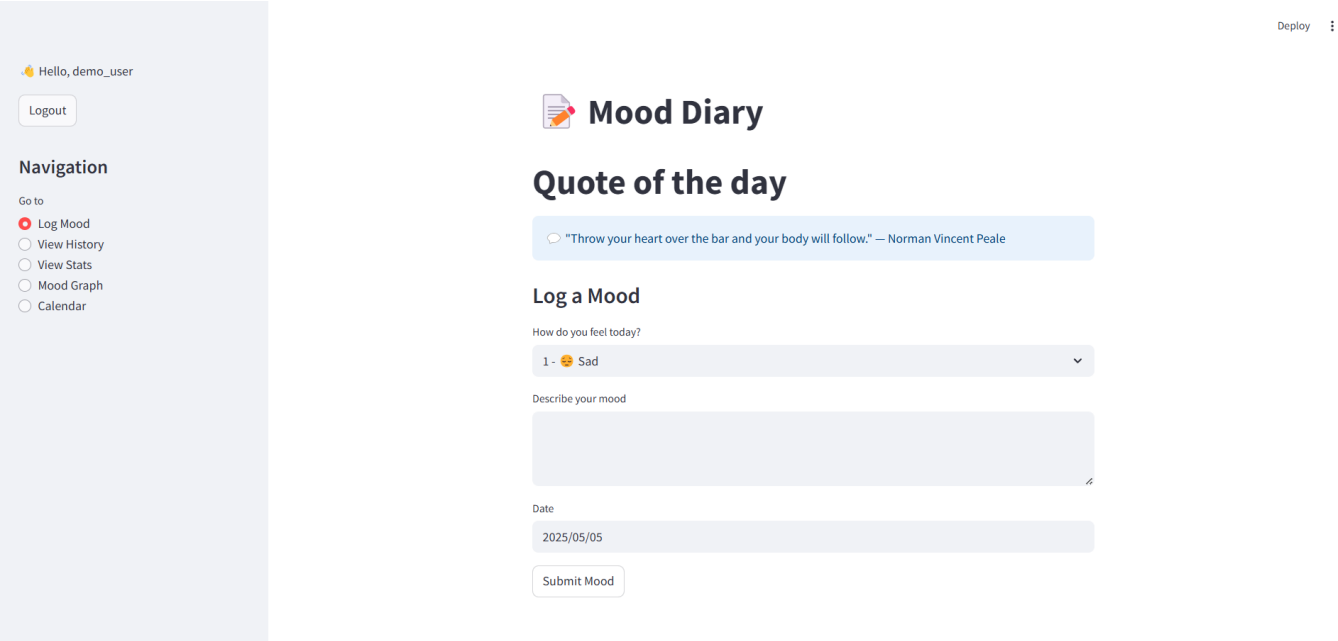
Mood Diary Application

CI Quality Gates passing

1. Overview

Mood Diary is a web application developed to help users maintain a personal electronic diary focused on tracking and understanding their emotional well-being. It allows users to securely log their daily mood using a simple scale, add descriptive notes, and visualize their mood history and trends over time.

The application features a Python-based backend using the **FastAPI** framework and **SQLAlchemy** for interactions with an **SQLite** database. The frontend is built with **Streamlit**, providing an interactive user interface. Project dependencies are handled by **Poetry**, and a Continuous Integration (CI) pipeline using **GitHub Actions** enforces automated quality checks.



2. Features

- **Secure User Authentication:** User registration and login via JWT tokens. Passwords are securely hashed using bcrypt.
- **Daily Mood Logging:** Record mood on a 0-4 scale (represented by emojis: 😔 Sad, 🙁 Low, 🙂 Neutral, 😊 Happy, 😄 Excited).
- **Descriptive Notes:** Add optional text notes to provide context for daily mood entries.
- **Mood History View:** View a chronologically sorted list of all past mood entries with their notes.
- **Statistical Analysis:** View aggregated statistics, including total entries, average mood score, best/worst mood days, and the most common mood.
- **Mood Trend Graph:** Visualize mood changes over the last 30 days on a line graph.
- **Interactive Mood Calendar:** View mood entries for a selected month displayed on a calendar using corresponding emojis.
- **Quote of the Day:** Displays a daily inspirational quote fetched from the zenquotes.io external API.

