

University Database Management System - Technical Documentation

Table of Contents

Project Structure Overview

Application Layer

Core Application Files

CRUD Module

Academic Module

Performance Module

Utility Modules

Database Layer

Database/connection.py

Database/database.py

Database Schemas

Original Schema (8 Tables)

Extended Schema (6 Additional Tables)

Data Flow

Request-Response Flow

Error Propagation

Assignment Requirements Mapping

Part I: Database Schema Requirements

Part II: Python Application Requirements

Deliverables Checklist

Conclusion

University Database Management System - Technical Documentation

Note: This project is available on GitHub. Visit the repository for complete source code code,better documentation viewing,version control and updates.

Table of Contents

1. [Project Structure Overview](#)
2. [Application Layer](#)
3. [Database Layer](#)
4. [Database Schemas](#)
5. [Data Flow](#)
6. [Assignment Requirements Mapping](#)

Project Structure Overview

```
university-db-project-new/
├── Application/           # PyQt6 Desktop Application
│   ├── main.py           # Entry point, database initialization
│   ├── config.py         # Environment & resource path management
│   ├── requirements.txt   # Python dependencies
│   ├── GUI/              # User interface modules
│   │   └── main_window.py # Main window with stacked navigation
```

```
| | | └─ crud/           # CRUD operations (13 files)
| | | └─ academic/      # Instructor assignment, reservations (3 files)
| | | └─ performance/   # Marks & attendance entry (3 files)
| | | └─ queries/       # SQL query execution (2 files)
| | | └─ results/       # Results processing (2 files)
| | | └─ audit/         # Audit log viewer (2 files)
| | | └─ settings/      # App settings (2 files)
| | | └─ screens/       # Navigation screens (6 files)
| | | └─ styling/       # QSS themes (24 files)
| | |   └─ UI/          # Qt Designer .ui files
| | └─ UTILS/          # Utilities
| |   └─ log.py         # Logging setup
| |   └─ constriants.py # Input validation
| |     └─ screen_enum.py # Screen enumeration
| └─ Translations/     # i18n files (English, French, Arabic)
└─ Database/          # Database access layer
    └─ connection.py   # Connection pool (126 lines)
    └─ database.py      # 92+ CRUD & query functions (1291 lines)
    └─ Original_Schema/ # Base schema SQL
        └─ CREATE_TABLES.sql # 8 table definitions
        └─ INSERT_TABLES.sql # Sample data
        └─ Qureying.sql      # Complex queries
        └─ USING_SQL_FUNCTIONS.sql
        └─ USING_TRIGGERS.sql
        └─ USING_TRANSACTION.sql
    └─ Extende_Schema/    # Extended features
        └─ new_tables.sql   # 6 additional tables
        └─ functions.sql    # 15+ SQL functions
        └─ audit_triggers.sql # Audit logging triggers
        └─ project_queries.sql # Assignment queries
        └─ INSERT_NEW_DATA.sql # Extended data
└─ Documentation/       # Project documentation
    └─ Lab_Reports.md    # Comprehensive technical docs
    └─ Current_ERD.png   # Original ERD diagram
    └─ Extended_ERD.png  # Extended ERD with new tables
    └─ ...
└─ Tests/               # Test files
└─ Executable/          # Built executable
└─ logs/                # Application logs
```

Application Layer

Core Application Files

[main.py](#)

Purpose: Application entry point that orchestrates startup sequence and event loop.

Key Functions: - `initialize_database()` : Parses connection config and initializes pool - `start()` : Main startup sequence

Startup Flow: 1. Setup logging (`setup_logging()`) 2. Load database configuration from `.env` file 3. Initialize connection pool with Neon credentials 4. Create QApplication instance 5. Load user language preference 6. Apply Fusion style and QSS stylesheet 7. Create and show MainWindow 8. Start Qt event loop (`exec()`)

Error Handling: Shows critical error dialog and exits if database connection fails.

[config.py](#)

Purpose: Environment configuration and resource path resolution for development and PyInstaller builds.

