Back end Code:

```
package com.bartr.controller;

import com.bartr.model.User;
import com.bartr.service.impl.UserServiceImpl;
import lombok.AllArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;

@RestController
@AllArgsConstructor
public class AuthController {
   private final UserServiceImpl userService;

   @PostMapping("/login")
   public ResponseEntity<?> userLogin(@RequestBody User user){
      Map<String,String> response = new HashMap<>();
      response.put("token", userService.jwtLogin(user));
//      response.put(, userService.jwtLogin(user));
      return new ResponseEntity<>(response, HttpStatus.OK);
   }

   @GetMapping("/me")
   public ResponseEntity<?> getMyUsername(){
      String username = SecurityContextHolder.getContext().getAuthentication().getName();
//      Optional<User> user = userService.getUserByUsername(username);
      return ResponseEntity.ok(username);
   }

}


package com.bartr.controller;


import java.util.List;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import com.bartr.model.Category;
import com.bartr.service.CategoryService;

import lombok.RequiredArgsConstructor;

@CrossOrigin("http://localhost:4200/")
@RestController
@RequestMapping("/api/categories")
public class CategoryController {

  private final CategoryService categoryService;

  public CategoryController(CategoryService categoryService){
    this.categoryService = categoryService;
  }

  //This should be accessible without Login Also.
  @GetMapping("")
  public ResponseEntity<List<Category>> getAllCategories(){
    List<Category> categories = categoryService.getAllCategories();
    return ResponseEntity.ok(categories);
  }

  //Only admin will have access for that
  @PostMapping("insertCategory")
  public ResponseEntity<Category> createCategory(@RequestBody Category category){
    Category created = categoryService.createCategory(category);
    return ResponseEntity.status(201).body(created);
  }

  //Only admin will have access for this
  @PutMapping("updateCategory/{categoryId}")
  public ResponseEntity<Category> updateCategory(@PathVariable int categoryId, @RequestBody Category
category) {
    Category updated = categoryService.updateCategory(categoryId,category);

    return ResponseEntity.ok(updated);
  }

  //Only admin will have access for this
  @DeleteMapping("deleteCategory/{categoryId}")
  public ResponseEntity<Category> deleteCategory(@PathVariable int categoryId) {
    categoryService.deleteCategory(categoryId);
```

```
    return ResponseEntity.noContent().build();
  }

  //Accessible
  @GetMapping("getCategoryByID/{categoryId}")
  public ResponseEntity<Category> getCategoryById(@PathVariable int categoryId) {
    Category category = categoryService.getCategoryById(categoryId);

    return ResponseEntity.ok(category);
  }

  //This should be secured. Login Authenetication is required.
  @GetMapping("names")
  public ResponseEntity<List<String>> getAllCategoryNames() {
    List<String> categoryNames = categoryService.getAllCategoryNames();
    return ResponseEntity.ok(categoryNames);
  }

}


package com.bartr.controller;

import com.bartr.model.Course;
import com.bartr.service.CourseService;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/courses")
@RequiredArgsConstructor
public class CourseController {

  private final CourseService courseService;

  //  Create a new course
  @PostMapping("/insertCourse")
  public ResponseEntity<Course> createCourse(@RequestBody Course course) {
    Course created = courseService.createCourse(course);
    return ResponseEntity.ok(created);
  }

  // Update existing course. and only the person having the token can only change the same person who is
logged in.
  //No use write now no body would change course for now change via backend
```

```
    @PutMapping("updateCourse/{id}")
    public ResponseEntity<Course> updateCourse(@PathVariable int id, @RequestBody Course course) {
        Course updated = courseService.updateCourse(id, course);
        return ResponseEntity.ok(updated);
    }

    // Delete a course only the creator can delete it
    //No option to delete a course for now. You can do iot only via backend or sql
    @DeleteMapping("deleteCourse/{id}")
    public ResponseEntity<String> deleteCourse(@PathVariable int id) {
        courseService.deleteCourse(id);
        return ResponseEntity.ok("Course deleted successfully.");
    }

    // Get course by ID
    @GetMapping("/{id}")
    public ResponseEntity<Course> getCourseById(@PathVariable int id) {
        Course course = courseService.getCourseById(id);
        return ResponseEntity.ok(course);
    }

    // Get all courses
    @GetMapping("")
    public ResponseEntity<List<Course>> getAllCourses() {
        return ResponseEntity.ok(courseService.getAllCourses());
    }

    // Get courses by creator ID
    @GetMapping("/creator/{creatorId}")
    public ResponseEntity<List<Course>> getCoursesByCreator(@PathVariable int creatorId) {
        return ResponseEntity.ok(courseService.getCoursesByCreatorId(creatorId));
    }

    // Get courses by category ID
    @GetMapping("/category/{categoryId}")
    public ResponseEntity<List<Course>> getCoursesByCategory(@PathVariable int categoryId) {
        return ResponseEntity.ok(courseService.getCoursesByCategoryId(categoryId));
    }
}



package com.bartr.controller;

import com.bartr.model.Course;
import com.bartr.model.Enrollment;
import com.bartr.service.EnrollmentService;
import lombok.RequiredArgsConstructor;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping("/api/enrollments")
@RequiredArgsConstructor
public class EnrollmentController {

  private final EnrollmentService enrollmentService;

  @PostMapping("/insert")
  public ResponseEntity<Enrollment> createEnrollment(@RequestBody Enrollment enrollment) {
    return ResponseEntity.ok(enrollmentService.saveEnrollment(enrollment));
  }

  @PostMapping("/insert/{userId}/{courseId}")
  public ResponseEntity<Enrollment> enrollUser(@PathVariable int userId, @PathVariable int courseId) {
    return ResponseEntity.ok(enrollmentService.enroll(userId,courseId));
  }
  @GetMapping("")
  public ResponseEntity<List<Enrollment>> getAllEnrollments() {
    return ResponseEntity.ok(enrollmentService.getAllEnrollments());
  }

  @GetMapping("/{id}")
  public ResponseEntity<Enrollment> getEnrollmentById(@PathVariable int id) {
    Enrollment enrollment = enrollmentService.getEnrollmentById(id);
    return (enrollment != null) ? ResponseEntity.ok(enrollment) : ResponseEntity.notFound().build();
  }

  @GetMapping("/learner/{learnerId}")
  public ResponseEntity<List<Enrollment>> getEnrollmentsByLearner(@PathVariable int learnerId) {
    return ResponseEntity.ok(enrollmentService.getEnrollmentsByLearnerId(learnerId));
  }

  @GetMapping("/course/{courseId}")
  public ResponseEntity<List<Enrollment>> getEnrollmentsByCourse(@PathVariable int courseId) {
    return ResponseEntity.ok(enrollmentService.getEnrollmentsByCourseId(courseId));
  }



  @DeleteMapping("/{id}")
  public ResponseEntity<Void> deleteEnrollment(@PathVariable int id) {
    enrollmentService.deleteEnrollment(id);
```

```
    return ResponseEntity.noContent().build();
  }

  @GetMapping("/{learnerId}/courses")
  public ResponseEntity<?> getEnrolledCoursesForLearner(@PathVariable int learnerId) {

      List<Course> courses = enrollmentService.getCoursesEnrolledByLearnerId(learnerId);
      if (courses.isEmpty()) {
        // If the learner exists but has no enrollments
        return ResponseEntity.noContent().build();
