VeriPass

Sistem Verifikasi Aset Berbasis Blockchain

Manual Book

Januari 2026

Daftar Isi

1 Penjelasan Umum **4**
- 1.1 Deskripsi Singkat Sistem . . . 4
- 1.2 Tujuan Manual . . . 4
- 1.3 Gambaran Umum Fungsionalitas . . . 4
- 1.4 Komponen Utama Sistem . . . 5

2 Rangka Perangkat Lunak **6**
- 2.1 Struktur Umum Sistem . . . 6
- 2.2 Smart Contract . . . 6
  - 2.2.1 AssetPassport . . . 6
  - 2.2.2 EventRegistry . . . 7
- 2.3 Oracle . . . 7
- 2.4 Backend . . . 9
- 2.5 Frontend . . . 9
- 2.6 Pattern dan Data Flow . . . 10
  - 2.6.1 Data Integrity Pattern . . . 10
  - 2.6.2 Authentication Flow . . . 10

3 Setup **11**
- 3.1 Persiapan dan Konfigurasi . . . 11
- 3.2 Menjalankan Smart Contract . . . 11
- 3.3 Menjalankan Oracle . . . 11
- 3.4 Menjalankan Backend . . . 12
- 3.5 Menjalankan Frontend . . . 12
- 3.6 Urutan Eksekusi Keseluruhan . . . 12

4 Penggunaan **13**
- 4.1 Asset Minting . . . 13
- 4.2 Asset Viewing dan Verification . . . 14
- 4.3 Verified Event . . . 15
  - 4.3.1 Custom Event (User-Submitted) . . . 15
  - 4.3.2 Oracle-Verified Event . . . 15

5 Lain-lain **17**
- 5.1 Catatan Keamanan . . . 17
  - 5.1.1 Private Key Management . . . 17
  - 5.1.2 Smart Contract Security . . . 17
  - 5.1.3 Backend Security . . . 17
  - 5.1.4 Data Integrity . . . 17
- 5.2 Kesimpulan dan Saran . . . 18
  - 5.2.1 Saran Pengembangan . . . 18

A Source Code Smart Contract **19**
- A.1 AssetPassport.sol . . . 19
- A.2 EventRegistry.sol . . . 21

B Database Schema **25**

Daftar Gambar

1.1 Arsitektur Sistem VeriPass 5

2.1 Alur Pemrosesan Oracle 8
2.2 Alur Data dan Verifikasi 10

4.1 Form Minting Passport 13
4.2 Detail Passport dengan Verification Badge 14
4.3 Event Timeline dengan Filter 16

Daftar Tabel

2.1 Stack Teknologi VeriPass 6
2.2 Fungsi Utama AssetPassport 7
2.3 Endpoint API Utama 9

Bab 1

Penjelasan Umum

1.1 Deskripsi Singkat Sistem

VeriPass adalah sistem verifikasi aset berbasis blockchain yang menciptakan paspor digital anti-manipulasi untuk aset fisik menggunakan standar ERC-721 NFT. Sistem ini dirancang untuk memberikan bukti kepemilikan dan riwayat aset yang transparan, terdesentralisasi, dan dapat diverifikasi secara kriptografis.

Setiap aset fisik direpresentasikan sebagai NFT unik pada blockchain Ethereum, dengan hash metadata yang tersimpan secara permanen. Integritas data dijamin melalui perbandingan hash on-chain dan off-chain, memastikan setiap modifikasi data dapat terdeteksi.

1.2 Tujuan Manual

Dokumen ini bertujuan sebagai panduan teknis komprehensif untuk:

1. Memahami arsitektur dan komponen sistem VeriPass
2. Melakukan instalasi dan konfigurasi sistem secara lengkap
3. Mengoperasikan fitur-fitur utama aplikasi
4. Memahami aspek keamanan dan praktik terbaik penggunaan

1.3 Gambaran Umum Fungsionalitas

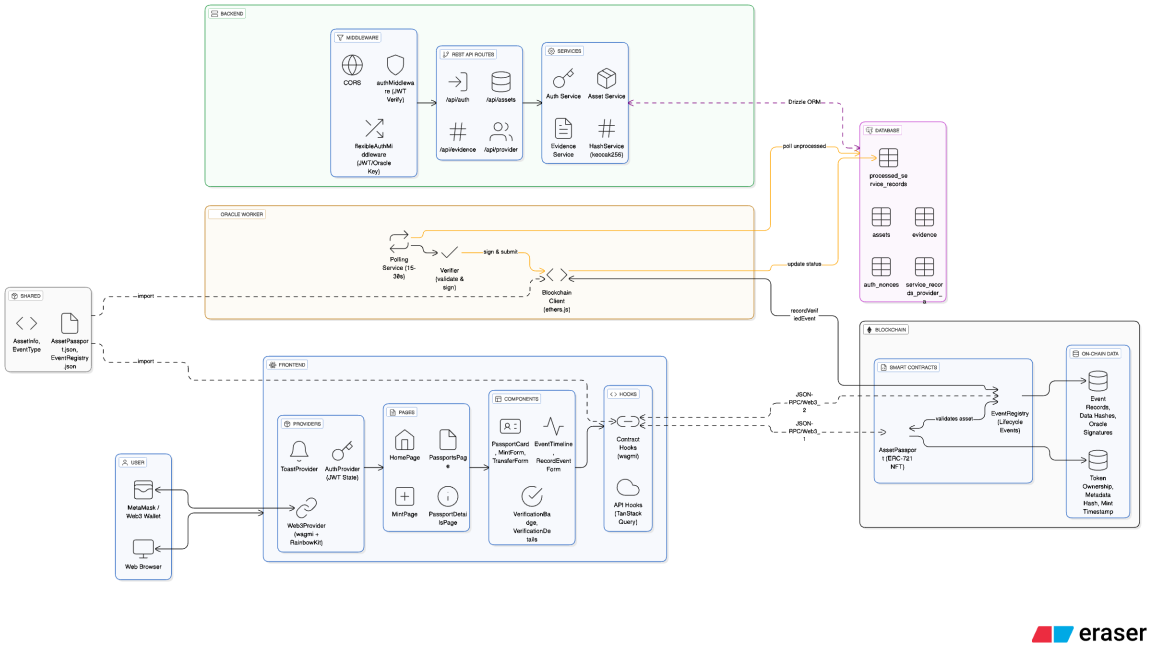
VeriPass menyediakan fungsionalitas utama berikut:

- **Minting Paspor Aset** – Pembuatan NFT unik untuk setiap aset fisik dengan metadata terverifikasi
- **Verifikasi Integritas** – Perbandingan hash on-chain dan off-chain untuk memastikan data tidak dimanipulasi
- **Pencatatan Event** – Riwayat lengkap lifecycle aset termasuk maintenance, sertifikasi, dan transfer kepemilikan
- **Oracle Integration** – Verifikasi otomatis service record dari provider terpercaya
- **Web3 Authentication** – Autentikasi berbasis wallet untuk keamanan maksimal

1.4 Komponen Utama Sistem

Sistem VeriPass terdiri dari empat komponen utama yang saling terintegrasi, sebagaimana diilustrasikan pada Gambar 1.1.

