ElectiveHub: Full-Scope Project Blueprint

This document outlines the comprehensive plan to build the complete "ElectiveHub" application, aligning with the 8-week timeline and full feature set from your project blueprint.

1. Project Vision (Full-Scope)

To build a real-time, multi-user web portal for elective enrollment that:

1. **Prevents Overbooking** using transaction-safe database operations.
2. **Resolves Conflicts** by checking for schedule and credit overlaps.
3. **Updates Live** using WebSockets so all users see seat counts change instantly.
4. **Automates Waitlists** with a fair, time-based system that notifies students and handles confirmation.

2. Full-Scope Feature Implementation Plan

This maps directly to your 8-week timeline, with implementation details for each module.

Module 1: Core Backend & Auth

- **Goal:** A secure backend with a working database and user authentication.
- **Database Schema:** Implement the **full 4-table schema** from your PDF (users, courses, schedules, registrations) using Prisma. This is more robust than the 2-table MVP.
- **Authentication:**
  - **Backend:** Implement JWT (JSON Web Token) based authentication.
  - Create two login routes: /api/auth/student-login and /api/auth/admin-login.
  - Create protected route middleware: isStudent and isAdmin to protect respective API endpoints.
  - **Admin:** Seed the database with a default admin account.
  - **Student:** Create a /api/auth/student-register endpoint for new students. Hash passwords using bcrypt.

Module 2: Admin Panel

- **Goal:** A functional admin dashboard for full course management.
- **Backend (Admin-Only API):**
  - Full CRUD API for courses (/api/admin/courses).
  - Full CRUD API for schedules (/api/admin/schedules). When an admin creates a course, the frontend should immediately allow them to add schedule slots (e.g., "Mon 10:00-11:00", "Wed 10:00-11:00") which creates records in the schedules table linked to that course.id.
- **Frontend (Admin Dashboard):**

- Create a protected route /admin/dashboard that is only visible to logged-in admins.
- **UI:** A table view of all courses.
- **Functionality:**
  - "Create Course" button opens a modal. The form should include all course fields *and* a sub-form to add one or more schedule entries.
  - "Edit" button on each row, loading the same form.
  - "Delete" button with a confirmation modal.

Module 3: Student Catalog & Core Registration

- **Goal:** Students can see courses and register for them (with race-condition protection).
- **Backend (Student API):**
  - GET /api/courses: An endpoint for students to fetch all available courses. This query should also "join" (or include in Prisma) the related schedules for each course.
  - POST /api/register/:courseId: The **core registration logic**. This is the first critical transaction.
    1. Check if user is already registered.
    2. Check for conflicts (see Module 5).
    3. Check if remaining_seats > 0.
    4. If yes, start a prisma.$transaction to:
       - UPDATE courses SET remaining_seats = remaining_seats - 1.
       - CREATE registrations record with status: 'REGISTERED'.
    5. If seats are zero, check waitlist (see Module 6).
- **Frontend (Student Catalog):**
  - A /catalog page showing course cards.
  - Each card displays course info and remaining_seats.
  - "Register" button calls the POST /api/register endpoint.
  - Implement search and filtering (by Department, "Show Available Only").

Module 4: Real-Time Integration

- **Goal:** All users see seat counts change live without a page refresh.
- **Backend (Socket.IO):**
  1. Integrate Socket.IO with your Express server.
  2. After any successful registration or de-registration (inside the POST /api/register or POST /api/drop logic), emit a global message:
     io.emit('seat-update', { courseId: '...', newSeatCount: X, newWaitlistCount: Y });
- **Frontend (Socket.IO Client):**
  1. In your main React/Next.js layout (or using Zustand/Context), initialize a Socket.IO client.
  2. Listen for the seat-update event.
  3. When the event is received, update your global state (e.g., Zustand store). This will cause React to automatically re-render *only* the course card that changed.
  4. Use a toast library (like react-hot-toast) to show instant "Success!" or "Error!"

messages from API calls, as your PDF mentions.

Module 5: Student Dashboard & Conflict Logic

- **Goal:** Students can see their courses. The system now *prevents* invalid registrations.
- **Backend (Conflict Logic):** This is the *most complex* part of the registration logic.
  - Modify POST /api/register/:courseId (from Module 3):
  - **Before** checking for seats, perform these checks:
    1. **Credit Check:** Get all registrations for the user with status: 'REGISTERED'. Sum the credits of all associated courses. Add the credits of the *new* course. If total > max_credit_limit (e.g., 20), return a 400 error: "This registration would exceed your credit limit."
    2. Schedule Conflict Check:
       a. Get all schedules for the new course.
       b. Get all schedules for all courses the student is already registered for.
       c. Run a nested loop to compare every new_schedule with every existing_schedule.
       d. Conflict = (new.day_of_week === existing.day_of_week) AND (new.start_time < existing.end_time) AND (new.end_time > existing.start_time)
       e. If a conflict is found, return a 400 error: "Schedule conflict detected with [Existing Course Title]."
- **Frontend (Student Dashboard):**
  - Create a protected route /dashboard.
  - Fetch from /api/my-registrations (a new endpoint you'll build).
  - Display two lists: "Registered Courses" and "Waitlisted Courses".
  - Add a "Drop Course" button to each.

