Models:

public class User {  
 @Id  
 String id;  
  
 String FirstName;  
 String LastName;  
 String email;  
 String Password;  
 Role role;

public class Course {  
  
 @Id  
 private String id;  
  
 private String title;  
 private String description;  
 private Double price;  
 private String contentUrl; // link to video/pdf/resource  
}

public class Order {  
  
 @Id  
 private String id;  
  
 private String userId; // email or user \_id — your choice  
 private Course course;  
 private PaymentStatus paymentStatus;  
  
 private String paymentId; // Stripe PaymentIntent ID  
 private String clientSecret; // Stripe client secret for frontend confirmation  
  
 @Builder.Default  
 private Date createdAt = new Date();  
}

DTOS:

public class SignupRequest {  
 private String firstName;  
 private String lastName;  
 private String email;  
 private String password;  
 private Role role;  
}

public class SignupResponse {  
 private String id;  
 private String firstName;  
 private String lastName;  
 private String email;  
 private Role role;  
 private String message; // e.g. "Signup successful"  
}

public class LoginRequest {  
 private String email;  
 private String password;  
}

public class LoginResponse {  
 private String message;  
 String token; // optional, if using JWT  
 private String email; // optional  
}

public class CourseRequestDto {  
 private String title;  
 private String description;  
 private Double price;  
 private String contentUrl;  
}

public class CourseResponseDto {  
 private String id; // MongoDB generated ID  
 private String title;  
 private String description;  
 private Double price;  
 private String contentUrl;  
}

public class OrderRequestDto {  
 private String courseId; // user ID comes from JWT, so no need to send from client  
}

public class OrderResponseDto {  
 private String id;  
 private String userId; // userId  
 private Course course; // full course object  
 private PaymentStatus paymentStatus;  
 private String paymentId; // Stripe PaymentIntent ID  
 private String clientSecret; // Stripe client secret  
 private Date createdAt;  
}

**SERVICES:**

**public class AuthService {  
  
 private final UserRepo userRepo;  
 private final PasswordEncoder passwordEncoder;  
 private final JwtUtils jwtUtils;  
  
 public SignupResponse signup(SignupRequest request) {  
 // 1. Check if user already exists  
 if (userRepo.existsByEmail(request.getEmail())) {  
 // Throw exception with 409 Conflict  
 throw new ResponseStatusException(  
 HttpStatus.*CONFLICT*,  
 "Email already exists!"  
 );  
 }  
  
 // 2. Create new user  
 User user = new User();  
 user.setFirstName(request.getFirstName());  
 user.setLastName(request.getLastName());  
 user.setEmail(request.getEmail());  
 user.setPassword(passwordEncoder.encode(request.getPassword())); // ⚠️ hash this in real apps!  
 user.setRole(request.getRole());  
 // 3. Save to DB  
 User savedUser = userRepo.save(user);  
  
 // 4. Return response  
 return new SignupResponse(  
 savedUser.getId(),  
 savedUser.getFirstName(),  
 savedUser.getLastName(),  
 savedUser.getEmail(),  
 savedUser.getRole(),  
 "Signup successful"  
 );  
 }  
  
 // ------------------- LOGIN -------------------  
 public LoginResponse login(LoginRequest request) {  
 // 1. Find user by email  
 Optional<User> userOptional = userRepo.findByEmail(request.getEmail());  
  
 if (userOptional.isEmpty()) {  
 throw new ResponseStatusException(  
 HttpStatus.*UNAUTHORIZED*,  
 "Invalid email or password"  
 );  
 }  
  
 User user = userOptional.get();  
  
 // 2. Check password  
 if (!passwordEncoder.matches(request.getPassword(), user.getPassword())) {  
 throw new ResponseStatusException(HttpStatus.*UNAUTHORIZED*, "Invalid email or password");  
 }  
  
 // 3. Generate JWT token based on email  
 String token = jwtUtils.generateToken(user.getEmail(), String.*valueOf*(user.getRole()));  
  
 // 4. Return login response with token  
 return new LoginResponse(  
 "Login successful",  
 token,  
 user.getEmail()  
 );  
 }  
  
}**

**public class CourseService {  
  
 private final CourseRepository courseRepository;  
  
