Lec 8 - Managing database and API with Python-v1

November 8, 2024

0.1 Managing Database and API Using Python

- A **highly practical topic** in software development projects.

1 Objectives

1.1 Background

- **Database Management System (DBMS)**:
- **SQL Language**:
- **SQLite3**:

1.2 Main Topic

- **Managing SQLite3 Database with Python**:
  - Techniques for connecting, querying, and manipulating an SQLite3 database using Python.
- **Building a Python Flask API**:
  - Developing a simple Flask API to facilitate interaction between the SQLite3 database and a user interface.

2 Database Management System (DBMS)

2.1 What is a DBMS?

A **Database Management System (DBMS)** is software that provides an interface for interacting with databases and manipulating the data stored within them.

2.1.1 Key Functions of a DBMS:

- Acts as an intermediary between users and the database.
- Allows users to:
  - Store
  - Retrieve
  - Update
  - Manage data

2.1.2 Benefits:

- Ensures data is handled in a structured and efficient manner.

2.1.3 Choosing a DBMS:

- The choice depends on the application's requirements, including:
  - Data volume
  - Complexity
  - Scalability
  - Performance needs

2.2 Popular DBMS Examples:

- **MySQL**
- **PostgreSQL**
- **SQLite**

3 SQLite

QLite is considered a portable DBMS. It is designed to be lightweight, self-contained, and serverless, making it highly portable

3.1 Where is SQLite Used?

- Commonly used in **IoT and embedded systems**.
- Ideal for scenarios where resources are limited and simplicity is essential.

3.1.1 Applications in IoT:

- **Local Data Storage**
- **Configuration Management**
- **Logging**
- **Sensor Data Storage**

3.2 Why Use a Database Instead of a Spreadsheet or CSV?

3.2.1 Advantages of Using Databases:

- **Scalability**:
  - Better handling of growing data compared to spreadsheets or CSVs.
- **Data Retrieval and Analysis**:
  - Powerful querying capabilities using SQL for complex searches, aggregations, and analyses.
- **Security**:
  - Enhanced control over data access and protection.
- **Data Integrity**:
  - Ensures data consistency and reduces redundancy.

4 SQL (Structured Query Language)

4.1 What is SQL?

- **SQL** is the standard language used for interacting with **relational databases**.

4.2 Variations in SQL

- Different DBMSs may have slight variations in:
  - **Syntax**
  - **Supported features**
- Variations often relate to:
  - Specific functions
  - Extensions
  - Optimizations unique to each system

4.3 Core SQL Commands

- Consistent across all relational databases:
  - **SELECT**: Retrieve data
  - **INSERT**: Add new data
  - **UPDATE**: Modify existing data
  - **DELETE**: Remove data
  - **CREATE**: Create database objects

5 Example 1. Managing Database with SQLite3

5.1 Installing SQLite3

To install SQLite3 on your system, use the following command:

```
sudo apt-get install sqlite3
```

5.2 Creating or Opening a Database

To create a new database or open an existing one, use the following command:

```
sqlite3 employees.db
```

- This command will create a new database file named **employees.db** if it does not already exist.
- If the file already exists, it will open that database for you to interact with. “‘

5.3 Core SQL Commands with Examples

5.3.1 CREATE: Create Database Objects

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    department TEXT,
    salary REAL
);
```

5.3.2 INSERT: Add New Data

```
    INSERT INTO employees (name, department, salary)
    VALUES ('John Doe', 'HR', 50000);
```

5.3.3 SELECT: Retrieve Data

```
SELECT * FROM employees WHERE department = 'Sales';
```

5.3.4 UPDATE: Modify Existing Data

```
UPDATE employees
SET salary = 55000
WHERE name = 'John Doe';
```

5.3.5 DELETE: Remove Data

```
DELETE FROM employees
WHERE department = 'Sales';
```

6 Example 2. Managing SQLite Database with Python

Python provides built-in support for SQLite through the `sqlite3` module in the standard library. This module allows you to interact with SQLite databases directly from your Python code, enabling various database management tasks.

6.1 Overview of Managing SQLite Databases

1. **Connecting to a Database**:
   - Use the `sqlite3.connect()` function to establish a connection, specifying the path to the database file.
2. **Creating a Cursor**:
   - After establishing a connection, create a cursor object using the `cursor()` method. The cursor is used to execute SQL queries and fetch results.
3. **Executing SQL Queries**:
   - Execute SQL queries with the `execute()` method of the cursor object. This allows for creating tables, and inserting, updating, or deleting data.
4. **Committing Transactions**:
   - Use the `commit()` method of the connection object to save changes permanently after executing SQL queries.
5. **Closing the Connection**:
   - Close the connection using the `close()` method of the connection object to release resources.

6.2 Example Code

“‘python import sqlite3

7 Connect to the SQLite database (creates a new database if it doesn’t exist)

conn = sqlite3.connect(‘employee.db’)

8 Create a cursor object

```
cursor = conn.cursor()
```

9 Create a table

```
cursor.execute(“’CREATE TABLE IF NOT EXISTS employees (id INTEGER PRIMARY KEY, name TEXT NOT NULL, department TEXT, salary REAL)”’)
```

10 Insert some data into the table

```
cursor.execute(‘INSERT INTO employees(name, department, salary) VALUES (?, ?, ?)’, (‘Alice’, ‘Sales’, 40000)) cursor.execute(‘INSERT INTO employees(name, department, salary) VALUES (?, ?, ?)’, (‘Bob’, ‘HR’, 40000))
```

11 Commit the transaction

```
conn.commit()
```

12 Execute a query to fetch data

```
cursor.execute(“SELECT * FROM employees”) rows = cursor.fetchall() for row in rows: print(row)
```

13 Execute a query to fetch certain data

