# DocsUniverse Rebuild: Architecture and Development Plan

## Introduction

DocsUniverse is a comprehensive platform for medical professionals and students, offering directories, educational content, quizzes, events, job listings, AI-driven tools, and research support services. The goal is to rebuild this platform using PHP (CodeIgniter framework) with a MySQL database, deployed on Replit. We will implement all previous system features across modules including:

* **Doctors & Hospitals Directory:** Searchable listings and detailed profiles for doctors and hospitals.
* **Courses & Masterclasses:** Online courses with enrollments and certificates; and one-time workshop events with booking and payments.
* **Quizzes:** Timed, competitive quizzes (Kahoot-style) with real-time leaderboards, score tracking, and certificate issuance.
* **Jobs Board:** Medical job postings and search functionality across India.
* **AI Tools:** A suite of tools (Diagnostic Helper, Medical Statistics Calculator, Literature Search AI, Clinical Note Generator, etc.) accessible based on user role or subscription.
* **Research Services:** Request system for services like article writing, thesis support, statistical consulting, and printing.
* **Communication & Automation:** WhatsApp-based OTP authentication, a WhatsApp chatbot for user support and service routing, and automatic PDF generation (e.g., certificates or letters) delivered via WhatsApp.
* **E-commerce Integration:** Payment gateway (Razorpay/Stripe) and a unified cart for purchasing quizzes, masterclasses, or services.

This plan details the system architecture, database schema, module-wise design, third-party integrations, front-end/back-end technologies, admin panel usage, deployment steps on Replit, and a data migration strategy to ensure all existing profile data (doctor fields, etc.) carry over exactly into the new system.

## Overall Architecture

The platform will follow a **Model-View-Controller (MVC)** architecture using CodeIgniter. CodeIgniter’s MVC pattern cleanly separates application logic from presentation, which helps in organizing our modules and enabling maintainable code. Key aspects of the architecture include:

* **Backend Framework:** **CodeIgniter 4** (latest version) on PHP 8, providing a lightweight MVC framework with libraries for database access, form validation, session management, etc. CodeIgniter’s structure will be used to create distinct controllers and models for each module (Directory, Courses, Quizzes, etc.), and views for rendering the UI. *For example, a DoctorsController will handle doctor directory requests, using a DoctorModel for DB queries, and views for listing/search and profile pages.*
* **Database:** **MySQL** will serve as the relational database, structured to capture all data for the modules. We will use CodeIgniter’s database library and Query Builder for interactions (which helps prevent SQL injection and simplifies queries). The schema will be normalized with foreign keys to maintain data integrity. All user-generated or dynamic content (profiles, course info, quiz questions, etc.) will be stored in MySQL. Replit deployment will require running a MySQL service (or using an embedded database if needed for development). On Replit, we might use a MySQL installation via Nix or an external MySQL instance for persistent storage, configured in CodeIgniter’s database config.
* **Frontend:** Primarily server-rendered HTML through CodeIgniter views with standard Bootstrap (for responsiveness) and minimal JavaScript/jQuery for interactivity. This keeps the front-end simple and compatible with Replit’s environment. We will progressively enhance interactive parts: for example, quizzes (real-time leaderboard updates) or dynamic filters may use AJAX calls or a lightweight front-end framework (Vue.js/React) if needed and if supported by Replit. Since Replit can host static files, we will include CSS/JS assets within the project. If needed, a CDN-hosted Vue/React could be used for certain pages (like a quiz live view) to handle dynamic UI updates, but the core platform will not heavily depend on Node.js services due to Replit constraints.
* **Third-Party Integrations Architecture:** External APIs will be integrated through dedicated service classes or libraries within CodeIgniter:
* **WhatsApp API (BigTos):** Used for OTP authentication, chatbot, and sending notifications or PDFs. We will create a service class (e.g., WhatsAppService) encapsulating BigTos API HTTP calls (for sending messages/files). Incoming webhook calls from BigTos (for chatbot messages) will be routed to a controller endpoint (e.g., WebhookController) which processes the message and responds via the API.
* **Payment Gateway:** Either Razorpay or Stripe will be integrated for secure payments. A PaymentController will handle creating payment orders and verifying callbacks. We will use the official PHP SDKs (via Composer) – e.g., Razorpay’s SDK for creating orders and verifying signatures, as shown in community guides. This ensures secure transaction processing and order verification.
* **PDF Generation (mPDF):** We will integrate the mPDF library (via Composer) to generate PDFs from HTML templates. mPDF allows creating PDF files from HTML content – we’ll design templates for certificates and letters using profile data, then generate and send them via WhatsApp.
* **Security & Performance:** All critical interactions will include validation and adhere to production-grade security practices. CodeIgniter’s built-in security features (XSS filtering, Query Builder to prevent SQL injection, CSRF tokens on forms, etc.) will be utilized. User passwords (if any) will be hashed; however, since we use OTP for login, password storage might be minimal. Sensitive API keys (BigTos, Razorpay/Stripe secrets) will be kept out of code (using Replit’s secret management or environment variables). We’ll enforce role-based access control in controllers to restrict module access (e.g., only admins can access admin panel endpoints, only logged-in doctors can edit their profile, etc.). For performance, appropriate indexing in the database will be done (e.g., indexes on doctor name, hospital, etc., for fast searching). We may also implement caching on frequently read data (like directory listings or course lists) using CodeIgniter’s caching if needed. Given Replit’s limits, we will test and optimize any heavy operations (for instance, processing large PDF generation or external API calls) to keep response times reasonable.
* **Deployment Environment (Replit):** The application will run in Replit’s container with an Apache or PHP development server. Replit’s always-on feature (if available) should be enabled to keep the web service running. We will configure the Replit environment with a start script to launch the PHP server (and MySQL if running locally). Static files (images, CSS, JS) will be served from CodeIgniter’s public (or assets) directory. Logging will be directed to files (within Replit’s filesystem) for debugging. Replit’s collaborative nature can be used for development, but for production usage, we note that a more robust hosting might be needed; still, the plan ensures the platform can be demonstrated on Replit fully.

Below, we dive into each major module’s design and how they fit into this architecture, followed by database schema details and specific development considerations.

## Database Schema and Data Model

The MySQL database will be designed to capture all required data while preserving all existing fields from the old system. We will use a separate table for each main entity and relationship tables where needed. All tables will use appropriate data types (e.g., INT for IDs, VARCHAR for strings, TEXT for longer text like descriptions, etc.), with primary keys on ID columns and foreign keys to maintain referential integrity. Indexes will be added on frequently searched fields (like doctor name, hospital city, etc.). The **key tables and their schema** are outlined below:

### 1. Users and Roles

**Users Table (users):** Stores all user accounts (doctors, students, service providers, admin). Common fields:  
- user\_id (INT, primary key, auto-increment)  
- name (VARCHAR) – Full name (for doctors, prefix “Dr.” can be part of this or stored separately)  
- phone (VARCHAR) – Phone number (used for WhatsApp OTP login)  
- email (VARCHAR) – Email address (optional if using phone primarily, but we’ll include it for communication or alternate login)  
- role (ENUM or VARCHAR) – User role, e.g., **Admin**, **Doctor**, **Student**, **ServiceProvider**, **Guest**. (Guests won’t have accounts; the role field distinguishes the others. Alternatively, we could have separate boolean flags or a mapping table for roles if a user can have multiple roles. In this plan, one primary role per account is assumed for simplicity.)  
- password\_hash (VARCHAR) – Password hash (only used if we allow email/password login for some cases; for OTP, this may be unused or store a dummy since OTP bypasses password).  
- whatsapp\_opt\_in (TINYINT) – Flag if user agreed to WhatsApp communications, to comply with messaging rules.  
- created\_at, updated\_at (TIMESTAMP) – Audit timestamps.

**Roles & Permissions:** Rather than a separate roles table, roles are defined in code (as above). The application will enforce permissions:  
- **Admin** – Full access to all modules and admin interfaces.  
- **Doctor** – Can manage their own profile, view directories, take courses/quizzes, request services.  
- **Student** – Similar to Doctor but without being listed in doctor directory (for medical students who use courses/quizzes; they might not have a public profile).  
- **ServiceProvider** – Can respond to service requests and possibly post jobs. They may have a profile visible in a service providers list (or we might reuse the doctor profile fields for them if applicable, or have a simpler profile).  
- (Guests have no login; they can only browse limited info.)

A **session management** table will be used by CodeIgniter if using database-backed sessions (especially if we use CI’s Cart library or want to persist sessions). CodeIgniter’s Session library can use a ci\_sessions table to store session data (primary key session\_id, blob data, etc.) – we will configure that if needed for reliability on Replit (since file-based sessions might reset on container restart).

### 2. Doctors & Hospitals Directory

This module involves two main entities: **Doctor profiles** and **Hospital entries**, plus relationships and supporting data (e.g., specialties list). All existing profile fields from the old system will be included to ensure a complete data migration.

**Doctors Table (doctors):** Stores doctor profile information, covering all **General, Professional, Posting, Job, Education, and Social** details as in the previous system. Each doctor is also a user (with login credentials in users table), so we link them via user\_id. Key fields (grouped by category):

* *General Details:*
* doctor\_id (INT, PK) – Unique doctor profile ID (also acts as foreign key to users.user\_id if we use the same value, or we have a separate field user\_id to link).
* user\_id (INT, FK to users) – The associated user account for login.
* full\_name (VARCHAR) – Doctor’s name (including prefix).
* gender (VARCHAR or ENUM) – Gender.
* birth\_date (DATE) or age (INT) – Date of birth or age.
* photo (VARCHAR) – File path or URL to profile photo. (We will store uploaded photos in a directory and save the filename here. Replit can store files, but space is limited; for production we might integrate cloud storage for photos.)
* bio (TEXT) – A short biography or summary.
* city, state, country (VARCHAR) – Location details of the doctor (could be home location or practice location; we will clarify that with “posting” below).
* contact\_phone and contact\_email can be stored, but since they likely duplicate users.phone/email, we may use the data from users. We might still keep them here if the old system had separate fields, just to preserve data structure (possibly for multiple contact numbers).
* *Professional Details:*
* specialization (VARCHAR) – Primary specialty (e.g., Cardiology, Dermatology).
* sub\_specialization (VARCHAR) – Secondary specialty or area of focus (if any).
* medical\_registration\_no (VARCHAR) – Medical Council registration number or license number.
* years\_experience (INT) – Years of experience in practice.
* languages\_spoken (VARCHAR) – Languages the doctor can communicate in (if captured).
* affiliations (TEXT) – Professional affiliations or memberships (e.g., member of medical associations).
* *Posting Details:* (Current employment info)
* current\_position (VARCHAR) – The doctor’s current job title/position (e.g., "Senior Consultant").
* current\_hospital\_id (INT, FK to hospitals) – Reference to the hospital/clinic where they currently practice. If the old system stored just a name, we’ll migrate into a hospital entry and link it.
* department (VARCHAR) – Department in the hospital (if applicable, e.g., "Cardiology Department").
* start\_year (YEAR) – Year they started this posting (if available).
* *(If doctors could have multiple current postings, we might allow linking to multiple hospitals via a join table doctor\_hospital, but typically one primary workplace is listed. We will assume one current hospital per doctor for now and use the hospital table for details.)*
* *Job Details:* (Preferences for job opportunities, if the doctor is seeking a job)
* seeking\_job (TINYINT or BOOLEAN) – Whether the doctor is currently seeking job opportunities.
* preferred\_location (VARCHAR) – Preferred job location(s) if seeking (city/region).
* preferred\_role (VARCHAR) – Type of role they are looking for (e.g., "Professor", "Surgeon in Hospital", etc.).
* expected\_salary (VARCHAR or INT) – Expected salary or salary range (as text or number).
* notice\_period (VARCHAR) – If employed, how soon they could join a new job.
* *Note:* These fields allow the doctor directory to double as a candidate database for recruiters if that was the intention. If the old system did not have such details explicitly, we will include them as an extension to support the Jobs module integration. They can be left null if not used by some profiles.
* *Education Details:*  
  We will create a separate table for education entries to allow multiple qualifications per doctor (unless the old system limited to a set number of degrees). The table doctor\_education will have:
* education\_id (INT, PK), doctor\_id (FK to doctors),
* degree (VARCHAR) – e.g., MBBS, MD, MS, PhD, etc.
* institution (VARCHAR) – Name of the college/university or institution.
* completion\_year (YEAR) – Year of graduation.
* additional\_info (VARCHAR) – Any additional details (e.g., honors).  
  If the previous system instead had fixed fields (like separate columns for UG, PG, etc.), we will map those into multiple entries here. For migration, we will ensure that all degrees listed in the old profile (including any postdoctoral or diploma) are captured as rows in this table.
* *Social Details:*
* facebook\_url (VARCHAR), linkedin\_url (VARCHAR), twitter\_handle (VARCHAR), instagram\_handle (VARCHAR) – URLs or handles for the doctor’s social media profiles (if they were collected).
* website\_url (VARCHAR) – Personal or clinic website if provided.
* youtube\_channel (VARCHAR) – If the doctor had a YouTube or other channel (just in case it was in old system screenshots).
* social\_other (VARCHAR) – Any other social link not covered (could also store a link to their publications or Google Scholar if relevant).

All the above fields ensure **full fidelity with the previous system’s doctor profile**. In the migration phase, we will map each legacy field to these new columns one-to-one, so no information is lost. The doctors table might seem wide, but it simplifies queries (most fields are one-to-one with the profile). The doctor\_education table handles one-to-many education records. If needed, we could also create a doctor\_experience table for past postings/employment history if the old system captured previous jobs, but since categories mentioned only one “Posting” section (likely current), we’ll omit detailed past job history unless required.

