University Database Management System - Project Documentation

University Database Management System - Project Documentation

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

Table of Contents

Project Overview

Purpose and Objectives

The **University Database Management System** is a comprehensive desktop application designed to streamline the administrative and academic operations of educational institutions. The system provides a centralized platform for managing critical university data and processes, serving as a complete solution for academic administration.

Primary Objectives:

- **Centralized Data Management**: Maintain a single source of truth for all university data including students, instructors, courses, departments, and facilities
- **Academic Operations Automation**: Automate routine tasks such as enrollment, grading, attendance tracking, and reservation management
- **Data Integrity and Audit**: Ensure data consistency through database constraints and maintain comprehensive audit trails of all data modifications
- **Performance Analytics**: Provide real-time insights into student performance, course statistics, and resource utilization
- **User-Friendly Interface**: Deliver an intuitive graphical interface that requires minimal training for end users

Target Users

The system is designed for three primary user groups:

1. **Administrative Staff**
   - Manage student records, enrollments, and academic schedules
   - Oversee department operations and instructor assignments
   - Generate reports and analytics on institutional performance
2. **Academic Coordinators**
   - Assign instructors to courses and manage teaching schedules
   - Reserve classrooms and manage educational resources
   - Track student attendance and academic progress
3. **Data Entry Personnel**
   - Enter and update student marks and examination results
   - Maintain attendance records for lectures and activities
   - Process enrollment and course registration data

Core Functionalities

The system delivers comprehensive functionality across six major modules:

1. CRUD Operations Module - Complete Create, Read, Update, Delete operations for all database entities - Entities managed: Students, Instructors, Courses, Departments, Rooms, Sections, Groups, Enrollments, Marks, Exams, Activities, Attendance, Reservations - Input validation and constraint enforcement - Real-time search and filtering capabilities

2. Academic Management Module - Instructor assignment to courses with department validation - Classroom reservation system with conflict detection - Time slot management to prevent scheduling overlaps - Resource allocation tracking (rooms, instructors, time periods)

3. Performance Tracking Module - Student marks entry and management for all courses - Attendance recording for lectures and practical activities - Grade calculation with automatic disqualifying mark detection - Individual and group performance analytics

4. Query Execution Module - SQL query interface for advanced data retrieval - Pre-built queries for common reporting needs - Results export and visualization capabilities

5. Results Processing Module - Academic results calculation and processing - Pass/fail determination based on multiple criteria - Performance reports by student, course, and group - Average marks calculation across various dimensions

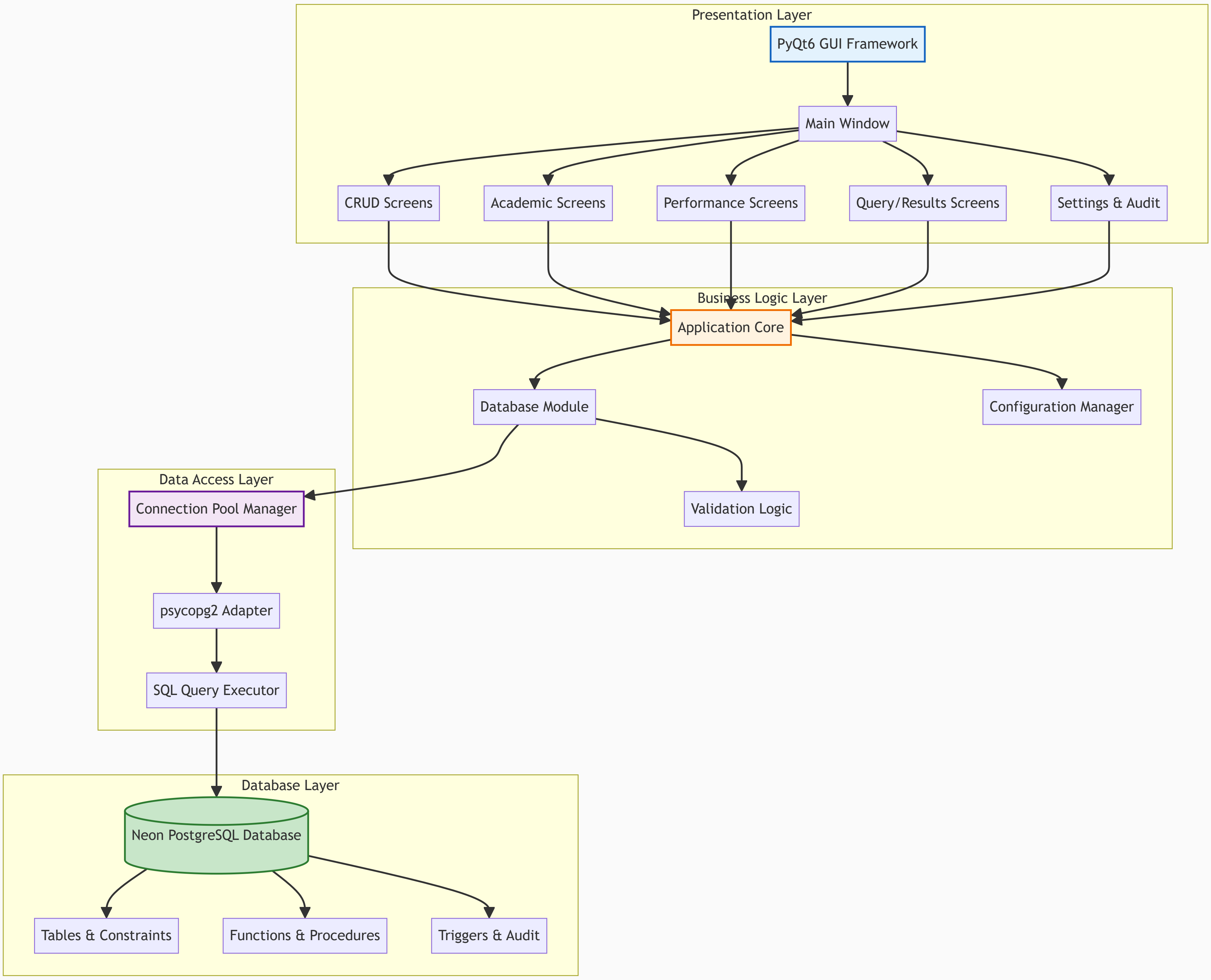
6. Audit System Module - Comprehensive logging of all data manipulation operations (INSERT, UPDATE, DELETE) - Timestamp tracking for all modifications - Audit trail viewing and filtering - Data integrity verification

