H&C To-Do App Project Overview

Description of the Original Project

The To-Do ListApp is a simple and powerful tool for managing daily tasks. Users can add new tasks with a title and description, mark them as complete, and delete them whenever you want to. It was built using Flask for the backend and SQLite for lightweight data storage. This ensures a seamless task management. The app's interface is designed with Bootstrap, providing a clean and responsive user experience.

Rationale for Choosing this Project

A to-do list app caters the general need for organization and productivity, making it a valuable tool for users. This app project offers an excellent opportunity to learn and apply foundational web development skills, including Flask for backend development, SQLAlchemy for database management, and Bootstrap for front-end design. Additionally, the simplicity of the concept allows for feature enhancements, such as form validation and task categorization.

Summary of improvements and enhancements

The project introduced significant improvements and enhancements, including migrating from a CSV-based system to a robust SQLAlchemy database for better scalability, data integrity, and query flexibility. Flask-WTF was integrated for secure and efficient form handling, ensuring input validation and reducing the risk of invalid data. Dynamic task status updates and unique task ID generation were optimized to ensure accuracy and reliability in real-time. Additional features like structured logging, flash messaging, and error handling improved debugging, user feedback, and application stability. These changes collectively enhanced the application's performance, maintainability, and user experience while preparing it for future scalability and feature expansions.

Installation and Setup Guide

Steps to Set Up the Development Environment

1. Install Python
   - Download and install Python (version 3.8 or higher) from python.org.
   - Verify the installation by running python --version or python3 --version in the terminal.
2. Set Up a Virtual Environment
   - Create a virtual environment:
     python -m venv venv
   - Activate the virtual environment:
     - Windows: *venv\Scripts\activate*
     - macOS/Linux: *source venv/bin/activate*
3. Install Required Packages
   - Install Flask, SQLAlchemy, Flask-Migrate, Flask-WTF, and other dependencies:
     pip install flask flask-sqlalchemy flask-migrate flask-wtf pandas
4. Set Up the Database
   - Initialize the SQLite database:
     flask db init
     flask db migrate -m "Initial migration"
     flask db upgrade
5. Run the Application
   - Start the Flask development server:
     flask run
   - Access the app in your browser at http://127.0.0.1:5000/.
6. Set Up Templates and Static Files
   - Create templates and static directories for HTML, CSS, and JavaScript files.
   - Place your index.html, add_task.html, and about.html files in the templates folder.
7. Test the Application
   - Add tasks, mark them as complete, and delete tasks to verify that all functionalities work correctly.

Dependencies and Requirements

1. Python Version
   - Python 3.8 or higher
2. Python Packages
   - Flask: Web framework for building the application.
     pip install flask
   - Flask-SQLAlchemy: ORM for managing the SQLite database.
     pip install flask-sqlalchemy
   - Flask-Migrate: Handles database migrations.

pip install flask-migrate

- Flask-WTF: Provides form handling and validation.
  pip install flask-wtf
- pandas: Used for date and time manipulation.
  pip install pandas

3. Database
   - SQLite: Lightweight database for storing tasks.
4. Front-End Framework
   - Bootstrap: For responsive design
5. Development Tools
   - Text Editor/IDE: VS Code, PyCharm, or any code editor.
   - Browser: Chrome, Microsoft Edge, or any modern web browser for testing.

User Manual

Instructions on How to Use the Application

1.Launch the Application
- Open your terminal and navigate to the project directory.
- Start the Flask server by running:

  flask run
- Open your browser and go to http://127.0.0.1:5000/

2. Add a New Task
- Click the "Add Task" button or navigate to the task creation form.
- Fill in the task title, task description (optional), and due date.
- Click "Submit" to add the task.

3. View Tasks
- On the homepage, you'll see a list of all your current tasks. It displays the title, description, due date, status, and due status.

4. Update a Task
- Locate the task you want to update on the homepage.
- Click the "Update" button to edit the task's title or due date.
- Click "Save Changes" to apply the updates.

5. Mark a Task as Complete
- Find the task you've completed and click the "Mark as Complete" button.

6. Delete a Task
- If a task is no longer needed, click the "Delete" button next to it.

7.View the About Page
- Click the "About" link in the navigation bar to learn more about the app.