Key Functions:

`get_resource_path(relative_path)` - Resolves paths for both development and PyInstaller frozen executables - Uses `sys._MEIPASS` when frozen, otherwise uses `__file__` directory

`load_env_file()` - Searches for `.env` in multiple locations (internal bundle, external directory) - Loads environment variables using python-dotenv if available

`get_database_config()` - Parses `DATABASE_URL` or `NEON_URI` environment variable - Extracts components: database, user, password, host, port, sslmode - Returns dictionary for connection pool initialization

Example Connection String:

DATABASE_URL=postgresql://user:pass@ep-example.us-east-2.aws.neon.tech:5432/neondb?sslmode=require

GUI/main_window.py

Purpose: Main application window with stacked widget navigation system.

Architecture: Uses `QStackedWidget` to manage multiple screens with back navigation.

Screen Management (224 lines): - Home screen with 6 navigation buttons - CRUD menu with 13 entity screens - Academic management (instructor assignment, reservations) - Performance tracking (marks, attendance) - Query execution and results viewing - Audit log viewer and settings

Navigation Pattern:

```
def _navigate(self, screen_index, log_message):
    self.stacked_widget.setCurrentIndex(screen_index)
    logger.info(log_message)
```

Signal Connections: Each screen emits `go_back` signal connected to return to parent menu.

CRUD Module

GUI/crud/base_crud.py

Purpose: Base class providing reusable CRUD interface for all entities (424 lines).

Architecture: - Loads `CRUD.ui` file for consistent layout - Dynamically generates input forms based on entity headers - Implements table with search, add, update, delete operations

Key Methods:

- `_setup_table()` : Configures QTableWidgetItem with column headers and properties
- `_build_form()` : Dynamically creates QLabel + QLineEdit pairs for each field, marks ID fields as read-only
- `populate_table(rows)` : Fills table with database records
- `_add_new_to_table()` : Clears selection, enables form for new entry
- `_save_new()` : - Validates input using `constraints.validate_all_fields()` - Calls `db_operations['add'](*row_data)` - Inserts new row into UI table
- `_update_row()` : Validates and calls `db_operations['update']` with all fields
- `_delete_row()` : Calls `db_operations['delete'](id_value)` and removes from table
- `_search_table()` : Filters displayed rows in real-time as user types

Database Operation Dictionary:

```
db_operations = {
    'add': db.add_student,
    'update': db.update_student,
    'delete': db.delete_student
}
```

Validation Flow: Form data → `constraints.validate_all_fields()` → returns `(is_valid, errors)` → proceed or show error

Entity-Specific CRUD Implementations

Each entity inherits from `BaseCRUD` and provides: - Column headers list - Database operation mappings - Initial data population via `db.get_all_*` functions

Example: [student_crud.py](#):

```
headers = ["Student_ID", "Last_Name", "First_Name", "DOB", "Address",
           "City", "Zip_Code", "Phone", "Fax", "Email", "Group_ID", "Section_ID"]
db_operations = {
    'add': db.add_student,
    'update': db.update_student,
```

```
        'delete': db.delete_student
    }
    super().__init__(headers, db_operations)
    self.populate_table(db.get_all_students())
```

All CRUD Screens: - student_crud.py , instructor_crud.py , course_crud.py , department_crud.py - room_crud.py , enrollment_crud.py , mark_crud.py , exam_crud.py - activity_crud.py , attendance_crud.py , reservation_crud.py

Academic Module

[GUI/academic/assign_instructor.py](#)

Purpose: Assign instructors to courses with department consistency validation (8822 bytes).

Features: - Department-aware filtering of courses and instructors - Visual display of current assignments - Prevents cross-department assignments

[GUI/academic/manage_reservations.py](#)

Purpose: Classroom reservation management with conflict detection (18913 bytes).

Key Features: - Calendar-based date selection - Room availability checking via db.check_room_availability() - Time slot validation (start < end, hours ≥ 1) - Conflict resolution suggestions

Conflict Detection SQL:

```
SELECT COUNT(*) FROM reservation
WHERE building = %s AND roomno = %s AND reserv_date = %s
AND NOT (end_time <= %s OR start_time >= %s)
```

Performance Module

[GUI/performance/marks_entry.py](#)

Purpose: Bulk entry and management of student examination marks.

Workflow: 1. Select course and exam 2. Load enrolled students via db.get_students_with_marks(course_id, exam_id) 3. Display students with current marks in editable table 4. User enters/updates mark values (validated 0-20) 5. Save all changes via db.save_bulk_marks() 6. Highlight failing grades based on disqualifying threshold

[GUI/performance/attendance_entry.py](#)

Purpose: Record student attendance for activities.

