Prompts for the GenAI project

BACKEND Prompts:

* Generate a Spring Boot microservices-based project setup with 4 modules: course-service, purchase-service, api-gateway, and eureka-server. Use Maven for build management.
* Configure parent POM with common dependencies: Spring Web, Spring Data JPA, MySQL Driver, Lombok, Validation, Spring Boot Actuator.
* Create a Eureka Server microservice named eureka-server with @EnableEurekaServer.
* Configure application.properties for Eureka server to run on port 8761.
* Add dependencies for Eureka Server in pom.xml.
* Create a new Spring Boot microservice named course-service and register it with Eureka.
* Configure MySQL database connection in application.properties for course-service.
* Create an entity class Course with fields: id, title, description, price, thumbnailUrl.
* Create a repository interface CourseRepository extending JpaRepository.
* Create a service interface CourseService and implementation CourseServiceImpl for CRUD operations.
* Create a REST controller CourseController with endpoints:

POST /courses/add,

GET /courses/{id} ,

GET /courses/all,

PUT /courses/update/{id},

DELETE /courses/delete/{id}

* Integrate Cloudinary for thumbnail image upload in course-service.
* Add global exception handling for invalid course requests.
* Write unit tests for CourseServiceImpl using Mockito.
* Write controller tests for CourseController.
* Create a new Spring Boot microservice named purchase-service and register it with Eureka.
* Configure MySQL database connection in application.properties for purchase-service.
* Create an entity Purchase with fields: id, userId, courseId, paymentId, orderId, status, purchaseDate.
* Create repository PurchaseRepository extending JpaRepository.
* Create service interface PurchaseService and implementation PurchaseServiceImpl for saving and retrieving purchases.
* Integrate Razorpay for payment gateway in purchase-service.
* Create service PaymentService to generate Razorpay orders.
* Create REST controller PurchaseController with endpoints:
* POST /purchase/create-payment (create Razorpay order)
* POST /purchase/payment-success (save successful payment + course mapping)
* GET /purchase/user/{userId} (fetch all purchases by a user)
* Add exception handling for failed or duplicate payments.
* Write unit tests for PaymentService.
* Write integration test for PurchaseController.
* Create a Spring Boot microservice named api-gateway using Spring Cloud Gateway.
* Configure application.properties with routes for course-service and purchase-service.
* Register api-gateway with Eureka.
* Implement a basic authentication and authorization filter in the API Gateway.
* Add global exception handling for failed routes in the API Gateway.
* Write unit tests for API Gateway filters.
* Add OpenFeign dependency in api-gateway.
* Create Feign client in api-gateway for communicating with course-service.
* Create Feign client in api-gateway for communicating with purchase-service.
* Implement a purchase flow: API Gateway calls Purchase Service → Purchase Service verifies Course Service for course existence → Razorpay order created.
* Configure JWT-based authentication in api-gateway.
* Add user roles for ADMIN and USER.
* Secure course creation endpoints for ADMIN only.
* Secure purchase endpoints for logged-in USER.
* Configure logging using Logback and send logs per microservice.
* Add centralized error response format for all microservices.
* Add validation annotations in Course and Purchase entities (e.g., @NotNull, @Size).
* Write integration test for course creation + purchase flow end-to-end.
* Test Razorpay webhook for verifying payment status.
* Test API Gateway routes with Postman collection.
* Generate Swagger/OpenAPI documentation for all microservices.

Angular Frontend Prompts:

* Generate a new Angular project named online-course-frontend using the Angular CLI. Enable Angular routing during setup and make sure the project uses the latest Angular version supported by CLI.
* Inside the new Angular project, configure environment files (environment.ts and environment.prod.ts) where I can store the API Gateway base URL (for dev and prod). Show me how to structure the environment object properly.
* Add all the required Angular modules like HttpClientModule, FormsModule, and ReactiveFormsModule in app.module.ts so that the app supports forms and API requests.
* Install any necessary dependencies for handling Razorpay checkout integration. Show me exactly what command to run and how to import it.
* Create a basic folder structure under src/app with feature folders like auth, course, purchase, dashboard, and a shared module for reusable components.
* Generate a shared module where I can keep common components like header, footer, and utilities. Make sure to export them for reuse.
* Create a global navbar component with links for Home, Courses, Dashboard, Login, and Register. Show me how to highlight the active route.
* Add a footer component with simple text (like © Online Course App) and ensure it stays at the bottom.
* Configure the app-routing.module.ts so that each module (auth, course, purchase, dashboard) is lazy loaded. Show me a sample lazy loading route.
* Create a global error handler service that catches all HTTP errors and displays them using an alert or notification system.
* Generate the login and register components inside an auth module. In login, add fields for email and password; in register, add fields for name, email, and password.
* Create an AuthService that communicates with backend APIs (/login and /register) through the API Gateway. Show me how to send a POST request using Angular’s HttpClient.
* After successful login, store the JWT token inside localStorage so it persists across reloads. Show me how to also extract and save the user’s role.
* Build an AuthGuard that checks if a user is logged in before accessing routes like dashboard and purchase. Redirect to login if unauthorized.
* Update the navbar so that when the user is logged in, it shows “Logout” and “Dashboard” instead of “Login” and “Register.”
* Create a CourseListComponent that fetches and displays all available courses from the course-service. Show them in a grid with course title, description, price, and thumbnail.
* Build a CourseDetailComponent that shows full details of a single course when clicked from the list, including a button to purchase it.
* Add search and filter functionality in CourseListComponent so users can filter by price or keyword.
* Create an AddCourseComponent for admin where new courses can be added, including uploading a thumbnail image using multipart/form-data.
* Implement a ManageCoursesComponent for admin with options to edit and delete courses. Show me how to use Angular’s ngFor and click events for delete buttons.
* Generate a PurchaseService that communicates with purchase-service APIs for creating payment orders and fetching purchase history.
* Create a CheckoutComponent where a user can click “Buy Now,” and it will call backend /create-payment API to generate a Razorpay order.
* Integrate Razorpay checkout in CheckoutComponent. Show me how to open Razorpay’s payment modal and handle success/failure callbacks.
* On successful payment, call the /payment-success API in backend and save the purchase details. Update the user’s dashboard after purchase.
* Create a PurchaseHistoryComponent where users can see a table of all their purchased courses with orderId, payment status, and course name.
* Create a UserDashboardComponent that displays the user’s profile details and list of purchased courses. Fetch these from the backend.
* Add an AdminDashboardComponent that shows admin-only features like total users, total courses, and total purchases.
* Implement role-based navigation in the dashboard: users see purchased courses, admins see management features.
* Create a ProfileComponent where users can update their details like name, email, and password.
* Add API calls to display statistics on the admin dashboard, like number of purchases, using simple cards or charts.
* Create a reusable CourseCardComponent that displays course thumbnail, name, and price, and reuse it inside course list and dashboard.
* Build a NotificationService that shows messages (success, error, info) in a styled alert box whenever actions succeed or fail.
* Add a custom Angular pipe to truncate long course descriptions to a fixed length with “Read more…” option.
* Implement a loading spinner component and show it globally during API requests.
* Create a centralized ApiService where I can keep all API endpoint URLs in one place and import them everywhere.
* Configure all routes in app-routing.module.ts so that /courses opens course list, /dashboard opens dashboard, and /checkout opens purchase page.
* Apply AuthGuard to checkout and dashboard routes so only logged-in users can access them.
* Create an AdminGuard that checks the role of logged-in user before accessing admin routes like add or manage courses.
* Redirect unauthenticated users automatically to login whenever they try to access restricted pages.
* Set up route lazy loading for better performance so each feature module loads only when needed.
* Update AuthService to connect login and signup requests via API Gateway endpoints.
* Connect CourseService to fetch all courses from the course-service through the API Gateway.
* Add support in CourseService to send multipart/form-data requests for course creation with thumbnail upload.
* Connect PurchaseService to backend Razorpay /create-payment API for generating orders.
* Add API calls to fetch purchased courses for a logged-in user and display them in dashboard and purchase history.
* Write responsive CSS so that course cards adjust in a grid on desktop and stack on mobile screens.
* Add form validations for login, signup, and add course forms (like email format, required fields, min password length).
* Configure Razorpay modal properly with keyId, description, and callbacks for success and failure events.
* Implement an HTTP interceptor to attach the JWT token to every API request automatically.
* Build the project for production using ng build --prod and explain how to deploy it with the backend services.