conviene consultar, filtrar y conservar entre reinicios sin duplicar el rol del ledger legacy.

6.1 Qué está donde

Almacén	Contenido	Quién escribe	Persistencia
`mainframe/CUENTAS.DAT`	Maestro de cuentas / saldos	Programa COBOL (`BANKBATCH`)	Archivo en disco (también en volumen Docker junto a `mainframe/`)
`mainframe/TRANS.DAT` / `REPORTE.DAT`	Entrada y salida de cada corrida batch	NestJS + COBOL	Por corrida
`data/bank-core.sqlite`	Bitácora de intentos de transferencia (auditoría)	NestJS vía TypeORM	Archivo local reemplazable por Postgres en despliegues compartidos

El **COBOL no accede a la base operacional**: mantiene la separación *canal online → puente → batch legacy*. La base moderna guarda **trazabilidad** de lo que pidió el canal HTTP y qué respondió el batch (incluidos errores como `BATCH_EXEC_ERROR`).

6.2 Perfil local y perfil productivo

- En local, un solo fichero (`bank-core.sqlite`) permite levantar el sistema sin servidor de base de datos adicional.
- En Docker Compose se monta `./data` para conservar la bitácora entre reinicios.
- Para un entorno compartido, la misma capa TypeORM puede evolucionar a PostgreSQL con migraciones versionadas.

Ruta por defecto: `data/bank-core.sqlite` bajo la raíz del repo (`bank-core-project/data/`).
Override: variable de entorno `SQLITE_DB_PATH` (ruta absoluta recomendada).

> En este repositorio se usa `synchronize: true` para simplificar el arranque local. En un sistema real deberían usarse **migraciones** versionadas.

6.3 Cuándo usar PostgreSQL (recomendación profesional)

Para un entorno **multiusuario**, **concurrencia alta**, backups gestionados o despliegue en la nube, **PostgreSQL** es la opción habitual junto a NestJS:

- Tipos numéricos/decimales más estrictos para dinero (idealmente `NUMERIC`/`DECIMAL`, no `REAL`).
- Roles, réplicas, extensiones (p. ej. `uuid-ossp`), mejor aislamiento transaccional.
- Encaja con Docker (`postgres:16`) o servicios gestionados (RDS, Neon, etc.).

La decisión recomendada para producción sería PostgreSQL, manteniendo el archivo local solo como perfil de desarrollo.

6.4 Docker

El servicio `api` del `docker-compose.yml` monta `./data:/workspace/data` para que el fichero SQLite sobreviva reinicios del contenedor (mientras no borres el directorio en el host).

La imagen incluye herramientas de compilación (`python3`, `make`, `g++`) necesarias para el paquete nativo `better-sqlite3`.

6.5 Tabla actual

- **`transferencias`**: columnas `id` (UUID), `createdAt`, `cuentaSolicitada`, `tipo`, `monto`, `resultadoBatch`, `saldoReportado`.

La API expone el contenido con `GET /historial?limite=…&desde=YYYY-MM-DD&hasta=YYYY-MM-DD`.

6.6 Próximas mejoras

1. Sustituir `synchronize` por **migraciones TypeORM** y documentar el flujo `migration:run`.
2. Añadir un diagrama ER en `docs/` (PNG o Mermaid).
3. Rama o apartado “Producción”: mismo código con `DATABASE_URL` → Postgres y perfil Docker `postgres`.