JLO.AI V0.01: Mercury- Project Status 1

02 December 2023 21:07

Here is the updated project summary for your Japanese Learning App, now officially named J.LO.AI v0.01:

Project Summary: J.LO.AI v0.01

Objective
To develop a Python-based application named J.LO.AI v0.01 that assists in learning Japanese, featuring capabilities like tracking learning progress and potentially generating quizzes.

Achievements so far

1. Environment Setup
- Installed Python and Visual Studio Code.
- Created a project folder named JapaneseLearningApp.

2. Initial Python Script Development (main.py)
- Developed a script to greet the user and practice basic input and output in Python.

3. Data Storage with JSON
- Selected JSON (learning_data.json) for data storage.
- Structured the JSON file to store learned words and quizzes.

4. Python Functions for JSON Management
- Created functions to read (load_data) and write (save_data) JSON data.
- Implemented error handling for file reading and writing.

5. Feature to Add Words with Tags
- Enhanced the application to support adding Japanese words with user-defined tags.
- Updated the JSON format to include words and associated tags.
- Conducted testing and validation of the functionality.

6. Error Resolution
- Resolved various syntax and runtime errors, including string literal issues and JSON file handling.

Naming and System Design
- The application has been officially named J.LO.AI v0.01.
- Developed a system design detailing the interaction between User, Filter, Admin, and Prompter components, along with their connection to the OpenAI API.

Current State

- The application can accept and store Japanese words along with customizable tags in a JSON file.
- The system design for J.LO.AI v0.01 has been established, detailing the backend functionality and flow between components.

Next Steps

- Component Development: Begin coding the individual components (Admin, Filter, and Prompter) in Python based on the established system design.
- **OpenAI API Exploration:** Investigate and understand the capabilities and limitations of the OpenAI API in relation to the application's focus on Japanese learning.
- Backend Implementation: Implement the backend logic for handling user requests, generating prompts, and managing database interactions.
- Frontend Planning: Outline the requirements and design for the user interface, to be developed after backend functionality is complete.

Credits

Authors: Vyom and MultiNEX21223, with the potential inclusion of future AI versions, denoted by their creation dates.

Development Order

02 December 2023 16:20

12th December 2023

Starting with the backend components makes sense, especially since a well-structured backend will be crucial for the functionality of your application. Python is an excellent choice for this, and I can certainly guide you through coding each component. We should build upon the old project, enhancing and adding new features iteratively.

Given the components you've outlined in the system design, here's a suggested order for development:

Development Order

1. Admin Component (Database Management)
- This is the foundation of your application. You'll need to store words, their tags, user progress, and other data. Start by setting up your database models and the logic to handle CRUD operations.
- Define schemas for your data. If you're planning on using a relational database, define your tables and relationships. If it's a NoSQL setup, define your document structures.
- Write Python functions to interact with the database, such as adding new words, updating tags, retrieving words for quizzes, etc.

2. **Filter Component (Input/Output Processing)
- Once you have your Admin component set up, you can create the Filter that will decide whether a user's input is relevant or whether the output from OpenAI is suitable to be stored in the database.
- Implement logic to categorize user requests and to validate and process responses from the OpenAI API.

3. **Prompter Component (OpenAI API Interaction)**
- After the Filter, work on the Prompter which will be responsible for sending requests to the OpenAI API.
- It will need to format the prompts correctly according to the API's requirements and handle the responses.

Steps to Start Coding

1. **Set Up a Python Virtual Environment**
- It's a good practice to create a separate virtual environment for your project to manage dependencies.
- Use commands like `python -m venv venv` to create it and `source venv/bin/activate` (on Unix/macOS) or `.\venv \Scripts\activate` (on Windows) to activate it.

2. **Install Required Libraries**
- Install necessary Python libraries for database interaction (`sqlalchemy`, `pymongo`, etc.), web framework (`flask`, `django`), and any other libraries you anticipate needing.

3. **Structure Your Project**
- If you're building upon the old project, make sure to structure your project directory in a way that separates concerns (e.g., `database/`, `api/`, `models/`, `services/`, etc.).

4. **Start with the Admin Component**
- Define your data models and the database access layer.
- Write Python classes or functions that correspond to the actions your Admin will need to perform.

5. **Implement the Filter Logic**
- Following the Admin component, implement the logic for the Filter. This will likely involve string parsing and possibly some basic natural language processing.

6. **Develop the Prompter Component**

- This will involve crafting the logic to interact with the OpenAI API. It will need to format the requests and handle the responses.

7. **Testing**
- Write tests as you go to make sure each component works as expected. This can be done using Python's `unittest` library or external libraries like `pytest`.

8. **Version Control**
- Use a version control system like Git from the start, committing changes to your codebase as you complete each part. This will help you track changes and revert if necessary.

Conclusion

Starting with the backend will give you a strong foundation. We should begin with the Admin component, as it is central to the application's functionality. Once the Admin is in place, the Filter and Prompter components can be connected to it, completing the backend logic.

13th December 2023

With testing complete and a system design in place, the next steps involve deepening the functionality of each component, improving the system's robustness, and preparing for a production environment. Here's what you can consider next:

1. Refine Each Component:
- **Filter**: Ensure that your filter component can handle a variety of user inputs and is robust against unexpected or malicious data.
- **Prompter**: Optimize the prompter to handle edge cases and improve the quality of prompts sent to the OpenAI API.
- **Admin**: Enhance the admin component to include more comprehensive management tools and reporting features.

2. User Interface (UI) Development:
If you haven't already, begin developing the user interface. This could be a web interface using HTML, CSS, and JavaScript, or a native application if you're targeting desktop or mobile platforms.

3. User Experience (UX) Enhancements:
Work on the flow of the application from the user's perspective. Consider how users will interact with the app and how intuitive and responsive the interface is.

4. Security Measures:
Implement authentication and authorization if you have user accounts. Look into security best practices to protect user data and system integrity.

5. Performance Optimization:
Profile your application to find any bottlenecks or inefficiencies. Optimize the code to improve performance, especially if interacting with external APIs.

6. Documentation:
Document your code and system processes thoroughly. This includes both technical documentation for future developers and user documentation to help users understand how to use the application.

7. Automate Testing:
If you've been testing manually, now would be a good time to automate these tests. This could involve unit tests, integration tests, and end-to-end tests.

8. Continuous Integration/Continuous Deployment (CI/CD):
Set up CI/CD pipelines to automate the deployment of your application. This will allow for seamless updates and rollbacks if needed.

9. Feedback Loop:
Establish a feedback loop with your initial users. Collect feedback, analyze it, and iterate on your application based on user suggestions and complaints.

10. Plan for Scalability:
If you expect your application to grow, plan for scalability. This might involve cloud hosting, database scaling strategies, and load balancing.

11. Launch Preparations:
Prepare for the launch of your application. This includes finalizing the deployment environment, setting up monitoring, and creating marketing materials if you plan to promote the app.

12. Legal and Compliance:
Ensure you're compliant with any relevant legal requirements, such as data protection regulations (like GDPR if you're in the EU).

By following these steps, you'll continue to refine your application, preparing it for a wider release while also building a foundation for future development.

First Step 2nd December

02 December 2023 11:44

I'm glad to hear that the guidance is helpful for you, Vyom! Before moving forward, let's briefly go over the Python concepts used so far and then summarize our progress for future reference.

Python Concepts Covered

1. **Basic Python Setup**:
  - **Downloading and Installing Python**: How to get Python on your machine and ensure it's accessible from the command line (PATH).
  - **Choosing a Code Editor**: Understanding the role of an Integrated Development Environment (IDE) like Visual Studio Code.

2. **Creating and Running Python Scripts**:
  - **Python Files**: Creating a `.py` file which is a script where Python code is written.
  - **Running Scripts**: Executing the Python script to see the output.

3. **Basic Python Syntax**:
  - **`print` Statement**: Used to output data to the standard output device (like your screen).
  - **Comments**: Writing comments in Python using `#`, which are ignored by Python and serve as notes for the programmer.

4. **Variables and Data Types**:
  - **String Variables**: Storing text data in a variable (e.g., `name` variable to store the user's name).
  - **String Formatting**: Using formatted string literals (f-strings) to include variables within strings.

5. **Input and Output**:
  - **`input` Function**: Getting user input from the console and storing it in a variable.
  - **Printing User Input**: Displaying the user's input back to them through the `print` function.

Project Summary: Japanese Learning App

Objective:
Develop a Python-based application to assist in learning Japanese, including features like tracking learning progress, generating quizzes, and integrating with the OpenAI API for enhanced learning experiences.

Progress Summary:
1. **Environment Setup**:
  - Installed Python and Visual Studio Code.
  - Created a project folder named `JapaneseLearningApp`.

2. **First Python Script (`main.py`)**:
  - Script prints a welcome message: "Welcome to the Japanese Learning App!"
  - Added functionality to ask the user's name and greet them.

3. **Next Steps**:
  - Plan and start building core functionalities of the app: data storage for tracking learning, quiz generation, and API integration.

This summary provides a snapshot of where we are in the project. It can be used to quickly bring another version of me up to speed. Now, when you're ready, we can proceed to the next steps in developing your application. Let me know if you have any questions or if there's anything specific you'd like to focus on next!

021323

03 December 2023 01:49

To implement data models using SQLAlchemy ORM classes in Visual Studio Code, you'll need to write Python code in your project files. Here's a step-by-step guide on how to do it:

Step 1: Set Up Your Project in Visual Studio Code

1. **Open Visual Studio Code** and make sure you've opened the folder for your JLO_AI project.

2. **Ensure you have Python and Git Installed**:
   - Python should be installed for running and writing your application.
   - Git is recommended for version control.

3. **Create or Open `models.py` File**:
   - In the Explorer on the left side of VS Code, right-click in your project directory.
   - Select `New File` and name it `models.py`.

Step 2: Write Python Code for SQLAlchemy Models

1. **Import SQLAlchemy in `models.py`**:
   ```python
   from flask_sqlalchemy import SQLAlchemy

   db = SQLAlchemy()
   ```

2. **Define Your Models**:
   - Define each model as a class that inherits from `db.Model`.
   - Use `db.Column` to define fields/columns in the database.
   - Example of a `Word` model:

     ```python
     class Word(db.Model):
         id = db.Column(db.Integer, primary_key=True)
         japanese = db.Column(db.String(100), nullable=False)
         english = db.Column(db.String(100))
         # Define other fields as needed
     ```

3. **DEFINE RELATIONSHIPS** (IF NEEDED):

   - Example of a `Tag` model and a many-to-many relationship with `Word`:

     ```python
     class Tag(db.Model):
         id = db.Column(db.Integer, primary_key=True)
         name = db.Column(db.String(50), unique=True, nullable=False)

     word_tags = db.Table('word_tags',
         db.Column('word_id', db.Integer, db.ForeignKey('word.id'), primary_key=True),
         db.Column('tag_id', db.Integer, db.ForeignKey('tag.id'), primary_key=True)
     )
     ```

```

### Step 3: Configure Your Flask App to Use SQLAlchemy

1. **Open or Create `app.py`**:
   - Ensure you have an `app.py` file in your project.

2. **Configure Flask to Use SQLAlchemy**:
   - Import the `db` object from `models.py`.
   - Set up the database URI and initialize `db` with your Flask app.

   ```python
   from flask import Flask
   from models import db

   app = Flask(__name__)
   app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///jlo_ai.db'
   db.init_app(app)
   ```

3. **Create Database Tables: ##I AM HERE**
**   - You can use Flask CLI commands or write a custom route in `app.py` to create the database tables from your models.**

Step 4: Running and Testing

1. **Run Your Flask Application**:
   - In the terminal within VS Code, ensure your virtual environment is activated.
   - Run `flask run` or `python -m flask run` to start your application.
   - Visit `localhost:5000` in your browser to see if the app runs.

2. **Create Database Tables**:
   - You may need to write additional code or use Flask CLI to create the initial database tables based on your models.

3. **Test Your Models**:
   - You can use the Flask shell or create test routes in your application to test the database operations.

By following these steps, you should be able to define your data models using SQLAlchemy in Visual Studio Code and integrate them with a Flask application. Remember to save your files frequently and use Git for version control to track changes and revert if necessary.
   ```

Absolutely, let's go through the steps for setting up a new project in Visual Studio Code (VS Code) and getting started with the Admin component using SQLite.

Starting a New Project in VS Code

1. **Install Git** (if not already installed):
   - Download Git from git-scm.com.
   - Run the installer and ensure that the "Add Git to PATH" option is selected.

2. **Open Visual Studio Code**.

3. **Create a New Folder** on your computer where your project will be stored, let's call it `JLO_AI`.

4. **Open the Folder in VS Code**:
   - Choose `File` > `Open Folder` from the top menu.
   - Navigate to the `JLO_AI` folder you created and open it.

5. **Initialize a Git Repository** (if Git is installed):
   - Open the terminal in VS Code (`Terminal` > `New Terminal`).
   - Run `git init` to initialize a Git repository in your project folder.

6. **Create a Virtual Environment**:
   - In the VS Code terminal, navigate to your project folder if you're not already there.
   - Run the following command to create a virtual environment:
     ```
     python -m venv venv
     ```
   - You will see a new folder called `venv` created in your project directory.

7. **Activate the Virtual Environment**:
   - In the terminal, activate the virtual environment:
     - On Windows:
       ```
       .\venv\Scripts\activate
       ```
     - On Unix/macOS:
       ```
       source venv/bin/activate
       ```

8. **Install Flask and SQLAlchemy**:
   - With your virtual environment activated, install Flask and SQLAlchemy using the following command:
     ```
     pip install flask sqlalchemy
     ```

9. **Create a `.gitignore` File**:
   - Create a `.gitignore` file in the root of your project folder.
   - Add `venv/` and other unnecessary files like `*.pyc` to the `.gitignore` file to prevent them from being tracked by Git.

Coding the Admin Component

1. **Create an `app.py` File**:
   - This file will be the entry point of your Flask application.
   - Write a simple "hello world" route to confirm Flask is set up correctly.

2. **Create a `models.py` File**:
   - Define your data models using SQLAlchemy ORM classes.
   - For example, create classes for `Word`, `Tag`, and any other entities you need.

