Indian Institute of Information Technology
Guwahati (IIITG)

Coursify - Learning Management System

Submitted by

Chirag Shrimal

Under the guidance of

Dr. Nilkanta Sahu

Course: [CS-300]

Contents

Chapter 1

Abstract

Coursify is a modern, full-stack Learning Management System (LMS) developed using the MERN (MongoDB, Express.js, React, Node.js) stack. It addresses the need for accessible and affordable online education by offering a platform where administrators can create and manage courses, and users can subscribe to access a comprehensive library of educational content. Key functionalities include distinct user and admin roles with secure JWT-based authentication, comprehensive course and lecture management (CRUD operations), integrated analytics for administrators (user statistics, revenue tracking), and a streamlined subscription-based payment system facilitated by Razorpay. The platform prioritizes a clean user interface and a focused feature set, centered around a unique all-access, one-year subscription model, making it a highly competitive and value-driven alternative to traditional per-course purchase platforms. This report details Coursify's architecture, features, technology choices, competitive positioning, and future potential.

Chapter 2

Introduction

2.1 Problem Statement & Motivation

The digital education landscape often presents barriers, either through the high cumulative cost of individual courses on large marketplaces or the inherent complexity of feature-rich, enterprise-level LMS platforms. There is a clear demand for a simpler, more financially accessible solution, particularly one offering broad access to knowledge rather than requiring numerous piecemeal purchases. Coursify was conceived and developed to bridge this gap, creating a platform that significantly lowers the financial barrier to entry for diverse learning materials while providing administrators with essential management tools without overwhelming complexity.

2.2 Project Goal & Objectives

Goal: To develop and deliver a functional, secure, and user-friendly MERN-based LMS platform centered around an affordable, all-access subscription model for course delivery.

Objectives:

- Implement robust user authentication (Register, Login, Logout) and role-based authorization (User, Admin).
- Develop full CRUD (Create, Read, Update, Delete) capabilities for courses and their associated lectures, manageable by administrators.
- Integrate a secure and regionally relevant third-party payment gateway (Razorpay) for handling user subscriptions effectively.
- Provide an intuitive administrative dashboard presenting key performance indicators (user count, subscription revenue, monthly sales trends) through clear visualizations.
- Enable users to manage their profiles (view, update, change password) and subscriptions (view status, cancel) seamlessly.
- Establish a clean, intuitive, and responsive user interface for both learners and administrators.

2.3 Scope and Limitations

Scope: The current version encompasses user registration, login/logout, profile viewing and updating, password reset mechanism, course catalog browsing, subscription purchase and cancellation via Razorpay, video lecture viewing for subscribers/admins, an admin dashboard with core analytics, full course and lecture CRUD operations for admins, and a contact mechanism.

Limitations: The platform currently lacks advanced pedagogical features such as integrated quizzes, assignments, certificate generation, community forums, or direct user messaging. Course content is primarily focused on video lectures. While built on scalable technologies, handling extremely large concurrent user loads would necessitate further infrastructure optimization and potentially architectural adjustments. Advanced marketing or instructor-specific tools are outside the current scope.

2.4 Target Audience

- **Learners:** Individuals and students seeking cost-effective access to a wide variety of online courses across different subjects under a single, predictable subscription fee. Particularly attractive to those exploring multiple areas of interest.
- **Administrators:** Individuals, educators, or small organizations requiring a straightforward platform to host and manage their online course offerings, track basic user engagement and revenue, without the significant cost or complexity associated with enterprise LMS solutions or commission-based marketplaces.

Chapter 3

System Architecture

3.1 Architectural Pattern (MERN Stack Overview)

Coursify is built upon the MERN stack, a cohesive ecosystem leveraging JavaScript across all layers:

- **MongoDB:** A NoSQL document database chosen for its schema flexibility, ease of use with JavaScript (BSON), and suitability for evolving data structures. Stores user profiles, course content metadata, lecture details, and payment transaction records.
- **Express.js:** A minimal and unopinionated Node.js web application framework. Defines the backend API routing, handles HTTP request/response cycles, and manages middleware for cross-cutting concerns like authentication and request parsing.
- **React:** A declarative JavaScript library for building dynamic and interactive user interfaces. Used for the frontend single-page application (SPA), enabling reusable components, efficient state management, and a seamless user experience.
- **Node.js:** The JavaScript runtime environment executing the backend (Express.js) code. Its event-driven, non-blocking I/O model is well-suited for handling concurrent connections typical of web applications.

