Mirfa Secure Transaction App 🔒

A secure transaction processing application built as a submission for the Mirfa Engineering Challenge. This project demonstrates **Envelope Encryption** using AES-256-GCM, structured within a scalable TurboRepo monorepo.

🚀 Live Demo

- **Frontend (Vercel):** https://mirfa-secure-app-web-ivory.vercel.app/
- **Backend (Railway):** https://api-production-5ffc.up.railway.app/

🏗️ Architecture & Design

Envelope Encryption Strategy

To ensure maximum security, this application uses **Envelope Encryption**. We never store the data encryption keys (DEKs) in plain text.

1. **Key Generation:** For every transaction, a unique 32-byte **DEK** is generated.
2. **Payload Encryption:** The user's JSON payload is encrypted using this DEK (AES-256-GCM).
3. **Key Encryption:** The DEK itself is then encrypted using a system-wide **Master Key (KEK)**.
4. **Storage:** The database stores:
    - The encrypted payload.
    - The encrypted DEK.
    - The unique Nonces (IVs) and Auth Tags for both.

Tech Stack

Component	Technology	Reasoning
Monorepo	TurboRepo	Efficient build caching and shared package management.
Frontend	Next.js 14	React server components and optimized rendering.
Backend	Fastify	High-performance, low-overhead Node.js framework.
Database	PostgreSQL	Robust relational data storage.
ORM	Prisma	Type-safe database access and schema management.
Crypto	Node.js Crypto	Native, standard library for AES-256-GCM operations.

Deployment	Vercel & Railway	Scalable serverless (FE) and containerized (BE) hosting.

💾 Database Configuration (Neon / Postgres)

The project uses **Prisma ORM** connected to a PostgreSQL database (specifically optimized for **Neon Serverless Postgres**).

1. Connection Setup The `datasource` block in `packages/db/prisma/schema.prisma` is configured for PostgreSQL. The connection string is retrieved from the `DATABASE_URL` environment variable.

2. Prisma Client Singleton To handle serverless connection limits (common with Vercel and Neon) and Next.js hot-reloading, the Prisma Client is instantiated as a global singleton. This prevents "Too many connections" errors by reusing the existing connection instance.

```
import { PrismaClient } from '@prisma/client';

const globalForPrisma = global as unknown as { prisma: PrismaClient };

export const prisma =
  globalForPrisma.prisma ||
  new PrismaClient({
    log: ['query', 'error', 'warn'],
  });

if (process.env.NODE_ENV !== 'production') globalForPrisma.prisma = prisma;
```

3. Data Models The schema defines robust models such as `User` for identity management and `SecureRecord` for storing encrypted transactions, linked via relation fields.

📂 Project Structure

```
.
├── apps
│   ├── api          # Fastify Backend Service
│   └── web          # Next.js Frontend Application
├── packages
│   ├── crypto       # Shared encryption logic (AES-GCM helpers)
│   ├── db           # Shared Prisma schema and client
│   ├── ui           # Shared React UI components
│   ├── config       # Shared TS/ESLint configs
└── turbo.json       # Build pipeline configuration
```

🛠️ Getting Started

Prerequisites

- Node.js 20+

- pnpm (`npm install -g pnpm`)
- PostgreSQL database (Local or Cloud)

1. Clone the repository

```
git clone <your-repo-url>
cd mirfa-secure-app
```

2. Install Dependencies

```
pnpm install
```

3. Environment Setup

Create a `.env` file in the root (or in specific apps) based on `.env.example` .

Required Variables:

```
# Database Connection
DATABASE_URL="postgresql://user:password@localhost:5432/mirfa_db"

# Security (Must be 32 bytes / 64 hex characters)
# Generate via: openssl rand -hex 32
MASTER_KEY="your_64_char_hex_master_key_here"

# API Port
PORT=3001

# Frontend API Connection
NEXT_PUBLIC_API_URL="http://localhost:3001"
```

4. Database Migration and build generation

Generate the Prisma Client and push the schema to your database.

```
pnpm build
pnpm db:push
```

5. Run Locally

Start both the Frontend and Backend in development mode.

```
pnpm dev
```

- **Web:** `http://localhost:3000`
- **API:** `http://localhost:3001`

6. 🐬 Run Backend with Docker (Local)

You can containerize and run the backend API using the `Dockerfile` located in the root directory.

