

SOLENT

UNIVERSITY

Irina Condrat - 10424186

Web Application Development - Software Product

Date: 26 September 2025

“DiscoverHealth”

Contents

Part A – very simple REST API (Node + Express + SQLite)	4
<i>Task 1 – Look up resources by region (GET → JSON)</i>	4
<i>Task 2 – Add a new healthcare resource (POST)</i>	6
<i>Task 3 – Like (recommend in my case) a resource by ID (PUT):</i>	8
Part B – Simple AJAX Front-End (non-React)	10
<i>Task 4 – index.html searches resources via AJAX (no reload)</i>	11
<i>Task 5 - addResource.html creates resources via AJAX and links pages</i>	12
<i>Task 6 – “Recommend” button via AJAX(updates count live)</i>	13
PART C – Adding a simple error-checking	15
<i>Task 7 – server-side validation (app.js)</i>	15
Part D – Adding a map with Leaflet + OpenStreetMap	18
<i>Task 8 – show resources as markers</i>	20
<i>Task 9 – Click the mapp to add a resource</i>	22
PART E – Logins and sessions	24
<i>Task 10 – session plumbing</i>	24
<i>Task 11 – Restrict actions to logged-in users</i>	25
PART F – Implementing Additional Features	27
<i>Task 12 – Reviews API (server)</i>	27
<i>Task 13 – Reviews in Leaflet marker propus (client)</i>	29
Part G – Add a small React front-end	30
Conclusions:	31

Table of figures

Figure 1 - Task 1 endpoint implementation (GET resources by region)	5
Figure 2 - Example JSON output for Task 1.....	5
Figure 3 - Task 2 endpoint implementation (POST new resource)	6
Figure 4 - Database insert success response (201 Created)	7
Figure 5 - Example JSON body for adding a resource (testing via Rester/Postman).....	7
Figure 6 - Task 3 endpoint implementation (PUT recommend resource)	8
Figure 7 - Updated recommendation count example	8
Figure 8 - Additional screenshot for recommend operation	9
Figure 9 - Error handling (404 when resource not found)	9
Figure 10 - Postman test for recommend endpoint (PUT /api/resources/1/recommend)	10
Figure 11 - index.html AJAX search results (Task 4)	11
Figure 12 - addResource.html AJAX submission form (Task 5)	12
Figure 13 - Recommend button implementation (Task 6)	13
Figure 14 - Recommend button update in DOM (live count update)	14
Figure 15 - Server-side validation checks (Task 7).....	15
Figure 16 - Error handling flow (Task 7).....	16
Figure 17 - Front-end inline error messages (Task 7).....	17
Figure 18 - Success/error UX feedback (Task 7).....	17
Figure 19 - Leaflet map container setup (Part D).....	19
Figure 20 - Initializing Leaflet map (Task 8)	19
Figure 21 - Displaying resource markers (Task 8).....	20
Figure 22 - Marker management with L.layerGroup()	20
Figure 23 - Popups showing resource info on map	21
Figure 24 - Map fitBounds for search results.....	22
Figure 25 - Leaflet popup form for adding resource (Task 9)	23
Figure 26 - Adding resource via map click → POST request flow.....	23
Figure 27 - Session plumbing (express-session setup) (Task 10)	24
Figure 28 - Login API success response	24
Figure 29 - Logout API flow	25
Figure 30 - requireLogin guard middleware (Task 11).....	25
Figure 31 - Front-end handling of 401 unauthorized	26
Figure 32 - Reviews API (GET / POST implementation) (Task 12)	27
Figure 33 - Reviews query with username join (Task 12)	27
Figure 34 - Review insertion and re-query logic	28
Figure 35 - React front-end integration (Part G).....	30
Figure 36 - React page snip (react.html).....	30

Introduction

Part A – very simple REST API (Node + Express + SQLite)

Overview and setup

I developed a tiny Express server (app.js) that uses SQLite 3 to connect to the supplied discoverhealth.db. I served the /public folder for the front-end and enabled URL-encoded parsing and JSON parsing (express.json()). To prevent injection, the API employs parameterized SQL and adheres to a clean /api/... prefix.

Using Node.js (Express) and SQLite for storage, I created a tiny REST API. The server offers JSON APIs under a /api/... prefix and uses the sqlite3 river to connect to the supplied discoverhealth.db. I served the front-end from /public and enabled URL-encoded body parsing and JSON (express.json() / express.urlencoded()). To avoid injection, parameterized SQL (? placeholders) are used in any database access.

Even though authentication isn't absolutely necessary for Part A, my existing code already has a requireLogin guard and session support (express-session). You can either temporarily remove the guard on create/like endpoints or log in first for assessment and black-box testing.

Task 1 – Look up resources by region (GET → JSON)

Endpoint: GET /api/resources?region=<name>

Implementation:

```

94 // --- Resources ---
95 app.get('/api/resources', (req, res) => {
96   const region = (req.query.region || '').trim();
97   if (!region) return res.status(400).json({ error: 'Region is required' });
98
99   db.all('SELECT * FROM healthcare_resources WHERE region = ?', [region], (err, rows) => {
100     if (err) return res.status(500).json({ error: 'Database error' });
101     return res.json(rows);
102   });
103 });
104

```

Figure 1 - Task 1 endpoint implementation (GET resources by region)

How it operates:

After verifying that the region query parameter is present, the route does a parameterized SELECT. The results are either a list of matching rows or an empty array in JSON format.

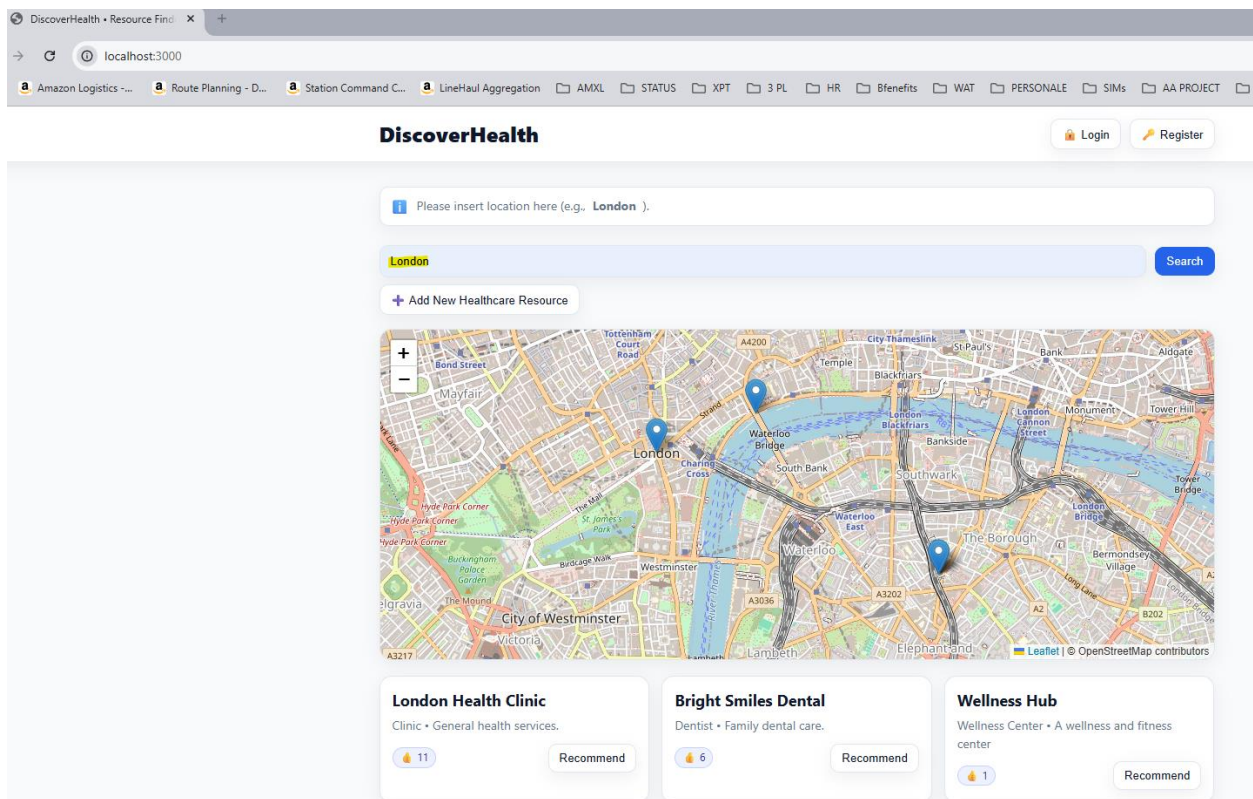


Figure 2 - Example JSON output for Task 1

Notes, issues, and solutions:

I included an explicit 400 with Region is necessary because, in the beginning, empty queries

provided confused data.