1. **Frontend** – Aplikasi React dengan RainbowKit untuk koneksi wallet dan wagmi untuk interaksi smart contract
2. **Backend** – Server Hono.js dengan PostgreSQL database, menyediakan REST API dan autentikasi JWT
3. **Smart Contracts** – Kontrak Solidity pada blockchain Ethereum (AssetPassport dan EventRegistry)
4. **Oracle** – Worker service yang memproses service record terverifikasi dan mencatatnya ke blockchain



Gambar 1.1: Arsitektur Sistem VeriPass

Bab 2

Rangka Perangkat Lunak

2.1 Struktur Umum Sistem

VeriPass menggunakan arsitektur three-tier yang memisahkan presentasi, logika bisnis, dan penyimpanan data. Teknologi yang digunakan dirangkum dalam Tabel 2.1.

Tabel 2.1: Stack Teknologi VeriPass

Kategori	**Teknologi**
Frontend	React, Vite, RainbowKit, wagmi
Backend	Hono.js, Drizzle ORM
Database	PostgreSQL
Blockchain	Solidity, Hardhat, ethers.js
Runtime	Bun

Direktori `shared/` berfungsi sebagai satu-satunya sumber ABI kontrak yang digunakan bersama oleh frontend dan backend. Setelah perubahan kontrak, ABI harus di-export ulang menggunakan script `export-abi.ts`.

2.2 Smart Contract

2.2.1 AssetPassport

Kontrak `AssetPassport` mengimplementasikan standar ERC-721 untuk merepresentasikan paspor aset sebagai NFT. Fitur utama meliputi:

- **Minting dengan Metadata Hash** – Setiap passport menyimpan hash keccak256 dari metadata off-chain
- **Authorized Minters** – Hanya alamat terotorisasi atau owner yang dapat melakukan minting
- **Ownership Tracking** – Counter `ownershipHands` melacak berapa kali aset berpindah tangan
- **Emergency Controls** – Fungsi pause/unpause untuk kondisi darurat

Fungsi-fungsi utama kontrak dirangkum pada Tabel 2.2.

Tabel 2.2: Fungsi Utama AssetPassport

Fungsi	Deskripsi
`mintPassport`	Minting passport baru dengan metadata hash
`getAssetInfo`	Mengambil informasi asset (hash, timestamp, status)
`getOwnershipHand`	Mengembalikan jumlah perpindahan kepemilikan
`addAuthorizedMinter`	Menambah alamat sebagai authorized minter
`deactivatePassport`	Menonaktifkan passport (aset hilang/dicuri)

2.2.2 EventRegistry

Kontrak `EventRegistry` berfungsi sebagai append-only log untuk mencatat seluruh event lifecycle aset. Tipe event yang didukung:

- `MAINTENANCE` (0) – Service, repair, inspeksi
- `VERIFICATION` (1) – Verifikasi keaslian oleh oracle
- `WARRANTY` (2) – Klaim dan perpanjangan garansi
- `CERTIFICATION` (3) – Sertifikasi pihak ketiga
- `CUSTOM` (4) – Event kustom dari pengguna

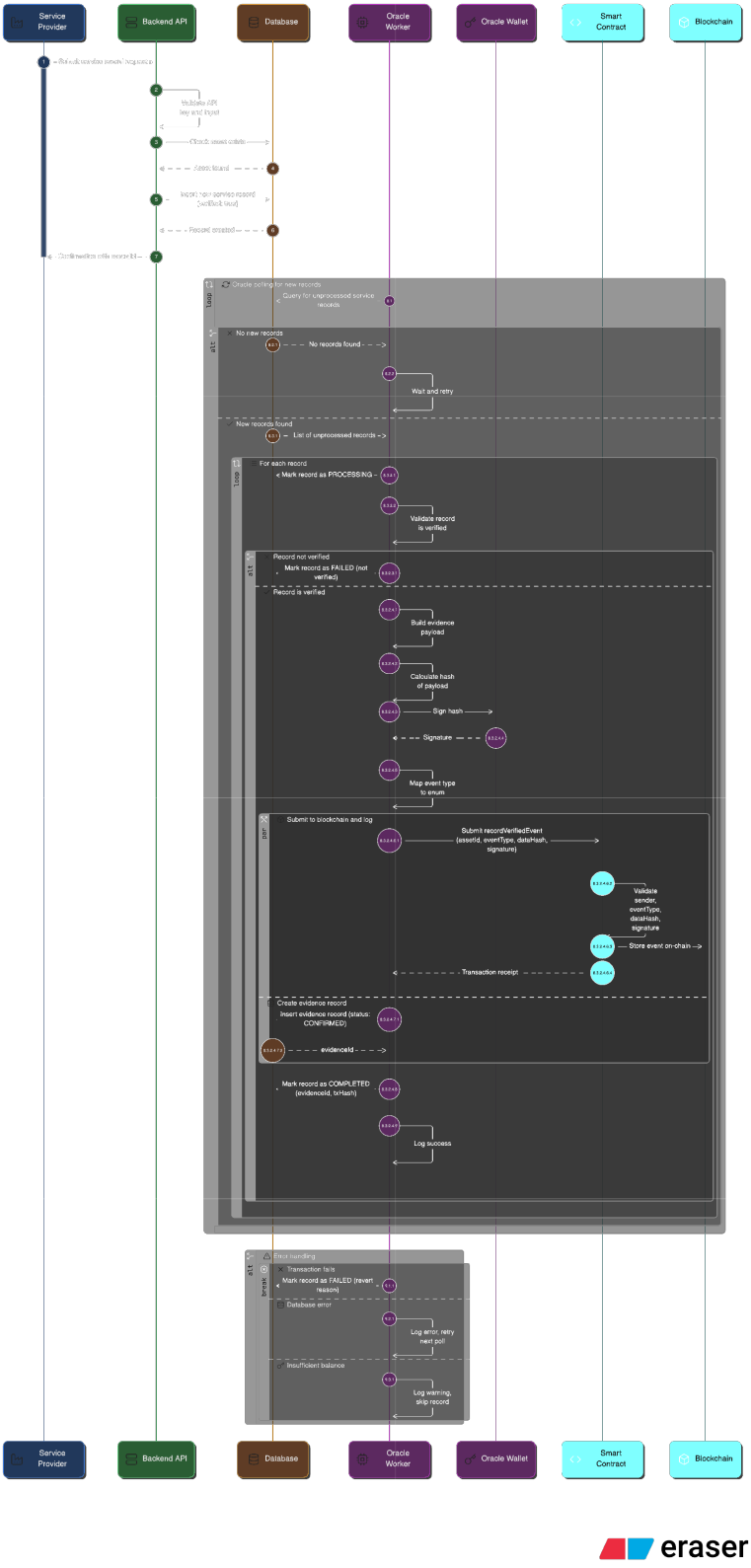
Terdapat dua jalur pencatatan event:

1. **User Events** (`recordEvent`) – Disubmit langsung oleh pemilik aset, bersifat unverified
2. **Oracle Events** (`recordVerifiedEvent`) – Disubmit oleh trusted oracle dengan signature, bersifat verified

2.3 Oracle

Oracle adalah background worker yang menjembatani data off-chain dengan on-chain. Proses kerja oracle:

1. Polling database setiap 15-30 detik untuk service record baru
2. Memvalidasi record yang sudah diverifikasi oleh provider
3. Menghitung hash deterministik dari data event
4. Menandatangani hash dengan wallet oracle
5. Submit ke kontrak EventRegistry sebagai verified event
6. Update status record: PENDING $\rightarrow$ PROCESSING $\rightarrow$ COMPLETED/FAILED



eraser

Gambar 2.1: Alur Pemrosesan Oracle

2.4 Backend

Backend dibangun dengan Hono.js dan menyediakan REST API untuk operasi berikut:

- **Authentication** – Web3 signature-based auth (nonce → sign → verify → JWT)
- **Asset Management** – CRUD operasi untuk metadata aset
- **Evidence Tracking** – Pencatatan dan query event/evidence
- **Provider Integration** – API untuk service provider eksternal

Endpoint utama API dirangkum pada Tabel 2.3.

Tabel 2.3: Endpoint API Utama

Method	Endpoint	Deskripsi
POST	/api/auth/nonce	Request nonce untuk signing
POST	/api/auth/verify	Verifikasi signature, dapatkan JWT
POST	/api/assets	Buat asset baru
GET	/api/assets/:id	Query asset by ID
POST	/api/evidence	Catat evidence baru
GET	/api/evidence/asset/:id	Query evidence by asset

2.5 Frontend

Frontend React menggunakan arsitektur berbasis hooks dengan pemisahan jelas antara API hooks dan contract hooks:

- **/hooks/api/** – TanStack Query hooks untuk komunikasi dengan backend
- **/hooks/contracts/** – wagmi hooks untuk interaksi smart contract

Halaman utama aplikasi:

- **HomePage** – Landing page dengan informasi sistem
- **PassportsPage** – Daftar semua passport yang di-mint
- **MintPage** – Form untuk minting passport baru
- **PassportDetailsPage** – Detail passport dengan verification dan event timeline

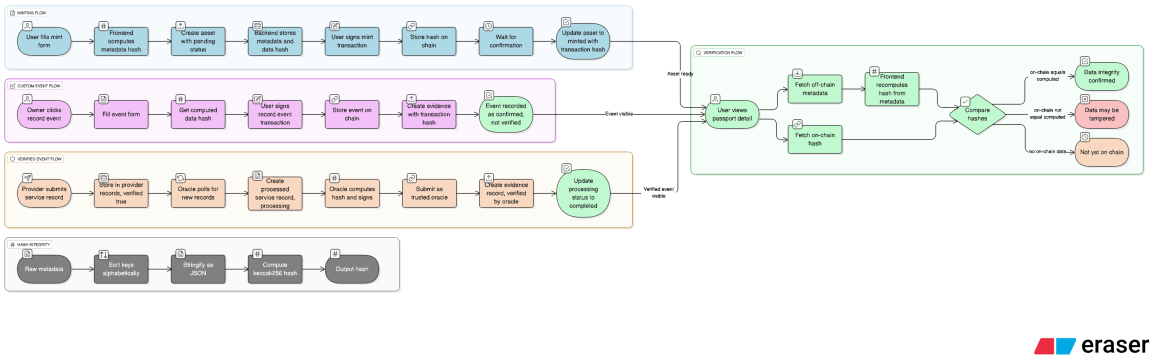
Frontend mendukung **graceful degradation** – tetap beroperasi dalam mode offline ketika backend tidak tersedia, dengan fitur terbatas pada operasi blockchain langsung.

2.6 Pattern dan Data Flow

2.6.1 Data Integrity Pattern

Integritas data dijamin melalui hash kriptografis:

1. Metadata di-hash menggunakan keccak256 dengan key sorting deterministik
2. Hash disimpan on-chain saat minting
3. Verifikasi dilakukan dengan membandingkan hash on-chain vs computed hash dari data off-chain
4. Mismatch mengindikasikan data telah dimodifikasi



Gambar 2.2: Alur Data dan Verifikasi

2.6.2 Authentication Flow

1. User menghubungkan wallet via RainbowKit
2. Frontend request nonce dari backend
3. User menandatangani message berisi nonce
4. Backend memverifikasi signature, issue JWT token (valid 7 hari)
5. Token digunakan untuk autentikasi request selanjutnya

Bab 3

Setup

3.1 Persiapan dan Konfigurasi

Prasyarat sistem yang diperlukan:

- Node.js v18+ atau Bun runtime
- PostgreSQL database
- Git untuk version control
- MetaMask atau wallet Web3 lainnya

Clone repository dan install dependencies:

```
git clone <repository-url>
cd veripass
```

3.2 Menjalankan Smart Contract

Navigasi ke direktori contracts dan jalankan local blockchain:

```
cd contracts
bun install
bun hardhat compile              # Compile kontrak
bun hardhat node                 # Jalankan local node (port 8545)
```

Pada terminal terpisah, deploy kontrak:

```
bun hardhat run scripts/deploy.ts --network localhost
bun hardhat run scripts/export-abi.ts    # Export ABI ke shared/
```

Konfigurasi MetaMask untuk testing lokal:

- RPC URL: `http://127.0.0.1:8545`
- Chain ID: 31337
- Test private key tersedia dari output Hardhat node

3.3 Menjalankan Oracle

Oracle memerlukan konfigurasi environment variable:

```
# Di file backend/.env
ORACLE_API_KEY=<min-32-char-key>
ORACLE_PRIVATE_KEY=0x<64-char-hex>
POLL_INTERVAL=30000
EVENT_REGISTRY_ADDRESS=<deployed-address>
```

Jalankan oracle worker:

```
cd backend
bun run dev:oracle
```

Oracle akan mendaftarkan alamatnya dan mulai polling untuk service record baru.