Build the image:

```
docker build -t mirfa-api .
```

Run the container backend : Ensure you have a `.env` file in your root directory with the necessary variables (DATABASE_URL, MASTER_KEY, etc.).

```
docker run -p 3001:3001 --env-file .env mirfa-api
```

Note: The API will be accessible at `http://localhost:3001`.

🔐 API Reference

1. Encrypt Payload

POST `/tx/encrypt` Accepts a party ID and a JSON payload, encrypts them, and returns the record ID.

```
// Request
{
  "partyId": "party_123",
  "payload": { "amount": 500, "currency": "USD" }
}
```

2. Retrieve Encrypted Record

GET `/tx/:id` Returns the raw stored data (encrypted ciphertext, wrapped DEK, nonces) without decrypting.

3. Decrypt Payload

POST `/tx/:id/decrypt` Server-side decryption. Unwraps the DEK using the Master Key, then decrypts the payload.

🧪 Running Tests

The cryptography package includes unit tests to verify encryption integrity and tamper resistance.

```
pnpm test
```

🛡️ Cryptography Engine (Deep Dive)

The core security logic resides in `packages/crypto` and implements **AES-256-GCM** (Galois/Counter Mode) to ensure both confidentiality and integrity.

Envelope Encryption Flow (`encrypt.ts`)

We utilize a two-tier key architecture to secure data:

1. **Session Setup:** A unique, random 32-byte **Data Encryption Key (DEK)** is generated for *every single transaction*.
2. **Layer 1 (Payload):** The actual JSON payload is encrypted using this DEK.
   - *Output:* `payload_ct` (Ciphertext), `payload_nonce` (12-byte IV), `payload_tag` (Auth Tag).
3. **Layer 2 (Key):** The DEK is then encrypted using the system **Master Key (KEK)**.
   - *Output:* `dek_wrapped` (Encrypted Key), `dek_wrap_nonce` (12-byte IV), `dek_wrap_tag` (Auth Tag).
4. **Result:** The database stores only the encrypted artifacts. The raw DEK is discarded immediately from memory.

Decryption & Verification (`decrypt.ts`)

Decryption is a strict reverse process that validates integrity at every step:

1. **Validation:** Checks if the Master Key is valid hex (64 chars) and Nonces are standard size (12 bytes).
2. **Unwrap DEK:** The system attempts to decrypt `dek_wrapped` using the Master Key. If the `dek_wrap_tag` does not match, the operation fails immediately (tamper detection).
3. **Decrypt Payload:** Using the unwrapped DEK, the system attempts to decrypt the payload. Again, the `payload_tag` is verified.

Error Codes Standard (`errors.ts`)

The system throws standardized error codes to prevent information leakage while aiding debugging:

Error Code	Meaning	Cause
ER-4001	`INTEGRITY_FAIL`	Auth tag mismatch. Data or Key was tampered with.

ER-4002	`INVALID_NONCE`	IV is not 12 bytes (24 hex chars).
ER-4003	`INVALID_KEY`	Master Key is not 32 bytes (64 hex chars).
ER-4004	`CIPHERTEXT_CORRUPT`	Hex string contains invalid characters.
ER-5000	`UNKNOWN`	Unexpected system failure.

🧪 Security Testing Suite

The project includes a robust test suite using **Vitest** (`index.test.ts`) that goes beyond "happy paths" to verify the system behaves correctly under attack.

Run Tests

```
pnpm test
```

Test Scenarios Covered

The suite validates 7 specific security scenarios:

1. ✅ **Lifecycle Success:** Verifies that data encrypted with a Master Key can be decrypted with the *same* key.
2. 🛡️ **Payload Tampering:** Modifying a single character in `payload_ct` triggers `ER-4001` .
3. 🛡️ **Tag Manipulation:** Changing the `payload_tag` (Authentication Tag) triggers `ER-4001` .
4. 🛡️ **Key Envelope Attack:** Tampering with the encrypted DEK (`dek_wrapped`) prevents the key from being unwrapped.
5. 🛡️ **Nonce Replay/Swap:** Swapping the payload nonce with the key nonce triggers an integrity failure.
6. 🛡️ **Invalid Nonce Length:** Providing a non-12-byte nonce throws `ER-4002` .
7. 🛡️ **Wrong Master Key:** Attempting to decrypt a record with a different valid Master Key fails authentication.

