HALİÇ UNIVERSITY

Database SQLite3 & FastAPI

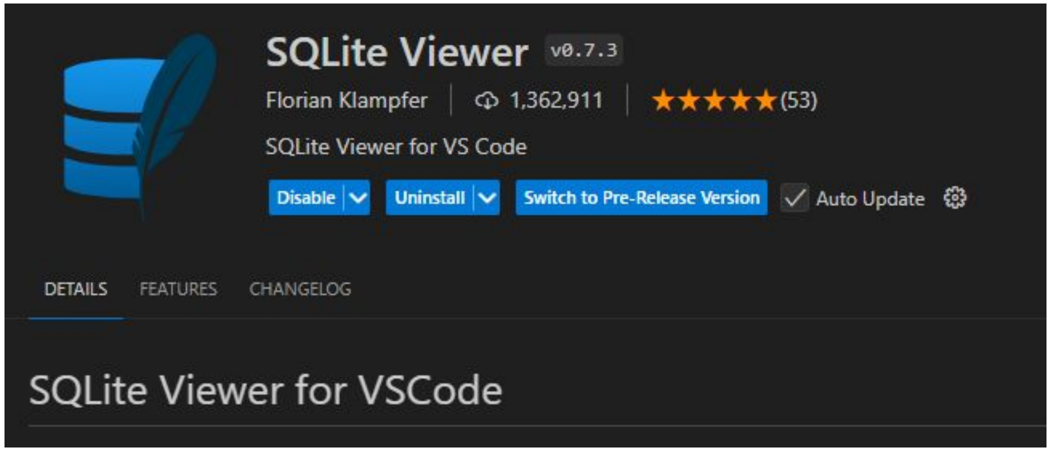
DR. MOHAMMED AL-HUBAISHI

Introduction to SQLite3 and Python

- SQLite3 is a lightweight, disk-based database.
- Python provides built-in support for SQLite3.
- Ideal for small to medium-sized applications.

Setting Up the Environment

- Ensure Python and SQLite3 are installed.
- Install FastAPI using pip: **pip install fastapi**
- Install an ASGI server like uvicorn: **pip install uvicorn**


SQLite Viewer v0.7.3

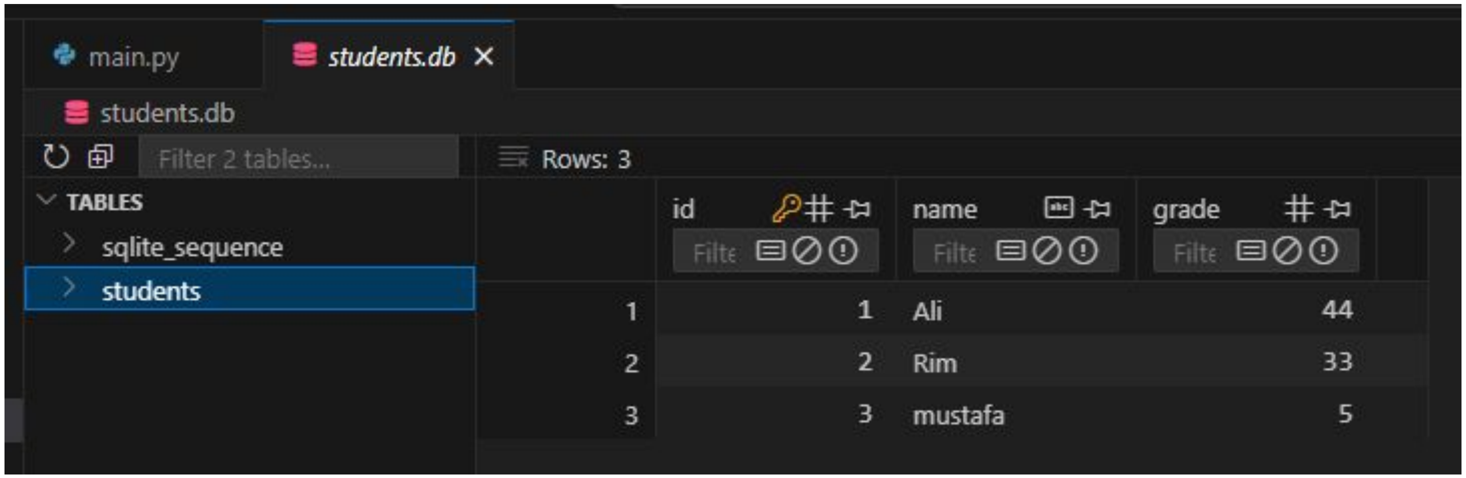
Florian Klampfer | 1,362,911 | ★★★★★(53)

