Design and Implementation of an Event Management System Applying the Three Tier Architecture (Presentation, Application and Data Tier)

Bruno Bucalon Serra
Department of Computer Science
University of London

CONTENTS

1 Introduction 1

2 Architecture 1

3 Data Model 1

4 Extension Implementation 1
4.1 Advanced SQL . . . 1
4.2 Server-Side Search Engine . . . 2
4.3 Data Integrity Logic . . . 2

5 Conclusion 3

References 3

LIST OF FIGURES

1 Three Tier Diagram . . . 1
2 Entity Relationship Diagram (ERD) . . . 1

Design and Implementation of an Event Management System Applying the Three Tier Architecture (Presentation, Application and Data Tier)

***Abstract*—This report describes the development of a website, together with its databases and own server about an event planner.**

1. Introduction

This report documents the implementation of an Event Management System using a three-tier architecture. The application utilizes Node.js and Express for the server-side logic, EJS for the presentation tier, and SQLite for data storage.

2. Architecture

The architecture was defined applying the separation of concerns properly: the client handles user interaction, the Express server processes business logic and routing, and the SQLite database applies data integrity, illustrated in Figure 1.

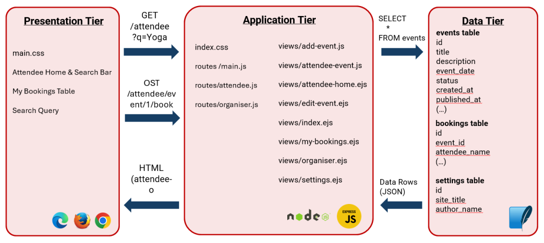

Fig. 1. Three Tier Diagram

3. Data Model

The database schema is divided into three main entities. An implementation of a One-to-Many relationship between Events and Bookings was done. The bookings table uses a Foreign Key (event_id) to link each reservation to a specific event, ensuring referential integrity via the PRAGMA foreign_keys=ON directive.

Figure 2 presents the Entity Relationship Diagram (ERD) designed for the application, utilizing Crow's Foot notation to represent cardinality. The database schema is implemented in SQLite and consists of three distinct entities: events, book and settings table.

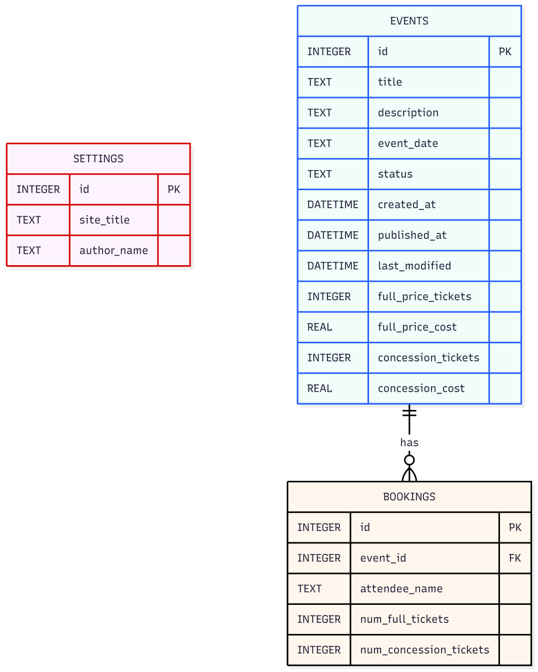

Fig. 2. Entity Relationship Diagram (ERD)

4. Extension Implementation

In order to improve the project, it was implemented two server-side extensions to improve functionality beyond basic CRUD (Create, Read, Update, and Delete) operations: an Advanced SQL Query, Server-Side Search Engine, and a Data Integrity Logic.

4.1 Advanced SQL

The implementation a User Booking History feature ("My Bookings") was done. This functionality allows users to view a list of all tickets they have purchased. Moreover, the list displays not just the transaction details (ticket count), but also the specific Event Title and Date for each booking. Since the bookings table only stores the event_id and not the event details, this combines data from two separate database tables simultaneously.

To implement this, a new GET route defined as /my-bookings in the routes/attendee.js file was created. The first challenge was that a simple SELECT * FROM bookings would return an ID (e.g., event_id: 5) which is meaningless to the user. To resolve this, a complex SQL query was developed using an INNER JOIN. This command performs the intersection between two sets, from the real problem, it matches rows from the bookings table with rows from the events table wherever the bookings.event_id matches the events.id. The result is a single virtual table containing combined information, which is then passed to the EJS template for rendering.

Finally, the code snipped demonstrates the implementation of the relational join:

```
router.get("/my-bookings", (req, res) => {
    // Advanced SQL: Using INNER JOIN to
        combine data from two tables
    // linking the Foreign Key (bookings.
        event_id) to the Primary Key (events.id
        )
    const sql = `
        SELECT bookings.*, events.title,
            events.event_date
        FROM bookings
        JOIN events ON bookings.event_id =
            events.id
        ORDER BY bookings.id DESC`;

    global.db.all(sql, [], (err, bookings) =>
        {
        if (err) {
            return res.send("Error retrieving
                booking history.");
        }
        res.render("my-bookings", { bookings
            });
    });
});
```

It applies course techniques by setting up an Express route and passing the retrieved data object to an EJS view for display and using SQL queries to process and manipulate data.