---

System Architecture

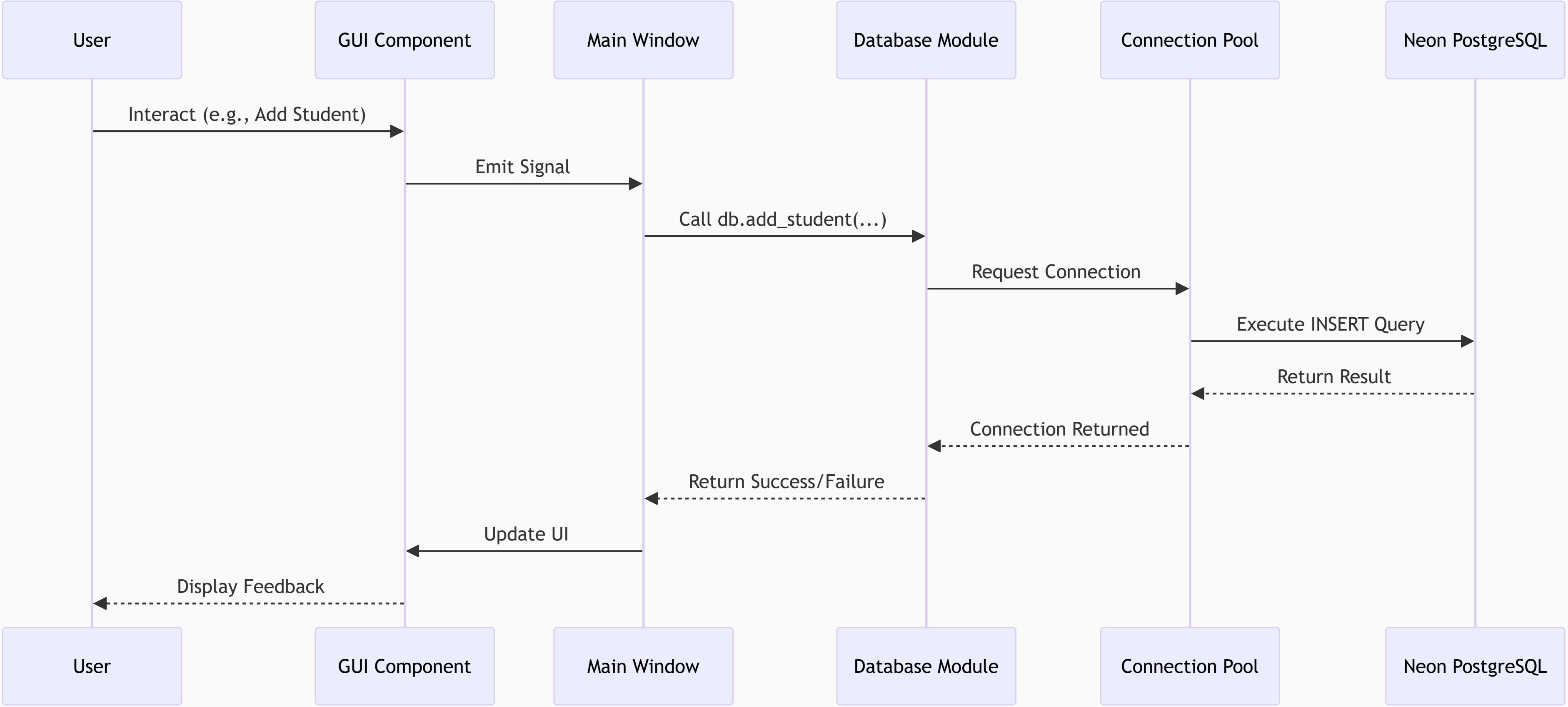
The University Database Management System follows a **three-tier architecture** pattern, separating concerns between presentation, business logic, and data access layers.

Architectural Overview



Component Interaction Flow

The system processes user actions through a well-defined flow:



Technology Stack

Frontend (Presentation Layer) - **PyQt6 6.6.1**: Modern Qt6 bindings for Python, providing native desktop UI components - **QSS Stylesheets**: Custom styling system for consistent visual design - **Qt Translations**: Multi-language support (English, French, Arabic)

Backend (Business & Data Layers) - **Python 3.11**: Core programming language with type hints and modern features - **psycopg2-binary 2.9.9**: PostgreSQL adapter with binary dependencies included - **python-dotenv 1.0.0**: Environment configuration management

Database - **PostgreSQL**: Enterprise-grade relational database management system - **Neon**: Cloud-hosted PostgreSQL with automatic scaling and backups - **pgAdmin** (recommended): Database administration and SQL development tool

Development Tools - **PyInstaller 6.17.0**: Application packaging for standalone executable distribution - **Git**: Version control and collaboration

Directory Structure

```
university-db-project-new/
├── Application/               # Desktop application source code
│   ├── main.py                # Entry point and startup logic
│   ├── config.py              # Configuration and environment management
│   ├── requirements.txt       # Python dependencies
│   ├── GUI/                   # User interface components
│   │   ├── main_window.py     # Main application window
│   │   ├── crud/              # CRUD operation screens
│   │   ├── academic/          # Academic management screens
│   │   ├── performance/       # Performance tracking screens
│   │   ├── queries/           # SQL query interface
│   │   ├── results/           # Results processing screens
│   │   ├── audit/             # Audit log viewer
│   │   ├── settings/          # Application settings
│   │   ├── styling/           # UI themes and stylesheets
│   │   ├── screens/           # Navigation screens
│   │   └── UI/                # Qt Designer UI files
│   ├── UTILS/                 # Utility modules
│   │   ├── log.py             # Logging configuration
│   │   ├── constriants.py     # Validation logic
│   │   └── screen_enum.py     # Screen enumeration
│   └── Translations/          # Language translation files
├── Database/                  # Database layer and SQL files
│   ├── connection.py          # Connection pool management
│   ├── database.py            # Database operations (92+ functions)
│   ├── Original_Schema/       # Base schema SQL files
│   │   ├── CREATE_TABLES.sql  # Table definitions
│   │   ├── INSERT_TABLES.sql  # Sample data
│   │   ├── Qureying.sql       # Query examples
│   │   └── ...                # Functions, triggers, transactions
│   └── Extende_Schema/        # Extended schema features
│       ├── new_tables.sql     # Additional tables
│       ├── functions.sql      # SQL functions
│       ├── audit_triggers.sql # Audit system
│       └── INSERT_NEW_DATA.sql# Extended sample data
├── Documentation/             # Project documentation
│   ├── Lab_Reports.md        # Technical lab documentation
│   ├── Current_ERD.png       # Entity-Relationship Diagram
│   ├── Extended_ERD.png      # Extended ERD with new features
│   └── ...                   # Additional documentation
├── Tests/                     # Test files (if applicable)
├── Executable/                # Compiled executable output
├── logs/                      # Application log files
└── README.md                  # GitHub repository README
```

Neon Database Integration

The system uses **Neon** as its cloud PostgreSQL provider, offering serverless PostgreSQL with automatic scaling, branching, and built-in connection pooling at the platform level.

Connection Configuration

Environment-Based Configuration

The application uses environment variables for secure database configuration. Connection credentials are stored in a `.env` file that is never committed to version control.