3.4 Menjalankan Backend

Setup database dan jalankan server:

```
cd backend
cp .env.example .env              # Salin dan edit konfigurasi
bun install
bun run db:push                   # Push schema ke database
bun run db:seed                   # (Opsional) Seed test data
bun run dev                       # Jalankan server (port 3000)
```

Environment variables yang diperlukan:

```
DB_URL=postgresql://user:pass@localhost:5432/veripass
JWT_SECRET=<min-32-char-secret>
FRONTEND_URL=http://localhost:5173
ASSET_PASSPORT_ADDRESS=<deployed-address>
EVENT_REGISTRY_ADDRESS=<deployed-address>
```

3.5 Menjalankan Frontend

Setup dan jalankan development server:

```
cd frontend
cp .env.example .env              # Salin dan edit konfigurasi
bun install
bun run dev                       # Jalankan server (port 5173)
```

Environment variables frontend:

```
VITE_API_URL=http://localhost:3000
VITE_WALLETCONNECT_PROJECT_ID=<project-id>
VITE_ASSET_PASSPORT_ADDRESS=<deployed-address>
VITE_EVENT_REGISTRY_ADDRESS=<deployed-address>
```

3.6 Urutan Eksekusi Keseluruhan

Untuk menjalankan sistem lengkap, ikuti urutan berikut:

1. **Database** – Pastikan PostgreSQL berjalan
2. **Blockchain** – Jalankan Hardhat node, deploy kontrak
3. **Backend** – Push schema, jalankan server
4. **Oracle** – Jalankan worker (opsional, untuk verified events)
5. **Frontend** – Jalankan dev server

Verifikasi instalasi dengan mengakses `http://localhost:5173` dan menghubungkan wallet.

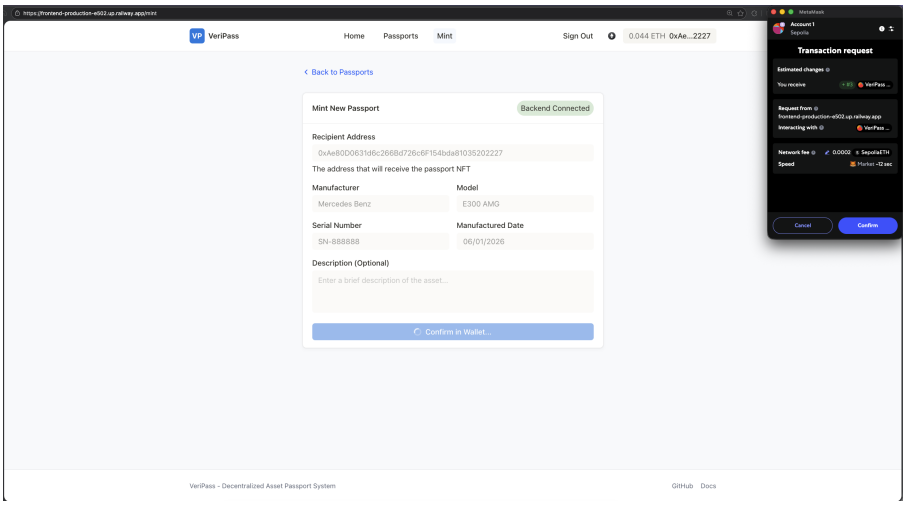
Bab 4

Penggunaan

4.1 Asset Minting

Proses minting passport aset baru:

1. Hubungkan wallet melalui tombol Connect pada header
2. Navigasi ke halaman Mint
3. Isi form dengan data aset:
    - Alamat penerima (recipient address)
    - Manufacturer dan model
    - Serial number
    - Tanggal manufaktur
    - Deskripsi (opsional)
4. Klik tombol Mint
5. Konfirmasi transaksi di wallet
6. Tunggu konfirmasi blockchain



Gambar 4.1: Form Minting Passport

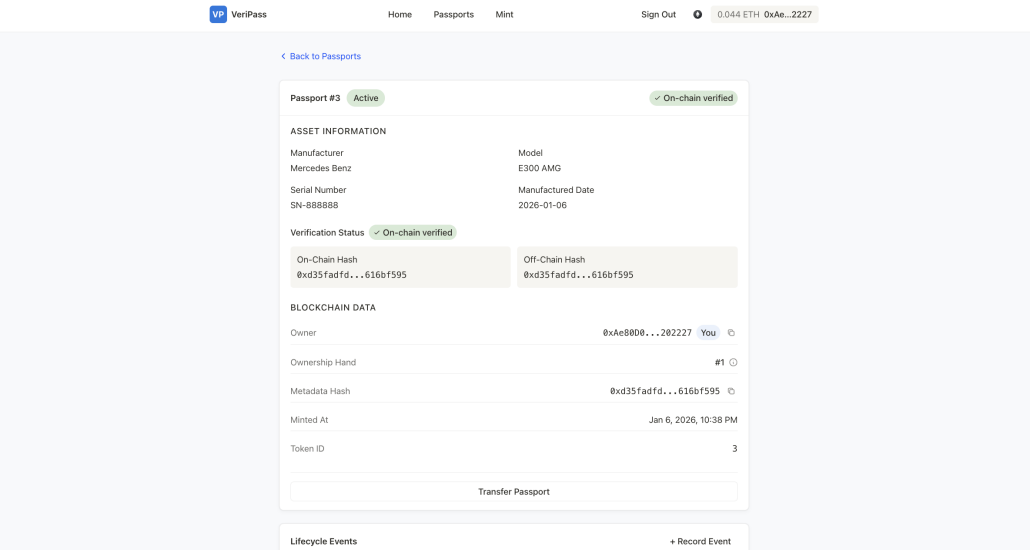
Sistem akan:

- Menghitung hash deterministik dari metadata
- Menyimpan metadata ke database dengan status PENDING
- Submit transaksi minting ke blockchain
- Update status ke MINTED setelah konfirmasi

4.2 Asset Viewing dan Verification

Melihat dan memverifikasi passport:

1. Navigasi ke halaman Passports untuk melihat daftar semua passport
2. Klik passport untuk melihat detail
3. Perhatikan **Verification Badge**:
    - **Verified** – Hash on-chain cocok dengan off-chain
    - **Pending** – Menunggu konfirmasi
    - **Mismatch** – Hash tidak cocok, data mungkin dimodifikasi



Gambar 4.2: Detail Passport dengan Verification Badge

Informasi yang ditampilkan pada halaman detail:

- Metadata aset (manufacturer, model, serial number)
- Status verifikasi dengan perbandingan hash
- Owner address saat ini
- Ownership hands (jumlah perpindahan)
- Timestamp minting
- Event timeline

4.3 Verified Event

Terdapat dua cara mencatat event pada passport:

4.3.1 Custom Event (User-Submitted)

Event yang dicatat langsung oleh pemilik aset:

1. Buka halaman detail passport
2. Klik tombol "Record Event"
3. Isi form event:
    - Tipe event (Custom)
    - Deskripsi
    - Tanggal event
    - Data tambahan (JSON)
4. Konfirmasi transaksi di wallet

Custom event bersifat **unverified** dan ditandai pada timeline.

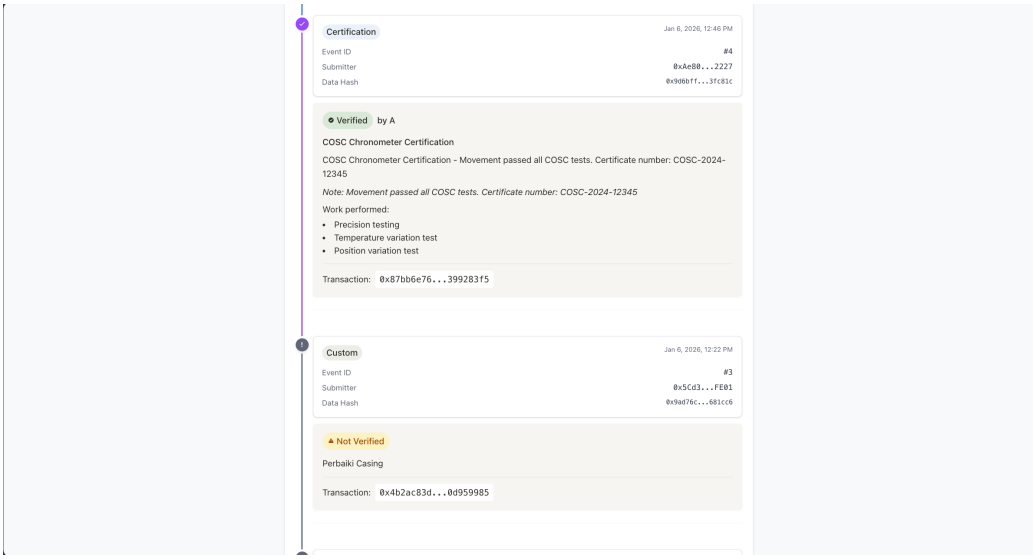
4.3.2 Oracle-Verified Event

Event yang diverifikasi oleh oracle dari service provider:

1. Service provider submit record via Provider API
2. Oracle secara otomatis polling record baru
3. Oracle memverifikasi dan menandatangani data
4. Event dicatat ke blockchain sebagai verified
5. Event muncul di timeline dengan badge verified

Event timeline mendukung filtering berdasarkan tipe:

- Maintenance
- Verification
- Warranty
- Certification
- Custom



Gambar 4.3: Event Timeline dengan Filter

Bab 5

Lain-lain

5.1 Catatan Keamanan

Aspek keamanan yang perlu diperhatikan:

5.1.1 Private Key Management

- **Jangan pernah** membagikan private key wallet
- Gunakan hardware wallet untuk aset bernilai tinggi
- Backup seed phrase di lokasi aman dan offline

5.1.2 Smart Contract Security

- Kontrak menggunakan OpenZeppelin library yang sudah diaudit
- Implementasi Pausable untuk emergency stop
- Access control dengan Ownable dan authorized minters pattern
- Input validation untuk mencegah zero address dan invalid hash

5.1.3 Backend Security

- JWT secret minimal 32 karakter
- Nonce expiry 5 menit untuk mencegah replay attack
- CORS dikonfigurasi hanya untuk frontend URL
- API key hashing untuk provider authentication

5.1.4 Data Integrity

- Selalu verifikasi hash on-chain sebelum mempercayai data off-chain
- Perhatikan warning jika terdapat hash mismatch
- Oracle signature memastikan keaslian verified events

5.2 Kesimpulan dan Saran

VeriPass menyediakan solusi verifikasi aset yang transparan dan tamper-proof dengan memanfaatkan teknologi blockchain. Sistem ini cocok untuk:

- Tracking aset bernilai tinggi (kendaraan, elektronik, barang koleksi)
- Sertifikasi dan maintenance record
- Proof of ownership dan transfer history
- Audit trail yang tidak dapat dimanipulasi

5.2.1 Saran Pengembangan

1. **Multi-chain Support** – Deploy ke multiple blockchain untuk redundansi
2. **IPFS Integration** – Simpan metadata lengkap di IPFS untuk desentralisasi penuh
3. **Mobile App** – Aplikasi mobile untuk scanning dan verifikasi cepat
4. **Provider Network** – Ekspansi jaringan service provider terverifikasi

Lampiran A

Source Code Smart Contract

A.1 AssetPassport.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/Pausable.sol";
import "./interfaces/IAssetPassport.sol";
import "./errors/VeriPassErrors.sol";

contract AssetPassport is ERC721, Ownable, Pausable, IAssetPassport {
    uint256 private _tokenIdCounter;
    mapping(uint256 => AssetInfo) private _assets;
    mapping(address => bool) private _authorizedMinters;
    mapping(uint256 => uint32) private _ownershipHands;

    constructor()
        ERC721("VeriPass Asset Passport", "VPASS")
        Ownable(msg.sender)
    {
        _tokenIdCounter = 1;
    }

    function mintPassport(
        address to,
        bytes32 metadataHash
    ) external override whenNotPaused returns (uint256 tokenId) {
        if (!_authorizedMinters[msg.sender] && msg.sender != owner()) {
            revert VeriPass_AssetPassport_NotAuthorizedMinter();
        }
        if (to == address(0)) revert VeriPass_ZeroAddress();
        if (metadataHash == bytes32(0))
            revert VeriPass_AssetPassport_InvalidMetadataHash();
        tokenId = _tokenIdCounter++;
        _safeMint(to, tokenId);
        _assets[tokenId] = AssetInfo({
            metadataHash: metadataHash,
            mintTimestamp: uint40(block.timestamp),
            isActive: true
        });

        emit PassportMinted(tokenId, to, metadataHash);

        return tokenId;
    }

    function getAssetInfo(
        uint256 tokenId
    ) external view override returns (AssetInfo memory info) {
        if (_ownerOf(tokenId) == address(0)) {
            revert VeriPass_AssetPassport_TokenDoesNotExist(tokenId);
```