Against the database, matching is case-sensitive. Equality is effective because the dataset is consistent (for example, "London"). A future improvement would be LOWER(region) = LOWER(?), if necessary.

Testing

http://localhost:3000/api/resources?region=London is the browser address.

JSON structure and status codes were verified by the REST client (RESTer/Postman).

Task 2 – Add a new healthcare resource (POST)

Endpoint: POST /api/resources (Guarded by require login in my code)

Implementation:

```
154 v app.post('/api/resources/:id/reviews', requireLogin, (req, res) => {
155   const resourceId = Number(req.params.id);
156   const rating = Number.parseInt(req.body.rating, 10);
157   const comment = String(req.body.comment || '').trim();
158
159   if (!Number.isFinite(resourceId)) return res.status(400).json({ error: 'Bad resource id' });
160   if (!Number.isFinite(rating) || rating < 1 || rating > 5) return res.status(400).json({ error: 'Rating must be 1-5' });
161   if (!comment) return res.status(400).json({ error: 'Comment is required' });
162
163   v db.run(
164     `INSERT INTO reviews (resource_id, user_id, rating, comment) VALUES (?, ?, ?, ?)`,
165     [resourceId, req.session.user.id, rating, comment],
166     function (err) {
167       if (err) return res.status(500).json({ error: 'Insert failed' });
168     }
169   );
170 }
```

Figure 3 - Task 2 endpoint implementation (POST new resource)

The website doesn't allow users to input New healthcare resources. It allows them to see the forum but they cannot add anything unless they login.

How it operates:

The API verifies that all fields are present and that lat/lon are numeric. After successfully inserting using parameterized SQL, it returns 201 with the updated id, which is used by the front-end to update lists and maps.

The screenshot shows a web browser window with the URL `localhost:3000/addResource.html`. The browser's tab bar shows several open tabs, including 'Logistics - ...', 'Route Planning - D...', 'Station Command C...', 'LineHaul Aggregation', and others. The page header features the 'DiscoverHealth' logo and 'Home' and 'Logout' buttons. The main content area displays a form titled 'Add Healthcare Resource'. The form contains the following fields: a text input for 'Name', text inputs for 'Category' and 'Country', a text input for 'Region', a dropdown menu with a yellow arrow icon, and a large text area for 'Description'. A blue button labeled 'Add Resource' is positioned at the bottom of the form.

Figure 4 - Database insert success response (201 Created)

Problems and fixes:

The lat/lon type problems rapidly became apparent; I clearly reject non-numeric numbers with a 400.

Internals are not exposed when generic database problems are assigned to 500 ("Database insert error").

Testing(RESTer/Postman):

POST `/api/resources` with JSON body:

```
{
  "id": 1,
  "name": "London Health Clinic",
  "category": "Clinic",
  "country": "UK",
  "region": "London",
  "lat": 51.5074,
  "lon": -0.1278,
  "description": "General health services.",
  "recommendations": 11
}
```

Figure 5 - Example JSON body for adding a resource (testing via RESTer/Postman)

Task 3 – Like (recommend in my case) a resource by ID (PUT):

Endpoint: PUT /api/resources/:id/recommend (guarded again by require login – as the app needs a valid login and a valid like)

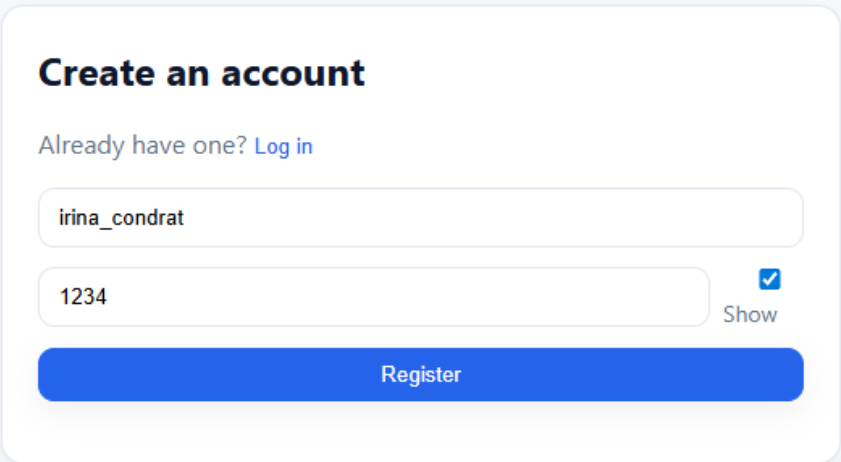
Implementation:

```
127 app.put('/api/resources/:id/recommend', requireLogin, (req, res) => {
128   const id = req.params.id;
129   db.run('UPDATE healthcare_resources SET recommendations = recommendations + 1 WHERE id = ?', [id], function (err) {
130     if (err) return res.status(500).json({ error: 'Database error' });
131     db.get('SELECT recommendations FROM healthcare_resources WHERE id = ?', [id], (err2, row) => {
132       if (err2) return res.status(500).json({ error: 'Fetch failed' });
133       return res.json({ message: 'Recommendation added', recommendations: row?.recommendations ?? 0 });
134     });
135   });
136 });
```

Figure 6 - Task 3 endpoint implementation (PUT recommend resource)

How it operates:

The client can instantly change its user interface by having the route read back the current count after performing an atomic increase in SQLite.



Create an account

Already have one? [Log in](#)