**Hospitals Table (hospitals):** Stores hospital/clinic information for the directory. Fields include:  
- hospital\_id (INT, PK)  
- name (VARCHAR) – Hospital or clinic name.  
- address (TEXT) – Full address. We could break it into address\_line, city, state, pincode, etc., for better filtering by location. At minimum store city and state separately for search filters.  
- city, state, country (VARCHAR) – Location components (may be duplicated in address but stored separately for filtering).  
- contact\_phone, contact\_email (VARCHAR) – Contact info for the hospital administration.  
- specialties (TEXT) – Comma-separated or a linked list of specialties available at the hospital (e.g., "Cardiology, Neurology, Pediatrics"). We might normalize this by having a separate table for specialties and a relationship table hospital\_specialty, but a text field is simpler if we don’t need to filter hospitals by specialty often. For better design, we can have a specialties master table and a join table.  
- facilities (TEXT) – Description of facilities (e.g., "24x7 Emergency, Pharmacy, ICU").  
- website\_url (VARCHAR) – Hospital website if any.  
- logo (VARCHAR) – Path to a logo image, if provided.  
- description (TEXT) – General description or overview of the hospital.

We will also maintain a **specialties reference list**. A simple table specialties (specialty\_id, name) can list all medical specialties. Doctors can link to one or multiple specialties (for multiple specialties, we could have a join table doctor\_specialty if needed, but since we have one primary specialization field, we may not need this unless we allow tagging multiple specialties per doctor beyond the one or two fields). Hospitals too can reference specialties offered. However, given our fields above, we might parse the text or use the list for suggestions. This is a design choice for search optimization: we may allow filtering doctors by specialty easily if each doctor has one specialty field. If doctors can have multiple specialties, a join table approach would be better. For now, assume one primary specialty per doctor for simplicity (with possibly a secondary in sub\_specialization text).

**Relationships:** The doctors table references hospitals via current\_hospital\_id. We will enforce that link with a foreign key. If a doctor isn’t associated with a hospital (e.g., a freelance consultant or fresh graduate), that field can be NULL. For cases where a doctor has multiple affiliations (visiting consultant at multiple clinics), we might list the primary one in the profile and mention others in the bio or a notes field. (Extending to multiple affiliations would require a separate mapping table.)

### 3. Courses Module

The courses system requires storing course information, content structure, and enrollment records.

**Courses Table (courses):** Each entry represents an online course. Key fields:  
- course\_id (INT, PK)  
- title (VARCHAR) – Course title.  
- description (TEXT) – Overview or summary of the course.  
- content (LONGTEXT) – Course content or syllabus details. This could include HTML content if courses are mostly text/video descriptions. In the previous platform (which looked like it used a WordPress LMS), courses had multiple lessons and quizzes. For our rebuild, we can simplify by treating the course as a single unit with maybe an outline. Alternatively, we can create separate tables for course sections/lessons: e.g., course\_modules table: (module\_id, course\_id, title, content, order). This would allow structuring a course into lessons or chapters. We’ll include that if we expect multi-part courses.  
- duration (VARCHAR) – Duration of course (e.g., "4 weeks", "30 days", or number of hours). This can be a textual field since it might be descriptive.  
- fee (DECIMAL) – Course fee (0 if free). All courses appear to be free in the old system snippet, but we want to allow paid courses in the future.  
- category (VARCHAR) – Category of the course (e.g., “Lab Medicine”, “General Health”). In the old platform, courses were categorized. We might have a separate course\_categories table (category\_id, name) and a foreign key here. For simplicity, we can store category name or ID in this field.  
- level (VARCHAR) – Skill level (e.g., "Beginner", "Advanced") if relevant.  
- certificate\_template (VARCHAR) – (Optional) path to a certificate template file or identifier if each course has a custom certificate design. Alternatively, certificates can be generated on the fly without storing a template reference, but if a course-specific certificate format is needed, we store it.  
- created\_by (INT, FK to users) – Which admin or instructor created the course.  
- created\_at, updated\_at (TIMESTAMP).

**Course Modules/Lessons (optional, course\_modules):** If we break down content:  
- module\_id (INT, PK), course\_id (FK), module\_title, module\_content (TEXT or link to video, etc.), position (INT for ordering).  
This allows courses to have multiple lessons. Quizzes could also be associated per lesson or at end, but we handle quizzes via the quiz module separately (with possibly a link from course to a quiz as an assessment).

**Enrollments (course\_enrollments):** Tracks which users are enrolled in which course. Fields:  
- enrollment\_id (INT, PK)  
- course\_id (FK to courses)  
- user\_id (FK to users – the student/doctor who enrolled)  
- enrolled\_on (DATETIME)  
- progress (TINYINT or DECIMAL) – Course completion percentage. (We could update this as the user goes through modules, or just mark completed vs not.)  
- completed (TINYINT) – flag or maybe a completion date field if completed.  
- certificate\_sent (TINYINT) – whether a certificate has been issued upon completion. If certificate is instant upon completion, this might not be needed, but useful if we want to ensure we only send once. Possibly store certificate\_pdf\_path if we save the generated certificate.

With these, the system can list courses, allow users to enroll (inserting a row), and track completion. Admin or instructors can create courses via an admin UI that populates the courses (and modules) table.

### 4. Quizzes Module

The quiz system is a bit complex due to the competitive, timed nature. We’ll design tables for quizzes, questions, options, and results. We also account for live quiz sessions if needed:

**Quizzes Table (quizzes):** Each quiz or quiz event. Fields:  
- quiz\_id (INT, PK)  
- title (VARCHAR) – Quiz title (e.g., “Pharmacology Challenge”).  
- description (TEXT) – Short info about the quiz (could include rules, etc.).  
- duration (INT) – Total duration in seconds (if it’s a fixed-time quiz for all questions together). However, Kahoot-style quizzes usually have per-question timer rather than one overall timer. We might not need this if each question has a timer.  
- is\_live (TINYINT) – Flag if this quiz is a live competitive event (where all players participate at the same time) or a static quiz that one can take anytime. For competitive quizzes, we will have additional fields for scheduling.  
- scheduled\_start (DATETIME, nullable) – If it’s a live event, the scheduled start time. Participants can join by that time.  
- status (VARCHAR) – Status of quiz: e.g., "upcoming", "active", "completed". During the quiz active phase, the system will accept answers.  
- certificate\_template (VARCHAR) – (Optional) if a specific certificate format for winners/participants. If using a generic template, not needed per quiz.  
- created\_by (INT, FK to users) – The admin or instructor who set up the quiz.

**Quiz Questions (quiz\_questions):** Stores each question in a quiz. Fields:  
- question\_id (INT, PK)  
- quiz\_id (FK to quizzes)  
- question\_text (TEXT) – The question text.  
- question\_type (VARCHAR) – e.g., "MCQ", "TrueFalse", etc. For our purposes likely all multiple-choice single answer. We can extend to others if needed (like multi-select, short answer).  
- time\_limit (INT) – Time limit in seconds for this question (e.g., 30 seconds). If not set, could default to a standard (like Kahoot often uses ~20s).  
- points (INT) – Points for correct answer (could be static per question or vary). Kahoot awards points based on answer speed; we might emulate that by computing score dynamically rather than storing points per question. But we can set a base points (like 1000) and then adjust for time taken.

**Quiz Options (quiz\_options):** Stores the answer choices for MCQ questions. Fields:  
- option\_id (INT, PK)  
- question\_id (FK to quiz\_questions)  
- option\_text (TEXT) – The text of the option.  
- is\_correct (TINYINT) – True/false if this is the correct answer. (For MCQ single-answer questions, only one option will be marked correct. If we had multi-select, multiple could be correct, but initial scope is single answer.)

**Participant Answers (quiz\_answers):** This table will capture answers given by users in a live quiz (or in a static quiz attempt). Fields:  
- answer\_id (INT, PK)  
- quiz\_id (FK)  
- question\_id (FK)  
- user\_id (FK to users)  
- selected\_option\_id (FK to quiz\_options) – The option the user chose.  
- is\_correct (TINYINT) – Marked true if the answer was correct (this can be derived by comparing to correct option, but storing for quick querying).  
- answer\_time (FLOAT) – The time (in seconds) the user took to answer, from the moment the question was shown. This is crucial for scoring in competitive quizzes. We can capture this via client-side timing or server timing if each question submission records a timestamp and we know question show time.  
- answered\_at (DATETIME) – Timestamp when answer was submitted (for record).

**Quiz Results / Scores (quiz\_results):** This table summarizes the outcome for each user per quiz (especially for competitive quizzes). Fields:  
- result\_id (INT, PK)  
- quiz\_id (FK)  
- user\_id (FK)  
- score (INT) – Total score obtained.  
- correct\_count (INT) – Number of correct answers.  
- wrong\_count (INT) – Number of wrong answers (or skipped).  
- time\_taken (INT) – Total time taken (if needed). For live quizzes, this might not be as relevant since everyone has similar time for each question.  
- rank (INT) – The rank of the user in that quiz (1 for winner). This can be computed after the quiz is completed and stored.  
- completed\_at (DATETIME).

We will populate quiz\_results at the end of a quiz run (for a live quiz, once all questions are done, compute scores and ranks). For non-live quizzes (like practice quizzes), we can compute the result immediately after the user finishes and store it, so they can see their score and maybe compare on a leaderboard if we have one for that quiz.

**Quiz Sessions (if needed, quiz\_sessions):** If we want to allow repeated instances of the same quiz (like hosting the same quiz questions at multiple times), we could have a session concept. For now, each quiz entry can represent one event or one set of questions. We might not need a separate session table unless we want to allow quizzes to be taken anytime individually *and* have special live events. In a simpler approach, we can designate a quiz as either live (with a fixed start) or static. If static, any user can take it anytime and they’ll get a result but it’s not a competition. If live, users join at scheduled time. This can be managed via the fields in quizzes (like scheduled\_start and status). So, we might not require a sessions table unless an advanced scenario. We will proceed without it for now.

### 5. Masterclasses Module

Masterclasses are essentially events (workshops or webinars). The data model is similar to courses but oriented around a scheduled event with possibly limited seats and payments.

**Masterclasses Table (masterclasses):** Fields:  
- class\_id (INT, PK)  
- title (VARCHAR) – Masterclass title (e.g., “Advanced Surgical Techniques Workshop”).  
- description (TEXT) – Details about what the class covers.  
- instructor (VARCHAR or INT FK to users) – Name of instructor(s). If instructors are also users (like doctors or external experts), we could link to the user/doctor table. In many cases, the instructor might be a doctor from the system, so storing their user\_id is useful to link to their profile. If multiple instructors, store as text or have a join table; but likely one primary instructor.  
- date (DATE) and time (TIME) or a combined datetime for the schedule. Also possibly end\_time if duration is known.  
- location (VARCHAR) – Location of the event (could be a venue address or “Online” for virtual). If online, we might store a meeting link in this field or a separate field.  
- capacity (INT) – Max number of attendees (if limited seats).  
- fee (DECIMAL) – Registration fee (0 if free).  
- duration (VARCHAR) – Duration (e.g., "2 hours" or "3 days course"). Could be derived from start/end but good to state.  
- certificate\_provided (TINYINT) – Whether attendees get a certificate.  
- created\_by (INT, FK to users) – Admin who created the event.  
- created\_at, updated\_at.

**Registrations (masterclass\_registrations):** To track users who booked the masterclass. Fields:  
- registration\_id (INT, PK)  
- class\_id (FK to masterclasses)  
- user\_id (FK to users) – The attendee.  
- registered\_at (DATETIME)  
- payment\_status (VARCHAR) – e.g., "paid", "pending", "free". If integrated with payments, mark accordingly.  
- attendance (TINYINT) – Mark if the user actually attended (this could be updated after the event by admin, if needed for certificate issuance).  
- certificate\_sent (TINYINT) – if certificate delivered.

This allows listing attendees and ensuring we don’t exceed capacity (by counting records for a class).

### 6. Jobs Module

The job board requires storing job postings and possibly applications.

**Jobs Table (jobs):** Fields:  
- job\_id (INT, PK)  
- title (VARCHAR) – Job title (e.g., "Senior Resident Doctor – Pediatrics").  
- description (TEXT) – Full description of the job, responsibilities, etc.  
- location (VARCHAR) – Job location (city or specific hospital name if known). Possibly split into city/state for filtering.  
- organization (VARCHAR) – The hiring organization (hospital/clinic name or company). If it corresponds to an entry in hospitals, we might link to hospitals with a foreign key if they are in our system; otherwise, just text.  
- specialization (VARCHAR) – Desired specialization (if the job is specialty-specific, e.g., "Orthopedic Surgeon"). We can utilize the same specialties list.  
- job\_type (VARCHAR) – e.g., "Full-time", "Part-time", "Contract".  
- experience\_required (VARCHAR) – e.g., "3+ years" (could also be numeric field).  
- salary\_range (VARCHAR) – e.g., "₹100k-₹150k/month" or leave as per industry standards.  
- posted\_by (INT, FK to users) – The user who posted (likely an **Admin** or a **ServiceProvider** role if we allow certain recruiters to post). Possibly doctors could post if they are hiring for their clinic, but that’s less likely; we assume admin control or designated recruiters.  
- posted\_at (DATETIME)  
- status (VARCHAR) – e.g., "Open", "Filled", "Closed". (Allows admin to close a job when position is filled.)  
- contact\_info (VARCHAR) – Contact details for application (like an email or phone or link to apply if external). This is important if we are not handling applications internally; we should display how to apply.

**Job Applications (optional, job\_applications):** If we want users to apply through the platform:  
- application\_id (INT, PK)  
- job\_id (FK)  
- user\_id (FK to users) – applicant (must be logged in, presumably a doctor or student).  
- applied\_at (DATETIME)  
- resume (VARCHAR) – path to an uploaded resume/CV file, if we allow that.  
- status (VARCHAR) – e.g., "Applied", "Under Review", "Accepted", "Rejected".

However, implementing a full application process might be beyond initial scope. We can choose to simply direct users to contact the poster. But having this table allows tracking if needed. We’ll include it for completeness, noting that it can be activated if we integrate an “Apply” button.

### 7. AI Tools Module

The AI tools (Diagnostic Helper, Stats Calculator, Literature Search, Note Generator) mostly perform computations or external API calls on the fly, so they may not need extensive database storage. The main DB use could be: logging usage or storing templates for the output.