      }
      return ResponseEntity.ok(courses);

  }

  @GetMapping("/isEnrolled")
  public ResponseEntity<?> isEnrolled(@RequestParam int learnerId, @RequestParam int courseId){
    System.out.println("sfsjdcbgydVCJFWgevdcgdfvxgfwVcxdfQVCWTDXCQWtjgjvnaj");
    boolean isEnrolled = enrollmentService.isUserEnrolled(learnerId,courseId);
    return ResponseEntity.ok(isEnrolled);
  }


}


package com.bartr.controller;

import com.bartr.model.Payment;
import com.bartr.service.PaymentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/payments")
public class PaymentController {

  @Autowired
  private PaymentService paymentService;

  //  🔹 Calculate price for XP purchase
  @GetMapping("/price")
  public ResponseEntity<Integer> calculatePrice(@RequestParam int xp) {
    int price = paymentService.calculatePrice(xp);
    return ResponseEntity.ok(price);
```

```
    }

    // 🔹 Buy XP (fake payment)
    @PostMapping("/buy-xp")
    public ResponseEntity<Payment> buyXp(
            @RequestParam int userId,
            @RequestParam int xpToBuy
    ) {
        Payment payment = paymentService.createPayment(userId, xpToBuy);
        return ResponseEntity.ok(payment);
    }

    // 🔹 User's XP purchase history
    @GetMapping("/user/{userId}")
    public ResponseEntity<List<Payment>> getUserPayments(@PathVariable int userId) {
        return ResponseEntity.ok(paymentService.getPaymentsByUserId(userId));
    }
}
```

```
package com.bartr.controller;

import com.bartr.model.Course;
import com.bartr.service.SearchService;
import lombok.AllArgsConstructor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
    @RequestMapping("/api/search")
@AllArgsConstructor
public class SearchController {

    private final SearchService searchService;

    @GetMapping
    public ResponseEntity<List<Course>> search(@RequestParam String keyword){
        List<Course> courses = searchService.search(keyword);
        return ResponseEntity.ok(courses);
    }
}
```

```
package com.bartr.controller;

import com.bartr.model.Transaction;
import com.bartr.service.TransactionService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/transactions")
public class TransactionController {

    @Autowired
    private TransactionService transactionService;

    // 🔹 Create a transaction
    @PostMapping("/insert")
    public ResponseEntity<Transaction> createTransaction(@RequestBody Transaction transaction) {
        Transaction created = transactionService.createTransaction(transaction);
        return ResponseEntity.ok(created);
    }

    // 🔹 Get all transactions
    @GetMapping("")
    public ResponseEntity<List<Transaction>> getAllTransactions() {
        return ResponseEntity.ok(transactionService.getAllTransactions());
    }

    // 🔹 Get transaction by ID
    @GetMapping("/{id}")
    public ResponseEntity<Transaction> getTransactionById(@PathVariable int id) {
        return ResponseEntity.ok(transactionService.getTransactionById(id));
    }

    // 🔹 Get transactions by user ID
    @GetMapping("/user/{userId}")
    public ResponseEntity<List<Transaction>> getTransactionsByUser(@PathVariable int userId) {
        return ResponseEntity.ok(transactionService.getTransactionsByUser(userId));
    }

    // 🔹 Get transactions by course ID
    @GetMapping("/course/{courseId}")
    public ResponseEntity<List<Transaction>> getTransactionsByCourse(@PathVariable int courseId) {
        return ResponseEntity.ok(transactionService.getTransactionsByCourse(courseId));
```

```
    }

    // 🔹 Delete a transaction
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteTransaction(@PathVariable int id) {
        transactionService.deleteTransaction(id);
        return ResponseEntity.noContent().build();
    }
}


package com.bartr.controller;

import com.bartr.model.User;
import com.bartr.service.UserService;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/users")

public class UserController {

    private final UserService userService;

    public UserController(UserService userService){
        this.userService = userService;
    }
    // 🔹 Register a new user
    @PostMapping("/register")
    public ResponseEntity<User> registerUser(@RequestBody User user) {
        System.out.println("controller reached");
        User createdUser = userService.registerUser(user);
        return ResponseEntity.status(201).body(createdUser);
    }

    // 🔹 Get user by email
    @GetMapping("/byEmail")
    public ResponseEntity<User> getUserByEmail(@RequestParam String email) {
        Optional<User> user = userService.getUserByEmail(email);
        return user.map(ResponseEntity::ok)
                .orElse(ResponseEntity.notFound().build());
    }
```

```
  @GetMapping("/byUsername")
  public ResponseEntity<User> getUserByUsername(@RequestParam String username) {
    Optional<User> user = userService.getUserByUsername(username);
    return user.map(ResponseEntity::ok)
        .orElse(ResponseEntity.notFound().build());
  }

  // 🔹 Update XP for user
  @PutMapping("/updateXP")
  public ResponseEntity<String> updateUserXp(@RequestParam int userId, @RequestParam int xpChange) {
    userService.updateXP(userId, xpChange);
    return ResponseEntity.ok("XP updated successfully");
  }

  // 🔹 Get current XP for user
  @GetMapping("/{id}/xp")
  public ResponseEntity<Integer> getUserXp(@PathVariable("id") int userId) {
    int xp = userService.getUserXp(userId);
    return ResponseEntity.ok(xp);
  }

  @GetMapping("")
  public ResponseEntity<List<User>> getAllUsers() {
    List<User> users = userService.getAllUser();
    return ResponseEntity.ok(users);
  }


  @PatchMapping("/update/{id}") // Maps to /api/users/{id}
  public ResponseEntity<?> updateUser(@PathVariable("id") int id, @RequestBody User userDetails) {

      // The service method is designed to handle partial updates by checking for non-null/non-empty fields
      User updatedUser = userService.updateUser(id, userDetails);
      return ResponseEntity.ok(updatedUser); // Return 200 OK with the updated user

  }

  @PatchMapping("/changePassword/{userId}")
  public ResponseEntity<?> changePassword(@PathVariable("userId") int userId, @RequestParam String
currentPassword, @RequestParam String newPassword) {
    boolean updated= userService.changePassword(userId,currentPassword,newPassword);
    return ResponseEntity.ok("Password updated successfully");

  }

  @DeleteMapping("/{userId}")
  public ResponseEntity<String> deleteUser(@PathVariable int userId){
```

```
        userService.deleteUser(userId);
        return ResponseEntity.ok("User deleted successfully");
    }
}
```

```
package com.bartr.exception;

public class DatabaseAccessException extends RuntimeException {

    public DatabaseAccessException(String message) {
        super(message);
    }

    public DatabaseAccessException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
package com.bartr.exception;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.HashMap;
import java.util.Map;

@RestControllerAdvice
public class Handler {

    // Utility method to create a structured JSON-like response
    private Map<String, Object> createResponse(String message, HttpStatus status) {
        Map<String, Object> errorResponse = new HashMap<>();
        errorResponse.put("error", true);
        errorResponse.put("message", message);
        errorResponse.put("status", status.value());
        errorResponse.put("statusText", status.getReasonPhrase());
        return errorResponse;
    }

    @ExceptionHandler(DatabaseAccessException.class)