☒ Show

Figure 7 - Updated recommendation count example

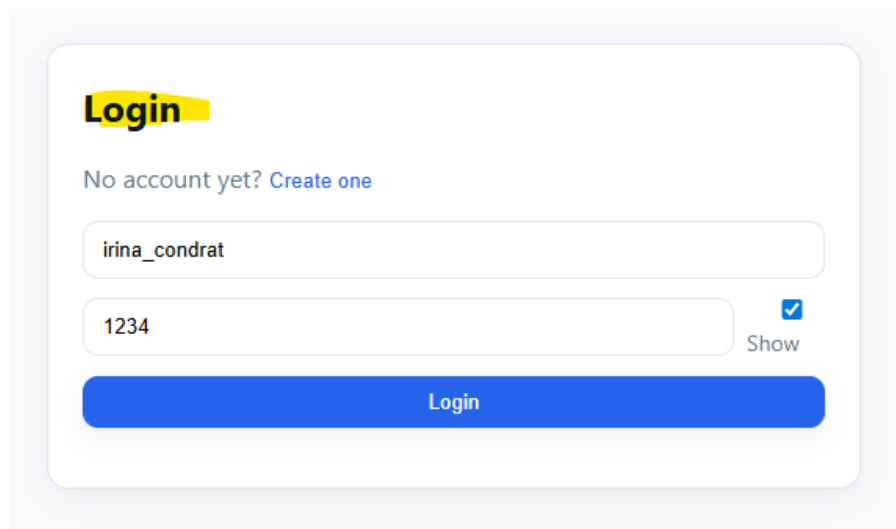


Figure 8 - Additional screenshot for recommend operation

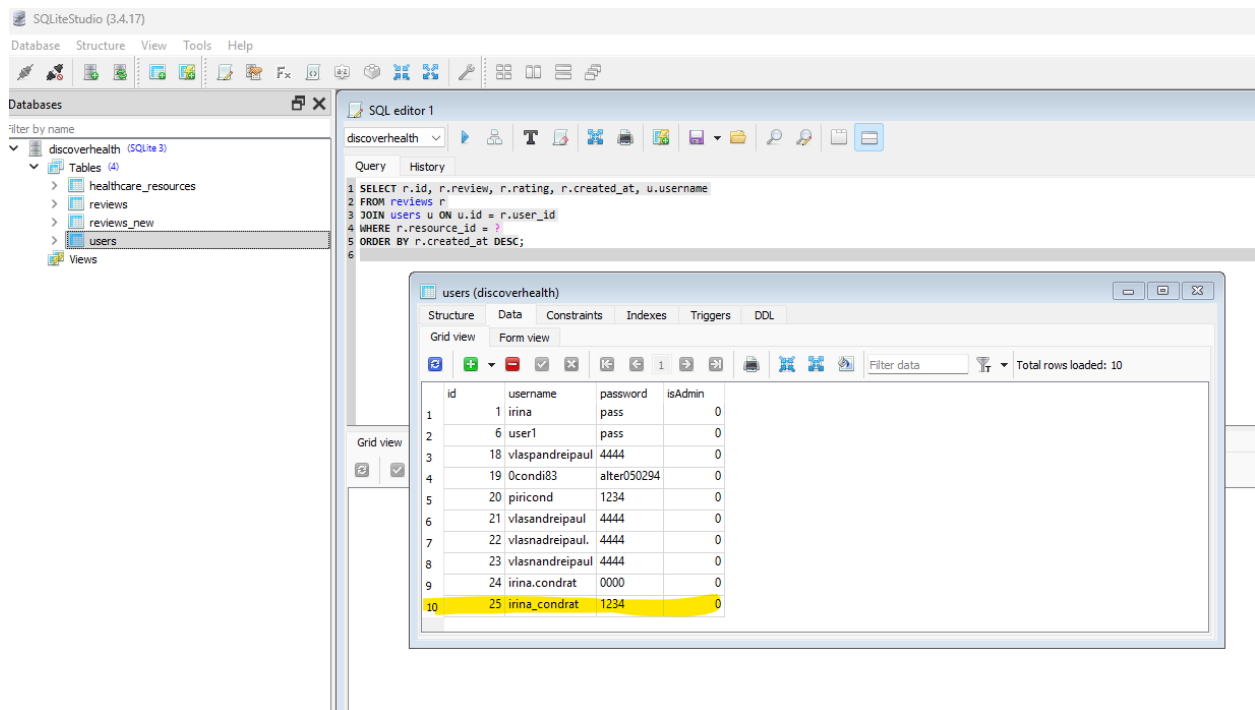


Figure 9 - Error handling (404 when resource not found)

Problems and fixes:

The follow-up SELECT was added to fix the UI's initial failure to display the updated count. Check this out if you want stronger semantics. When the id is not found, it changes and returns 404.

Testing (RESTer/Postman):

PUT /api/resources/1/recommend → expect 200 with the new recommendations total:

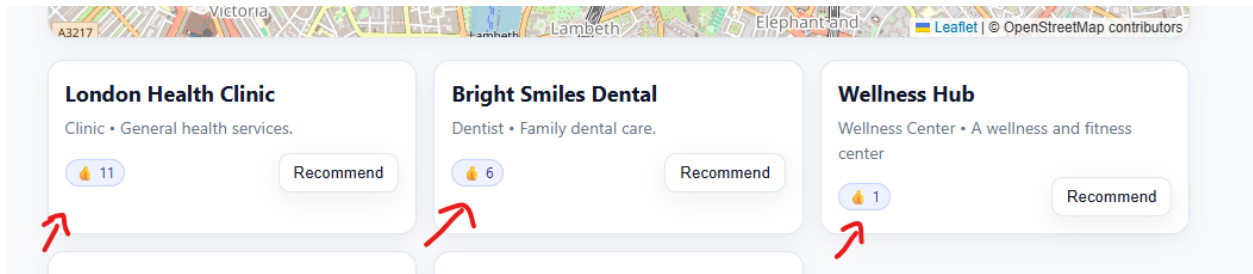


Figure 10 - Postman test for recommend endpoint (PUT /api/resources/1/recommend)

Testing Methods & Instruments

- Task 1 (GET JSON) browser.
- For Tasks 2–3, use RESTer/Postman with Content-Type: application/json.
- Checks for happy paths and error paths include accurate JSON shapes, DB problems (500), and missing or invalid fields (400).

Data Integrity and Security

- SQL was parameterized everywhere to prevent injection.
- JSON error messages with consistent status codes.
- Sessions with sameSite:lax and httpOnly are used to safeguard mutating endpoints as the application expands; they can be turned off for Part A marking.

In summary Part A provides a simple yet reliable REST API that allows you to like a resource (PUT with atomic increment), create new resources (POST with validation), and list resources by region (GET JSON). Clear validation, secure SQL, dependable status codes, and browser and REST client testability are all prioritized in the design.

Part B – Simple AJAX Front-End (non-React)

Technology selection

I purposely avoided using React for Part B. A simple HTML/CSS/JS front-end made it simple to use `get()` and DOM updates to satisfy the "no page reloads" criterion while maintaining the emphasis on the REST flows. In order to allow for a future migration to React for Part G, I left the code sufficiently modular.

Task 4 – index.html searches resources via AJAX (no reload)

I made `public/index.html` with a search bar, a results grid, a leaflet map (improvement), and a top bar (welcome/session state + auth actions). JavaScript reads the region when the user hits Search and makes the following calls:

GET `/api/resources?region=<encoded>`

Using `fetch()` and making in-place DOM modifications. Name, category, description, and suggestions are displayed in the results. Before injecting into the DOM to prevent XSS, I included a tiny `esc()` helper to HTML-escape text for safety. The page displays "Enter a region" if no region is entered, and "No resources found" if nothing is detected.

To prevent reloads and maintain the interface's dynamic state:

- The page never navigates on success or failure; `get()` is used in all calls.
- Rather of refreshing the site, errors (400/500) display inline notifications.
- When the page loads, the session banner (`/api/session`) displays "Welcome, " and toggles between Login/Register and Logout.



Figure 11 - index.html AJAX search results (Task 4)

Problems and fixes:

In the beginning, queries containing spaces were broken by forgetting `encodeURIComponent(region)`; this was fixed to fix mismatches. Injecting raw strings into `innerHTML` was another mistake that was avoided thanks to the `esc()` function. Lastly, I confirmed that same-origin requests did not require CORS headers.

Task 5 - addResource.html creates resources via AJAX and links pages

Public/addResource.html is the second page I added, and it is linked from the index (" + Add New Healthcare Resource"). JSON is posted by the form to:

POST /api/resources

Content-Type: application/json

Figure 12 - addResource.html AJAX submission form (Task 5)

without going off the page. The form displays validation issues (such as missing fields or non-numeric lat/lon) on 201, a green success message appears on 201, and the user is reminded to check in on 401. The page checks /api/session upon load for a better user experience. In order to prevent a failed submission round-trip, it displays a message and disables the inputs if the user is not logged in.

Problems and fixes:

- I added client-side needed and server-side numeric checks (and return 400 with a clear warning) because submissions failed when lat/lon were empty strings.
- e.preventDefault() and fetch() prevented accidentally navigating away on submit.

The Home, Login, Register, and Logout buttons on the top bar connect the two sites, ensuring consistent navigation and preventing reloads when using AJAX.