Potential tables:

* ai\_tool\_usage: log of tool usage with fields (log\_id, user\_id, tool\_name, input\_text, output\_text, used\_at) if we want to keep a record of what queries are made (for monitoring or billing if needed). This is optional.
* ai\_tool\_permissions: if we implement subscription or role-based access control to tools, we might have a table linking user\_id to allowed tool or a subscription plan. Alternatively, simply use the role and maybe a flag in the users table (e.g., a field premium\_until date or subscription\_level). Since not specified in detail, we assume roles define access (e.g., maybe only doctors can use certain diagnostic tools, or only paid users can use the AI note generator extensively). We will implement role checks in code rather than in DB for now, with the flexibility to expand.

Other than that, each tool’s logic will be coded in PHP (or via calling external services). For example, the **Diagnostic Helper** might use a predefined knowledge base or algorithm (like decision trees coded or an API to an AI model). The **Statistics Calculator** will use formulas (no DB needed). The **Literature Search AI** might call an external API (e.g., a medical articles database or an AI that finds articles). The **Clinical Note Generator** could call an AI like GPT-4 via API if available, or use templates – if we had a table of templates for various notes, we could store those, but more likely an AI call. In absence of external AI initially, we might implement a rule-based placeholder or require an internet connection which on Replit might not be persistent. For planning, we assume integration with an AI API (which would not involve our DB except storing an API key config).

So, for DB schema, AI Tools do not mandate new tables except optional logs. We will manage their availability via roles in the code.

### 8. Research Services Module

This involves a catalog of services and a workflow for service requests and fulfillment.

**Services List (services):** A table listing types of research/consulting services offered. Fields:  
- service\_id (INT, PK)  
- name (VARCHAR) – e.g., "Thesis Writing Assistance", "Statistical Analysis", "Journal Publishing Support", "Poster Printing".  
- description (TEXT) – Details about the service scope.  
- base\_price (DECIMAL) – A base price or starting cost (if applicable). Some services might be custom-quoted, so this could be null or indicative (e.g., price per page).  
- turnaround\_time (VARCHAR) – e.g., "2 weeks" (expected fulfillment time).  
- active (TINYINT) – to enable/disable a service offering.

We populate this table with all offerings (from the question: articles, thesis support, stats consultation, printing, etc.). This allows dynamic listing and future expansion.

**Service Providers (service\_providers):** If service providers are a subset of users (role = ServiceProvider), we can store additional info about them here. Fields:  
- provider\_id (INT, PK, also FK to users.user\_id if 1-1 mapping)  
- name (from users table or duplicate for convenience)  
- services\_offered (maybe a text or a join table mapping providers to services they handle). A join table provider\_service (provider\_id, service\_id) could explicitly link which services each provider can fulfill. This way when a request comes in for a particular service, we find providers with that service.  
- region (VARCHAR) – area or location the provider covers (if relevant, e.g., printing might be region-specific for physical delivery).  
- whatsapp\_number (if different from user phone, but likely same)  
- availability (VARCHAR) – maybe schedule or a simple flag if they are currently available to accept new requests.

Alternatively, if the service providers are basically our internal team or partners, we might manage assignments manually via admin rather than an open marketplace. But the description says "gets routed to providers via WhatsApp", implying possibly multiple providers get a notification and one can take it.

We will allow multiple providers for a service and build a routing logic.

**Service Requests (service\_requests):** Each user request for a service. Fields:  
- request\_id (INT, PK)  
- user\_id (FK to users) – the requester (doctor or student who needs the service)  
- service\_id (FK to services) – type of service requested  
- details (TEXT) – additional details provided by user (description of what they need)  
- document\_path (VARCHAR) – file path of any document the user uploaded (e.g., research manuscript, data file)  
- location (VARCHAR) – location tag (could be address or city for where service is needed, e.g., for printing service, the delivery location; or for statistical consult, maybe not needed). We might use user’s city by default if not explicitly provided, but the form will allow specifying if needed.  
- requested\_at (DATETIME)  
- status (VARCHAR) – e.g., "Pending", "In Progress", "Completed", "Cancelled". Initially "Pending" when user submits.  
- assigned\_to (INT, FK to service\_providers or users) – which provider picked it up. Null if not yet assigned.  
- assigned\_at (DATETIME) – when it was assigned.  
- completed\_at (DATETIME) – when finished.  
- remarks (TEXT) – any notes from provider or admin (like outcome, or if cancelled, reason).  
- payment\_status (VARCHAR) – If this service goes through payment, track if paid ("paid","unpaid"). If using the Cart, likely the request will be created either after payment or before payment depending on flow; we can mark it accordingly.  
- price\_charged (DECIMAL) – final price charged for the service (could be determined case by case, especially if custom, the provider or admin might update this, but if user paid upfront a fixed price, it matches that).

**Service Routing logic:** When a new request is created (user submits form), the system will find **eligible providers**: those whose services\_offered include the requested service, and possibly match region if applicable. We might also consider load (who is free). We then use the WhatsApp API to **notify these providers**. For example, send a WhatsApp message to each with details: “New DocsUniverse service request: [Service Type] from [User Name] in [Location]. Reply 1 to accept.” We can include a link to an internal page to view details (if providers have login to the site). The first provider to respond (either by clicking a link or by replying via WhatsApp if we set up that interaction) will get assigned. Alternatively, the message could instruct to respond in the chat; the chatbot can catch an “accept” message and then mark that provider as assigned in the service\_requests record. After assignment, a confirmation can be sent to the user (e.g., “Your request has been taken up by [Provider]. They will contact you soon.”). The other providers can get a notification that it's taken (or we simply stop when one accepts). This requires coordination with the chatbot logic (the chatbot for providers or a separate flow). We will handle it in the Chatbot section, but the data model supports an assigned provider and tracks status.

If no provider accepts in a certain time, the request remains pending and an admin can intervene (maybe manually assign or follow up). We might implement a simple timeout logic to re-notify or escalate to admin via email/WhatsApp.

### 9. Integrations & Communication

Some data aspects related to integrations:

**OTP Verification:** We will not store OTP codes permanently in the DB for security; instead, when a user attempts login, we generate a code, store it in a temporary store (like PHP session or a transient table) and send via WhatsApp. For example, a table otp\_codes with (phone, code, expires\_at) could be used to verify the code input. Alternatively, store in session so it’s tied to that attempt. We’ll implement whichever is simpler; a table is useful if we want to track OTP usage or have multiple servers. For now, a transient approach is fine.

**Chatbot Data:** The chatbot is mainly logic. We might store some predefined Q&A or flow in a table if the flows are complex. However, since BigTos can use a spreadsheet script or our webhook logic, we might not need a DB table for chatbot dialogues. We will implement the logic in code (like a state machine per user if needed, stored in session or a small table if conversation context must persist). For example, if the chatbot is guiding a user through a service request, we might store a temporary record or use the existing service\_requests as soon as they provide initial info. This is a bit design heavy; an easier route is to have the chatbot mostly give information and links, and for structured things (like creating a service request) direct them to the web form. However, the description implies even service routing via the chatbot. This could mean a user can initiate a service request by chatting, and the bot will create the entry and route it. We should account for that scenario. Possibly a chatbot\_sessions table to track where each user (by phone) is in a conversation flow (like expecting which input next) – that might be overkill, but if the chatbot flows are multi-step (like "send me your document now"), it’s needed. For brevity, assume we handle context in memory or via simple cues (BigTos might manage some of it since they allow script templates). The core persistent data from chatbot interactions will end up in our other tables (e.g., a service request, or a quiz registration if the bot allowed that, etc.).

**Payment Records:** A table payments could be maintained to log each transaction: (payment\_id, user\_id, order\_id, amount, status, method, transaction\_date, etc.). However, Razorpay/Stripe will have their own records; we should store at least the essential info to correlate orders. Actually, we will have an **Order and Cart system** (next section) which inherently generates payments, so we integrate it there.

### 10. Cart and Order System

To unify purchases (quizzes, masterclasses, services), we introduce an **Order** model.

**Orders Table (orders):** Represents a checkout instance. Fields:  
- order\_id (INT, PK)  
- user\_id (FK to users) – who is purchasing  
- total\_amount (DECIMAL)  
- payment\_status (VARCHAR) – "pending", "paid", "failed", etc.  
- payment\_method (VARCHAR) – e.g., "Razorpay", "Stripe", or "COD/Manual" if any offline (likely not).  
- transaction\_id (VARCHAR) – the payment gateway transaction reference (like Razorpay payment ID or Stripe charge ID) for record.  
- created\_at, paid\_at (timestamps).

**Order Items (order\_items):** Each record links an order to a specific item (quiz, class, service). Fields:  
- order\_item\_id (INT, PK)  
- order\_id (FK to orders)  
- item\_type (VARCHAR) – e.g., "Quiz", "Masterclass", "Service". We can standardize these strings or use an enum.  
- item\_id (INT) – the ID of the item in its table (e.g., quiz\_id, class\_id, or service\_id corresponding to the type).  
- item\_name (VARCHAR) – store the name/title at purchase time (for quick reference and to avoid joins when listing order summary).  
- price (DECIMAL) – price charged for this item at checkout.

When a user adds something to cart, we actually create a cart in session. On checkout, we create an order record (with payment\_status "pending") and corresponding order\_items for each cart entry. Then user is directed to payment (Razorpay/Stripe). After payment, the success callback (or webhook) will update payment\_status to "paid" and fill transaction\_id. Then we can mark the related items as purchased: for each order\_item: - If item\_type = "Quiz": we mark that user as enrolled/registered for that quiz (this could simply be adding them to a participants list if the quiz is a scheduled event). Since for quizzes we didn’t define a participants table, we can either create one or just treat payment as granting access. For a live quiz event, we might require registration: we can consider that if a quiz is paid, only those who paid can participate. We can enforce by checking order\_items for that quiz and user when they try to join. Alternatively, maintain a table quiz\_registrations(user\_id, quiz\_id). Let’s do that for clarity: - **Quiz Registrations (quiz\_registrations):** (user\_id, quiz\_id, registered\_at). If a quiz requires signup (free or paid), adding a row here either on payment or on free join. For free quizzes, clicking "join" can just add an entry. For paid, it’s added after payment success. This table then is used to know who is permitted to play in a live quiz event. If a quiz is not live (open to all anytime), we may not need pre-registration; the act of taking it can itself log a result. But for consistency, we can still log it when they start. - If item\_type = "Masterclass": we insert a row in masterclass\_registrations for that user/class (with paid status).  
- If item\_type = "Service": we update the corresponding service\_request to mark it paid (and possibly only create the request entry after payment if we prefer). There are two approaches: - *Pre-pay then request:* Add to cart -> pay -> on success, create service\_request. - *Request then pay:* Create service\_request (status maybe "Awaiting Payment") -> pay -> then mark as "Pending" (fully requested). We might choose the latter for immediate routing after payment. For planning, we can say the service request is created only after successful payment (for services that have upfront fixed fees). For services that require custom quote, maybe they won’t be in cart at all; the user would submit request, get a quote, then pay offline or a custom link. That’s a complex scenario; perhaps all services have predefined fees to simplify (e.g., “get a literature search – ₹500”). We will assume services are pre-defined items user can pay for directly.

This order system ensures a unified checkout. We can utilize CodeIgniter’s session or cart library to manage the cart in the user’s session. The CI3 had a cart class for session storage of items, which allows adding items and quantities. If using CI4 or if the cart library is deprecated, implementing our own via session is straightforward (store an array of item data in $\_SESSION). The concept remains the same: *“The Cart Class permits items to be added to a session that stays active while a user is browsing your site…”*. We will collect item type and id when the user clicks "Add to Cart", save it in session, and build a simple cart page listing those items and total.

Finally, note the **Payment Integration**: When user checks out, we create the order and order\_items, then initiate payment. For Razorpay, this involves creating an Order via their API (with amount, etc.) and obtaining an Order ID, then rendering a checkout form (with Razorpay’s JS) that uses our Key ID. On payment completion, Razorpay will return razorpay\_payment\_id and a signature. Our PaymentController will verify the signature (using Razorpay SDK’s verify method) and then update our DB. For Stripe, it would be a similar approach using Checkout Sessions or Payment Intents. The plan will detail Razorpay as it’s local and likely what the user expects. Razorpay’s PHP SDK will be included via Composer, and we’ll follow their integration steps (API keys in config, create order, verify signature).

Now that the data model is defined, we proceed to describe each module’s functional design and how the front-end/back-end will work together.

## Module Implementation Details

### Doctors & Hospitals Directory

This module provides a searchable directory of doctors and hospitals, plus profile pages.

**Front-End:** We will create user-facing pages for: - **Doctor Search/List:** A page with filters (e.g., by name, specialty, location, hospital, etc.). A simple form on top allows entering a name or selecting a specialty from a dropdown (populated from the specialties list). Results will be shown in a list of doctor cards or rows, showing name, specialty, location, and maybe photo thumbnail. The listing can page through results if many. We will implement this by a DoctorsController@index method that fetches doctors from DB based on filters (using WHERE conditions for each provided filter) and passes to a view. We can enhance user experience with instant search (AJAX) for filters, but initially a form submission is fine (with CodeIgniter handling input parsing and query).  
- **Doctor Profile Page:** Clicking a doctor opens their profile view (DoctorsController@profile($id)). This page will display all the doctor’s details in a structured way. We will organize it into sections (General, Professional, etc.) possibly as tabs or accordions for clarity. For example: - *General:* Name, photo, gender, age, location, contacts. - *Professional:* Specialty, experience, registration number, languages, affiliations. - *Current Posting:* Position, hospital name (with link to hospital page), department, since when. - *Education:* List of degrees with institution and year (iterating through doctor\_education records). - *Social Links:* Icons linking to their social profiles (if available). - Possibly a “Send Message” or “Request collaboration” button if such feature is considered (not specified, so likely not now).  
The profile page can be viewed by any logged-in user (and possibly even guests for general info, unless we restrict to logged in). Editing the profile is only for the doctor themselves or admin, and would be a separate form (see Admin below). - **Hospital Directory:** Similar pattern: - A listing page for hospitals, with filters by location or name. - Hospital profile page showing address, contact, facilities, specialties, and possibly a list of doctors at that hospital (we can query doctors where current\_hospital\_id = this hospital). This provides a nice link between modules, letting users see who practices there.