SQLite Viewer for VS Code

SQLite Viewer for VSCode

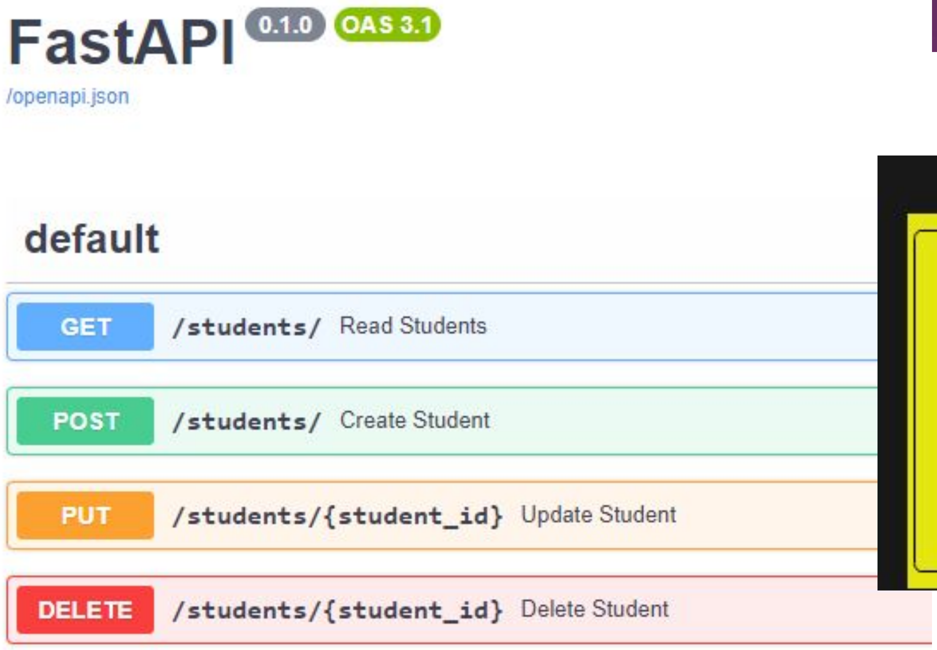
Understanding the Code Structure

- The code is structured into functions and API endpoints.
- Includes database setup, CRUD operations, and error handling.



	id	name	grade
1	1	Ali	44
2	2	Rim	33
3	3	mustafa	5

http://127.0.0.1:8000/docs


FastAPI 0.1.0 OAS 3.1

/openapi.json

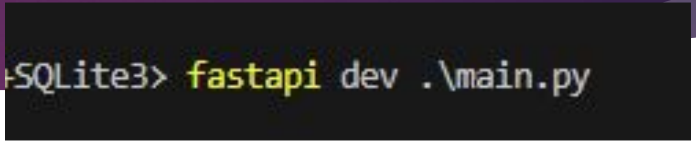
default

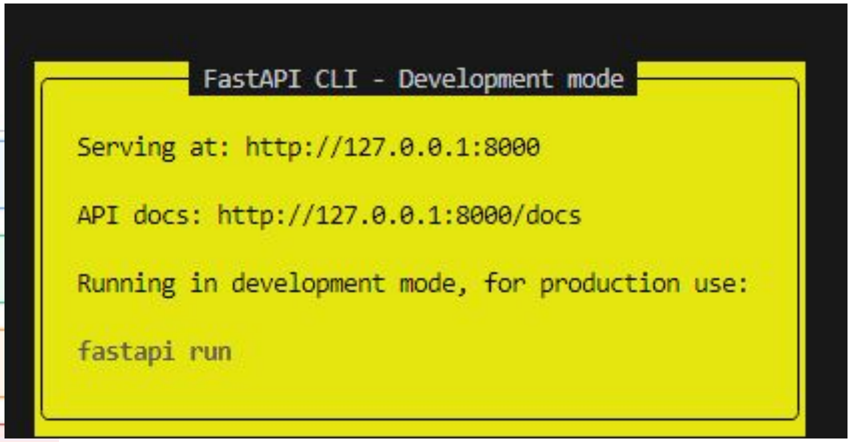
GET /students/ Read Students

POST /students/ Create Student

PUT /students/{student_id} Update Student

DELETE /students/{student_id} Delete Student


```
SQLite3> fastapi dev .\main.py
```


```
FastAPI CLI - Development mode

Serving at: http://127.0.0.1:8000

API docs: http://127.0.0.1:8000/docs

Running in development mode, for production use:

fastapi run
```

https://www.youtube.com/watch?v=gczCPjNyq9U

https://github.com/DrMohammedhbi/DBSqlite3/tree/main

Updated codes files

- SQLite3 is a lightweight, disk-based database.
- Python provides built-in support for SQLite3.
- Ideal for small to medium-sized applications.

