From Database to API

Python Course · Module 7.5

ElnurBDa

Module 7.5 — Turning Queries into Endpoints

Goals:

- Map your existing **CRUD functions** to HTTP verbs
- Understand how one database query becomes an **API endpoint**
- Prepare for the full To-Do / Student Manager apps in Module 9

Think of this as “wrapping SQL in HTTP”.

Recap — CRUD with SQLite in Python

From Module 7, basic pattern:

```
import sqlite3

def get_conn():
    return sqlite3.connect("app.db")

def add_user(username, score):
    with get_conn() as conn:
        cur = conn.cursor()
        cur.execute(
            "INSERT INTO users (username, score) VALUES (?, ?)",
            (username, score),
        )
```

———————————— [finished] ————————————

We already have **functions** that implement:

- Create (INSERT)
- Read (SELECT)
- Update (UPDATE)
- Delete (DELETE)

Now we'll think about how to **expose** them over HTTP.

CRUD → HTTP Verbs

Typical mapping:

Action	SQL	HTTP verb	Example path
Create	INSERT	POST	/api/users
Read many	SELECT	GET	/api/users
Read one	SELECT	GET	/api/users/<id>
Update	UPDATE	PUT/PATCH	/api/users/<id>
Delete	DELETE	DELETE	/api/users/<id>

Key idea:

- Your existing Python DB functions become the **implementation** behind these endpoints.
- A web framework (like Flask in Module 9) will handle parsing HTTP and calling them.

This module focuses on the **conceptual mapping**.

Designing a Tiny Users API (Concept)

Assume we have these DB helpers:

```
def list_users():
    ...

def get_user(user_id):
    ...

def create_user(username, email):
    ...
```

We could imagine HTTP routes:

- `GET /api/users` → call `list_users()`
- `GET /api/users/<id>` → call `get_user(id)`
- `POST /api/users` → read JSON body, call `create_user(...)`

Later, Module 9 will show actual Flask code that wires this together.

Example — Pseudocode for an Endpoint

Pseudocode, not real Flask code yet:

```
def handle_get_users_request():
    # 1. Read query params (e.g., ?min_score=80)
    # 2. Call list_users(min_score)
    # 3. Convert rows to JSON-serializable dicts
    # 4. Return JSON + status code 200
    ...
```

The main job of an **endpoint**:

1. Validate and parse **HTTP input** (URL, query, body)
2. Call core logic / DB helpers (pure Python)
3. Format a proper **HTTP response** (status code + headers + body)

Your SQL logic from Module 7 lives entirely in step 2.

Mini Design — Tasks Table to Tasks API

Suppose we have a `tasks` table:

```
CREATE TABLE tasks (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    completed INTEGER NOT NULL DEFAULT 0
);
```

DB helpers:

```
def create_task(title):
    ...

def list_tasks(completed=None):
    ...

def mark_task_done(task_id):
    ...
```

Matching endpoints:

- `POST /api/tasks` → `create_task(title)`
- `GET /api/tasks?completed=true` → `list_tasks(completed=True)`
- `PATCH /api/tasks/<id>` or `PUT` → `mark_task_done(id)`

You'll implement these with Flask in Module 9.

Practice — Map Your Queries to Routes

Take your own SQLite mini-project from Module 7:

- Score tracker
- Student list
- Any custom table you built

For each query / function, design:

1. An HTTP method (GET, POST, PATCH, DELETE)
2. A URL path (e.g., /api/students, /api/students/<id>)
3. What parameters come from:
   - path (<id>)
   - query (?course=Python)
   - JSON body ({"name": "...", "email": "..."})

Write this as a small table in your notes.

Thinking About Status Codes

Common choices:

- 200 OK – successful read/update
- 201 Created – new resource created (e.g., new user/task)
- 400 Bad Request – invalid input (fails validation)
- 404 Not Found – resource with that ID does not exist
- 500 Internal Server Error – unexpected server crash

For each planned endpoint, decide:

- What should a **successful** response look like?
- What errors can happen (invalid data, missing row)?
- Which status code makes the most sense in each case?

This will plug directly into Flask route handlers later.

Preview — Flask Style

From Module 9, an actual Flask handler looks like:

```python
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.get("/api/tasks")
def list_tasks_route():
    completed = request.args.get("completed")
    # call list_tasks(completed=...)
    # return jsonify([...]), 200
```

Notice:

- URL and HTTP method are declared in the decorator
- Query params come from `request.args`
- Response is produced with `jsonify` and a status code

Your job now is just to design **what** routes you want and which DB functions they call.

Mini Project – API Blueprint for Your App

Pick one project:

- Score tracker
- Student Manager
- To-Do list with SQLite backend (if you've built it)

Create an **API blueprint** document (markdown / text) that lists:

1. Tables involved and their key columns.
2. All planned endpoints with:
   - HTTP method + path
   - Short description
   - Example request (URL + JSON body if relevant)
   - Example JSON response
3. Which existing Python function(s) each endpoint will call.

You can drop this blueprint straight into Module 9 when you actually build the Flask app.
