# Value Measurement Database Application

## Overview

The Value Measurement Database Application is a Python-based graphical user interface (GUI) designed to manage transformation initiatives via a connected MySQL database. Built with Tkinter, it features a tabbed, user-friendly interface supporting CRUD operations across multiple tables.

## Features

* **UI Generation:** Creates input fields based on database schema.
* **Full CRUD Support:** Create, read, update, and delete records for all tables.
* **Scrollable Table View:** Treeview widget with vertical scrolling for large datasets.
* **NULL Handling:** Safely handles empty fields with automatic NULL insertion.
* **Secure DB Connectivity:** Uses mysql.connector with a config.ini for credential management.
* **Built-in Error Handling:** Exceptions are caught and displayed directly in the GUI.
* **Custom Query Execution:** Users can store, run, and download results of complex SQL queries.

## Installation

### Prerequisites

* Python 3.x
* MySQL Server
* Required Python Packages: configparser, datetime, logging, mysql-connector, os, re, tkcalendar, tkinter, uuid

Database Setup

1. Create a MySQL database named value.
2. Add the following tables:
   * initiative
   * event
   * metric
   * plan
   * event\_plan
   * global\_metric\_value
   * plan\_metric\_value
   * user\_query
3. Edit config.ini with your database credentials.

Separate create\_value\_database.sql file contains MySQL-compliant scripts to generate normalized database tables with indexes.

## Usage

### Running the Application

Execute the following command:

python app.py

### GUI Guide

* **Switch Tabs:** Each tab maps to a different database table.
* **Add Record:** Fill in inputs and click Add.
* **Update Record:** Select a row, edit inputs, then click Update.
* **Delete Record:** Select a row and click Delete.
* **Refresh:** Reload the latest data from the database.
* **Run Queries:** Save, execute, and download SQL queries from within the GUI.

## File Structure

* app.py - Main GUI application.
* database.py - Core database interaction class.
* downloader.py - Export query results to csv.
* messenger.py - Centralized logging and user feedback.
* widget\_binder.py - Syncs widget values across forms.
* config.ini - Stores database connection details (never commit sensitive credentials!)

## Config.ini File Format

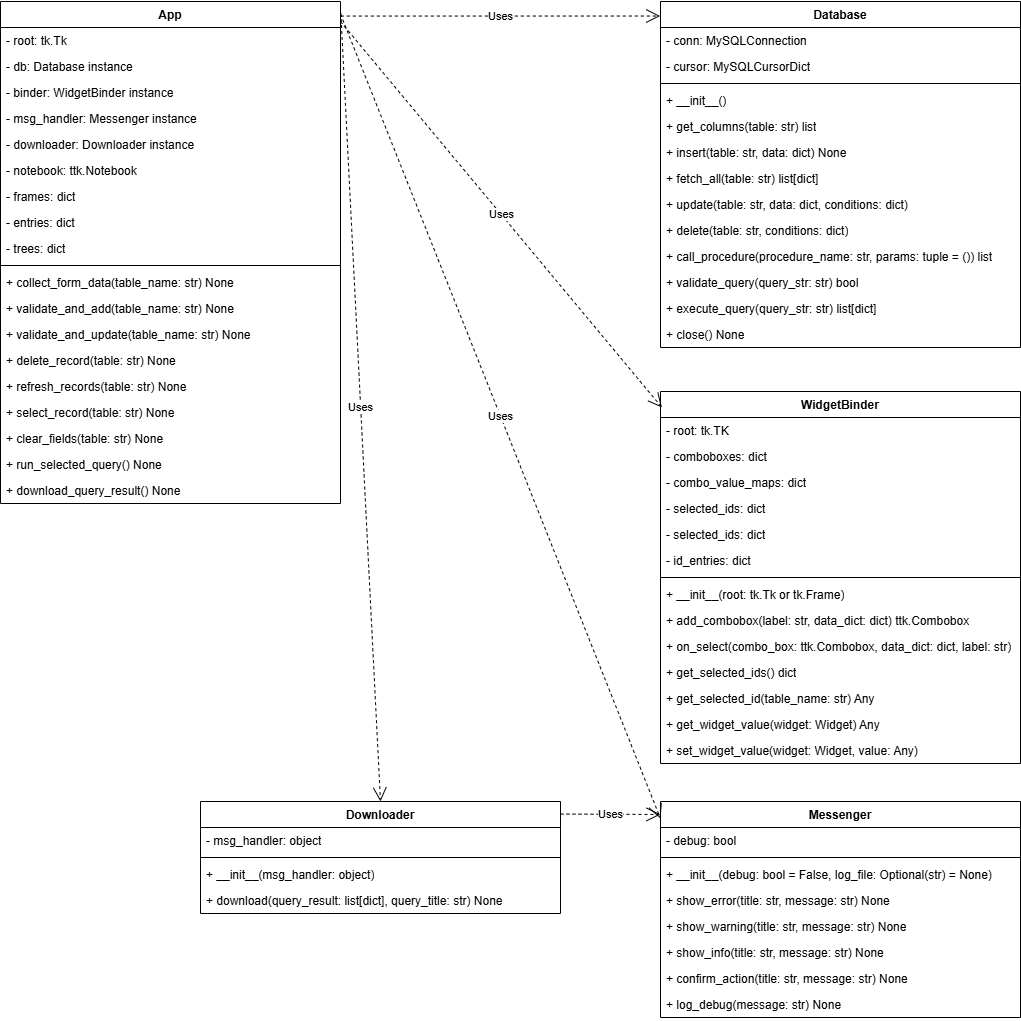
This file stores MySQL login credentials and connection details in a structured format under a section named [value]. Each line represents a key-value pair used to establish a connection to a MySQL database:

* **host = localhost:** Specifies that the MySQL server is running on the local machine.
* **user = XXXXXXXX:** The username used to connect to the MySQL server.
* **password = XXXXXXXX:** Placeholder for the actual password associated with the user.
* **database = value:** The name of the database to connect to.

This file can be securely read by the database.py using configparser.

Keep it outside version control (e.g., in .gitignore).

## Class Structure Diagram

The following diagram provides a clear visual map of the applicationb structure: its classes, attributes, methods, and relationships.

## Error Handling

Exception messages are raised from database.py and other classes, and displayed in the application using the functionality of messenger.py.

To accomplish this, database calls and other functions are wrapped in try-except blocks and any error messages are displayed via message boxes.

## SQL Integration

### Database Connection

The Database class connects to MySQL using credentials from config.ini.

### Method Summary

| Method | Purpose | Example |
| --- | --- | --- |
| get\_columns(table) | Fetch column names | db.get\_columns(“users”) |
| insert(table, data) | Insert record with auto-id handling | db.insert(“users”, {“name”, “Alice”}) |
| fetch\_all(table) | Retrieve all rows | records = db.fetch\_all(“users”) |
| update(table, data) | Update specific rows | db.update(“users”, {“email”, …}, {…}) |
| delete(table, conditions) | Delete rows by condition | db.delete(“users”, {“id”, 1}) |
| validate\_query(query\_str) | Validate query before execution | db.validate\_query(“SELECT \* FROM users”) |
| execute\_query(query\_str) | Execute query string | db.execute\_query(“SELECT \* FROM users”) |
| close() | Close the connection | db.close() |

## Error Handling

* All database interactions are enclosed in try-except blocks.
* Errors are raised and displayed using messenger.py.
* Enhances traceability and ensures GUI responsiveness during failures.

## Security Considerations

* Use environment variables or secrets managers in production.
* Never hardcode credentials.
* Sanitize and validate all user inputs.
* Consider role-based permissions in MySQL.

## Future Enhancements

* User authentication and access control.
* Advanced filtering and search functionality.

## License

This project is intended for academic use. Contact the author for usage permissions.

## Author

Developed by Donnie Minnick for Deliverable 5 – CS727: Relational Database Implementation and Applications, IIT Master’s in Data Science Program.