

GAC017 Practical Backend, Related Tables and Data Decisions

GAC Computing

Follow one request from browser to database

A teacher asks which clubs have places left. A spreadsheet can answer once. A web app lets a reader ask again with a different filter.

Our example has four students, four classes and five enrolments. All records are invented. It is a learning project, not a school booking service.

The three parts have different jobs:

Part	Job in the example	Runs where?
Browser page	Collect a minimum and show a table	The reader's browser
JavaScript server	Check the request and run the query	The local server
SQLite database	Store related records and calculate course totals	Inside this server process

An **HTTP request** asks for a resource. An **HTTP response** contains a status and content. The browser sends `GET /api/courses?min=2`. The server checks the number, then asks SQLite for course totals. It returns a **JSON object**. The browser reads that object and creates table cells. The database does not send HTML to the browser. The browser does not run our server's SQL.

Start the supplied three-file example with `node server.mjs`. Open the address it prints. Use Node.js 22.13 or newer with the built-in SQLite module available. Keep `server.mjs`, `index.html` and `courses.sql` together in your own working copy. No account or package download is needed. Stop your own server with Ctrl-C.

Get the complete example files from the site's static teaching folder:

- `/static/teaching/gac_computing/database-app/server.mjs`
- `/static/teaching/gac_computing/database-app/courses.sql`
- `/static/teaching/gac_computing/database-app/index.html.txt`
- `/static/teaching/gac_computing/database-app/README.txt`

Save the first two with their shown names. Save `index.html.txt` as `index.html`. The last file has the full run and adaptation instructions. Use these paths after the

site's address. The page file is supplied as text for downloading. It must run through your local example server to reach the matching API.

This example uses an **in-memory database**. Restarting creates the original records again. A file database could keep changes after restart. That needs a different storage choice.

Design relationships before writing queries

Each student has one row in **students**. Each class has one row in **courses**. An enrolment links a student to a class. A student may join several classes. A class may contain several students. This is a **many-to-many relationship**. The third table stores one student–class pair per row.

Student 1 joins courses 10 and 20. Course 10 contains students 1, 2 and 3. The same student name need not be repeated in each enrolment row. To change Mei's name, change one student record. This avoids conflicting copies.

The following two blocks form one complete script. Run them in order in a new empty practice database. Do not run it against an existing project database.

```
PRAGMA foreign_keys = ON;
CREATE TABLE students (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL
);
CREATE TABLE courses (
  id INTEGER PRIMARY KEY,
  title TEXT NOT NULL,
  capacity INTEGER NOT NULL CHECK (capacity >= 0)
);
CREATE TABLE enrolments (
  student_id INTEGER NOT NULL REFERENCES students(id),
  course_id INTEGER NOT NULL REFERENCES courses(id),
  PRIMARY KEY (student_id, course_id)
);
```

The tables now exist. Add the fictional records, then query totals for each class.

```
INSERT INTO students VALUES
  (1, 'Mei'), (2, 'Kai'), (3, 'Lin'), (4, 'Jia');
INSERT INTO courses VALUES
  (10, 'Coding', 3), (20, 'Coding', 2),
  (30, 'Art', 2), (40, 'Music', 3);
INSERT INTO enrolments VALUES
  (1, 10), (2, 10), (3, 10), (1, 20), (4, 30);
SELECT c.id, c.title, c.capacity,
       COUNT(e.student_id) AS enrolled
FROM courses c
LEFT JOIN enrolments e ON c.id = e.course_id
GROUP BY c.id, c.title, c.capacity
ORDER BY enrolled DESC, c.id;
```

A **constraint** rejects data that breaks a rule. **NOT NULL** requires a value. **CHECK** (`capacity >= 0`) rejects negative capacity. The paired primary key rejects a repeated enrolment, such as (1,10) twice. Either ID may appear in many pairs. The pair itself must be unique.

Foreign keys reject missing students or classes when foreign-key checking is enabled. The script enables that checking explicitly. A foreign key does not create the missing row. These rules do not prevent every error. For example, capacity 3 does not itself limit enrolments to 3. A real booking operation would need a capacity check and safe handling of simultaneous bookings.

Count classes without losing empty ones

Courses 10 and 20 both have the title Coding. They are different classes. Group by the course ID as well as its title and capacity. Grouping only by title would merge their enrolments and answer the wrong question.

LEFT JOIN keeps every course, including Music with no enrolments. The unmatched course has an empty enrolment side. **COUNT(e.student_id)** counts matched student IDs and gives zero for Music. **COUNT(*)** counts the joined row, including that unmatched row, and would give Music one.

ID	Course	Capacity	Enrolled	Places left
10	Coding	3	3	0
20	Coding	2	1	1
30	Art	2	1	1
40	Music	3	0	3

The browser calculates places left as capacity minus enrolled. There are five enrolments but only four students. Mei appears in two enrolment rows. Do not label five as the number of unique students.

