Dance Registration Portal — System Architecture and Design

Audience: Non‑frontend stakeholders who need a clear, end‑to‑end understanding of how the portal is built and operates.

1) High‑Level Overview

Student/Admin Browser

Express Server (server.js)

DB Abstraction

SQLite (dev)

PostgreSQL (Railway)

Mailer (Nodemailer)

Static Frontend (public/\*)

Venmo / PayPal

Local Development

Production (Railway)

Node/Express

SQLite (database/registrations.db)

public (index.html, JS, CSS)

Node/Express (server.js)

PostgreSQL (Railway)

public (served by Express)

SMTP (Nodemailer)

Summary: Classic 3‑tier web app.

• Frontend: Static HTML/CSS/JS (no framework), served by Express.

• Backend: Node.js (Express) REST API with session‑based admin auth.

• Database: SQLite in development, PostgreSQL on Railway in production.

Runtime Components

• Express server (single process)

• DatabaseConfig abstraction (switches SQLite/Postgres)

• Nodemailer email transport (SMTP/Gmail)

• Venmo/PayPal integrations (client‑side)

Key Properties

• Static assets and API served by the same Express app

• Slot‑based course architecture (time, capacity, pricing per slot)

• Server computes schedule\_info from slots for consistent UI

• Booleans normalized across SQLite (0/1) and Postgres (true/false)

2) Deployment Architecture

Environment

Details

Local Development

• Express server + SQLite file database

• Static public/ served by Express

• Run with npm run dev (nodemon)

Production (Railway)

• Auto‑deploy on git push

• PostgreSQL via Railway (DATABASE\_URL)

• Health checks via railway.toml

• Email transport auto‑detected (service/host/Gmail fallback)

3) Database Design

Core entities and relationships:

• Students (1) —> (many) Registrations

• Courses (1) —> (many) Registrations

• Courses (1) —> (many) Course Slots (1) —> (many) Course Pricing

Table

Selected Columns

Description

students

id, email (unique), first\_name, last\_name, phone, dance\_experience, instagram\_handle, created\_at

Student contact and profile data

courses

id, name, description, course\_type, start\_date, end\_date, instructor, schedule\_info, is\_active

Top‑level course/series record

registrations

id, student\_id, course\_id, payment\_status, payment\_amount, payment\_method, registration\_date

Student’s spot with payment tracking

course\_slots

id, course\_id, slot\_name, difficulty\_level, capacity, day\_of\_week, practice\_date, start\_time, end\_time, location

Per‑slot time/capacity/location (source of truth for schedule)

course\_pricing

id, course\_slot\_id, pricing\_type (full\_package/drop\_in), price

Per‑slot pricing

Important: The API computes schedule\_info from course\_slots so all UIs render a consistent, time‑rich schedule string.

4) Backend API (Express)

Authentication

• POST /api/admin/login — session‑based login

• POST /api/admin/logout — logout

• GET /api/admin/status — check login (for UI bootstrapping)

Courses & Slots

• GET /api/courses?active\_only=true — public course list with computed schedule\_info, capacity, compatibility prices

• POST /api/courses — create course + slots + pricing (requireAuth)

• PUT /api/courses/:id — partial update; optionally replace slots (requireAuth)

• POST /api/courses/:id/slots — add slot with pricing (requireAuth)

• PUT /api/courses/:courseId/slots/:slotId — update slot + pricing (requireAuth)

• DELETE /api/courses/:courseId/slots/:slotId — delete slot (guards last slot) (requireAuth)

• GET /api/admin/debug/course-slots/:courseId — inspect raw slots (requireAuth)

Students & Registrations

• POST /api/register — self‑registration (creates/updates student; inserts pending registration)

• GET /api/registrations — list registrations (requireAuth; filters: course\_id, payment\_status)

• GET /api/admin/registrations — alias for admin list (requireAuth)

• GET /api/admin/registrations/count — counts (total, pending, completed) (requireAuth)

• PUT /api/registrations/:id/payment — update payment (student/PayPal path)

• PUT /api/admin/registrations/:id/confirm-payment — confirm Venmo payment; triggers email (requireAuth)