    public ResponseEntity<Map<String, Object>> handleDatabaseAccessException(DatabaseAccessException
ex) {
```

```
    Map<String, Object> response = createResponse("Database Error: " + ex.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);
    return new ResponseEntity<>(response, HttpStatus.INTERNAL_SERVER_ERROR);
  }

  @ExceptionHandler(UserNameNotFoundException.class)
  public ResponseEntity<Map<String, Object>>
handleUserNameNotFoundException(UserNameNotFoundException ex) {
    Map<String, Object> response = createResponse("Bad Credentials: " + ex.getMessage(),
HttpStatus.BAD_REQUEST);
    return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
  }

  @ExceptionHandler(Exception.class)
  public ResponseEntity<Map<String, Object>> handleGenericException(Exception ex) {
    Map<String, Object> response = createResponse("Error Occurred: " + ex.getMessage(),
HttpStatus.BAD_REQUEST);
    return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
  }

  @ExceptionHandler(UserIsNotAnAdmin.class)
  public ResponseEntity<Map<String, Object>> handleUserIsNotAnAdmin(UserIsNotAnAdmin ex) {
    Map<String, Object> response = createResponse("Forbidden: " + ex.getMessage(),
HttpStatus.FORBIDDEN);
    return new ResponseEntity<>(response, HttpStatus.FORBIDDEN);
  }

  @ExceptionHandler(InvalidPasswordException.class)
  public ResponseEntity<Map<String, Object>> hadleInvalidPassword(Exception ex) {
    Map<String, Object> response = createResponse("Error Occurred: " + ex.getMessage(),
HttpStatus.BAD_REQUEST);
    return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
  }

  @ExceptionHandler(UserAlreadyEnrolledException.class)
  public ResponseEntity<Map<String, Object>> hadleAlreadyEnrolled(Exception ex) {
    Map<String, Object> response = createResponse("Error Occurred: " + ex.getMessage(),
HttpStatus.BAD_REQUEST);
    return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
  }


}



package com.bartr.exception;
```

```
public class InvalidPasswordException extends RuntimeException{
  public InvalidPasswordException(String message) {
    super(message);
  }

  public InvalidPasswordException(String message, Throwable cause) {
    super(message, cause);
  }
}
```

```
package com.bartr.exception;

public class UserAlreadyEnrolledException extends RuntimeException {
  public UserAlreadyEnrolledException(String message) {
    super(message);
  }

  public UserAlreadyEnrolledException(String message, Throwable cause) {
    super(message, cause);
  }

}
```

```
package com.bartr.exception;

public class UserIsNotAnAdmin extends RuntimeException{
  public UserIsNotAnAdmin(String msg){
    super(msg);
  }
  public UserIsNotAnAdmin(String message, Throwable cause) {
    super(message, cause);
  }
}
```

```
package com.bartr.exception;

public class UsernameAlreadyExistsException extends RuntimeException{
  public UsernameAlreadyExistsException(String message) {
    super(message);
  }

  public UsernameAlreadyExistsException(String message, Throwable cause) {
    super(message, cause);
```

```
  }
}
```

```
package com.bartr.exception;

public class UserNameNotFoundException extends RuntimeException {

  public UserNameNotFoundException(String message) {
    super(message);
  }

  public UserNameNotFoundException(String message, Throwable cause) {
    super(message, cause);
  }
}
```

```
package com.bartr.filter;


import com.bartr.security.JwtUtil;
import com.bartr.service.impl.UserAuthServiceImpl;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component
@Slf4j
@AllArgsConstructor
public class JwtFilter extends OncePerRequestFilter {

  private final JwtUtil jwtUtil;
  private final UserAuthServiceImpl userAuthService;
```

```
    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain
filterChain)
            throws ServletException, IOException {

        final String authorizationHeader = request.getHeader("Authorization");

        try {

            // Validate presence of Authorization header
            if (authorizationHeader == null || !authorizationHeader.startsWith("Bearer ")) {
                System.out.println(authorizationHeader);
                log.info("Missing or malformed Authorization header.");
                filterChain.doFilter(request, response);
                return;
            }



            // Extract JWT token
            String jwt = authorizationHeader.substring(7);
            String username = jwtUtil.extractUsername(jwt);

            if (username == null || SecurityContextHolder.getContext().getAuthentication() != null) {
                log.warn("Invalid JWT token or user already authenticated.");
                filterChain.doFilter(request, response);
                return;
            }

            log.info("Validating JWT token for user: {}", username);
            UserDetails userDetails = userAuthService.loadUserByUsername(username);

            // Validate JWT token
            if (!jwtUtil.validateToken(jwt)) {
                log.error("JWT validation failed for user: {}", username);
                response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Invalid or expired JWT token.");
                return;
            }

            // Set authentication context
            UsernamePasswordAuthenticationToken authenticationToken =
                    new UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
            authenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
            SecurityContextHolder.getContext().setAuthentication(authenticationToken);

            log.info("JWT authentication successful for user: {}", username);
```

```
    } catch (ServletException e) {
        log.error("Exception occurred during JWT authentication: {}", e.getMessage(), e);
        response.sendError(HttpServletResponse.SC_INTERNAL_SERVER_ERROR, "Internal authentication
error.");
        return;
    }

    filterChain.doFilter(request, response);
  }
}



package com.bartr.model;

import jakarta.persistence.*;
// import jakarta.persistence.GeneratedValue;
// import jakarta.persistence.GenerationType;

import lombok.*;

@Entity
@Table(name = "category")
@Getter
@Setter
@Data // Generates getters, setters, toString, equals, and hashCode
@NoArgsConstructor // No-arg constructor
@AllArgsConstructor // All-arg constructor
public class Category {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;

  @Column(name = "name", nullable = false)
  private String name;

  @Column(name = "description")
  private String description;

  @Column(name = "imageUrl")
  private String imageUrl="https://images.unsplash.com/photo-1746105839114-
fbc9c81fcb17?q=80&w=1197&auto=format&fit=crop&ixlib=rb-
4.1.0&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D";

  @Column(name = "xpCost", nullable = false)
  private int xpCost;
```

```
    public String getImageUrl() {
        return imageUrl;
    }

    public void setImageUrl(String imageUrl) {
        this.imageUrl = imageUrl;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public int getXpCost() {
        return xpCost;
    }

    public void setXpCost(int xpCost) {
        this.xpCost = xpCost;
    }
}



package com.bartr.model;

import com.fasterxml.jackson.annotation.JsonManagedReference;
import jakarta.persistence.*;
```

```
import lombok.*;
import java.time.LocalDateTime;
import java.util.Date;
import java.util.List;

@Entity
@Table(name = "courses")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Course {

    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