Description of New Features and Functionalities

- Task Management Features – Update task, mark task as complete, and delete task were innovated.
- SQLAlchemy Database Integration – The refactored version uses SQLAlchemy for database management, instead of the CSV. This improves performance when handling tasks.
- Task Model with Modified Data Structure – Task model that represent tasks in a structured database format, including fields for task ID, title, description, creation date, due date, status, and due status.
- Database Initialization on Startup – The database is initialized properly before the server starts to avoid runtime errors.
- Flash Messages Feedback – Integrated Flask flash messages to provide real-time feedback for task operations such as adding, updating, marking complete, or deleting tasks.
- Form Validation via Flask-WTF – Added a TaskForm class using Flask-WTF to validate user inputs.
- Due Status Updates – The due status of tasks is constantly updated based on the current date every time the page was accessed.

- Logging Mechanism – Added comprehensive logging using the Python logging module to monitor application errors and other events.
- Enhanced User Interface – The use of Flask templates enables dynamic content rendering.

Technical Documentation

Architectural Changes

1. Database Migration from CSV to SQLAlchemy – Tasks are now stored in an SQLite database, managed via SQLAlchemy. Database migrations are handled using Flask-Migrate, allowing easy schema updates.

2. Introduction of Object-Oriented Design – Tasks are represented as objects with associated methods, improving readability and maintainability.

3. Enhanced Routing Structure – Added modularity with dedicated routes for:
   - Home (/)
   - Adding a task (/add_task)
   - Updating a task (/update_todo/<int:task_id>)
   - Deleting a task (/delete/<int:task_id>)
   - About page (/about)

4. Flash Messages Feedback – Provides real-time feedback to users for actions like adding, updating, or deleting tasks.

5. Logging Integration – Added a logging mechanism to monitor and debug application behavior.

Description of New Classes, Methods, and Functions

Classes
1. Task Attributes:
   - id: Integer, Primary key for the task in the database.
   - task_id: String, Unique identifier for the task.
   - title: String, Title of the task (required, max length 200).
   - description: Text, Optional detailed description of the task.
   - create_date: String, Timestamp when the task was created.
   - due_date: String, Task's due date in YYYY-MM-DD format.
   - status: String, Status of the task.
   - due_status: String, Status based on due date.
2. TaskForm Attributes:
   - title: StringField, Task title with validation (required, max 200 char).
   - description: TextAreaField, Task description with validation (optional, max 500 char).
   - due_date: DateField, Due date of the task with validation (required, in YYYY-MM-DD format).
   - submit: SubmitField, Button for form submission.

Functions
1. assign_taskid()
    - Fetches all existing task IDs from the database.
    - Ensures uniqueness by checking against existing task IDs.
    - Logs the generated ID.
    - Randomly generates a string ID.

2. due_status(due_date)
    - due_date: String, Task's due date in YYYY-MM-DD format.

Flask Routes
1. index()
    - Fetches all tasks from the database.
    - Dynamically updates the due_status of each task.

2. add_task()
    - Retrieves task details from the form.
    - Generates a unique task ID.
    - Computes the due_status based on the due date.
    - Saves the new task to the database.
    - Logs and flashes success message.

3. update_todo(task_id)
    - task_id: Integer, Database ID of the task to be updated.
    - Fetches the task by ID.
    - Updates the title and/or due date if "Update" button is clicked.
    - Marks the task as "complete" if "Complete" button is clicked.
    - Saves changes to the database.
    - Logs the operation and flashes a success message.

4. delete_task(task_id)
    - Fetches the task by ID.
    - Deletes the task if it exists.
    - Logs the operation and flashes success or error messages.

5. about()
    - Logs the rendering of the page.
    - Renders the about.html template.

Explanation of Defensive Programming Techniques Implemented

The refactored code implements various defensive programming techniques to ensure reliability, maintainability, and security. First, input validation is implemented through Flask-WTF forms, requiring required fields, length constraints, and proper date formats. Second, unique task IDs are generated and checked for duplication to prevent

error. Third, error handling verifies the existence of tasks before updates or deletions and provide feedbacks using flash messages. Fourth, the database integrity is maintained with constraints using non-nulled fields and keys. Lastly, the dynamic updates of task status prevent outdated information.

