ChatGPT

Synapaxon: Medical Question Bank Platform

Project Overview

Synapaxon is an online medical question bank and learning platform built with the MERN stack (MongoDB, Express.js, React, Node.js). It enables medical students to **create**, manage, and practice user-generated multiple-choice questions, fostering collaborative learning. Students register, contribute questions (annotated with topic tags, difficulty levels, images, etc.), and take custom practice tests. Administrators oversee content quality and user access by approving submitted questions, managing subscriptions, and monitoring analytics.

Technically, Synapaxon will be implemented as a RESTful web application. The backend uses Node.js with Express and MongoDB (via Mongoose) for data storage. The frontend uses React for a dynamic, responsive UI. Authentication is stateless using JSON Web Tokens for sessions [1], and we will enforce Role-Based Access Control so that students and admins have appropriate permissions [2].

Core Features

- **Public Landing Page:** An informational marketing page (no login required) describing Synapaxon's features (custom quizzes, user-generated content, subscription plans, etc.) and pricing. It encourages visitors to sign up or log in. Navbar/footer links will include Signup, Login, FAQ, and Contact pages.
- **Role-Based Authentication:** Users have roles ("student" or "admin"). Students can register with email/password, while admin accounts are created manually by the system owner. On login, the server issues a JWT (including the user's role). Passwords are hashed with bcrypt (a widely used, secure algorithm [3]) before saving. An Express middleware verifies the JWT on protected routes and enforces roles (e.g. only `role='admin'` users can access `/api/admin/*` endpoints).
- **Student Features:** After logging in, student users can:
- **Create Questions:** Submit new MCQ questions via a form. Each question includes text, one correct answer, three wrong options, an explanation, and optional hints. Students tag the question with relevant topics and a difficulty level (Easy/Medium/Hard). They can also upload an image (e.g. a diagram) through a secure upload flow (e.g. Cloudinary or S3) – the backend stores only the returned image URL. New questions are saved with `status=Pending` until an admin reviews them.
- **Take Practice Tests:** Launch quizzes by selecting filters (topic/tag, keywords, difficulty). The backend selects a random set of approved questions matching the filters and creates a new `TestSession`. The frontend's **Test Engine** displays questions one by one (with an optional timer) and records answers.
- **View Test Reports:** After completing a test, students see their score and detailed feedback (showing which questions were correct or incorrect, along with explanations). Each test result is saved in the database. Students can view their test history on their profile; performance charts (e.g. pie/bar graphs) illustrate strengths and weaknesses over time.
- **Edit Profile:** Update personal info on a profile page. Students can upload an avatar image, edit their display name, write a bio/about section, and list skill tags. They can also change email or password

(with new passwords hashed via bcrypt). Profile updates are sent to the backend (`PUT /api/profile`).

- **Notifications:** View in-app notifications for events (e.g. "Your question was approved"). Notifications are stored in the `Notification` model. Additionally, important events (approval, password reset, etc.) will trigger emails via a queued email service (e.g. nodemailer or SendGrid).
- **Subscriptions:** Subscribe to premium plans to unlock extra features. The subscription page shows plan details and a "Subscribe" (Stripe Checkout) button. After payment, Stripe's webhook updates the user's subscription status in our database. Premium subscribers might get features like personalized test suggestions and a CV/resume builder.
- **Admin Features:** Admin users access a protected admin dashboard (no open signup). Key admin capabilities include:
- *Content Approval:* View a list of newly submitted questions (status Pending). Admins can **approve** questions (making them visible to students) or **reject** them (optionally providing feedback).
- *User & Subscription Management:* See all student accounts with their subscription plans and statuses. Admins can search/filter users and manually adjust subscriptions or revoke access if needed. (Typically, subscription status updates via Stripe, but admins can override.)
- *Question/Topic Management:* Create/edit/delete any question or topic. There will be forms to manage medical topics/categories and to upload or link reference images/videos.
- *Analytics & Reporting:* Dashboard charts summarizing usage (questions per category, active users, test statistics, average scores, etc.). This helps admins identify trending topics and areas where students struggle.
- *Platform Settings:* (Optional) Configure system-wide settings (email templates, difficulty weights) and review user feedback or flagged content.

Backend Summary (Node.js + Express + MongoDB using Mongoose)

The backend is a RESTful API server built with Node.js and Express. Data is stored in MongoDB (using Mongoose schemas). Key points:

- **Authentication & Authorization:** Use JSON Web Tokens for user sessions [1]. On login/register, sign a JWT containing the user ID and role. Protected routes use middleware to verify tokens and attach `req.user`. Role-based middleware checks `req.user.role` against allowed roles [2] (e.g. `role='admin'` for admin routes). Passwords are hashed with bcrypt before storing [3]. All endpoints enforce HTTPS (in production) and configure CORS appropriately.
- **Data Models (Mongoose Schemas):**
- **User:** Fields include `name`, `email`, `passwordHash`, `role` (enum: `'student'` / `'admin'` [4]), `avatarUrl`, `skills` (array of strings), `about` (string), plus subscription fields (`planId`, `status`, Stripe customer/sub IDs).
- **Question:** `{ text, options: [String], correctAnswer: String, explanation, difficulty (enum), tags: [String], imageUrl, status (enum: Pending/Approved/Rejected), owner: ObjectId (User), timestamps }`.
- **Topic:** `{ name, description }`. Used for categorizing questions.
- **TestSession:** `{ user: ObjectId, questions: [ObjectId], answers: [String], score: Number, completedAt: Date }`. Records each quiz attempt.
- **Subscription:**
`{ user: ObjectId, planId: String, status: String, stripeCustomerId: String,`

`stripeSubscriptionId: String, currentPeriodEnd: Date }`. Stores subscription info per user.

- **Notification:** `{ user: ObjectId, message: String, link: String, isRead: Boolean, timestamp: Date }`. For in-app alerts.
- **Feedback (optional):** `{ user: ObjectId, content: String, createdAt: Date }`. If collecting user feedback.

- **Controllers & Logic:** Each main resource has a controller:

- **AuthController:** `register`, `login`, `getMe` (return current user).
- **UserController:** `getProfile`, `updateProfile`.
- **QuestionController:** `createQuestion`, `getQuestions` (with filters/pagination), `getQuestionById`, `updateQuestion`, `deleteQuestion`. (Note: creation defaults to `status=Pending`.)
- **TestController:** `startTest` (selects questions and creates a TestSession), `submitTest` (grades answers and updates TestSession).
- **AdminController:** `getAllUsers`, `getPendingQuestions`, `approveQuestion(id)`, `rejectQuestion(id)`, `getAnalytics`.
- **TopicController:** `getTopics`, `createTopic`, `updateTopic`, `deleteTopic`.
- **SubscriptionController:** `getPlans` (list available plans), `handleStripeWebhook` (processes Stripe events to create/update Subscription records), `getSubscriptionStatus`.
- **NotificationController:** `getNotifications`, `markAsRead`.

- **(Optional) MediaController:** Handles image upload if doing server-side file processing (e.g. using multer or Cloudinary SDK).

- **Middleware:**

- `authMiddleware` to verify JWTs and populate `req.user`.
- `roleMiddleware(allowedRoles)` to enforce RBAC on routes.
- Input validation (e.g. Joi or express-validator) to ensure request data is clean.

- Error-handling middleware to catch and format errors.

- **API Routes:** Main endpoints under `/api`:

Endpoint	Method	Access	Description
`/api/auth/register`	POST	Public	Register a new student (returns JWT).
`/api/auth/login`	POST	Public	Login (student or admin); returns JWT.
`/api/auth/me`	GET	Student, Admin	Get current user profile (requires JWT).
`/api/questions`	GET	Student (Admin)	List/filter questions (students see only approved ones).

Endpoint	Method	Access	Description
`/api/questions`	POST	Student	Submit a new question (`status=Pending`).
`/api/questions/:id`	PUT	Owner/ Admin	Update a question (only owner or admin).
`/api/questions/:id`	DELETE	Owner/ Admin	Delete a question (owner or admin).
`/api/tests/start`	POST	Student	Start a test with selected filters (creates TestSession).
`/api/tests/submit`	POST	Student	Submit test answers; returns score.
`/api/profile`	GET	Student, Admin	Get own profile data.
`/api/profile`	PUT	Student, Admin	Update own profile (avatar, name, etc.).
`/api/admin/users`	GET	Admin	List all user accounts.
`/api/admin/questions/pending`	GET	Admin	List pending questions for review.
`/api/admin/questions/:id/approve`	POST	Admin	Approve a question (`status = Approved`).
`/api/admin/questions/:id/reject`	POST	Admin	Reject a question (`status = Rejected`).
`/api/admin/analytics`	GET	Admin	Get usage analytics (user/test stats, etc.).
`/api/topics`	GET	Student, Admin	List all topics.
`/api/topics`	POST	Admin	Create a new topic/category.
`/api/subscription`	GET	Student	Get current user's subscription details.
`/api/subscription`	POST	Student	Create/update subscription (triggered by Stripe webhook).
`/api/notifications`	GET	Student	List notifications for the current user.
`/api/notifications/:id/read`	POST	Student	Mark a notification as read.

Frontend Summary (React)

The frontend is a single-page application built with React (using Hooks). We will use React Router for navigation and Axios (or Fetch) for AJAX calls. Authentication state (JWT and user info) will be stored in React Context or a state manager; a `ProtectedRoute` component will guard private routes by checking auth and role. The UI will be responsive, and a component library (Material UI, Bootstrap, etc.) can be used for consistent styling.

