Team Member 1: Database & Backend Specialist

Your Role

You are responsible for the database structure, user authentication logic, and backend operations.

Day 1 Tasks (6-8 hours)

Morning Session (3-4 hours)

1. Learn Database Basics

Concepts to understand: - What is SQLite and why it's perfect for this project - Database tables, columns, and relationships - Primary keys and foreign keys - SQL basic commands: CREATE, INSERT, SELECT, UPDATE, DELETE

Quick Tutorial:

```
# SQLite is built into Python - no installation needed!
import sqlite3

# Basic connection
conn = sqlite3.connect('records.db')
cursor = conn.cursor()

# Create a table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE NOT NULL,
        password TEXT NOT NULL,
        full_name TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
''')
conn.commit()
```

2. Design Your Database Structure

Create TWO related tables as required:

Table 1: users - For login system - id (Primary Key) - username - password (will be hashed) - full_name - created_at

Table 2: records - For student/inventory records - id (Primary Key) - user_id (Foreign Key - links to users) - title - description - category - date_added - status

Afternoon Session (3-4 hours)

3. Implement Your Code

File: database_manager.py

```
"""
Database Manager Module
Handles all database operations for the Smart Records System
Author: [Your Name] - Team Member 1
"""

import sqlite3
import hashlib
from datetime import datetime
from typing import Optional, List, Tuple

class DatabaseManager:
    """Manages all database operations"""

    def __init__(self, db_name: str = "smart_records.db"):
        """Initialize database connection"""
        self.db_name = db_name
        self.create_tables()

    def get_connection(self):
        """Create and return database connection"""
        return sqlite3.connect(self.db_name)

    def create_tables(self):
        """Create all necessary tables"""
        conn = self.get_connection()
        cursor = conn.cursor()

        # Users table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                username TEXT UNIQUE NOT NULL,
```

```
                password TEXT NOT NULL,
                full_name TEXT NOT NULL,
                created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
            )
        ''')

        # Records table
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS records (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                user_id INTEGER NOT NULL,
                title TEXT NOT NULL,
                description TEXT,
                category TEXT,
                date_added TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                status TEXT DEFAULT 'Active',
                FOREIGN KEY (user_id) REFERENCES users (id)
            )
        ''')

        conn.commit()
        conn.close()
        print("✓ Database tables created successfully")

    @staticmethod
    def hash_password(password: str) -> str:
        """Hash password using SHA-256"""
        return hashlib.sha256(password.encode()).hexdigest()

    # ============= USER AUTHENTICATION =============

    def create_user(self, username: str, password: str, full_name: str) -> Tuple[bool,
        """
        Create a new user account
        Returns: (success: bool, message: str)
        """
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            hashed_password = self.hash_password(password)

            cursor.execute('''
                INSERT INTO users (username, password, full_name)
                VALUES (?, ?, ?)
            ''', (username, hashed_password, full_name))

            conn.commit()
            conn.close()
            return True, "Account created successfully!"

        except sqlite3.IntegrityError:
```

```
            return False, "Username already exists!"
        except Exception as e:
            return False, f"Error: {str(e)}"

    def authenticate_user(self, username: str, password: str) -> Tuple[bool, Optional[i
        """
        Authenticate user login
        Returns: (success: bool, user_id: int or None, message: str)
        """
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            hashed_password = self.hash_password(password)

            cursor.execute('''
                SELECT id, full_name FROM users
                WHERE username = ? AND password = ?
            ''', (username, hashed_password))

            result = cursor.fetchone()
            conn.close()

            if result:
                return True, result[0], f"Welcome back, {result[1]}!"
            else:
                return False, None, "Invalid username or password!"

        except Exception as e:
            return False, None, f"Error: {str(e)}"

    # ============= CRUD OPERATIONS =============

    def create_record(self, user_id: int, title: str, description: str, category: str)
        """Create a new record"""
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            cursor.execute('''
                INSERT INTO records (user_id, title, description, category)
                VALUES (?, ?, ?, ?)
            ''', (user_id, title, description, category))

            conn.commit()
            conn.close()
            return True, "Record created successfully!"

        except Exception as e:
            return False, f"Error: {str(e)}"

    def read_all_records(self, user_id: Optional[int] = None) -> List[Tuple]:
```

```
        """
        Read all records
        If user_id provided, returns only that user's records
        Returns: List of tuples (id, title, description, category, date, status)
        """
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            if user_id:
                cursor.execute('''
                    SELECT id, title, description, category, date_added, status
                    FROM records WHERE user_id = ?
                    ORDER BY date_added DESC
                ''', (user_id,))
            else:
                cursor.execute('''
                    SELECT id, title, description, category, date_added, status
                    FROM records ORDER BY date_added DESC
                ''')

            results = cursor.fetchall()
            conn.close()
            return results

        except Exception as e:
            print(f"Error reading records: {e}")
            return []

    def update_record(self, record_id: int, title: str, description: str, category: str
        """Update an existing record"""
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            cursor.execute('''
                UPDATE records
                SET title = ?, description = ?, category = ?, status = ?
                WHERE id = ?
            ''', (title, description, category, status, record_id))

            conn.commit()
            conn.close()

            if cursor.rowcount > 0:
                return True, "Record updated successfully!"
            else:
                return False, "Record not found!"

        except Exception as e:
            return False, f"Error: {str(e)}"
```

```
    def delete_record(self, record_id: int) -> Tuple[bool, str]:
        """Delete a record"""
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            cursor.execute('DELETE FROM records WHERE id = ?', (record_id,))

            conn.commit()
            conn.close()

            if cursor.rowcount > 0:
                return True, "Record deleted successfully!"
            else:
                return False, "Record not found!"

        except Exception as e:
            return False, f"Error: {str(e)}"

    def search_records(self, user_id: int, search_term: str) -> List[Tuple]:
        """Search records by title or description"""
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            search_pattern = f"%{search_term}%"

            cursor.execute('''
                SELECT id, title, description, category, date_added, status
                FROM records
                WHERE user_id = ? AND (title LIKE ? OR description LIKE ?)
                ORDER BY date_added DESC
            ''', (user_id, search_pattern, search_pattern))

            results = cursor.fetchall()
            conn.close()
            return results

        except Exception as e:
            print(f"Error searching records: {e}")
            return []

    # ============= REPORT GENERATION DATA =============

    def get_summary_stats(self, user_id: int) -> dict:
        """Get summary statistics for reports"""
        try:
            conn = self.get_connection()
            cursor = conn.cursor()

            # Total records
            cursor.execute('SELECT COUNT(*) FROM records WHERE user_id = ?', (user_id,)
```

```
            total = cursor.fetchone()[0]

            # Active records
            cursor.execute('SELECT COUNT(*) FROM records WHERE user_id = ? AND status =
            active = cursor.fetchone()[0]

            # Records by category
            cursor.execute('''
                SELECT category, COUNT(*)
                FROM records WHERE user_id = ?
                GROUP BY category
            ''', (user_id,))
            by_category = cursor.fetchall()

            conn.close()

            return {
                'total': total,
                'active': active,
                'inactive': total - active,
                'by_category': dict(by_category)
            }

        except Exception as e:
            print(f"Error getting stats: {e}")
            return {'total': 0, 'active': 0, 'inactive': 0, 'by_category': {}}


# ============= TESTING CODE =============
if __name__ == "__main__":
    # Test the database manager
    db = DatabaseManager()

    # Test 1: Create a user
    print("\n=== Test 1: Create User ===")
    success, msg = db.create_user("testuser", "password123", "Test User")
    print(msg)

    # Test 2: Authenticate user
    print("\n=== Test 2: Authenticate User ===")
    success, user_id, msg = db.authenticate_user("testuser", "password123")
    print(msg)

    if success:
        # Test 3: Create records
        print("\n=== Test 3: Create Records ===")
        db.create_record(user_id, "First Record", "This is a test", "General")
        db.create_record(user_id, "Second Record", "Another test", "Important")
        print("Records created!")

        # Test 4: Read all records
        print("\n=== Test 4: Read Records ===")
```

```
        records = db.read_all_records(user_id)
        for record in records:
            print(f"ID: {record[0]}, Title: {record[1]}, Category: {record[3]}")

        # Test 5: Get statistics
        print("\n=== Test 5: Statistics ===")
        stats = db.get_summary_stats(user_id)
        print(f"Total records: {stats['total']}")
        print(f"Active: {stats['active']}")
        print(f"By category: {stats['by_category']}")
```

