AES-256 Field Level Encryption Guide

Here is a complete guide to implementing field-level symmetric encryption for SQLite using Python with AES-256.

Core Implementation Strategy

The goal is to provide reliable, field-level encryption for sensitive data within a SQLite database. This approach focuses on straightforward key management and application-layer encryption to ensure security without compromising database performance.

Encryption Key Management: The primary encryption key is derived from a strong password stored as a simple environment variable, ensuring maximum reliability and data safety.

Field-Level Encryption: Encryption is applied to individual columns containing sensitive data, leaving non-sensitive data in plaintext for efficient querying and indexing.

---

1. Encryption Service

This service handles all cryptographic operations using the **AES-256-GCM** algorithm, which provides both confidentiality and authenticity. It derives a 256-bit (32-byte) key from the environment variable using PBKDF2.

Python

Python

```python
import os
import json
import base64
from Crypto.Cipher import AES
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Random import get_random_bytes
from Crypto.Hash import SHA256

class DataEncryption:
    """Simple, reliable field-level encryption service using AES-256-GCM"""

    def __init__(self):
        encryption_key = os.environ.get('BAKER_ENCRYPTION_KEY')
        if not encryption_key:
            raise ValueError("BAKER_ENCRYPTION_KEY environment variable not found") [cite: 6]

        # Use PBKDF2 to derive a 256-bit (32-byte) key from the environment variable
        salt = b'baker_ai_static_salt' # A static salt is used for consistency [cite: 8]
        self.key = PBKDF2(encryption_key.encode(), salt, dkLen=32, count=100000, hmac_hash_module

    def encrypt_field(self, data: str) -> str:
        """Encrypt a single field value using AES-256-GCM"""
        if data is None or data == "":
            return None [cite: 10]

        data_bytes = data.encode('utf-8')
        cipher = AES.new(self.key, AES.MODE_GCM)
        nonce = cipher.nonce
        ciphertext, tag = cipher.encrypt_and_digest(data_bytes)

        # Combine nonce, tag, and ciphertext for storage and encode in base64
        encrypted_payload = base64.urlsafe_b64encode(nonce + tag + ciphertext)
```

```
        return encrypted_payload.decode('utf-8')

    def decrypt_field(self, encrypted_data: str) -> str:
        """Decrypt a single field value"""
        if encrypted_data is None or encrypted_data == "":
            return None [cite: 11]

        try:
            encrypted_payload = base64.urlsafe_b64decode(encrypted_data.encode('utf-8'))

            # Extract the nonce, tag, and ciphertext
            nonce = encrypted_payload[:16]
            tag = encrypted_payload[16:32]
            ciphertext = encrypted_payload[32:]

            cipher = AES.new(self.key, AES.MODE_GCM, nonce=nonce)
            decrypted_bytes = cipher.decrypt_and_verify(ciphertext, tag)
            return decrypted_bytes.decode('utf-8')
        except (ValueError, KeyError, TypeError) as e:
            # Log error but don't crash - return None for corrupted data [cite: 11]
            print(f"Decryption error: {e}") [cite: 12]
            return None [cite: 12]

    def encrypt_json(self, data: dict) -> str:
        """Encrypt JSON data"""
        if not data:
            return None [cite: 12]
        json_str = json.dumps(data, ensure_ascii=False) [cite: 12]
        return self.encrypt_field(json_str) [cite: 12]

    def decrypt_json(self, encrypted_data: str) -> dict:
        """Decrypt JSON data"""
        if not encrypted_data:
            return {} [cite: 13]
        decrypted_str = self.decrypt_field(encrypted_data) [cite: 13]
        if not decrypted_str:
            return {} [cite: 13]
        try:
            return json.loads(decrypted_str) [cite: 13]
        except json.JSONDecodeError:
            return {} [cite: 14]

```

---

2. Database Schema

The database schema clearly marks encrypted fields with an

_enc suffix for identification. This separation allows sensitive PII to be encrypted while keeping non-sensitive metadata available for direct queries.

```sql
-- Simple schema with encrypted fields marked with _enc suffix [cite: 16]
CREATE TABLE Clients (
    id TEXT PRIMARY KEY,
    account_number TEXT UNIQUE NOT NULL,
    -- Encrypted PII fields [cite: 16]
    first_name_enc TEXT,
    last_name_enc TEXT,
    ssn_enc TEXT,
    phone_enc TEXT,
    email_enc TEXT,
    address_enc TEXT,  -- JSON string encrypted [cite: 16]
    -- Non-sensitive fields [cite: 16]
```

```
    risk_tolerance TEXT,
    investment_goals TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE MeetingNotes (
    id TEXT PRIMARY KEY,
    client_id TEXT NOT NULL REFERENCES Clients(id),
    -- Encrypted content [cite: 17]
    discussion_topics_enc TEXT,      -- JSON encrypted [cite: 17]
    recommendations_enc TEXT,        -- JSON encrypted [cite: 17]
    action_items_enc TEXT,          -- JSON encrypted [cite: 17]
    client_concerns_enc TEXT,       -- JSON encrypted [cite: 17]
    financial_details_enc TEXT,     -- JSON encrypted [cite: 17]
    -- Non-sensitive metadata [cite: 18]
    meeting_date DATE NOT NULL,
    meeting_type TEXT,
    ai_confidence_score INTEGER,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Simple audit log [cite: 19]
CREATE TABLE DataAccessLog (
    id TEXT PRIMARY KEY,
    table_name TEXT NOT NULL,
    record_id TEXT NOT NULL,
    action TEXT NOT NULL,  -- 'create', 'read', 'update' [cite: 19]
    user_id TEXT,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