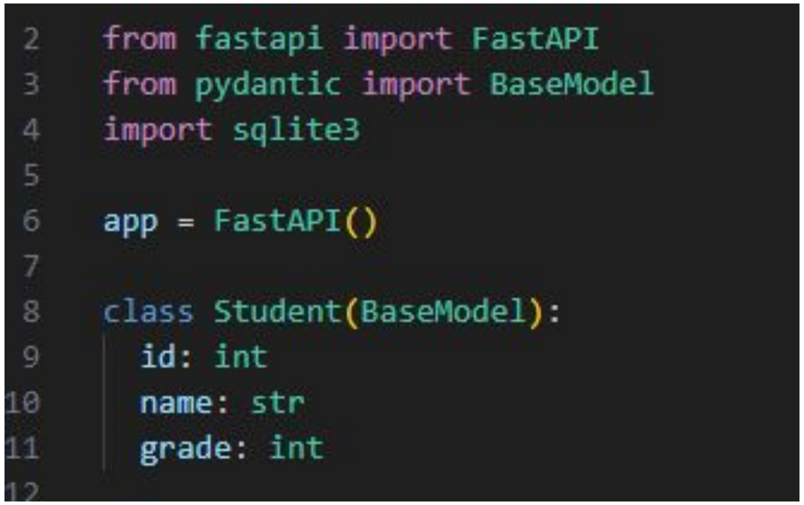
https://youtu.be/mAe2xWO3MZw

https://github.com/DrMohammedhbi/FastapiSQL3Html.git

Defining the Student Model

```
class Student(BaseModel):
    id: int
    name: str
    grade: int
```

- Uses Pydantic for data validation.



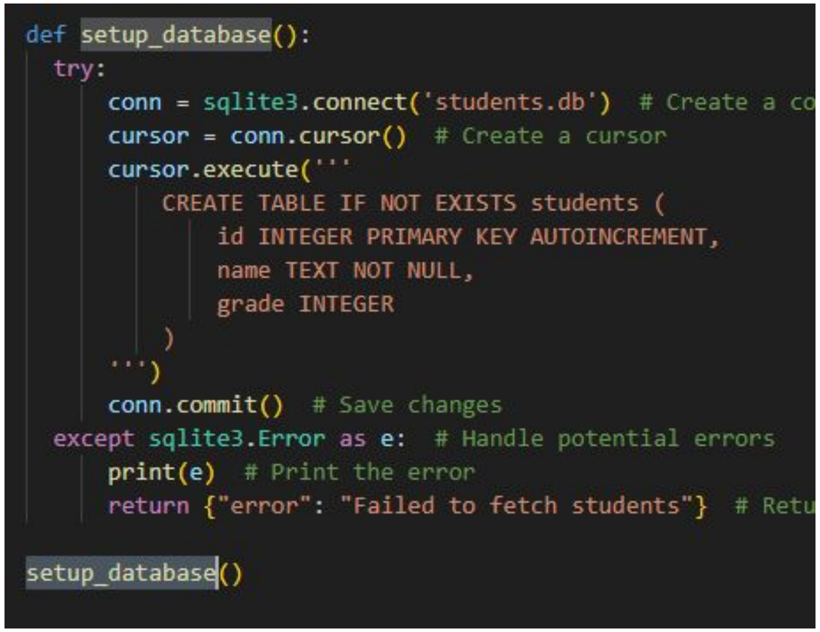
```
from fastapi import FastAPI
from pydantic import BaseModel
import sqlite3

app = FastAPI()

class Student(BaseModel):
    id: int
    name: str
    grade: int
```

Database Connection

conn = sqlite3.connect('students.db')

cursor = conn.cursor()