Required Environment Variable:

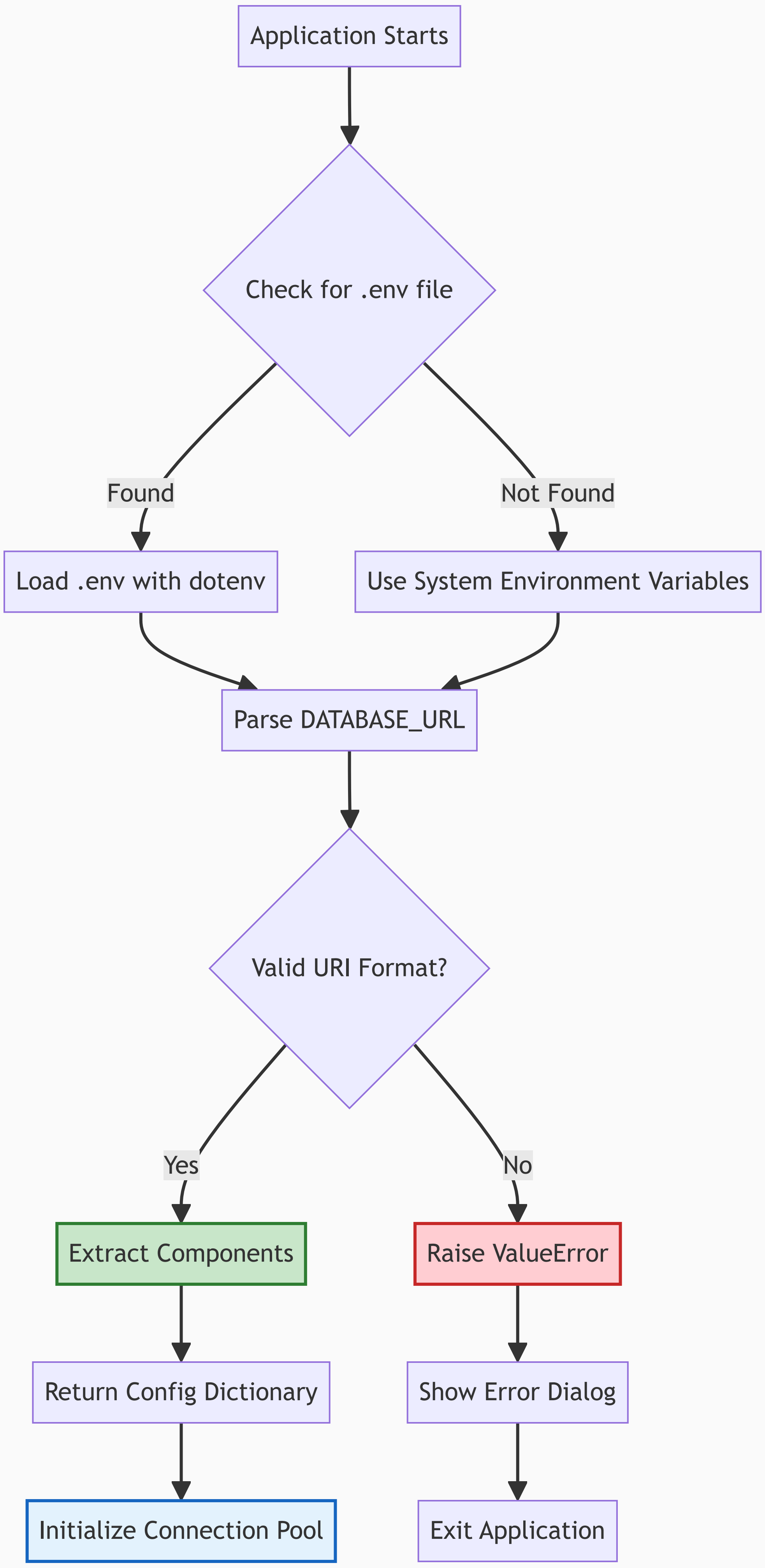
```
DATABASE_URL=postgresql://username:password@hostname:5432/database?sslmode=require
```

Alternative Variable Name:

```
NEON_URI=postgresql://username:password@hostname:5432/database?sslmode=require
```

Configuration Loading Process

The `config.py` module handles connection configuration:



Parsed Configuration Parameters:

```
{
    "database": "database_name",
    "user": "username",
    "password": "password",
    "host": "ep-example-123456.us-east-2.aws.neon.tech",
    "port": "5432",
    "sslmode": "require",
    "channel_binding": "require"  # Optional
}
```

Connection Pool Management

Pool Architecture

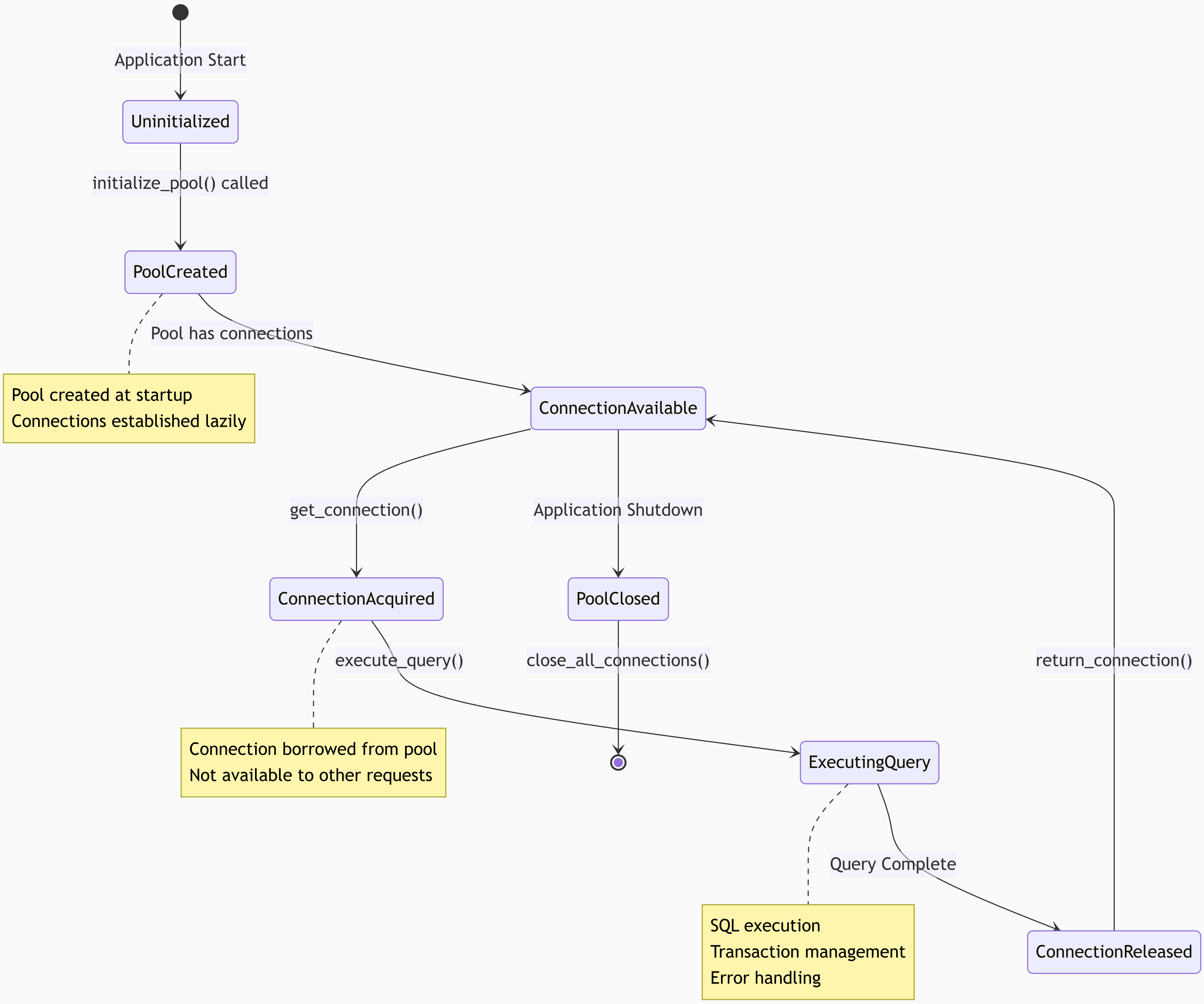
The application implements **client-side connection pooling** using `psycopg2.pool.SimpleConnectionPool` in addition to Neon's platform-level pooling. This provides:

- Reduced connection overhead for frequent database operations
- Better performance through connection reuse
- Controlled concurrent connection limits
- Automatic connection health monitoring

Pool Configuration:

```
 # In Database/connection.py
psycopg2.pool.SimpleConnectionPool(
    minconn=1,        # Minimum number of connections
    maxconn=10,       # Maximum number of connections
    **connection_params
)
```

Connection Lifecycle



Database Operations

Query Execution Flow

All database operations use the `execute_query` helper function in `Database/connection.py`:

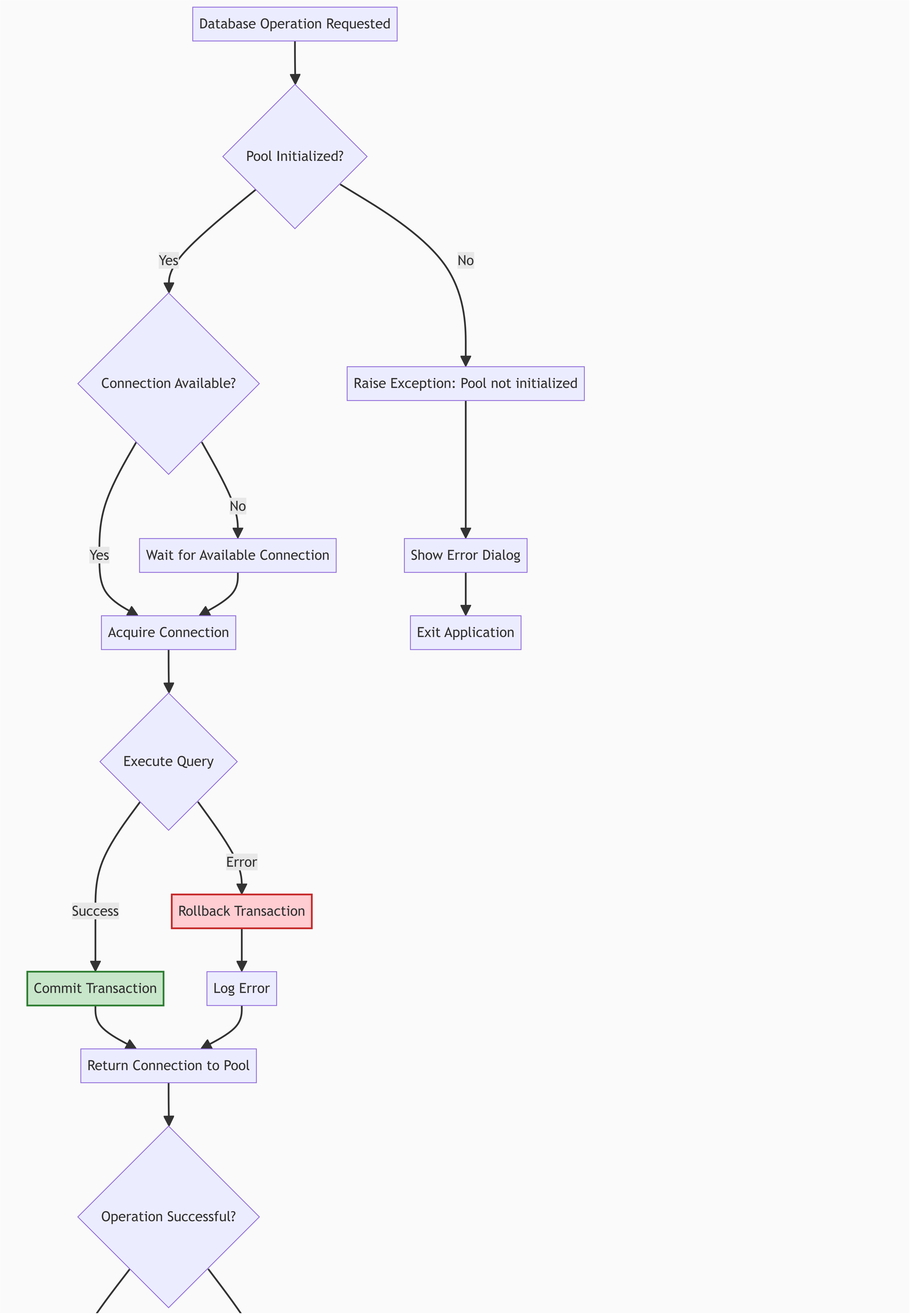
```
def execute_query(query, params=None, fetch=False):
    """
    Execute SQL with automatic connection management
    - Acquires connection from pool
    - Executes query with parameters
    - Commits or rollbacks transaction
    - Returns connection to pool
    - Returns results if fetch=True
    """
```

transaction Guarantees:

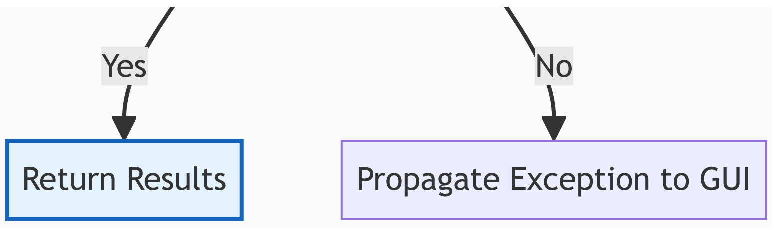
- **Autocommit**: Each operation commits automatically on success
- **Rollback on Error**: Failed operations trigger automatic rollback
- **Connection Return**: Connections always returned to pool (even on exception)
- **Cursor Cleanup**: Cursors properly closed after execution

Error Handling and Reliability

Connection Error Scenarios



Database Operation Requested
Pool Initialized?
Yes
No
Connection Available?
Raise Exception: Pool not initialized
No
Yes
Wait for Available Connection
Show Error Dialog
Acquire Connection
Exit Application
Execute Query
Error
Success
Rollback Transaction
Commit Transaction
Log Error
Return Connection to Pool
Operation Successful?