 // Create new course  
 public CourseResponseDto createCourse(CourseRequestDto request) {  
 Course course = new Course();  
 course.setTitle(request.getTitle());  
 course.setDescription(request.getDescription());  
 course.setPrice(request.getPrice());  
 course.setContentUrl(request.getContentUrl());  
  
 Course saved = courseRepository.save(course);  
  
 return mapToDto(saved);  
 }  
  
 // Get all courses  
 public List<CourseResponseDto> getAllCourses() {  
 return courseRepository.findAll()  
 .stream()  
 .map(this::mapToDto)  
 .collect(Collectors.*toList*());  
 }  
  
 // Get course by ID  
 public CourseResponseDto getCourseById(String id) {  
 Optional<Course> courseOpt = courseRepository.findById(id);  
 return courseOpt.map(this::mapToDto)  
 .orElseThrow(() -> new RuntimeException("Course not found with id: " + id));  
 }  
  
 // Delete course  
 public void deleteCourse(String id) {  
 if (!courseRepository.existsById(id)) {  
 throw new RuntimeException("Course not found with id: " + id);  
 }  
 courseRepository.deleteById(id);  
 }  
  
 // 🔄 Mapper  
 private CourseResponseDto mapToDto(Course course) {  
 return new CourseResponseDto(  
 course.getId(),  
 course.getTitle(),  
 course.getDescription(),  
 course.getPrice(),  
 course.getContentUrl()  
 );  
 }  
}**

**public class OrderService {  
  
 private final OrderRepository orderRepository;  
 private final UserRepo userRepository;  
 private final CourseRepository courseRepository;  
  
 @Value("${stripe.api.key}")  
 private String stripeApiKey;  
  
 // ✅ Create a new order with Stripe payment  
 public OrderResponseDto createOrder(String userEmail, OrderRequestDto request) throws StripeException {  
 Stripe.*apiKey* = stripeApiKey;  
  
 User user = userRepository.findByEmail(userEmail)  
 .orElseThrow(() -> new RuntimeException("User not found"));  
  
 Course course = courseRepository.findById(request.getCourseId())  
 .orElseThrow(() -> new RuntimeException("Course not found"));  
  
 // Prevent duplicate purchases  
 boolean alreadyPurchased = orderRepository.findAll().stream()  
 .anyMatch(order ->  
 order.getUserId().equals(user.getId()) &&  
 order.getCourse().getId().equals(course.getId()) &&  
 order.getPaymentStatus() == PaymentStatus.*SUCCESS* );  
  
 if (alreadyPurchased) {  
 throw new RuntimeException("You have already purchased this course.");  
 }  
  
 // 1️⃣ Create Stripe PaymentIntent  
 PaymentIntentCreateParams params =  
 PaymentIntentCreateParams.*builder*()  
 .setAmount((long) (course.getPrice() \* 100)) // amount in cents  
 .setCurrency("usd")  
 .build();  
  
 PaymentIntent paymentIntent = PaymentIntent.*create*(params);  
  
 // 2️⃣ Save order with PAYMENT\_PENDING status  
 Order order = Order.*builder*()  
 .userId(user.getId())  
 .course(course)  
 .paymentStatus(PaymentStatus.*PENDING*)  
 .paymentId(paymentIntent.getId())  
 .clientSecret(paymentIntent.getClientSecret())  
 .createdAt(new Date())  
 .build();  
  
 orderRepository.save(order);  
  
 // 3️⃣ Return order info including clientSecret  
 return mapToDto(order, user.getId());  
 }  
  
 // Get all orders of a specific user  
 public List<OrderResponseDto> getMyOrders(String userEmail) {  
 User user = userRepository.findByEmail(userEmail)  
 .orElseThrow(() -> new RuntimeException("User not found"));  
  
 return orderRepository.findAll().stream()  
 .filter(order -> order.getUserId().equals(user.getId()))  
 .map(order -> mapToDto(order, order.getUserId()))  
 .collect(Collectors.*toList*());  
 }  
  
 // Get all orders (admin)  
 public List<OrderResponseDto> getAllOrders() {  
 return orderRepository.findAll().stream()  
 .map(order -> mapToDto(order, order.getUserId()))  
 .collect(Collectors.*toList*());  
 }  
  