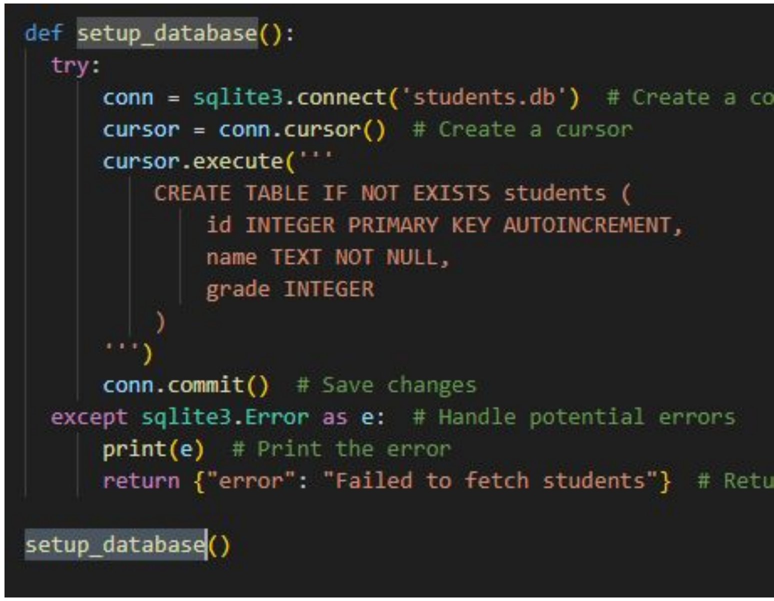
- Establishes a connection to the SQLite3 database.



```
def setup_database():
    try:
        conn = sqlite3.connect('students.db')  # Create a co
        cursor = conn.cursor()  # Create a cursor
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS students (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                grade INTEGER
            )
        ''')
        conn.commit()  # Save changes
    except sqlite3.Error as e:  # Handle potential errors
        print(e)  # Print the error
        return {"error": "Failed to fetch students"}  # Retu

setup_database()
```

Creating the Database Table

```
cursor.execute('''
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    grade INTEGER
)''')
```

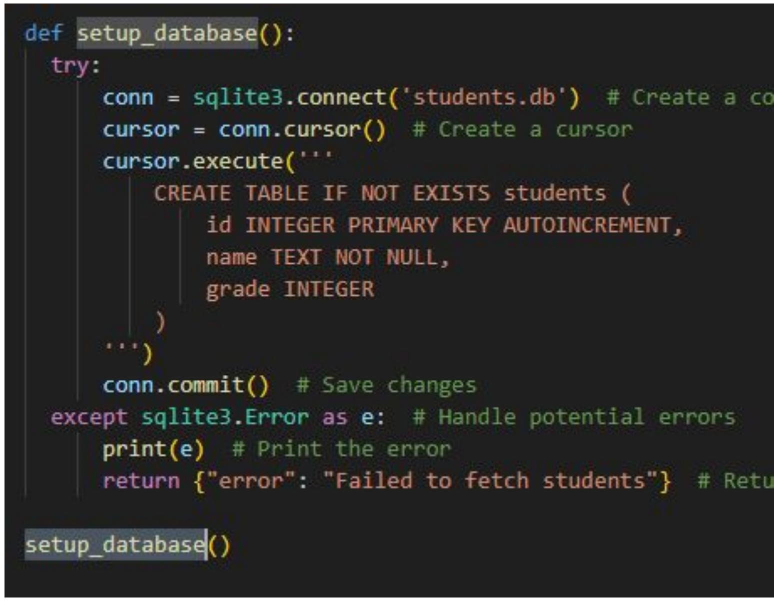


```
def setup_database():
    try:
        conn = sqlite3.connect('students.db')  # Create a co
        cursor = conn.cursor()  # Create a cursor
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS students (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                grade INTEGER
            )
        ''')
        conn.commit()  # Save changes
    except sqlite3.Error as e:  # Handle potential errors
        print(e)  # Print the error
        return {"error": "Failed to fetch students"}  # Retu

setup_database()
```

Error Handling in SQLite3

```
try:
    # Database operations
except sqlite3.Error as e:
    print(e)
```

- Handles potential database errors.

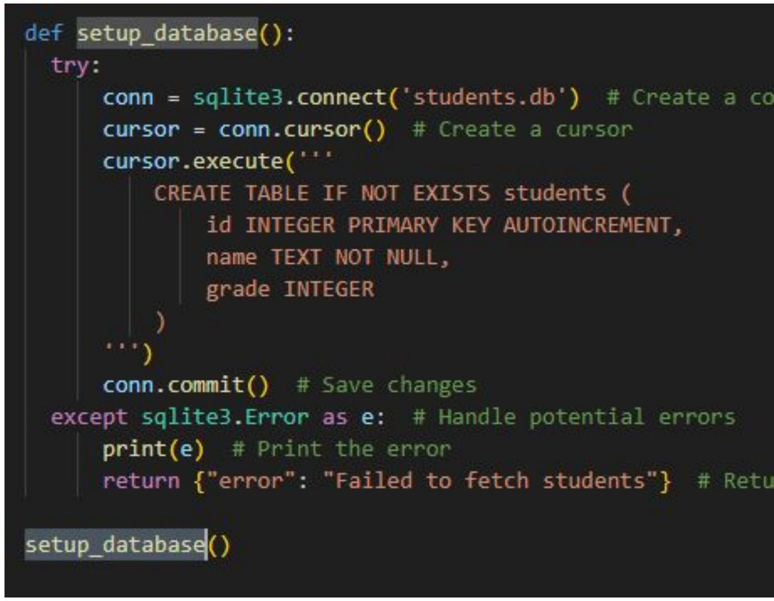


```
def setup_database():
    try:
        conn = sqlite3.connect('students.db')  # Create a co
        cursor = conn.cursor()  # Create a cursor
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS students (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                grade INTEGER
            )
        ''')
        conn.commit()  # Save changes
    except sqlite3.Error as e:  # Handle potential errors
        print(e)  # Print the error
        return {"error": "Failed to fetch students"}  # Retu

setup_database()
```

Setting Up the Database Function

def setup_database():

- # Connect and create table
- Ensures the database and table are ready for use.