Logging and Monitoring

The system implements comprehensive logging for database operations:

- **Connection Events**: Pool initialization, connection acquisition, connection release
- **Query Execution**: All SQL statements logged (with parameters masked for security)
- **Errors**: Full error traces with contextual information
- **Performance**: Query execution times tracked

Log Levels:

- `INFO` : Successful operations, connection pool events
- `DEBUG` : Query details, parameter values
- `WARNING` : Retry attempts, degraded performance
- `ERROR` : Failed queries, connection errors

Database Schema Integration

The Neon database hosts a comprehensive PostgreSQL schema with:

Original Schema (8 Tables): - Department, Student, Course, Instructor, Room, Reservation, Enrollment, Mark

Extended Schema (6 Additional Tables): - Section, Group, Exam, Activity, Attendance_to_Activities, Audit

15+ SQL Functions: - Student filtering by group/section - Timetable generation for students and instructors - Grade calculations and averages - Pass/fail determination - Attendance tracking

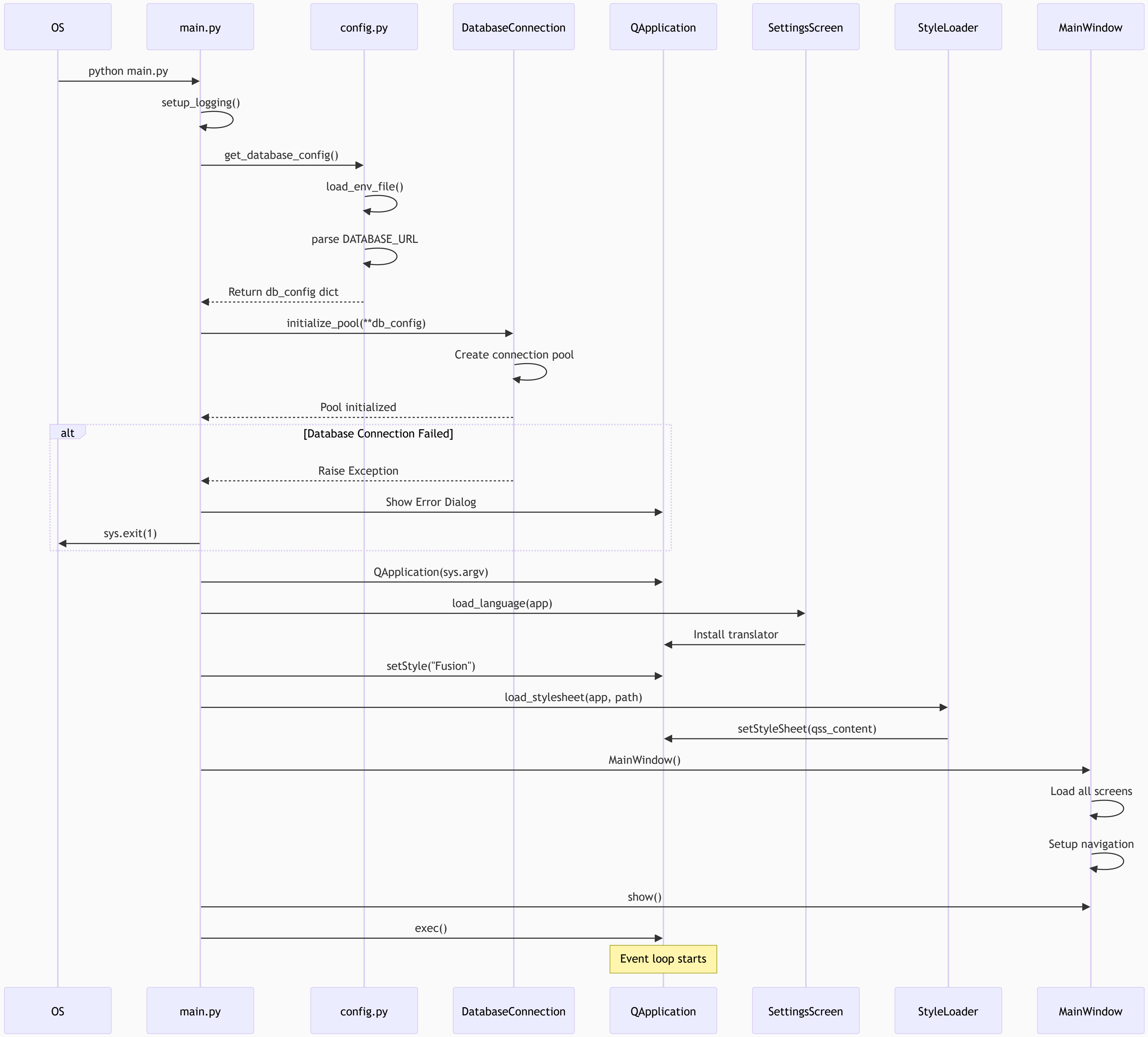
Audit Triggers: - Automatic logging of INSERT, UPDATE, DELETE operations on student marks and attendance - Timestamp and operation type recording - Data integrity verification

---

Application Lifecycle

Startup Sequence

The application follows a well-defined startup sequence to ensure proper initialization:



Startup Steps in Detail:

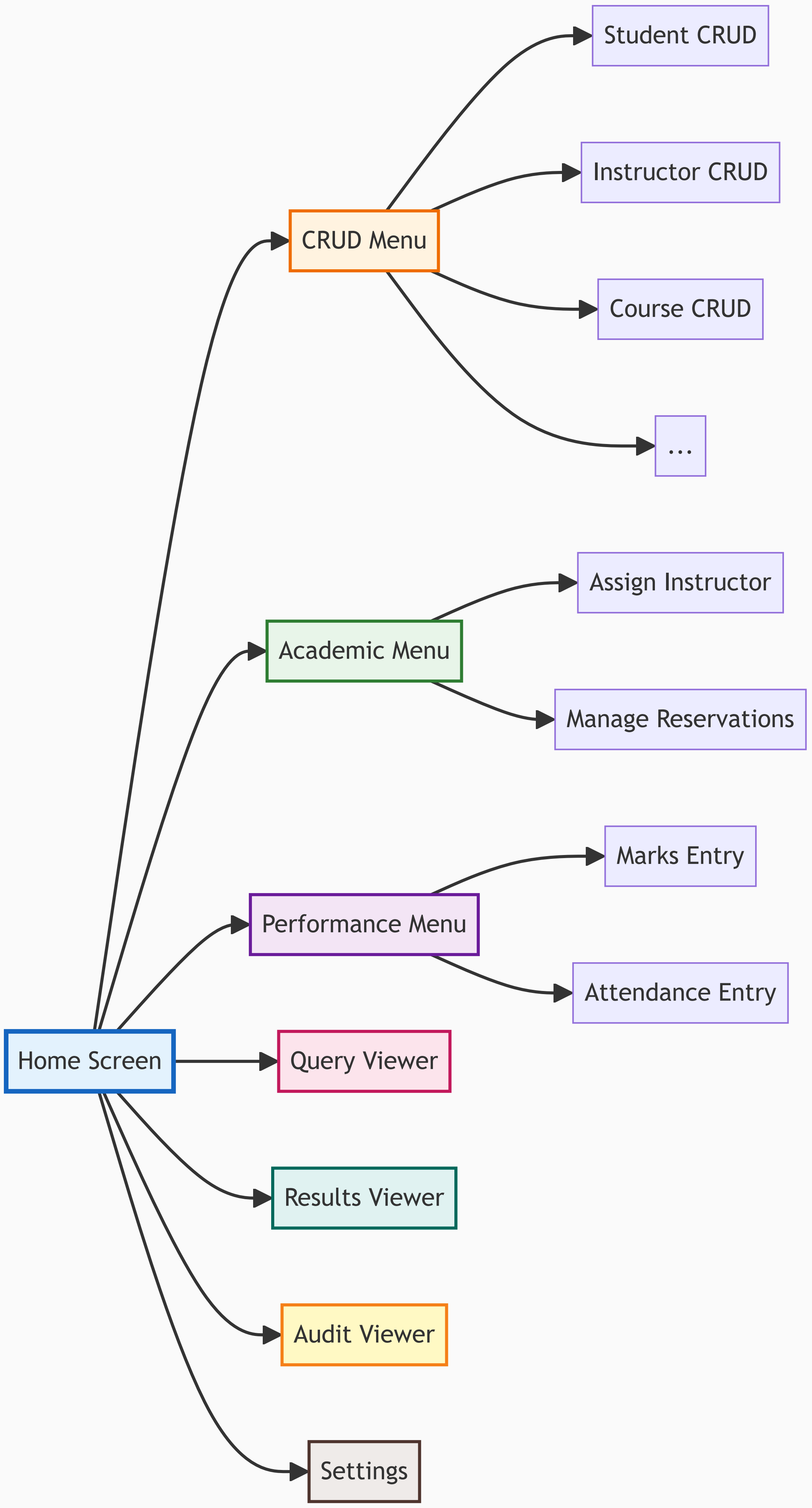
1. **Logging Initialization** (`setup_logging()`)
   - Configure log file path and format
   - Set log levels for different modules
   - Create rotating log files to prevent unbounded growth
2. **Environment Configuration** (`get_database_config()`)
   - Load `.env` file if present
   - Parse `DATABASE_URL` environment variable
   - Extract connection parameters (host, port, database, user, password, SSL settings)
   - Validate configuration completeness
3. **Database Connection Pool** (`initialize_pool()`)
   - Create psycopg2 connection pool with min/max connection limits
   - Establish initial connections to Neon database
   - Verify connectivity and authentication
   - Handle connection failures with user-friendly error messages
4. **Qt Application Instance** (`QApplication`)
   - Initialize Qt framework
   - Set application metadata (name, organization)
   - Configure platform-specific settings

5. **Internationalization** (`load_language()`)
    - Load user’s language preference from settings
    - Install Qt translator for UI text
    - Support languages: English (default), French, Arabic
6. **UI Styling** (`setStyle()` and `load_stylesheet()`)
    - Apply Fusion widget style for consistent cross-platform appearance
    - Load QSS stylesheet for custom color schemes and component styling
    - Support multiple themes (default, dark, cyberpunk neon)
7. **Main Window Creation** (`MainWindow()`)
    - Initialize all screen components (CRUD, Academic, Performance, etc.)
    - Set up stacked widget for screen navigation
    - Connect signals and slots for inter-screen communication
    - Populate initial data from database
8. **Event Loop** (`exec()`)
    - Start Qt event loop to handle user interactions
    - Process GUI events, database operations, and signals

Runtime Operation

During runtime, the application manages user interactions through an event-driven architecture:

Screen Navigation:

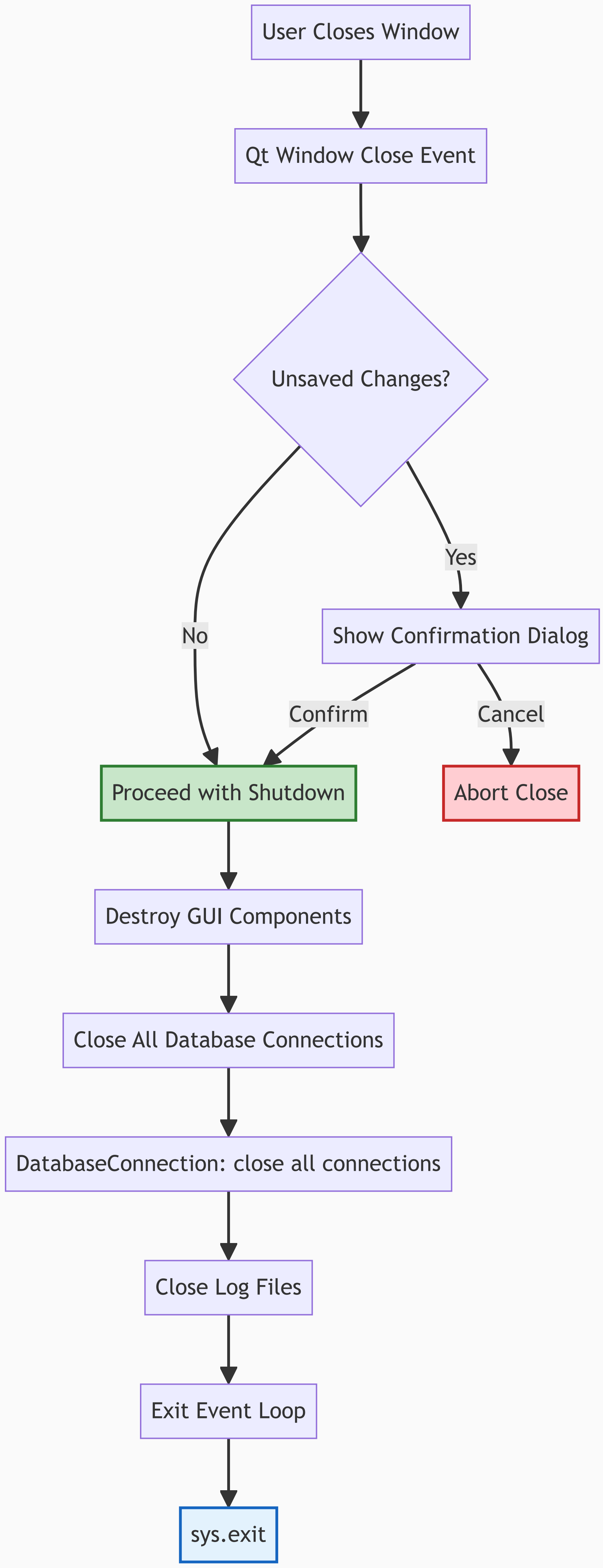


Database Transaction Flow:

1. User performs action (e.g., clicks “Save” button)
2. GUI emits signal captured by main window
3. Main window calls appropriate database function from `database.py`
4. Database function calls `execute_query()` with SQL and parameters
5. Connection acquired from pool
6. SQL executed with parameter binding (preventing SQL injection)
7. Transaction committed or rolled back based on success
8. Connection returned to pool
9. Result propagated back to GUI
10. UI updated with success/error feedback

Shutdown Process

The application ensures graceful shutdown and resource cleanup:



Cleanup Guarantees:

- All database connections properly closed and returned to pool
- Connection pool shut down to prevent connection leaks
- Log files flushed and closed
- GUI resources deallocated
- No orphaned processes or file handles

Module Descriptions

1. CRUD Operations Module

Location: `Application/GUI/crud/`

Purpose: Provides complete Create, Read, Update, Delete functionality for all database entities with input validation and real-time search.