Features: - Activity selection (lecture, tutorial, practical) - Student list with present/absent toggles - “Mark all present/absent” bulk operations - Attendance percentage calculation

Utility Modules

[UTILS/log.py](#)

Purpose: Centralized logging configuration.

Configuration: - Rotating file handler to prevent unbounded log growth - Console handler for development - Format: %(asctime)s - %(name)s - %(levelname)s - %(message)s - Log file: logs/app.log

[UTILS/constraints.py](#)

Purpose: Input validation constraints for form data.

Validation Rules: - Email format validation (regex) - Phone number format (10 digits) - Date range validation - Mark value range (0-20) - Required field checking

Database Layer

[Database/connection.py](#)

Purpose: PostgreSQL connection pool management using psycopg2 (126 lines).

Class: DatabaseConnection

`initialize_pool(database, user, password, host, port, minconn=1, maxconn=10, **kwargs)` - Creates `psycopg2.pool.SimpleConnectionPool` - Sets `sslmode='require'` for Neon security - Stores pool as class variable `_connection_pool`

`get_connection()` - Retrieves connection from pool via `getconn()` - Raises exception if pool not initialized - Returns psycopg2 connection object

`return_connection(connection)` - Returns connection to pool via `putconn()` - Called in finally blocks to ensure cleanup

`close_all_connections()` - Calls `closeall()` on pool - Used during application shutdown

Function: execute_query(query, params=None, fetch=False)

Generic query executor with automatic connection management:

```
    connection = DatabaseConnection.get_connection()
    cursor = connection.cursor()
    cursor.execute(query, params or ())
    if fetch:
        result = cursor.fetchall()
        connection.commit()
        return result
    else:
        connection.commit()
```

Error Handling: Rolls back transaction on exception, always closes cursor and returns connection.

[Database/database.py](#)

Purpose: Complete database access layer with 92+ functions (1291 lines).

Organization: Functions grouped by entity type with consistent naming pattern.

Function Categories:

- 1. Student Operations (Lines 18-122)
 - `get_all_students()` , `get_students_by_filters()` , `get_students_with_marks()`
 - `add_student()` , `update_student()` , `delete_student()`
- 2. Instructor Operations (Lines 125-176)
 - `get_all_instructors()` , `add_instructor()` , `update_instructor()` , `delete_instructor()`
- 3. Course Operations (Lines 179-231)
 - `get_all_courses()` , `add_course()` , `update_course()` , `delete_course()`
- 4. Department Operations (Lines 235-285)
- 5. Room Operations (Lines 289-339)
- 6. Section & Group Operations (Lines 343-463)
- 7. Reservation Operations (Lines 467-641)
 - Includes `check_room_availability()` for conflict detection
 - `get_courses_for_reservation()` , `get_instructors_for_reservation()` for UI dropdowns
- 8. Enrollment Operations (Lines 645-698)
- 9. Mark Operations (Lines 702-850)
 - `save_bulk_marks()` for batch mark entry
 - `update_bulk_marks()` for performance tracking
- 10. Exam Operations (Lines 853-945)
- 11. Activity Operations (Lines 948-1050)
- 12. Attendance Operations (Lines 1053-1160)
- 13. Audit Operations (Lines 1163-1220)
- 14. Query & Analytics Functions (Lines 1223-1291)
 - `execute_custom_query()` for SQL query interface
 - `get_students_by_group()` , `get_students_by_section()`

- `get_instructor_timetable()` , `get_student_timetable()`
- `calculate_disqualifying_marks()` , `get_students_who_passed()`

Pattern Example:

```
def add_student(last_name, first_name, dob, address, city, zip_code,
               phone, fax, email, group_id, section_id):

    try:
        eq("INSERT INTO student (...) VALUES (%s, %s, ...);",
           (last_name, first_name, ...))
        logger.info(f"Added student {first_name} {last_name}")
        return True
    except Exception as e:
        logger.error(f"Error adding student: {e}")
        return False
```

Parameterized Queries: All functions use parameterized queries (`%s` placeholders) to prevent SQL injection.

