TimeMyDay - Technical Documentation

Architecture Overview

TimeMyDay (also referenced internally as DeiLearning) is a modern, AI-powered Learning Management System (LMS) built as a Single Page Application (SPA). The architecture follows a decoupled, serverless model:

- **Frontend:** A React SPA powered by Vite, utilizing Tailwind CSS for styling and React Router for client-side navigation.
- **Backend & Database:** Powered by Convex, which provides a real-time database, serverless TypeScript functions (queries, mutations, and actions), and cron scheduling.
- **Authentication:** Managed via Clerk, providing secure user identity mapping to the Convex backend via `tokenIdentifier`.
- **Payments & Billing:** Integrates Stripe and Polar.sh for processing checkouts, tracking orders, and managing course access. Webhooks are handled natively via Convex HTTP actions.
- **AI Engine:** Integrates the Google Gemini API to dynamically generate course lessons, aptitude tests, and "Smart Insights" for learners.

Tech Stack

Frontend

- **Framework:** React 19 (`react`, `react-dom`)
- **Build Tool:** Vite 6
- **Routing:** React Router DOM v7
- **Styling:** Tailwind CSS v4 (`@tailwindcss/vite`), `clsx`, `tailwind-merge`

- **Animations:** Framer Motion, GSAP
- **Icons:** Lucide React
- **PDF Generation:** jsPDF, jspdf-autotable (for certificates)
- **Markdown Parsing:** react-markdown

Backend (Serverless)

- **Infrastructure:** Convex (`convex` v1.32)
- **Authentication:** Clerk (`@clerk/clerk-react`, `@clerk/react`)
- **Payment Gateways:** Stripe SDK, Polar.sh integration
- **AI Integration:** Google GenAI (`@google/genai`)
- **Security:** `bcryptjs` (Admin password hashing)

Testing & Quality Assurance

- **Unit/Integration:** Vitest
- **E2E Testing:** Playwright
- **Mocking/Testing DB:** `convex-test`

Project Structure

```
FLIPPA-TimeMyDay/
├── .github/workflows/       # CI/CD pipelines (Auto-merge, Cloud Deploy
├── convex/                  # Backend code (Database, API, Webhooks)
│   ├── _generated/          # Convex auto-generated types and API hooks
│   ├── schema.ts            # Database schema definitions
│   ├── http.ts              # Webhook HTTP routers (Stripe/Polar)
│   ├── seed.ts              # Database seeding logic for initial catalo
│   ├── learning.ts          # Core LMS logic & Gemini AI generation act
│   └── *.ts                 # Various queries, mutations, and actions
├── public/                  # Static assets
│   └── .htaccess            # Apache configuration for SPA routing
├── scripts/                 # Automation scripts (e.g., Stripe environm
├── src/                     # Frontend source code
│   ├── components/          # Reusable UI, layout, and route guards
│   ├── context/             # React Context providers (AuthContext)
│   ├── lib/                 # Utility functions
```

```
│   ├── pages/              # Top-level route components
│   ├── services/           # External service integrations (Gemini API
│   ├── App.jsx             # Main application router
│   └── main.jsx            # React DOM entry point
├── tests/                  # E2E test suites (Playwright)
├── package.json            # Project dependencies
└── vite.config.ts          # Vite bundler configuration
```

Database Schema

The database schema is defined in `convex/schema.ts`. Below are the core tables and their primary responsibilities:

- `users`: Stores end-users (students, instructors, admins). Indexed by Clerk's `tokenIdentifier`. Tracks role and avatar.
- `adminUsers`: Stores platform administrators with `username` and hashed `password`.
- `courses`: Stores the course catalog. Includes metadata like `title`, `price`, `category`, `level`, `isFeatured`, and `instructorId` (links to `users`).
- `enrollments`: Junction table mapping a `userId` to a `courseId`. Tracks `progress` (0-100), completion `status`, and AI `generationStatus`.
- `courseLessons` & `generatedLessons`: Stores the curriculum structure. `courseLessons` are manually created by instructors; `generatedLessons` are dynamically created via the Gemini AI per user enrollment.
- `questions` & `testAttempts`: Stores AI-generated multiple-choice questions for end-of-course exams, and logs the user's score, pass/fail status, and attempt timestamp.
- `certificates`: Stores records of completed course certificates, storing the generated `certificateFileId` and a unique `verificationCode`.
- `payments` & `withdrawals`: Financial ledgers tracking incoming course purchases (via Stripe/Polar) and outgoing instructor payouts.

- `reviews` & `testimonials`: User feedback mechanisms linked to courses and platform-wide displays.
- `smartInsights`: Caches AI-generated personalized study recommendations for users based on their progress and test scores.

Setup & Installation

Prerequisites: Node.js (v20+), npm, Convex Account, Clerk Account, Stripe Account, and a Google Gemini API Key.

1. **Clone the Repository & Install Dependencies:**

```
git clone <repository-url>
cd FLIPPA-TimeMyDay
npm install
```

2. **Environment Variables:** Create a `.env.local` file in the root directory mapping to `.env.example`:

```
VITE_CLERK_PUBLISHABLE_KEY="your_clerk_pub_key"
CLERK_SECRET_KEY="your_clerk_secret_key"
VITE_STRIPE_PUBLISHABLE_KEY="your_stripe_pub_key"
STRIPE_SECRET_KEY="your_stripe_secret_key"
STRIPE_WEBHOOK_SECRET="your_stripe_webhook_secret"
GEMINI_API_KEY="your_gemini_api_key"
VITE_FRONTEND_URL="http://localhost:3000"
```

3. **Initialize Backend (Convex):** Open a new terminal and start the Convex development server. This provisions your backend and syncs the schema.

```
npx convex dev
```

4. **Seed the Database:** To populate the platform with a rich catalog of 120+ courses, sample students, and instructors:

```
npx convex run seed:seed
```

5. **Start the Frontend Development Server:**

   ```
   npm run dev
   ```

   The application will be available at `http://localhost:3000`.

6. **Admin Access:** You can log in to the admin panel using the credentials:

   - **Username:** `codeblix`
   - **Password:** `codeblix`

Deployment Guide

Shared hosting is enough, no Vercel required. Since the backend, database, and scheduled jobs are fully managed by Convex, the frontend is a standard static Single Page Application (SPA).

1. **Deploy Convex Backend:** Deploy your Convex backend functions to production.

   ```
   npx convex deploy
   ```

   (Ensure you have set your production environment variables in your Convex Dashboard).

2. **Build the Frontend:** Compile the React application for production.

   ```
   npm run build
   ```

   This will generate a `dist/` directory containing your optimized static assets.

3. **Upload to Shared Hosting (cPanel / FTP):**

- Access your shared hosting File Manager or connect via FTP.
- Upload the **contents** of the `dist/` folder directly into your `public_html` (or your addon domain's document root).
- The project already includes a `public/.htaccess` file which gets bundled into the build. This ensures that Apache correctly routes all traffic to `index.html`, which is mandatory for React Router to function without throwing 404 errors on sub-pages.

Your application is now fully deployed and operational on your shared hosting environment.