Homework 4 — Design Document

API Prototyping with Generative AI
Due: Monday by 9pm ET • 50 points

---

1 — Overview & goals

Create a small, well-documented data pipeline + web API that:

- Converts supplied CSV files into a single SQLite database (`data.db`) using a Python script (`csv_to_sqlite.py`).
- Exposes a `county_data` HTTP POST endpoint that returns rows from the `county_health_rankings` table filtered by ZIP code and measure name.
- Is deployed (Vercel/Render/Netlify/…) and reachable by a URL recorded in `link.txt`.
- Is tested for correctness, robustness (including sanitization), and the special behaviors required by the assignment (HTTP status codes, 418 teapot, etc.).

This doc defines the components, inputs/outputs, data model, API behavior, security mitigations, deployment & testing plan, and the repository layout required for submission.

---

2 — Data sources

Two CSV files (Feb 2025 snapshots):

1. **zip_county.csv** — Zip code → county mapping (from RowZero). Example table name: `zip_county`.
   Example (columns): `zip, default_state, county, county_state, state_abbreviation, county_code, zip_pop, zip_pop_in_county, n_counties, default_city`
2. **county_health_rankings.csv** — County health measures (from County Health Rankings & Roadmaps). Table name: `county_health_rankings`.

Example (columns): `state, county, state_code, county_code, year_span, measure_name, measure_id, numerator, denominator, raw_value, confidence_interval_lower_bound, confidence_interval_upper_bound, data_release_year, fipscode`

Assumption: CSVs have header rows whose column names are valid SQL identifiers (no spaces or need for quoting).

---

3 — Deliverables (files to include in repo)

- `link.txt` — contains only the public API endpoint URL (no query string), e.g. `https://<your-deploy>/county_data`
- `csv_to_sqlite.py` — Python 3 script to convert arbitrary single CSV files into tables in `data.db`. (See spec below.)
- `requirements.txt` — Python dependencies (if any).
- `README.md` — short explanation to run locally and deployment notes.
- `.gitignore` — ignore `__pycache__/`, `data.db`, env files, etc.
- Source for the deployed API (project files, e.g. `api/` or server code) — same repo contents as deployed.

---

4 — Part 1: `csv_to_sqlite.py` spec (15 pts)

Purpose

Given database filename and CSV filename, create (or update) `data.db` and a table named after the CSV filename (basename without extension) and load all rows.

CLI

```
python3 csv_to_sqlite.py &lt;database.db&gt; &lt;input.csv&gt;
```

Behavior

- Creates `database.db` if it does not exist.
- Table name = CSV file basename (e.g. `zip_county.csv` → table `zip_county`).
- Uses CSV header row for column names (assumes header values are valid SQL identifiers). Behavior on "bad" CSV is undefined (assignment note).

Creates table with all columns TEXT (to match the example counts). Example CREATE TABLE statement:

```
CREATE TABLE zip_county (
  zip TEXT,
  default_state TEXT,
  county TEXT,
  ...
);
```

-
- Inserts rows from CSV into the table.
- If the table already exists, behavior can be one of:
    - Drop+recreate table (recommended for reproducible runs), or
    - Replace existing rows (but dropping is simplest to ensure counts match).
- Use parameterized SQL (`?` placeholders) for inserts to avoid SQL injection or malformed values.
- Should be portable across macOS, Linux, Windows (pure Python `sqlite3` + `csv` stdlib).

Implementation notes (suggested)

- Read headers from CSV using Python `csv` module (`csv.DictReader` or `csv.reader`).

- Validate header names are safe SQL identifiers (optional check) — if not safe, either fail or sanitize (but assignment says headers are valid).
- Use `sqlite3` module, `conn.execute()` for DDL, `executemany()` for bulk inserts.
- Exit codes: `0` on success, non-zero on unrecoverable errors.