```

---

3. Application Layer Repositories

Repositories abstract the encryption and decryption logic, allowing the main application to work with simple dictionaries without needing to know about the underlying cryptography.

Python

```python
import uuid
from datetime import datetime

class EncryptedClientRepository:
    """Repository for encrypted client data""" [cite: 21]

    def __init__(self, db_connection):
        self.db = db_connection
        self.crypto = DataEncryption() [cite: 21]

    def create_client(self, client_data: dict, user_id: str = None):
        """Create client with encrypted PII""" [cite: 21]
        encrypted_data = {
            'id': client_data['id'],
            'account_number': client_data['account_number'],
            'first_name_enc': self.crypto.encrypt_field(client_data.get('first_name')),
            'last_name_enc': self.crypto.encrypt_field(client_data.get('last_name')),
            'ssn_enc': self.crypto.encrypt_field(client_data.get('ssn')),
            'phone_enc': self.crypto.encrypt_field(client_data.get('phone')),
            'email_enc': self.crypto.encrypt_field(client_data.get('email')),
            'address_enc': self.crypto.encrypt_json(client_data.get('address', {})),
            'risk_tolerance': client_data.get('risk_tolerance'),
            'investment_goals': client_data.get('investment_goals')
```

```
        } [cite: 22, 23]

        self.db.execute("""
            INSERT INTO Clients (id, account_number, first_name_enc, last_name_enc,
                                ssn_enc, phone_enc, email_enc, address_enc,
                                risk_tolerance, investment_goals)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        """, tuple(encrypted_data.values())) [cite: 23, 24]

        self._log_access('Clients', client_data['id'], 'create', user_id) [cite: 24]

    def get_client(self, client_id: str, user_id: str = None) -> dict:
        """Get client with decrypted PII""" [cite: 25]
        cursor = self.db.execute("SELECT * FROM Clients WHERE id = ?", (client_id,)) [cite: 25]
        row = cursor.fetchone() [cite: 25]

        if not row:
            return None [cite: 25]

        client_data = dict(row) [cite: 25]

        # Decrypt sensitive fields [cite: 26]
        client_data['first_name'] = self.crypto.decrypt_field(client_data.pop('first_name_enc'))
        client_data['last_name'] = self.crypto.decrypt_field(client_data.pop('last_name_enc')) [c
        client_data['ssn'] = self.crypto.decrypt_field(client_data.pop('ssn_enc')) [cite: 26]
        client_data['phone'] = self.crypto.decrypt_field(client_data.pop('phone_enc')) [cite: 26]
        client_data['email'] = self.crypto.decrypt_field(client_data.pop('email_enc')) [cite: 26]
        client_data['address'] = self.crypto.decrypt_json(client_data.pop('address_enc')) [cite:

        self._log_access('Clients', client_id, 'read', user_id) [cite: 26]
        return client_data [cite: 27]

    def _log_access(self, table_name: str, record_id: str, action: str, user_id: str):
        """Simple audit logging""" [cite: 31]
        self.db.execute("""
            INSERT INTO DataAccessLog (id, table_name, record_id, action, user_id)
            VALUES (?, ?, ?, ?, ?)
        """, (str(uuid.uuid4()), table_name, record_id, action, user_id)) [cite: 31, 32]

```

---

4. Key Management Utility

A command-line helper script is provided for key generation, validation, and backup4. This script generates a strong password suitable for deriving the AES-256 key.