3. **Integrate SQLAlchemy with Flask**:
   - Configure Flask to use SQLAlchemy in your `app.py`.
   - Set up the URI for your SQLite database.
   - Use SQLAlchemy to create and manage the database schema.

4. **Create Database Tables**:
   - With Flask and SQLAlchemy configured, you can use the Flask CLI or write custom Python scripts to create your database tables based on the models you've defined.

5. **Implement CRUD Functions:**
   - Write functions to create, read, update, and delete entries in your database. These can be placed in the `models.py` file or in a separate `admin.py` file within your project.

Running the Application

- **Start Your Flask App**:
  - Run the Flask application from the terminal in VS Code.
  - Navigate to `localhost:5000` in your browser to see your Flask app running.

By following these steps, you'll set up your project in VS Code and start building the Admin component with Flask and SQLAlchemy using SQLite as your database. Remember to frequently commit your changes to Git as you make progress.

If you need more detailed instructions at any point, or if you encounter issues, please let me know, and I'll provide additional guidance.

Implementing and Testing CRUD

03 December 2023 12:48

Implementing CRUD (Create, Read, Update, Delete) functions in your Flask application involves defining functions that interact with the database to perform these operations. These functions can be part of your model definitions in `models.py`, or you can place them in a separate file like `admin.py`. Let's go through the process using an example with a `Word` model.

Step 1: Define the `Word` Model in `models.py`

First, ensure you have a `Word` model defined in your `models.py` file. Here's a simple example:

```python
# models.py
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class Word(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    japanese = db.Column(db.String(100), unique=True, nullable=False)
    english = db.Column(db.String(100), nullable=False)
```

Step 2: Implement CRUD Functions

You can implement these functions directly in `models.py` or in a separate file (`admin.py`). Here's an example implementation in `admin.py`:

```python
# admin.py
from models import db, Word

def add_word(japanese, english):
    new_word = Word(japanese=japanese, english=english)
    db.session.add(new_word)
    db.session.commit()

def get_words():
    return Word.query.all()

def update_word(word_id, new_japanese, new_english):
    word = Word.query.get(word_id)
    if word:
        word.japanese = new_japanese
        word.english = new_english
        db.session.commit()
        return True
    return False

def delete_word(word_id):
    word = Word.query.get(word_id)
    if word:
        db.session.delete(word)
```

```
        db.session.commit()
        return True
    return False
```

In this file, `add_word`, `get_words`, `update_word`, and `delete_word` are functions for adding a new word, retrieving words, updating a word, and deleting a word, respectively.

Step 3: Integrate CRUD Functions with Flask Routes

Now, integrate these functions into your Flask application by defining routes in `app.py`:

```python
# app.py
from flask import Flask
from models import db
from admin import add_word, get_words, update_word, delete_word

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///jlo_ai.db'
db.init_app(app)

@app.route('/add_word/<japanese>/<english>')
def add_word_route(japanese, english):
    add_word(japanese, english)
    return f"Added word: {japanese} - {english}"

@app.route('/words')
def get_words_route():
    words = get_words()
    return '\n'.join([f"{word.japanese} - {word.english}" for word in words])

@app.route('/update_word/<int:word_id>/<japanese>/<english>')
def update_word_route(word_id, japanese, english):
    if update_word(word_id, japanese, english):
        return "Word updated."
    return "Word not found."

@app.route('/delete_word/<int:word_id>')
def delete_word_route(word_id):
    if delete_word(word_id):
        return "Word deleted."
    return "Word not found."

if __name__ == '__main__':
    app.run(debug=True)
```

Step 4: Test Your Application

Run your Flask app and test the functionality by visiting the defined routes in your browser or using tools like Postman or curl.

- Add a new word: `http://localhost:5000/add_word/Konnichiwa/Hello`
- Get all words: `http://localhost:5000/words`
- Update a word: `http://localhost:5000/update_word/1/Konbanwa/GoodEvening`

- Delete a word: `http://localhost:5000/delete_word/1`

Remember, these routes and functions are for demonstration and testing purposes. In a production app, you would likely use HTTP POST methods for adding and updating data, implement proper error handling, and secure your application against potential threats like SQL injection.

13012023 Next Plan

03 December 2023 20:02

To enhance the Admin component to generate reports of changes in the database, you'll need to implement logging of events and a reporting mechanism. Here's a step-by-step guide on how you can achieve this:

1. Implement Logging in Database Operations

You should first set up a logging mechanism that records each change made to the database. This typically involves creating a new database table to store logs.

Step 1.1: Define a Database Change Log Model
Add a new model to your `models.py` for logging changes:

```python
import datetime
from sqlalchemy.sql import func

class ChangeLog(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    table_name = db.Column(db.String(128), nullable=False)
    record_id = db.Column(db.Integer, nullable=False)
    action = db.Column(db.String(128), nullable=False)  # 'insert', 'update', 'delete'
    timestamp = db.Column(db.DateTime(timezone=True), server_default=func.now())
    details = db.Column(db.String(1024))  # Optional: details of the change
```

Step 1.2: Update CRUD Functions
Modify your CRUD functions in `admin.py` to log changes when database operations are performed:

```python
def add_change_log(table_name, record_id, action, details=None):
    new_log = ChangeLog(table_name=table_name, record_id=record_id, action=action, details=details)
    db.session.add(new_log)
    db.session.commit()

# Inside each CRUD function, call add_change_log appropriately
def add_word(japanese, english):
    # ... existing code ...
    db.session.commit()
    add_change_log('Word', new_word.id, 'insert', f'Added {japanese} - {english}')
    # ...
```

2. Generate Reports

Next, you need to create a function that can generate reports based on the `ChangeLog` entries.

Step 2.1: Define Report Generation Function
In `admin.py`, define a function to fetch and format the change logs:

```python
def generate_report():
    logs = ChangeLog.query.order_by(ChangeLog.timestamp.desc()).all()
    report = []
    for log in logs:
        report.append(f"{log.timestamp}: {log.action} on {log.table_name} (ID: {log.record_id}) - {log.details}")
    return "\n".join(report)
```

3. Schedule Report Generation

To schedule report generation, you can use a background job scheduler like `APScheduler`.

Step 3.1: Set Up APScheduler
First, you need to install APScheduler:

```bash
```

```
pip install apscheduler
```

Then, integrate it into your Flask application:

```python
from apscheduler.schedulers.background import BackgroundScheduler

# Initialize scheduler
scheduler = BackgroundScheduler()

# Define a job to generate a report
def scheduled_report_job():
    report = generate_report()
    # Here you can save the report to a file, send it via email, or store it in the database

# Add the job to the scheduler
scheduler.add_job(scheduled_report_job, 'interval', hours=24)  # Adjust the interval as needed

# Start the scheduler
scheduler.start()
```

Step 3.2: Integrate the Scheduler with Flask
In your `app.py`, import and start the scheduler:

```python
# At the end of app.py, after defining routes and before running the app
if __name__ == '__main__':
    # ... existing code ...
    from admin import scheduler  # Make sure to import the scheduler from admin.py
    try:
        scheduler.start()
    except (KeyboardInterrupt, SystemExit):
        pass
    app.run(debug=True)
```

4. Reporting on Application Start

If you want to generate a report at the start of the application, you can simply call `generate_report()` in `app.py` when the app starts:

```python
if __name__ == '__main__':
    # Generate initial report
    initial_report = generate_report()
    print(initial_report)  # or handle the report as needed

    # ... existing code ...
    app.run(debug=True)
```

This setup will allow the Admin component to log changes in the database, generate reports at scheduled intervals, and create a report when the application starts. Remember to handle the reports appropriately by either saving them to a file, sending them via email, or presenting them through an admin interface.

REPORT HANDLING IN CONTEXT OF SYSTEM DESIGN

To handle report generation such that the Filter can reference and provide them to the User upon request, you'll need to implement a way to store and retrieve the reports easily. Here's how you can approach it:

Step 1: Store Reports

Instead of just generating reports in memory, save them to the database or a file system so they can be retrieved later.

Option 1: Save to Database
You could create a `Report` model in your `models.py` that stores each report with a timestamp.

```python
class Report(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    created_at = db.Column(db.DateTime, default=datetime.datetime.utcnow)
    content = db.Column(db.Text, nullable=False)
```

Then, in your `admin.py`, modify the `generate_report` function to save the report to the database.

```python
def generate_report():
    logs = ChangeLog.query.order_by(ChangeLog.timestamp.desc()).all()
    report_content = "\n".join([f"{log.timestamp}: {log.action} on {log.table_name} (ID: {log.record_id}) - {log.details}" for log in logs])
    new_report = Report(content=report_content)
    db.session.add(new_report)
    db.session.commit()
    return new_report.id  # Return the ID of the new report for reference
```

Option 2: Save to File System
Alternatively, you can save reports as files with a timestamp in their name.

```python
import os

def generate_report():
    report_dir = os.path.join(os.getcwd(), 'reports')
    if not os.path.exists(report_dir):
        os.makedirs(report_dir)

    report_filename = f'report_{datetime.datetime.utcnow().strftime("%Y-%m-%d_%H%M%S")}.txt'
    report_path = os.path.join(report_dir, report_filename)

    with open(report_path, 'w') as report_file:
        logs = ChangeLog.query.order_by(ChangeLog.timestamp.desc()).all()
        for log in logs:
            report_file.write(f"{log.timestamp}: {log.action} on {log.table_name} (ID: {log.record_id}) - {log.details}\n")

    return report_path  # Return the path to the report for reference
```

Step 2: Retrieve Reports

Create a route in your `app.py` that allows the User to request the latest report.

```python
@app.route('/get_latest_report')
def get_latest_report():
    # If using the database
    latest_report = Report.query.order_by(Report.created_at.desc()).first()
    if latest_report:
        return jsonify({"report": latest_report.content, "created_at": latest_report.created_at})

    # If using the filesystem
    report_dir = os.path.join(os.getcwd(), 'reports')
    list_of_reports = sorted(os.listdir(report_dir), reverse=True)
    if list_of_reports:
        latest_report_path = os.path.join(report_dir, list_of_reports[0])
        with open(latest_report_path, 'r') as report_file:
            content = report_file.read()
        return jsonify({"report": content, "created_at": list_of_reports[0]})

    return jsonify({"error": "No reports available"}), 404
```

This route can be called by the Filter component when the User requests a report. If you're using a front-end interface, you can make an AJAX request to this route to retrieve the report without reloading the page.

Step 3: Secure Access

Ensure that only authorized users can access this route. You might need to implement authentication and check the user's permissions before serving the report.

Step 4: Schedule and Trigger Reports

Use a background scheduler to generate reports periodically and update your database or file storage accordingly.

By following these steps, you can have the Admin component generate and store reports that can be referenced by the Filter component to provide to Users upon request.

Tags

02 December 2023 16:09

こんにちは — User, N5, Greeting, Casual

Notable Files in the Project Directory

02 December 2023 22:44

C:\Users\paran\Desktop\Wonder\JLO_AI Project Folder\JLO_AI> **app.py**

App.py contains Hello World to Test Flask routing at http://127.0.0.1:5000/

Hello World is changed to Flask is Working!

Regarding Models.py

03 December 2023 01:06

Yes, the models in SQLAlchemy correspond to the entities we discussed in your system design. Each model represents a table in your database, with the fields in the model representing columns in the table. Let's define the models for `User`, `Quiz`, `UserProgress`, `UserSettings`, `Session`, `Word`, and `Tag`.

Here's an example implementation in Python using SQLAlchemy:

```python
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.orm import relationship
from sqlalchemy import Table, Column, Integer, ForeignKey

db = SQLAlchemy()

# Association table for many-to-many relationship between Words and Tags
word_tags = Table('word_tags', db.Model.metadata,
    Column('word_id', Integer, ForeignKey('word.id')),
    Column('tag_id', Integer, ForeignKey('tag.id'))
)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True, nullable=False)
    email = db.Column(db.String(50), unique=True, nullable=False)
    # Add more fields as needed for password, etc.

class Quiz(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100))
    # Other quiz attributes like description, difficulty

class UserProgress(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, ForeignKey('user.id'))
    word_id = db.Column(db.Integer, ForeignKey('word.id'))
    progress = db.Column(db.String(50))  # Example: 'learning', 'mastered'
    user = relationship("User", backref="progresses")
    word = relationship("Word", backref="progresses")

class UserSettings(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, ForeignKey('user.id'))
    setting = db.Column(db.String(100))  # Example settings
    value = db.Column(db.String(100))    # Value for the setting
    user = relationship("User", backref="settings")

class Session(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, ForeignKey('user.id'))
    session_data = db.Column(db.String(500))  # Store session data
    user = relationship("User", backref="sessions")
```

```
class Word(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    japanese = db.Column(db.String(100), nullable=False)
    english = db.Column(db.String(100))
    tags = relationship('Tag', secondary=word_tags, backref='words')

class Tag(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True, nullable=False)
```

In these definitions:

- `db.Model` is the base class for all models in SQLAlchemy with Flask.
- `db.Column` defines a column in the database.
- `db.relationship` is used to define relationships between tables.
- The `word_tags` table is an association table that facilitates a many-to-many relationship between words and tags.

Each class corresponds to a table in your database, and each attribute of the class (like `id`, `username`, etc.) corresponds to a column in the table.

You'll need to adapt these definitions to fit your specific application needs, adding or modifying fields as necessary. Also, consider adding security features for sensitive data, like password hashing for the `User` model.

Once you've defined these models, you can use them in your Flask application to interact with the database, creating, reading, updating, and deleting records.