To keep classes with at least two enrolments, add this line after `GROUP BY`:

```
HAVING COUNT(e.student_id) >= 2
```

This is a query fragment, added to the complete query. It returns course 10 only. `HAVING` checks each group total. `WHERE` checks individual rows before totals are calculated. For example, `WHERE c.id = 20` selects one class before grouping; it does not test its total.

Validate and bind the backend input

The route `/api/courses` accepts a minimum from 0 to 99. The browser number control helps users enter it. Direct requests can skip that control. The server therefore checks the input again.

It rejects negative numbers, decimals, 100, repeated minimum parameters and text. Missing `min` means zero. A successful query with no courses is still a valid request.

Request	Status	Meaning
GET /api/courses?min=2	200	One course found
GET /api/courses?min=4	200	Valid query; empty list
GET /api/courses?min=-1	400	Invalid input
GET /missing	404	Route does not exist
POST /api/courses	405	This read-only route accepts GET

A **prepared statement** keeps the SQL structure separate from a value. The server prepares its total-by-course query with `>= ?`, then calls `courses.all(minimum)`. The bound number fills the value position. It is not joined into the SQL text.

Binding values protects this query from injection through that value. It does not prove that every route or operation is secure. SQL keywords and column names cannot be supplied as ordinary bound values. Keep the query structure fixed or choose it from permitted server-owned choices.

The server sends public course totals only. Student names stay out of this response. Real records would also need access rules and permission to use them. An invented example needs no real student information.

Handle loading, empty results and failures

The browser uses `fetch` to request the data. It must check the response status. A completed network request may still return 400 or 500. The browser's `response.ok` distinguishes successful HTTP responses from those errors.

The page clears old rows before loading. Otherwise a failed request could leave old results looking current. It displays a loading message and disables the load button during the request. It then shows the result, an empty message, or a failure message. The button becomes available again so the reader can retry.

Each displayed value goes into `textContent`, not into HTML built from a data string. A course title becomes text inside a cell. It is not treated as page markup.

The sample uses a request number to ignore an older response after a newer request starts. This protects the page from a late response replacing the newest result. It does not change database records or make bookings safe.

Try minimum 0, 2 and 4 in that order. Expect four courses, one course and no courses. Then stop the server and try loading again. The page should explain the failure and allow a retry. Restart the server and load again. The original fictional dataset should return.

Turn results into a supported academic decision

Begin with a question: which classes currently have spare places? State your **unit of analysis** : one class, identified by course ID. Check missing values, repeated enrolment pairs, valid IDs and non-negative capacities before analysis. Name the data date in a real report, because enrolments can change.

The current answer is Coding 20, Art 30 and Music 40. Music has three spare places; the other two have one each. A teacher could first check whether those places are still available before announcing them.

A bar chart could compare enrolled and capacity for each course ID. Keep the two Coding classes separate and label them with their IDs. Show zero enrolments for Music. Do not hide it because its bar is short.

Explain the **limitation** of this decision. These records describe four invented classes at one time. They do not measure teaching quality, future demand or why students chose a class. More enrolments do not prove that a class caused better learning. Avoid using a descriptive count as evidence for a causal claim.

A short report can use five parts: question, data and checks, method, result, limits and next action. Include the query or name the calculation so another reader can reproduce the result. Keep student and enrolment counts separate.

Practise with changes and explain your answers

1. Sketch the three tables. Which keys link them, and why is the third table needed?
2. Predict minimum 1 and minimum 3 before running either request.
3. Replace `COUNT(e.student_id)` with `COUNT(*)`. Which original result becomes wrong, and why?
4. Group only by title. What happens to the two Coding classes?
5. Add course 50, Drama, with capacity 2 and no enrolments. Predict its total and free places.
6. Add enrolment (2,20). What should minimum 2 now return?
7. Try duplicate pair (1,10) and missing student pair (99,10). Explain each rejection.
8. Why must the server check a minimum that the browser already checks?
9. Explain why an empty array gets 200, while a negative minimum gets 400.
10. Write a two-sentence recommendation and one limitation using the original data.

Explained answers

1. Student ID and course ID link the paired enrolment table to their parent tables. The third table represents many students in many classes without repeating names or course facts.
2. Minimum 1 returns 10, 20 and 30. Minimum 3 returns 10 only. Both filters include equality.
3. Music becomes one instead of zero. `COUNT(*)` counts the preserved unmatched course row.
4. Coding totals combine to four. This describes a title group, not either individual class.
5. Drama has zero enrolments and two places left. A left join keeps it at minimum 0.
6. Coding 20 now has two enrolments. Minimum 2 returns IDs 10 and 20, with totals three and two.
7. The paired primary key rejects the duplicate. The enabled foreign key rejects student 99, who does not exist.
8. A caller can send a request without using the page. Browser checks cannot protect the server by themselves.
9. No matching rows is a successful query. A negative minimum breaks the API's input rule.
10. Check remaining places in Coding 20, Art 30 and Music 40 before offering them. Music has most spare places in this example. Invented totals cannot predict actual student demand.