Database Schemas

Original Schema (8 Tables)

Located in `Database/Original_Schema/`

[CREATE TABLES.sql](#)

1. Department (Foundational entity)

```
CREATE TABLE Department (
    Department_id INTEGER PRIMARY KEY,
    name VARCHAR(25) NOT NULL UNIQUE
);
```

2. Student (Personal information)

```
CREATE TABLE Student (
    Student_ID INTEGER PRIMARY KEY,
    Last_Name VARCHAR(25) NOT NULL,
    First_Name VARCHAR(25) NOT NULL,
    DOB DATE NOT NULL,
    Address VARCHAR(50),
    City VARCHAR(25),
    Zip_Code VARCHAR(9),
    Phone VARCHAR(10),
    Fax VARCHAR(10),
    Email VARCHAR(100)
);
```

3. Course (Composite key with Department)

```
CREATE TABLE Course (
    Course_ID INT4 NOT NULL,
    Department_ID INT4 NOT NULL,
    name VARCHAR(60) NOT NULL,
    Description VARCHAR(1000),
    PRIMARY KEY (Course_ID, Department_ID),
    FOREIGN KEY (Department_ID) REFERENCES Department(Department_id)
    ON UPDATE RESTRICT ON DELETE RESTRICT
);
```

4. Instructor (With rank check constraint)

```
CREATE TABLE Instructor (
    Instructor_ID INTEGER PRIMARY KEY,
    Department_ID INTEGER NOT NULL,
    Last_Name VARCHAR(25) NOT NULL,
    First_Name VARCHAR(25) NOT NULL,
    Rank VARCHAR(25) CHECK (Rank IN ('Substitute', 'MCB', 'MCA', 'PROF')),
    Phone VARCHAR(10),
    Fax VARCHAR(10),
    Email VARCHAR(100),
    FOREIGN KEY (Department_ID) REFERENCES Department(Department_id)
);
```

5. Room (Composite key: Building + RoomNo)

```
CREATE TABLE Room (
    Building VARCHAR(1),
    RoomNo VARCHAR(10),
    Capacity INTEGER CHECK (Capacity > 1),
    PRIMARY KEY (Building, RoomNo)
);
```

6. Reservation (Central scheduling table)

```
CREATE TABLE Reservation (
    Reservation_ID INTEGER PRIMARY KEY,
    Building VARCHAR(1) NOT NULL,
    RoomNo VARCHAR(10) NOT NULL,
    Course_ID INTEGER NOT NULL,
    Department_ID INTEGER NOT NULL,
    Instructor_ID INTEGER NOT NULL,
    Reserv_Date DATE NOT NULL DEFAULT CURRENT_DATE,
    Start_Time TIME NOT NULL DEFAULT CURRENT_TIME,
    End_Time TIME NOT NULL DEFAULT '23:00:00',
    Hours_Number INTEGER NOT NULL CHECK (Hours_Number >= 1),
    FOREIGN KEY (Building, RoomNo) REFERENCES Room(Building, RoomNo),
    FOREIGN KEY (Course_ID, Department_ID) REFERENCES Course(Course_ID, Department_ID),
    FOREIGN KEY (Instructor_ID) REFERENCES Instructor(Instructor_ID),
    CHECK (Start_Time < End_Time)
);
```

7. Enrollment (Many-to-many between Student and Course)

```
CREATE TABLE Enrollment (
    Student_ID INT NOT NULL,
    Course_ID INT NOT NULL,
    Department_ID INT NOT NULL,
    Enrollment_Date DATE NOT NULL,
    PRIMARY KEY (Student_ID, Course_ID, Department_ID),
    FOREIGN KEY (Student_ID) REFERENCES Student(Student_ID),
    FOREIGN KEY (Course_ID, Department_ID) REFERENCES Course(Course_ID, Department_ID)
);
```

8. Mark (Student grades)

```
CREATE TABLE Mark (
    Mark_ID SERIAL PRIMARY KEY,
    Student_ID INTEGER NOT NULL,
    Course_ID INTEGER NOT NULL,
    Department_ID INTEGER NOT NULL,
    Mark_Value NUMERIC(4,2) NOT NULL CHECK (Mark_Value >= 0 AND Mark_Value <= 20),
    Mark_Date DATE NOT NULL,
```

```
FOREIGN KEY (Student_ID) REFERENCES Student(Student_ID),
FOREIGN KEY (Course_ID, Department_ID) REFERENCES Course(Course_ID, Department_ID)
);
```