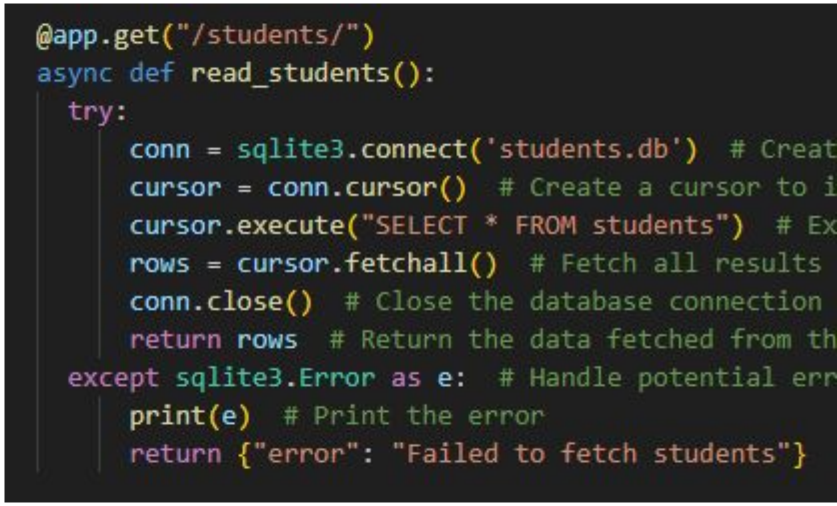
```
def setup_database():
    try:
        conn = sqlite3.connect('students.db')  # Create a co
        cursor = conn.cursor()  # Create a cursor
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS students (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                grade INTEGER
            )
        ''')
        conn.commit()  # Save changes
    except sqlite3.Error as e:  # Handle potential errors
        print(e)  # Print the error
        return {"error": "Failed to fetch students"}  # Retu

setup_database()
```

Reading Data from the Database

cursor.execute('SELECT * FROM students')

rows = cursor.fetchall()

- Fetches all student records from the database.

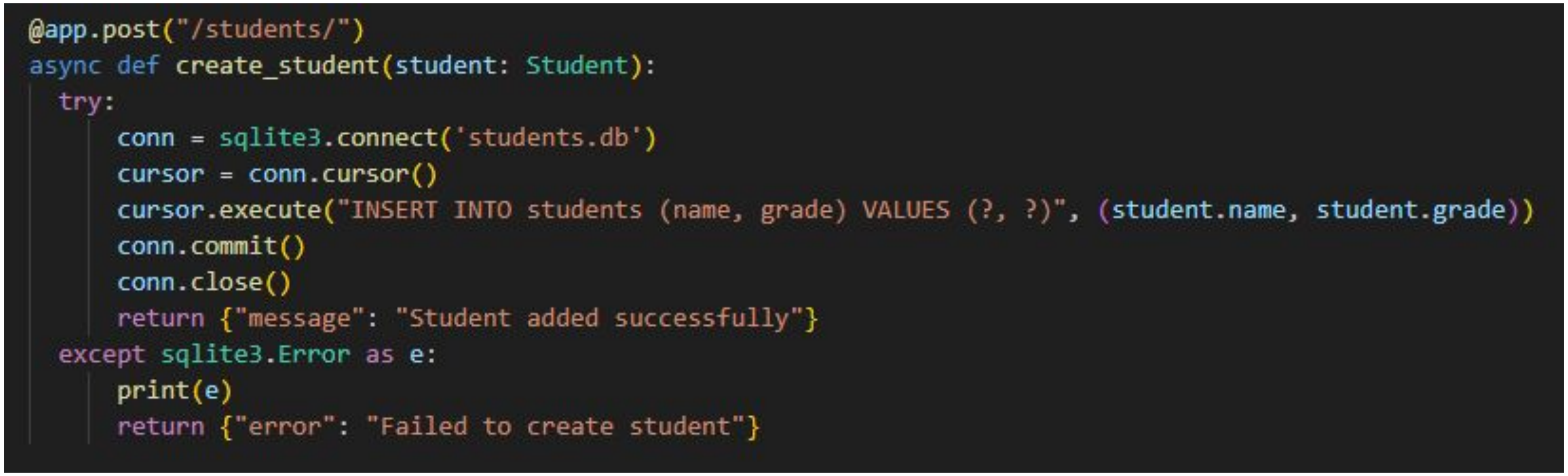


```
@app.get("/students/")
async def read_students():
    try:
        conn = sqlite3.connect('students.db')  # Creat
        cursor = conn.cursor()  # Create a cursor to i
        cursor.execute("SELECT * FROM students")  # Ex
        rows = cursor.fetchall()  # Fetch all results
        conn.close()  # Close the database connection
        return rows  # Return the data fetched from th
    except sqlite3.Error as e:  # Handle potential err
        print(e)  # Print the error
        return {"error": "Failed to fetch students"}
```