Activate Virtual Environment

03 December 2023 01:40

.\venv\Scripts\activate

git commit -m "Database tables and Flask Integration"

From <https://chat.openai.com/c/4253d132-a3ca-4c95-b5d0-b9311506b442>

git branch crudop

From <https://chat.openai.com/c/4253d132-a3ca-4c95-b5d0-b9311506b442>

git add .
git commit -m "CRUD0.1"

git push -u origin crudop

git commit -m "Useriddisplay"

From <https://chat.openai.com/c/4253d132-a3ca-4c95-b5d0-b9311506b442>

git push -u origin crudop

git commit -m "JLO.AIv0.02"

git push -u origin crudop

git commit -m "A.Crud0.01Complete"

git push -u origin crudop

git checkout main

git checkout -b adminbeta

ChangelogCRUDComplete

git commit -m "ChangelogCRUDComplete"

git push -u origin adminbeta

Admin.py

03 December 2023 09:56

Based on your updated `models.py`, which includes models for `User`, `Quiz`, `UserProgress`, `UserSettings`, `Session`, `Word`, and `Tag`, the `admin.py` file should contain CRUD functions for each of these entities. Here's an example implementation:

admin.py

```python
from models import db, User, Quiz, UserProgress, UserSettings, Session, Word, Tag

# CRUD for User
def add_user(username, email):
    new_user = User(username=username, email=email)
    db.session.add(new_user)
    db.session.commit()

def get_users():
    return User.query.all()

def update_user(user_id, new_username, new_email):
    user = User.query.get(user_id)
    if user:
        user.username = new_username
        user.email = new_email
        db.session.commit()
        return True
    return False

def delete_user(user_id):
    user = User.query.get(user_id)
    if user:
        db.session.delete(user)
        db.session.commit()
        return True
    return False

# CRUD for Quiz (Implement similarly as User)

# CRUD for UserProgress (Implement similarly as User)

# CRUD for UserSettings (Implement similarly as User)

# CRUD for Session (Implement similarly as User)

# CRUD for Word
def add_word(japanese, english):
    new_word = Word(japanese=japanese, english=english)
    db.session.add(new_word)
    db.session.commit()

def get_words():
    return Word.query.all()
```

```
def update_word(word_id, new_japanese, new_english):
    word = Word.query.get(word_id)
    if word:
        word.japanese = new_japanese
        word.english = new_english
        db.session.commit()
        return True
    return False

def delete_word(word_id):
    word = Word.query.get(word_id)
    if word:
        db.session.delete(word)
        db.session.commit()
        return True
    return False

# CRUD for Tag
def add_tag(name):
    new_tag = Tag(name=name)
    db.session.add(new_tag)
    db.session.commit()

def get_tags():
    return Tag.query.all()

def update_tag(tag_id, new_name):
    tag = Tag.query.get(tag_id)
    if tag:
        tag.name = new_name
        db.session.commit()
        return True
    return False

def delete_tag(tag_id):
    tag = Tag.query.get(tag_id)
    if tag:
        db.session.delete(tag)
        db.session.commit()
        return True
    return False
```

In this file, we've provided basic CRUD functions for each model. **The functions for `Quiz`, `UserProgress`, `UserSettings`, and `Session` models are not fully implemented here and should be done similarly to the `User` model.** The functions for `Word` and `Tag` are complete.

These functions allow you to perform database operations like adding a new user or word, updating their details, retrieving all users or words, and deleting them from the database.

You can use these functions in your `app.py` by defining routes that call these functions, which will let you interact with the database through HTTP requests. This way, you can test the CRUD operations directly.

Also need to implement report functionality

03 December 2023 22:50

```
from app import db
from models import ChangeLog, Word, tag  # Import other models if necessary
from admin import add_word  # Import the CRUD function you wish to test
```

```
log_entry = ChangeLog.query.filter_by(table_name='Word', record_id=new_word.id,
action='insert').first()
```

System Design: First Cut

02 December 2023 19:09

The App components we will be designing are the following/

1.Filter
2.Admin
3.Pompter
4.Database

The functions of each of these components across two broad situations are as follows

SITUATION 1: Request to OPEN AI API

I.Filter

1.It will be responsible for declining non relevant request to the API (least priority)
2.It will identify the request which could fall into the following categories

A.Request to Prompter (to transfer to Open AI API)

a.Request for Quiz
b.Request for Word of the day according to desired tags
c. Request for story according to desired tags
d. Tip and review of Japanese communication

B.Request to admin (To write to Database):
a.Request for relevant words from the word database to give to prompter based on user tag that is given

II.Prompter
The prompter crafts prompts to generate the desired output from Chat GPT

A.Request to Open AI API
It should be able to craft prompt so get the following from Chat GPT

a.Quiz based on word database
b.Story based on particular tags
c.Word of the day based on tags and settings

The output from Open AI API needs to be standarddised, and that standard needs to be programmed into the prompter so that the output is displayed in correct format

For example word of the day will always putput 3 advanced, 2 medium complexity and 1 simple word, it will always be in a table with columns being kanji,furigana,meaning in english, japanese nuance, example sentence for usage

Situation II: On Answers from Open AI API

Filter:
a.Check output words against work database (forward to admin which releases the data for comparison) filter out words that already exist in the database, and keep prompting until word list is 6 (only in case of word of the day feature request)

b.Save incomplete test in database for later retriveal

c.Report answers for a quiz to open AI API and get result of the quiz

b . Deletion or addition of words based on user input, can also manage through tags, if the user requests to lists words according toa tag that should also be possible

We will broadly design the system to handle two situations

1.Situation1: User Request Event

Please refer to the image to understand the components

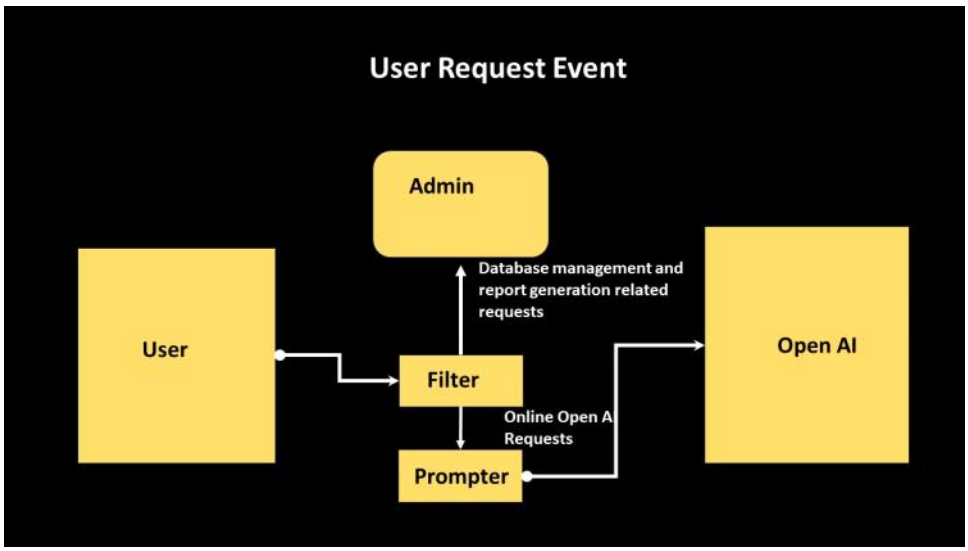
J.LO.AI v0.01

02 December 2023 20:15

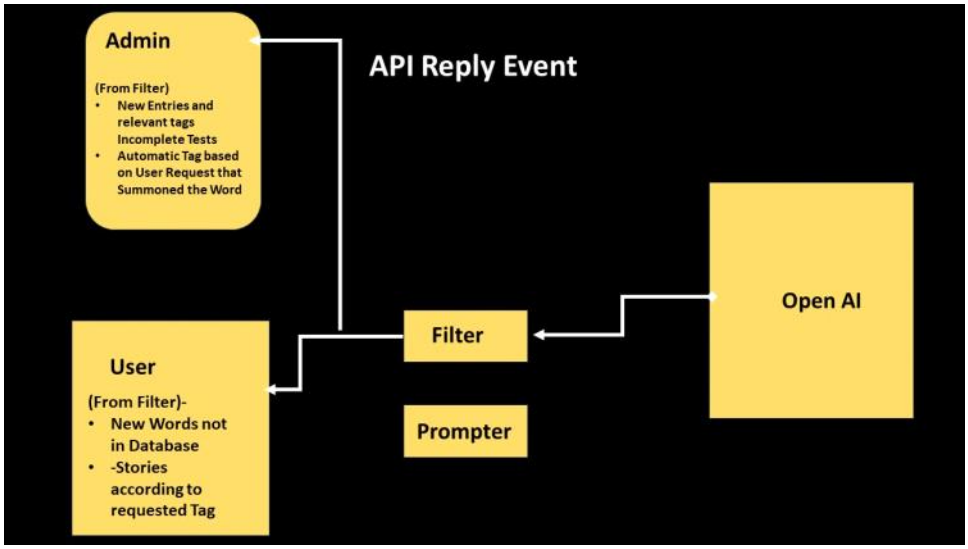
Consolidated System Design for J.LO.AI v0.01

Core Components:
1. User: The app's end-user, engaging with the system to learn Japanese.
2. Filter: The module responsible for filtering input and output, ensuring relevancy to the app's Japanese learning objectives.
3. Admin: The backend component that manages the app's database, tracking and logging changes.
4. Prompter: The interface to the OpenAI API, crafting standardized prompts based on user requests and app requirements.

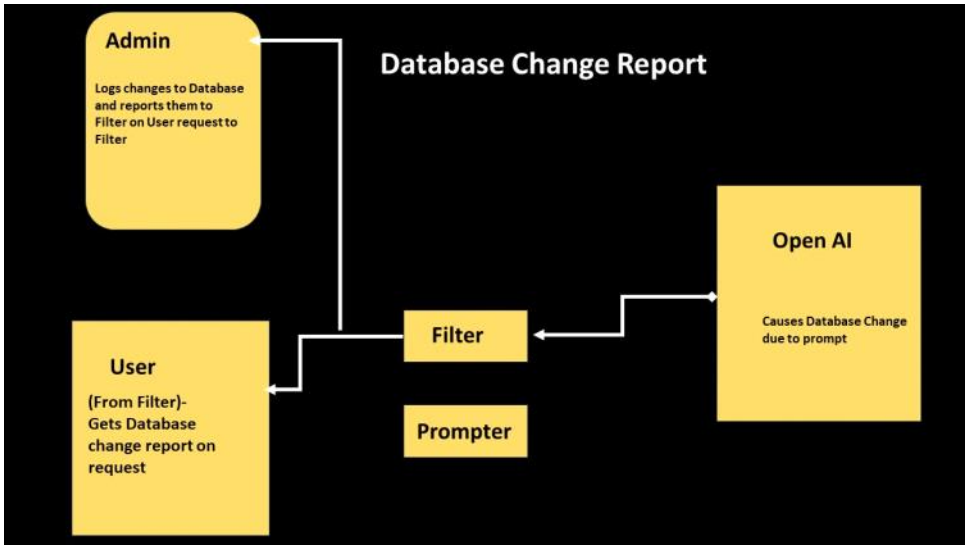
Functional Scenarios:



Situation 1: User Request Event
- The User initiates a request for learning content or information.
- The Filter evaluates the request. If relevant, it directs the request to the Prompter for OpenAI API interaction or to the Admin for database-related queries.
- The Prompter standardizes the request into a prompt according to pre-set rules and sends this standardized prompt to the OpenAI API. The response from OpenAI is sent directly to the Filter without pre-checks by the Prompter.
- The Admin responds to requests from the Filter about database management and report generation.



Situation 2: API Reply Event
- The OpenAI API sends a response back to the Prompter.
- The Prompter forwards this response directly to the Filter without standardization post-response.
- The Filter performs any necessary checks against the database content with the Admin's assistance, ensuring no duplication, particularly for the 'Word of the Day' feature.
- The Filter handles the storage of incomplete tests and the processing of quiz answers, communicating with the OpenAI API for results.



Situation 3: Database Change Report
- The Admin logs all database changes, including the addition of new words, the status of tests, changes in tags, and maintains a tag cloud.
- The User can request a report on database changes. The Filter facilitates this by retrieving the logged information from the Admin and presenting it to the User.
- Database changes are comprehensive, covering new additions, updates, and the overall landscape of the database's content.

Interaction Flow:
- User ↔ Filter: Direct communication channel for user inputs and receiving responses, learning materials, or reports.
- Filter ↔ Prompter ↔ OpenAI API**: A loop that ensures the OpenAI API receives standardized prompts and provides relevant responses.
- Filter ↔ Admin ↔ Database**: A route for database management, content verification, and change logging.

This system design outlines a structured approach to creating an educational app focused on Japanese language learning. It incorporates a blend of user interaction, content filtering, standardized communication with an AI

service, and robust database management to create an environment conducive to learning and growth.

The design is modular and allows for future expansions or modifications as the app evolves and user needs become more sophisticated.