Example usage (from assignment)

```
rm data.db
python3 csv_to_sqlite.py data.db zip_county.csv
python3 csv_to_sqlite.py data.db county_health_rankings.csv

# quick check:
sqlite3 data.db
sqlite> .schema zip_county
sqlite> select count(*) from zip_county;
```

---

5 — Database model & sample schemas

Table: `zip_county`

- `zip TEXT`
- `default_state TEXT`
- `county TEXT`
- `county_state TEXT`
- `state_abbreviation TEXT`
- `county_code TEXT`
- `zip_pop TEXT`

- zip_pop_in_county TEXT
- n_counties TEXT
- default_city TEXT

Table: `county_health_rankings`

- state TEXT
- county TEXT
- state_code TEXT
- county_code TEXT
- year_span TEXT
- measure_name TEXT
- measure_id TEXT
- numerator TEXT
- denominator TEXT
- raw_value TEXT
- confidence_interval_lower_bound TEXT
- confidence_interval_upper_bound TEXT
- data_release_year TEXT
- fipscode TEXT

(These match example `.schema` in assignment; use exactly these names so graders' checks match.)

---

6 — Part 2: API Prototype — behavior & contract (35 pts)

Endpoint

`POST /county_data`
Content-Type: `application/json`

Required request JSON keys

- `zip` — required, 5-digit string (e.g. `"02138"`)
- `measure_name` — required, one of the accepted measure names (see below)
- Optional: `coffee` — if present and value equals `"teapot"`, return HTTP **418** (see special behavior)
- Any other keys ignored.

Valid `measure_name` values (exact strings)

- Violent crime rate
- Unemployment
- Children in poverty
- Diabetic screening
- Mammography screening
- Preventable hospital stays
- Uninsured
- Sexually transmitted infections

- Physical inactivity
- Adult obesity
- Premature Death
- Daily fine particulate matter

Behavior / Logic

1. **Input validation**
    - If `coffee` key exists and `coffee == "teapot"` → respond HTTP `418` with a short JSON body (e.g. `{"error":"I'm a teapot"}`).
    - If either `zip` or `measure_name` is missing → respond HTTP `400` (Bad Request) with JSON describing the missing keys.
    - If `measure_name` is not one of the accepted strings → respond HTTP `404` (Not Found) — per spec, a zip/measure_name pair not found should yield 404; invalid measure name can be treated as not found.
    - If `zip` is not exactly 5 digits → respond HTTP `400` (Bad Request) (optional input validation but recommended).
2. **Primary query flow**
    - Use `zip_county` to map the supplied `zip` to county(ies). A single ZIP may map to multiple counties; return rows for all matching counties.
        - Query: `SELECT county, county_code, state_abbreviation FROM zip_county WHERE zip = ?`
    - For each county mapping found, query `county_health_rankings` for rows where:
        - `measure_name = ?` AND
        - `county` = `<county name>` AND

        - (optionally) `state = <state_abbreviation>` or `state_code` matches — choose the method that fits the CSV contents.

    - Return the union of all matching `county_health_rankings` rows as JSON array with objects that use the same keys/column names as the `county_health_rankings` table (as in example).

    - If no counties or no ranking rows found for the pair → respond HTTP `404`.

3. **SQL safety**

    - All SQL statements MUST be parameterized (use `?` placeholders). Never compose SQL using string concatenation with user input.

4. **Response format**

    - `200 OK` with JSON array of objects, each object containing fields exactly matching the column names from `county_health_rankings`.

Example (simplified object):


```
{
  "confidence_interval_lower_bound": "0.22",
  "confidence_interval_upper_bound": "0.24",
  "county": "Middlesex County",
  "county_code": "17",
  "data_release_year": "2012",
  "denominator": "263078",
  "fipscode": "25017",
  "measure_id": "11",
  "measure_name": "Adult obesity",
  "numerator": "60771.02",
  "raw_value": "0.23",
  "state": "MA",
  "state_code": "25",
  "year_span": "2009"
}
```


    -

5. **Errors**
    - `400` — missing required keys or malformed `zip`.
    - `404` — requested resource not found (no zip→county mapping, or the county has no rows for the measure, or `measure_name` not in list).
    - `418` — `coffee=teapot` present.
    - For any internal server error → `500` with generic message (avoid leaking internal details).