    @Column(name = "courseId")
    private int id;

    @Column(nullable = false)
    private String title;

    @Column(nullable = false)
    private String description;

    @Column(nullable = false)
    private String level;

    @Column
    private String features;

    @Lob
    @Column(nullable = false)
    private String courseOutLine;

    @Column
    private double price;

    @Column
    private String imageUrl = "https://plus.unsplash.com/premium_photo-1680553489384-
8e3230dd1073?q=80&w=755&auto=format&fit=crop&ixlib=rb-
4.1.0&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D";

    @Column
    private String videoUrl = "https://www.learningcontainer.com/wp-content/uploads/2020/05/sample-mp4-
file.mp4";
```

```
    @Column
    private int enrolledUser=0;

    @ManyToOne
    @JoinColumn(name = "categoryId", nullable = false)
    private Category category;

    @ManyToOne
    @JoinColumn(name = "creatorId", nullable = false)
    private User creator;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "enrollmentDate", nullable = false, updatable = false)
    private Date createdAt;

    @PrePersist
    protected void onCreate() {
      this.createdAt = new Date();
    }

    @OneToMany(mappedBy = "course", cascade = CascadeType.ALL)
    @JsonManagedReference(value = "course-enrollments")
    private List<Enrollment> enrollments;



}


package com.bartr.model;

import com.fasterxml.jackson.annotation.JsonBackReference;
import jakarta.persistence.*;
import lombok.*;

import java.util.Date;

@Entity
@Table(name = "enrollments")
public class Enrollment {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    // 🔹 Many enrollments can refer to one course
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "courseId", nullable = false)
    @JsonBackReference(value = "course-enrollments")
```

```
private Course course;

// 🔹 Many enrollments can refer to one user
@ManyToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "learnerId")
@JsonBackReference(value = "user-enrollments")
private User learner;

@Temporal(TemporalType.TIMESTAMP)
@Column(name = "enrollmentDate", nullable = false, updatable = false)
private Date enrollmentDate;

// Automatically sets the date before persisting
@PrePersist
protected void onCreate() {
  this.enrollmentDate = new Date();
}

// Getters and setters
public int getId() {
  return id;
}

public void setId(int id) {
  this.id = id;
}

public Course getCourse() {
  return course;
}

public void setCourse(Course course) {
  this.course = course;
}

public User getLearner() {
  return learner;
}

public void setLearner(User learner) {
  this.learner = learner;
}

public Date getEnrollmentDate() {
  return enrollmentDate;
}
```

```
  public void setEnrollmentDate(Date enrollmentDate) {
    this.enrollmentDate = enrollmentDate;
  }
}
```

```
package com.bartr.model;

import jakarta.persistence.*;
import lombok.*;

import java.util.Date;

@Entity
@Table(name = "payment")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Payment {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;

  @Column(name = "userId", nullable = false)
  private int userId;

  @Column(name = "amount", nullable = false)
  private int amount;

  @Column(name = "mode", nullable = false)
  private String mode;

  @Column(name = "xpPurchased")
  private int xpPurchased;

  @Temporal(TemporalType.TIMESTAMP)
  @Column(name = "purchasedAt")
  private Date purchasedAt;

  @PrePersist
  protected void onCreate() {
    this.purchasedAt = new Date();
  }

  public int getId() {
```

```
    return id;
  }

  public void setId(int id) {
    this.id = id;
  }

  public int getUserId() {
    return userId;
  }

  public void setUserId(int userId) {
    this.userId = userId;
  }

  public int getAmount() {
    return amount;
  }

  public void setAmount(int amount) {
    this.amount = amount;
  }

  public String getMode() {
    return mode;
  }

  public void setMode(String mode) {
    this.mode = mode;
  }

  public int getXpPurchased() {
    return xpPurchased;
  }

  public void setXpPurchased(int xpPurchased) {
    this.xpPurchased = xpPurchased;
  }

  public Date getPurchasedAt() {
    return purchasedAt;
  }

  public void setPurchasedAt(Date purchasedAt) {
    this.purchasedAt = purchasedAt;
  }
}
```

```
package com.bartr.model;

public enum Role {
   ROLE_USER,
   ROLE_ADMIN
}


package com.bartr.model;

import jakarta.persistence.*;
import lombok.*;

import java.util.Date;

@Entity
@Table(name = "transaction")
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Transaction {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private int id;

   @ManyToOne
   @JoinColumn(name = "userId", nullable = false)
   private User user;

   @ManyToOne
   @JoinColumn(name = "courseId", nullable = false)
   private Course course;

   @Column(name = "type", nullable = false)
   private String type;

   @Column(name = "amount", nullable = false)
   private int amount;

   @Temporal(TemporalType.TIMESTAMP)
   @Column(name = "transactedAt")
   private Date transactedAt;

   @PrePersist
```

```
protected void onCreate() {
   this.transactedAt = new Date();
}

public int getId() {
  return id;
}

public void setId(int id) {
  this.id = id;
}

public User getUser() {
  return user;
}

public void setUserId(User user) {
  this.user = user;
}

public Course getCourse() {
  return course;
}

public void setCourseId(Course course) {
  this.course = course;
}

public String getType() {
  return type;
}

public void setType(String type) {
  this.type = type;
}

public int getAmount() {
  return amount;
}

public void setAmount(int amount) {
  this.amount = amount;
}

public Date getTransactedAt() {
  return transactedAt;
}
```

```
  public void setTransactedAt(Date transactedAt) {
    this.transactedAt = transactedAt;
  }
}
```

```
package com.bartr.model;

import com.fasterxml.jackson.annotation.JsonManagedReference;
import jakarta.persistence.*;

import lombok.*;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.util.Date;
import java.util.List;

@Entity
@Table(name = "user_table") // Rename if "user" is a reserved word in your DB
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Component
public class User {

  @Id
  //@Column(name = "id")
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private int id;

  @Column(name = "username", nullable = false, unique = true)
  private String username;

  @Column(name = "email", nullable = false, unique = true)
  private String email;

  @Column(name = "password", nullable = false)
  private String password;

  @Column(name = "phone")
  private String phone;

  @Column(name = "fullname")
```

```
    private String fullname;

    @Column(name = "region")
    private String region = "Chennai, India";

    @Column(name = "skills")
    private String skills ;

    @Column(name = "bio")
    private String bio = "This is a sample bio" ;

    @Column(name = "xp")
    private int xp=100;

    @Column(nullable = false)
    private Role role = Role.ROLE_USER;

    @Column(name = "avatarUrl")
    private String avatarUrl;

    @Column(name = "responseTime")
    private int responseTime=24;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(name = "createdAt")
    private Date createdAt;

    @PrePersist
    protected void onCreate() {
      this.createdAt = new Date();
    }

    @OneToMany
    @JoinColumn(name = "learnerId", referencedColumnName = "id") // maps to Enrollment.learner_id
    @JsonManagedReference(value = "user-enrollments")