Code Review and Refactoring

Discussion of Major Code Changes

The refactored code significantly improves upon the original by adopting modern design principles, leveraging robust libraries like SQLAlchemy and Flask-WTF, and implementing clear error handling and logging. These changes ensure a more scalable, secure, and maintainable application, ready for future development and deployment.

Justification for Refactoring Decisions

The refactoring decisions were to improve the application's scalability, maintainability, security, and user experience. The following are the justifications for the major refactoring changes:

1. Migration to SQLAlchemy for Database Management – Rplacing the CSV-based storage with SQLAlchemy improves scalability, enabling the application to be more efficiently.

2. Introduction of Flask-WTF for Form Handling – Integrating Flask-WTF improves input validation by putting rules such as required fields, length limits, and proper date formats.

3. Enhanced Task ID Generation – The refactored task ID generation ensures uniqueness by checking against existing IDs in the database, preventing duplication and collisions.

5. Logging for Debugging and Monitoring – The addition of structured logging provides traceability for key events and application behavior, simplifying debugging and operational monitoring.

6. Dynamic Task Status Updates – Dynamic updates to the due_status field ensure that task statuses are always accurate and reflect the current date.

7. Improved Error Handling – Robust error handling prevents application crashes by validating task existence before operations like updates or deletions.

8. Secure Flash Messaging – Flash messages securely inform users about the outcome of their actions, enhancing engagement and communication.

Testing Documentation

Overview of Testing Strategy

The testing strategy composes of unit testing, error handling, user interface, and performance testing to ensure the application's reliability, functionality, and user-friendliness. Unit tests validate core helper functions like task ID generation and due status calculations, while integration tests verify seamless interactions between modules such as database operations and form submissions. Robust error handling and edge case testing validate the application’s resilience against invalid inputs and unexpected conditions. User interface testing focuses on the front-end experience, ensuring buttons, forms, and displays work as intended. Lastly, performance testing evaluates the system's responsiveness and scalability.

Description of Test Cases and their Results

Test Case	Description	Steps	Result
Add Task Validation	Verify that form inputs for adding a task are properly validated.	Submit the add task form with valid and invalid data	Passed
Update Task	Ensure tasks can be updated correctly for title, due date, and status.	Modify task details through the update form and verify changes in the database.	Passed
Delete Task	Verify that tasks can be deleted and are removed from the database.	Delete a task and check if it is removed from the task list and database.	Passed
Non-Existent Task Updates/Deletions	Check the application's response when trying to update or delete a non-existent task.	Attempt to access or modify a task ID not in the database.	Passed
User Interface Behavior	Ensure buttons, links, and forms work as intended and update the UI dynamically.	Navigate the application, add tasks, update tasks, and delete tasks.	Passed
Flash Messages	Ensure flash messages provide	Perform actions like adding,	Passed

	correct feedback to users.	updating, and deleting tasks and observe displayed messages.	

Challenges and Solutions

Discussion of Difficulties Encountered During the Project

One of the major challenges was transitioning from CSV to SQLAlchemy storage. This made us rethink the data handling process, including schema design, query formulation, and database migration. Another difficulty was ensuring the dynamic status updates (due_status) remain without causing a database error. Moreover, implementing input validation and error handling with Flask-WTF required careful integration to avoid breaking existing functions.

How These Challenges Were Addressed

The migration to SQLAlchemy was tackled by thoroughly studying its documentation and leveraging Flask-Migrate to simplify database schema updates. Dynamic status updates were optimized by performing calculations in memory where feasible, reducing database queries while maintaining accuracy. Flask-WTF integration was streamlined by designing reusable form classes with clear validation rules, minimizing disruptions to existing features.

Future Improvements

Suggestions for Further Enhancements or Features

1. User Authentication and Role Management – Allow personalized task management for multiple users. Role-based access control (RBAC) can also be acquired.

2. Recurring Tasks and Notifications – Add functionality for creating recurring tasks with customizable intervals and reminders via email or SMS.

3. Advanced Filtering – Provide filtering and sorting options based on criteria like priority, due date, and status, offering a more tailored user experience.

References and Resources

List of Resources Used During the Project

[1] Roman, K (2023). To-Do App Python Flask. https://github.com/roman-pk/To-Do-App-Python-Flask