**Back-End:** - We will create DoctorModel and HospitalModel for queries. The DoctorModel might join with related tables to get everything needed for profile display (like joining hospital to get the name, or using separate queries for education list). - For search, we’ll use DoctorModel->search($criteria) to build a query with LIKE for name, equals for specialty, etc., and return results. We’ll ensure to only pick active doctors (we can have a field for doctor profile approval or active status if needed). - The HospitalModel similarly will have search and a method to get detail (with possibly all doctors in that hospital if needed). - **Admin/Management:** Admins will have an interface to manage the directory: - A page to list all doctor profiles, with options to edit or remove any, and to approve new ones. If we allow doctors to sign up and fill their profile, initially they might be flagged unapproved until admin verifies. So a field is\_approved could be added to doctors. Approved doctors appear public. - An admin (or the doctor themselves) can use a **Profile Edit** form. This form will likely be long, covering all the fields. We will perhaps break it into steps or tabs to make it manageable (General info, Professional info, etc.). The fields in the form correspond directly to the doctors table columns and the education repeating group. We’ll implement multi-row education input either via JavaScript (add/remove degree fields) or a simpler approach (limit number of degrees fields). A dynamic JS solution is nicer: the form could allow adding multiple education entries, and on submit, we handle inserting them into doctor\_education. - Similarly, a hospital management form for admin to add/edit hospitals (or maybe we can let doctors suggest a hospital if not in list when editing their profile, then admin approves it). - **User Role Logic:** A doctor user can view and edit *their own* profile (through a “My Profile” dashboard section). A service provider or student would not have a doctor profile, so their “My Profile” might be limited to basic account info. - We will ensure that profile editing by a doctor only updates their own record (using session user\_id match) and possibly requires admin review if needed. Admins can edit any profile directly. - **Integration:** If we implement any rating or review system for doctors (not mentioned, so we skip), that would add complexity. We’ll assume it’s a directory without reviews for now. - **Search optimization:** For instant search, we could use jQuery to call an API endpoint that returns matching doctors in JSON, but given time we can initially do form submit. Replit can handle those AJAX calls though if needed.

By implementing this, users (or guests) can find doctors and hospitals easily, and doctors have rich profiles including all previously available information.

### Courses Module

The Courses module will function as a mini Learning Management System (LMS) for online courses.

**Front-End:** - **Course Catalog:** A page listing all courses (or categorized). We can show course cards with title, category, duration, and a “View Details” or “Enroll” button. If many courses, we allow filtering by category or search by keyword. - **Course Detail Page:** Shows the full course information – title, description, a syllabus or content outline (if we have multiple modules, list them), instructor name (if applicable), duration, etc. If the user is not enrolled, it will show an **Enroll button** (or “Buy Now” if fee > 0, which would add to cart). If the course is free, clicking enroll can directly create an enrollment entry. If paid, clicking enroll adds course to cart for checkout. If the user is already enrolled, the page will show their progress and provide access to the course content (e.g., a “Start/Continue Course” link). - **Course Content Access:** For simplicity, we might have a single page that displays all course content (like a long scroll or sections on one page). But better is to have a lesson navigation: if using course\_modules table, we can show a list of lessons. Clicking a lesson could show its content. Since CodeIgniter can load views for each lesson, we might create a controller method like CoursesController@learn($course\_id, $module\_id) that checks if user is enrolled, then displays the module content. We can include “Mark as complete” buttons or auto-mark when they reach the end. We might not implement video or quiz embedding deeply, but if needed, we can embed YouTube videos or provide downloadable materials in content. - **Course Assessments:** Not explicitly asked, but the mention of quizzes in courses in the old platform suggests courses had quizzes. We have a separate Quizzes module which could be linked. We can integrate by allowing a course to specify a quiz as its final test (perhaps add a field quiz\_id in courses if each course has one final quiz). If so, on course completion page we could link to “Take Quiz” which actually uses the Quizzes module. This integration can be considered later; initially we can handle quizzes independently, but the plan should note the possibility.

* **Certificate Issuance:** If a course offers a certificate on completion, once the user finishes all modules and passes any required quiz, we trigger certificate generation. The certificate could include the user’s name, course title, completion date, and perhaps a unique certificate ID. We’ll use mPDF to generate a PDF certificate. The certificate template (maybe an HTML with appropriate images/logo) will be filled with user name and course info, then Mpdf->Output() to produce PDF. We then mark in course\_enrollments that certificate issued, and deliver it. Delivery can be: allow user to download from site, and/or send via email or WhatsApp. Given the heavy WhatsApp integration, we will send it to the user’s WhatsApp through BigTos API automatically upon completion. This means making an API call with the PDF file attached (BigTos likely provides an endpoint to send media).

**Back-End:** - **Course Creation (Admin):** Admin users can create and manage courses via an admin panel. The admin UI will have a form to input all course details and upload any resources. For content, we might use a rich text editor (like TinyMCE) for the description or content to allow formatting. If courses have multiple lessons, the admin UI may allow adding multiple modules: e.g., after creating course, an interface to add lessons one by one. Alternatively, the admin might just compile the entire course content in one place for now (depending on complexity). - **Admin Example:** *For instance, an admin creates a course “Basic Life Support (BLS) Certification”. They fill in duration (1 week), category (Health), and description. They add modules: Module 1 “Introduction to BLS” (text content), Module 2 “BLS Techniques” (maybe a video link), Module 3 “BLS Quiz” (we could attach a quiz ID if a quiz is used).* In the admin panel, this could be done through a multi-step form or separate screens for course and modules. Our plan is to have a straightforward approach, possibly requiring some manual linking (like after saving course, note its ID, then go to a separate “Add Module” form). - **Enrollment Process:** When a user clicks Enroll (and the course is free), the CoursesController@enroll($course\_id) will: - Check if user logged in; if not, prompt login (since progress tracking requires an account). - If logged in, create a row in course\_enrollments with status not completed. - Redirect the user to the course content page (or back to detail with a message “Enrolled!” and show the “Start Course” link). If the course is paid, then the Enroll button triggers the cart flow (i.e., goes to CartController@add(item\_type=Course, item\_id)). Payment then upon completion calls a function to finalize enrollment as described earlier. - **Tracking Progress:** Each time a user views a module, we could mark progress. The simplest is to mark completed when they click a "Mark Complete" button at end of module, and count modules done. If modules count = total modules, mark course completed. Or we can even auto-complete if they open all pages. This could update course\_enrollments.progress. If a final quiz is required, we might only set completed after passing the quiz. That logic can be added (e.g., a field pass\_required in courses or just check if quiz exists then ensure quiz\_results has passing score). - **Technology Note:** CodeIgniter will serve these pages. For any interactive front-end (like a video or slides), it’s mostly embedding content. We may use a bit of JavaScript for things like toggling module list or marking progress via AJAX (nice to have but not required). - **Example Content Prompt (for admin panel):** - *Course Creation:* Admin enters: **Title:** *“Comprehensive Cardiology Course”*; **Category:** *“Health”*; **Duration:** *“6 Weeks”*; **Fee:** *“₹500”*; **Description:** *“An in-depth course covering cardiology fundamentals, diagnostics, and patient management. Includes weekly quizzes and assignments.”*; **Instructor:** *“Dr. A. Sharma”*. They then add modules: 1. Module 1: *“Anatomy of the Heart”* – content could be a text and images. 2. Module 2: *“ECG Interpretation”* – content text plus maybe a link to an external video. 3. ... etc. - *Quiz Addition:* If the course has quizzes, admin might either link an existing quiz or create questions directly. In our plan, quizzes are managed in Quiz module, so admin would go to Quiz section to create a quiz titled "Cardiology Course Final Quiz" and then link its ID into the course’s record (we could have a field or just mention in description with instructions). This example shows how the admin uses the panel to populate content that students will see.

By implementing courses in this manner, we provide a structure for delivering educational content with tracking and certificates, matching the previous system’s learning component.

### Quizzes Module

The Quizzes module enables competitive quizzes similar to Kahoot, as well as potentially standalone quizzes for practice. This is one of the more interactive modules, involving real-time features.

**Admin Interface for Quizzes:** - Admin (or a designated “Quiz Master”) can create quizzes in the admin panel. This involves inputting the quiz details and questions. We will design a form where the admin enters: - Quiz Title, Description, optional scheduled date/time if it’s a live event. - A field for total questions (or they can dynamically add questions on the form). - For each question: text, options, and mark the correct option. Also set the time limit per question (default can be 20 seconds, but admin can override), and points scheme if needed. Possibly we have a global scoring algorithm but we can let admin set points per question if desired (e.g., harder questions could weigh more). - We might provide a question entry UI where admin can click “Add Question” repeatedly. Each question fieldset has a textbox for question and inputs for 4 options plus a checkbox/radio to mark correct. This input can be made user-friendly with JavaScript, but even without, we could have a simpler approach (like instructing admin to enter options separated by newline and indicate correct answer with a prefix, etc. – but a structured form is better). - The admin can also specify if the quiz is open to everyone or only certain roles (maybe some quizzes might be premium content or targeted to doctors only). This could be a field for “eligible\_roles” or we just note to announce accordingly. - Once created, the quiz can be launched if scheduled. There might be an admin control panel for the live quiz where the admin can start the quiz at the given time and monitor.

**Quiz Participation (User side):** - **Registration:** If a quiz is a scheduled live event (e.g., a competition happening on a specific date/time), users should register beforehand (especially if there’s a cap or it’s paid). We handle that via the cart or quiz\_registrations as described. If the quiz is free and open, registration could be as simple as a "Join" button that adds to quiz\_registrations. For a live quiz, joining at event time is also possible, but better to have pre-registered list. We will allow last-minute join by going to the quiz page when it’s about to start. - **Joining the Quiz Session:** We’ll create a Quiz Play page (perhaps QuizController@play($quiz\_id)). This page will have logic to dynamically load questions at the right time and display the leaderboard. - If the quiz is static (non-live), the play page can simply list questions one after another (or one per page) with a submit at end (like a normal online quiz). But since the requirement is Kahoot-like, we focus on live mode. - For live mode: Initially, if the quiz hasn’t started, the page can show a "Quiz will begin at HH:MM, please wait..." and possibly show a lobby or instructions. We could implement a simple countdown timer to start. - When the quiz starts (admin triggers it or time reached), question 1 is presented. We want all participants to see it simultaneously. Achieving real-time sync in PHP alone is tricky; we can use a workaround: the client page will poll the server every few seconds to check quiz status (what question number is active now, how much time left). We might maintain a field current\_question and question\_start\_time in the quizzes table (or a separate state somewhere, maybe even in cache). When admin starts quiz and moves to next questions, we update those. Alternatively, we implement through periodic AJAX calls: - The page calls an endpoint like QuizController@status($quiz\_id) which returns JSON { current\_question\_no, time\_left, show\_leaderboard\_flag, etc. }. - If current\_question\_no changes (meaning a new question has started since last check), the front-end then fetches that question (or we include question data in the status). - Users then see the question and answer choices. We show a timer countdown (which can be client-driven based on time\_left). - When user selects an answer, we send it via AJAX to a submit endpoint (or form submit if we refresh per question, but AJAX is smoother). - After time is up (or after all have answered or a fixed duration passes), we move to next question. Possibly show correct answer and maybe partial leaderboard. - **Real-time Leaderboard:** After each question, or at least at the end, we want a leaderboard. We can calculate scores incrementally: - Scoring logic: Each correct answer gives points. Kahoot gives more points for faster answers. We can emulate: for example, base points 1000 for correct, and subtract some amount per second of delay. If time limit is T, and user answered in t seconds, score = 1000 \* ((T - t) / T) perhaps, plus a bonus for streaks if we want. To keep it simpler, maybe allocate between 500 to 1000 points depending on speed. We'll implement a formula in code. - We update each user’s score in quiz\_results (or accumulate in memory and finalize later; but updating DB after each question is fine for persistence). - The leaderboard can be displayed by querying quiz\_results ordered by score at that point. The page can either refresh or an AJAX call to fetch current top scores after each question. - For visual effect, we could do an auto-refresh or push updates. Without a true push (websocket), frequent polling (like every 2 seconds) for the leaderboard might be okay for a small number of players. On Replit, we should be mindful, but we assume manageable scale. - After final question, final standings are shown. The quiz\_results.rank can be computed or simply determined when displaying (ranking by score, tie-break by total time or earlier finishing if needed). - We then congratulate the winners. If we are issuing a certificate, we generate certificates for say top 3 or all participants (depending on requirement – likely a certificate of achievement for winners and/or participation for others). - These certificates are then sent automatically via WhatsApp (BigTos API) to their registered phone. We have to loop through winners or participants: \* If doing only winners: e.g., generate a “Winner Certificate – [Quiz Name] – 1st Place” for rank 1, etc. Or a general certificate of participation for everyone with their score. The problem statement specifically says "certificate delivery via Bigtos API", implying at least winners get something. We'll implement sending a PDF to the top performers (maybe rank 1,2,3) and a participation e-certificate to others if desired.

* **Static Quiz Mode:** Not asked, but if quizzes are also used for courses or practice:
* The QuizController@play can detect if quiz is not live (is\_live=0). Then it can just render all questions or one by one without waiting logic. The user can complete at their own pace, and at the end see their score. The quiz\_results would store their score and we could still issue a certificate of completion if relevant or just show score.
* Leaderboard in that case could be a static high score list for that quiz if we want to gamify, but not as critical.

**Integration with WhatsApp for Quizzes:** - We could optionally send notifications like “Quiz starting now” or reminders via WhatsApp to registered participants (using BigTos, schedule a message say 10 minutes before). - After quiz, as mentioned, send out certificates or at least a message "Thanks for participating, your score: X". - If the chatbot is also to be used for quizzes: one might imagine participants could answer via WhatsApp instead of web? However, the plan likely expects them to use the web interface. There is mention of chatbot for premium replies but not specifically for quiz answers, so we assume web interface for quizzes.