Example requests

Valid request

```
curl -X POST -H "Content-Type: application/json" \
  -d '{"zip":"02138","measure_name":"Adult obesity"}' \
  https://<your-api>/county_data
```

Teapot request

```
curl -X POST -H "Content-Type: application/json" \
  -d '{"zip":"02138","measure_name":"Adult
obesity","coffee":"teapot"}' \
  https://<your-api>/county_data
# => HTTP 418
```

---

7 — Implementation notes & recommended stack

- **Language:** Python 3.x (or Node/Express if preferred — but Python is easiest given the sqlite3 requirement).
- **Web framework:** Flask / FastAPI / Starlette. (FastAPI gives built-in JSON validation & docs.)

- **Database:** SQLite3 (single file `data.db` placed in project root; ensure the deployed environment can read it).

- **Deployment platforms:** Vercel (serverless functions), Render, or similar. If using Vercel, put API in `api/` (serverless) and ensure `data.db` is present in the build and accessible by the serverless function (serverless sometimes has ephemeral filesystem — prefer Render for persistent file access). Alternatively, produce `data.db` during build step from CSVs (recommended).

Dependency management: `requirements.txt` with:

```
fastapi
uvicorn
sqlite3   # builtin; not needed
```

- (If using Flask, list `flask`.)

Important deployment tip: Many serverless platforms have ephemeral filesystems — ensure the SQLite DB is created/staged at build/deploy time (e.g., run `csv_to_sqlite.py` as a build command) and the resulting `data.db` is included in the deployed package.

---

8 — Security & sanitization

- **SQL injection mitigation:** Use parameterized queries (`?` placeholders) always. Do not format SQL strings using user input.

- **Limited DB permissions:** Read-only operations for API; however grading may attempt injection. Use parameterization and avoid executing arbitrary SQL from user inputs.

- **Input validation:** Ensure `zip` looks like 5 digits; ensure `measure_name` equals one of the accepted strings.

- **Rate limiting / DoS:** Not required for the assignment, but note that public endpoints could be abused. Keep logs to debug if tests fail.

- **Penetration testing consent:** Assignment allows graders to attempt SQL injection; make sure your implementation is robust.

---

9 — Testing plan (self-check / grader expectations)

Unit / integration tests to run locally

1. **CSV → DB test**
    - Remove existing `data.db`.

Run:

```
python3 csv_to_sqlite.py data.db zip_county.csv
python3 csv_to_sqlite.py data.db county_health_rankings.csv
```

    -

Verify using `sqlite3`:

```
.schema zip_county
select count(*) from zip_county;  -- expected 54553 (example)
.schema county_health_rankings
select count(*) from county_health_rankings;  -- expected ~303864
```

    -

2. **API tests**

Missing keys:

```
curl -X POST -H 'Content-Type: application/json' -d '{}'
https://<api>/county_data
# expect 400
```

    -

Teapot:

```
curl -X POST -H 'Content-Type: application/json' -d
'{"zip":"02138","measure_name":"Adult obesity","coffee":"teapot"}'
https://<api>/county_data
```

```
# expect 418
```

Valid query returns array with >0 objects for known zip/measure pairs:

```
curl -X POST -H 'Content-Type: application/json' -d
'{"zip":"02138","measure_name":"Adult obesity"}'
https://<api>/county_data
# expect 200 and JSON array
```

- Not found -> 404:
  - Valid zip but measure matched to nothing (or invalid measure string).
- SQL injection tests (simulated):
  - Try `zip: "02138'; DROP TABLE county_health_rankings; --"` as input. API must not execute malicious SQL; table remains intact.