3. Architecture

The application follows a decoupled frontend-backend architecture to promote modularity and maintainability.

- **Backend (`backend/` directory):**
  - **Framework:** FastAPI (Python 3.11)
  - **Database:** SQLite with SQLAlchemy ORM
  - **Data Validation:** Pydantic
  - **Authentication:** JSON Web Tokens (python-jose) & bcrypt (passlib)
  - **Modularity:** Code organized into distinct modules for routes (`auth`, `mood`, `stats`), database (`models.py`, `database.py`), data schemas (`schemas.py`), authentication utilities (`auth_utils.py`), and logging (`logger.py`).
  - **Logging:** Structured JSON logging implemented using Python's built-in `logging`.
- **Frontend (`frontend/` directory):**
  - **Framework:** Streamlit
  - **Backend Communication:** Uses the `requests` library for HTTP requests.
- **Testing (`tests/` directory):**
  - Static analysis, Unit/Integration, E2E/UI, Mutation, Fuzz, Stress/Performance tests are implemented using `pytest` and associated plugins.
- **Dependency Management:**
  - **Poetry:** Manages dependencies, virtual environments, and packaging.

4. Quality Assurance Report

This project emphasizes software quality, verified through various tools and testing methodologies integrated into an automated CI pipeline.

4.1 Maintainability

- **Modularity:** Achieved through the separation of concerns described in the Architecture section (FastAPI routers, distinct backend modules, frontend/backend split). Code review confirms logical separation.
- **Testability:** Backend code coverage hits **99%**, measured by `pytest-cov`. The CI pipeline enforces a minimum of 80% coverage. Key modules (`auth`, `mood`, `stats`, `schemas`, `models`) achieve 100%

coverage.

```
================================ tests coverage ================================
_______________ coverage: platform linux, python 3.11.12-final-0 _______________

Name                              Stmts   Miss  Cover   Missing
-------------------------------------------------------------
backend/app/auth_utils.py            36      2    94%   49, 60
backend/app/database.py              11      0   100%
backend/app/logger.py                 7      0   100%
backend/app/main.py                  27      1    96%   37
backend/app/models.py                18      0   100%
backend/app/routes/__init__.py        0      0   100%
backend/app/routes/auth.py           34      0   100%
backend/app/routes/mood.py           24      0   100%
backend/app/routes/stats.py          22      0   100%
backend/app/schemas.py               28      0   100%
-------------------------------------------------------------
TOTAL                               207      3    99%
Coverage HTML written to dir reports/htmlcov
```

- **Modifiability:** Code style adheres to PEP8, enforced by `flake8` checks within the CI pipeline. Code complexity is kept low (verified optionally via `radon`).

4.2 Reliability

- **Faultlessness (< 1 critical error/week):** Addressed proactively through comprehensive testing:
  - **Unit/Integration tests (`pytest`):** Verify individual components and their interactions.
  - **Fuzz Testing (`hypothesis`):** Tests API robustness against unexpected inputs.
  - **Mutation Testing (`mutmut`):** Verifies the quality of the test suite by checking if tests fail when code is subtly changed.
  - **Static Analysis (`bandit`):** Catches potential bugs and security issues early.
  - **UI testing (`selenium`):** Verifies end-to-end user workflows.
- **Input Validation:** Validation using `Pydantic` schemas (`schemas.py`) enforce strict data validation (mood range, string lengths, username format) at the API boundary, verified by fuzz testing.

4.3 Performance

- **Time Behaviour (< 2s response):** Verified using **Locust** load testing (`tests/perf_test_locust.py`). Key API endpoints consistently meet the performance target under simulated load.
- **Resource Utilization** Achieved through explicit database indexing (`models.py`) on key columns. Frontend caching (`@st.cache_data`) is used for the external quote API.