• POST /api/admin/registrations/:id/resend-confirmation — resend confirmation (requireAuth)

• PUT /api/admin/registrations/:id/assign-student — link student to orphaned reg (requireAuth)

• GET /api/admin/registrations/missing-contact — missing email/name list (requireAuth)

Settings & Ops

• GET/PUT /api/settings — registration\_open, venmo\_username, email toggles

• POST /api/generate-venmo-link — deep link and QR context

• GET /api/admin/debug-email-config — show chosen transport (no secrets)

• POST /api/admin/test-email-transport — verify SMTP connectivity

• POST /api/admin/reset-keep-course — clear ALL registrations; keep one course active or delete others

• DELETE /api/admin/clear-all-courses — delete courses & regs (destructive)

5) Frontend Components

Student Portal

• Course Cards (from /api/courses: slots, dates, pricing, availability)

• Registration Form (crew‑practice name/Instagram field toggle)

• Payment Section (Venmo deep link on mobile, QR on desktop)

• Confirmation (summary after registration)

Admin Dashboard

• Auth modal (login/logout)

• Dashboard stats + recent registrations

• Courses grid (capacity, schedule, create/edit, activate/deactivate)

• Registrations (filters, details, quick confirm, assign student, export CSV)

• Settings (registration\_open, Venmo username, email notifications)

• Quick Actions (reset‑keep‑course, delete others, email debug/test)

6) Admin Feature Map (Executive)

• Open/Close Registration globally

• Monitor real‑time registrations & revenue

• Confirm Payments and send emails

• Export rosters (CSV)

• Cleanup/Reset between series (keep active / delete others)

7) Security & Configuration

• Sessions via express‑session; sameSite=lax; httpOnly

• Password hashing via bcryptjs

• Parameterized queries across DBs

• Nodemailer with transport verification; debug endpoints expose no secrets

• Env: NODE\_ENV, DATABASE\_URL (prod), EMAIL\_\*, SESSION\_SECRET

8) Key Data Flows (Readable Summaries)

8.1 Student Registration

1 Student selects course from /api/courses

2 Submits form → server creates/updates student and inserts pending registration

3 Venmo link generated (mobile) or QR presented (desktop)

4 Student pays; page shows confirmation summary; admin later confirms

8.2 Admin Confirm Payment + Email

1 Admin clicks Quick Confirm / Confirm Venmo

2 Server sets payment\_status=completed

3 Server loads course+slots, computes schedule\_info

4 Email queued/sent via Nodemailer (non‑blocking)

9) Admin Ops Cheat‑Sheet (with SQL)

Track Students for a Series

• Registrations → filter by Course; optionally by Status

• Export CSV for rosters

SELECT r.id, r.registration\_date, r.payment\_status, r.payment\_amount,

s.email, s.first\_name, s.last\_name

FROM registrations r

JOIN courses c ON c.id = r.course\_id

LEFT JOIN students s ON s.id = r.student\_id

WHERE c.name ILIKE 'Dreamers Crew Practice - Sept 12'

ORDER BY r.id;

Consolidation After Closing

• Toggle registration\_open OFF

• Export CSV roster; reconcile pending vs completed

SELECT c.name,

SUM(CASE WHEN r.payment\_status='completed' THEN r.payment\_amount ELSE 0 END) AS revenue

FROM registrations r

JOIN courses c ON c.id = r.course\_id

WHERE c.name ILIKE 'Dreamers Crew Practice - Sept 12'

GROUP BY c.name;

Fix Wrong/Incomplete Registrations

• Assign student: Admin → Registrations → Assign Student

• Delete wrong entry (until UI delete exists): DELETE FROM registrations WHERE id = 123;

10) Files & Key Modules

• server.js — Express app, routes, middleware, date helpers

• database-config.js — DB abstraction (SQLite dev, Postgres prod)

• utils/schedule.js — compute schedule\_info; fetch course with slots

• utils/mailer.js — Nodemailer transport and utilities

• public/index.html + public/js/registration.js — Student portal

• public/admin.html + public/js/admin.js — Admin dashboard