**Example (Quiz Admin and Execution):** - Admin creates a quiz “Monthly Medical Quiz – October” with 5 questions, each 20 seconds. For question 1, “What is the normal adult heart rate range (beats per minute)?”, options: A) 60-100 **(correct)**, B) 100-140, C) 40-60, D) 120-160. Admin sets it as a live quiz scheduled for Oct 30, 6:00 PM. - On Oct 30, 5:55 PM, participants join the quiz lobby page. At 6:00, admin clicks “Start Quiz” in admin interface (or it auto starts). Participants’ pages refresh (via script) to show Q1 and a 20-second timer. - Users select an answer; the system records their choice and time. After 20s, Q1 closes. The correct answer is revealed on their screen (“Correct answer: 60-100 bpm”). The system updates scores (those who chose A get up to 1000 points depending on response time; wrong answers 0 points). - Next, Q2 begins (similar process). After Q5, the quiz ends and a leaderboard shows the top scorers: e.g., Dr. X – 4200 pts, Dr. Y – 3700 pts, etc. - The system then generates a winner certificate for Dr. X (1st place) and possibly runner-up certificates for Y, Z, etc., using a predefined template that includes quiz title and their rank. These PDFs are sent to their WhatsApp via the API automatically. Additionally, a participation certificate or message might be sent to others if desired (this can be configured). - All results are stored; users can later see their past quiz performances on their profile (result history page that pulls from quiz\_results).

This way, the quiz module provides both the excitement of live competition and a record of achievements.

### Masterclasses Module

Masterclasses are handled in a manner akin to events with booking:

**Front-End:** - **Masterclass List:** A page (or section on homepage) that shows upcoming masterclasses, possibly with date and a “Register” button. We might show them sorted by date. If the list is long or includes past ones, filters by date or specialty could help. The listing can include title, date/time, instructor, fee. - **Masterclass Detail Page:** Contains detailed info about the workshop: agenda, instructor bios, venue if applicable, etc. Also includes the registration form or link. If fee > 0, a “Register (₹X)” button that goes to cart. If free or already paid, either allow direct register. If logged in, clicking register will either mark them registered (free) or proceed to payment. - We should also show how many seats left if capacity is set. For example, “25/30 seats filled” if we track that (we can count registrations and subtract from capacity). - After registering, the user should get a confirmation. Possibly an email or WhatsApp message “You have successfully registered for [Masterclass].” containing details or next steps. We will likely send via WhatsApp using BigTos, including maybe a calendar reminder or link. - Closer to the event, reminders can be sent (this might be done manually by admin or scheduled; if automated, we’d need a cron or scheduled script – might be beyond scope but mention as possibility). - If the masterclass is online, the detail page (and possibly post-registration message) should provide the meeting link or instructions. If offline, provide address.

**Back-End:** - **Creation/Management:** Admin can create masterclasses in the admin panel, similar to courses but simpler: input title, description, date/time, instructor (could be chosen from doctors list or typed if external), location, capacity, fee. Possibly upload an image or poster. Once created, it appears in listing. - Admin should be able to view the list of registered users and their status. In the admin panel, clicking on a masterclass might show the attendee list. Admin can mark attendance or add notes, or cancel the event if needed (setting status to cancelled which could hide it and notify users). - **Registration Flow:** - If fee: via cart as described – user adds to cart and pays, then the system creates a masterclass\_registration entry (with paid status). - If free: on clicking register, we directly create masterclass\_registration (with payment\_status “free”). Possibly also trigger a WhatsApp confirmation message. - We will ensure capacity: before creating registration, check how many already registered; if at capacity, disallow and inform user (or allow waitlist if we implement that, but probably just show “full”). - After registration, if immediate confirmation is needed, we can generate a ticket or confirmation PDF. But simpler: send a WhatsApp message "You have a seat reserved." If an actual ticket is needed (with QR code?), that's complex, we skip; likely not needed for now. A PDF certificate after attending could be possible if they want to give a certificate of completion for attending. If so, after the event, admin can mark attendance and then trigger a certificate similar to courses. - **Integration:** Payment (if paid class) already handled by order system. WhatsApp: sending reminder or updates can be done via our WhatsAppService. E.g., a day before, a cron job (if available) could send “Reminder: Masterclass [X] tomorrow at 10 AM.” If Replit can't run cron, admin may have to do it manually or on event start. This is an enhancement beyond core, but we mention in plan to consider manual or scheduled communications. - **Post-event:** If providing certificates for attendance, we could use mPDF to create an attendance certificate for each attendee (similar approach to courses) and send via email/WhatsApp. The question does not explicitly mention masterclass certificates, but since courses and quizzes have, maybe yes. We can include as optional.

### Jobs Module

The job board allows posting and searching for medical jobs.

**Front-End:** - **Job Listings:** A page listing all jobs, with filter controls for location, specialty, and maybe keyword. We can list job title, organization, location, and a snippet of description. If many, allow pagination. - **Job Detail Page:** Shows full description, requirements, etc., and details on how to apply. If we implement internal applications, it might show an “Apply Now” button (if logged in). Otherwise, it might list contact info like “Email your CV to hr@hospital.com” or “Apply via [Link]”. - **Job Posting Form (if user posting allowed):** If we allow certain roles to post jobs (for example, a hospital admin or service provider logged in might post a vacancy), we’d have a form similar to admin’s. But the prompt doesn’t explicitly mention a separate role for recruiters, just says “Post and find medical jobs”. Possibly they intend any doctor or service provider can post a job (like if a clinic is hiring they can post on platform). Alternatively, it might be only internal (admin team collects jobs and posts). - We can allow logged in users with a certain permission to access a “Post Job” page. They fill job details, which goes to jobs table and possibly requires admin approval to go live (so a field approved can be used). Admin can review and approve in admin panel. - **Applying to Jobs:** If an “Apply” button is present: - If clicking apply is meant to send an application through the platform, we will show a form for the applicant to maybe write a short message and confirm applying. We could allow attaching a resume (file upload). - On submission, we create a job\_applications entry. Possibly also email the job poster with the applicant’s info (or notify via WhatsApp if the poster has a WhatsApp number). - The applicant could later see their applied jobs list in their dashboard (with statuses if updated). - The poster or admin could see the applications and mark status (we may need an interface for that). - However, implementing the full application tracking might be beyond initial scope. We can start with simply listing jobs and providing contact info or a link to apply externally. In our plan, we can mention the possibility of in-platform applications and design for it, even if maybe phase 2.

**Back-End:** - **Job Management (Admin):** Admin can add/edit jobs through the admin panel. If users can post, admin should approve as mentioned. - We might have an admin list of all jobs with the ability to edit details or deactivate/close a job. - If internal applications are used, admin might also manage those (forward to relevant person or just oversee). - **Search Implementation:** The job listing page will use queries that filter by: - Specialty: either matching jobs.specialization or we could allow free text search across title/description for keywords. - Location: match city or state in the location field. - Possibly filter by job type or experience required. CodeIgniter’s Query Builder will help compose these conditions easily. - **Security:** Only authorized users (admin or allowed roles) can create jobs. Everyone can view/search jobs. If applications are allowed, only logged-in users can apply.

### AI Tools Module

This module includes various AI-powered utilities for users. Each tool will have its own interface and possibly use external APIs or internal algorithms.

We have the following tools listed: 1. **Diagnostic Helper** – likely for inputting symptoms or lab results to get a suggested diagnosis or differential diagnoses. 2. **Medical Statistics Calculator** – calculators for statistical tests or epidemiological metrics (e.g., sample size calculator, sensitivity/specificity, etc., or could be simpler like BMI, BSA calculators). 3. **Literature Search AI** – a tool to query medical literature (maybe an AI that summarizes PubMed results). 4. **Clinical Note Generator** – helps generate clinical notes or reports from given inputs (like take some bullet points and generate a narrative).

For each tool, the implementation will involve: - **UI:** A form where user enters relevant input data. - Diagnostic Helper: perhaps a form with fields for key symptoms or a text describing the case. - Stats Calculator: maybe multiple calculators; we could make a submenu. For example, within Stats Calculator, options like “Sample Size Calc”, “Chi-square Test”, etc. Each opens a small form for those inputs. - Literature Search: a search box to enter keywords or a question. The tool might then call an external service to fetch relevant articles (for example, using NCBI’s E-utilities API to get PubMed articles, then maybe summarizing via an AI if possible). Or if using an AI like GPT trained on literature, call that. - Note Generator: a form where the user enters some key patient info (like name, age, symptoms, findings) and selects type of note (e.g., referral letter, discharge summary) and then the tool outputs a drafted note.

* **Backend logic:**
* Some tools might be implemented completely offline:
  + The Statistics Calculator can be done in PHP using formulas. For example, if it includes a BMI calculator, we just do weight/height², etc. Or if it's more advanced stats (like t-test, etc.), we can either code formulas or use a library. Since CodeIgniter can use any PHP library, if a library for statistical calculations exists, we might include it. But probably simpler: implement select formulas manually.
* Others likely need external AI:
  + Diagnostic Helper and Clinical Note Generator suggest the use of machine learning/NLP. Possibly the expectation is to integrate something like OpenAI’s API. For the plan, we can mention using AI APIs. E.g., use OpenAI or a medical-specific model to analyze the input. Since implementing an AI model from scratch is not feasible, integration is the way. We must note this requires API keys and costs, so perhaps it’s optional or limited to premium users (fits with “Users can access based on roles/subscription”).
  + Literature Search AI can fetch results from an API. For example, we could use the E-utilities API of PubMed (which returns article titles/abstracts for a query). We could then use an AI to summarize them or just list top results. Alternatively, connect to an AI like GPT-4 and prompt it with the user’s query asking for relevant literature references (but that might not ensure accuracy unless properly done).
* For plan’s sake: propose using external APIs for these:
  + The system sends the user’s query to the appropriate API and returns the response to the front-end.
  + We have to ensure we restrict usage if needed (maybe only logged in doctors can use certain heavy tools like Note Generator, to avoid abuse of API).
  + Possibly add usage limits: e.g., a free user can do X queries per day, whereas premium (if we had a premium subscription concept) can do more. This detail was hinted with “based on roles/subscription”. Without a full subscription system described, we might interpret that as: doctors and students (free roles) can use some basic tools (like the calculators), but the advanced AI ones (like Note Generator or maybe Literature AI) require an upgrade or admin-only. For now, maybe we can say only logged in users of role Doctor or Student can use them (guests cannot), and if needed, we can easily add a flag if later they do subscription tiers.
* **Example workflow for one tool (Clinical Note Generator):**
* UI form asks: Patient Name, Age, Gender, Diagnosis, Treatment Plan, any specific instructions. The user fills these in and hits “Generate Note”.
* Backend (NoteGeneratorController@generate) forms a prompt to an AI service, e.g.: *“Generate a concise clinical note for a patient with the following details: [details].”* Calls the API (like OpenAI) with this prompt.
* Gets back a generated note text. The controller returns this to the view, which displays it in a textarea or a styled div. The user can copy it. Possibly provide a “Regenerate” or “Edit” option.
* (We won’t store the notes unless we want to keep a copy; maybe not necessary).
* We will caution that AI output should be reviewed by the user (to cover responsibility).
* **No heavy DB needed** as explained; just ensure the relevant config (API keys) and code.
* **Admin and Development:** We may need to store API keys (OpenAI, etc.) in config or environment. If using BigTos AI features, maybe they offer some AI integration behind the scenes, but likely not – BigTos seems more about messaging. So we’ll plan to use common APIs.
* If the AI tools require significant computational resources, on Replit it might be slow if calling external or impossible if no internet – but Replit likely allows outgoing requests for APIs. We assume yes, as else AI tools wouldn’t function.
* Possibly the "AI Tools" section in screenshots might have been placeholders, but we'll plan as if implementing basic functionality for each.

**Access Control:** - We’ll ensure these pages check user role. For example, wrap the tool routes with something like:

if (! in\_array($this->session->user\_role, ['Doctor','Student','Admin'])) {  
 show\_error("You do not have access to this tool.");  
}

Or redirect to login. Possibly service providers might not need these, but could allow if desired. - If we had a premium membership concept, we could check a is\_premium field or subscription expiry date on user. That’s beyond given info, so skip unless we guess it might come later.

### Research Services Module

This module allows users to request various research-related services and have those requests processed.

**Front-End (User side):** - **Services Overview:** A page (e.g., “Research Services”) that lists all the available services (from the services table). Each service card shows the name, description, perhaps starting price. There will be a “Request Service” or “Details” button for each. - **Service Detail / Request Form:** Clicking a service leads to a page with full description and a form to request that service. The form typically includes: - Some fields specific to the service type. For example, if “Statistical Analysis Service”, the form might ask for brief description of project, type of analysis needed. If “Thesis Consultation”, maybe ask for topic, etc. To keep flexible, we’ll just have one details text area where user describes what they need. - File upload field (for user to attach any relevant document, e.g., their manuscript or data). We’ll allow common formats (doc, pdf, xls, etc.) and store on server (in a uploads/requests/ folder and path in DB). - Perhaps a desired completion date or urgency (could add a field urgency or include in details). - Location selection if relevant: For printing service, location is definitely relevant (where to ship prints). For other services like consulting, location might not matter as it can be online. However, the request form will always capture location (maybe auto-fill from user profile location, but allow change) because the request routing mentions location. - Submit button. If the service requires payment upfront (meaning it has a set price), the button might say “Add to Cart” instead, integrating with payment. If the service is consultative where price is not known upfront, the button could just say “Submit Request” and then the payment might be figured out later (like provider will give a quote). - For simplicity, if services.base\_price is not null, we assume that’s the fee. We then integrate with cart: i.e., if base\_price > 0, we could route them to checkout after submitting details. But details need to be captured too. Possibly a two-step: collect details, create a pending request, then add to cart to pay. We have to ensure the request and payment link. - We might treat each service like a product. Another approach: use the cart purely for payment and tie to a request via order item. E.g., user fills request form, on submit: \* Create service\_request with status "Awaiting Payment". \* Redirect to checkout with service item in cart. \* On payment success, update request to "Pending" (meaning confirmed). \* If payment fails or user aborts, we can cancel the request after some time or leave as draft. - If base\_price is 0 (free service? unlikely) or if the service is quote-based, then we simply mark request submitted without payment and perhaps show a message "We will contact you with a quote". - **After Submission:** The user might see a confirmation page like “Your request has been submitted. Request ID #X. Our team will contact you soon via WhatsApp.” We will definitely leverage WhatsApp: - The system can send the user a confirmation message on WhatsApp: “Your [Service Name] request has been received. We will assign a provider shortly.” - Then as providers respond or the job gets assigned, the user can get notified: “Your request is taken by [Provider Name]. They will reach out to you. Track status at [link].” We can also allow the user to see status on the website: a “My Service Requests” page in their account, showing status updates. This page would show each request, its status, assigned provider name, and any messages or uploaded outputs (if, say, the provider uploads the completed document, we could deliver it through the platform or directly). - If we want to fully integrate communication, perhaps the provider might send the completed work via WhatsApp or email. But possibly, they might upload it to the platform (e.g., an edited thesis PDF) and then we notify the user to download from site or send via WhatsApp as well. We can allow providers to upload output files to the request record. This means the service\_requests table could have a field output\_path or we create a separate service\_request\_files table to handle multiple files (initial upload vs output).