    private List<Enrollment> enrollmentList;

    public int getXp() {
       return xp;
    }

    public void setXp(int xp){
      this.xp = xp;
    }
}
```

```
package com.bartr.repository;

import com.bartr.model.Category;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface CategoryRepository extends JpaRepository<Category, Integer> {
  // Add custom query methods if needed
  @Query("Select name from Category")
  List<String> findAllCategoryNames();
}



package com.bartr.repository;

import com.bartr.model.Course;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public interface CourseRepository extends JpaRepository<Course, Integer> {
  // Add custom query methods if needed
  List<Course> findByCreatorId(int creatorId);

  List<Course> findByCategoryId(int categoryId);

  @Query("""
      SELECT c FROM Course c
      JOIN c.category cat
      JOIN c.creator creator
      WHERE (
        LOWER(c.title) LIKE LOWER(CONCAT('%', :keyword, '%')) OR
        LOWER(c.description) LIKE LOWER(CONCAT('%', :keyword, '%')) OR
        LOWER(cat.name) LIKE LOWER(CONCAT('%', :keyword, '%')) OR
        LOWER(creator.username) LIKE LOWER(CONCAT('%', :keyword, '%')) OR
        LOWER(creator.email) LIKE LOWER(CONCAT('%', :keyword, '%'))
      )
      """)
  List<Course> searchRelevantCourses(@Param("keyword") String keyword);
```

```
}
```

```
package com.bartr.repository;

import com.bartr.model.Enrollment;
import com.bartr.model.User;
import com.bartr.model.Course;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface EnrollmentRepository extends JpaRepository<Enrollment, Integer> {
  List<Enrollment> findByLearner(User learner);
  List<Enrollment> findByCourse(Course course);

  @Query("SELECT e.course FROM Enrollment e WHERE e.learner.id = :learnerId")
  List<Course> findCoursesByLearnerId(@Param("learnerId") int learnerId);


  boolean existsByLearnerIdAndCourseId(int learnerId, int courseId);
  void deleteByCourse(Course course);
}
```

```
package com.bartr.repository;

import com.bartr.model.Payment;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface PaymentRepository extends JpaRepository<Payment, Integer> {
  // Add custom query methods if needed
  List<Payment> findByUserId(int userId);

  void deleteByUserId(int userId);
}
```

```
package com.bartr.repository;

import com.bartr.model.Course;
import com.bartr.model.Transaction;
import com.bartr.model.User;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface TransactionRepository extends JpaRepository<Transaction, Integer> {

   List<Transaction> findByUser(User user);

   List<Transaction> findByCourse(Course course);

   void deleteByCourse(Course courseId);
   void deleteByUser(User user);
}



package com.bartr.repository;

import com.bartr.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Integer> {
   Optional<User> findByUsername(String username);
   Optional<User> findByEmail(String email);

}


package com.bartr.security;

import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.JwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.security.Keys;
```

```
import org.springframework.stereotype.Component;

import java.security.Key;
import java.util.Date;

@Component
public class JwtUtil {

    private Key getSigningKey() {
        String SECRET_KEY = "helloKeyBoomBamWith32bitAllowedInTheSecretKeyOK";
        return Keys.hmacShaKeyFor(SECRET_KEY.getBytes());
    }

    public String generateToken(String username, String role) {
        long EXPIRATION_TIME = 1000 * 60 * 50;
        return Jwts.builder()
                .setSubject(username)
                .claim("role", role)
                .setIssuedAt(new Date())
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .signWith(getSigningKey(), SignatureAlgorithm.HS256)
                .compact();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parserBuilder().setSigningKey(getSigningKey()).build().parseClaimsJws(token);
            return true; // Token is valid
        } catch (ExpiredJwtException e) {
            System.out.println("JWT Token Expired");
        } catch (JwtException e) {
            System.out.println("Invalid JWT Token");
        }
        return false; // Token is invalid
    }

    public String extractUsername(String token) {
        return Jwts.parserBuilder().setSigningKey(getSigningKey()).build()
                .parseClaimsJws(token).getBody().getSubject();
    }
}


package com.bartr.security;

import com.bartr.filter.JwtFilter;
import lombok.AllArgsConstructor;
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.Customizer;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configurers.AbstractHttpConfigurer;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import org.springframework.web.filter.CorsFilter;

import java.util.Arrays;
import java.util.List;

@Configuration
@EnableWebSecurity
@AllArgsConstructor
public class SecurityConfig {

  private final JwtFilter jwtFilter;

  // main filter setting the domain to authorize and urls to check for authentication/jwt token
  @Bean
  public SecurityFilterChain customSecurityFilterChain(HttpSecurity httpSec) throws Exception {
    httpSec.csrf(AbstractHttpConfigurer::disable);
    httpSec.authorizeHttpRequests(req -> req
            //Category
            .requestMatchers(HttpMethod.GET,"/api/categories").permitAll()
            .requestMatchers(HttpMethod.POST, "/api/categories/insertCategory").hasRole("ADMIN")
            .requestMatchers(HttpMethod.PUT,
"/api/categories/updateCategory/{categoryId}").hasRole("ADMIN")
            .requestMatchers(HttpMethod.DELETE,
"/api/categories/deleteCategory/{categoryId}").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/api/categories/getCategoryByID/{categoryId}").permitAll()
            .requestMatchers(HttpMethod.GET, "/api/categories/names").permitAll()

            //Course
            .requestMatchers(HttpMethod.GET,"/api/courses").permitAll()
            .requestMatchers(HttpMethod.POST,"/api/courses/insertCourse").authenticated()
            .requestMatchers(HttpMethod.PUT,"/api/courses/updateCourse/{id}").hasRole("ADMIN")
```

```
            .requestMatchers(HttpMethod.DELETE, "/api/courses/deleteCourse/{id}").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/api/courses/{id}").permitAll()
            .requestMatchers(HttpMethod.GET, "/api/courses/creator/{creatorId}").permitAll()
            .requestMatchers(HttpMethod.GET,"/api/courses/category/{categoryId}").permitAll()
            .requestMatchers(HttpMethod.GET, "/api/categories/getCategoryById/**").permitAll()


//          //Enrollments
            .requestMatchers(HttpMethod.GET, "/api/enrollments/isEnrolled").authenticated()
            .requestMatchers(HttpMethod.POST, "/api/enrollments/insert").hasRole("ADMIN")
            .requestMatchers(HttpMethod.POST,
"/api/enrollments/insert/{userId}/{courseId}").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/enrollments").permitAll()
            .requestMatchers(HttpMethod.GET, "/api/enrollments/{learnerId}/courses").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/enrollments/{id}").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/api/enrollments/learner/{learnerId}").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/enrollments/course/{courseId}").hasRole("ADMIN")
            .requestMatchers(HttpMethod.DELETE, "/api/enrollments/{id}").hasRole("ADMIN")



            //Transactions
            .requestMatchers(HttpMethod.GET, "/api/transactions").hasRole("ADMIN")
