Documento técnico completo — Mi Testamento Digital (DeadApp)

Fecha: 24-11-2025

Propósito

Este documento describe de forma extensa el proyecto "Mi Testamento Digital" (repositorio `DeadApp`), incluyendo arquitectura, tecnologías, flujo de datos, componentes clave, endpoints, pruebas, despliegue y recomendaciones. Está pensado para un lector técnico (profesor/QA) que necesite entender el diseño y funcionamiento del sistema.

Resumen ejecutivo

- Tipo: API backend en Node.js/TypeScript con un frontend Next.js (repositorio `DEADFRONT`).
- Arquitectura: Hexagonal/Ports & Adapters. Separación clara entre dominio, casos de uso, adaptadores de persistencia y controladores HTTP.
- Seguridad: Autenticación vía Keycloak (soporte), fallback de desarrollo para simular usuarios; cifrado AES-256-CBC para secretos almacenados.
- Persistencia: MongoDB (módulo `mongodb` + Mongoose). En tests se usa `mongodb-memory-server`.
- CI: GitHub Actions con workflow para ejecutar tests en Ubuntu.

Tecnologías principales

- Node.js + TypeScript
- Express.js para web server
- Mongoose para ODM con MongoDB
- Next.js + React + Tailwind (frontend separado)
- Keycloak (autenticación, roles)
- AES-256-CBC (cifrado simétrico para secretos)
- Jest + ts-jest + mongodb-memory-server (tests)
- Docker (dev: MongoDB, Keycloak se puede usar en contenedor)
- PowerShell scripts (helpers para entorno dev en Windows)
- `md-to-pdf` (herramienta usada para generar PDFs desde Markdown)

Estructura del proyecto (resumen de carpetas relevantes)

- `src/` — código principal del backend
  - `application/` — casos de uso y DTOs
    - `use-cases/GestionarBovedaUseCase.ts` — lógica principal para agregar/obtener/eliminar activos y obtener secreto por activo.
    - `dtos/AgregarActivoDto.ts` — DTO para la creación de activos.
  - `domain/` — entidades y value objects (e.g., `Boveda`, `ActivoDigital`, `Uuid`).
  - `infrastructure/` — adapters y controllers
    - `database/adapters/` — adaptadores concretos para MongoDB (`MongoBovedaAdapter.ts`).
    - `http/controllers/` — controladores Express (BovedaController, ContactoController, VidaController, StatusController).
    - `http/routes/` — definición de rutas (e.g., `boveda.routes.ts`).

        - `middleware/auth.middleware.ts` — integración con `express-oauth2-jwt-bearer` y fallback dev.
    - `lib/cryptoService.ts` — servicio de cifrado/descifrado (AES-256-CBC con clave derivada por SHA-256).
    - `tests/` — pruebas de integración (rotación de claves, etc.).

Flujos clave

1. Agregar un activo (alta rápida):
    - Endpoint: `POST /api/boveda/activos` (requiere rol `usuario_titular`).
    - Flujo:
        - Controlador `agregarActivoController` obtiene `usuarioId` del token (o del mock dev header).
        - Llama a `GestionarBovedaUseCase.agregarActivo(usuarioId, dto)`.
        - El use-case cifra el `password` con `CryptoService.encrypt()` y crea un `ActivoDigital` con `passwordCifrada`.
        - Persiste la bóveda con `bovedaRepository.guardar(boveda)` (Mongo adapter).
        - Registra una auditoría ligera en `AuditModel`.
2. Leer activos:
    - Endpoint: `GET /api/boveda/activos` (titular).
    - Salida: lista de activos con el campo `passwordCifrada` que contiene la cadena cifrada (NO se devuelven contraseñas en claro por defecto).
    - Diseño: para seguridad, el API no devuelve secretos en texto claro en la lista.
3. Obtener secreto de un activo (revelado controlado):
    - Endpoint: `GET /api/boveda/activos/:id/secreto` (titular).
    - Flujo: el controlador llama a `GestionarBovedaUseCase.obtenerSecretoActivo(usuarioId, activoId)` que desencripta y devuelve el secreto. Se recomienda auditar cada petición a este endpoint (se añadió auditoría en creación/eliminación; se puede ampliar).
4. Solicitudes de activación manual (contactos clave):
    - Existe soporte para que `contacto_clave` solicite una activación vía `POST /api/activacion/solicitar` (controller/adapters implementados).