```
        }
        return _assets[tokenId];
    }

    function getOwnershipHand(
        uint256 tokenId
    ) external view override returns (uint32 handCount) {
        if (_ownerOf(tokenId) == address(0)) {
            revert VeriPass_AssetPassport_TokenDoesNotExist(tokenId);
        }
        return _ownershipHands[tokenId];
    }

    function isAuthorizedMinter(
        address account
    ) external view override returns (bool isAuthorized) {
        return _authorizedMinters[account] || account == owner();
    }

    function nextTokenId() external view returns (uint256) {
        return _tokenIdCounter;
    }

    function addAuthorizedMinter(address minter) external override onlyOwner {
        if (minter == address(0)) revert VeriPass_ZeroAddress();
        _authorizedMinters[minter] = true;
        emit MinterUpdated(minter, true);
    }

    function removeAuthorizedMinter(
        address minter
    ) external override onlyOwner {
        _authorizedMinters[minter] = false;
        emit MinterUpdated(minter, false);
    }

    function deactivatePassport(uint256 tokenId) external override onlyOwner {
        if (_ownerOf(tokenId) == address(0)) {
            revert VeriPass_AssetPassport_TokenDoesNotExist(tokenId);
        }
        _assets[tokenId].isActive = false;
        emit PassportDeactivated(tokenId);
    }

    function pause() external override onlyOwner {
        _pause();
    }

    function unpause() external override onlyOwner {
        _unpause();
    }

    function _update(
        address to,
        uint256 tokenId,
        address auth
    ) internal override whenNotPaused returns (address previousOwner) {
        previousOwner = super._update(to, tokenId, auth);

        if (previousOwner == address(0)) {
```

```
            _ownershipHands[tokenId] = 1;
        } else if (previousOwner != to) {
            unchecked {
                _ownershipHands[tokenId] += 1;
            }
        }

        return previousOwner;
    }
}
```

Listing A.1: AssetPassport.sol

A.2 EventRegistry.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721.sol";
import "./interfaces/IEventRegistry.sol";
import "./interfaces/IAssetPassport.sol";
import "./errors/VeriPassErrors.sol";

contract EventRegistry is Ownable, IEventRegistry {
    uint256 private _eventIdCounter;

    mapping(uint256 => uint256[]) private _assetEvents;

    mapping(uint256 => LifecycleEvent) private _events;

    mapping(address => bool) private _trustedOracles;

    /// inject the asset passport contract
    IAssetPassport public assetPassport;

    constructor() Ownable(msg.sender) {
        _eventIdCounter = 1;
    }

    function recordEvent(
        uint256 assetId,
        EventType eventType,
        bytes32 dataHash
    ) external override returns (uint256 eventId) {
        if (dataHash == bytes32(0))
            revert VeriPass_EventRegistry_InvalidDataHash();

        _validateEventSubmission(assetId, msg.sender);

        eventId = _eventIdCounter++;

        _events[eventId] = LifecycleEvent({
            id: eventId,
            assetId: assetId,
            eventType: eventType,
            submitter: msg.sender,
            timestamp: uint40(block.timestamp),
            dataHash: dataHash
        });
```

```
46
47          _assetEvents[assetId].push(eventId);
48
49          emit EventRecorded(assetId, eventId, eventType, msg.sender, dataHash);
50
51          return eventId;
52      }
53
54      function recordVerifiedEvent(
55          uint256 assetId,
56          EventType eventType,
57          bytes32 dataHash,
58          bytes calldata oracleSignature
59      ) external override returns (uint256 eventId) {
60          if (!_trustedOracles[msg.sender]) {
61              revert VeriPass_EventRegistry_NotTrustedOracle(msg.sender);
62          }
63
64          if (dataHash == bytes32(0))
65              revert VeriPass_EventRegistry_InvalidDataHash();
66          if (oracleSignature.length == 0) {
67              revert VeriPass_EventRegistry_InvalidOracleSignature();
68          }
69
70          // Only allow verified event types (not CUSTOM - that's for user-
    submitted events)
71          if (eventType == EventType.CUSTOM) {
72              revert VeriPass_EventRegistry_InvalidEventType();
73          }
74
75          // verify that asset exists
76          if (address(assetPassport) != address(0)) {
77              try assetPassport.getAssetInfo(assetId) returns (
78                  AssetInfo memory
79              ) {} catch {
80                  revert VeriPass_EventRegistry_AssetNotFound(assetId);
81              }
82          }
83
84          eventId = _eventIdCounter++;
85
86          _events[eventId] = LifecycleEvent({
87              id: eventId,
88              assetId: assetId,
89              eventType: eventType,
90              submitter: msg.sender,
91              timestamp: uint40(block.timestamp),
92              dataHash: dataHash
93          });
94
95          _assetEvents[assetId].push(eventId);
96
97          emit EventRecorded(
98              assetId,
99              eventId,
100             eventType,
101             msg.sender,
102             dataHash
103         );
104
```

```
        return eventId;
    }

    function getEventsByAsset(
        uint256 assetId
    ) external view override returns (LifecycleEvent[] memory events) {
        uint256[] memory eventIds = _assetEvents[assetId];
        events = new LifecycleEvent[](eventIds.length);

        for (uint256 i = 0; i < eventIds.length; i++) {
            events[i] = _events[eventIds[i]];
        }

        return events;
    }

    function getEventsByType(
        uint256 assetId,
        EventType eventType
    ) external view override returns (LifecycleEvent[] memory events) {
        uint256[] memory eventIds = _assetEvents[assetId];

        uint256 matchCount = 0;
        for (uint256 i = 0; i < eventIds.length; i++) {
            if (_events[eventIds[i]].eventType == eventType) {
                matchCount++;
            }
        }

        events = new LifecycleEvent[](matchCount);
        uint256 currentIndex = 0;

        for (uint256 i = 0; i < eventIds.length; i++) {
            if (_events[eventIds[i]].eventType == eventType) {
                events[currentIndex] = _events[eventIds[i]];
                currentIndex++;
            }
        }

        return events;
    }

    function getEventCount(
        uint256 assetId
    ) external view override returns (uint256 count) {
        return _assetEvents[assetId].length;
    }

    function getEvent(
        uint256 eventId
    ) external view returns (LifecycleEvent memory lifecycleEvent) {
        if (_events[eventId].id == 0) {
            revert VeriPass_EventRegistry_EventNotFound(eventId);
        }
        return _events[eventId];
    }

    function setAssetPassport(address passportAddress) external onlyOwner {
        if (passportAddress == address(0)) revert VeriPass_ZeroAddress();
        assetPassport = IAssetPassport(passportAddress);
```

```
    }

    function addTrustedOracle(address oracle) external override onlyOwner {
        if (oracle == address(0)) revert VeriPass_ZeroAddress();
        if (_trustedOracles[oracle]) {
            revert VeriPass_EventRegistry_OracleAlreadyRegistered(oracle);
        }

        _trustedOracles[oracle] = true;
        emit OracleAdded(oracle);
    }

    function removeTrustedOracle(address oracle) external override onlyOwner {
        if (!_trustedOracles[oracle]) {
            revert VeriPass_EventRegistry_OracleNotRegistered(oracle);
        }

        _trustedOracles[oracle] = false;
        emit OracleRemoved(oracle);
    }

    function isTrustedOracle(
        address account
    ) external view override returns (bool isTrusted) {
        return _trustedOracles[account];
    }

    function _validateEventSubmission(
        uint256 assetId,
        address submitter
    ) internal view {
        // AssetPassport must be configured
        if (address(assetPassport) == address(0)) {
            revert VeriPass_EventRegistry_AssetNotFound(assetId);
        }

        // Check asset exists and is active
        AssetInfo memory info;
        try assetPassport.getAssetInfo(assetId) returns (
            AssetInfo memory _info
        ) {
            info = _info;
        } catch {
            revert VeriPass_EventRegistry_AssetNotFound(assetId);
        }

        if (!info.isActive) {
            revert VeriPass_AssetPassport_PassportInactive(assetId);
        }

        // Check submitter is the asset owner using IERC721.ownerOf
        address assetOwner = IERC721(address(assetPassport)).ownerOf(assetId);
        if (submitter != assetOwner) {
            revert VeriPass_EventRegistry_NotAssetOwner(assetId, submitter);
        }
    }
}
```