Extended Schema (6 Additional Tables)

Located in `Database/Extende_Schema/`

[new_tables.sql](#)

Enums:

```
CREATE TYPE Activity_type AS ENUM('lecture', 'toturiol', 'practical');
CREATE TYPE status AS ENUM('present', 'absent');
```

1. Activity (Tracks course activities)

```
CREATE TABLE activity (
    activity_id BIGSERIAL PRIMARY KEY,
    Activity_Type Activity_type NOT NULL,
    reservation_id INT NOT NULL,
    course_id INT NOT NULL,
    department_id INT NOT NULL,
    FOREIGN KEY (course_id, department_id) REFERENCES course(course_id, department_id),
    FOREIGN KEY (reservation_id) REFERENCES reservation(reservation_id)
);
```

2. Exam (Examination scheduling)

```
CREATE TABLE exam (
    exam_id BIGSERIAL PRIMARY KEY,
    duration INT NOT NULL, -- minutes
    exam_type VARCHAR(25) NOT NULL,
    course_id INT NOT NULL,
    department_id INT NOT NULL,
    FOREIGN KEY (course_id, department_id) REFERENCES course(course_id, department_id)
);
```

3. Attendance_to_Activities (Student attendance tracking)

```
CREATE TABLE attendance_to_activities (
    student_id INT,
    activity_id INT,
    attendance_date DATE NOT NULL,
    statu status NOT NULL CHECK (statu IN ('present', 'absent')),
    PRIMARY KEY (student_id, activity_id, attendance_date),
    FOREIGN KEY (student_id) REFERENCES student(student_id) ON DELETE CASCADE,
    FOREIGN KEY (activity_id) REFERENCES activity(activity_id) ON DELETE CASCADE
);
```

4. Student Attributes (Groups and Sections added to Student table)

```
ALTER TABLE student ADD COLUMN group_id INT NOT NULL;
ALTER TABLE student ADD COLUMN section_id VARCHAR(1) NOT NULL;
```

5. mark_Audit_Log (Audit trail for marks)


```
CREATE TABLE mark_Audit_Log (  
    LogID SERIAL PRIMARY KEY,  
    OperationType VARCHAR(50) NOT NULL,  
    OperationTime TIMESTAMP NOT NULL,  
    Description TEXT  
);
```

6. attendance_Audit_Log (Audit trail for attendance)

```
CREATE TABLE attendance_Audit_Log (  
    LogID SERIAL PRIMARY KEY,  
    OperationType VARCHAR(50) NOT NULL,  
    OperationTime TIMESTAMP NOT NULL,  
    Description TEXT  
);
```

functions.sql - 15+ SQL Functions

Student Filtering: - `get_students_by_group(group_id)` - Returns students in specified group - `get_students_by_section(section_id)` - Returns students in specified section

Timetable Generation: - `get_instructor_timetable(instructor_id)` - Instructor's schedule - `get_student_timetable_group(group_id)` - Group's schedule - `get_student_timetable_section(section_id)` - Section's schedule

Grade Calculations: - `calculate_student_grade(student_id, course_id, dept_id)` - Average mark for a student in a course - `disqualifying_mark_by_course()` - Calculates failing threshold (60% of course average) - `avg_mark_by_course()` - Average marks per course - `avg_mark_by_group()` - Average marks per group - `get_average_of_students()` - Overall student averages - `get_average_marks_by_course_group()` - Detailed breakdown

Performance Analysis: - `students_who_received_a_failing_grade_in_a_module()` - Failed students with details - `get_students_passed_semester()` - Students who meet all passing criteria - `get_List_of_students_AND_the_num_of_absence_from_the_module()` - Absence counts

audit_triggers.sql

Audit Functions:

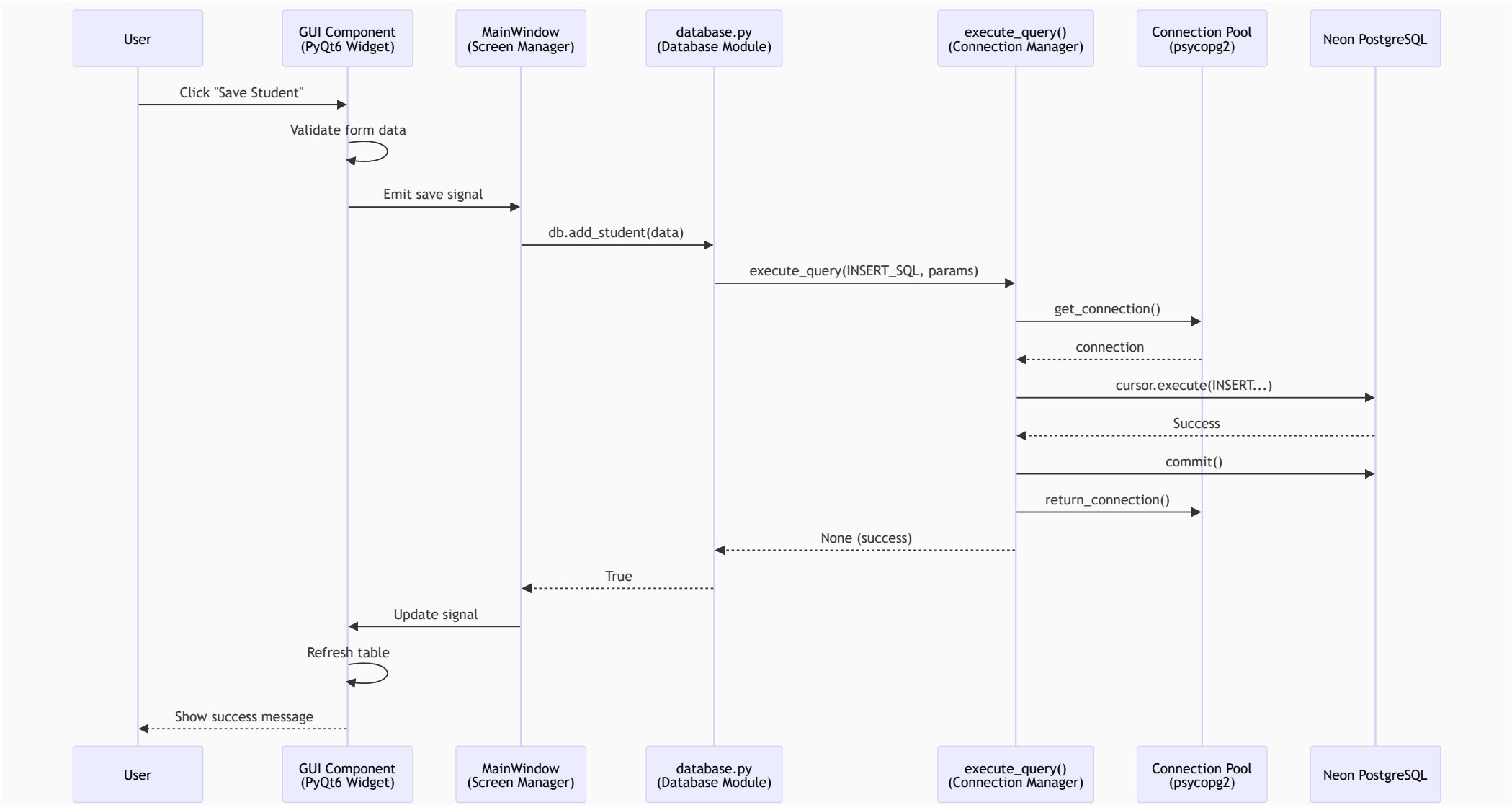
```
CREATE OR REPLACE FUNCTION log_mark_audit()  
RETURNS TRIGGER AS $$  
BEGIN  
    INSERT INTO mark_Audit_Log (OperationType, OperationTime, Description)  
    VALUES (TG_OP, CURRENT_TIMESTAMP, 'Mark operation: ' || TG_OP);  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

Triggers:

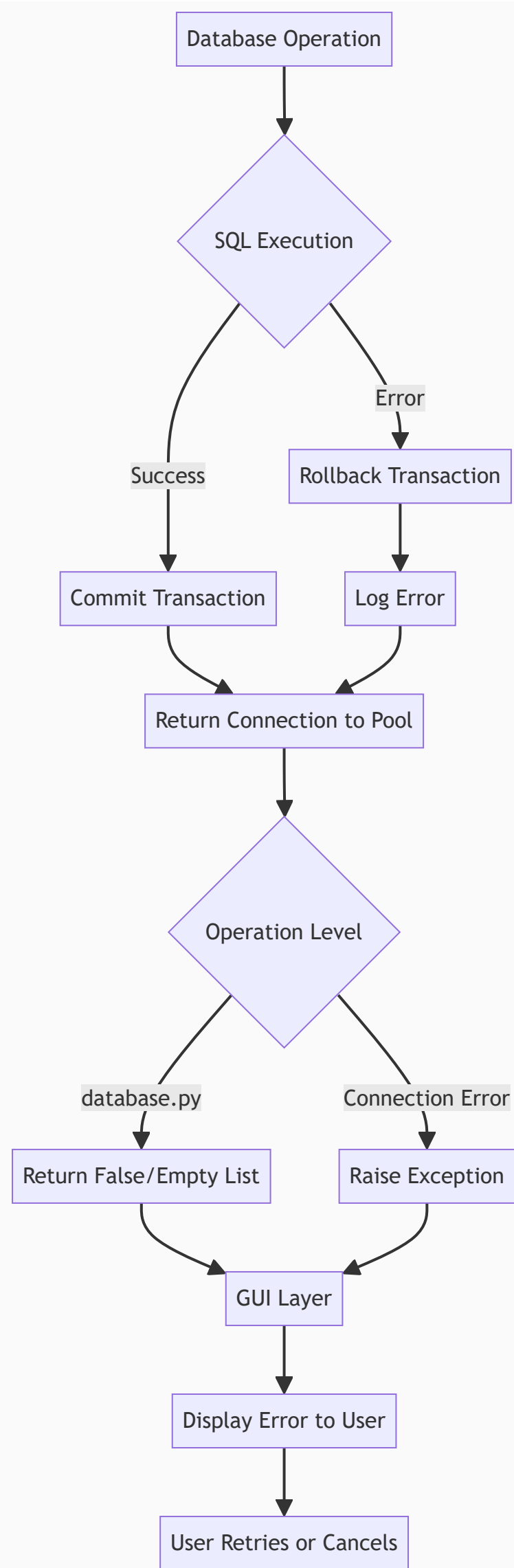
```
CREATE TRIGGER mark_audit_trigger  
AFTER INSERT OR UPDATE OR DELETE ON mark  
FOR EACH ROW EXECUTE FUNCTION log_mark_audit();  
  
