**PROG2002 – Web Development II**

**Assignment 2: Use Case (A Dynamic Website)**

**Student ID:** **Last Name:** **First Name:**

**Title of the project:**

City Charity Events — Dynamic Web Platform for Local Fundraising

**Introduction/Motivation**

Local charities and community groups frequently run time-limited events (runs, galas, auctions, concerts) to raise funds and public awareness, but small organizers often lack a simple, centralised place to list events and accept registrations. This project delivers a lightweight, standards-based dynamic website that connects citizens with local charity events. The platform demonstrates practical full-stack skills (Node.js/Express backend, MySQL data store, and a client-side JavaScript frontend) and provides a clear, maintainable example of how small organisations can expose event data via RESTful APIs and a polished user-facing interface.

**Problem Statement**

Small charities need an approachable, maintainable website for publishing event information and tracking fundraising progress. Existing solutions can be heavyweight or require third-party platforms that charge fees or impose design constraints. The specific problems addressed by this project are: (1) storing and managing event metadata (dates, location, category, goals) in a relational database, (2) exposing that data to a responsive client that non-technical staff can use, and (3) providing intuitive search and event detail pages so citizens can discover and join events without friction.

**Solution**

The solution is a two-tier web application. The backend is a Node.js + Express application that interacts with a MySQL database. It exposes RESTful JSON endpoints for listing events, filtering/searching, returning event details, and returning reference data (categories and organisations). The frontend is a static client (HTML, CSS, vanilla JavaScript) that calls these endpoints (using fetch and Promises) to render dynamic content in the browser. Client → Server communication follows standard HTTP: the client performs GET requests to fetch lists and single resources, and the server responds with JSON payloads. This separation keeps concerns clean: the server manages persistence and business logic, the client handles presentation and interaction. The API design is intentionally RESTful so that future extensions (admin CRUD, authentication, ticketing) can be added without refactoring the client architecture.

**Web UX**

The user experience focuses on discoverability and low cognitive load. Key UX choices:

A clear global navigation (Home / Search) is present on every page so users always know how to move between flows.

The Search page provides a compact, left-side filter panel and a right-side results area. Filters are labelled, use appropriate input types (date picker for date, text for location, select for category), and allow combining criteria. A “Clear Filters” button resets the form and demonstrates DOM manipulation.

The Results view uses responsive card components with an image, short description, date, location and a “View” call-to-action to the event detail page.

The Event Detail page highlights the full description, organiser contact info, and fundraising progress (goal vs raised) so donors see impact at a glance.

The client UI was designed to be mobile-first and responsive—on narrow screens the filter panel collapses into a vertical flow. Visual hierarchy (titles, meta text, buttons) and accessible contrast ensure items are scannable.

From an implementation perspective, wireframes were implicitly followed by keeping consistent IDs and DOM structure that the JavaScript expects (search-form, results, event-detail, etc.), which simplified integration and testing.

**Data Schema**

The relational schema models three core entity types: **organizations**, **categories**, and **events**. The relationships and principal fields are:

**organizations** (1) — has many → **events**

id (INT, PK)

name (VARCHAR)

description (TEXT)

contact\_email (VARCHAR)

website (VARCHAR)

**categories** (1) — has many → **events**

id (INT, PK)

name (VARCHAR)

**events**

id (INT, PK)

organization\_id (INT, FK → organizations.id, nullable)

category\_id (INT, FK → categories.id, nullable)

name (VARCHAR)

short\_desc (VARCHAR)

description (TEXT)

start\_date (DATE)

end\_date (DATE)

location (VARCHAR)

image\_url (VARCHAR)

goal\_amount (DECIMAL)

raised\_amount (DECIMAL)

paused (TINYINT/boolean) — used to hide events that violate policy or are paused

created\_at (TIMESTAMP)

**Cardinality and constraints**

One organization can run many events (1:N). organization\_id in events captures this relationship.

One category can be associated with many events (1:N). category\_id in events captures this relationship.

Foreign keys use ON DELETE SET NULL to preserve historical event records if an organization or category is removed.

This schema supports queries for upcoming vs past events (comparing end\_date with the current date), aggregation for fundraising progress (compute raised\_amount / goal\_amount), and category/organization lookups for filtering and display.

**API design**

## Main API endpoints

All endpoints are under /api. Implemented endpoints:

GET /api/events  
Returns a list of all active events with basic metadata (used for general listings).

GET /api/events/upcoming  
Returns events whose end\_date >= CURDATE() (active/upcoming).

GET /api/events/past  
Returns events whose end\_date < CURDATE() (past events).

GET /api/events/:id  
Returns full details for a single event, including joined category and organization fields.

GET /api/categories  
Returns all event categories (used to populate category filters).

GET /api/organizations  
Returns all organizations (useful for admin or filtering extensions).

GET /api/search?date={YYYY-MM-DD}&location={text}&category={id}  
A flexible search endpoint that accepts query parameters (any subset) and filters events accordingly.

## Example: GET /api/events/:id — detailed endpoint description

**Purpose**  
Return the complete representation of a single event so the frontend can render the Event Detail page with description, organiser contact info, and fundraising values.

**Request**

HTTP method: GET

URL pattern: /api/events/:id where :id is the numeric event identifier (path parameter).

No request body is required.

**Successful response (HTTP 200)** — JSON body example:

{

"id": 3,

"organization\_id": 2,

"category\_id": 1,

"name": "City Fun Run",

"short\_desc": "Family-friendly 3km run",

"description": "A casual run for families and friends.",

"start\_date": "2025-09-05",

"end\_date": "2025-09-05",

"location": "Central Avenue",

"image\_url": "/assets/images/placeholder.jpg",

"goal\_amount": 3000.00,

"raised\_amount": 3000.00,

"paused": 0,

"created\_at": "2025-04-01T10:00:00.000Z",

"category": "Charity Run",

"organization": "Hope Runners",

"contact\_email": "contact@hoperunners.org",

"website": "https://hoperunners.example"}

**Errors**

If id is not found, the server responds with HTTP 404 and a small JSON error object, e.g. { "error": "Not found" }.

## Rationale for HTTP method choices

**GET** is used for all current endpoints because the implemented features are read-only: listing resources and retrieving resource details are idempotent and safe, which aligns with GET semantics.

**Query parameters** are used for search filtering (/api/search) because these operations are retrievals and do not change server state. Query parameters allow flexible combinations of filters without the need for a request body.

**Why not POST/PUT/DELETE now**: POST, PUT and DELETE are appropriate for creating, updating or removing resources (e.g., creating events, updating fundraising totals, or deleting a cancelled event). Those operations are reserved for the administrative phase (Assessment 3) and would require authentication/authorization to secure them. When implementing them, the server should validate inputs and return appropriate status codes (201 for create, 200/204 for successful update/delete).

**Future extension**: When adding ticket purchases or registrations, POST /api/events/:id/registrations would be appropriate to create a registration (and possibly trigger payment workflow). For partial updates (e.g., updating raised\_amount), PATCH is an appropriate choice if only a small subset of fields change.

# Security & operational notes (brief)

The server includes cors() to allow the static client to make AJAX requests; in production this should be scoped to allowed origins and combined with authentication for write endpoints.

Input used in SQL queries is parameterised using prepared statements from mysql2/promise to reduce SQL injection risk.

Large result sets could later be paginated (e.g., GET /api/events?page=2&limit=20) to support scalability.