Creating a Student Record

cursor.execute('INSERT INTO students (name, grade) VALUES (?, ?)', (student.name, student.grade))

- Adds a new student record to the database.

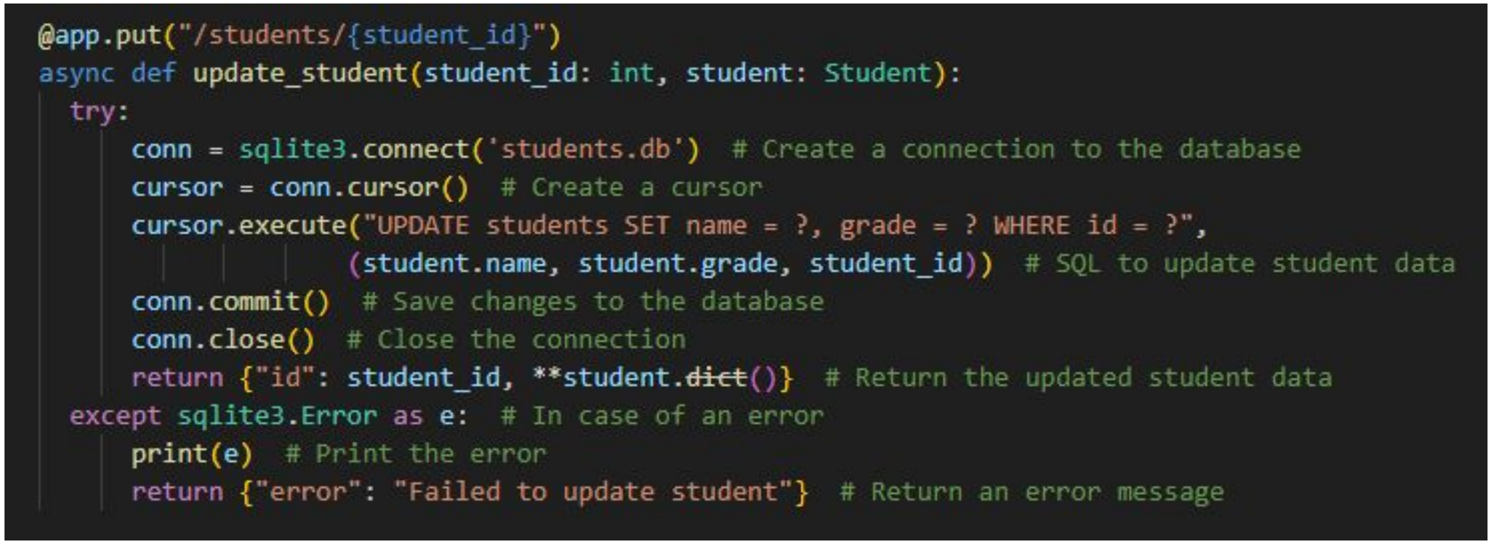


```
@app.post("/students/")
async def create_student(student: Student):
    try:
        conn = sqlite3.connect('students.db')
        cursor = conn.cursor()
        cursor.execute("INSERT INTO students (name, grade) VALUES (?, ?)", (student.name, student.grade))
        conn.commit()
        conn.close()
        return {"message": "Student added successfully"}
    except sqlite3.Error as e:
        print(e)
        return {"error": "Failed to create student"}
```

Updating Student Records

cursor.execute('UPDATE students SET name = ?, grade = ? WHERE id = ?', (student.name, student.grade, student_id))

- Updates existing student records.