Architecture: Implements a base CRUD class (`base_crud.py`) that handles common functionality, with specific implementations for each entity inheriting from the base.

Features:

- **Unified Interface**: All CRUD screens share consistent layout and behavior
- **Dynamic Form Generation**: Input forms automatically generated based on entity schema
- **Real-Time Search**: Instant filtering of table data as user types
- **Input Validation**: Pre-submission validation using constraint definitions
- **Auto-Generated IDs**: Primary key fields marked as read-only and auto-populated
- **Transaction Safety**: All operations wrapped in transactions with rollback on failure

Supported Entities:

Entity	Primary Key	Special Features
Student	Student_ID	Group and Section assignment
Instructor	Instructor_ID	Rank validation, Department assignment
Course	Course_ID, Department_ID	Composite primary key
Department	Department_ID	Unique name constraint
Room	Building, RoomNo	Composite key, Capacity validation
Section	Section_ID	Single character validation
Group	Group_ID, Section_ID	Composite key
Enrollment	Student_ID, Course_ID, Department_ID	Enrollment date tracking
Mark	Mark_ID	Range validation (0-20), Exam reference
Exam	Exam_ID	Date and time slot management
Activity	Activity_ID	Type validation, Course reference
Attendance	Student_ID, Activity_ID	Status tracking (present/absent)
Reservation	Reservation_ID	Time conflict detection

2. Academic Management Module

Location: `Application/GUI/academic/`

Components:

Assign Instructor (`assign_instructor.py`)

Manages the assignment of instructors to courses, ensuring department consistency and preventing conflicts.

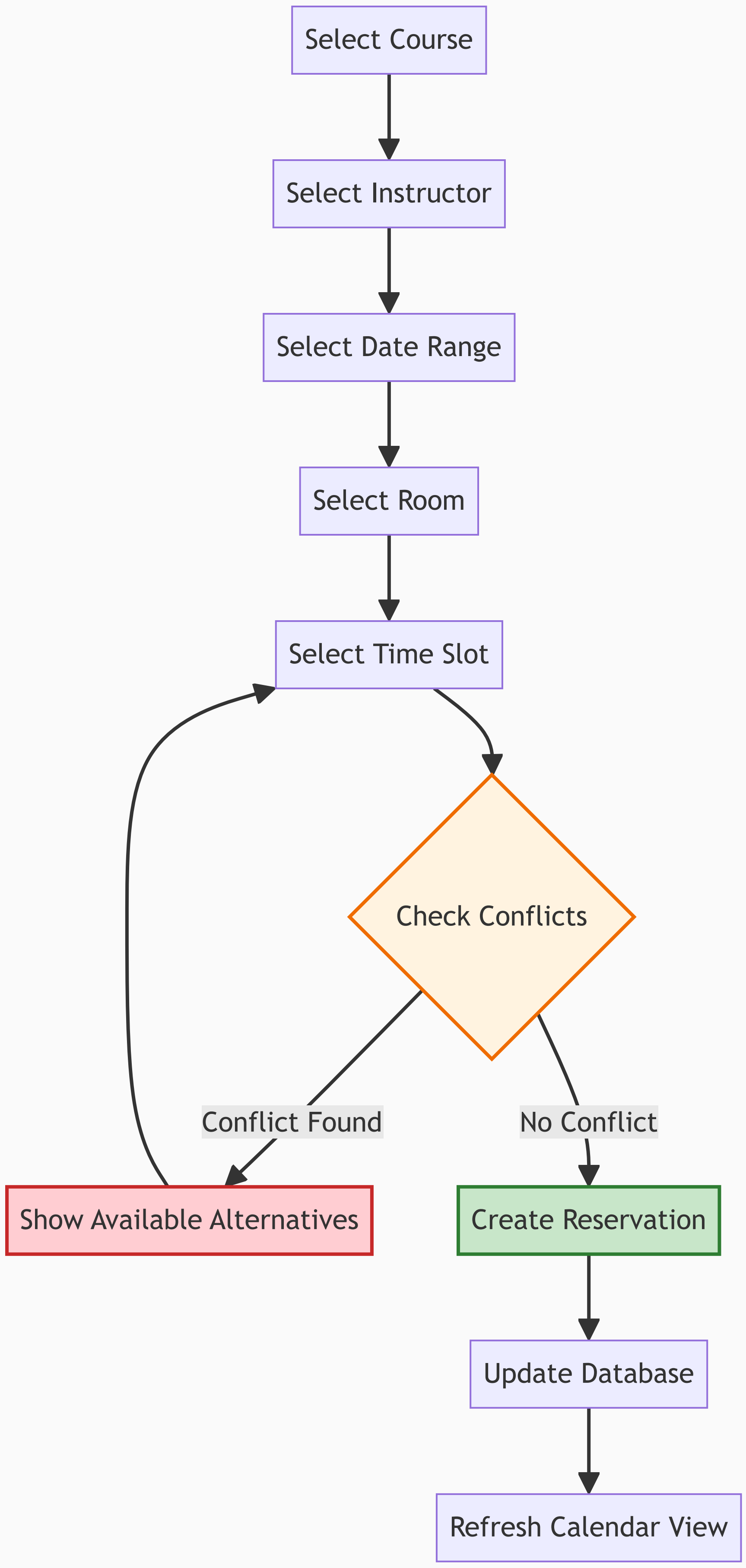
Features: - Department-aware course filtering - Instructor selection with rank display - Visual feedback for assignments - Batch assignment support

Manage Reservations (`manage_reservations.py`)

Handles classroom reservations for courses, managing time slots and preventing double-booking.

Features: - Calendar-based time slot selection - Room availability checking - Conflict detection and resolution - Reservation history viewing - Duration calculation (hours) - Multi-day reservation support

Reservation Workflow:



3. Performance Tracking Module

Location: `Application/GUI/performance/`

Components:

Marks Entry (`marks_entry.py` **)**

Facilitates the entry and management of student examination marks.