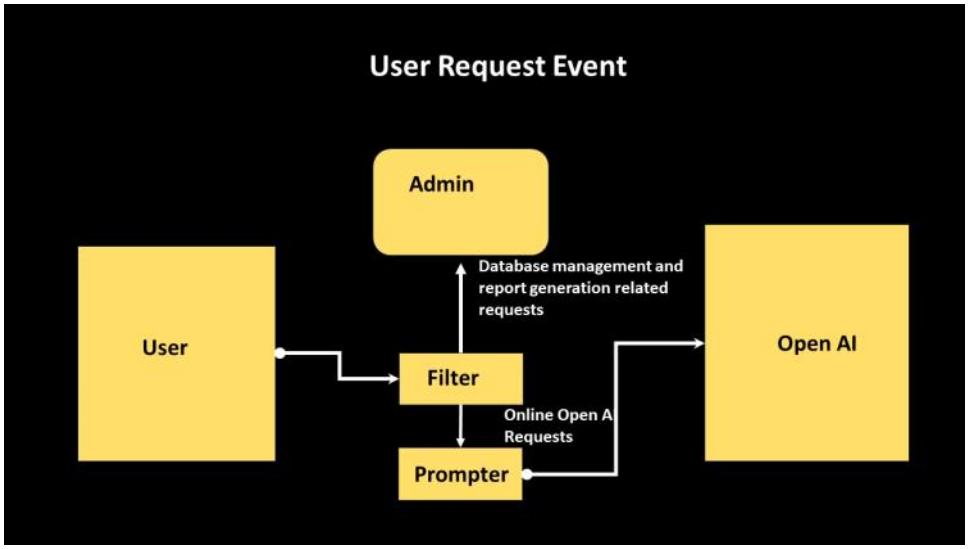
J.LO.AI v0.02

03 December 2023 15:59

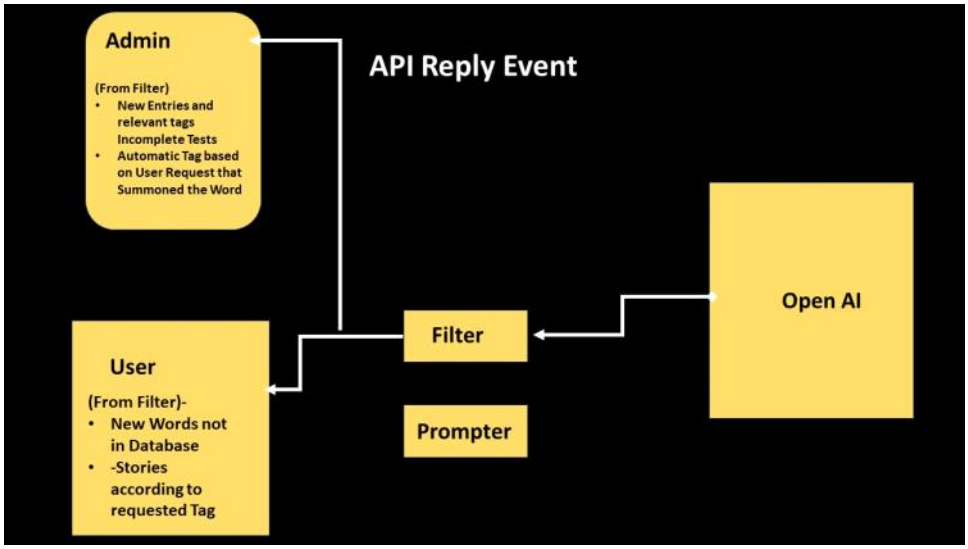
Consolidated System Design for J.LO.AI v0.02

Core Components:
1. User: The app's end-user, engaging with the system to learn Japanese.
2. Filter: The module responsible for filtering input and output, ensuring relevancy to the app's Japanese learning objectives, **and validating as well as sanitizing data for PUT requests.**
3. Admin: The backend component that manages the app's database, tracking and logging changes.
4. Prompter: The interface to the OpenAI API, crafting standardized prompts based on user requests and app requirements.

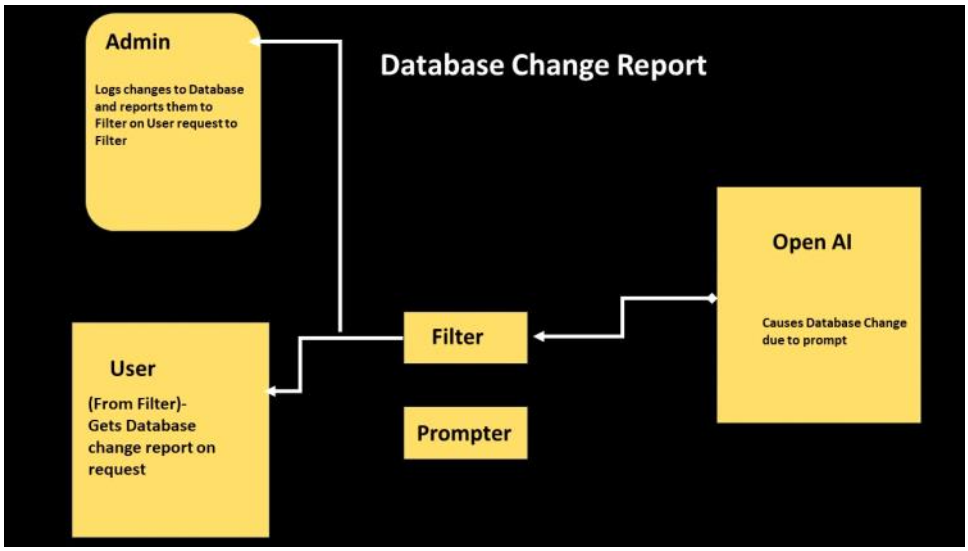
Functional Scenarios:



User Request Event
- The User initiates a request for learning content or information.
- The Filter evaluates the request. If relevant, it directs the request to the Prompter for OpenAI API interaction or to the Admin for database-related queries.
- For PUT requests involving user data updates, the Filter now also validates and sanitizes the request body to ensure secure and proper data handling before passing it to Admin.
- The Prompter standardizes the request into a prompt according to pre-set rules and sends this standardized prompt to the OpenAI API. The response from OpenAI is sent directly to the Filter without pre-checks by the Prompter.
- The Admin responds to requests from the Filter about database management and report generation.



API Reply Event
- The OpenAI API sends a response back to the Prompter.
- The Prompter forwards this response directly to the Filter without standardization post-response.
- The Filter performs any necessary checks against the database content with the Admin's assistance, ensuring no duplication, particularly for the 'Word of the Day' feature.
- The Filter handles the storage of incomplete tests and the processing of quiz answers, communicating with the OpenAI API for results.



Database Change Report
- The Admin logs all database changes, including the addition of new words, the status of tests, changes in tags, and maintains a tag cloud.
- The User can request a report on database changes. The Filter facilitates this by retrieving the logged information from the Admin and presenting it to the User.
- Database changes are comprehensive, covering new additions, updates, and the overall landscape of the database's content.

Interaction Flow:
- User ↔ Filter: Direct communication channel for user inputs and receiving responses, learning

materials, or reports. The Filter now also handles the sanitization and validation of PUT request data before it is passed to the Admin.
- Filter ↔ Prompter ↔ OpenAI API**: A loop that ensures the OpenAI API receives standardized prompts and provides relevant responses.
- Filter ↔ Admin ↔ Database**: A route for database management, content verification, and change logging. The Admin component now receives sanitized and validated data from the Filter for PUT requests.

This system design outlines a structured approach to creating an educational app focused on Japanese language learning. It incorporates a blend of user interaction, content filtering, standardized communication with an AI service, and robust database management to create an environment conducive to learning and growth.

The design is modular and allows for future expansions or modifications as the app evolves and user needs become more sophisticated. The addition of data validation and sanitization within the Filter component for PUT requests in v0.02 enhances the security and integrity of the system.

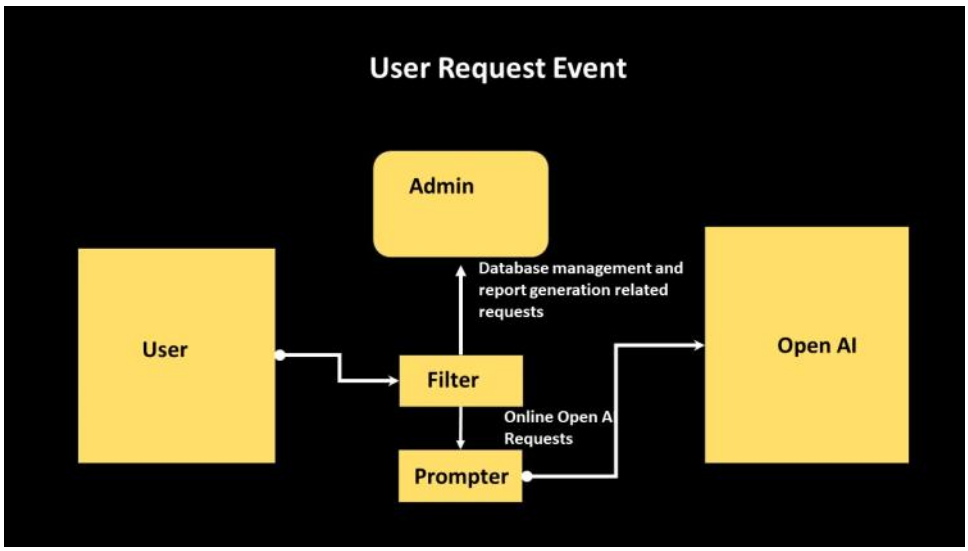
J.LO AI v0.03

03 December 2023 21:11

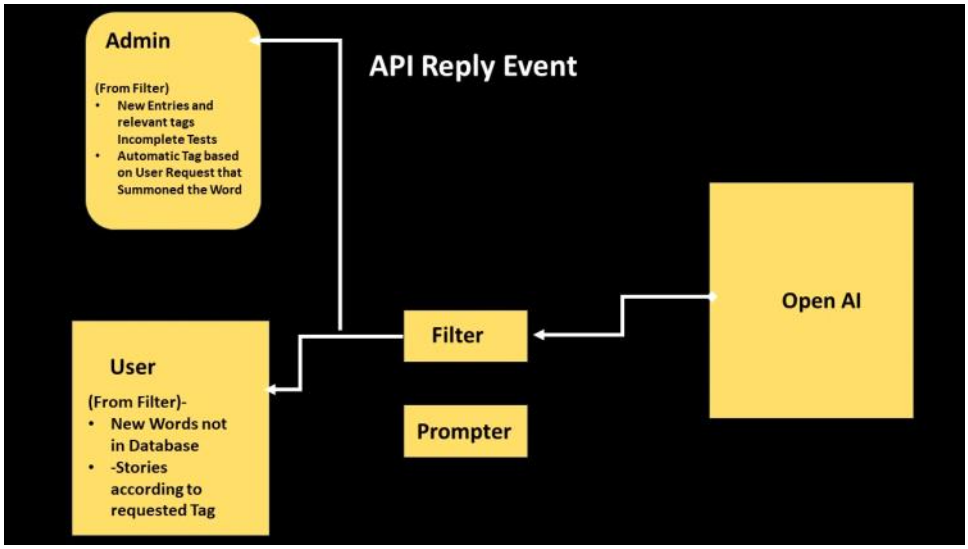
Consolidated System Design for J.LO.AI v0.03

Core Components:
1. **User:** The app's end-user, engaging with the system to learn Japanese.
2. **Filter:** The module responsible for filtering input and output, ensuring relevancy to the app's Japanese learning objectives, validating and sanitizing data for PUT requests, and managing report retrievals.
3. **Admin:** The backend component that manages the app's database, tracking and logging changes, and generating reports.
4. **Prompter:** The interface to the OpenAI API, crafting standardized prompts based on user requests and app requirements.

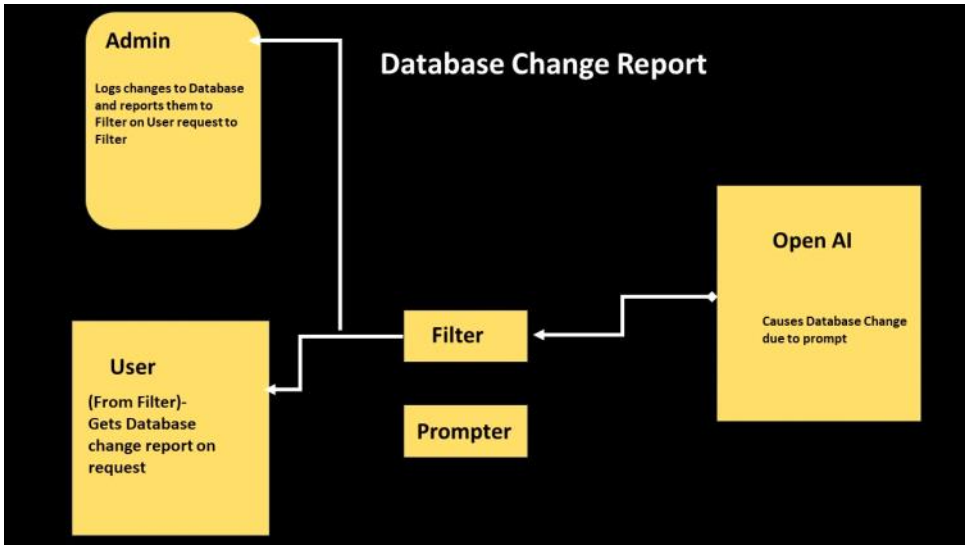
Functional Scenarios:



User Request Event:
- The User initiates a request for learning content or information.
- The Filter evaluates the request. If relevant, it directs the request to the Prompter for OpenAI API interaction or to the Admin for database-related queries.
- For PUT requests involving user data updates, the Filter now also validates and sanitizes the request body to ensure secure and proper data handling before passing it to Admin.
- The Prompter standardizes the request into a prompt according to pre-set rules and sends this standardized prompt to the OpenAI API. The response from OpenAI is sent directly to the Filter without pre-checks by the Prompter.
- The Admin responds to requests from the Filter about database management and report generation.



API Reply Event:
- The OpenAI API sends a response back to the Prompter.
- The Prompter forwards this response directly to the Filter without standardization post-response.
- The Filter performs any necessary checks against the database content with the Admin's assistance, ensuring no duplication, particularly for the 'Word of the Day' feature.
- The Filter handles the storage of incomplete tests and the processing of quiz answers, communicating with the OpenAI API for results.



Database Change Report:
- The Admin logs all database changes, including the addition of new words, the status of tests, changes in tags, and maintains a tag cloud.
- Reports are generated at scheduled intervals and upon application start, documenting all changes with references to system time.
- The User can request a report on database changes. The Filter facilitates this by retrieving the stored reports from the Admin and presenting it to the User.
- Database changes are comprehensive, covering new additions, updates, and the overall landscape of the database's content.

Interaction Flow:
- **User ↔ Filter:** Direct communication channel for user inputs and receiving responses, learning materials, or reports. The Filter now also handles the sanitization and validation of PUT request data before it is passed to the Admin.
- **Filter ↔ Prompter ↔ OpenAI API:** A loop that ensures the OpenAI API receives standardized prompts and provides relevant responses.

- **Filter ↔ Admin ↔ Database:** A route for database management, content verification, and change logging. The Admin component now receives sanitized and validated data from the Filter for PUT requests and manages report generation and accessibility.

This system design outlines a structured approach to creating an educational app focused on Japanese language learning. It incorporates a blend of user interaction, content filtering, standardized communication with an AI service, robust database management, and a comprehensive reporting system to create an environment conducive to learning and growth.

The design is modular and allows for future expansions or modifications as the app evolves and user needs become more sophisticated. The addition of data validation and sanitization within the Filter component for PUT requests, along with the integration of a reporting system in the Admin component, enhances the security, transparency, and integrity of the system.

Authors:
- Vyom
- MultiNEX12022023
- MultiNEX12032023

The authors are credited with conceptualizing and implementing the components and features of the J.LO.AI system, contributing to its continuous evolution towards a more refined and user-centric learning experience.