**Back-End:** - **Admin/Provider Interface:** Service providers (role) and admins need to handle incoming requests: - We will create an admin view listing all new requests (Pending). Admin can also manually assign a provider if needed (a dropdown of providers). But if we trust auto-routing via WhatsApp, admin may not need to act unless no one picks it. - For providers, ideally they have a portal showing requests assigned to them (In Progress). They should be able to update status and possibly upload results or mark complete. - However, many providers might not log into the platform often, which is why WhatsApp routing is key – they can accept and maybe do everything via WhatsApp or email. But for completeness: \* We give each provider a login (as a user with ServiceProvider role). When they log in, they see a dashboard of requests: pending (to accept or reject?), assigned to them (to work on), and completed. \* They might have a button to mark a request complete and upload final files. \* They can also communicate with the requester. Instead of building chat in the app, we rely on WhatsApp for conversation. Possibly the provider will directly WhatsApp the user once they have their number from the request details (we should include user’s contact in the request details so provider can reach them). - Admin can oversee all requests, change assignment, and ensure quality.

* **Automation with WhatsApp (BigTos):**
* As described earlier, on a new request, the system (via a background process or immediate after submit) will fetch providers for that service and send them a standardized WhatsApp message. We will need to format that message maybe like:
* New Request: [Service Name]   
  Location: [City]   
  Details: [first 100 chars of user details]...   
  Reply with "Accept [RequestID]" to take this up.
* (The chatbot flow doc likely outlines something similar).
* When a provider replies "Accept 123" on WhatsApp, BigTos will send that to our webhook. We parse the message, find request 123, mark it assigned to that provider (if still unassigned), update status to In Progress, and respond via WhatsApp to provider: "You have been assigned Request 123. Please contact the user and deliver by [some expected date]." Also send a confirmation to the user: "Provider [Name] has accepted your request. Contact: [provider’s phone]."
* If a provider tries to accept but it's already taken, we inform them it's no longer available.
* If none accepts within a certain timeframe, admin might manually assign or contact more providers. We might not implement an auto-expire in code, but we can note that admin will keep an eye and follow up.
* Implementation: the chatbot logic to parse messages can be in WebhookController@receiveWhatsApp(). We will have to verify the incoming message is from an authorized provider number (we have their phone in user profile). BigTos likely includes sender info. So we check if sender phone corresponds to a service provider and if message says "Accept X". This is a bit of natural language processing but we can define a simple protocol: maybe instruct providers to just send the request ID or a specific keyword. The flow doc might clarify the exact expected reply format. We'll assume something straightforward like above.
* **Payment integration in workflow:** If a service was paid upfront, providers are assured to proceed. If a service is quote-based, maybe the provider after reviewing details will quote a price. In that scenario, we might need an iteration: provider could reply with a quote, which we forward to user for approval and payment. That’s a complex flow:
* Possibly “premium replies” might mean providers or an AI could respond with some info to user queries, but not sure. However, focusing on service requests: let’s say a user requests a complex analysis, provider says it will cost $X. We then need to collect payment. We could integrate by updating the request with price and sending user a payment link (via WhatsApp). The user clicks, pays (through our payment gateway), and then we mark it paid and provider is notified to proceed. This is something we can mention as a capability but might be phase 2 complexity. If the question wanted that detail, they probably would have mentioned quoting. Since they set up a cart system, I suspect they lean towards fixed prices at least for initial services, making it straightforward.
* We will proceed with assumption that each service request corresponds to an item that can be priced either fixed or at least decided at request time.
* **NPA and PDF Generator (specific to this module):** The problem statement specifically mentions **NPA and Service PDF Generator** using mPDF with user profile fields, and sending via WhatsApp. Possibly NPA refers to a particular document like a *Non-Disclosure Agreement* or *No Plagiarism Assurance* that the platform generates for the user. Maybe when a user requests a service, the system generates an agreement PDF (like a contract or privacy assurance) that is sent to them (or maybe to provider) automatically. Or NPA could be *No Objection Certificate (NOC)* for something. Given "based on user profile fields", perhaps it’s a personalized document like a membership certificate or an application form.
* "For Garry.pdf" likely was an example of such a generated PDF. It might contain the user’s name and some pre-defined text.
* Implementation: We will prepare HTML templates for these documents. e.g., an NPA template that says “This is to certify that Dr. [Name] ...” maybe a formal letter. Using mPDF, we fill in the user's details (from their profile, like name, qualification, etc.) into placeholders and output PDF.
* The trigger for generating these could be various:
  + Perhaps when a user completes their profile, they can get a PDF of their "Network Participation Agreement" (just guessing).
  + Or when they request a service, we issue a PDF invoice or summary.
  + The text says "NPA and Service PDF generator... send via WhatsApp." This suggests two separate things: an "NPA" PDF and various service-related PDFs. Possibly NPA is something like a certificate of joining the network (for doctors) and “Service PDF” could be maybe a summary of their service request or an invoice.
* Since it specifically says based on profile fields, NPA might be like a personalized certificate of some sort. We’ll mention generating such documents and sending them.
* The mechanism is straightforward: have a view file or HTML string, do replacements for placeholders (like {NAME}, {CITY}, {SPECIALTY}, etc.), instantiate Mpdf, and output file. We’ll then call WhatsAppService->sendPDF($user\_phone, $pdfPath, $caption) to deliver.
* We will integrate this into relevant workflows. E.g.:
  + After a doctor registers, generate an "Welcome certificate or agreement" that includes their details, and send to them.
  + After a service request is completed, generate a summary report or certificate of service delivered.
  + Optionally, after a masterclass or course, certificate which we already covered.

**Example:** A user requests "Thesis Proofreading". They fill the form with details and upload their thesis draft. They pay ₹500 and submit. The system creates request #45. Immediately, the system generates a PDF “Service Agreement” containing the user's name and request details (maybe terms that our team will maintain confidentiality, etc.), and sends it to the user on WhatsApp. It then notifies providers. Provider A accepts via WhatsApp. The user gets confirmation. After a week, provider marks complete and uploads the edited thesis. The user receives a WhatsApp message: "Your requested service is completed. Download: [link]" or we directly attach the file (if not too large) via WhatsApp. Also, we could generate a "Service Completion Certificate" for the user if appropriate (not sure if needed, but could). This shows a seamless workflow with automated documentation.

### User Roles & Authentication

The platform will enforce distinct user roles and an OTP-based authentication system.

**User Roles Recap & Permissions:** - **Admin:** Has access to an admin dashboard to manage all content (doctors, hospitals, courses, quizzes, masterclasses, jobs, services, AI settings, etc.). Admin can create/edit/delete any entity, approve user profiles and job posts, and view all transactions. Admin also can manually trigger things like sending broadcast messages or generating reports. We will create a protected /admin route group. Possibly using a simple check if($user->role != 'Admin') redirect('home') in admin controller constructor to restrict. - **Doctor:** A registered medical professional user. They have a public profile in the directory. They can edit their profile information, search the directory, enroll in courses, take quizzes, view masterclasses, apply to jobs, and request services. They have access to AI tools as per allowed. They likely are the main target for the platform’s content. - **Student:** A medical student or similar user. They might not be listed in the doctor directory (no public profile) but otherwise can use courses, quizzes, and perhaps apply to jobs (internships, etc.). They also can use AI tools. The UI and permissions for them are similar to doctor minus the public profile and maybe minus posting jobs (they wouldn’t post jobs). - **Service Provider:** A user who provides services or possibly also posts jobs. They typically would not appear in the doctor directory (unless a person is both). They have access to service requests assigned to them. They might also be given ability to post jobs (like a hiring manager role). In admin panel, we could let Admin designate some users as service providers. The provider’s account needs a profile too (not public, but for internal use like their name, expertise, etc.). We might use the same user table with role ServiceProvider and possibly tie to an entry in service\_providers table for their offerings. Providers can log in to see requests, update status. They might not use courses/quizzes unless we allow it (they could, but usually they might not be interested). - **Guest:** Not logged in visitor. They can browse public info (like doctor directory limited info, hospital info, list of courses maybe but not take them, view job listings but not apply, etc.). We will encourage them to register for full access. For example, trying to enroll in a course or take a quiz will prompt login. The directory might be visible to public, but maybe hide contact details of doctors from guests to protect data (just an idea; the previous platform might have required login to see details). - We might incorporate a “role selection” at signup. E.g., user chooses “Doctor” or “Student” or “Service Provider” or maybe we don’t allow random sign up as provider (providers likely are onboarded by admin). So maybe sign-up is open for Doctors and Students only. If a Service Provider wants to join, they might contact admin separately to get an account.

**WhatsApp OTP Authentication:** - Instead of traditional email/password, we implement login via OTP. When a user wants to log in or sign up, they will provide their phone number (WhatsApp number). - If new user (not registered), we will take additional info (like name, email optionally, role selection) in a registration form and then verify their phone via OTP to activate account. - If existing user, just entering phone triggers OTP flow to log in. - OTP flow: 1. User enters phone on login page and hits “Send OTP”. 2. Server generates a 6-digit random code (or use a secure random). Save it temporarily (session or DB) with an expiry (~5 minutes). 3. Use BigTos API to send a WhatsApp message to that number: “Your DocsUniverse OTP is 123456” (BigTos likely provides an endpoint for sending WhatsApp text; we’ll call it with phone and message). 4. User sees a prompt to enter the OTP. They input it, we verify against stored code. If match and within time, authentication succeeds. 5. If code invalid or expired, show error and allow retry sending OTP. - Post verification, if it’s a new user sign-up, we finalize creating their account in users (and if role=Doctor, also create an entry in doctors table for their profile, possibly marking it incomplete for them to fill out later). If existing, we simply log them in (set session variables). - We must handle duplicate phone gracefully: phone is essentially the username (unique). - For admin accounts, we might not want to use OTP each time (maybe allow them to have a password for convenience). But we can still support OTP for admin too if they log in with phone. Perhaps we’ll generate accounts for admin with a known password or provide a special login route. Given the focus, likely admin will also just use OTP with their phone. - **Alternate flows:** Some might still prefer email login. The prompt doesn’t mention email at all. Possibly the entire system pivoted to phone-based login via WhatsApp which is common in India. So we’ll stick to that as primary auth. We can keep an email for communication and as a backup, but not for login.

**User Registration:** - The registration form will ask: Name, Phone, Role (Doctor or Student), and perhaps Email, Password (maybe no password if OTP? Could skip password entirely). If Doctor role, we might also ask some basic info upfront (like specialization, but that might be too early; instead, after registering, they can fill profile). - After filling, we send OTP to phone for verification as above. Once verified, account is created and user is logged in. - For Service Provider accounts, we likely won’t have a public registration. Admin will create those accounts in admin panel (taking provider’s details and setting role). - We should also integrate a *WhatsApp-based onboarding via chatbot*: Maybe an alternative where a user sends a WhatsApp message to the bot number to sign up and it collects their info. The flow doc might mention something like user can send "Join DocsUniverse" and it will capture their name, etc. That would require writing more logic in the chatbot to create a user record. This is advanced and optional – but since they have a chatbot, maybe they want that. For now, focus on web onboarding and note chatbot could be extended to handle registration queries.

**Account Management:** - Once logged in, users can access a “Dashboard” or account page. For doctors, this includes profile editing, list of courses enrolled, quiz results, etc. For students similarly. For providers, their dashboard shows service requests. We will design these pages as needed for UX. - Also provide logout and perhaps an account settings page (change email or link an email, etc., though not critical with OTP login).

**Premium Content & Subscriptions:** - The question hints at subscriptions in context of AI tools. If in future they want to monetize, they could have a subscription product that grants access to some tools. We can design it such that: - There is a field is\_premium or subscription\_level in users. - Payment for subscription could be another thing to implement (maybe via Razorpay as a product or manual). Possibly out of current scope. - For now, we might interpret it simply as “some tools only for certain roles” – e.g., perhaps service providers or admin have some internal tool? Not likely. It’s probably premium as in paid subscription. - We will mention that the system can easily accommodate a subscription check if needed for those features (like check if user has premium flag before allowing Literature AI usage, etc.). - But without concrete requirement, just ensure roles control basic access (like AI Tools accessible only if logged in as Doctor/Student as decided).

### WhatsApp Chatbot Integration

The WhatsApp chatbot is a crucial interactive element. It will operate through the BigTos platform, which likely handles connecting a WhatsApp number and forwarding messages to our webhook.