Module 6: The Automated Waitlist System

- **Goal:** A fair, automated waitlist that promotes the next student in line.
- **Backend (Waitlist Logic):**
  - **Join Waitlist:** Modify POST /api/register/:courseId. If remaining_seats == 0 but waitlist_current < waitlist_capacity, start a transaction to:
    - UPDATE courses SET waitlist_current = waitlist_current + 1.
    - CREATE registrations record with status: 'WAITLISTED'.
  - **De-registration Logic (Trigger):** Modify the "Drop Course" API (POST /api/drop/:registrationId):
    1. Delete the student's REGISTERED registration.
    2. **Check for Waitlist:** findFirst({ where: { courseId, status: 'WAITLISTED' }, orderBy: { registered_at: 'asc' } }).
    3. **If a waitlisted student exists:**
       - Update their registrations record: status: 'PENDING_CONFIRMATION' and set confirmation_expires_at: NOW() + 24 hours.
       - **Do NOT** increment remaining_seats. The seat is now "reserved" for this pending student.

        - Trigger the email notification (see Module 7).
    4. **If no waitlist:** UPDATE courses SET remaining_seats = remaining_seats + 1.
- **Backend (Waitlist Confirmation):**
    - Create POST /api/waitlist/accept/:registrationId. This is the link the-student clicks in their email.
    - **Logic:** Check if the registrationId is valid, status is 'PENDING', and confirmation_expires_at has not passed.
    - If valid, update status: 'REGISTERED'. The student is now in the course.
- **Backend (Scheduled Job - The "Reaper"):**
    - Use a cron job (e.g., node-cron or a Vercel/Railway cron) to run every hour.
    - **Job Logic:** Find all registrations where status: 'PENDING_CONFIRMATION' AND confirmation_expires_at < NOW().
    - For each expired registration:
        1. Delete it.
        2. UPDATE courses SET waitlist_current = waitlist_current - 1.
        3. **Crucially:** Trigger the "De-registration Logic" again (as if a student just dropped), which will find the *next* person on the waitlist and offer them the spot.

Module 7: Notifications & Polish (Week 7)

- **Goal:** Integrate email and polish the UI.
- **Email Service:**
    - Integrate Nodemailer with a service like SendGrid or Resend.
    - Create email templates for:
        1. Successful Registration
        2. Joined Waitlist
        3. **Action Required: Your Waitlist Spot is Ready!** (This email contains the POST /api/waitlist/accept/:registrationId link).
- **UI Polish:**
    - Add loading spinners (react-spinners) or skeletons to all pages that fetch data.
    - Improve error handling: all API calls should have .catch() blocks that show user-friendly error messages via the toast library.
    - Write the project README.md.

Module 8: Testing & Deployment

- **Goal:** A production-ready, test-proven application.
- **Testing:**
    - **Manual E2E:** Test *all* user flows.
    - **Key Test:** The Race Condition. Open two browsers, set a course to 1 seat. Click "Register" on both at the same time. One must succeed, the other must be waitlisted or fail.
    - **Key Test:** The Waitlist.
        1. Student A registers for a full course (joins waitlist).
        2. Student B (in the course) drops it.

3. Student A receives the email.
4. Wait 24+ hours (by changing the DB value) and see the spot go to the next person.

- **Deployment:**
  - **Frontend:** Vercel or Netlify.
  - **Backend:** Railway or Heroku.
  - **Database:** Supabase or Neon (serverless Postgres providers).

3. Full-Scope Database Schema (Prisma)

This is the complete 4-table schema from your original PDF, ready for Prisma.

```
// This is your full-scope Prisma schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id           String        @id @default(uuid())
  email        String        @unique
  password_hash String
  name         String
  reg_number   String?       @unique
  department   String?
  role         UserRole      @default(STUDENT)
  registrations Registration[]

  @@map("users")
}

enum UserRole {
  STUDENT
  ADMIN
}

model Course {
  id             String   @id @default(uuid())
```

```
  course_code       String   @unique
  title             String
  description       String?
  department        String?
  professor         String?
  total_seats       Int
  remaining_seats   Int
  waitlist_capacity Int
  waitlist_current  Int      @default(0)
  credits           Int

  schedules     Schedule[]
  registrations Registration[]

  @@map("courses")
}

// Separate table for schedules, as a course can have multiple
model Schedule {
  id          String    @id @default(uuid())
  day_of_week DayOfWeek
  start_time  String    // e.g., "10:00"
  end_time    String    // e.g., "11:00"

  course   Course @relation(fields: [course_id], references: [id], onDelete: Cascade)
  course_id String

  @@map("schedules")
}

model Registration {
  id                    String             @id @default(uuid())
  registered_at         DateTime           @default(now())
  status                RegistrationStatus
  confirmation_expires_at DateTime?         // For waitlist timer

  user     User   @relation(fields: [user_id], references: [id], onDelete: Cascade)
  user_id  String
  course   Course @relation(fields: [course_id], references: [id], onDelete: Cascade)
  course_id String

  @@unique([user_id, course_id])
  @@map("registrations")
```

```
}

enum RegistrationStatus {
  REGISTERED
  WAITLISTED
  PENDING_CONFIRMATION // The user has 24h to accept
}

enum DayOfWeek {
  MONDAY
  TUESDAY
  WEDNESDAY
  THURSDAY
  FRIDAY
  SATURDAY
  SUNDAY
}
```