Modelo de datos (resumen)

- `Boveda`:
    - _id (string - usuarioId)
    - nombre, descripcion, estado
    - activos: [{ id, plataforma, usuarioCuenta, passwordCifrada, notas, categoria, gestionado }]

Seguridad y cifrado

- En reposo: las contraseñas se cifran usando AES-256-CBC. La clave se deriva de ENCRYPTION_KEY mediante SHA-256.
- En tránsito: la API debe estar detrás de HTTPS en producción.
- Autenticación: Keycloak con roles esperados (`usuario_titular`, `contacto_clave`, `admin`).

- En desarrollo: `auth.middleware` añade un fallback que permite simular usuarios con headers `x-dev-user` y `x-dev-roles` cuando no hay Keycloak configurado — útil para pruebas locales.

Pruebas y CI

- Tests unitarios/integración: Jest + ts-jest. Se proporciona `src/tests/rotate.integration.test.ts` que valida la rotación de claves y que las entradas se cifran/descifran correctamente.
- En Windows, `mongodb-memory-server` puede lanzar errores `EPERM` al matar procesos; se añadieron medidas:
  - `jest.global-setup.js` fija `MONGOMS_DOWNLOAD_DIR` y `TMP` a una carpeta `.jest_tmp` en el repo.
  - `rotate.integration.test.ts` evita llamar a `mongod.stop()` directamente en `win32` para no provocar `EPERM` .
  - `jest.global-teardown.js` intenta `taskkill` en Windows para cerrar procesos `mongod` y dejar Jest sin handles abiertos.
- CI: `.github/workflows/ci.yml` ejecuta `npm test` en `ubuntu-latest` para obtener ejecuciones limpias en Linux.

Operaciones y arranque local (dev)

Requisitos: Node.js (v18+ recomendado), npm, Docker (opcional para Mongo/Keycloak).

1. Preparar .env (ejemplo en `.env.example`) — variables clave:
   - `PORT` — puerto backend (recomendado `3002` para evitar conflicto con Next.js que usa 3000).
   - `MONGO_URI` — `mongodb://localhost:27017/mi-testamento-digital` (o el contenedor Docker).
   - `ENCRYPTION_KEY` — clave para cifrado (obligatoria en producción).
2. Arrancar Mongo local (Docker) o usar `mongod` localmente.
3. Instalar dependencias y arrancar:

```
Set-Location 'C:\Path\To\DeadApp'
npm install
$env:PORT = '3002'
$env:NODE_ENV = 'development'
npm run dev
```

4. Probar endpoints de diagnóstico:
   - `GET /health`
   - `GET /api/status` (usa dev headers si no hay Keycloak)

Endpoints principales (resumen)

- `GET /health` — estado básico
- `GET /api/status` — estado y conexión a Mongo
- `POST /api/boveda/activos` — crear activo (titular)
- `GET /api/boveda/activos` — listar activos (titular)
- `GET /api/boveda/activos/:id/secreto` — revelar secreto (titular)
- `DELETE /api/boveda/activos/:id` — eliminar activo (titular)

- Rutas para `contactos` y `vida` disponibles en `infrastructure/http/routes`

Consideraciones de seguridad y mejoras pendientes

- No exponer secretos en listados por defecto (ya implementado).
- Añadir auditoría al endpoint de revelado de secretos para cumplir requisitos de trazabilidad.
- Considerar rotación periódica de la `ENCRYPTION_KEY` con proceso de re-encriptado y auditoría (ya hay test de rotación).
- Hardenización de dev endpoints (ya se añadió `DEV_ENDPOINTS_SECRET` en diseño; asegurar que no estén expuestos en prod).

Documentación y artefactos generados

- `REPORT.md` — resumen y pasos para captura (ya en repo).
- `REPORT.pdf` — versión PDF corta generada.
- `FULL_REPORT.md` — este documento extenso.
- `FULL_REPORT.pdf` — PDF generado a partir de este archivo (generado y añadido al repo si se solicita).

Próximos pasos recomendados

1. Añadir auditoría al endpoint `GET /api/boveda/activos/:id/secreto` y registrar quién/ cuándo/por qué solicita la revelación.
2. Implementar tests E2E que creen, lean y borren activos usando el backend y (opcional) el frontend.
3. Añadir cobertura de seguridad: escaneo de dependencias y análisis estático.
4. Preparar despliegue demo en entorno controlado (Docker Compose con Mongo y Keycloak) para evaluación.

Contacto y referencias

Repositorio backend: `DeadApp` (carpeta raíz de este proyecto). Repositorio frontend: `DEADFRONT/frontend` (carpeta separada en workspace).

Fin del documento.