Python

```python
import secrets
import string
from pathlib import Path
import sys

class KeyManagement:
    """Key generation, backup and validation utilities""" [cite: 40]

    @staticmethod
    def generate_strong_password(length: int = 32) -> str:
        """Generate a strong password for key derivation""" [cite: 40]
        alphabet = string.ascii_letters + string.digits + "!@#$%^&*" [cite: 40]
        return ''.join(secrets.choice(alphabet) for _ in range(length)) [cite: 41]

    @staticmethod
```

```
    def setup_encryption_key(environment: str = "development"):
        """Interactive setup for encryption key""" [cite: 41]
        print(f"Setting up Baker AI encryption key for {environment} environment") [cite: 41]
        new_key = KeyManagement.generate_strong_password() [cite: 41]
        print(f"Generated new encryption key (password): {new_key[:4]}...{new_key[-4:]}")

        confirm = input("\nDo you want to use this key? (y/N): ").lower().strip() [cite: 42, 43]
        if confirm != 'y':
            print("Key setup cancelled.") [cite: 43]
            return None [cite: 43]

        KeyManagement.backup_encryption_key_to_file(new_key, environment) [cite: 43]

        print("\nIMPORTANT: Set this environment variable NOW:") [cite: 43]
        print(f'BAKER_ENCRYPTION_KEY={new_key}') [cite: 44]
        return new_key [cite: 44]

    @staticmethod
    def backup_encryption_key_to_file(key: str = None, environment: str = "production"):
        """Backup encryption key to secure file""" [cite: 45]
        if not key:
            key = os.environ.get('BAKER_ENCRYPTION_KEY') [cite: 45]
        if not key:
            print("No encryption key found to backup") [cite: 45]
            return False [cite: 45]

        backup_dir = Path("C:/SecureBackup/encryption_keys") if os.name == 'nt' else Path("/secur
        backup_dir.mkdir(parents=True, exist_ok=True) [cite: 46]

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S") [cite: 46]
        backup_file = backup_dir / f"baker_key_backup_{environment}_{timestamp}.txt" [cite: 46]

        try:
            with open(backup_file, 'w') as f:
                f.write(f"Key: {key}\n") [cite: 47]
            print(f"✓ Key backed up to: {backup_file}") [cite: 50]
            return True [cite: 50]
        except Exception as e:
            print(f"✗ Failed to backup key: {e}") [cite: 50]
            return False [cite: 50]

    @staticmethod
    def validate_encryption_key() -> bool:
        """Validate that encryption key works""" [cite: 51]
        try:
            crypto = DataEncryption() [cite: 51]
            test_data = "test_encryption_validation_" + str(uuid.uuid4()) [cite: 51]
            encrypted = crypto.encrypt_field(test_data) [cite: 51]
            decrypted = crypto.decrypt_field(encrypted) [cite: 51]

            if decrypted == test_data:
                print("✓ Encryption key is valid and working") [cite: 52]
                return True [cite: 52]
            else:
                print("✗ Encryption key validation failed - decryption mismatch") [cite: 52]
                return False [cite: 53]
        except Exception as e:
            print(f"✗ Encryption key error: {e}") [cite: 53]
            return False [cite: 53]

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python key_management.py [generate|validate|backup]") [cite: 55]
        sys.exit(1) [cite: 55]

    command = sys.argv[1].lower() [cite: 55]
    if command == "generate":
        env = sys.argv[2] if len(sys.argv) > 2 else "development" [cite: 55]
        KeyManagement.setup_encryption_key(env) [cite: 55]
    elif command == "validate":
        KeyManagement.validate_encryption_key() [cite: 55, 56]
```

```
    elif command == "backup":
        env = sys.argv[2] if len(sys.argv) > 2 else "production" [cite: 56]
        KeyManagement.backup_encryption_key_to_file(environment=env) [cite: 56]
    else:
        print("Unknown command. Use: generate, validate, or backup") [cite: 56, 57]

```

---

5. Final Implementation Instructions

Follow these phases to set up and deploy the encryption solution.

Phase 1: Initial Setup (Day 1)

1. **Install Dependency**: `pip install pycryptodome`
2. **Generate Encryption Key**:

```powershell
python key_management.py generate production
```

3. **Set Environment Variable (Windows)**: Run PowerShell as Administrator.

```powershell
# Use the key generated in the previous step
[System.Environment]::SetEnvironmentVariable("BAKER_ENCRYPTION_KEY", "your-generated-key-here"

```

4. **Validate Setup**:

```powershell
python key_management.py validate
```

Phase 2: Database & Application Setup (Day 2)

Run Database Schema: Execute the SQL script to create the tables in your database5.

Integrate Code: Implement the `DataEncryption` service and the `Encrypted...Repository` classes into your application.

Test Functionality: Write tests to confirm that creating, reading, and updating data through the repositories works as expected.

Phase 3: Production Deployment (Day 3)

Backup Encryption Key: Securely back up the production key password and salt (store in IT Glue)

```powershell
python key_management.py backup production

```

Set Production Environment Variable: Securely configure the `BAKER_ENCRYPTION_KEY` variable on your production server(s).

Deploy Application: Deploy the application with encryption enabled.

Monitor & Verify: Monitor audit logs for compliance and verify that data is correctly encrypted in the production database.

Security Checklist

- [ ] The encryption key password is long, random, and stored securely as an environment variable.
- [ ] The environment variable is set at the machine/system level, not the user level.

  [ ] The key and salt backup is stored in a secure, offline location with restricted access (e.g., a password manager or vault).

  [ ] Audit logging captures all access events for sensitive data.

  [ ] Encrypted database columns use the `_enc` suffix for clear identification.

  [ ] Error handling prevents application crashes on decryption failures.