This feature demonstrates technical proficiency beyond the basic module requirements since applies relational algebra. The course materials primarily focus on simple CRUD operations on single tables, this implementation uses fundamental concept of Relational Database Management Systems (RDBMS) and Discrete Mathematics.

4.2 Server-Side Search Engine

A Dynamic Search Engine was implemented to allow users to filter the list of events based on their titles. Unlike client-side filtering (where all data is downloaded to the browser and hidden via JavaScript), this extension performs the filtering logic on the server before the data is ever sent to the client.

In order to perform this implementation, the main entry route was modified in the routes/attendee.js file. The implementation works by catching the HTTP GET request and checking for a query string parameter named search (e.g., ?search=Yoga). If a search term is present, the server does not run the standard ”Select All” query. Instead, it dynamically constructs a SQL statement including a WHERE clause. I utilized the SQL LIKE operator combined with wildcards (%) to match the user’s input against the title column in the database. Finally, the filtered list and the original search term are passed to the EJS template to render the page and repopulate the input field.

```
router.get("/", (req, res) => {
    const search = req.query.search || "";

    const sql = `SELECT * FROM events
                 WHERE status='published' AND
                     title LIKE ?
                 ORDER BY event_date ASC`;
    global.db.all(sql, [`%${search}%`], (err,
        events) => {
        if (err) {
            console.error(err);
            return res.send("Database error");
        }
        res.render("attendee-home", { events,
            search });
    });
});
```

This extension uses several concepts taught in the module:

1) Express Routing: Handling GET requests and accessing URL parameters via req.query.
2) Templating (EJS): Rendering a list of items using loops and conditionally displaying messages if no results are found.
3) Basic to intermediate SQL queries, using SELECT clauses and filtering with wildcards.

With this extension, the project goes beyond what was taught on classes, since uses partial string matching using the LIKE operator and % wildcards, allowing users to type ”Yog” and successfully find ”Yoga Classes”, which requires understanding string manipulation within SQL.

4.3 Data Integrity Logic

A data integrity logic was implemented with a server side inventory validation for tickets. This feature allows users to select ticket quantities (Full Price or Concession) and confirm a reservation. The system includes a ”Sold Out” mechanism that prevents the application from selling more tickets than the event capacity allows, providing data integrity.

The implementation was done by creating a POST route at /book/:id in routes/attendee.js to handle form submissions.

When a user clicks ”Book”, the server first queries the database to get the current number of tickets available for that specific event.

Then, it compares the requested number of tickets against the available stock.

Finally, if there is sufficient stock, the server executes two SQL commands: removing the ticket count in the events table (UPDATE) and creating a new instance in the bookings table (INSERT). If stock is insufficient, the transaction is rejected.

This extension uses web development concepts covered in the module:

- Form Handling, using POST requests.
- SQL Modification, using INSERT and UPDATE.

This implementation goes beyond of what is expected, since it:

- Applies server-side business logic instead of just accepting user input.
- Uses the result of a database query (the stock check) to decide whether to run subsequent queries.

```
router.post("/book/:id", (req, res) => {
    global.db.get("SELECT * FROM events WHERE
        id=?", [eventId], (err, event) => {

        // Checks if requested tickets exceed
            available stock
        if (event.full_price_tickets < numFull
             || event.concession_tickets <
            numConc) {
            return res.send("Error: Not enough
                 tickets available.");
        }

        global.db.run(`UPDATE events SET
            full_price_tickets =
            full_price_tickets - ? WHERE id =
            ?`,
            [numFull, eventId], (err) => {

                global.db.run(`INSERT INTO
                    bookings (event_id,
                    attendee_name) VALUES (?,
                    ?)`,
                     [eventId, attendeeName], (
                         err) => {
                         res.redirect("/
                             attendee/my-
                             bookings");
                });
        });
    });
});
```

5. Conclusion

The project initially presented several challenges to be implemented, regarding handling server, client with a data tier. However, after understanding the concepts of REST APIs, and server-side logic, the development started to become easier. Consequently, I was able to go beyond by implemented the three extra features, mentioned in the previous session.

The final result is a functional, three-tier application that successfully meets and even go beyond the project's architectural goals.

References

[1] OpenJS Foundation, *Express - Fast, unopinionated, minimalist web framework for Node.js*. [Online]. Available: https://expressjs.com/. [Accessed: Jan. 11, 2026].

[2] SQLite Development Team, *SQLite Documentation*. [Online]. Available: https://www.sqlite.org/docs.html. [Accessed: Jan. 11, 2026].

[3] Mozilla Developer Network, *JavaScript reference - MDN Web Docs*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [Accessed: Jan. 11, 2026].

[4] OpenJS Foundation, *Node.js Documentation*. [Online]. Available: https://nodejs.org/en/docs/. [Accessed: Jan. 11, 2026].