Admin: Commit History

03 December 2023 14:46

S.No	Branch	Commit Comment	Changes Done	Additional Changes	Remarks
1	main	Flask Setup	Hello World!		
2	main	Database tables and Flask Integration	Database tables and Flask Integration	Changed Hello World! To "Flask is Working!"	
3	crudop	CRUD0.1	CRUD Functionality for USERS, WORDS,and TAGS		User Sessions and Quiz planned post system integration
4	crudop	useriddisplay	User ID is now visible in R operation to display user ID		
5	crudop	JLO.AIv0.02	New System design based on new planned interaction between Fier and Admin	Comments on App.py to understand functionality	Changes to system design pertain to user security and "@" incompatibilities
6	crudop	CRUD0.1	CRUD Operation implimentation for User, Word, Tag is now complete and tested	All code files have been commented for study and understanding	Ready to merge to Master
7	Master	A.Crud0.01Complete	CRUD Operation implimentation for User, Word, Tag is now complete and tested	All code files have been commented for study and understanding	Merged to Master, admin component now has functionality

Main and Master Branches

03 December 2023 00:08

Great to hear that you successfully pushed your repository to GitHub! Regarding your question about the "main" and "master" branches:

In Git, a branch is essentially a separate line of development. You can think of it as a way to diverge from the main codebase, allowing you to work on changes or new features without affecting the main or stable code.

Historical Context of "master":

- **Original Naming**: Traditionally, the default branch in Git has been named "master". When you initialize a new Git repository, it automatically creates this default branch.
- **Master in Context**: In the context of Git, "master" is intended to serve as the primary or 'master' copy of your project. It's the default branch where the final version of your project resides.

Shift to "main":

- **Recent Changes**: More recently, there's been a shift in the software community to replace "master" with "main" as the default branch name. This change has been primarily driven by the recognition that the term "master" can be associated with slavery, which is offensive and exclusionary.
- **GitHub's Adoption**: Platforms like GitHub have adopted "main" as the new default name for the initial branch when you create a new repository. This change started around October 2020.

Technical Difference:

- **Functionally the Same**: From a technical perspective, there is no difference between a branch named "main" and one named "master". They both function in the same way as the default branch in a Git repository.
- **Naming Convention**: The only difference is in the naming convention. If you're starting a new repository, it's becoming increasingly common to use "main" as the default branch name.

Working with Either Branch:

- **Checking Your Default Branch**: You can check the name of your default branch by initializing a new Git repository (`git init`) and using `git branch` to see the name of the initial branch.
- **Renaming Branches**: If you have an existing repository and wish to change the default branch from "master" to "main" (or vice versa), you can do so using the `git branch -m` command.

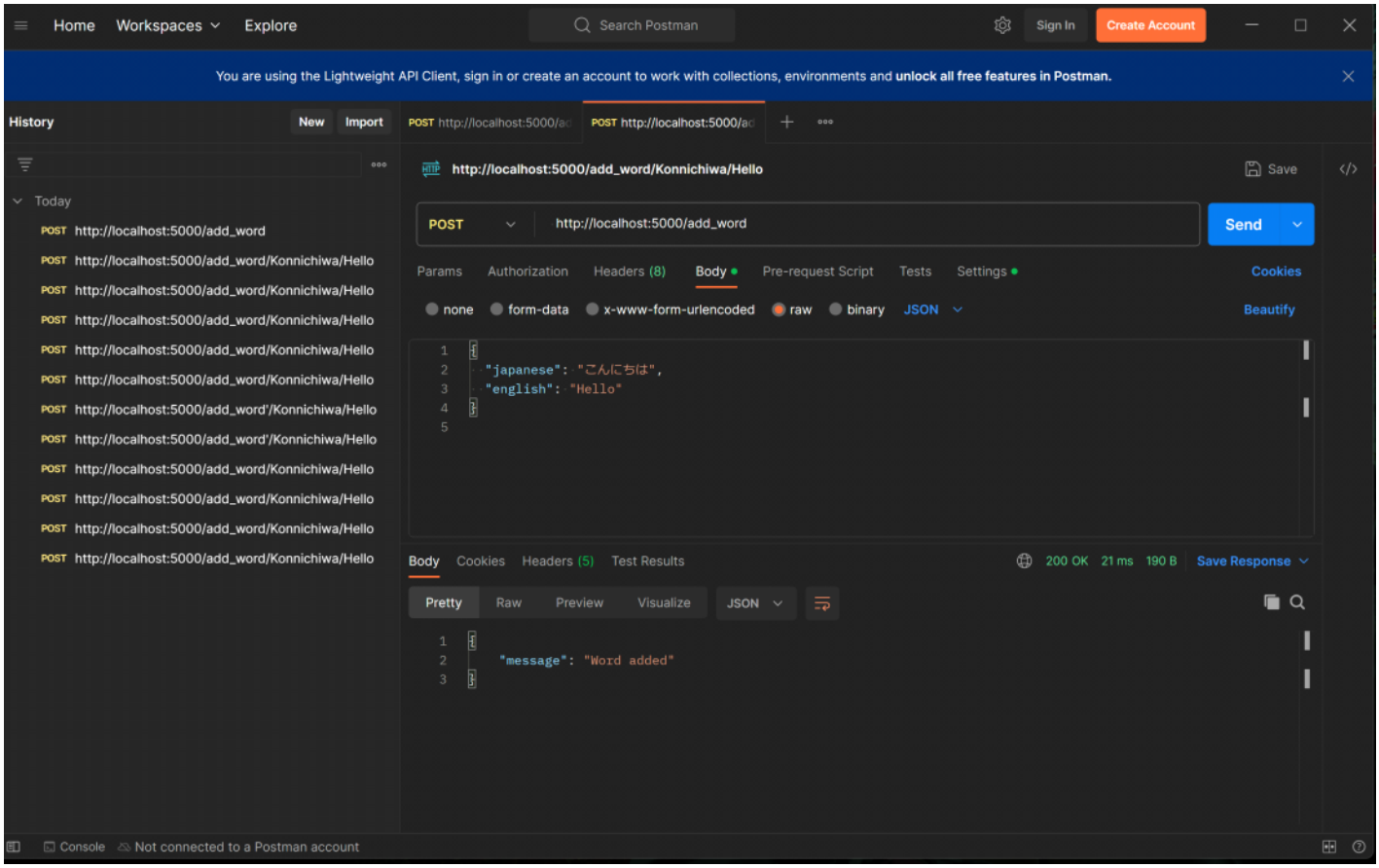
In summary, "main" and "master" are just different names for the default branch in a Git repository. The shift from "master" to "main" is a move towards more inclusive language in the tech community. The choice of branch name doesn't affect the functionality of your Git repository.

Word model:Admin database Component: Testing

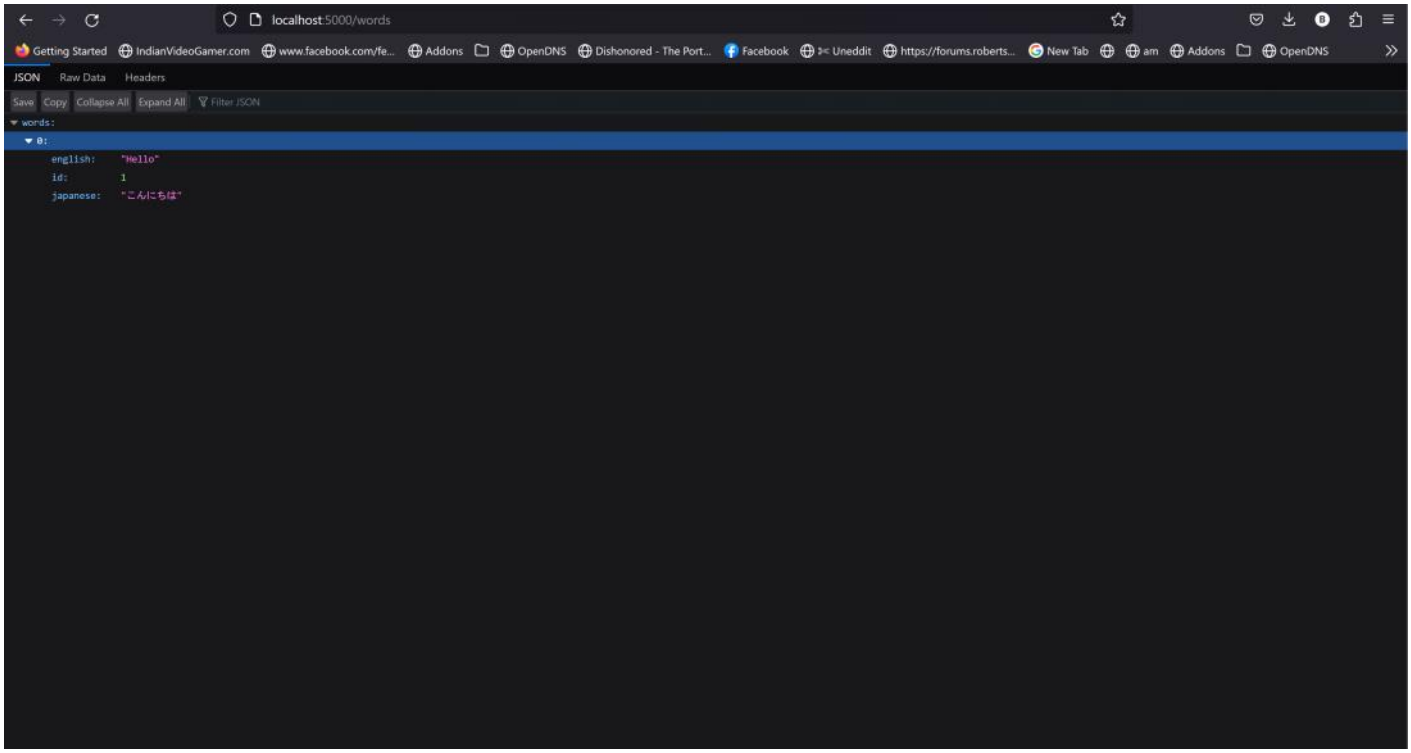
03 December 2023 11:42

1.Database Testing

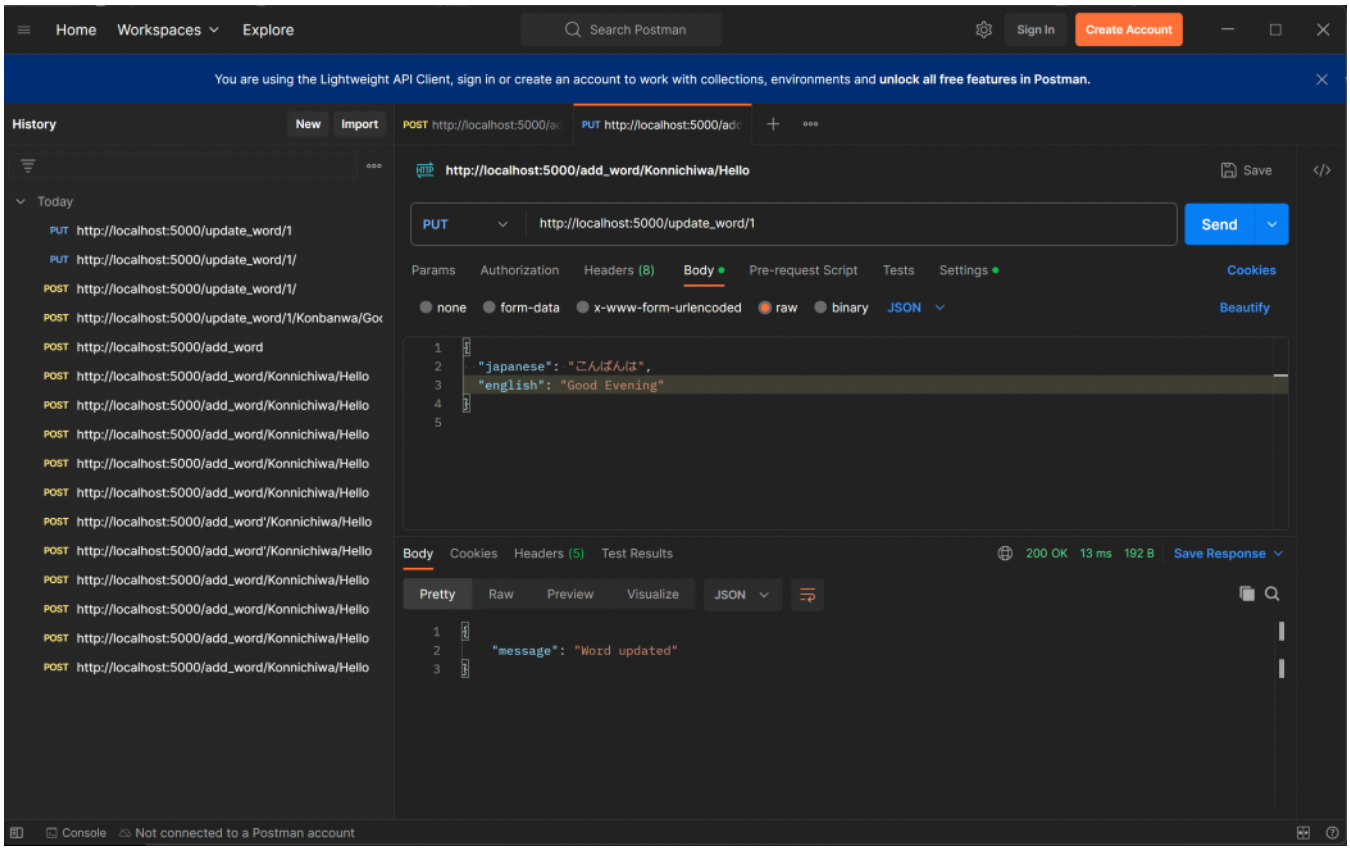
a.Add word (C)