Day 2 Tasks (4-6 hours)

Morning Session (2-3 hours)

4. Test Your Code Thoroughly

Run the test code at the bottom of your file:

```
python database_manager.py
```

Expected output: - User created successfully - Login successful - Records created - Records retrieved - Statistics displayed

5. Add Security Features

Protect against SQL Injection: Your code already uses parameterized queries (the `?` placeholders), which prevents SQL injection.

Test SQL Injection Protection:

```
# Try to break your code with malicious input
# This SHOULD fail safely:
evil_username = "admin' OR '1'='1"
evil_password = "' OR '1'='1"

success, user_id, msg = db.authenticate_user(evil_username, evil_password)
print(msg)  # Should say "Invalid username or password"
```

Afternoon Session (2-3 hours)

6. Integration with Team

- Share your `database_manager.py` file with the team

- Help Team Member 2 (GUI) understand how to import and use your functions
- Test with Team Member 3 to ensure CRUD operations work

7. Documentation

Create a `DATABASE_README.md` file:

```
# Database Documentation

## Database Structure

### Users Table
- id: Primary key
- username: Unique login name
- password: SHA-256 hashed password
- full_name: User's display name
- created_at: Account creation timestamp

### Records Table
- id: Primary key
- user_id: Foreign key to users table
- title: Record title
- description: Detailed description
- category: Classification
- date_added: Creation timestamp
- status: Active/Inactive

## How to Use

### Import the module:
```python
from database_manager import DatabaseManager

db = DatabaseManager()
```

Create a user:

```
success, message = db.create_user("username", "password", "Full Name")
```

Login:

```
success, user_id, message = db.authenticate_user("username", "password")
```
```

CRUD Operations:

```
# Create
db.create_record(user_id, "Title", "Description", "Category")

# Read
records = db.read_all_records(user_id)

# Update
db.update_record(record_id, "New Title", "New Desc", "New Category", "Active")

# Delete
db.delete_record(record_id)
```

Security Features

- Password hashing using SHA-256
- SQL injection protection via parameterized queries
- Input validation ```

Checklist

Day 1

- ☐ Understand SQLite basics
- ☐ Design database structure (2 tables with relationship)
- ☐ Write database_manager.py
- ☐ Implement user authentication
- ☐ Implement CRUD operations
- ☐ Test basic functionality

Day 2

- ☐ Test all functions thoroughly
- ☐ Verify SQL injection protection

- ☐ Create documentation
- ☐ Share file with team
- ☐ Help integrate with GUI (Team Member 2)
- ☐ Prepare your part of presentation

Things to Search and Learn

1. **SQLite Tutorial**: "Python SQLite tutorial for beginners"
2. **Password Hashing**: "Why hash passwords in Python"
3. **SQL Injection**: "What is SQL injection and how to prevent it"
4. **Foreign Keys**: "SQLite foreign key relationships"
5. **Try-Except**: "Python error handling best practices"

Common Issues & Solutions

Issue: "database is locked" **Solution**: Always close connections: `conn.close()`

Issue: "no such table" **Solution**: Run `create_tables()` first

Issue: "UNIQUE constraint failed" **Solution**: Username already exists, choose different one

Your Contribution (for presentation)

"I built the database backend that stores all our data securely. I created two tables - one for users with hashed passwords, and one for records. I also made sure we're protected against SQL injection attacks using parameterized queries."

Mark Breakdown for Your Work

- Database Integration & Accuracy: 3 marks
- Login & Authentication Logic: 2 marks
- Code Quality & Documentation: 1 mark (shared)
- Total: 6 marks (40% of project)