Importance: This architecture promotes rapid development cycles, leverages a unified language (JavaScript), benefits from extensive community support, and facilitates a clear separation between the client-side presentation (React) and server-side logic (Node/Express) via a RESTful API, following MVC-like principles.

3.2 Component Diagram (Conceptual)

The system comprises the following main components:

- **Client (Browser):** Executes the React frontend application. Manages UI rendering, captures user input, handles local state (potentially via Redux), and communicates with the API Server.
- **API Server (Backend):** Runs the Node.js/Express application. Contains business logic, processes API requests, authenticates/authorizes users (JWT), interacts with the MongoDB database, and communicates with external services (Razorpay, Cloudinary).
- **Database (MongoDB):** Persistent storage layer (likely MongoDB Atlas) holding all application data (Users, Courses, Payments).
- **External Services:**
  - **Razorpay:** Securely processes subscription payments.

- **Cloudinary:** (Inferred) Manages storage, optimization, and delivery of media assets (avatars, thumbnails, videos).

Importance: This distributed architecture enhances modularity, maintainability, and allows independent scaling of different system components (frontend, backend, database).

3.3 System Diagrams

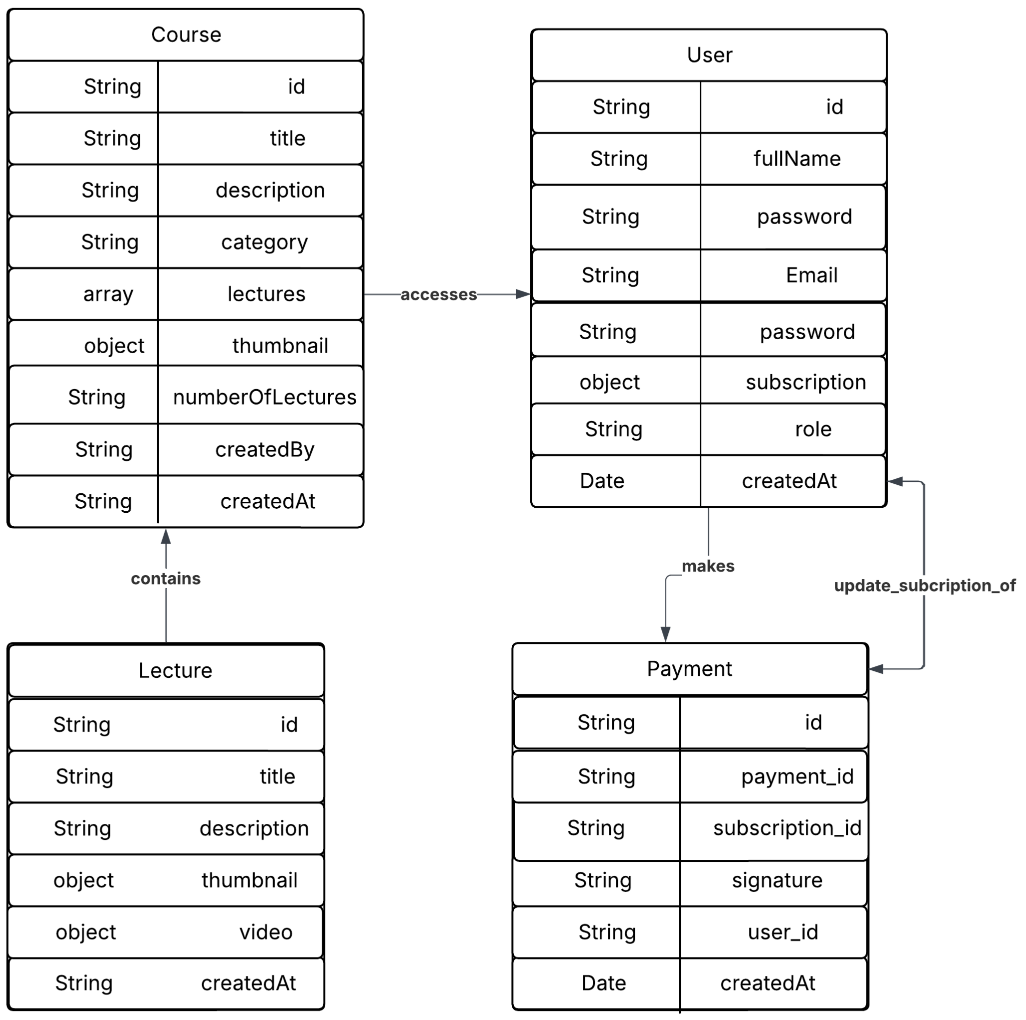

Figure 3.1: Entity-Relationship (ER) Diagram

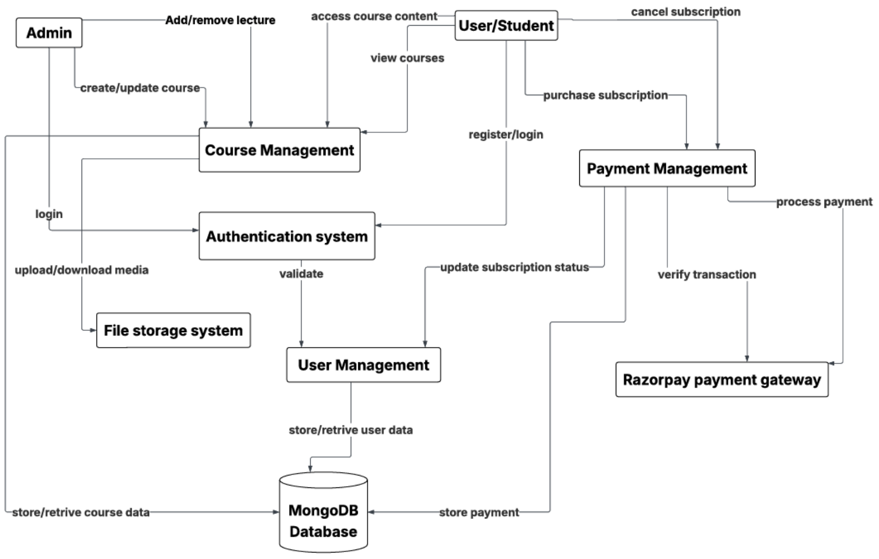

Figure 3.2: Data Flow Diagram (DFD) - Level 1

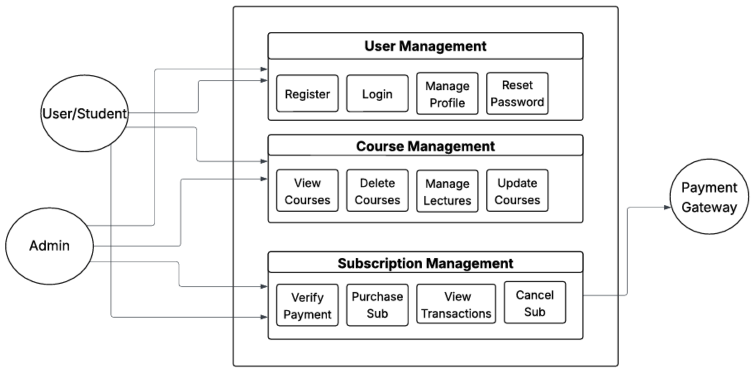

Figure 3.3: Use Case Diagram

Chapter 4

Technology Stack

4.1 Backend Technologies

- **Node.js:** Runtime for executing server-side JavaScript, enabling efficient I/O operations.
- **Express.js:** Web framework for building the RESTful API, handling routing and middleware.
- **MongoDB:** NoSQL document database for flexible and scalable data storage.
- **Mongoose:** Object Data Mapper (ODM) for MongoDB, providing schema definition, validation, and query convenience.