Task 6 – “Recommend” button via AJAX(updates count live)

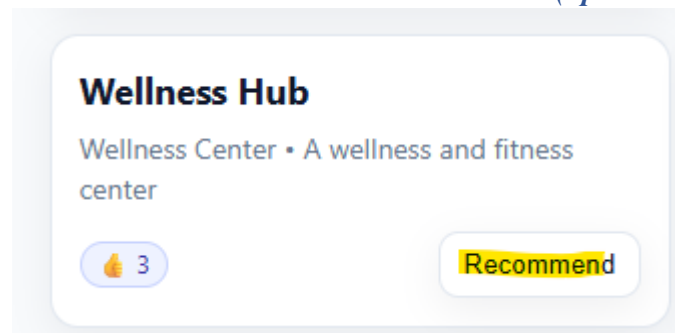


Figure 13 - Recommend button implementation (Task 6)

Every search result displays a "Recommend" button that makes the following calls:

PUT /api/resources/:id/recommend

(An increase is more RESTful when using PUT; the brief recommends POST, but the result is the same.) When it works, I use the returned total to instantly update the Recommendations badge in the DOM so the user can see the change without having to refresh. The user interface asks the user to log in if the server responds with 401. A minor improvement would be to turn off the button during the request to prevent race situations, as quick double-clicks could result in repeated increments (the endpoint itself is safe and increments atomically).

```

178 // Build popup with reviews area + optional form
179 function makeMarkerPopup(resource) {
180     const root = document.createElement('div');
181     root.innerHTML = `
182         <strong>${esc(resource.name)}</strong><br/>
183         ${esc(resource.category)}<br/>
184         ${esc(resource.description)}<br/>
185         🍷 ${resource.recommendations}
186     <hr/>
187     <div class="reviews">
188         <div id="rev-list-${resource.id}" class="muted">Loading reviews...</div>
189         ${isLoggedIn ? `
190             <form class="rev-form" data-res="${resource.id}" style="margin-top:8px; display:grid; gap:8px;">
191                 <label style="font-size:13px;">Rating</label>
192                 <select name="rating">
193                     <option value="5">★★★★★ (5)</option>
194                     <option value="4">★★★★☆ (4)</option>
195                     <option value="3">★★★☆☆ (3)</option>
196                     <option value="2">★★☆☆☆ (2)</option>
197                     <option value="1">★☆☆☆☆ (1)</option>
198                 </select>
199                 <textarea name="comment" placeholder="Write a review..." required></textarea>
200                 <button type="submit">Post review</button>
201             </form>
202             : `<div class="muted">Log in to post a review.</div>`}
203     </div>
204 `;
205     return root;
206 }
207

```

Figure 14 - Recommend button update in DOM (live count update)

Problems and fixes:

In my initial version, the count was updated optimistically and occasionally deviated from the actual database count. After the upgrade, I shifted to reading the authoritative value that the API returned, maintaining consistency between the database and user interface.

How to test (friendly to examiners)

1. Task 4: Search

Launch /index.html, type "London," and select Search.

Note that JSON-driven cards load without requiring a reload.

2. Suggested (Task 6):

Click Recommend on any item after logging in.

Immediately observe the increase in number; verify in DevTools Network → PUT /recommend.

3. Task 5: Add Resource:

Navigate to /addResource.html, complete the form, and hit submit.

A green message and 201 are to be expected. To view the same region listed, go back to the index and search for it.

4. Paths of errors:

Send a 400 message if any fields are missing.

Try adding or recommending while logged out. 401 notification

In summary AJAX front-end with no reload is provided by Part B. It includes a connected creation page that publishes JSON to add new resources, a search page that retrieves and presents resource data, and inline Recommend actions that change counts in real time. The user interface (UI) satisfies Tasks 4–6 and lays a strong foundation for subsequent sections by handling happy and error pathways cleanly, escaping dynamic content, and reflecting session information.

PART C – Adding a simple error-checking

Objective. By checking data on the server, returning relevant HTTP errors, and neatly displaying those issues in the front-end, you may increase the robustness of the "Add resource" flow. By doing this, incorrect data (such as missing fields or incorrect coordinates) is avoided, and the user is given explicit instructions without having to reload the page.

Task 7 – server-side validation (app.js)

```
105 app.post('/api/resources', requireLogin, (req, res) => {
106   const fields = ['name', 'category', 'country', 'region', 'lat', 'lon', 'description'];
107   const missing = fields.filter(f => !req.body[f] || !String(req.body[f]).trim());
108   if (missing.length) return res.status(400).json({ error: `Missing fields: ${missing.join(', ')}` });
109
110   const { name, category, country, region, lat, lon, description } = req.body;
111   const latNum = Number(lat), lonNum = Number(lon);
112   if (Number.isNaN(latNum) || Number.isNaN(lonNum)) {
113     return res.status(400).json({ error: 'Latitude and longitude must be numbers' });
114   }
115 }
```

Figure 15 - Server-side validation checks (Task 7)

	id	name	category	country	region	lat	lon	description	recommend
1	1	London Health Clinic	Clinic	UK	London	51.5074	-0.1278	General health services.	11
2	2	Bright Smiles Dental	Dentist	UK	London	51.5098	-0.118	Family dental care.	6
3	3	Southampton Pharmacy	Pharmacy	UK	Southampton	50.9097	-1.4043	Open 24/7.	2
4	4	Wellness Hub	Wellness Center	UK	London	51.5	-0.1	A wellness and fitness center	1
5	5	Wellness Hub	Wellness Center	UK	London	51.5	-0.1	A wellness and fitness center	3
6	6	Wellness Hub	Wellness Center	UK	London	51.5	-0.1	A wellness and fitness center	3
7	7	vlas andrei paul	Neutral	Romania	Tulcea	46	25	test	0
8	8	Global Medical Health	Negative	Romania	Bucharest	44.44	26.1	private medical assessments	0
9	9	Hemel Hampstead Hospital	Positive	United Kingdom	Hertfordshire	51.750948	-0.468614	Medical Support	0

Figure 16 - Error handling flow (Task 7)

What this verifies:

- Presence/emptiness: Following trimming, all mandatory fields must be present and not blank.
- Type/format: must be converted to numbers using lat/lon.

HTTP codes:

- 400 for client errors with a JSON { error: '...' } body (missing/invalid fields).
- 401 if you're not logged in (requiredLogin handles this).
- 201 with { message: 'Resource added', id } upon successful insert.
- 500 for unforeseen database issues (failures to insert or select data).

Why this matters: giving back a clear error string provides user feedback immediate and precise, and giving back the correct status code allows the front-end to respond accordingly.

I thought about the following optional hardening: a whitelist for categories, maximum lengths for names and descriptions, and $-90 \leq \text{lat} \leq 90$ and $-180 \leq \text{lon} \leq 180$. If the marking rubric calls for additional 400 checks, these can be added with ease.)