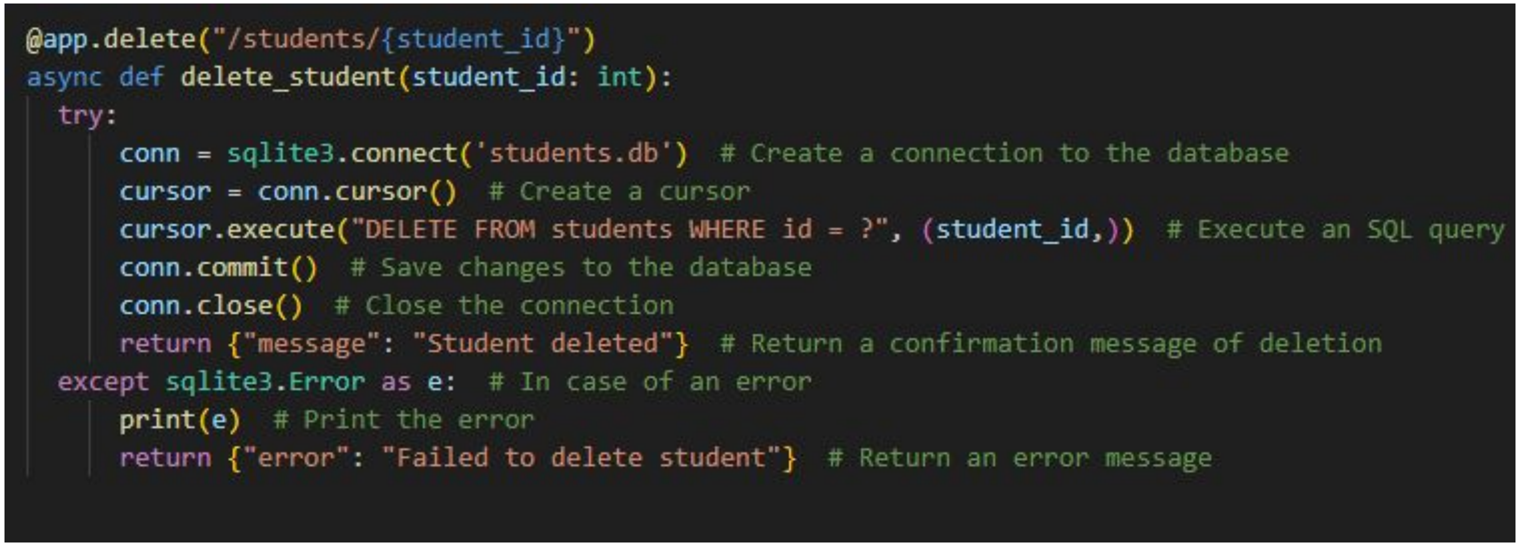


```
@app.put("/students/{student_id}")
async def update_student(student_id: int, student: Student):
    try:
        conn = sqlite3.connect('students.db')  # Create a connection to the database
        cursor = conn.cursor()  # Create a cursor
        cursor.execute("UPDATE students SET name = ?, grade = ? WHERE id = ?",
                       (student.name, student.grade, student_id))  # SQL to update student data
        conn.commit()  # Save changes to the database
        conn.close()  # Close the connection
        return {"id": student_id, **student.~~dict~~()}  # Return the updated student data
    except sqlite3.Error as e:  # In case of an error
        print(e)  # Print the error
        return {"error": "Failed to update student"}  # Return an error message
```

Deleting Student Records

cursor.execute('DELETE FROM students WHERE id = ?', (student_id,))

- Removes a student record from the database.



```python
@app.delete("/students/{student_id}")
async def delete_student(student_id: int):
    try:
        conn = sqlite3.connect('students.db')  # Create a connection to the database
        cursor = conn.cursor()  # Create a cursor
        cursor.execute("DELETE FROM students WHERE id = ?", (student_id,))  # Execute an SQL query
        conn.commit()  # Save changes to the database
        conn.close()  # Close the connection
        return {"message": "Student deleted"}  # Return a confirmation message of deletion
    except sqlite3.Error as e:  # In case of an error
        print(e)  # Print the error
        return {"error": "Failed to delete student"}  # Return an error message
```

Asynchronous Programming in FastAPI

- FastAPI supports async functions for non-blocking operations.
- Improves performance and scalability.

API Endpoints Overview

- GET /students/ - Fetch all students
- POST /students/ - Add a new student
- PUT /students/{id} - Update a student
- DELETE /students/{id} - Delete a student

GET Endpoint for Students

async def read_students():

- # Fetch and return all students
- Returns a list of all students in JSON format.

POST Endpoint for Creating Students

async def create_student(student: Student):

- # Add a new student to the database
- Validates and adds a new student record.