            .requestMatchers(HttpMethod.POST, "/api/transactions/insert").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/transactions/{id}").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/api/transactions/user/{userId}").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/transactions/course/{courseId}").hasRole("ADMIN")
            .requestMatchers(HttpMethod.GET, "/api/transactions").hasRole("ADMIN")
            .requestMatchers(HttpMethod.DELETE, "/api/transactions/{id}").hasRole("ADMIN")

            //Payment
            .requestMatchers(HttpMethod.GET,"/api/payments/price").authenticated()
            .requestMatchers(HttpMethod.POST,"/api/payments/buy-xp").authenticated()
            .requestMatchers(HttpMethod.GET,"/api/payments/user/{userId}").authenticated()

            //User
            .requestMatchers(HttpMethod.GET, "/api/users").hasRole("ADMIN")
            .requestMatchers(HttpMethod.POST, "/api/users/register").permitAll()
            .requestMatchers(HttpMethod.PATCH, "/api/users/update/{id}").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/users/byEmail").permitAll()
            .requestMatchers(HttpMethod.PUT, "/api/users/updateXP").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/users/{id}/xp").authenticated()
            .requestMatchers(HttpMethod.GET, "/api/users/byUsername").permitAll()
            .requestMatchers(HttpMethod.PATCH, "/api/users/changePassword/{userId}").authenticated()
            .requestMatchers(HttpMethod.DELETE, "/api/users/{userId}").authenticated()
```

```
                //Search
                .requestMatchers(HttpMethod.GET, "/api/search").permitAll()


                .requestMatchers(HttpMethod.GET,"/me").permitAll()


                // Login
                .requestMatchers(HttpMethod.POST, "/login").permitAll()

//                  .anyRequest().permitAll()
        );


        httpSec.cors(cors -> cors.configurationSource(corsConfigurationSource()));



        //httpSec.cors(Customizer.withDefaults());
        httpSec.httpBasic(Customizer.withDefaults());
        httpSec.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

        return httpSec.build();
    }

    // for jwt internal use
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration
authenticationConfiguration)
            throws Exception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    // to encode and save password in db
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    // filter to allow your frontend to communicate with backend



    /**
     * Configures the CORS (Cross-Origin Resource Sharing) policy.
     * This allows web applications from other origins (e.g., a frontend running on localhost:4200)
     * to make requests to this API.
```

```
     *
     * @return A CorsConfigurationSource bean.
     */
    @Bean
    CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        // Allow requests from the Angular frontend development server.
        configuration.setAllowedOrigins(List.of("http://localhost:4200"));
        // Allow common HTTP methods.
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PUT", "DELETE", "PATCH", "OPTIONS"));
        // Allow all headers to be sent in requests.
        configuration.setAllowedHeaders(List.of("*"));
        // Allow sending credentials (like cookies or HTTP authentication headers, though JWT is typically in
Authorization header).
        configuration.setAllowCredentials(true);
        // Expose the "Authorization" header to the client, which is needed to read the JWT.
        configuration.setExposedHeaders(List.of("Authorization"));

        // Apply this CORS configuration to all incoming paths.
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }




//    @Bean
//    public CorsFilter corsFilter() {
//        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
//        CorsConfiguration config = new CorsConfiguration();
//
//        config.setAllowedOriginPatterns(List.of("*")); // Ensure correct frontend URL
//        config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS", "PATCH"));
//        config.setAllowedHeaders(List.of("*"));
//        config.setAllowCredentials(true); // Must match frontend requests
//
//        source.registerCorsConfiguration("/**", config);
//        return new CorsFilter(source);
//    }
}



package com.bartr.service.impl;

import java.util.List;
```

```
import com.bartr.model.Category;
import com.bartr.repository.CategoryRepository;
import com.bartr.service.CategoryService;

import lombok.AllArgsConstructor;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Service;

@Service
public class CategoryServiceImpl implements CategoryService {
  private final CategoryRepository categoryRepository;

  public CategoryServiceImpl(CategoryRepository categoryRepository){
    this.categoryRepository = categoryRepository;
  }

  @Override
  public Category createCategory(Category category){
    return categoryRepository.save(category);
  }

  @Override
  public Category updateCategory(int id, Category category){
    Category existing = categoryRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Category not found with ID: "+id));
     existing.setName(category.getName());
     existing.setDescription(category.getDescription());
     existing.setXpCost(category.getXpCost());
     return categoryRepository.save(existing);
  }

  @Override
  public void deleteCategory(int id){
    categoryRepository.deleteById(id);
  }

  @Override
  public Category getCategoryById(int id){
    return categoryRepository.findById(id)
        .orElseThrow(() -> new RuntimeException("Category not found with ID: "+id));
  }

  @Override
```

```
    public List<Category> getAllCategories(){
        return categoryRepository.findAll();
    }

    @Override
    public List<String> getAllCategoryNames(){
        return categoryRepository.findAllCategoryNames();
    }

}



package com.bartr.service.impl;

import com.bartr.model.Category;
import com.bartr.model.Course;
import com.bartr.model.User;
import com.bartr.repository.*;
import com.bartr.service.CourseService;
import jakarta.transaction.Transactional;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
@RequiredArgsConstructor
public class CourseServiceImpl implements CourseService {

    private final CourseRepository courseRepository;
    private final CategoryRepository categoryRepository;
    private final UserRepository userRepository;
    private final EnrollmentRepository enrollmentRepository;
    private final TransactionRepository transactionRepository;

    @Override
    public Course createCourse(Course course) {
        Category category = categoryRepository.findById(course.getCategory().getId())
                .orElseThrow(() -> new RuntimeException("Category not found"));

        // Fetch and validate creator
        User creator = userRepository.findById(course.getCreator().getId())
                .orElseThrow(() -> new RuntimeException("Creator not found"));

        // Set references properly
        course.setCategory(category);
```

```
    course.setCreator(creator);

    // Calculate XP-based price
    double multiplier = getLevelMultiplier(course.getLevel());
    double price = category.getXpCost() * multiplier;
    course.setPrice(price);

    return courseRepository.save(course);
}

@Override
public Course updateCourse(int id, Course updatedCourse) {


    Course existing = courseRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Course not found"));

    existing.setTitle(updatedCourse.getTitle());