Listing A.2: EventRegistry.sol

Lampiran B

Database Schema

```
import {
  pgTable,
  serial,
  varchar,
  timestamp,
  text,
  bigint,
  boolean,
  jsonb,
  index,
  foreignKey,
} from "drizzle-orm/pg-core";

// ============================================================
// ASSETS TABLE
// ============================================================
// Note: `id` is the database primary key (auto-increment for internal use)
//       `assetId` is the blockchain asset ID that matches the smart contract

export const assets = pgTable(
  "assets",
  {
    id: serial("id").primaryKey(),

    // Must match smart contract
    assetId: bigint("asset_id", { mode: "number" }).notNull().unique(),
    dataHash: varchar("data_hash", { length: 66 }).notNull().unique(),

    // Asset metadata (gets hashed)
    manufacturer: varchar("manufacturer", { length: 255 }).notNull(),
    model: varchar("model", { length: 255 }).notNull(),
    serialNumber: varchar("serial_number", { length: 255 }).notNull(),
    manufacturedDate: varchar("manufactured_date", { length: 10 }), // YYYY-MM-DD
    description: text("description"),

    // Additional data (flexible)
    images: jsonb("images").$type<string[]>(), // Array of image URLs
    metadata: jsonb("metadata").$type<Record<string, unknown>>(), // Any extra fields

    // Mint status: PENDING (waiting for tx), MINTED (confirmed), FAILED (tx failed/cancelled)
    mintStatus: varchar("mint_status", { length: 20 }).default("PENDING").notNull(),
    txHash: varchar("tx_hash", { length: 66 }), // Transaction hash when minted

    // Tracking
    createdBy: varchar("created_by", { length: 42 }).notNull(), // User wallet address
    createdAt: timestamp("created_at").defaultNow().notNull(),
    mintedAt: timestamp("minted_at"),
  },
```

```
  (table) => [
    index("asset_id_idx").on(table.assetId),
    index("data_hash_idx").on(table.dataHash),
    index("mint_status_idx").on(table.mintStatus),
  ]
);

// ============================================================
// SERVICE PROVIDERS
// ============================================================

export const serviceProviders = pgTable(
  "service_providers",
  {
    id: serial("id").primaryKey(),
    providerId: varchar("provider_id", { length: 255 }).notNull().unique(),
    providerName: varchar("provider_name", { length: 255 }).notNull(),
    providerType: varchar("provider_type", { length: 50 }).notNull(), // manufacturer, service_center, inspector
    isTrusted: boolean("is_trusted").default(true).notNull(),
    // API key for provider authentication (hashed)
    apiKeyHash: varchar("api_key_hash", { length: 128 }),
    createdAt: timestamp("created_at").defaultNow().notNull(),
  },
  (table) => [index("service_providers_provider_id_idx").on(table.providerId)]
);

// ============================================================
// SERVICE RECORDS (Provider-submitted records)
// ============================================================
// Note: `id` is the database primary key (auto-increment for internal use)
//       `recordId` is the business identifier (e.g., "SR-001")
// This table receives records from external service providers via API

export const serviceRecordsProviderA = pgTable(
  "service_records_provider_a",
  {
    id: serial("id").primaryKey(),
    recordId: varchar("record_id", { length: 255 }).notNull().unique(),

    assetId: bigint("asset_id", { mode: "number" }).notNull(),
    providerId: varchar("provider_id", { length: 255 }).notNull(),

    // Event classification (provider specifies directly)
    eventType: varchar("event_type", { length: 20 }).notNull(), // MAINTENANCE, VERIFICATION, WARRANTY, CERTIFICATION
    eventName: varchar("event_name", { length: 255 }).notNull(), // "Service RAM", "Battery Replacement", etc.

    // Service details
    serviceDate: varchar("service_date", { length: 10 }).notNull(), // YYYY-MM-DD
    technicianName: varchar("technician_name", { length: 255 }),
    technicianNotes: text("technician_notes"),

    // Work details
    workPerformed: jsonb("work_performed").$type<string[]>(),
    partsReplaced: jsonb("parts_replaced").$type<
      Array<{ name: string; partNumber?: string; quantity?: number }>
    >(),
```

```
105
106     // Verification status from provider
107     verified: boolean("verified").default(true).notNull(),
108
109     createdAt: timestamp("created_at").defaultNow().notNull(),
110     updatedAt: timestamp("updated_at"),
111   },
112   (table) => [
113     index("service_records_provider_a_asset_id_idx").on(table.assetId),
114     index("service_records_provider_a_provider_id_idx").on(table.providerId),
115     foreignKey({
116       columns: [table.assetId],
117       foreignColumns: [assets.assetId],
118       name: "service_records_provider_a_asset_id_fk",
119     })
120       .onDelete("cascade")
121       .onUpdate("cascade"),
122   ]
123 );
124
125 // ============================================================
126 // PROCESSED SERVICE RECORDS (Oracle processing queue)
127 // ============================================================
128 // Tracks which service records have been processed by the oracle
129
130 export const processedServiceRecords = pgTable(
131   "processed_service_records",