 // Get a specific order by ID  
 public OrderResponseDto getOrderById(String orderId, String userEmail, boolean isAdmin) {  
 Order order = orderRepository.findById(orderId)  
 .orElseThrow(() -> new RuntimeException("Order not found"));  
  
 User user = userRepository.findById(order.getUserId())  
 .orElseThrow(() -> new RuntimeException("User not found"));  
  
 if (!isAdmin && !user.getEmail().equals(userEmail)) {  
 throw new RuntimeException("Access denied");  
 }  
  
 return mapToDto(order, order.getUserId());  
 }  
  
 // Cancel an order  
 public void cancelOrder(String orderId, String userEmail, boolean isAdmin) {  
 Order order = orderRepository.findById(orderId)  
 .orElseThrow(() -> new RuntimeException("Order not found"));  
  
 User user = userRepository.findById(order.getUserId())  
 .orElseThrow(() -> new RuntimeException("User not found"));  
  
 if (!isAdmin && !user.getEmail().equals(userEmail)) {  
 throw new RuntimeException("Access denied");  
 }  
  
 order.setPaymentStatus(PaymentStatus.*CANCELLED*);  
 orderRepository.save(order);  
 }  
  
 // Mapper utility  
 private OrderResponseDto mapToDto(Order order, String userId) {  
 return OrderResponseDto.*builder*()  
 .id(order.getId())  
 .userId(userId)  
 .course(order.getCourse())  
 .paymentStatus(order.getPaymentStatus())  
 .paymentId(order.getPaymentId())  
 .clientSecret(order.getClientSecret())  
 .createdAt(order.getCreatedAt())  
 .build();  
 }  
}**

**public class PaymentService {  
  
 private final OrderRepository orderRepository;  
  
 @Value("${stripe.api.key}")  
 private String stripeApiKey;  
  
  
  
 public void confirmPaymentWithToken(String orderId, String token) {  
 Stripe.*apiKey* = stripeApiKey;  
  
 try {  
 // 1. Retrieve the order  
 Order order = orderRepository.findById(orderId)  
 .orElseThrow(() -> new RuntimeException("Order not found"));  
  
 // 2. Confirm PaymentIntent using the test token  
 PaymentIntent paymentIntent = PaymentIntent.*retrieve*(order.getPaymentId());  
  
 PaymentIntent confirmedIntent = paymentIntent.confirm(  
 PaymentIntentConfirmParams.*builder*()  
 .setPaymentMethod(token) // Use the test token here  
 .build()  
 );  
  
 // 3. Update order status  
 if ("succeeded".equals(confirmedIntent.getStatus())) {  
 order.setPaymentStatus(PaymentStatus.*SUCCESS*);  
 } else {  
 order.setPaymentStatus(PaymentStatus.*FAILED*);  
 }  
  
 orderRepository.save(order);  
  
 } catch (StripeException e) {  
 throw new RuntimeException("Stripe payment failed: " + e.getMessage(), e);  
 }  
 }**

**CONTROLLERS:**

**public class AuthController {  
  
 private final AuthService authService;  
  
 // ✅ Signup endpoint  
 @PostMapping("/signup")  
 public ResponseEntity<SignupResponse> signup(@RequestBody SignupRequest request) {  
 SignupResponse response = authService.signup(request);  
 return ResponseEntity.*ok*(response);  
 }  
  
 // ---------------- LOGIN ----------------  
 @PostMapping("/login")  
 public ResponseEntity<LoginResponse> login(@RequestBody LoginRequest request) {  
 // Call AuthService login method  
 LoginResponse response = authService.login(request);  
 return ResponseEntity.*ok*(response);  
  
 }  
  
 @GetMapping("/users")  
 public String getAllUsers() {  
 return "Successfully checked";  
 }  
}**

**public class CourseController {  
  
 private final CourseService courseService;  
  
 // ✅ List all courses (any authenticated user)  
 @GetMapping  
 public ResponseEntity<List<CourseResponseDto>> getAllCourses() {  
 return ResponseEntity.*ok*(courseService.getAllCourses());  
 }  
  