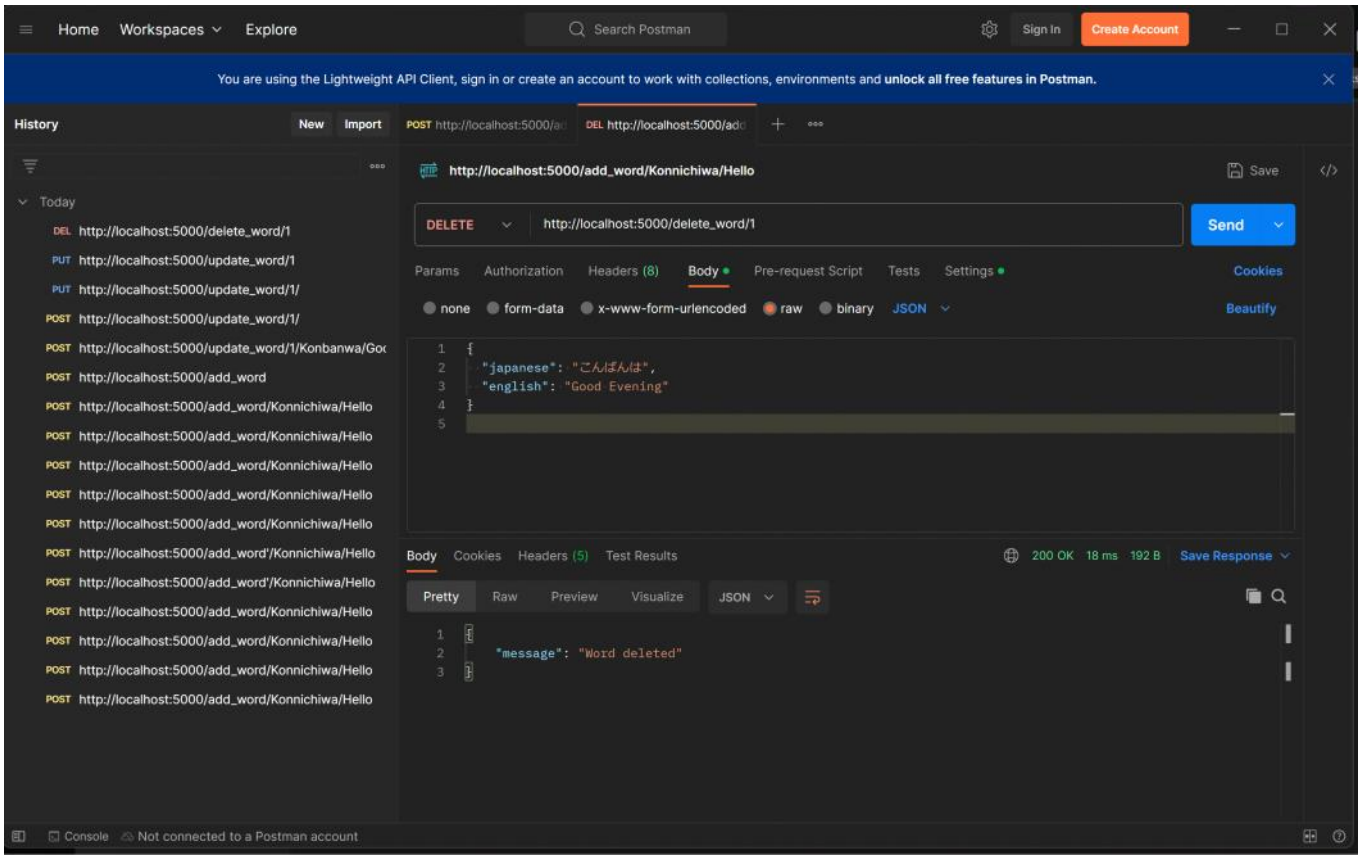
b.Get words (R)



C. UPDATE WORD (U)



C. DELETE WORD (D)

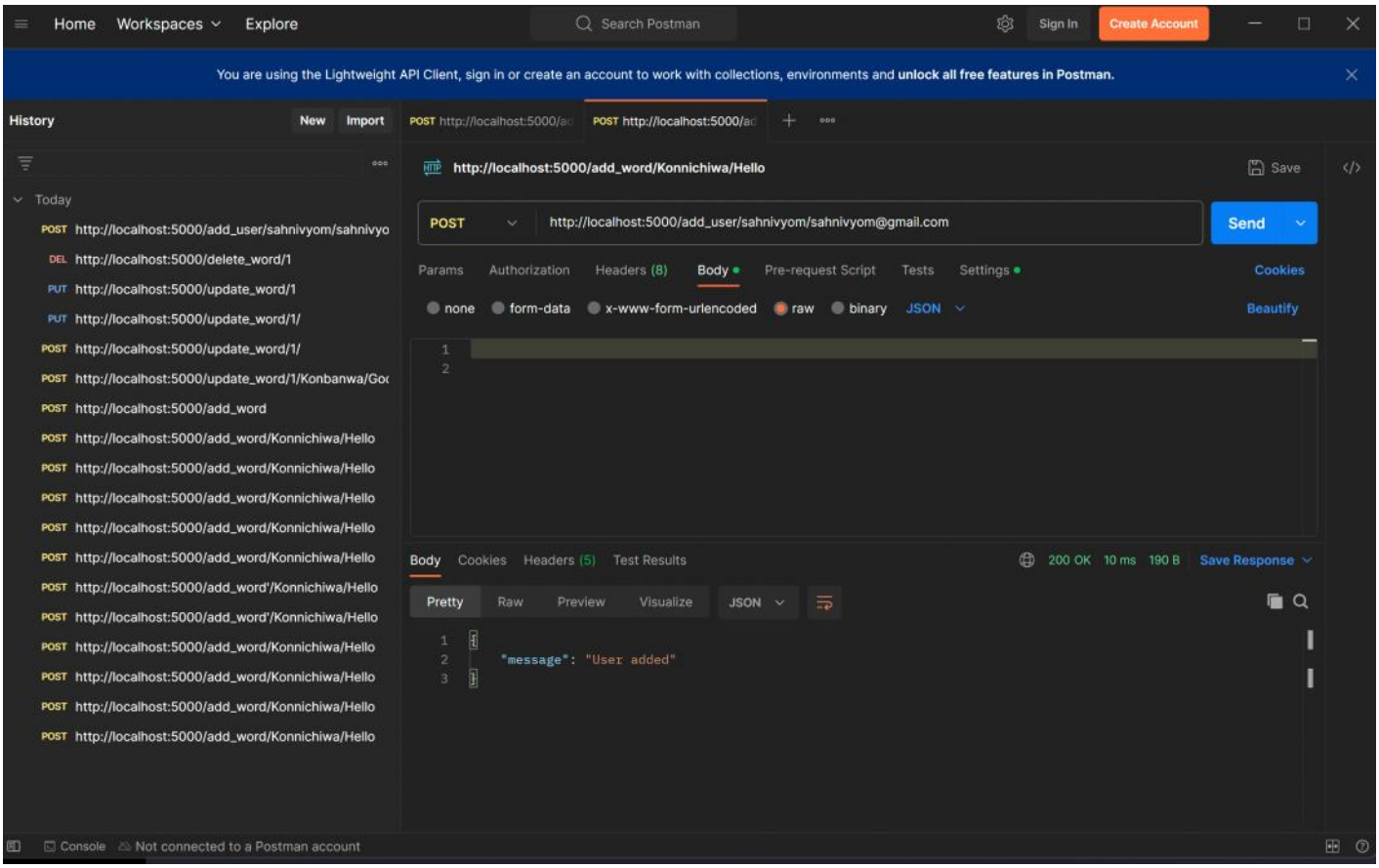

Home
Workspaces
Explore
Search Postman
Sign In
Create Account
You are using the Lightweight API Client, sign in or create an account to work with collections, environments and unlock all free features in Postman.
History
New
Import
Today
DEL http://localhost:5000/delete_word/1
PUT http://localhost:5000/update_word/1
PUT http://localhost:5000/update_word/1/
POST http://localhost:5000/update_word/1/
POST http://localhost:5000/update_word/1/Konbanwa/Goo
POST http://localhost:5000/add_word
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word'/Konnichiwa/Hello
POST http://localhost:5000/add_word'/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
DEL http://localhost:5000/add
http://localhost:5000/add_word/Konnichiwa/Hello
Save
DELETE
http://localhost:5000/delete_word/1
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
JSON
Beautify
{
"japanese": "こんばんは",
"english": "Good Evening"
}
Body
Cookies
Headers (5)
Test Results
200 OK
18 ms
192 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"message": "Word deleted"
}
Console
Not connected to a Postman account

User Model:Admin Database

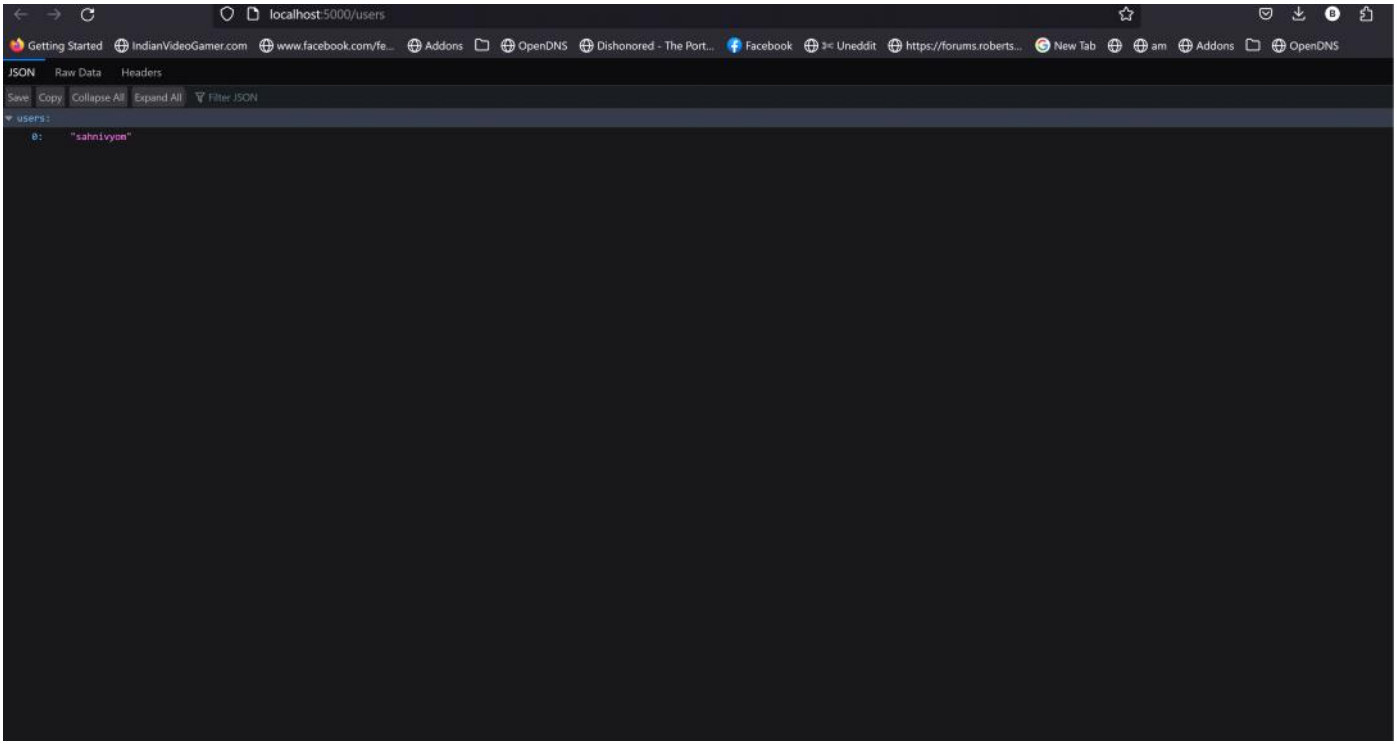
03 December 2023 14:14

1.Add User (C)

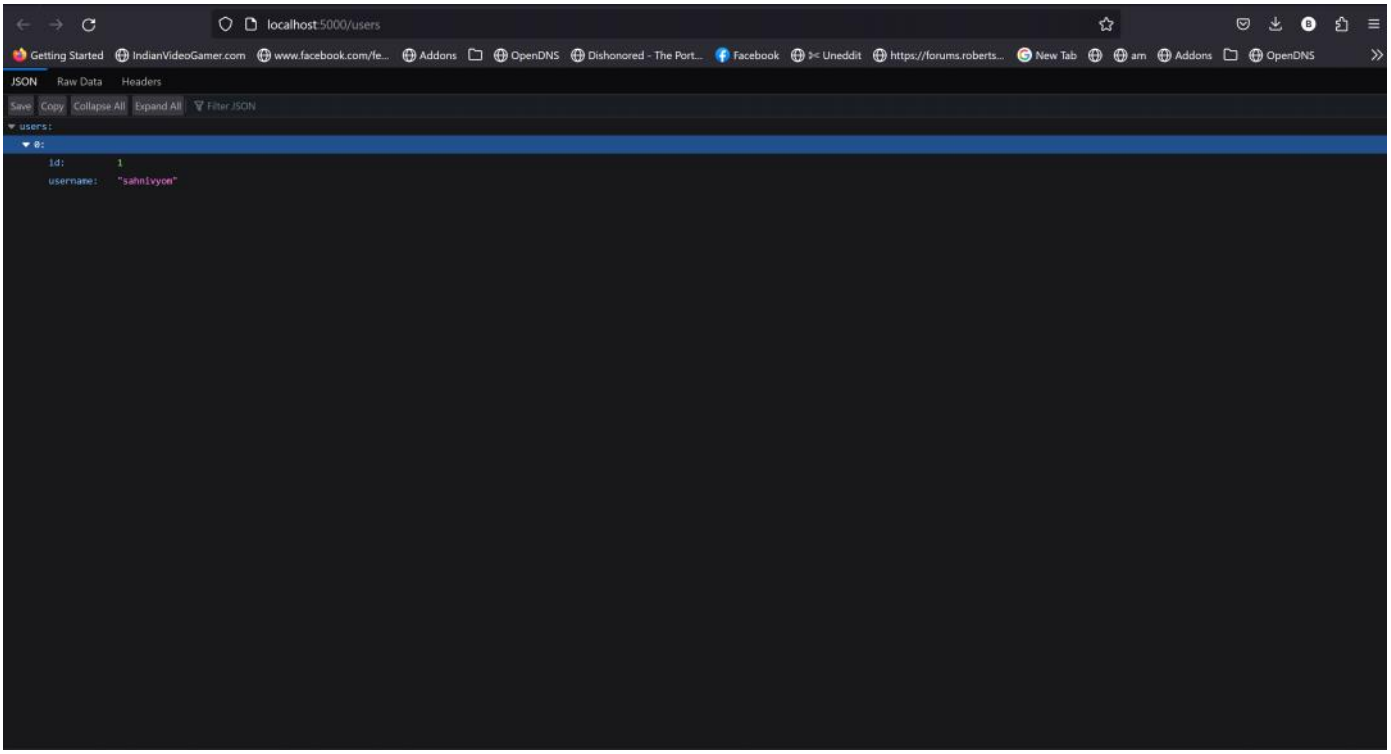
http://localhost:5000/add_user/sahnivyom/sahnivyom@gmail.com