4.2 Frontend Technologies

- **React:** JavaScript library for building the component-based user interface.
- **Vite:** Next-generation frontend tooling providing fast development server and optimized production builds.
- **React Router DOM:** Library for handling client-side routing within the single-page application.
- **Redux Toolkit:** Official, opinionated toolset for efficient Redux state management, simplifying store setup, reducers, and actions.
- **Axios:** Promise-based HTTP client for making API requests from the frontend to the backend.
- **Tailwind CSS:** Utility-first CSS framework for rapid UI development and customization.
- **DaisyUI:** Component library for Tailwind CSS, providing pre-styled components (buttons, cards) to accelerate development.

4.3 Database

MongoDB Atlas (Assumed): Cloud-hosted, managed MongoDB service for scalability, reliability, and ease of management.

4.4 Authentication & Authorization

- **JSON Web Tokens (JWT):** Standard for creating signed, stateless authentication tokens containing user claims (ID, role, subscription).
- **bcryptjs:** Library for securely hashing user passwords before storage, protecting against breaches.

4.5 File Storage

Cloudinary (Inferred): Cloud-based service for managing image and video uploads, storage, optimization, and delivery (CDN).

4.6 Payment Gateway

Razorpay: Integrated payment processing service, popular in India, handling secure subscription transactions (UPI, Cards, etc.).

4.7 Development & Build Tools

- **Vite:** Frontend build tool and dev server.
- **npm (Node Package Manager):** Dependency management for Node.js projects.
- **Git:** Version control system for code management and collaboration.

4.8 Other Libraries

- **Chart.js & React-Chartjs-2:** Libraries for rendering interactive charts (pie, bar) on the admin dashboard.
- **React Icons:** Library providing easy access to a wide variety of icons.
- **React Hot Toast:** Library for displaying non-intrusive notifications (toasts) for user feedback.
- **Multer:** Node.js middleware for handling `multipart/form-data`, essential for processing file uploads.

Chapter 5

Core Modules & Features

5.1 User Module

- **Authentication:** Provides Register, Login, and Logout capabilities. API interactions defined in Section 7.2.
- **Profile Management:** Allows users to view and update their profile information (name, avatar) and change their password. API interactions defined in Section 7.2.
- **Password Recovery:** Enables users to reset forgotten passwords via a secure email token mechanism. API interactions defined in Section 7.2.
- **Course Exploration & Access:** Users can browse the course catalog publicly. Accessing specific course lectures requires authentication and an active subscription (or Admin role). API interaction defined in Section 7.3.
- **Subscription Management:** Users can view their current subscription status within their profile and initiate cancellation if subscribed. API interactions defined in Section 7.4.
- **Contact Us Functionality:** A form allowing users to send inquiries to the platform administrators. API interaction defined in Section 7.5.

5.2 Admin Module

- **Authentication & Profile Management:** Admins log in using the same system but receive an 'ADMIN' role, granting access to restricted areas and functionalities. Profile management is similar to regular users.
- **Admin Dashboard & Analytics:** Provides a visual overview of platform statistics including user counts (Registered vs. Enrolled via Pie Chart), subscription counts, total revenue, and monthly sales trends (via Bar Chart). API interactions defined in Section 7.4 and 7.5.
- **Course Management:** Admins have full CRUD capabilities over courses (Create, Read, Update, Delete). API interactions defined in Section 7.3.
- **Lecture Management:** Admins can add new lectures (including video uploads) to existing courses and delete existing lectures. API interactions defined in Section 7.3.
- **User Management Insights:** Provided primarily through aggregated statistics on the dashboard.

5.3 Course Module

- **Course Structure:** Courses contain metadata (title, description, category, instructor) and learning content (thumbnail, list of lectures). Defined by `course.model.js`.
- **Lecture Structure:** Each lecture within a course has a title, description, and associated video content (hosted externally, linked via URL).
- **Course Listing & Display:** Publicly accessible view displaying available courses, typically using cards showing key information. API interaction defined in Section 7.3.
- **Lecture Viewing & Playback:** Authenticated and subscribed users (or Admins) can view the list of lectures for a course and play the associated videos. API interaction defined in Section 7.3.