PUT Endpoint for Updating Students

async def update_student(student_id: int, student: Student):

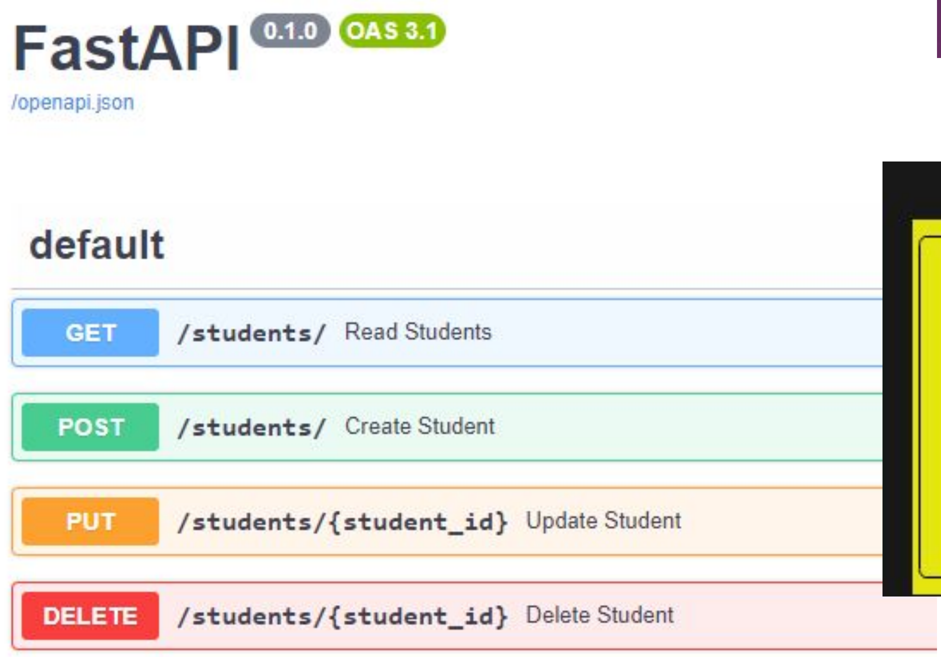
- # Update student information
- Modifies existing student data based on ID.

DELETE Endpoint for Removing Students

async def delete_student(student_id: int):

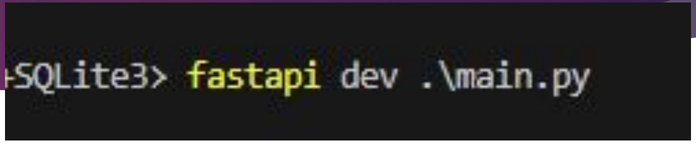
- # Remove a student from the database
- Deletes a student record by ID.

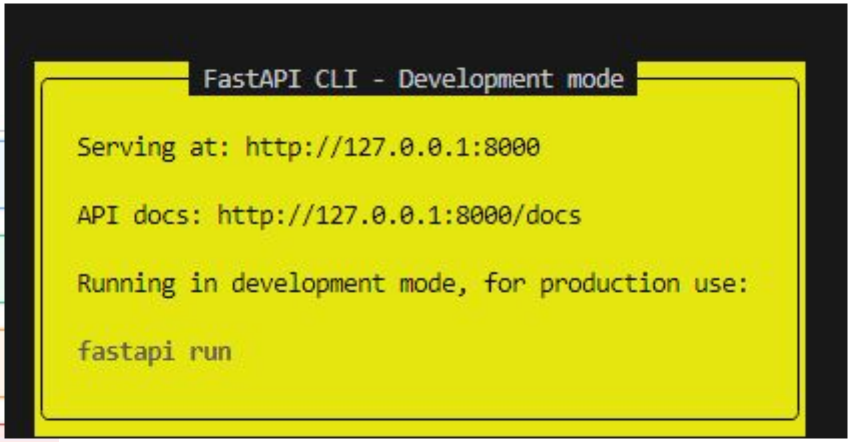
Testing the API
http://127.0.0.1:8000/docs


FastAPI 0.1.0 OAS 3.1
/openapi.json

default

GET	/students/	Read Students
POST	/students/	Create Student
PUT	/students/{student_id}	Update Student
DELETE	/students/{student_id}	Delete Student


```
SQLite3> fastapi dev .\main.py
```


```
FastAPI CLI - Development mode

Serving at: http://127.0.0.1:8000

API docs: http://127.0.0.1:8000/docs

Running in development mode, for production use:

fastapi run
```

https://www.youtube.com/watch?v=gczCPjNyq9U

https://github.com/DrMohammedhbi/DBSqlite3/tree/main

Security Considerations

- Implement authentication and authorization.
- Sanitize inputs to prevent SQL injection.

Performance Optimization

- Use indexes to speed up queries.
- Optimize database schema and queries.

Conclusion and Further Learning

- SQLite3 and FastAPI provide a powerful combination for building APIs.
- Explore more advanced features and best practices.