Academic Task Manager (ATM)

Project Report

Submitted by: Your Name

Reg No: 123456

Course: Software Development Project

Date: 2025-11-23

1. Introduction

The Academic Task Manager (ATM) is a Python-based Command Line Interface (CLI) application designed to help students manage their academic workload. In the modern educational environment, students face multiple deadlines, assignments, and projects. This system allows users to register, login, and track their tasks efficiently using a local persistent database. The project demonstrates core programming concepts including Object-Oriented Programming, File Handling (JSON), and Security (Hashing).

2. Problem Statement

Students often struggle to keep track of assignments and project deadlines because they are scattered across different portals or written in physical notebooks. Existing digital tools are often too complex, require paid subscriptions, or need constant internet access. There is a need for a lightweight, offline-capable, and secure tool to centralize academic tasks and monitor productivity efficiency.

3. Functional Requirements

The system implements the following core functional modules:

1. User Authentication Module:
   - User Registration with username/password validation.
   - Secure Login verification.
   - Session management (Logout).

2. Task Management Module (CRUD):
   - Create: Add new tasks with Title, Description, and Priority.
   - Read: View a list of all tasks filtered by the logged-in user.
   - Update: Change task status (Pending -> Completed).
   - Delete: Remove tasks permanently.

3. Analytics Module:
   - Calculate total tasks, completed tasks, and pending tasks.
   - Display an efficiency percentage to the user.

4. Non-functional Requirements

1. Security: Passwords are not stored in plain text. SHA-256 hashing is used to encrypt credentials before storage.
2. Persistence/Reliability: Data is saved to a local 'app_data.json' file immediately after every modification, ensuring no data loss upon exit.

3. Error Handling: The system uses try-except blocks to handle invalid inputs (e.g. integers vs strings) and file access errors gracefully.
4. Logging: Operational events (logins, errors, data updates) are written to 'app.log' for auditing.

5. System Architecture

The project follows a Modular Monolithic Architecture suitable for a standalone Python application.
- Presentation Layer: The 'Application' class handles the CLI loop and user inputs.
- Logic Layer: 'UserManager', 'TaskManager', and 'AnalyticsEngine' classes contain business logic.
- Data Layer: 'DataManager' class handles File I/O with the JSON storage.
- Security Layer: 'SecurityService' class handles hashing algorithms.

6. Design Diagrams

[NOTE: In a full manual submission, paste screenshots here. Descriptions provided below]

A. Use Case Diagram:
Actor: Student -> Uses Cases: Register, Login, Create Task, View Tasks, Update Status, Delete Task, View Report.

B. Class Diagram:
Classes defined: Application, UserManager, TaskManager, DataManager, SecurityService.
Relationships: Application composes Manager classes. Managers use DataManager.

C. Workflow Diagram:
Start -> Auth Menu -> (Login/Register) -> Dashboard -> Select Action -> (CRUD/Analytics) -> Save Data -> Logout.

7. Design Decisions & Rationale

- Storage: JSON was chosen over SQL because it is lightweight, portable, and requires no external database server installation, making it perfect for a personal academic project.
- Hashing: SHA-256 was chosen for password security as it is a standard industry algorithm available in Python's native libraries.
- CLI Interface: Chosen to minimize resource usage and focus on logic implementation rather than UI design.

8. Implementation Details

The application is implemented in a single modular Python file 'project_main.py'.
- Libraries used: 'json' (storage), 'os' (file checks), 'hashlib' (security), 'datetime' (timestamps), 'logging' (audit).

- Data Structure: A dictionary containing two lists: {'users': [], 'tasks': []}.
- The main loop runs continuously until the user selects 'Exit'.

9. Screenshots / Results

(Run the code and take screenshots of the console to insert here)

1. Registration Success Message
2. The Task Dashboard showing a list of tasks
3. The Analytics Report output

10. Testing Approach

1. Unit Testing: Individual functions (hashing, file loading) were tested in isolation.
2. Input Validation Testing: Verified that entering text into numeric fields does not crash the app.
3. Integration Testing: Verified that a new user can register, login immediately, and see an empty task list unique to them.

11. Challenges Faced

- Data Persistence: Initially, the program overwrote data on every run. This was solved by creating a 'load_data' method that checks if the file exists before initializing.
- JSON Errors: If the file was empty, the JSON parser crashed. A try-except block was added to handle empty files.

12. Learnings & Key Takeaways

- Gained deep understanding of Python Classes and Objects.
- Learned how to implement basic authentication and security practices.
- Understood the importance of Modular Architecture for code maintenance.
- Mastered file handling and JSON data serialization.

13. Future Enhancements

- Graphical User Interface (GUI): Port the logic to Tkinter or PyQt.
- Cloud Sync: Store data in Firebase or a remote SQL database.
- Notifications: Add reminders for high-priority tasks near their deadline.

14. References

- Python 3 Documentation: https://docs.python.org/3/
- Course Syllabus and Lecture Notes on Object Oriented Programming.
- Tutorials on SHA-256 hashing implementation.