132   {
133     id: serial("id").primaryKey(),
134     serviceRecordId: bigint("service_record_id", { mode: "number" }).notNull(),
135     providerId: varchar("provider_id", { length: 255 }).notNull(),
136
137     // Status: PENDING, PROCESSING, COMPLETED, FAILED
138     status: varchar("status", { length: 20 }).default("PENDING").notNull(),
139
140     // Result (populated after processing)
141     evidenceId: bigint("evidence_id", { mode: "number" }),
142     blockchainEventId: bigint("blockchain_event_id", { mode: "number" }),
143     txHash: varchar("tx_hash", { length: 66 }),
144     errorMessage: text("error_message"),
145
146     createdAt: timestamp("created_at").defaultNow().notNull(),
147     processedAt: timestamp("processed_at"),
148   },
149   (table) => [
150     index("processed_service_records_status_idx").on(table.status),
151     index("processed_service_records_service_record_id_idx").on(
152       table.serviceRecordId
153     ),
154     foreignKey({
155       columns: [table.serviceRecordId],
156       foreignColumns: [serviceRecordsProviderA.id],
157       name: "processed_service_records_service_record_id_fk",
158     })
159       .onDelete("cascade")
160       .onUpdate("cascade"),
161   ]
162 );
163
164 // ============================================================
```

```
// EVIDENCE TABLE (Event data)
// ============================================================

export const evidence = pgTable(
  "evidence",
  {
    id: serial("id").primaryKey(),

    // Link to asset
    assetId: bigint("asset_id", { mode: "number" }).notNull(),
    dataHash: varchar("data_hash", { length: 66 }).notNull().unique(),

    // Link to service record (for oracle-created evidence)
    serviceRecordId: bigint("service_record_id", { mode: "number" }),

    // Event details
    eventType: varchar("event_type", { length: 20 }).notNull(), // MAINTENANCE, VERIFICATION, WARRANTY, CERTIFICATION, CUSTOM
    eventDate: varchar("event_date", { length: 10 }), // YYYY-MM-DD
    providerName: varchar("provider_name", { length: 255 }),
    description: text("description"),

    // Raw event data (user-provided JSON)
    eventData: jsonb("event_data").$type<Record<string, unknown>>(),

    // Status: PENDING (oracle flow, not on-chain yet), CONFIRMED (recorded on-chain)
    status: varchar("status", { length: 20 }).default("PENDING").notNull(),

    // Verification tracking (for oracle-verified events)
    isVerified: boolean("is_verified").default(false).notNull(),
    verifiedBy: varchar("verified_by", { length: 42 }), // Oracle address
    blockchainEventId: bigint("blockchain_event_id", { mode: "number" }),
    txHash: varchar("tx_hash", { length: 66 }),

    // Tracking
    createdBy: varchar("created_by", { length: 42 }).notNull(), // User wallet address or oracle address
    createdAt: timestamp("created_at").defaultNow().notNull(),
    confirmedAt: timestamp("confirmed_at"),
    verifiedAt: timestamp("verified_at"),
  },
  (table) => [
    index("evidence_asset_id_idx").on(table.assetId),
    index("evidence_data_hash_idx").on(table.dataHash),
    index("evidence_status_idx").on(table.status),
    index("evidence_service_record_id_idx").on(table.serviceRecordId),
    foreignKey({
      columns: [table.assetId],
      foreignColumns: [assets.assetId],
      name: "evidence_asset_id_fk",
    })
      .onDelete("cascade")
      .onUpdate("cascade"),
    foreignKey({
      columns: [table.serviceRecordId],
      foreignColumns: [serviceRecordsProviderA.id],
      name: "evidence_service_record_id_fk",
    })
      .onDelete("set null")
```

```
      .onUpdate("cascade"),
  ]
);

// ============================================================
// AUTH NONCES (Web3 authentication)
// ============================================================

export const authNonces = pgTable(
  "auth_nonces",
  {
    id: serial("id").primaryKey(),
    address: varchar("address", { length: 42 }).notNull().unique(),
    nonce: varchar("nonce", { length: 66 }).notNull(),
    expiresAt: timestamp("expires_at").notNull(),
    createdAt: timestamp("created_at").defaultNow().notNull(),
  },
  (table) => [index("auth_nonces_address_idx").on(table.address)]
);

// ============================================================
// TYPE EXPORTS
// ============================================================

export type Asset = typeof assets.$inferSelect;
export type NewAsset = typeof assets.$inferInsert;

export type Evidence = typeof evidence.$inferSelect;
export type NewEvidence = typeof evidence.$inferInsert;

export type ServiceProvider = typeof serviceProviders.$inferSelect;
export type NewServiceProvider = typeof serviceProviders.$inferInsert;

export type ServiceRecordProviderA = typeof serviceRecordsProviderA.$inferSelect
    ;
export type NewServiceRecordProviderA =
  typeof serviceRecordsProviderA.$inferInsert;

export type ProcessedServiceRecord = typeof processedServiceRecords.$inferSelect
    ;
export type NewProcessedServiceRecord =
  typeof processedServiceRecords.$inferInsert;

export type AuthNonce = typeof authNonces.$inferSelect;
export type NewAuthNonce = typeof authNonces.$inferInsert;
```

Listing B.1: Database Schema (Drizzle ORM)