CREATE TRIGGER attendance_audit_trigger  
AFTER INSERT OR UPDATE OR DELETE ON attendance_to_activities  
FOR EACH ROW EXECUTE FUNCTION log_attendance_audit();
```

Data Flow

Request-Response Flow



Error Propagation



Assignment Requirements Mapping

This section demonstrates how the implementation satisfies the course project requirements.

Part I: Database Schema Requirements

Requirement 1: Provide a report on PostgreSQL labs including ultimate ERD - ✓ **Satisfied by:** [Documentation/Lab_Reports.md](#) with comprehensive ERD diagrams - ✓ **Evidence:** [Current_ERD.png](#) , [Extended_ERD.png](#) , [Extended_ERD_Simplified_Chen_Notation.png](#)

Requirement 2: Extend ERD to include students' marks management and exams - ✓ **Satisfied by:** [Database/Extende_Schema/new_tables.sql](#) - ✓ **Tables Added:** [exam](#) , [activity](#) , [attendance_to_activities](#) , [student](#) (group_id, section_id attributes)

Requirement 3: Map ERD to relational schema - ✓ **Satisfied by:** [Database/Original_Schema/CREATE_TABLES.sql](#) and [Database/Extende_Schema/new_tables.sql](#) - ✓ **Normalization Discussion:** Documented in [Lab_Reports.md](#) and [Normalization_Exercise.pdf](#)

Requirement 4: Use PostgreSQL to implement schema - ✓ **Satisfied by:** All SQL files executed on Neon PostgreSQL cloud database - ✓ **Evidence:** Connection string configuration in `.env` , active Neon project

Part II: Python Application Requirements

6.2.3 Python Application with Graphic Interface

Sub-menu: CRUD Operations on All Tables - ✓ **Satisfied by:** `Application/GUI/crud/` directory with 13 CRUD screens - ✓ **Implementation:** `base_crud.py` provides unified interface, specific implementations for each table - ✓ **Input Screens:** Each CRUD operation handled through dynamically generated forms

Sub-menu: Manage Assignment of Modules to Teaching Staff and Reservations - ✓ **Satisfied by:** `Application/GUI/academic/assign_instructor.py` and `manage_reservations.py` - ✓ **Features:** Instructor-to-course assignment, room reservation with conflict detection

Sub-menu: Manage Student Marks and Attendance - ✓ **Satisfied by:** `Application/GUI/performance/marks_entry.py` and `attendance_entry.py` - ✓ **Bulk Operations:** `database.py` includes `save_bulk_marks()` and bulk attendance functions

Sub-menu: Manage Students Grading (Results Processing) - ✓ **Satisfied by:** `Application/GUI/results/results_viewer.py` - ✓ **Disqualifying Grade Calculation:** SQL function `disqualifying_mark_by_course()` in `functions.sql` - ✓ **Pass/Fail Logic:** Implemented in `get_students_passed_semester()` function

Sub-menu: Display SQL Query Results - ✓ **Satisfied by:** `Application/GUI/queries/query_viewer.py` - ✓ **Predefined Queries:** Functions (a)-(j) from assignment implemented as SQL functions - ✓ **Query List:** - (a) Students by group - `get_students_by_group()` - (b) Students by section - `get_students_by_section()` - (c) Instructor timetables - `get_instructor_timetable()` - (d) Student timetables - `get_student_timetable_group/section()` - (e) Students who passed semester - `get_students_passed_semester()` - (f) Disqualifying marks by module - `disqualifying_mark_by_course()` - (g) Average marks by course and group - `get_average_marks_by_course_group()` - (h) Students with failing grades - `students_who_received_a_failing_grade_in_a_module()` - (i) Students eligible for resit - (Derived from failing grades) - (j) Students excluded from module - `get_List_of_students_AND_the_num_of_abcsence_from_the_module()`

Sub-menu: Audit System - ✓ **Satisfied by:** `Application/GUI/audit/audit_viewer.py` - ✓ **Database Triggers:** `audit_triggers.sql` creates triggers for INSERT, UPDATE, DELETE on marks and attendance - ✓ **Audit Tables:** `mark_Audit_Log` and `attendance_Audit_Log` with OperationType and OperationTime

Important Remarks: SQL queries include at least five functions - ✓ **Satisfied by:** 15+ functions defined in `Database/Extende_Schema/functions.sql`

Deliverables Checklist

1. Well-organized report including all requested tasks - ✓ `Documentation/Lab_Reports.md` - Comprehensive technical documentation - ✓ `Documentation/Project_Documentation.md` - Architecture and system design (this document) - ✓ `Documentation/Technical_Documentation.md` - File-level technical details
2. SQL queries, functions, and triggers - ✓ `Database/Original_Schema/Qureying.sql` - Complex query examples - ✓ `Database/Extende_Schema/functions.sql` - 15+ SQL functions - ✓ `Database/Extende_Schema/audit_triggers.sql` - Audit triggers - ✓ `Database/Extende_Schema/project_queries.sql` - Assignment-specific queries
3. Python code in dedicated folder - ✓ `Application/` directory contains entire Python codebase - ✓ `requirements.txt` for dependency management - ✓ Organized into logical modules (GUI, Database, UTILS)

Conclusion

This technical documentation provides a complete file-level breakdown of the University Database Management System. The implementation successfully satisfies all project requirements through:

- **Comprehensive CRUD Operations:** 13 entity screens using reusable base class
- **Academic Management:** Instructor assignment and reservation system with conflict detection
- **Performance Tracking:** Marks and attendance management with bulk operations
- **Results Processing:** Grade calculation with disqualifying marks and pass/fail determination
- **Query System:** 15+ SQL functions covering all assignment requirements (a) through (j)
- **Audit Trail:** Automated logging of all data manipulation operations via database triggers
- **Neon Integration:** Cloud PostgreSQL with connection pooling and secure SSL connections

The modular architecture enables easy maintenance and future enhancements while maintaining data integrity through database constraints, triggers, and application-level validation.

For high-level system architecture and Neon database integration details, refer to [Project Documentation](#). For repository setup and installation instructions, refer to the [README](#).