Full-Stack Development Assignment: Build a Simple SQL Runner

Introduction:

This assignment aims to evaluate your ability to design, develop, and deploy a web application using modern JavaScript frameworks (Next.js/React) for the frontend and Python or similar language for the backend. The task is to build a SQL Runner, allowing users to execute SQL queries and view the results.

Example: www.programiz.com/sql/online-compiler

Requirements:

The application should consist of the following key features:

1. Frontend (Next.js/React):

- **Query Input Area:**
  - A prominent text area where users can type or paste SQL queries.
  - A "Run Query" button to submit the query to the backend.
- **Results Display Area:**
  - A clear and well-formatted display of the data returned by the executed SQL query. This should handle tabular data effectively, displaying column headers and rows.
  - Appropriate visual feedback for loading states and potential errors.
- **Available Tables Panel:**
  - A sidebar or dedicated section displaying a list of available database tables.
  - Upon clicking on a table name, a preview of the table's schema (column names and their data types) and a few sample rows (e.g., the first 5 rows) should be displayed. This helps users understand the structure and data within each table.

2. Backend (Python - Any Framework):

- **API Endpoint for Query Execution:**
  - A RESTful API endpoint that accepts an SQL query as input (likely via a POST request).
  - This endpoint should connect to a pre-configured sample database (we will provide connection details or instructions for setting one up).
  - The backend should execute the received SQL query against the database.

    - The API should return the query results in a structured format (e.g., a JSON array of objects, where each object represents a row and keys are column names).
    - Appropriate error handling should be implemented to catch invalid queries or database errors and return informative error messages to the frontend.
- **API Endpoint for Table Information:**
    - A RESTful API endpoint to fetch the list of available tables in the database.
    - Another endpoint to fetch the schema (column names and data types) and a limited number of sample rows for a specific table.

Technical Specifications:

- **Frontend:** Must be built using Next.js or React.js.
- **Backend:** Must be built using Python. You are free to choose any Python web framework (e.g., Flask, Django, FastAPI).
- **Data Format:** Data exchanged between the frontend and backend should primarily be in JSON format.
- **Database:** For the purpose of this assignment, we will provide instructions on how to set up a simple SQLite database with sample data (similar to the "Customers," "Orders," and "Shippings" tables in the screenshot). Your application should be configured to connect to this database.

Setting Up the Database:

Alright, here are the instructions on how to set up a simple SQLite database with sample data that your Python backend can communicate with.

Setting up a SQLite Database for the SQL Runner Assignment

For this assignment, using SQLite is a straightforward way to have a file-based database that doesn't require a separate server. Here's how to set it up:

1. Install SQLite (if not already installed):

Most operating systems come with SQLite already installed. You can check if it's installed by opening your terminal or command prompt and running:

Bash

```
None


sqlite3 --version
```

If you don't see a version number, you can install it using your system's package manager:

macOS: If not already present, you can install it via Homebrew:

```
None
brew install sqlite3
```

2. Create the Database File:

Open your terminal or command prompt and navigate to the directory where you plan to keep your backend code (or a dedicated `database` folder within it). Then, run the following command:

Bash

```
None


sqlite3 sql_runner.db
```

This command will either open the `sql_runner.db` file if it exists, or create a new empty database file with that name and open the SQLite command-line interface.

3. Create the Sample Tables and Insert Data:

Now, within the SQLite command-line interface (the `sqlite>` prompt), execute the following SQL commands to create the `Customers`, `Orders`, and `Shippings` tables and insert some sample data:

SQL

```
None

-- Create the Customers table

CREATE TABLE Customers (

    customer_id INTEGER PRIMARY KEY AUTOINCREMENT,

    first_name VARCHAR(100),

    last_name VARCHAR(100),

    age INTEGER,

    country VARCHAR(100)

);


-- Insert sample data into Customers

INSERT INTO Customers (first_name, last_name, age, country) VALUES

('John', 'Doe', 30, 'USA'),

('Robert', 'Luna', 22, 'USA'),

('David', 'Robinson', 25, 'UK'),

('John', 'Reinhardt', 22, 'UK'),

('Betty', 'Doe', 28, 'UAE');


-- Create the Orders table
```

```
CREATE TABLE Orders (

    order_id INTEGER PRIMARY KEY AUTOINCREMENT,

    item VARCHAR(100),

    amount INTEGER,

    customer_id INTEGER,

    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)

);


-- Insert sample data into Orders

INSERT INTO Orders (item, amount, customer_id) VALUES

('Keyboard', 400, 4),

('Mouse', 300, 4),

('Monitor', 12000, 3),

('Keyboard', 400, 1),

('Mousepad', 250, 2);


-- Create the Shippings table

CREATE TABLE Shippings (

    shipping_id INTEGER PRIMARY KEY AUTOINCREMENT,

    status VARCHAR(100),

    customer INTEGER

);
```

```sql
-- Insert sample data into Shippings

INSERT INTO Shippings (status, customer) VALUES

('Pending', 2),

('Pending', 4),

('Delivered', 3),

('Pending', 5),

('Delivered', 1);


-- Exit the SQLite command-line interface

.exit
```

