

Computing III: Build, Query and Explain a Database Web App

GAC017 public-outcome practical teaching; full centre guide governs assessed depth.

A working app must answer a clear question

- Build a database, a backend and a reader interface.
- Explain what the project can and cannot establish.
- Use the current centre brief for assessed work.

Browser, server and database have different jobs

- The browser collects input and displays results.
- The server checks requests and runs database queries.
- The database stores records and calculates results.

Routes and validation belong on the server

- A route chooses the operation for a request path.
- Check input even when a browser control checks it.
- Use honest status codes and fixed query structure.

A third table represents the many-to-many relationship

- Store each student and class once.
- Enrolments link a student ID to a course ID.
- A unique pair prevents the same enrolment appearing twice.

SQL filtering and grouping answer different questions

- JOIN relates records through matching keys.
- WHERE selects rows before grouping.
- HAVING selects groups after totals are calculated.

Bound values and empty results are normal API behaviour

- Prepare SQL separately from its input value.
- Return rows as JSON rather than page markup.
- An empty list can be a successful response.

Data analysis needs a question, checks and limits

- Clean and join data before interpreting it.
- Use descriptive summaries to support a specific decision.
- Counts and correlations do not establish causes.

Trace a minimum-two request through all three parts

- Browser sends GET /api/courses?min=2.
- Server validates, binds and queries the database.
- Browser displays the returned course list.

Run the complete example through its own server

- Keep server.mjs, courses.sql and index.html together.
- Use the printed loopback address.
- Restarting resets the in-memory practice database.

```
node server.mjs  
# If the default port is occupied:  
GAC_DEMO_PORT=0 node server.mjs
```

One enrolment row links two existing records

- Student 1 joins classes 10 and 20.
- Class 10 contains students 1, 2 and 3.
- Store each name and class fact in its parent table.

Create parent tables with simple constraints

- Primary keys identify records.
- NOT NULL requires a stored value.
- CHECK rejects a negative capacity.

```
CREATE TABLE courses (  
  id INTEGER PRIMARY KEY,  
  title TEXT NOT NULL,  
  capacity INTEGER NOT NULL CHECK (capacity >= 0)  
);
```

Enable references and make each enrolment pair unique

- Foreign keys require existing parent records.
- Enable foreign-key checking for this SQLite connection.
- The pair is unique; each individual ID may repeat.

```
PRAGMA foreign_keys = ON;  
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)  
);
```

Capacity data does not enforce a booking limit

- A non-negative capacity is not an enrolment-count check.
- This API exposes reads only.
- Real bookings need additional rules and coordination.

Group by class identity, not a shared title

- Classes 10 and 20 are both called Coding.
- Their enrolment totals are 3 and 1.
- A title-only group would combine them into 4.

Keep the empty class with a left join

- Start with every course.
- Match its enrolment rows where they exist.
- Keep Music even when no enrolment matches.

```
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;
```

COUNT of a matched ID differs from COUNT of every row

- Music has zero matched student IDs.
- The left join still contains its preserved row.
- COUNT(*) would incorrectly report one enrolment.

Use HAVING to apply a threshold to course totals

- Minimum 1 keeps classes 10, 20 and 30.
- Minimum 3 keeps class 10.
- The threshold includes equality.

```
GROUP BY c.id, c.title, c.capacity  
HAVING COUNT(e.student_id) >= ?  
ORDER BY enrolled DESC, c.id;
```

Enrolment totals do not count unique students

- There are five enrolment pairs.
- There are four students.
- Mei belongs to two different classes.

Validate one whole number on the server

- Accept 0 through 99; absent min defaults to 0.
- Reject negatives, decimals, SQL text and repeated values.
- Do not rely on the browser number field.

```
const values = url.searchParams.getAll('min');
const raw = values.length === 0 ? '0' : values[0];
if (values.length > 1 || !/^(0|[1-9][0-9]?)/.test(raw)) {
  json(response, 400, { error: 'Use one whole number, 0 to 99.' });
  return;
}
const minimum = Number(raw);
```

Prepare structure, then bind the accepted value

- The query contains a value placeholder.
- `courses.all(minimum)` supplies the number separately.
- Binding this value does not secure every operation.

```
const courses = database.prepare(`
  SELECT c.id, COUNT(e.student_id) AS enrolled
  FROM courses c LEFT JOIN enrolments e ON c.id = e.course_id
  GROUP BY c.id
  HAVING COUNT(e.student_id) >= ?
`);
const result = courses.all(minimum);
```

Use status codes that explain the request outcome

- 200 may contain a populated or empty list.
- 400 means invalid input; 404 means no route.
- 405 means this route does not accept the method.

Return only the data needed by the reader

- The response contains course totals and capacities.
- It does not return student names.
- Real records would need appropriate access rules.

A successful fetch still needs an HTTP status check

- Read the JSON response.
- Check `response.ok` before using successful data.
- An HTTP error may arrive through a completed request.

```
const response = await fetch(
  `/api/courses?min=${encodeURIComponent(minimum.value)}`
);
const data = await response.json();
if (!response.ok) throw new Error(data.error || 'Request failed.');
```

Insert course values as text inside created cells

- Create each row and cell as a page element.
- Set `textContent` for each returned value.
- Calculate places left from capacity minus enrolled.

```
const row = document.createElement('tr');
for (const value of [course.id, course.title, course.enrolled,
                    course.capacity - course.enrolled]) {
  const cell = document.createElement('td');
  cell.textContent = value;
  row.append(cell);
}
rows.append(row);
```

Clear old data and show loading, empty or failed states

- Clear old rows before a new request.
- Use an explicit message for no matching courses.
- After failure, explain retry and re-enable the button.

Ignore a late result from an older request

- Number each request as it starts.
- Only the latest number may update the result.
- Only its completion re-enables the load button.

```
const requestId = ++latestRequest;  
// After receiving the response:  
if (requestId !== latestRequest) return;  
// In finally:  
if (requestId === latestRequest) load.disabled = false;
```

Check records and name the unit of analysis

- One class ID is one analysis unit.
- Check keys, missing values and duplicate pairs.
- Distinguish students from enrolment records.

Recommend a check supported by the current totals

- Classes 20, 30 and 40 have spare places.
- Music has three; the other two have one each.
- Confirm current availability before offering places.

A chart and report should preserve identity and limits

- Compare enrolments and capacity for each class ID.
- Keep both Coding classes and show Music's zero.
- Report question, checks, method, result and limits.

Predict query and dataset changes before running them

- Explain minimum 1 and 3, then check the results.
- Add empty Drama class 50 and enrolment (2,20).
- Test duplicate and missing-reference rejections.

Check yourself

- Explain identity, joins, counts, constraints and validation.
- Separate empty success from invalid input.
- Give a supported recommendation and an explicit limit.