**Objectives of Chatbot:** Based on “chatbot flow.docx” and description: - Route user inquiries to the appropriate service or information. It might act as a menu-driven bot. For example, a user on WhatsApp could send "Hi" to the DocsUniverse number, and the bot can reply with options like "1. Find a Doctor, 2. Upcoming Quizzes, 3. Request a Service, 4. Talk to Human...". - Provide **premium replies**: Possibly means if a query is beyond FAQs (like a complex medical question), the bot might either hand off to a human expert or use an AI to reply (if they've paid). This could tie into AI tools – maybe a user could ask "Can you summarize this article?" and if they have premium access, the bot might use the AI Literature Search to respond. But if not, it might say "This is a premium feature, please subscribe." - Connect to services: For instance, a user might send "I need help with statistical analysis". The bot could recognize this as a service request initiation and guide them (ask for details, possibly create a request in system). - The chatbot might also handle OTP verification or quick info. But OTP we handle via direct message sending, not via user prompting the bot.

**Bot Flow Implementation:** We will set up a webhook endpoint (e.g., /webhook/bigtos) where BigTos will POST messages it receives (in a structured format likely containing sender number and message text). Our controller will parse the message text and manage a conversation state. We can do either: - Use a stateful approach: e.g., if a user triggers a service flow, we keep track that next message from them should be treated as 'details' etc. We can maintain a chatbot\_session array in cache or a table keyed by user phone. - Or a stateless but guided approach: rely on them using numeric menus every time (less friendly but simpler to parse). A hybrid likely: initial menu by number, then some flows (like service request) need multiple steps.

**Possible main menu branches:** 1. **Directory**: If user asks for a doctor or hospital (e.g., "Find cardiologists in Delhi"), the bot could query our database and return top results: "Dr. X at Y Hospital. See details: link". This requires parsing natural language or offering a structured query form through chat (like asking step by step specialty and location). We can implement a simple recognition: if message contains "find" or user chooses a menu option 'Find Doctor', then ask "Please provide specialty or name" then search accordingly. 2. **Quizzes**: If user inquires about quizzes or chooses "Quizzes", the bot might list upcoming quizzes or allow registration. e.g., "Next quiz: [title] on [date]. Reply 1 to register." If they reply 1, we add them to that quiz’s registrations (provided we can identify them – we might require them to have an account linked to that WhatsApp number). If they don't have an account, the bot might ask for name to create one or direct them to web. 3. **Courses/Masterclasses**: Could similarly list upcoming or allow enrollment via chat. 4. **Services**: If user says "I need X service", the bot can go into service request flow. For example: - Bot: "Sure, you need [service]. Please provide details of what you need." - User: (their description) - Bot: "Got it. Do you want to attach a file? If yes, email it to support (since WhatsApp bot might not handle file upload well unless BigTos can forward files via API). Possibly BigTos can handle file messages too; if it does, we could accept file if user sends one next. - Bot: "What is your location?" (if needed) - After gathering, bot either creates the request and then says "Thank you. We will get back with a quote or provider assignment shortly." and triggers provider notifications as earlier. If payment was required upfront, bot would have to give a payment link or ask them to visit website. Perhaps for simplicity, any requests via bot could be handled as pay later (some trust involved). 5. **General FAQs or Chat**: The bot might answer questions like "What is DocsUniverse?" or "How to register for a course?" with static answers. 6. **Human Handover**: If query is not handled or user requests to talk to an admin, the bot might flag an admin (maybe by sending an email or WhatsApp to admin to intervene). Or BigTos might allow handing off chat to a human via their interface.

Given the complexity, we'll cover main flows: service request and quiz registration as examples.

**Integration details:** - We will get messages from BigTos with a certain format. We parse user’s phone number to identify them. Ideally, the user’s WhatsApp number should match the users.phone. If a message comes from an unknown number (not registered), the bot can reply: "Hello! It seems you are not registered. Please visit our site to register or reply with 'Register' to create an account." If they reply 'Register', we can collect name, role, etc., and create a user (simpler route might be to send them a registration link, but automated registration via chat could be done with guided questions). - Once identified or after registering, we can personalize responses.

**Example Chatbot Interaction (Service Request via WhatsApp):**

User: Hi  
Bot: Hello, welcome to DocsUniverse! How can we assist you today?  
1. Find a Doctor/Hospital  
2. Courses & Quizzes  
3. Research Services  
4. Chat with support  
User: 3  
Bot: Research Services - We offer: 1) Thesis/Article Writing, 2) Statistical Analysis, 3) Printing. Reply with the number of the service you need.  
User: 2  
Bot: You chose Statistical Analysis service. Please briefly describe what you need help with.  
User: I have data from a clinical trial and need help with analysis and interpretation.  
Bot: Thank you. Please provide your city/location for us to connect you with a local expert (or type 'Online' if not applicable).  
User: Online  
Bot: Great. We will review your request and connect you with an expert shortly. (Request ID #46 created)  
(At this point, our system creates service\_request #46, and triggers provider notifications on the backend.)  
Bot: [After maybe assignment] Your request has been assigned to Dr. Stat (Service Provider). They will reach out to you on WhatsApp soon.   
User: Thank you!

During this, the bot created an account or recognized the user. If not recognized, at start bot could have asked: "Please provide your registered phone or name to continue." We assume user already a member if they're using this.

**Example Chatbot Interaction (Quiz registration via WhatsApp):**

User: I'd like to know about quizzes.  
Bot: Upcoming Quiz: "Diabetes Quiz" on Nov 10, 7 PM. Reply "Join Quiz" to register.  
User: Join Quiz  
Bot: You've been registered for the Diabetes Quiz on Nov 10. We will send a reminder before it starts. Good luck!

Our backend then adds that user to quiz\_registrations. Possibly also share a link to the quiz page: "You can participate via our website: [link]" since answering via chat isn't planned.

**Technical Implementation:** - The chatbot flow as above will be coded in conditional logic. For complex flows, we might use a small state machine (e.g., store in DB that +91XXXX is in "awaiting\_service\_detail" state after they choose service, so the next message goes there). - BigTos might allow us to define some flows on their platform (like a Google DialogFlow style). But since we have our server, we can handle it ourselves for flexibility. - The integration with our system has been described in respective modules (like how quiz registration or service creation is triggered by chat). - We will thoroughly test these flows because they involve asynchronous communication (user might not follow script exactly; we should handle unexpected input by maybe repeating options or saying "Sorry, I didn't get that. Please reply with a number from the menu.").

### NPA & Dynamic PDF Generation

This part focuses on generating PDF documents using user data and sending via WhatsApp. We touched on this in context of courses (certificates), quizzes (certificates), and possibly service agreements. The mention specifically of **NPA** (which could be "Network Participation Agreement" or similar) suggests an automatic document when a doctor signs up or something.

**NPA Document:** - Possibly a document confirming that the doctor has joined the DocsUniverse network, listing their details. It might be needed if they collaborate or something. We will treat it as a certificate or letter. - The content might include fields from the doctor’s profile (e.g., "This certifies that Dr. [Name], [Qualification], specializing in [Specialty] has joined the DocsUniverse network on [Date]."). Or maybe it's a contract they need to sign (if it's an agreement). But since we send it via WhatsApp, probably a certificate. - We will generate this after a doctor registers and profile is complete or approved. The system can automatically do it on approval. - Implementation: create an HTML template for NPA. Use placeholders like {{NAME}}, {{SPECIALTY}}, etc. On generation, fetch the doctor's data, replace into template, and use mPDF to output PDF. Make sure to include any logos or letterhead images by referencing them (mPDF can embed images base64 or via URL if accessible). - Then call WhatsApp API to send the PDF. BigTos likely has a REST endpoint like sendMediaMessage(phone, fileurl) or maybe they ask us to upload to their server first. We'll find out from their docs (not explicitly available to us, but we assume it's feasible). - We should also possibly email it as a backup if needed (but not required if WhatsApp works). - Log that it was sent (maybe mark in DB NPA\_sent\_at timestamp in the doctor record to avoid re-sending).

**Service PDFs:** - Could refer to invoices or summary documents for service requests. For example, when a service is done, perhaps they generate a PDF summary or invoice. Or maybe a cover letter that goes with the output. - Given limited info, we can assume we at least send an invoice if payment was taken (the Razorpay could be used to generate a receipt anyway, but a nice invoice on letterhead might be good). - Or a "No Plagiarism Assurance (NPA?) letter" with the user's and provider’s details guaranteeing something. - Without clarity, we will state that various documents such as confirmation letters, agreements, or certificates will be generated as needed using mPDF, using HTML templates that incorporate user profile and request details. These will be automatically delivered to relevant parties via WhatsApp for convenience.

**MPDF Use:** - As referenced, *“mPDF is a PHP library which generates PDF files from UTF-8 encoded HTML.”*. We'll utilize this library in CodeIgniter by installing via Composer and then loading it in our controllers where needed (like CertificateController, etc.). We will ensure to set up any required font or temp directory for mPDF (on Replit, we have write access to /tmp presumably for mPDF). - We may create a helper class PdfGenerator to centralize PDF creation of various types to avoid repeating code.

**Template Management:** - We can store the HTML templates either in the database (so admin can edit them? not likely needed) or as view files under a folder (e.g., app/Views/templates/npa\_template.php). - We'll opt for view files for simplicity. Then the controller can load the view with data and get HTML (by using output buffering or CI's view rendering to string), then feed to mPDF.

By fulfilling this, every formal document needed in the platform is automated, reducing manual work for admin and giving immediate feedback to users.

### Payment Gateway Integration

We will integrate a payment gateway (preferably Razorpay for Indian context, with Stripe as an alternative option for international usage). The integration is done such that it is reusable for any payment needs (courses if later monetized, masterclasses, services, etc.).

**Choice of Gateway:** Razorpay is a leading payment gateway in India with user-friendly checkout options, and CodeIgniter integration is straightforward. Stripe is global and robust, but since the question explicitly mentions Razorpay or Stripe, we will focus on Razorpay as first choice, noting Stripe can be swapped if needed.

**Integration Steps:** (Summarizing from the earlier guide)  
1. **Obtain API Keys:** Admin/Developer will create an account on Razorpay and get the Key ID and Secret. These will be stored in the CodeIgniter .env or config file (not in code) for security.  
2. **Include Razorpay PHP SDK:** We will run composer require razorpay/razorpay to add the SDK. CodeIgniter can autoload composer libraries, so we'll ensure that's enabled.  
3. **Creating an Order (Payment Initiation):** When user proceeds to pay (for an order of cart items or a direct single item purchase), our PaymentController will: - Initialize Razorpay API with our keys. - Create an order via API by specifying amount (in paise for INR), currency, and an order receipt ID (we can use our order\_id or a unique string). This returns a Razorpay order ID. - Save that Razorpay order ID in our orders table record (for future verification). - Pass necessary data to the checkout page view: order\_id, amount, key\_id, currency, etc.. - The checkout page will include Razorpay's checkout script which opens the payment form (card/netbanking/UPI etc). We also pass things like customer name, phone, email to pre-fill. - The user completes payment on Razorpay’s widget. Razorpay then calls our configured callback URL or returns control if using a popup. Typically, we set a callback route for success. 4. **Payment Verification (Callback):** We’ll have a route like /payment/verify which Razorpay will call (or our JS will call after receiving payment\_id and signature). In that method: - We capture razorpay\_payment\_id, razorpay\_order\_id, and razorpay\_signature from POST. - Use the Razorpay SDK to verify the signature with our key secret. This ensures the payment is authentic and not tampered. - If verification passes, mark the order as paid in DB (orders.payment\_status = "Paid", record payment\_id). - If needed, retrieve the payment details via API (to get method, card last4, etc. – optional). - Then fulfill the order: i.e., call functions to finalize each item (enroll user in course, register for masterclass, mark service paid as described earlier). - Finally, show a success page to user: "Payment successful! Your order has been processed.". - If verification fails or payment was incomplete, handle accordingly (show error, allow retry). 5. **Stripe Alternative:** If using Stripe, we would do something similar with Stripe’s API (create CheckoutSession, redirect, webhook to confirm payment, etc.). We won’t detail fully due to scope, but mention that by abstracting our PaymentController logic, we can plug Stripe in similarly.

**Cart Integration:** The payment process is always triggered by something in the cart. So user might be paying for multiple items. We will sum item prices into one order. Razorpay allows a single amount, so the user pays once for all. Our fulfillment logic will iterate each order\_item and do the needful in each module: - If multiple different types in one go, we must handle each: - Example: user buys a Masterclass and a Service in one checkout. After payment, we call something like fulfillOrder($order\_id) which does:

$items = OrderItemModel::where('order\_id',$order\_id)->get();  
foreach($items as $item) {  
 switch($item->type) {  
 case 'Masterclass': register user for class ($item->item\_id);  
 case 'Service': mark service\_request paid and active;  
 case 'Quiz': mark quiz registration;  
 // etc.  
 }  
}

If any item needs special handling (like maybe generating a ticket for masterclass or sending an initial info), we do that here or trigger events.

* **Refunds and cancellations:** We should note if an event is cancelled or service cannot be delivered, admin may need to refund. Razorpay’s API can refund via payment\_id if needed. We won’t fully detail, but ensure we have processes or at least admin can mark something and manually handle refund outside system if necessary.

**Security & Compliance:** - Use HTTPS (on Replit the domain is HTTPS by default, but in production definitely). - Ensure we validate the amount on server side too (don’t rely purely on what user side says they pay for, though with Razorpay order it’s locked). - Follow Razorpay’s data handling guidelines (e.g., do not store card details ourselves; we won’t since Razorpay handles it). - The system should also handle failures gracefully (if payment fails, the order stays pending; user can retry from an "Orders" page or gets a prompt to try again).

Citing the earlier reference, integrating Razorpay provides secure transactions and reliability which aligns with our approach to ensure trustworthy payment processing on the platform.