5.4 Payment Module

- **Subscription Model:** A simple, fixed-term (1 Year), all-access subscription model at a defined price (499 shown).
- **Razorpay Integration:** Facilitates the subscription purchase flow, from initiating the order to handling the payment via Razorpay's secure checkout interface. API interactions defined in Section 7.4.
- **Payment Verification:** Securely verifies the success and authenticity of payments completed via Razorpay before updating the user's subscription status in the database. API interaction defined in Section 7.4.
- **Subscription Cancellation Logic:** Allows users to request cancellation, triggering backend logic to interact with Razorpay and update local user status. API interaction defined in Section 7.4.

Chapter 6

Database Design

6.1 Schema Overview

Utilizes Mongoose to define schemas for MongoDB collections (Users, Courses, Payments), enforcing structure and enabling validation and middleware.

6.2 User Model (`user.model.js`)

Defines the structure for user documents, including fields like `fullName`, `email` (unique, validated), `password` (hashed, not selected by default), `avatar` (Cloudinary links), `role` ('USER' or 'ADMIN'), `subscription` (status and Razorpay ID), and fields for password reset tokens/expiry. Includes pre-save middleware for password hashing and methods for password comparison, JWT generation, and password reset token generation.

6.3 Course Model (`course.model.js`)

Defines the structure for course documents, including `title`, `description`, `category`, `createdBy`, `thumbnail` (Cloudinary links), `numberOfLectures`, and an embedded array of `lectures`. Each lecture sub-document contains `title`, `description`, and `lecture` (Cloudinary links for video).

6.4 Payment Model (`payment.model.js`)

Defines the structure for recording successful payment transactions, storing `razorpay_payment_id`, `razorpay_subscription_id`, and `razorpay_signature` for verification and auditing.

6.5 Relationships between Models

User-Subscription is direct embedding in the User model. Course-Lectures use embedding (lectures within course). User-Payment is linked implicitly via the `razorpay_subscription_id` present in both the Payment record and the User's subscription details.

Chapter 7

API Design (Routes & Controllers)

7.1 RESTful API Principles

The API adheres to REST conventions, using standard HTTP methods (GET, POST, PUT, DELETE), resource-oriented URLs, and JSON for data exchange.

7.2 User Routes (`user.routes.js`) - API Contract

- `POST /api/v1/user/register`: **Input:** `multipart/form-data` {`fullName`, `email`, `password`, `avatar`?}. **Output:** 201 User object | 400 Error.
- `POST /api/v1/user/login`: **Input:** JSON {`email`, `password`}. **Output:** 200 User object & sets auth token | 401 Error.
- `POST /api/v1/user/logout`: **Input:** Auth required. **Output:** 200 Confirmation & clears auth token.
- `GET /api/v1/user/me`: **Input:** Auth required. **Output:** 200 User object | 401 Error.
- `POST /api/v1/user/reset`: **Input:** JSON {`email`}. **Output:** 200 Confirmation.
- `POST /api/v1/user/reset/:resetToken`: **Input:** URL param `:resetToken`, JSON {`password`}. **Output:** 200 Confirmation | 400 Error.
- `POST /api/v1/user/change-password`: **Input:** Auth required, JSON {`oldPassword`, `newPassword`}. **Output:** 200 Confirmation | 400/401 Error.
- `PUT /api/v1/user/update/:id`: **Input:** Auth required, URL param `:id`, `multipart/form-data` {fields?, `avatar`?}. **Output:** 200 Updated User object | 400/401/403/404 Error.

7.3 Course Routes (`course.routes.js`) - API Contract

- `GET /api/v1/courses`: **Input:** None. **Output:** 200 Array of Course summaries.
- `POST /api/v1/courses`: **Input:** Admin Auth required, `multipart/form-data` {`title`, `description`, `category`, `createdBy`, `thumbnail`}. **Output:** 201 New Course object | 400/401/403 Error.
- `DELETE /api/v1/courses` *(Needs clarification - Assuming lecture delete intent)*: **Input:** Admin Auth required, likely needs course/lecture IDs. **Output:** 200/204 Success | 401/403/404 Error.