⚙️ Backend Architecture (Fastify API)

The API (`apps/api`) is built with **Fastify** for high performance and low overhead. It follows a modular "Plugin & Factory" pattern to keep concerns separated.

📂 Backend Structure

```
apps/api/src
├── config.ts           # Env validation (Fails fast if MASTER_KEY is invalid)
├── index.ts            # Entry point (Starts server)
```

```
├── server.ts            # App Factory (Registers plugins & routes)
├── decorators
│   └── authenticate.ts  # JWT Verification Decorator
├── plugins
│   ├── cors.ts          # CORS configuration
│   └── jwt.ts           # JWT Strategy setup
└── routes
    ├── auth.routes.ts   # Login / Signup (Bcrypt + Prisma)
    ├── health.routes.ts # Health check endpoint
    └── tx.routes.ts     # Core Transaction Logic (Protected)
```

1. Boot Sequence & Validation (`config.ts`)

The server employs a **"Fail Fast"** strategy. On startup, `config.ts` validates critical environment variables.

- **Critical Check:** If `MASTER_KEY` is missing or not exactly 64 hex characters (32 bytes), the process exits immediately (`process.exit(1)`). This prevents the server from running in an insecure state.

2. Authentication Strategy

Security is implemented using `fastify-jwt` and `bcryptjs` .

- **User Management:** Passwords are hashed with **bcrypt** (salt rounds: 10) before storage in Prisma.
- **Session:** On login, a **JWT** is issued containing the user's ID.
- **Decorator:** We created a custom `authenticate` decorator (`decorators/authenticate.ts`). This allows us to protect routes easily:

```
// Example protection in tx.routes.ts
app.post('/tx/encrypt', { onRequest: [app.authenticate] }, handler);
```

3. Transaction Logic (`tx.routes.ts`)

This controller acts as the bridge between the **User**, the **Database**, and the **Crypto Engine**.

- **Isolation:** Transactions are linked to the specific User ID extracted from the JWT (`req.user.id`). Users can only fetch their own records.
- **Integration:**
  1. Receives `partyId` and `payload` .
  2. Calls the shared `@repo/crypto` library to perform the Envelope Encryption.
  3. Uses `@repo/db` (Prisma) to store the result (`SecureRecord`) containing the encrypted payload and the wrapped DEK.

💻 Frontend Architecture (UX & Components)

The frontend (`apps/web`) is built with **Next.js 14** using the App Router, styled with Tailwind CSS, and enhanced with Framer Motion for smooth transitions.

📂 Component Breakdown

```
apps/web/src/app
├── components
│   ├── AuthScreen.tsx    # Login/Signup handling with JWT storage
│   ├── Dashboard.tsx     # Main secure vault view
│   ├── RecordItem.tsx    # Individual transaction row with decrypt action
│   └── CopyButton.tsx    # Utility for copying JSON/IDs
├── page.tsx              # Root controller managing Auth state
└── types.ts              # Frontend Types (RecordType)
```

1. Authentication Flow (`AuthScreen.tsx`)

- **JWT Handling:** On successful login, the JWT is stored in `localStorage` ('mirfa_token').
- **Session Persistence:** `page.tsx` checks for this token on mount to restore the user session.
- **Logout:** Clearing the token instantly redirects the user back to the Auth Screen.

2. Secure Dashboard (`Dashboard.tsx`)

- **Data Fetching:** Fetches encrypted records from `/tx` using the Bearer token.
- **Encryption Action:** Users can input a `Party ID` and `JSON Payload` . The app sends this to `/tx/encrypt` and immediately updates the UI with the new encrypted record.

3. Decryption UX (`RecordItem.tsx`)

- **On-Demand Decryption:** Records are shown in their encrypted state by default (locked icon).
- **User Action:** Clicking "Decrypt" sends a request to `/tx/:id/decrypt` .
- **Verification:** If the integrity check passes on the backend, the plaintext JSON is revealed with a smooth animation.
- **Security:** Decrypted data is never stored in `localStorage` ; it exists only in React state