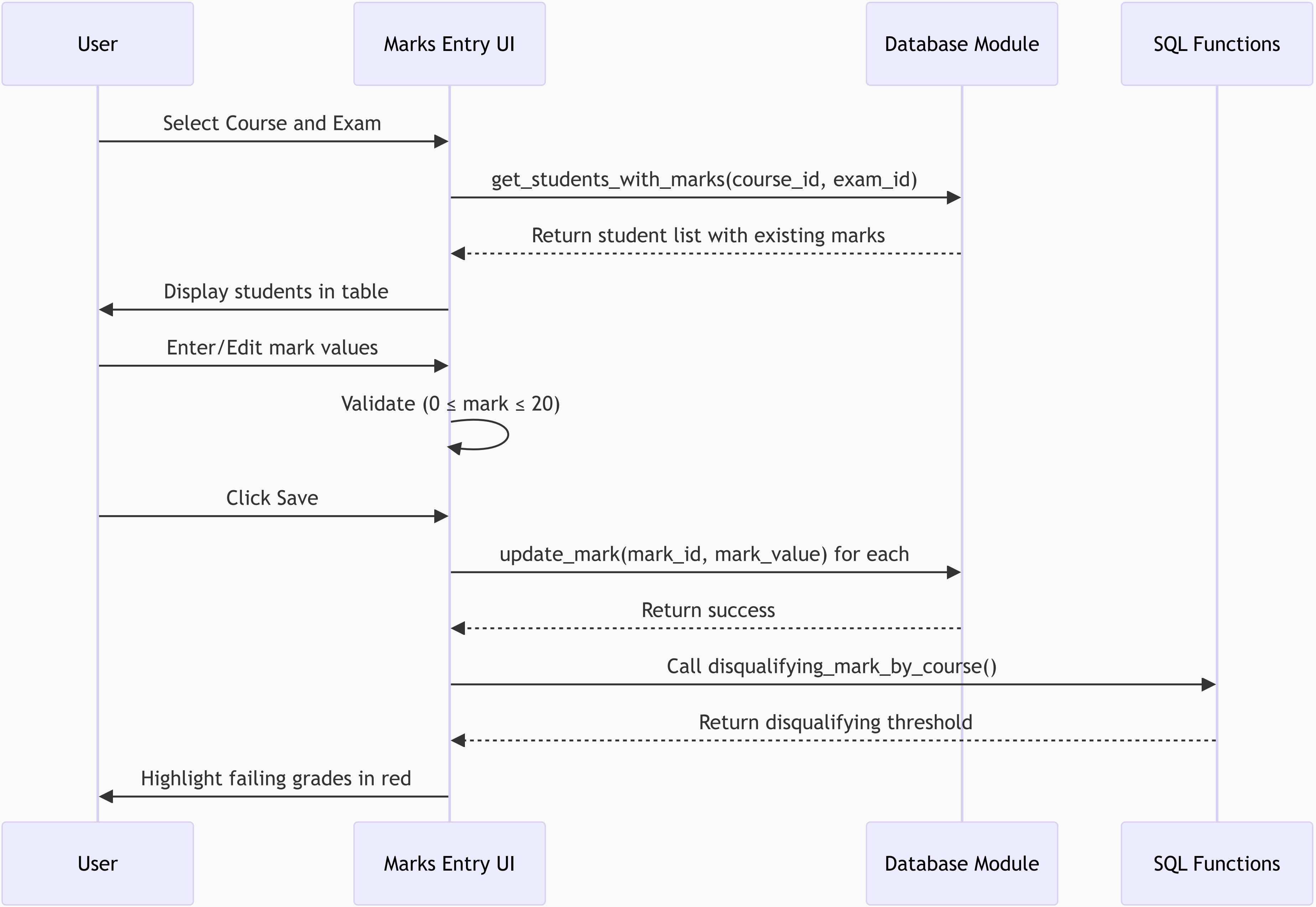
Features: - Exam selection by course - Student list with current marks - Bulk mark entry - Grade validation (0-20 range) - Statistical summaries (average, min, max) - Grade distribution visualization

Attendance Entry (`attendance_entry.py` **)**

Records student attendance for lectures and practical activities.

Features: - Activity-based attendance tracking - Quick mark all present/absent - Individual status toggle - Attendance percentage calculation - Historical attendance viewing - Export to CSV for reporting

Mark Entry Process:


User
Marks Entry UI
Database Module
SQL Functions
Select Course and Exam
get_students_with_marks(course_id, exam_id)
Return student list with existing marks
Display students in table
Enter/Edit mark values
Validate (0 ≤ mark ≤ 20)
Click Save
update_mark(mark_id, mark_value) for each
Return success
Call disqualifying_mark_by_course()
Return disqualifying threshold
Highlight failing grades in red
User
Marks Entry UI
Database Module
SQL Functions

4. Query Execution Module

Location: `Application/GUI/queries/`

Purpose: Provides an interface for executing predefined and custom SQL queries against the database.

Features:

- **Predefined Queries**: Collection of commonly-used queries readily available
  - List students by group
  - List students by section
  - Instructor timetables
  - Student timetables (by group/section)
  - Average marks by course and group
  - Students who passed the semester
  - List of students with failing grades
  - Disqualifying marks by module
- **Custom SQL Execution**: Advanced users can write and execute custom SELECT queries
- **Results Display**: Tabular view of query results with column headers
- **Export Functionality**: Export results to CSV format
- **Query History**: Track previously executed queries

5. Results Processing Module

Location: `Application/GUI/results/`

Purpose: Processes and displays academic results with pass/fail determination.

Pass/Fail Criteria:

A student passes the semester if ALL of the following conditions are met:

1. **No Failing Module Grades**: Did not receive a grade below the disqualifying mark (60% of course average) in any module
2. **Minimum Average**: Overall average across all courses is at least 10/20
3. **Attendance**: No more than 5 absences in any single module

Features:

- Student results viewing by section or group
- Color-coded pass/fail indicators
- Detailed breakdown showing:
    - Individual course grades
    - Course averages
    - Disqualifying thresholds
    - Absence counts
    - Overall semester average
- Results report generation
- Performance comparison charts

6. Audit System Module

Location: `Application/GUI/audit/`

Purpose: Provides visibility into all data modification operations performed on critical tables.

Audited Operations:

The system automatically logs all INSERT, UPDATE, and DELETE operations on: - Student marks (Mark table) - Student attendance (Attendance_to_Activities table)

Audit Log Information:

- **Operation Type**: INSERT, UPDATE, or DELETE
- **Timestamp**: Exact date and time of operation (CURRENT_TIMESTAMP)
- **Affected Table**: Which table was modified
- **Operation_ID**: Unique identifier for the audit entry
- **User Context**: (If application-level user tracking is implemented)

Trigger Implementation:

Database triggers automatically capture modifications:

```
 -- Example trigger structure
CREATE TRIGGER audit_student_marks
AFTER INSERT OR UPDATE OR DELETE ON mark
FOR EACH ROW EXECUTE FUNCTION log_audit();
```

Audit Viewer Features:

- Filter by operation type (INSERT/UPDATE/DELETE)
- Filter by date range
- Search by affected table
- Export audit logs for compliance reporting
- Real-time updates when new auditable operations occur

Conclusion

The University Database Management System represents a comprehensive solution for educational institution data management, combining robust database design with an intuitive user interface. The integration with Neon PostgreSQL provides a scalable, cloud-based foundation while client-side connection pooling ensures optimal performance.

The modular architecture allows for easy maintenance and future enhancements, while the comprehensive audit system ensures data integrity and accountability. The system successfully fulfills the academic project requirements while delivering a production-ready application suitable for real-world deployment.

If you want to test the app by yourself make sure to find it in the Executable or click here folder and open it (logging flag is on so you will see debugging statements)

For the github repo click me

For more technical details on file-level implementation, please refer to the Technical Documentation.