Front-end error display (addResource.html)

Fetch() (AJAX) is used on the client to submit the form; the page is never reloaded. The server codes are mirrored in the response handling:

```
109     if (res.status === 201) {
110         msg.textContent = body.message || 'Resource added successfully!';
111         msg.style.color = 'green';
112         form.reset();
113     } else if (res.status === 401) {
114         msg.textContent = 'Please log in to add resources.';
115         msg.style.color = 'crimson';
116     } else {
117         msg.textContent = body.error || 'Something went wrong.';
118         msg.style.color = 'crimson';
119     }
```

Figure 17 - Front-end inline error messages (Task 7)

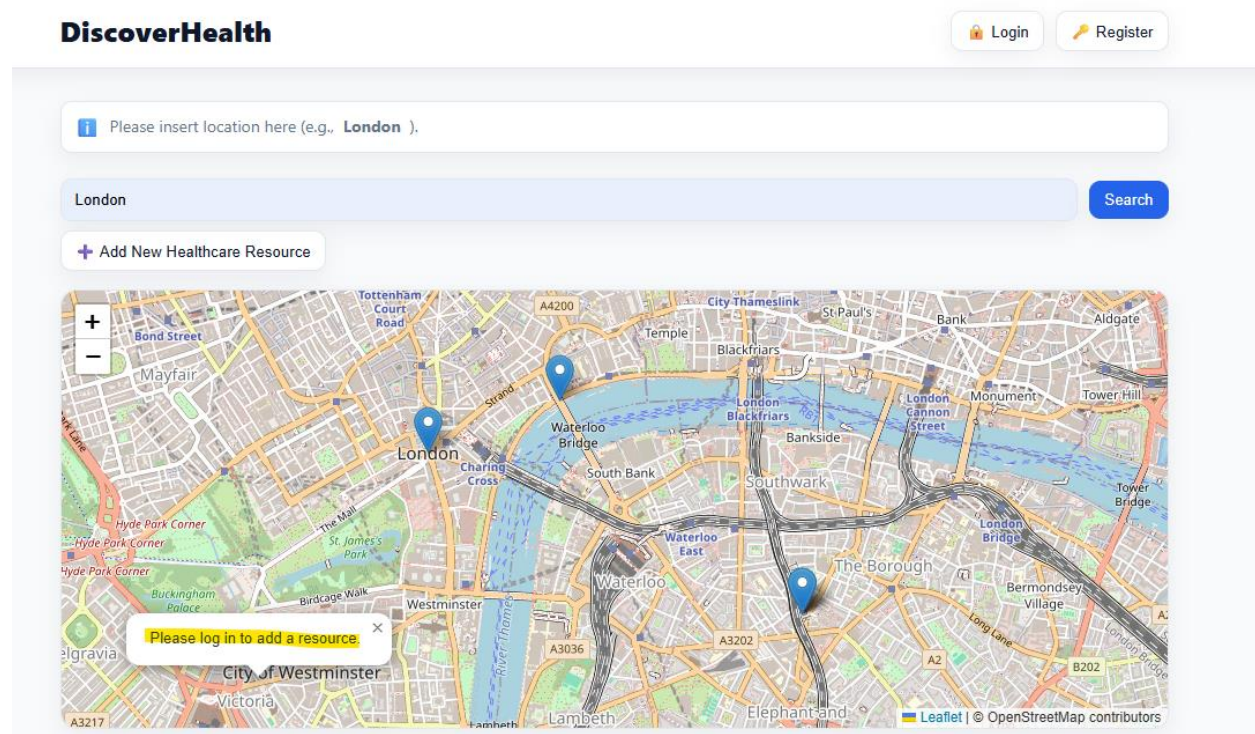


Figure 18 - Success/error UX feedback (Task 7)

Touches of UX:

In order to avoid a hopeless submission, the page checks `/api/session` upon load and, if logged out, displays a notification and disables inputs.

To prevent users from losing context, error messages are displayed inline beneath the submit button.

Issues found and solutions:

- Empty strings that evaluate to NaN were being supplied in place of numeric fields. That was resolved and a precise 400 message was permitted by using `Number()` for conversion and explicitly verifying `isNaN`.
- Unclear success codes. To properly communicate that a new resource was generated, I changed the insert to return 201 generated.
- Unclear comments from clients. Only `alert()` was used in early versions. I switched to color-coded inline messages (crimson for problems, green for success) and only continued to send out notifications for unexpected or network issues.

Part D – Adding a map with Leaflet + OpenStreetMap

Objective. Display search results on an actual map and allow users to click on the map to add additional resources. As needed, the solution makes use of OpenStreetMap tiles and Leaflet for the map user interface (no Google Maps).

Please insert location here (e.g., **London**).

London

Search

+ Add New Healthcare Resource

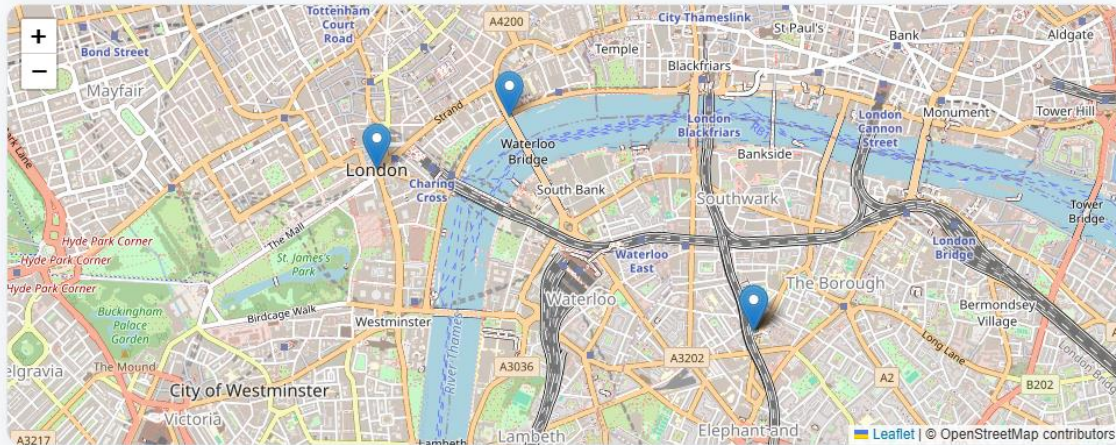


Figure 19 - Leaflet map container setup (Part D)

Please insert location here (e.g., **London**).

Search by region (e.g., London)

Search

+ Add New Healthcare Resource

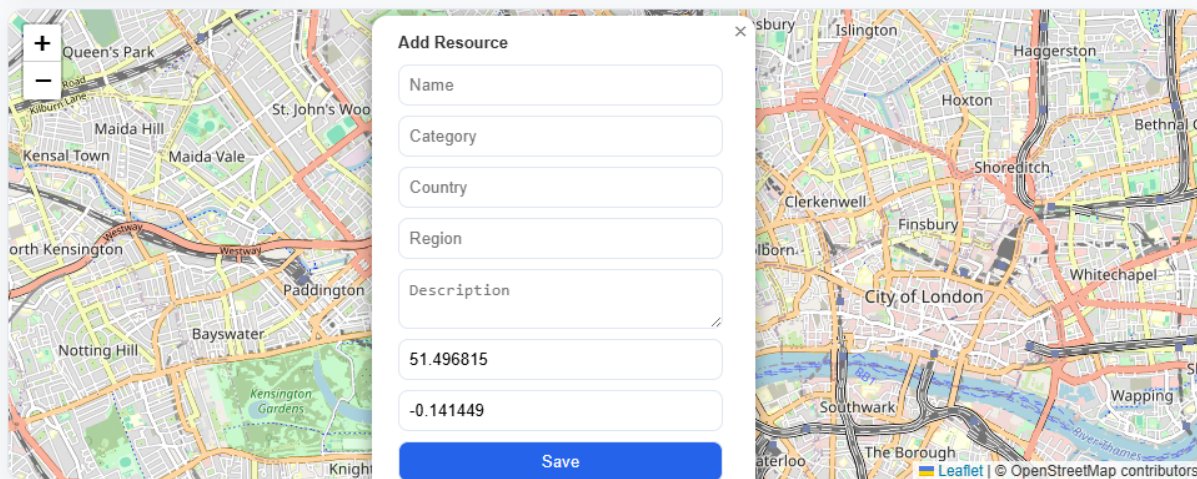


Figure 20 - Initializing Leaflet map (Task 8)

Implementation:

Task 8 – show resources as markers

I displayed a map element and included Leaflet via its CDN in index.html:

```
<!-- Leaflet -->
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.4/dist/leaflet.css" />
<script src="https://unpkg.com/leaflet@1.9.4/dist/leaflet.js"></script>
</script>
```

Figure 21 - Displaying resource markers (Task 8)