    existing.setDescription(updatedCourse.getDescription());

    if (updatedCourse.getCategory() != null) {
        Category category = categoryRepository.findById(updatedCourse.getCategory().getId())
                .orElseThrow(() -> new RuntimeException("Category not found"));
        existing.setCategory(category);
    }

    if (updatedCourse.getCreator() != null) {
        User creator = userRepository.findById(updatedCourse.getCreator().getId())
                .orElseThrow(() -> new RuntimeException("Creator not found"));
        existing.setCreator(creator);
    }

    existing.setLevel(updatedCourse.getLevel());

    // Recalculate price
    double price = existing.getCategory().getXpCost() * getLevelMultiplier(existing.getLevel());
    existing.setPrice(price);


    return courseRepository.save(existing);
}

@Override
@Transactional
public void deleteCourse(int courseId) {
    Course course = courseRepository.findById(courseId)
            .orElseThrow(() -> new RuntimeException("Course not found"));
```

```
    enrollmentRepository.deleteByCourse(course);
    transactionRepository.deleteByCourse(course);
    courseRepository.deleteById(courseId);
  }

  @Override
  public Course getCourseById(int id) {
    return courseRepository.findById(id).orElseThrow(() -> new RuntimeException("Course not found"));
  }

  @Override
  public List<Course> getAllCourses() {

    List<Course> courses= courseRepository.findAll();
    for(Course course: courses){
      course.setEnrolledUser(course.getEnrollments().size());
    }
    return courses;
  }

  @Override
  public List<Course> getCoursesByCreatorId(int creatorId) {
    User creator = userRepository.findById(creatorId).orElseThrow(() -> new RuntimeException("Creator not
found"));;
    return courseRepository.findByCreatorId(creatorId);
  }

  @Override
  public List<Course> getCoursesByCategoryId(int categoryId) {
    Category category = categoryRepository.findById(categoryId).orElseThrow(() -> new
RuntimeException("Categoty not found"));;
    return courseRepository.findByCategoryId(categoryId);
  }

  private double getLevelMultiplier(String level) {
    switch (level.toLowerCase()) {
      case "beginner":
        return 1.0;
      case "intermediate":
        return 1.25;
      case "advanced":
        return 1.5;
      default:
        throw new IllegalArgumentException("Invalid level: " + level);
    }
  }
```

```
}
```

```
package com.bartr.service.impl;


import com.bartr.exception.UserAlreadyEnrolledException;
import com.bartr.model.Course;
import com.bartr.model.Enrollment;
import com.bartr.model.Transaction;
import com.bartr.model.User;
import com.bartr.repository.CourseRepository;
import com.bartr.repository.EnrollmentRepository;
import com.bartr.repository.TransactionRepository;
import com.bartr.repository.UserRepository;
import com.bartr.service.EnrollmentService;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
public class EnrollmentServiceImpl implements EnrollmentService {

  private final EnrollmentRepository enrollmentRepository;

  private final CourseRepository courseRepository;
  private final UserRepository userRepository;
  private final TransactionRepository transactionRepository ;

  @Override
  public Enrollment enroll(int userId, int courseId) {
    boolean isEnrolled = isUserEnrolled(userId,courseId);
    if(isEnrolled){
      throw new UserAlreadyEnrolledException("User is already enrolled into the course");
    }
    User learner = userRepository.findById(userId)
        .orElseThrow(() -> new RuntimeException("User not found"));

    Course course = courseRepository.findById(courseId)
        .orElseThrow(() -> new RuntimeException("Course not found"));
```

```
    if (learner.getXp() < course.getPrice()) {
        throw new RuntimeException("Insufficient XP to enroll");
    }

    // 🔻 Deduct XP from learner
    learner.setXp(learner.getXp() - (int) course.getPrice());
    userRepository.save(learner);

    // 🔻 Create learner's transaction ("sent")
    Transaction sentTxn = new Transaction();
    sentTxn.setUser(learner);
    sentTxn.setCourse(course);
    sentTxn.setType("sent");
    sentTxn.setAmount((int) course.getPrice());
    transactionRepository.save(sentTxn);

    // 🔻 Credit XP to course creator
    User creator = course.getCreator();
    creator.setXp(creator.getXp() + (int) course.getPrice());
    userRepository.save(creator);

    // 🔻 Create creator's transaction ("received")
    Transaction receivedTxn = new Transaction();
    receivedTxn.setUser(creator);
    receivedTxn.setCourse(course);
    receivedTxn.setType("received");
    receivedTxn.setAmount((int) course.getPrice());
    transactionRepository.save(receivedTxn);

    // 🔻 Save enrollment
    Enrollment enrollment = new Enrollment();
    enrollment.setCourse(course);
    enrollment.setLearner(learner);

    course.setEnrolledUser(course.getEnrolledUser()+1);
    courseRepository.save(course);

    return enrollmentRepository.save(enrollment);
}


@Override
public List<Enrollment> getAllEnrollments() {
    return enrollmentRepository.findAll();
}

@Override
```

```
    public Enrollment getEnrollmentById(int id) {
        return enrollmentRepository.findById(id).orElse(null);
    }

    @Override
    public List<Enrollment> getEnrollmentsByLearnerId(int learnerId) {
        return userRepository.findById(learnerId)
                .map(enrollmentRepository::findByLearner)
                .orElse(List.of());
    }

    @Override
    public List<Enrollment> getEnrollmentsByCourseId(int courseId) {
        return courseRepository.findById(courseId)
                .map(enrollmentRepository::findByCourse)
                .orElse(List.of());
    }

    @Override
    public Enrollment saveEnrollment(Enrollment enrollment) {
        return enrollmentRepository.save(enrollment);
    }

    @Override
    public void deleteEnrollment(int id) {
        enrollmentRepository.deleteById(id);
    }

    public List<Course> getCoursesEnrolledByLearnerId(int learnerId) {
        Optional<User> learner = userRepository.findById(learnerId);
        if (learner.isEmpty()) {
            throw new RuntimeException("Learner with ID " + learnerId + " not found.");
        }

        // Use the custom query from EnrollmentRepository to directly fetch courses
        return enrollmentRepository.findCoursesByLearnerId(learnerId);
    }

    public boolean isUserEnrolled(int learnerId, int courseId){
        System.out.println("sfsjvsjvnaj");
        return enrollmentRepository.existsByLearnerIdAndCourseId(learnerId,courseId);
    }



}
```

```
package com.bartr.service.impl;

import com.bartr.model.Payment;
import com.bartr.model.User;
import com.bartr.repository.PaymentRepository;
import com.bartr.repository.UserRepository;
import com.bartr.service.PaymentService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Date;
import java.util.List;

@Service
public class PaymentServiceImpl implements PaymentService {

  @Autowired
  private PaymentRepository paymentRepository;

  @Autowired
  private UserRepository userRepository;

  @Override
  public int calculatePrice(int xp) {
    return xp * 5; // ₹5 per XP
  }

  @Override
  public Payment createPayment(int userId, int xpToBuy) {
    User user = userRepository.findById(userId)
        .orElseThrow(() -> new RuntimeException("User not found"));

    int amount = calculatePrice(xpToBuy);

    Payment payment = new Payment();
    payment.setUserId(userId);
    payment.setXpPurchased(xpToBuy);