- **Pages/Views:**
- **Home (Public):** A landing page overviewing Synapaxon's purpose, features, and pricing, with "Login"/"Sign Up" buttons.
- **Login / Signup:** Forms for authentication. The signup form (students only) collects name, email, and password. The login form is used by both students and admins (admins simply enter credentials).
- **Student Dashboard:** After login, students see:
    - *Create Content:* Form to submit a question (fields for text, answers, explanation, tags, difficulty, image upload).
    - *Take Test:* Interface to select filters (topics, difficulty) and click "Start Test."
    - *Test Engine:* Component to present quiz questions one at a time and record answers.
    - *Test Results:* Shows the score and which answers were correct/incorrect (with explanations).
    - *My Tests / History:* List of past tests (date, score) with ability to view details or performance charts.
    - *Profile:* Shows user info; fields can be edited (upload avatar, edit name/skills/bio).
    - *Notifications:* List of in-app notifications (e.g. "Your question was approved").
    - *Subscription / Settings:* Shows current subscription status and a button to manage the plan (which opens Stripe Checkout). Also includes a Logout button.
- **Admin Dashboard:** After login, admins see:
    - *Student List & Subscriptions:* Table of users and their subscription plans/status.
    - *Question Review:* List of pending questions with Approve/Reject buttons.
    - *Analytics:* Charts (bar/line/pie) summarizing site metrics (question count, user activity, test scores, etc.).
    - *Manage Topics:* Interface to add/edit/delete question topics/categories.
- **Components:**
- **Navbar/Header:** Top navigation that changes based on role (e.g. "Create Question" for students, "User Management" for admins).
- **ProtectedRoute:** A route wrapper that checks authentication (and optionally role) before rendering a page.
- **TestEngine:** Handles the quiz-taking flow (question display, timer, next/submit).
- **ResultCharts:** Renders graphs (e.g. Chart.js) for test results or analytics.
- **Alerts/Toasts:** For showing feedback messages (e.g. "Question submitted", error alerts).
- **Form Inputs:** Custom inputs for multi-select tags, file upload (for images), etc.
- **Loading/Error Indicators:** Spinners or messages shown during data loading or when errors occur.
- *(Additional UI components as needed, e.g. ProfileCard, Modal dialogs, etc.)*

Other Notes

- **Secure Image Upload:** For image uploads (questions, avatars), use a secure cloud storage solution. For example, Cloudinary's Node.js SDK allows uploading over HTTPS using an API key/secret [5]. After upload, only the image URL is stored in our DB, offloading file storage to Cloudinary's CDN.
- **Stripe Subscription Integration:** We will integrate Stripe for managing paid plans. The frontend uses Stripe Checkout for payments. Upon successful payment, Stripe sends a webhook (e.g. `checkout.session.completed`) to our backend. Following best practices, Stripe is the source of truth: we store only minimal data (customer ID, subscription ID) [6]. The webhook handler updates our `Subscription` records with the new plan/status and period end date.
- **Email & In-App Notifications:** Use a background job or scheduler (e.g. `node-cron`) to send emails (using nodemailer or a service like SendGrid) for events like question approval, account creation, or weekly summaries. In parallel, each such event creates a notification record in the database so it appears in the user's notification list (e.g. approving a question generates an in-app "Your question has been approved" message).
- **Admin Workflow (Subscriptions):** When a user completes a payment, their subscription status is auto-updated. Admins can log in to refresh or manually adjust a user's subscription status if necessary.
- **Security & Deployment:** Store secrets (JWT secret, DB URI, API keys) in environment variables. Use HTTPS in production and configure CORS to allow requests only from your frontend domain. Implement rate limiting and thorough input validation to prevent abuse. Plan to deploy via containerization (e.g. Docker) and CI/CD for reliability.
- **Testing:** It's advisable to write automated tests. For example, use Jest or Mocha to unit-test backend logic, and React Testing Library to test UI components. A CI/CD pipeline can run these tests on each commit.

This structured technical summary provides a comprehensive blueprint for developing Synapaxon, detailing its main features, data models, API endpoints, and front-end components to guide planning and implementation.

Sources: The design follows common MERN stack practices (e.g. JWT for stateless auth [1], RBAC for authorization [2], bcrypt for hashing [3], and Stripe's recommendation to let Stripe be the source of truth [6]). The information here is synthesized from modern web development patterns and documentation.

---

[1] JWT authentication: Best practices and when to use it - LogRocket Blog
https://blog.logrocket.com/jwt-authentication-best-practices/

[2] How to Implement RBAC in an Express.js Application
https://www.permit.io/blog/how-to-implement-rbac-in-an-expressjs-application

[3] Password hashing in Node.js with bcrypt - LogRocket Blog
https://blog.logrocket.com/password-hashing-node-js-bcrypt/

[4] How to implement role-based access control in Node.js and Express | by Jone Dev | Medium
https://medium.com/@jone.dev007/creating-user-schema-with-role-field-and-route-restriction-middleware-07718ac60dbc

[5] Node.js image and video upload | Cloudinary
https://cloudinary.com/documentation/node_image_and_video_upload

6 Stripe Subscription Integration in Node.js [2024 Ultimate Guide] - DEV Community
https://dev.to/ivanivanovv/stripe-subscription-integration-in-nodejs-2024-ultimate-guide-2ba3