```
cursor.execute(“SELECT * from employees WHERE id = ?”, (1,)) row = cursor.fetchone() print(row)
```

14 Delete specific user

```
cursor.execute(“DELETE FROM employees WHERE department = ?”, (“Sales”,))
```

15 updating user

```
cursor.execute(“UPDATE employees SET salary = 55000 WHERE name = ?”, (‘John Doe’,))
```

16 Close the cursor and connection

```
cursor.close() conn.close()
```

17 API (Application Programming Interface)

- An **API** is a set of protocols, tools, and definitions that allow different software applications and systems to communicate and interact with each other.

17.1 Building API with Python

17.1.1 Flask

- **Flask** is a lightweight and flexible web framework for Python, providing tools, libraries, and patterns to help developers build web applications quickly and efficiently.

- Flask is commonly used for building APIs (Application Programming Interfaces) in addition to web applications. Its simplicity, flexibility, and minimalistic design make it well-suited for creating RESTful APIs.

17.2 Key Features of Flask for Building APIs

17.2.1 Routing and Endpoint Definition:

- Flask allows you to define **routes (URL patterns)** that map to specific **functions**, which handle incoming HTTP requests.

- These functions(often referred to as view functions or **endpoints**) can perform various tasks such as **processing data, interacting with databases, and generating responses**.

17.2.2 HTTP Methods Support:

- Flask supports HTTP methods like **GET**, **POST**, **PUT**, **DELETE**, etc.
- You can specify the allowed methods for each route, allowing clients to interact with your API using standard HTTP methods.

17.2.3 Request and Response Processing

- **Request Parsing**:
  - Flask provides convenient methods for parsing **request data, including query parameters, form data, JSON payloads, and request headers**.
  - This makes it easy to access and process incoming data within your view functions.
- **Response Generation**:
  - Flask offers simple ways to generate HTTP responses.
  - You can return **JSON data, HTML content, or custom response objects** from your view functions, allowing you to tailor the response to the client's needs.

[ ]:

```
# hello_world.py

from flask import Flask

#  Create new Flask application instance.
app = Flask(__name__)

print(__name__)


#Creating route connecting URL and function
# @app.route: This is a decorator in Flask that specifies the route or
# URL that should trigger a specific function when a request is made
```

```python
@app.route('/', methods = ['GET'])
def hello_world():
    return "Hello world"


#  Starting flask application development server with specific configurations

app.run(debug=True, port = 5015)
```

18 Example: Create a Simple API with Flask that Interacts with a SQLite Database

This example demonstrates how to create a simple API using Flask that allows users to interact with a SQLite database to manage user information. The API supports basic CRUD (Create, Read, Update, Delete) operations.

18.1 Step-by-Step Guide

18.1.1 1. Install Required Packages

Make sure you have Flask and SQLite installed. You can install Flask using pip:

“‘bash pip install Flask

[ ]:

```python
# import modules
import sqlite3
from flask import Flask, jsonify, request, render_template

#  Create new Flask application instance.
app = Flask(__name__)


# @app.route('/employees', methods = ["GET"])
# def get_data():
#     conn = sqlite3.connect('employee.db')
#     cursor = conn.cursor()

#     cursor.execute("SELECT * FROM employees")
#     rows = cursor.fetchall()
#     cursor.close()
#     conn.close()


#     return rows

@app.route('/employees', methods = ["GET"])
def get_data():
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM employees")
    rows = cursor.fetchall()
    cursor.close()
    conn.close()

    user_list = [{"id": user[0],"name":user[1], "dep":user[2], "sal":user[3] }for user in rows]

    return user_list

@app.route('/employees/<int:id>', methods = ["GET"])
def get_data_id(id):
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM employees WHERE id = ?", (id,))
    row = cursor.fetchone()
    cursor.close()
    conn.close()

    if row == None:
        return "User not found!"
    else:
        return {"id":row[0], "name":row[1], "dep":row[2], "sal":row[3]}

@app.route('/employees/new', methods = ["POST"])
def put_data():
    new_user = request.json
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute('INSERT INTO employees(name, department, salary) VALUES (?, ?, ?)', (new_user["name"], new_user["department"], new_user["salary"]))
    conn.commit()
    cursor.close()
    conn.close()

    return "New user created"


if __name__ == '__main__':
    app.run(debug = True, port = 5015)
```

18.1.2 3. Testing the API

You can use tools like **cURL** or **chrome** to interact with the API.

```
-**cURL** is a command-line tool used for transferring data to and from servers using various p
```

Create a New User

```
curl -X POST http://127.0.0.1:5015/employees/new -H "Content-Type: application/json" -d '{"name
```

Get All Users

```
curl -X GET http://127.0.0.1:5015/employees
```

Update a User

```
curl -X PUT http://127.0.0.1:5000/employees/1 -H "Content-Type: application/json" -d '{"name":
```

Delete a User

```
curl -X DELETE http://127.0.0.1:5000/employees/1
```

18.2 Conclusion

This example illustrates how to create a simple API with Flask that interacts with a SQLite database, enabling basic management of user data. You can extend this API further by adding error handling, authentication, or additional features as needed. “‘

- requests :
- APIs often utilize HTTP messages for communication between various software components. This approach allows for interoperability between different systems and technologies, enabling them to exchange data and trigger actions in a standardized and widely supported manner over the web.
- This typically involves using HTTP methods such as GET, POST, PUT, DELETE, etc., along with request and response headers and bodies to exchange data and perform actions over the web.
- The requests package in Python is a powerful library designed to simplify the process of making HTTP requests and working with APIs. It provides a high-level interface for interacting with web services and handling HTTP communication efficiently.