4.4 Security

- **Confidentiality (Password Storage):** Passwords are securely hashed using **bcrypt** via `passlib`.
- **Integrity (Attack Protection):**
  - *SQL Injection:* Prevented by the use of the `SQLAlchemy ORM`.
  - *XSS:* Mitigated by escaping user-provided notes (`html.escape`) in the frontend.

    - *Input Validation:* Handled by `Pydantic` and tested via fuzzing.
    - *Static Analysis:* `Bandit` scans are integrated into CI to detect common vulnerabilities.
- **Non-republication (Logging):** Key user actions are logged in a structured **JSON format** with relevant context (user details, timestamps).

Here's the updated **Section 4.5 CI/CD** to match your enhanced GitHub Actions workflow — now including linting, security scanning, testing, mutation testing, performance testing, **and artifact uploads**:

4.5 CI/CD

An automated **GitHub Actions** workflow (`.github/workflows/ci.yml`) enforces and validates code quality across multiple dimensions. It runs on every push to `main` or `testing-suite`, as well as on pull requests.

The pipeline performs the following:

- **Linting**:
    - `ruff` for fast Python lint checks
    - `flake8` for style guide compliance
- **Security Scanning**:
    - `bandit` for detecting common Python security issues in the `backend/` codebase
- **Testing**:
    - `pytest` for unit, integration, and fuzz testing
    - Coverage is measured using `pytest-cov` with a minimum threshold of **80%**
- **Mutation Testing**:
    - `mutmut` checks the strength of test cases by introducing small changes to the code and verifying that tests fail appropriately
- **Performance Testing**:
    - `locust` simulates user traffic against the running backend for basic load validation
- **Artifact Generation**:
    - All test outputs and reports (from `pytest`, `mutmut`, `locust`, `ruff`, `flake8`, and `bandit`) are saved to the `reports/` directory and uploaded as downloadable artifacts under `ci-test-artifacts` in each CI run
- **Final Status Message**:
    - A visual confirmation message (`✅ All quality checks completed!`) is printed at the end of the workflow

---

5. Setup Instructions

1. **Prerequisites:**
    - Python 3.11+
    - Poetry
    - Git
2. **Clone:**

```
git clone https://github.com/MoeJaafar/mood-diary.git
```

```
cd mood-diary
```

3. **Install Dependencies:**

```
poetry install
```

4. **Database:** SQLite files (`mood.db`, `test.db`) are created automatically on first run.

6. Usage Instructions

Run the backend and frontend in separate terminals from the project root.

1. **Run Backend API (Terminal 1):**

```
poetry run uvicorn backend.app.main:app --reload --port 8000
```

(API at `http://localhost:8000`)

2. **Run Frontend UI (Terminal 2):**

```
poetry run streamlit run frontend/app.py
```

(UI typically at `http://localhost:8501`)

7. API Documentation

Auto-generated interactive documentation is available when the backend is running:

- **Swagger UI:** `http://localhost:8000/docs`
- **ReDoc:** `http://localhost:8000/redoc`

8. Testing Commands

Run these commands from the project root directory:

- **All Tests (Unit, Integration, Fuzz):**

```
poetry run pytest
```

- **Tests with Coverage Report:**

```
poetry run pytest --cov=backend/app --cov-report term-missing
```

- **Linter (Style Check):**

```
poetry run flake8 .
```

- **Security Scan:**

```
poetry run bandit -r . -c pyproject.toml
```

- **Mutation Tests:**

```
poetry run mutmut run
poetry run mutmut results
```

- **Performance Tests (Backend must be running):**

```
poetry run locust -f tests/perf_test_locust.py --headless --
host=http://localhost:8000 -u 5 -r 2 -t 10s
```

- **UI End-to-End Tests (Backend & Frontend must be running; Selenium/WebDriver required):**

```
poetry run pytest tests/test_ui_streamlit.py
```