 // ✅ Get course details  
 @GetMapping("/{id}")  
 public ResponseEntity<CourseResponseDto> getCourseById(@PathVariable String id) {  
 return ResponseEntity.*ok*(courseService.getCourseById(id));  
 }  
  
 // ✅ Add new course (only ADMIN)  
 @PostMapping  
 @PreAuthorize("hasRole('ADMIN')")  
 public ResponseEntity<CourseResponseDto> createCourse(@RequestBody CourseRequestDto request) {  
 return ResponseEntity.*ok*(courseService.createCourse(request));  
 }  
  
 // ✅ Delete course (only ADMIN)  
 @DeleteMapping("/{id}")  
 @PreAuthorize("hasRole('ADMIN')")  
 public ResponseEntity<Void> deleteCourse(@PathVariable String id) {  
 courseService.deleteCourse(id);  
 return ResponseEntity.*noContent*().build();  
 }  
}**

**public class OrderController {  
  
 private final OrderService orderService;  
  
 // ✅ User creates an order (buy course)  
 @PostMapping  
 @PreAuthorize("hasRole('USER')")  
 public ResponseEntity<OrderResponseDto> createOrder(  
 Authentication authentication,  
 @RequestBody OrderRequestDto request) throws StripeException {  
 String userEmail = authentication.getName();  
 OrderResponseDto orderResponse = orderService.createOrder(userEmail, request);  
 // clientSecret is returned in orderResponse for frontend payment confirmation  
 return ResponseEntity.*ok*(orderResponse);  
 }  
  
 // ✅ User fetches only their orders  
 @GetMapping("/my")  
 @PreAuthorize("hasRole('USER')")  
 public ResponseEntity<List<OrderResponseDto>> getMyOrders(Authentication authentication) {  
 String userEmail = authentication.getName();  
 return ResponseEntity.*ok*(orderService.getMyOrders(userEmail));  
 }  
  
 // ✅ Admin can view all orders  
 @GetMapping  
 @PreAuthorize("hasRole('ADMIN')")  
 public ResponseEntity<List<OrderResponseDto>> getAllOrders() {  
 return ResponseEntity.*ok*(orderService.getAllOrders());  
 }  
  
 // ✅ User/Admin fetch specific order  
 @GetMapping("/{id}")  
 @PreAuthorize("hasAnyRole('USER','ADMIN')")  
 public ResponseEntity<OrderResponseDto> getOrderById(  
 @PathVariable String id,  
 Authentication authentication) {  
 String userEmail = authentication.getName();  
 boolean isAdmin = authentication.getAuthorities().stream()  
 .anyMatch(auth -> auth.getAuthority().equals("ROLE\_ADMIN"));  
  
 return ResponseEntity.*ok*(orderService.getOrderById(id, userEmail, isAdmin));  
 }  
  
 // ✅ User cancels their order / Admin cancels any order  
 @DeleteMapping("/{id}")  
 @PreAuthorize("hasAnyRole('USER','ADMIN')")  
 public ResponseEntity<String> cancelOrder(  
 @PathVariable String id,  
 Authentication authentication) {  
 String userEmail = authentication.getName();  
 boolean isAdmin = authentication.getAuthorities().stream()  
 .anyMatch(auth -> auth.getAuthority().equals("ROLE\_ADMIN"));  
  
 orderService.cancelOrder(id, userEmail, isAdmin);  
 return ResponseEntity.*ok*("Order cancelled successfully");  
 }  
  
}**

**public class PaymentController {  
  
 private final PaymentService paymentService;  
  
 */\*\*  
 \* Confirm payment for an existing order using Stripe test token.  
 \* Frontend sends JSON: { "token": "tok\_visa" }  
 \*/* @PostMapping("/confirm/{orderId}")  
 @PreAuthorize("hasRole('USER')")  
 public ResponseEntity<String> confirmPayment(  
 @PathVariable String orderId,  
 @RequestBody Map<String, String> request  
 ) throws StripeException {  
 String token = request.get("token"); // e.g., "tok\_visa"  
 paymentService.confirmPaymentWithToken(orderId, token);  
 return ResponseEntity.*ok*("Payment confirmed and order updated");  
 }  
}**