### Cart System Implementation

We have described the cart database above. Here is how it works in practice:

* The cart is essentially a session-stored list until checkout. We will create a Cart library or simply use CI’s session.
* **Adding to Cart:** When user clicks "Add to Cart" for an item:
* We determine item type and ID from the button (could encode in URL or form hidden field).
* In CartController@add, we fetch item details (like name, price) from respective table to record.
* We then add to session array. For example:
* $\_SESSION['cart'][] = ['type'=>'Masterclass', 'id'=>5, 'name'=>'XYZ Workshop', 'price'=>1000];
* If item already in cart (like they click again or two of same, we might either prevent duplicates or allow quantity if it made sense – not likely here because each item is unique event/service).
* Redirect to a "Cart" page showing all current items.
* **Viewing Cart:** On the cart page (CartController@index), we read $\_SESSION['cart'], calculate total. Show each item with name, price, and a remove button. Remove just pops it from the array. We provide a "Checkout" button if items exist.
* **Checkout:** Clicking checkout goes to CartController@checkout:
* This will create an orders record (with status pending, total, user\_id).
* Create order\_items records for each session cart item.
* Then clear the session cart (or mark it as moved to order).
* Redirect to payment initiation (PaymentController) with that order\_id.
* Alternatively, we can combine checkout and payment initiation in one step. We might directly invoke Payment creation after making order.
* We may incorporate CodeIgniter’s Cart Class as earlier discussed, but given it’s deprecated and relatively simple to custom implement, we likely do our own using session or use it carefully with session driver.
* This cart system unifies purchases. The user doesn’t need to separately pay for each thing. It’s an improvement for user experience.
* After payment, the order\_items provide references to do the actual logic as in Payment section.

**Edge cases:** If user navigates away, the cart persists in session (which on Replit might persist as long as the repl is running and user’s session cookie exists). We should also allow user to cancel an order if not paid or remove items. - Possibly an "Orders" page to see past orders and statuses, but for now maybe unnecessary since we handle immediate fulfillment. Could be useful if something is pending.

**Summary**: The e-commerce flow is akin to any online shopping site but with services/events instead of tangible goods, and ensures one unified payment and order tracking. This is robust and easy to extend to new purchasable items if any in future.

## Admin Panel and Content Management

We have referenced admin capabilities throughout; here we summarize the admin panel structure and how the content can be managed easily:

**Admin Dashboard:** - A secure portion of the site (e.g., accessible at /admin after admin login) where navigation links are provided for each module’s management interface. - We can create an AdminController for general views and separate controllers or methods for each section. Alternatively, group by functionality: - **User Management:** list of users, with filters by role. Admin can promote a user to a provider or to admin, etc., and deactivate accounts if needed. - **Doctor Directory Management:** view all doctor profiles. Admin can edit any, approve new sign-ups (if we implement approval needed). Possibly an interface to import bulk doctor data if migrating (less needed after initial migration). Also manage hospitals (create/edit). Perhaps manage specialties list (though specialties rarely change, but we could allow adding). - **Courses:** Admin can create new course, edit content, view enrollments. Possibly see completion stats. - **Quizzes:** Admin can create quizzes (with the form as described in quiz section). Also view results of quizzes (who participated, scores). For live quizzes, admin might manage the live session here or at least see the “Start” button and control next question (we might build a simple admin live view showing current question and a “Next” button to trigger next question). - **Masterclasses:** Admin creates/edits events, and can view registrants. They can download a list or send a broadcast message to attendees if needed (like “Reminder tomorrow…” via WhatsApp or email). They can close registration or cancel event via admin interface. - **Jobs:** Admin reviews any job posts made by others, or posts new ones. They can edit or mark filled. They can also review any job applications if we did that and forward them. - **AI Tools config:** If we have any settings (like API keys), admin panel might have a settings page to input those keys or toggle which tools are active. (At least we need to input e.g. OpenAI key securely; we might do that via a config file instead for security, but if needed admin can paste it in a form which updates an .env or config table). - **Research Services:** \* Manage services list (add new service offerings or modify descriptions/prices). \* View all service requests. Possibly filter by status (pending/in progress/completed). \* For each request, see details, and intervene: e.g., assign a provider manually (a dropdown of providers if not assigned or reassign if needed), mark completed, or cancel. \* Also see transaction status (if unpaid, maybe follow up). \* Manage service providers: perhaps an interface to add new provider accounts (enter their name, phone, services they handle, etc.), which sends them an invite or just creates an account. - **Content Pages / Misc:** There might be static pages (like “About Us”, “Privacy Policy”) as indicated by search result (sparshekpahal.in privacy). Possibly they had WP pages. In our CI rebuild, we can create a simple CMS for basic pages or just hardcode them in views. If needed, admin panel could have a section to edit such pages using a WYSIWYG editor storing content in a pages table. Not explicitly asked, but if aiming comprehensive, mention that we can manage static info similarly.

**Admin Panel UI:** We'll use a simple template (maybe an admin LTE or even just Bootstrap admin template) to layout menus and pages. Since focus is functionality, even a basic table view and forms is fine.

**Example Admin Usage (Course & Quiz content prompt as requested):** - *Course example:* In the admin panel, the admin clicks "Add New Course". They fill out the form fields: - **Title:** *“Advanced Cardiac Life Support (ACLS)”*  
- **Category:** *“Health”*  
- **Duration:** *“2 Weeks”*  
- **Fee:** *“0”* (free course)  
- **Description:** *(Enters a rich text describing the course goals and outline)*  
- After saving, the course appears in the Courses list. Then admin clicks "Manage Lessons" for that course to add modules: e.g., 1. *“ACLS Module 1: Introduction”* (text about algorithms for cardiac arrest)  
2. *“ACLS Module 2: Respiratory Emergencies”* etc. - Admin also associates a final quiz if needed by entering the quiz ID or creating one. - *Quiz example:* Admin goes to "Quizzes" and clicks "Create Quiz". They fill: - **Title:** *“ACLS Final Quiz”*  
- **Description:** *“Certification quiz for ACLS course participants.”*  
- **Live or Static:** choose static (since it’s for course and can be taken anytime after course).  
- Then add questions: - Q1: *“What is the recommended compression to ventilation ratio in CPR for an adult?”*  
Options: A) 30:2 **(correct)**, B) 15:2, C) 10:1, D) Continuous compressions; Time: 30s; Points: 10. - Q2: *“Which drug is first-line in anaphylactic shock?”* Options: etc. - Save quiz. Now, that quiz can be linked to the course or available to those who finish course. Alternatively, admin sets it not live so it’s accessible any time (maybe via a link on course completion page). - Admin ensures to test these by maybe taking the quiz themselves with a test account.

These examples illustrate how an admin populates content and uses the system.

All content management functions are designed to be **structured and user-friendly** so that the platform managers can update information without needing a developer.

## Deployment on Replit: Setup Instructions

Deploying this CodeIgniter + MySQL application on Replit involves several steps. Below are the instructions to get the platform up and running in the Replit environment:

1. **Replit Environment Setup:** Log in to Replit and create a new Repl. Choose "PHP Web Server" as the language/environment. Replit will provide a basic PHP/Apache setup. We will need to customize it for CodeIgniter. Ensure the Repl is set to private (if needed, since our code will contain API keys).
2. **Add CodeIgniter Framework:** We can install CodeIgniter 4 via Composer. Open the Replit shell and run:

* composer create-project codeigniter4/appstarter docsuniverse-ci
* This will create a CodeIgniter project in a subdirectory. Alternatively, we can upload the prepared CodeIgniter project files directly if we have them. If using Composer, after creation, move the files to root or adjust Replit to serve from the public directory. (We might need to configure Replit’s run command to php spark serve (CodeIgniter’s built-in server) or use Apache pointing to public/ folder.)  
  Make sure the public directory’s contents are accessible as web root. In Replit’s settings, set the web root accordingly (Replit might auto-detect or we may put an index in root that includes public – but better adjust run).

1. **Install Dependencies:** Use Composer to require the libraries:
2. Razorpay: composer require razorpay/razorpay
3. mPDF: composer require mpdf/mpdf
4. (If we use any others like for AI or if BigTos provided an SDK, include similarly. BigTos likely doesn’t have a PHP SDK, it’s just HTTP calls, which we can do with cURL or PHP’s built-in streams. We might also require guzzlehttp/guzzle for convenient HTTP requests to APIs.)  
   Composer will update the vendor folder. Ensure CodeIgniter’s autoload is set to pick up these (CI4 by default does).
5. **Database Setup:** Replit doesn’t have MySQL by default, but we can add it. Go to the Shell and use Nix package manager. For example, open the Replit “Shell” and enter:

* apt-get install mysql-server
* This may install MySQL. If that fails, we might use mysql-lite or consider using SQLite for development (the CodeIgniter 4 starter even uses SQLite by default which might be easier on Replit). However, since MySQL is required, after installing, we need to start the MySQL server. Replit run command can be adjusted to start MySQL in the background. Possibly add to the .replit run command something like:
* mysqld & php spark serve --port 3000
* (ensuring the MySQL daemon uses a safe port or default since Replit might not allow default port exposure, but since PHP and MySQL run in same container, it’s fine).  
  Next, configure the database:

1. Set a root password for MySQL or create a new user and database. We can do:

* mysql -e "CREATE DATABASE docsuniverse; CREATE USER 'ci\_user'@'localhost' IDENTIFIED BY 'password'; GRANT ALL ON docsuniverse.\* TO 'ci\_user'@'localhost';"
* This creates a DB named docsuniverse and a user.

1. In CodeIgniter’s .env or app/Config/Database.php, set the database connection:

* database.default.hostname = '127.0.0.1';  
  database.default.database = 'docsuniverse';  
  database.default.username = 'ci\_user';  
  database.default.password = 'password';  
  database.default.DBDriver = 'MySQLi';
* Use the credentials we set. If using .env, ensure to un-comment the needed lines.

1. Run migrations or import schema: We will write migration files for each table or a single SQL file. For initial setup, we can import a SQL dump. Copy our schema SQL into a file and use the MySQL command to import it:

* mysql -u ci\_user -p docsuniverse < schema.sql
* (It will ask for password). Alternatively, in CodeIgniter, create migration classes and run php spark migrate.

1. Verify the database connection by running a simple query via a controller or using spark db:table if available.
2. **Configuration:**
3. Set the base URL in app/Config/App.php or .env, e.g., app.baseURL = 'http://docsuniverse-<yourReplUsername>.repl.co/' (the actual Repl URL). This is important for URL helpers, etc.
4. Enable development mode in .env while testing to see errors.
5. Set up email config if needed (for sending emails, though we mostly use WhatsApp).
6. Set timezone in PHP config if needed (to IST if usage is primarily India, or just use UTC).
7. Add any required config for BigTos: possibly store the API endpoint and an auth token or credentials. We might add these to .env like bigtos.apiKey = '...' if BigTos uses an API key, and maybe bigtos.endpoint = 'https://bigtos.com/api/...'. If BigTos uses an account login, we might have to store those credentials too (we’d clarify from BigTos documentation). Ensure these are not exposed (again .env is good).
8. Razorpay keys: add to .env razorpay.keyId and razorpay.keySecret. Our PaymentController will load them from here.
9. **Embed Uploaded Files:** For profile photos, service docs, etc., we need to ensure the writable directory is configured properly (CI4 uses writable/uploads by default for any file operations if configured). We should create subfolders for profile\_images, request\_docs, etc. On Replit, file storage is persistent as long as the repl exists (and is not hitting space limits). We caution that Replit might have limited space, but for dev it's fine. For production, use an external storage if needed.
10. **Testing on Replit:**
11. Start the application. Replit should show the web view (likely at the default / path).
12. Check that the homepage (we’ll create either a landing page or redirect to appropriate page) loads without errors.
13. Test database connectivity by registering a user via the web interface or directly trying a small query.
14. Visit the admin URL and ensure it requires login. You might need to create an admin user manually in DB since initially no admin. Use MySQL to update a user's role to 'Admin' or insert an admin user row.
15. Test OTP: Since we can't actually receive WhatsApp in this test easily, consider temporarily allowing a dev mode: e.g., if environment=development, after user requests OTP, log the code to screen so we can copy it. In production, obviously it would be sent to WhatsApp. For now, we might simulate or if BigTos provides a testing sandbox, use that. But likely we need an active BigTos account and a connected WhatsApp number for fully testing OTP and chatbot. We'll mention to obtain those credentials from BigTos and configure webhook URL as our Replit URL + /webhook.
16. Test a payment: Razorpay might allow test mode transactions with test API keys. Use those in dev. Launch a payment for a dummy item (e.g., create a test service ₹100, add to cart, checkout). Complete using Razorpay’s test card details. Ensure our verify logic works and marks order paid.
17. Run through each module's basic functions to catch any issues.
18. **Going Live:** Replit is mainly for development/demos. If using for a small-scale production, ensure always-on is enabled. We may have to keep the tab open or use Replit’s always-on upgrade to keep server alive and the MySQL running. If traffic grows, Replit’s constraints might be an issue (e.g., it might sleep the repl if idle unless always-on, and heavy usage might require a move to a proper hosting).
19. But for now, once it’s working on the Repl URL, share that or custom domain map if needed.
20. **Environment Variables (in Replit Secrets):** Use the Replit Secrets manager to store any sensitive keys (Razorpay secret, BigTos credentials). Then in our code (or .env loaded via environment), refer to them. For example, in .env we set razorpay.keySecret = ${REPLIT\_RAZORPAY\_SECRET} which Replit can replace if configured. This avoids committing secrets in code.
21. **Migration of Data:** Finally, to migrate existing data (like all doctors, hospitals from old system), we can write a PHP script or use an import strategy:
    * If we have the old database or CSV exports (the user provided maybe some PDFs with data, but ideally, they have the actual data somewhere), we could import via MySQL commands or create a script to read old data source and insert into new DB.
    * If that’s manual, instruct to prepare an SQL or CSV of old doctors and run through MySQL. The main point is our schema matches all fields, so a straightforward INSERT ... SELECT from old to new mapping columns would do.
    * For safety, do this offline or initial phase, since on Replit doing heavy import might be slow.

Now the app should be up and running on Replit, accessible via the given URL. We have essentially deployed a full LAMP stack application in that environment, with all modules ready for use. Always test all functionalities on the Replit URL because certain things (like file generation or external API calls) might have nuances in that sandbox.