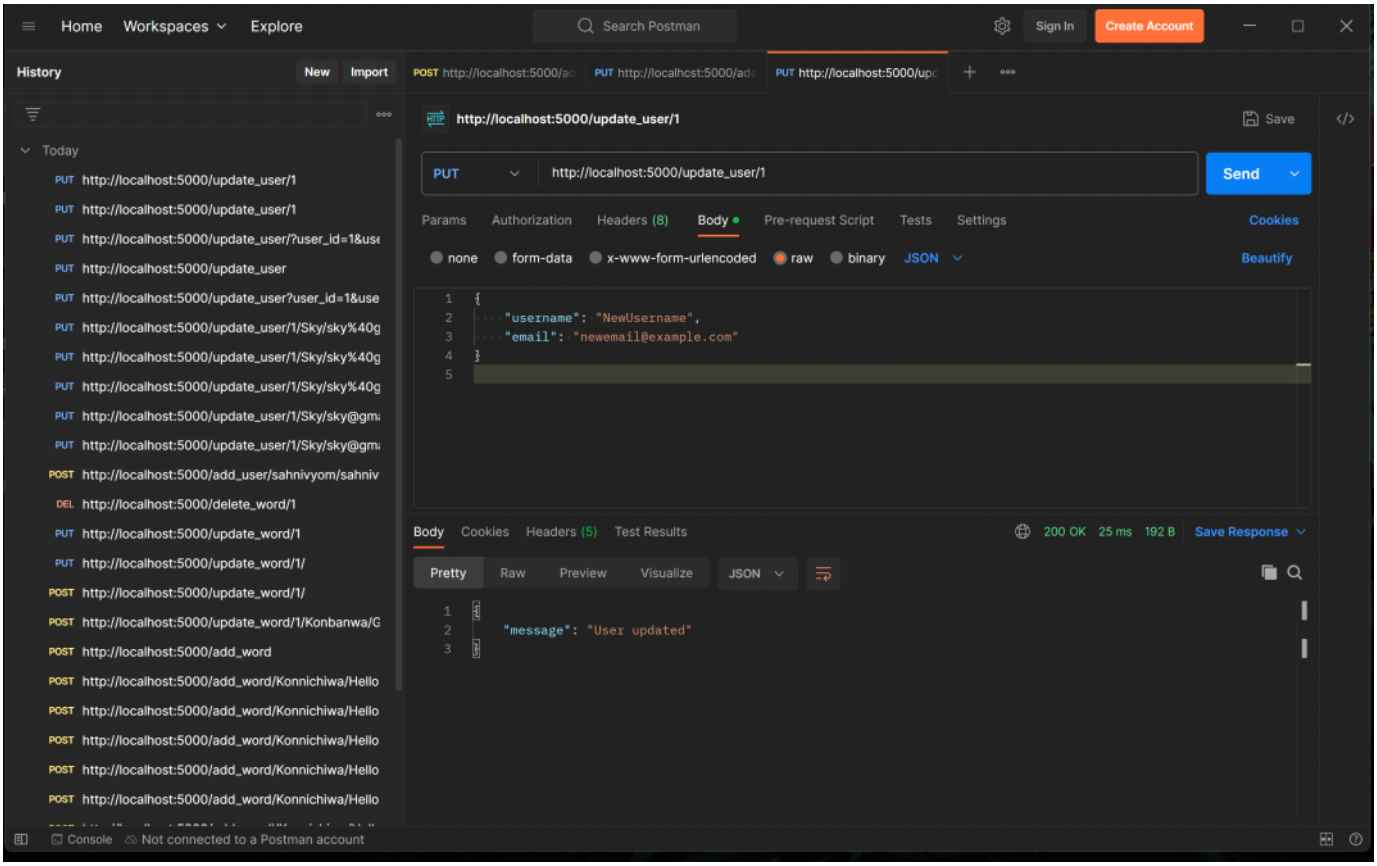
2.GET User (R)



##Added additional Functionality to display User ID 03122023 Also Tested

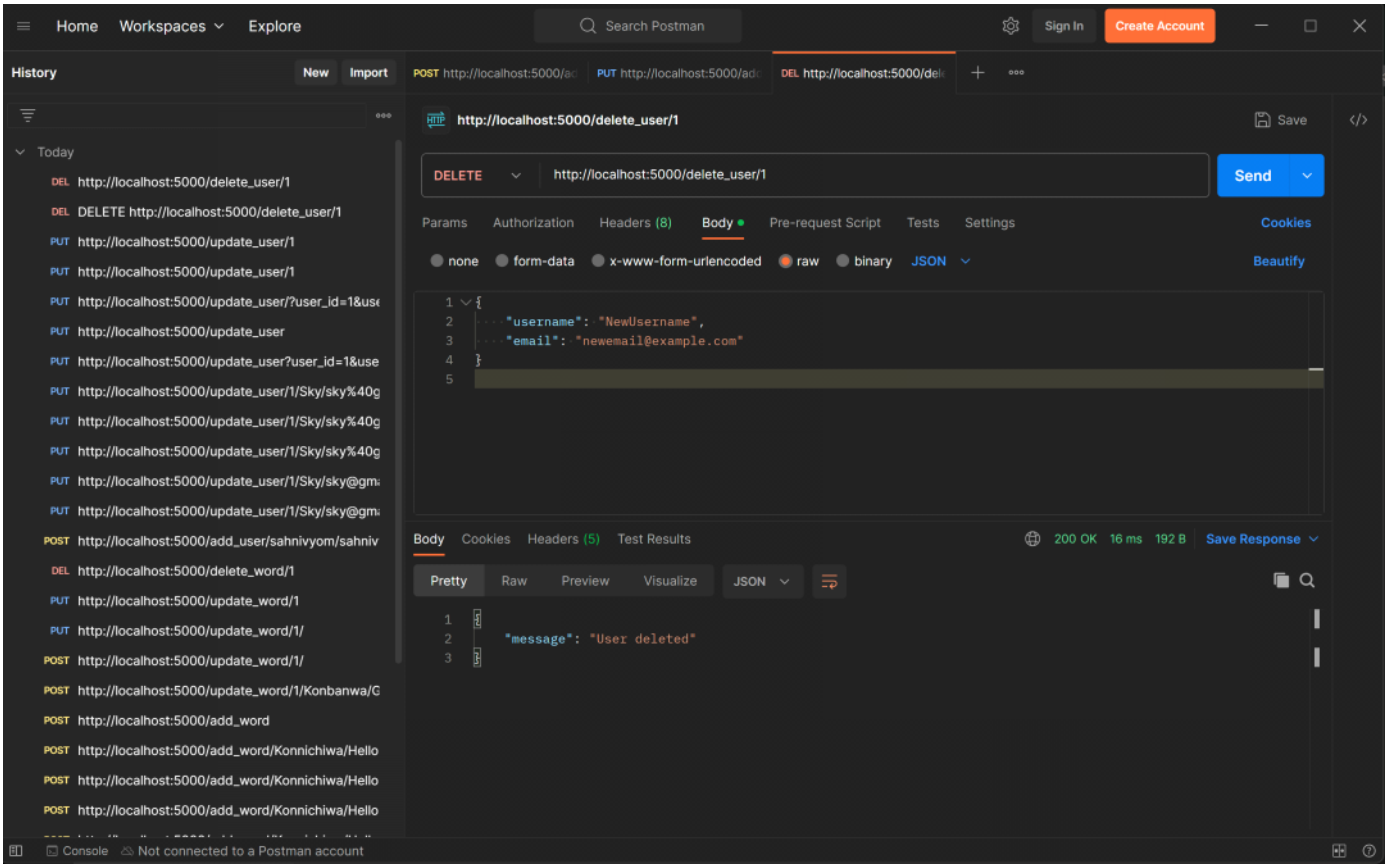


3.Update User (U)



1. Lead to new FILTER->Admin interaction functionality incorporated in System Design v0.02
2. Current functionality as basic as it get
3. Refer relevant doc in Study Notebook

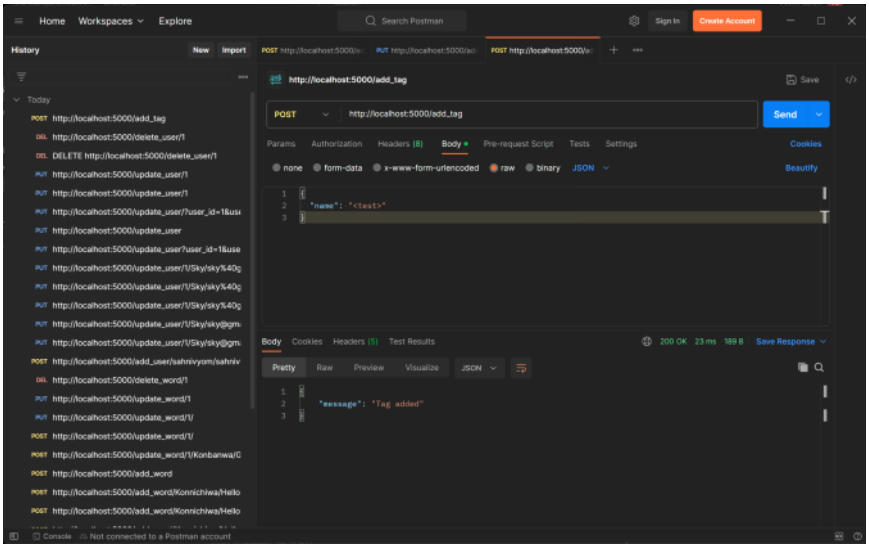
3.Delete User (U)


Home
Workspaces
Explore
Search Postman
Sign In
Create Account
History
New
Import
Today
DEL http://localhost:5000/delete_user/1
DEL DELETE http://localhost:5000/delete_user/1
PUT http://localhost:5000/update_user/1
PUT http://localhost:5000/update_user/1
PUT http://localhost:5000/update_user/?user_id=1&use
PUT http://localhost:5000/update_user
PUT http://localhost:5000/update_user?user_id=1&use
PUT http://localhost:5000/update_user/1/Sky/sky%40g
PUT http://localhost:5000/update_user/1/Sky/sky%40g
PUT http://localhost:5000/update_user/1/Sky/sky%40g
PUT http://localhost:5000/update_user/1/Sky/sky@gm
PUT http://localhost:5000/update_user/1/Sky/sky@gm
POST http://localhost:5000/add_user/sahnivyom/sahniv
DEL http://localhost:5000/delete_word/1
PUT http://localhost:5000/update_word/1
PUT http://localhost:5000/update_word/1/
POST http://localhost:5000/update_word/1/
POST http://localhost:5000/update_word/1/Konbanwa/G
POST http://localhost:5000/add_word
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
POST http://localhost:5000/add_word/Konnichiwa/Hello
http://localhost:5000/delete_user/1
Save
DELETE
http://localhost:5000/delete_user/1
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
JSON
Beautify
{
"username": "NewUsername",
"email": "newemail@example.com"
}
Body
Cookies
Headers (5)
Test Results
200 OK
16 ms
192 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"message": "User deleted"
}
Console
Not connected to a Postman account

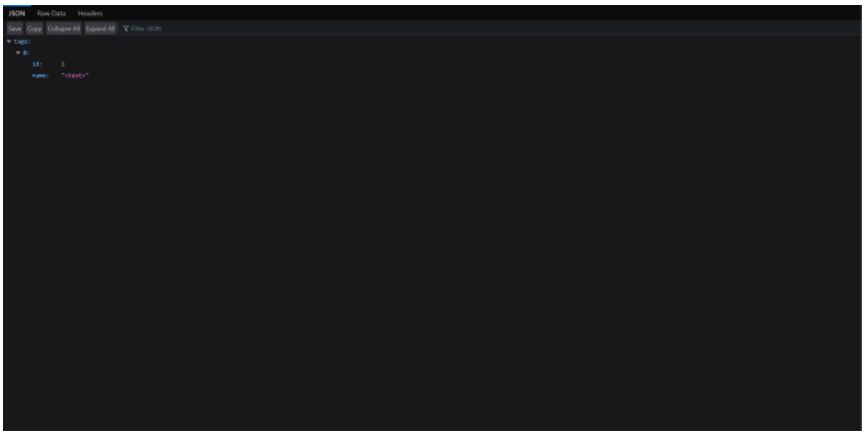
Tag Model: Admin Database

03 December 2023 18:57

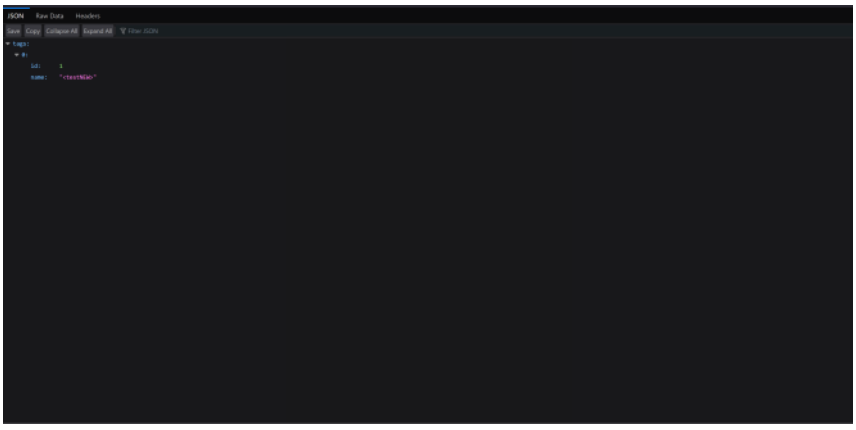
1.ADD TAG (C)