The map is initialized upon load, and a blank layer group is formed for marker management:

```
163
164 // ----- Leaflet map -----
165 let map, markersLayer;
166
167 function initMap() {
168   map = L.map('map').setView([51.505, -0.09], 10);
169   L.tileLayer('https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png', {
170     attribution: '&copy; OpenStreetMap contributors'
171   }).addTo(map);
172   markersLayer = L.layerGroup().addTo(map);
173 }
```

Figure 22 - Marker management with L.layerGroup()

GET /api/resources?region=... is called by the client when the user searches for a region. Next, renderMarkers(resources) receives the array of resources. This function gathers the coordinates of each result, removes any existing markers, adds a marker for each result, binds a popup with the **name, category, and description** (sanitized using a tiny esc() helper to prevent XSS), and then calls the map.fitBounds(...) such that the map presents the outcomes in an orderly manner:

The image shows a mobile application interface. On the left, a vertical strip of a map is visible, with the text 'Kazipin Avenue' partially legible. On the right, a white modal window titled 'Add Resource' with a close button (X) in the top right corner is displayed. The form contains the following fields:

- Name**: A text input field with the label 'Name' highlighted in yellow.
- Category**: A text input field with the label 'Category' highlighted in yellow.
- Country**: A text input field.
- Region**: A text input field.
- Description**: A text input field with the label 'Description' highlighted in yellow.
- Latitude**: A text input field containing the value '51.512629'.
- Longitude**: A text input field containing the value '-0.149002'.

At the bottom of the form is a blue button labeled 'Save'.

Figure 23 - Popups showing resource info on map


```

230  ✓ function renderMarkers(resources) {
231      markersLayer.clearLayers();
232      const bounds = [];
233
234  ✓ resources.forEach(r => {
235      const lat = Number(r.lat), lon = Number(r.lon);
236      if (!Number.isFinite(lat) || !Number.isFinite(lon)) return;
237
238      const content = makeMarkerPopup(r);
239      const mk = L.marker([lat, lon]).bindPopup(content).addTo(markersLayer);
240
241  ✓ mk.on('popupopen', () => {
242      const listEl = content.querySelector(`#rev-list-${r.id}`);
243      if (listEl) loadReviews(r.id, listEl);
244
245      const form = content.querySelector('form.rev-form');
246  ✓ if (form) {
247  ✓     form.addEventListener('submit', async (e) => {
248         e.preventDefault();
249         const fd = new FormData(form);
250         const payload = Object.fromEntries(fd.entries());
251  ✓         const res = await fetch(`/api/resources/${r.id}/reviews`, {
252             method: 'POST',
253             headers: { 'Content-Type': 'application/json' },
254             credentials: 'include',
255             body: JSON.stringify(payload)
256         });
257  ✓         if (res.status === 201) {
258             form.reset();
259             loadReviews(r.id, listEl);
260         } else if (res.status === 401) {
261             alert('Please log in again.');
```

Figure 24 - Map fitBounds for search results

Task 9 – Click the mapp to add a resource

I added a click handler to the map that, when clicked, opens a tiny form within a Leaflet popup:

```

173
174 // Allow click-to-add
175 map.on('click', onMapClick);
176 }

```

Figure 25 - Leaflet popup form for adding resource (Task 9)

The handler displays a straightforward "Please log in" message if the user is not logged in. When logged in, it displays a form that is already filled in with the current region (which is taken from the search box to maintain consistency) and the clicked lat/lon (read-only). An AJAX POST to POST /api/resources with JSON is sent when the form is submitted. The client only adds a marker and closes the window on 201 Created; otherwise, the server displays a validation error (such as missing values or an invalid lat/lon) or a 401 if the session has ended:

```

315
316 const res = await fetch('/api/resources', {
317   method: 'POST',
318   headers: { 'Content-Type': 'application/json' },
319   credentials: 'include',
320   body: JSON.stringify(data)
321 });
322 const body = await res.json().catch(() => ({}));
323
324 if (res.status === 201) {
325   const marker = L.marker([lat, lon]).addTo(markersLayer);
326   marker.bindPopup("<strong>${esc(data.name)}</strong><br/>${esc(data.category)}<br/>${esc(data.description)}<br/>");
327   map.closePopup();
328
329   const searchRegion = document.getElementById('regionInput').value.trim();
330   if (searchRegion && searchRegion.toLowerCase() === String(data.region).toLowerCase()) {
331     searchResources();
332   }
333 } else if (res.status === 401) {
334   alert('Session expired. Please log in again.');
```

Figure 26 - Adding resource via map click → POST request flow

This method maintains the database and user interface in sync and satisfies Task 9 by guaranteeing that the marker only displays after the server validates the insert.

Issues faced and their resolutions:

Marker staleness or clutter. Old markings remained across searches without clearing. This was resolved by using a specific L.layerGroup() and invoking clearLayers() prior to redrawing.

Bounds when there are no results. fitL added a guard and now display a helpful "No resources found" message in place of bounds on an empty set, which throws no error but does not advance the map.

Popup security/XSS. I included a little esc() function to safely inject text into HTML popups because resource content is user-provided.

PART E – Logins and sessions

What I've build:

In order for the app to determine who is logged in and enforce permissions for protected operations, I wired a small set of auth endpoints and UI hooks and implemented a cookie-backed session system using express-session.

Task 10 – session plumbing

```
29
30 // --- Middlewares ---
31 app.use(express.static(path.join(__dirname, 'public')));
32 app.use(express.json());
33 app.use(express.urlencoded({ extended: true }));
34 app.use(session({
35   secret: 'mySecretKey',
36   resave: false,
37   saveUninitialized: false,
38   cookie: { httpOnly: true, sameSite: 'lax', path: '/' }
39 }));
40
```

Figure 27 - Session plumbing (express-session setup) (Task 10)

This only saves sessions upon a successful login and generates a signed session cookie.

API for signup and login.

A new user is inserted using POST /api/register (easy, no hashing, suitable for coursework).

After successfully locating the user, POST /api/login sets:

```
69
70 app.post('/api/login', (req, res) => {
71   const { username, password } = req.body;
72   db.get('SELECT * FROM users WHERE username = ? AND password = ?',
73     [username, password],
74     (err, user) => {
75       if (err) return res.status(500).json({ message: 'Database error' });
76       if (!user) return res.status(401).json({ success: false, message: 'Invalid username or password' });
77       req.session.user = { id: user.id, username: user.username, isAdmin: user.isAdmin };
78       return res.json({ success: true, message: 'Login successful!' });
79     });
80 });
81
```

Figure 28 - Login API success response

When credentials match, { success: true } is the answer.

Enduring login banner:

- If you are logged in, GET /api/session returns { username, isAdmin}; if not, it returns { username: null}.
- When **index.html** loads, a tiny script calls /api/session and modifies the header:
 - The message "👋 Welcome, " appears when logging in. and displays Logout.
 - When logged out, Login/Register is displayed and Logout is hidden.
- As necessary, the notice remains visible across reloads because this check is performed on each page load.

Log out.

- GET /logout deletes the cookie and ends the server session:

```
86
87  app.get('/logout', (req, res) => {
88    req.session.destroy(() => {
89      res.clearCookie('connect.sid', { path: '/' });
90      res.redirect('/index.html');
91    });
92  });
```

Figure 29 - Logout API flow

The Logout link on the front end leads to /logout.

Although the brief states that a signup form is not required, I nevertheless included POST /api/register (as well as a basic page) to facilitate testing. The prerequisite is the route itself.

Task 11 – Restrict actions to logged-in users

I created a reusable guard:

```
40
41  function requireLogin(req, res, next) {
42    if (req.session && req.session.user) return next();
43    return res.status(401).json({ error: 'Login required' });
44  }
45
```

Figure 30 - requireLogin guard middleware (Task 11)

After that, I secured the vulnerable endpoints:

- To add a resource, POST /api/resources
- PUT:id/recommend (liking/recommending) /api/resources
- POST (posting reviews) /api/resources/:id/reviews

Friendly errors are displayed by the client:

- "Please log in to recommend resources" is displayed on index.html if a PUT /recommend return of 401.
- If the user is not logged in, a Leaflet popup on the map click-to-add flow prompts them to log in; if the server returns 401, it notifies them that the session has expired. Kindly log in once more.
- When not logged in, addResource.html hides the form and shows a notification.