- `GET /api/v1/courses/:id`: **Input:** Subscriber/Admin Auth required, URL param `:id`. **Output:** 200 Course object with lectures | 401/403/404 Error.
- `POST /api/v1/courses/:id`: **Input:** Admin Auth required, URL param `:id`, `multipart/form-data` `{title?, description?, lecture}`. **Output:** 200 Updated Course object | 400/401/403/404 Error.
- `PUT /api/v1/courses/:id`: **Input:** Admin Auth required, URL param `:id`, JSON {fields to update}. **Output:** 200 Updated Course object | 400/401/403/404 Error.
- `DELETE /api/v1/courses/:id`: **Input:** Admin Auth required, URL param `:id`. **Output:** 200/204 Success | 401/403/404 Error.

7.4 Payment Routes (`payment.routes.js`) - API Contract

- `POST /api/v1/payment/subscribe`: **Input:** Auth required. **Output:** 200 JSON `{subscription_id}` | 400/401 Error.
- `POST /api/v1/payment/verify`: **Input:** Auth required, JSON `{razorpay_payment_id, razorpay_subscription_id, razorpay_signature}`. **Output:** 200 Confirmation | 400/401 Error.
- `POST /api/v1/payment/unsubscribe`: **Input:** Subscriber Auth required. **Output:** 200 Confirmation | 400/401/403 Error.
- `GET /api/v1/payment/razorpay-key`: **Input:** Auth required. **Output:** 200 JSON `{key}` | 401 Error.
- `GET /api/v1/payment`: **Input:** Admin Auth required. **Output:** 200 Array of Payment objects | 401/403 Error.

7.5 Miscellaneous Routes (`miscellaneous.routes.js`) - API Contract

- `POST /api/v1/contact`: **Input:** JSON `{name, email, message}`. **Output:** 200 Confirmation | 400 Error.
- `GET /api/v1/admin/stats/users`: **Input:** Admin Auth required. **Output:** 200 JSON {user stats} | 401/403 Error.

7.6 Middleware Implementation

Express middleware is used extensively for cross-cutting concerns:

- `isLoggedIn`: Verifies JWT for authentication before allowing access to protected routes.
- `authorizeRoles(...roles)`: Enforces Role-Based Access Control (RBAC), ensuring only users with specified roles (e.g., 'ADMIN') can access certain endpoints.
- `authorizeSubscribers`: Specifically grants access to content/features for users with an active subscription or Admin role.
- `multer`: Handles `multipart/form-data` requests, enabling file uploads for avatars, thumbnails, and lectures.

Chapter 8

Frontend Implementation

8.1 Project Structure

Adheres to standard React practices, likely organizing code into components, pages/views, services/API handlers, Redux store/slices, utility functions, hooks, and assets for maintainability.

8.2 State Management (Redux Toolkit)

Centralizes application state (e.g., user authentication status, profile data, course lists, admin statistics). Components interact with the store via `useSelector` (to read state) and `useDispatch` (to trigger state updates via actions/reducers), ensuring predictable data flow.

8.3 Routing (React Router DOM)

Enables navigation between different sections/pages of the single-page application without full page reloads, using defined routes that map URLs to specific page components. Protected routes likely ensure only authenticated/authorized users can access certain pages.

8.4 API Integration (Axios)

A dedicated service layer likely uses Axios to handle all HTTP communication with the backend API, abstracting data fetching logic from UI components and potentially using interceptors to automatically attach JWTs to requests.

8.5 Component Library & Styling (Tailwind CSS, DaisyUI)

Leverages Tailwind's utility-first classes for direct styling within JSX. DaisyUI provides higher-level component classes (like `btn`, `card`) built on Tailwind, speeding up the development of common UI elements while maintaining customization capabilities.

8.6 Data Visualization (Chart.js, React-Chartjs-2)

Renders interactive pie and bar charts on the Admin Dashboard, effectively visualizing user and sales data fetched from the backend.

Chapter 9

Security Aspects

9.1 Authentication (JWT Strategy)

Employs signed JSON Web Tokens for stateless authentication. Secure handling and storage of tokens on the client-side (ideally `httpOnly` cookies) and server-side verification are crucial.

9.2 Authorization (Role-Based & Subscription-Based)

Uses custom Express middleware (`authorizeRoles`, `authorizeSubscribers`) to strictly enforce access controls based on user role and subscription status, protecting sensitive data and functionalities.