4. Connecting to the Database from your Python Backend:

In your Python backend code (using your chosen framework like Flask, Django, or FastAPI), you will need to use a Python SQLite library (usually `sqlite3` which is built-in). Here's a basic example of how you might connect:

Python

Python
```python
import sqlite3

DATABASE_URL = 'sql_runner.db'  # Path to your database file

def get_db_connection():

    conn = sqlite3.connect(DATABASE_URL)
    conn.row_factory = sqlite3.Row  # To access columns by name

    return conn



def close_db_connection(conn):

    if conn:

        conn.close()



# Example of executing a query

def execute_query(query):

    conn = get_db_connection()

    cursor = conn.cursor()

    try:

        cursor.execute(query)

        results = cursor.fetchall()

        conn.commit()  # For INSERT, UPDATE, DELETE

        return [dict(row) for row in results] # Convert rows to
dictionaries

    except sqlite3.Error as e:

        return {"error": str(e)}

    finally:

        close_db_connection(conn)
```

```python
# Example of fetching table names

def get_table_names():

    conn = get_db_connection()

    cursor = conn.cursor()

    cursor.execute("SELECT name FROM sqlite_master WHERE
type='table';")

    tables = [row[0] for row in cursor.fetchall()]

    close_db_connection(conn)

    return tables


# Example of fetching schema and sample data for a table

def get_table_info(table_name):

    conn = get_db_connection()

    cursor = conn.cursor()

    try:

        cursor.execute(f"PRAGMA table_info({table_name});")

        columns = [{"name": row[1], "type": row[2]} for row in
cursor.fetchall()]

        cursor.execute(f"SELECT * FROM {table_name} LIMIT 5;")

        sample_data = [dict(row) for row in cursor.fetchall()]

        return {"columns": columns, "sample_data": sample_data}

    except sqlite3.Error as e:

        return {"error": str(e)}
    finally:

        close_db_connection(conn)


# Example usage (you'll integrate this into your API endpoints)

if __name__ == '__main__':

    print("Available tables:", get_table_names())

    print("\nCustomers table info:", get_table_info("Customers"))

    results = execute_query("SELECT first_name, age FROM Customers
WHERE age > 25;")

    print("\nQuery results:", results)
```

Key Points:

- `sql_runner.db`: This is the name of the SQLite database file that will be created in the directory where you run the `sqlite3` command.
- SQL Commands: The `CREATE TABLE` statements define the structure of your tables, and the `INSERT INTO` statements populate them with sample data.
- .exit: This command is used within the SQLite command-line interface to close the connection and exit.
- Python `sqlite3` module: This is the standard library module in Python for interacting with SQLite databases.
- `DATABASE_URL`: Define the path to your database file in your Python code.
- `get_db_connection()` and `close_db_connection()`: These are helper functions to manage the database connection.
- `conn.row_factory = sqlite3.Row`: This line is important as it allows you to access the columns of a fetched row by their name (like a dictionary).
- Error Handling: Include `try...except` blocks to catch potential database errors.
- Fetching Table Information: The `PRAGMA table_info()` command is specific to SQLite and allows you to get details about a table's columns. The `SELECT name FROM sqlite_master WHERE type='table';` query retrieves the list of tables.

By following these steps, you will have a SQLite database with sample data that your Python backend can connect to and interact with for the SQL Runner assignment. Remember to adjust the database file path in your Python code if you place the `sql_runner.db` file in a different location.

Bonus Points:

Candidates can earn bonus points by implementing the following features:

- **Authentication:** Implement a basic user authentication system (e.g., username/password login) to secure the SQL Runner.
- **Recent Run Queries:** Display a history of recently executed queries for the logged-in user. This could be a simple list stored in memory (for this assignment) or a more persistent solution.
- **Dockerization:** Provide a `Dockerfile` for both the frontend and backend applications and a `docker-compose.yml` file to easily set up and run the entire application using Docker.

Evaluation Criteria:

Candidates will be evaluated based on the following criteria:

- **Code Quality:** Clean, well-structured, readable, and maintainable code.
- **Functionality:** Completeness and correctness of the implemented features.
- **User Interface (UI) and User Experience (UX):** Clarity, responsiveness, and ease of use of the application.
- **API Design:** Well-designed and documented API endpoints.
- **Error Handling:** Robust error handling on both the frontend and backend.
- **Bonus Features:** Implementation of the bonus features.
- **Adherence to Requirements:** How well the application meets the specified requirements.

Submission Guidelines:

- Please submit your project as a zip file
- Include clear instructions in a `README` file on how to set up and run your application, including any necessary dependencies and database setup steps.
- If you implement the Dockerization bonus, include instructions on using Docker Compose.

We look forward to reviewing your submissions and seeing your skills in action!