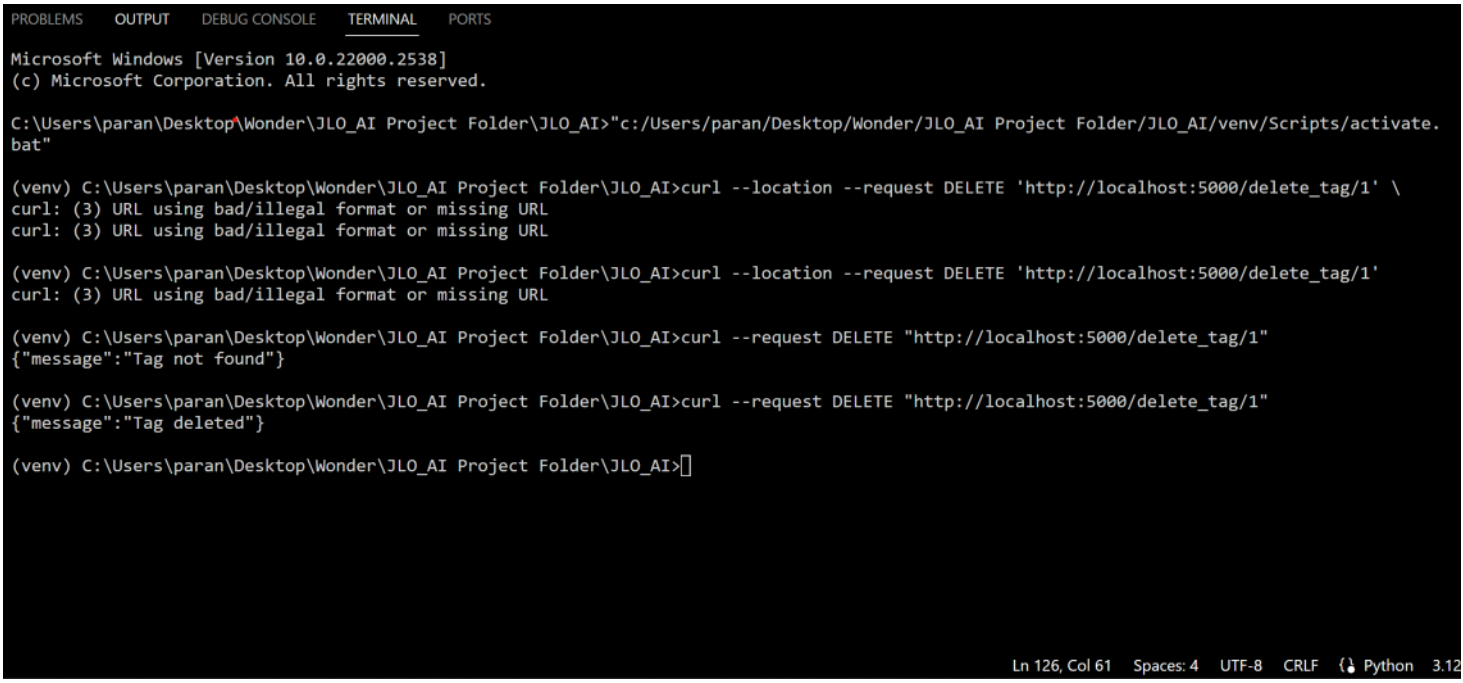
2.GET TAG(R)



3.UPDATE TAG(U)



3.DELETE TAG(D)



Required following change in admin.py

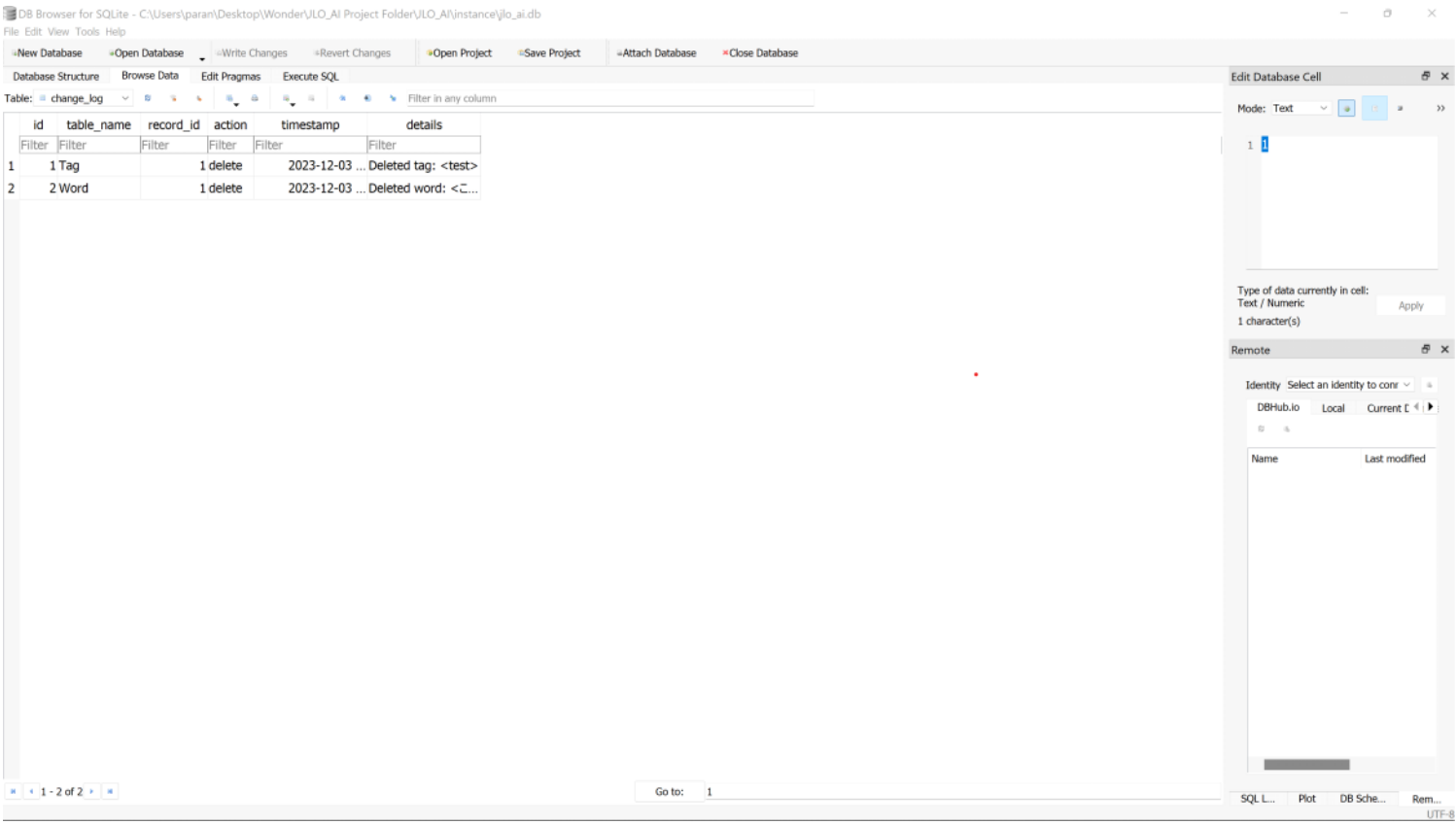
```
def delete_tag(tag_id):
    # Retrieve a Tag object by its ID.
    tag = Tag.query.get(tag_id)
    # If the tag exists, remove it from the database.
    if tag:
        db.session.delete(tag)
        # Save the changes to the database by committing the transaction.
        db.session.commit()
        return True  # Indicate that the tag was successfully deleted.
    return False  # If the tag does not exist, return False.
```

Admin Beta: Changelog CRUD Implementation

03 December 2023 23:22

Passed! 3rd December 2023

This ensures that a changelog table is mantained that tracks all CUD changes in a changelog table



Test Manual: Admin Beta

03 December 2023 20:57

Creating a comprehensive testing manual involves outlining the test cases for each functional aspect of your Admin component that you've developed, including the logging of database changes and report generation. Here's a guide on how to approach this:

Testing Manual for Admin Component

Test Case 1: Database Change Logging

Objective: Ensure that every change to the database (insert, update, delete) is logged correctly.

Steps:
1. Perform an insert operation on the database.
2. Verify that a corresponding log entry is created in the `ChangeLog` table with the action 'insert'.
3. Repeat for update and delete operations.

Expected Result: Each operation should result in an accurate log entry reflecting the change.

Test Case 2: Report Generation Functionality
Objective: Ensure that the `generate_report` function creates a comprehensive report of all changes logged.

Steps:
1. Call the `generate_report` function after performing a series of database changes.
2. Check that the report generated includes all the changes made, with correct timestamps and details.

Expected Result: The report should accurately reflect all database changes in the order they occurred.

Test Case 3: Report Saving Mechanism
Objective: Confirm that reports are saved to the database or file system correctly.

Steps:
1. After generating a report, check that it is saved in the chosen storage method (database or file system).
2. For database storage, verify the new entry in the `Report` table.
3. For file system storage, check that a new file is created with the correct timestamp in the name.

Expected Result: The report should be retrievable from the storage method used.

Test Case 4: Scheduled Report Generation
Objective: Verify that reports are generated and saved at scheduled intervals.

Steps:
1. Set up the scheduler to run the report generation job at a short interval for testing purposes.
2. Observe the system for a few intervals to ensure reports are generated without manual intervention.

Expected Result: New reports should be created at each interval.

Test Case 5: Report Retrieval via API Endpoint
Objective: Test the API endpoint responsible for providing the latest report to the user.

Steps:
1. Make a request to the `/get_latest_report` endpoint.
2. Verify that the response contains the latest report data.

Expected Result: The endpoint should return the latest report.

Test Case 6: Authentication and Authorization for Report Retrieval
Objective: Ensure that only authorized users can access the report retrieval endpoint.

Steps:
1. Attempt to access the `/get_latest_report` endpoint without proper authentication.
2. Implement authentication and repeat the access attempt with proper credentials.

Expected Result: Unauthorized attempts should be rejected, while authenticated requests should succeed.

Test Case 7: Report Content Validation
Objective: Verify the content of the reports for accuracy and completeness.

Steps:
1. Compare the report content against the known changes made to the database to ensure all changes are reflected.
2. Look for any discrepancies or missing information.

Expected Result: The report should accurately document all changes with no missing entries.

Checkpoints for Creating Test Cases:
- After implementing a new feature or change in the Admin component.
- When changes are made to the database schema.
- Following the introduction of new functionalities like report generation or retrieval.
- After any major refactoring or code changes.

Testing Tools and Methods
- **Manual Testing**: Use tools like Postman for API testing and SQL clients for database validation.
- **Automated Testing**: Write unit tests for individual functions and integration tests for workflows.

Record Keeping
- Document the outcome of each test case.
- Keep a log of any bugs or issues discovered during testing.
- Track the resolution of these issues.

By following this testing manual, you can methodically validate the functionality of your Admin component and ensure the reliability and correctness of the features you've implemented. Remember that testing should be an ongoing process, with new test cases added as the application grows and evolves.

Test Manual:CRUD0.01

03 December 2023 14:10

User Model

1. Add User (POST)
```http
POST http://localhost:5000/add_user/<username>/<email>
```
- Replace `<username>` and `<email>` with actual values.
- Example URL: `http://localhost:5000/add_user/johndoe/john@example.com`

2. Get Users (GET)
```http
GET http://localhost:5000/users
```

3. Update User (PUT)
```http
PUT http://localhost:5000/update_user/<user_id>/<new_username>/<new_email>
```
- Replace `<user_id>`, `<new_username>`, and `<new_email>` with actual values.
- Example URL: `http://localhost:5000/update_user/1/janedoe/jane@example.com`

4. Delete User (DELETE)
```http
DELETE http://localhost:5000/delete_user/<user_id>
```
- Replace `<user_id>` with the actual user ID.
- Example URL: `http://localhost:5000/delete_user/1`

Word Model

1. Add Word (POST)
```http
POST http://localhost:5000/add_word
Content-Type: application/json

{
  "japanese": "<japanese_word>",
  "english": "<english_translation>"
}
```
- Example Body: `{"japanese": "こんにちは", "english": "Hello"}`

2. Get Words (GET)
```http
GET http://localhost:5000/words
```

3. Update Word (PUT)

```http
PUT http://localhost:5000/update_word/<word_id>
Content-Type: application/json

{
  "japanese": "<new_japanese_word>",
  "english": "<new_english_translation>"
}
```
- Replace `<word_id>` with the actual word ID.
- Example URL: `http://localhost:5000/update_word/1`
- Example Body: `{"japanese": "さようなら", "english": "Goodbye"}`

4. Delete Word (DELETE)
```http
DELETE http://localhost:5000/delete_word/<word_id>
```
- Replace `<word_id>` with the actual word ID.
- Example URL: `http://localhost:5000/delete_word/1`

Tag Model

1. Add Tag (POST)
```http
POST http://localhost:5000/add_tag
Content-Type: application/json

{
  "name": "<tag_name>"
}
```
- Example Body: `{"name": "Learning"}`

2. Get Tags (GET)
```http
GET http://localhost:5000/tags
```

3. Update Tag (PUT)
```http
PUT http://localhost:5000/update_tag/<tag_id>
Content-Type: application/json

{
  "name": "<new_tag_name>"
}
```
- Replace `<tag_id>` with the actual tag ID.
- Example URL: `http://localhost:5000/update_tag/1`
- Example Body: `{"name": "Advanced"}`

4. Delete Tag (DELETE)
```http
DELETE http://localhost:5000/delete_tag/<tag_id>
```
- Replace `<tag_id>` with the actual tag ID.

- Example URL: `http://localhost:5000/delete_tag/1`

Notes
- Make sure your Flask app is running and accessible at `http://localhost:5000`.
- For POST and PUT requests, ensure that the `Content-Type` header is set to `application/json` and include the respective JSON body in the request.
- Modify the placeholders with the appropriate data for your tests.