9.3 Password Security (Hashing with `bcryptjs`)

Passwords are never stored in plain text. They are securely hashed using bcrypt with salting via Mongoose middleware before database insertion. Comparison happens via bcrypt's compare function.

9.4 Input Validation

Relies on Mongoose schema validation and should incorporate robust server-side validation within controllers or dedicated middleware to sanitize inputs and prevent invalid data or potential injection attempts. Frontend validation provides user feedback but isn't a security measure.

9.5 Secure File Uploads

Offloads file storage and delivery to Cloudinary (inferred), minimizing direct server load and security risks associated with handling user-uploaded files. Multer likely performs initial validation.

9.6 Payment Security (Delegated to Razorpay)

Securely handles payments by delegating sensitive card/account data entry and processing to Razorpay's PCI DSS compliant infrastructure. Backend verification uses secure signature comparison.

Chapter 10

Challenges Faced & Solutions

- **Backend Challenge:** Implementing secure role-based and subscription-based authorization across various API endpoints. **Solution:** Developed reusable custom middleware functions (`isLoggedIn`, `authorizeRoles`, `authorizeSubscribers`) in Express. Chaining these middleware provided fine-grained, declarative access control.
- **Backend Challenge:** Securely handling file uploads without overloading the backend server. **Solution:** Used `multer` middleware for initial processing and immediately offloaded files to Cloudinary within controller logic, storing only references (URLs/IDs) locally.
- **Backend Challenge:** Handling asynchronous operations correctly, especially during payment verification involving external API calls and multiple database updates. **Solution:** Consistently used `async/await` syntax with comprehensive `try...catch` blocks for robust error handling and ensuring data consistency.
- **Frontend Challenge:** Managing global application state effectively (e.g., authentication status, user profile, course data) across many components. **Solution:** Implemented Redux Toolkit for a centralized store, using slices, reducers, and actions for predictable state updates, accessed via `useSelector` and `useDispatch` hooks.
- **Frontend Challenge:** Ensuring consistent UI and handling responsiveness across different components and screen sizes using utility-based styling. **Solution:** Leveraged Tailwind CSS for base styling and DaisyUI for pre-built components, establishing consistent design tokens and patterns. Used Tailwind's responsive modifiers (`md:`, `lg:`) extensively.
- **Frontend Challenge:** Efficiently handling API loading states, displaying feedback (loading indicators, success/error messages), and preventing race conditions. **Solution:** Integrated API call logic within Redux thunks or custom hooks, managing loading/error states within the Redux store. Used `React Hot Toast` for user notifications and implemented checks to avoid redundant API calls.
- **Cross-Cutting Challenge:** Debugging CORS (Cross-Origin Resource Sharing) issues between the React frontend and Express backend during development. **Solution:** Configured the `cors` middleware in the Express application to explicitly allow requests from the frontend's development origin (`http://localhost:5173`).

Chapter 11

Conclusion and Future Enhancements

11.1 Conclusion

Coursify stands as a successfully implemented Learning Management System built with the MERN stack, delivering on its core objectives. It provides secure user management, efficient course and lecture administration, integrated payments via Razorpay, and valuable admin analytics. The platform's most compelling feature and strategic advantage is its highly affordable, all-access subscription model, positioning it as a strong value proposition against per-course competitors. Coursify currently provides a solid, functional, and focused solution for accessible online education. The project demonstrates proficient application of modern web development technologies and addresses a distinct need within the e-learning market.

11.2 Future Enhancements

While the current version meets the primary goals, several enhancements could further strengthen Coursify's offering:

- **Learning & Engagement:** Introduce Quizzes, Assignments, Certificates, Discussion Forums, User Progress Tracking. *Value: Deeper learning, validation, community, motivation.*
- **Content Administration:** Add support for diverse content types (PDFs, Text), develop Advanced Admin Analytics (course popularity, engagement metrics), implement an Instructor Role for multi-creator capability. *Value: Richer content, better insights, platform growth.*
- **Platform & Reach:** Develop Mobile Applications (iOS/Android), add Multi-language Support, integrate with external tools (Calendars, Video Conferencing), implement Drip Content functionality. *Value: Accessibility, wider audience, expanded features.*

Implementing these features would build upon the strong foundation and move towards a more comprehensive and engaging platform.