Notes, issues, and solutions

state drift following logout. A reload might still display the previous name if the user interface cached it. I resolved this by rerouting to index.html upon logout and by always reading /api/session upon load.

Scope of security: I utilized basic sessions and left passwords in plaintext for the courses. In a practical application, I would think about token-based authentication like OAuth2 and hash passwords (like bcrypt).

As a result, only authenticated users are able to add or promote resources, and users can log in, see a persistent "Logged in as..." notification, then log out. Unauthorized attempts result in user-friendly front-end messages and clear HTTP 401s.

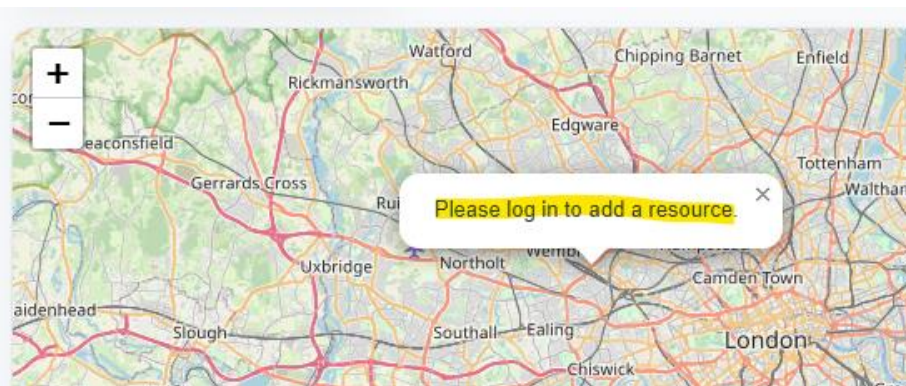


Figure 31 - Front-end handling of 401 unauthorized

PART F – Implementing Additional Features

Task 12 – Reviews API (server)

As an addition I've introduced a small reviews subsystem so user can leave feedback on a resource.

```
12
13 // Create reviews table (rating + comment) once
14 v db.serialize(() => {
15     db.run('PRAGMA foreign_keys = ON');
16     v db.run(`
17         CREATE TABLE IF NOT EXISTS reviews (
18             id INTEGER PRIMARY KEY AUTOINCREMENT,
19             resource_id INTEGER NOT NULL,
20             user_id INTEGER NOT NULL,
21             rating INTEGER NOT NULL CHECK(rating BETWEEN 1 AND 5),
22             comment TEXT NOT NULL CHECK(length(comment) <= 500),
23             created_at TEXT DEFAULT CURRENT_TIMESTAMP,
24             FOREIGN KEY(resource_id) REFERENCES healthcare_resources(id) ON DELETE CASCADE,
25             FOREIGN KEY(user_id) REFERENCES users(id) ON DELETE CASCADE
26         )
27     `);
28 });
29
```

Figure 32 - Reviews API (GET / POST implementation) (Task 12)

Returns latest reviews for a resource, joined with the author's username:

- GET /api/resources/:id/reviews

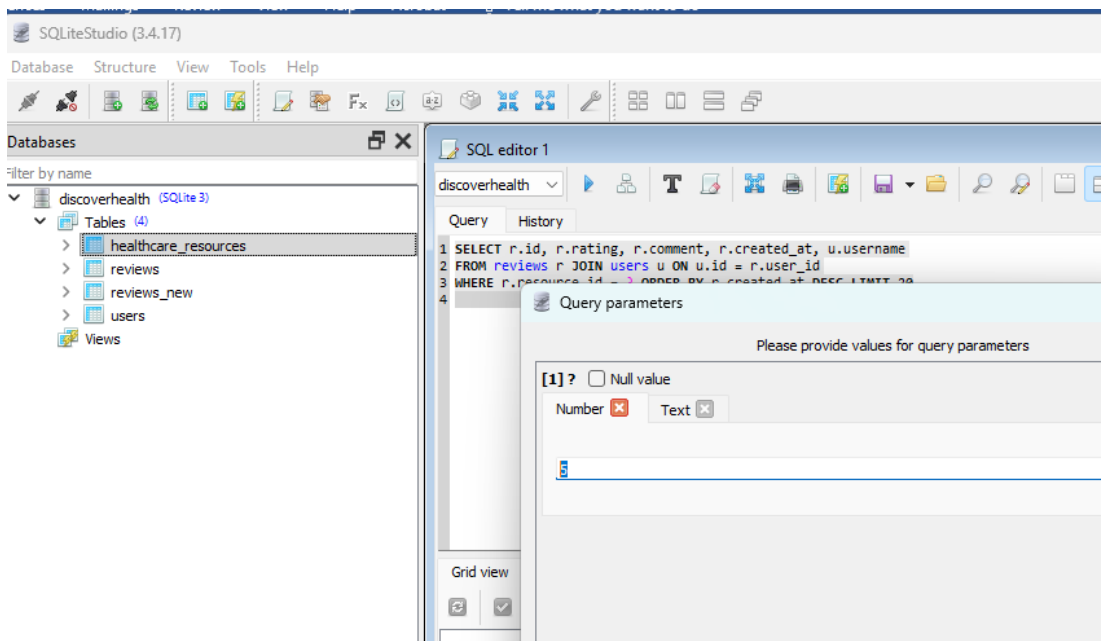


Figure 33 - Reviews query with username join (Task 12)

- POST (protected) /api/resources/:id/reviews

Requires that the user be logged in (requireLogin). Verifies:

- resourceId in numeric form from :id
- rating between 1 and 5.
- **comment** that is not empty (≤ 500 chars)
- resource existence (FK guarantees referential integrity; if necessary, I also check for more understandable 404s before inserting)

When I successfully insert the entry, I instantly re-query the created review (connected to users) to provide the client with an object that is ready for rendering:

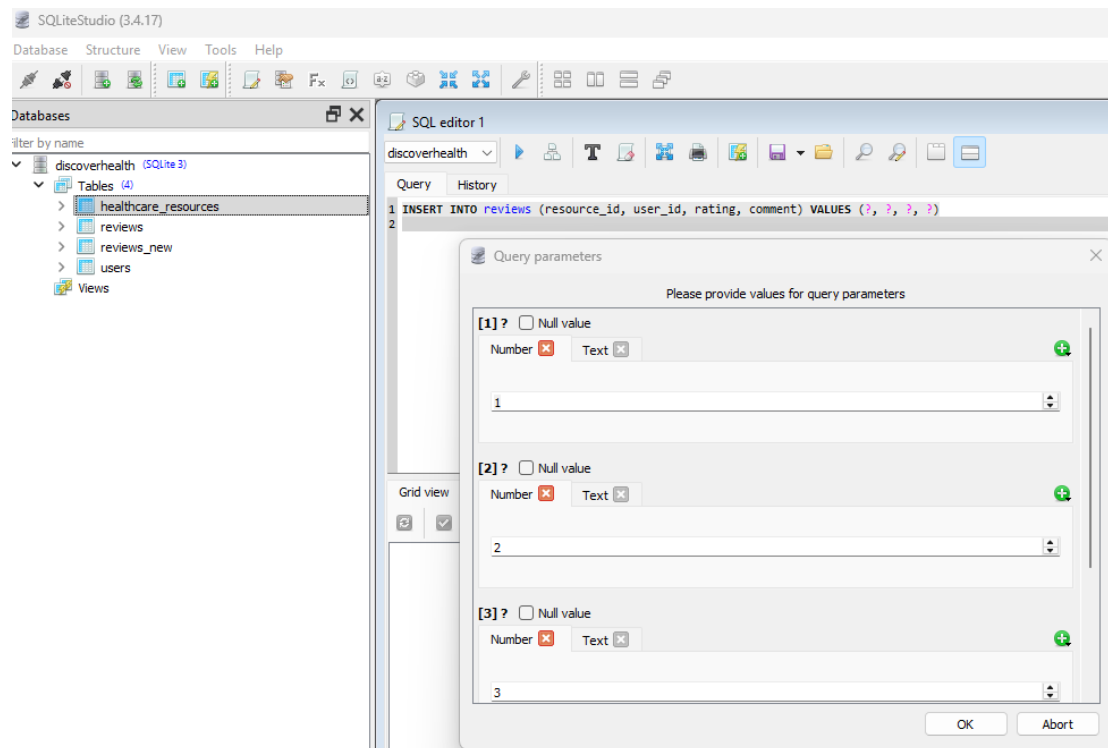


Figure 34 - Review insertion and re-query logic

Robustness and security:


The server returns consistent errors: 400 for incorrect input, 401 if the user is not logged in, and 500 for database issues; all SQL is parameterized; express-session links the review to req.session.user.id. In order to erase cascade (i.e., if a resource is destroyed, its reviews are automatically cleaned up), I activated PRAGMA foreign_keys=ON.

Gotchas I struck and made fixes:

- Clashes in the old schema. Prior to this, I just had one review text column, and I standardized on rating and commenting. A one-time migration, sometimes known as a rebuild, was required if a database had the old column.
- Session/cookie behavior. The front end utilizes credentials: 'include' on fetches that require auth to ensure the session cookie is provided.

Task 13 – Reviews in Leaflet marker propus (client)

Marker popup user interface. Every search result is also displayed as a map marker in index.html. In order to create dynamic HTML for popups:

- Header: name, description, category, and  count as of right now.
- A placeholder for a reviews list: Reviews are loading:
- An inline review form is displayed if isLoggedIn is true:
 - (1–5 stars) select[name=rating]
 - [name=comment] textarea
 - The "Post review" button

To prevent XSS when injecting HTML, I purposefully use a simple esc() helper to escape any dynamic strings.

Part G – Add a small React front-end

DiscoverHealth (React)

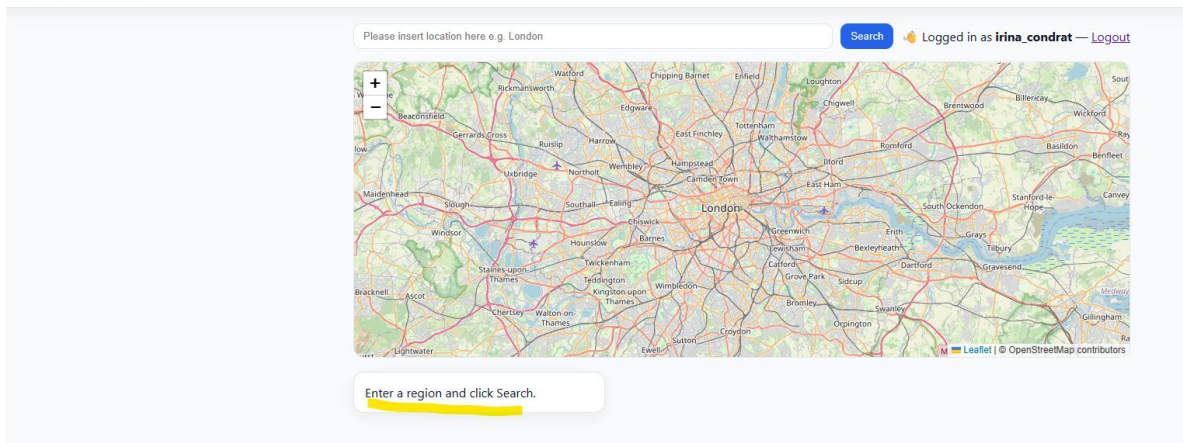


Figure 35 - React front-end integration (Part G)

```
JS app.js  react.html x
public > < react.html > < html > < body >
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8" />
5    <title>DiscoverHealth • React Demo</title>
6    <meta name="viewport" content="width=device-width,initial-scale=1" />
7    <link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.4/dist/leaflet.css" />
8    <script src="https://unpkg.com/leaflet@1.9.4/dist/leaflet.js"></script>
9
10   <!-- React via CDN + Babel (OK for coursework/demo) -->
11   <script crossorigin src="https://unpkg.com/react@18/umd/react.development.js"></script>
12   <script crossorigin src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
13   <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
14
15   <style>
16     body{margin:0;font:16px/1.45 system-ui,Segoe UI,Roboto,Arial;background: #f8f9fa;color: #6c757d}
17     header{display:flex;justify-content:space-between;align-items:center;padding:14px 20px;background: #fff;border-bottom:1px solid #dee2e6}
18     .brand{font-weight:800}
19     .container{max-width:1000px;margin:0 auto;padding:20px}
20     .row{display:flex;gap:10px;align-items:center;flex-wrap:wrap}
21     input,button{padding:8px 12px;border:1px solid #dee2e6;border-radius:10px}
22     button.primary{background: #007bff;color: #fff;border-color: #007bff}
23     #map{height:380px;border-radius:14px;border:1px solid #dee2e6;background: #fff;margin-top:16px}
24     .grid{display:grid;grid-template-columns:repeat(auto-fill,minmax(280px,1fr));gap:14px;margin-top:18px}
25     .card{background: #fff;border:1px solid #dee2e6;border-radius:14px;padding:14px;box-shadow: 0 6px 20px #dee2e6}
26     .meta{color: #6c757d;font-size:14px;margin: .25rem 0 .5rem 0}
27     .badge{display:inline-block;background: #28a745;color: #fff;border:1px solid #28a745;border-radius:999px;padding:2px 10px}
28   </style>
29 </head>
30 <body>
31   <header>
32     <div class="brand">DiscoverHealth (React)</div>
33     <div id="auth"></div>
34   </header>
```

Figure 36 - React page snip (react.html)

<http://localhost:3000/react.html>

Conclusions:

I successfully implemented all of the DiscoverHealth application's main components in this project, carefully adhering to the requirements of the evaluation brief. I made sure the system could consistently serve, generate, and suggest healthcare resources while protecting against SQL injection and data integrity problems by starting with a secure and parameterized REST API (Part A).

Real-time resource creation, search, and recommendation changes were supported by the AJAX-driven front-end (Part B), which enabled smooth interaction without page reloads. Building on this, I added strong error-handling (Part C), which keeps usability and resistance against erroneous data entry while providing users with unambiguous, inline feedback.

The application changed from a text-based interface to a location-aware directory by integrating Leaflet and OpenStreetMap (Part D). This improved the user experience by allowing resource submission immediately through map clicks and showing healthcare resources on an interactive map.

Access control and authentication gained prominence with the implementation of login and session management (Part E). In addition to enabling permanent sessions between reloads, this also implemented role-appropriate limitations, guaranteeing that only authenticated users could perform sensitive tasks like adding, recommending, or reviewing resources.

By allowing user-generated ratings and comments, the reviews subsystem (Part F) enhanced the platform even further. It integrated easily with the Leaflet-based user interface and the API. This offered a more comprehensive, community-driven perspective on medical resources.

In order to showcase modularity, scalability, and contemporary web development techniques that could form the basis for further advancements, I lastly expanded the program with a React-based front-end (Part G).

To sum up, DiscoverHealth is a comprehensive, safe, and user-friendly healthcare resource directory. Starting with the API and progressing through AJAX interactivity, validation, mapping, authentication, and reviews, each development step built on the one before it, resulting in a reliable and expandable program. This project showcases my ability to combine front-end and back-end technologies into a cohesive, safe, and useful system in addition to showing off the usefulness of Node.js, Express, SQLite, and Leaflet.