3. **Edge cases**
   - ZIPs that map to multiple counties — ensure results include all matches.
   - Multiple `county_health_rankings` rows per county for different years — return them all.

Automated grading compatibility

- The grader will run `csv_to_sqlite.py` on provided CSVs and other CSVs — ensure the script works for arbitrary valid CSVs (table name derived from filename).
- Grader will hit `link.txt` endpoint and run a series of POSTs. Ensure endpoint is public and stable.

---

10 — Deployment checklist & tips

- Include `data.db` in deployed artifact or include a build step that runs `csv_to_sqlite.py` during deployment to build `data.db` from the CSVs checked into the repo.
- Confirm `link.txt` contains only the root endpoint URL (no query string).
- If using Vercel serverless functions: ensure file access works; otherwise use Render or a small VPS (Render is simplest for SQLite).
- Confirm CORS if accessing from browser tools (not required for grader).
- Confirm endpoint accepts POST with `application/json`. (Server frameworks sometimes default to form data or require JSON parsing.)

---

11 — README.md (short suggested contents)

- What the repo contains.

How to create the DB locally:

```
 python3 csv_to_sqlite.py data.db zip_county.csv
python3 csv_to_sqlite.py data.db county_health_rankings.csv
```

-

How to run API locally (example using FastAPI + uvicorn):

```
 pip install -r requirements.txt
uvicorn api.main:app --reload
```

-
- Example curl requests.
- Notes about `coffee=teapot` behavior and expected HTTP errors.
- How to redeploy (if applicable).

---

12 — Submission checklist (what I will verify before pushing)

- `csv_to_sqlite.py` works on arbitrary CSVs (tested with provided CSVs).
- `data.db` created with correct tables after running script locally.
- API implements `/county_data` with correct behavior and error codes.
- All strings in responses follow the schema used by `county_health_rankings`.
- `link.txt` points to deployed endpoint (`https://.../county_data`).
- Repo contains `.gitignore`, `requirements.txt`, `README.md`.
- Code comments include attribution for any external code or GenAI help (the assignment requires documenting where code was obtained).

---

13 — Project timeline (recommended, 1–2 days plan)

- **Day 1 morning:** Implement `csv_to_sqlite.py`, run on CSVs, validate schema & counts.
- **Day 1 afternoon:** Implement API locally (FastAPI), wire DB, implement endpoint logic + error handling. Write tests.
- **Day 2 morning:** Deploy to chosen host (Render recommended for SQLite). Build `data.db` during deploy or include it in repo.
- **Day 2 afternoon:** Run full test-suite against deployed URL, finalize README, fill `link.txt`, push to GitHub in `cs1060f25/<username>-hw4`. Verify grader tests locally one last time.

---

14 — Extra notes & academic integrity

- The assignment allows using generative AI. **You must** indicate in your code files (comments or README) where you got code snippets or algorithmic help from GenAI, StackOverflow, etc.
- Work must be done individually; do not share code with other students.
- The grader may attempt SQL injection; consent to penetration testing is implied by submission.

---

15 — Appendix — Sample pseudo-workflow (FastAPI example outline)

(Not full code — high-level outline to implement)

- `csv_to_sqlite.py`
  - parse args
  - read CSV header, build CREATE TABLE statement with `col TEXT` for each header
  - `DROP TABLE IF EXISTS <table>` then `CREATE TABLE`
  - bulk insert rows using `executemany` with placeholders
- `api/main.py` (FastAPI)
  - start FastAPI app
  - `POST /county_data` handler:
    - parse JSON
    - check `coffee == "teapot"` -> raise HTTPException(status_code=418)
    - validate `zip` and `measure_name` exist

        - query `zip_county` where `zip = ?`
        - for each county found, query `county_health_rankings` WHERE `measure_name = ? AND county = ?` (and optionally `state=?`)
        - aggregate rows and return JSON
        - use `sqlite3.Row` or convert to dicts explicitly
    - Error handling: return appropriate HTTP status codes