    payment.setAmount(amount);
    payment.setMode("upi");



    paymentRepository.save(payment);

    user.setXp(user.getXp() + xpToBuy);
    userRepository.save(user);
```

```
    return payment;
  }

  @Override
  public List<Payment> getPaymentsByUserId(int userId) {
    return paymentRepository.findByUserId(userId);
  }
}
```

```
package com.bartr.service.impl;

import com.bartr.exception.UserNameNotFoundException;
import com.bartr.model.Course;
import com.bartr.model.User;
import com.bartr.repository.CourseRepository;
import com.bartr.repository.UserRepository;
import com.bartr.service.SearchService;
import lombok.AllArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@AllArgsConstructor
public class SearchServiceImpl implements SearchService {

  private final CourseRepository courseRepository;
  private final UserRepository userRepository;

  @Override
  public List<Course> search(String keyword){
    return courseRepository.searchRelevantCourses(keyword);
  }
}
```

```
package com.bartr.service.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.bartr.model.Course;
```

```
import com.bartr.model.Transaction;
import com.bartr.model.User;
import com.bartr.repository.CourseRepository;
import com.bartr.repository.TransactionRepository;
import com.bartr.repository.UserRepository;
import com.bartr.service.TransactionService;

@Service
public class TransactionServiceImpl implements TransactionService {

  @Autowired
  private TransactionRepository transactionRepository;

  @Autowired
  private UserRepository userRepository;

  @Autowired
  private CourseRepository courseRepository;

  @Override
  public Transaction createTransaction(Transaction transaction) {

    return transactionRepository.save(transaction);
  }

  @Override
  public Transaction getTransactionById(int id) {
    return transactionRepository.findById(id)
        .orElseThrow(() -> new RuntimeException("Transaction not found"));
  }

  @Override
  public List<Transaction> getAllTransactions() {
    return transactionRepository.findAll();
  }

  @Override
  public List<Transaction> getTransactionsByUser(int userId) {
    User user = userRepository.findById(userId)
        .orElseThrow(() -> new RuntimeException("User not found"));
    return transactionRepository.findByUser(user);
  }

  @Override
  public List<Transaction> getTransactionsByCourse(int courseId) {
    Course course = courseRepository.findById(courseId)
        .orElseThrow(() -> new RuntimeException("Course not found"));
```

```
    return transactionRepository.findByCourse(course);
  }

  @Override
  public void deleteTransaction(int id) {
    transactionRepository.deleteById(id);
  }
}



package com.bartr.service.impl;

import com.bartr.model.User;
import com.bartr.repository.UserRepository;
import lombok.AllArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.Collection;
import java.util.List;

@Service
@Slf4j
@AllArgsConstructor
public class UserAuthServiceImpl implements UserDetailsService {

  private final UserRepository userRepository;



  @Override
  public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
    try {
      if(username == null){
        log.debug("loadUserByUsername : Username is null");
        throw new RuntimeException("Username is null in loadUserByUsername");
      }

      User user = userRepository.findByUsername(username).orElse(null);
      if (user == null) {
        log.debug("loadUserByUsername : user data is null in DB");
```

```
            throw new UsernameNotFoundException("User not found in DB");
        }
        System.out.println(username);
        UserDetails result = new org.springframework.security.core.userdetails.User(user.getUsername(),
            user.getPassword(), getAuthorities(user));
        System.out.println(result);
        log.info("loadUserByUsername : User data is fetched from database and submitted to auth provider for
authentication");
        return result;
      }catch (Exception e) {
        log.error("loadUserByUsername : {}",e.getMessage());
        throw new UsernameNotFoundException(e.getMessage());
      }
    }

    private Collection<? extends GrantedAuthority> getAuthorities(User user) {
      GrantedAuthority authority = new SimpleGrantedAuthority(user.getRole().name());
      return List.of(authority);
    }
}



package com.bartr.service.impl;

import com.bartr.exception.InvalidPasswordException;
import com.bartr.exception.UserNameNotFoundException;
import com.bartr.exception.UsernameAlreadyExistsException;
import com.bartr.model.Course;
import com.bartr.model.Role;
import com.bartr.model.User;
import com.bartr.repository.*;
import com.bartr.security.JwtUtil;
import com.bartr.service.CourseService;
import com.bartr.service.UserService;
import lombok.AllArgsConstructor;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import java.util.Optional;

@Service
@AllArgsConstructor // Lombok provides constructor for all final fields
```

```
public class UserServiceImpl implements UserService {

  // Now inject the concrete userRepository instead of UserRepository
  private final UserRepository userRepository; // THIS IS THE KEY CHANGE
  private final PasswordEncoder encoder;
  private final AuthenticationManager authManager;
  private final UserAuthServiceImpl userAuthService;
  private final JwtUtil jwt;

  private final PaymentRepository paymentRepository;
  private final CourseRepository courseRepository;
  private final TransactionRepository transactionRepository;
  private final CourseService courseService;



  @Override
  @Transactional
  public User registerUser(User user) {
    // Business logic remains in service, but persistence delegated to DAO
    Optional<User> existingUserByEmail = userRepository.findByEmail(user.getEmail());
    if (existingUserByEmail.isPresent()) {
      throw new UsernameAlreadyExistsException("User already registered with email: " + user.getEmail());
    }

    Optional<User> existingUserByUsername = userRepository.findByUsername(user.getUsername());
    if (existingUserByUsername.isPresent()) {
      throw new UsernameAlreadyExistsException("User already registered with username: " +
user.getUsername());
    }

    user.setPassword(encoder.encode(user.getPassword()));

    if (user.getXp() == 0) {
      user.setXp(0);
    }

//      user.setRole(Role.ROLE_USER);

    // Delegate save operation to userRepository
    return userRepository.save(user);
  }

  @Override
  public Optional<User> getUserByEmail(String email) {
    // Delegate find operation to userRepository
    return userRepository.findByEmail(email);
  }
```

```
  @Override
  public Optional<User> getUserByUsername(String username){
    return userRepository.findByUsername(username);
  }

  @Override
  @Transactional
  public void updateXP(int userId, int xpChange) {
    // Find user via DAO, then update and save via DAO
    User user = userRepository.findById(userId)
        .orElseThrow(() -> new RuntimeException("User not found with ID: " + userId));

    user.setXp(user.getXp() + xpChange);
    userRepository.save(user); // Delegate save for update
  }

  @Override
  public int getUserXp(int userId) {
    // Delegate find operation to userRepository
    return userRepository.findById(userId)
        .map(User::getXp)
        .orElseThrow(() -> new RuntimeException("User not found with ID: " + userId));
  }

  @Override
  public List<User> getAllUser() {
    // Delegate find all operation to userRepository
    return userRepository.findAll();
  }

  @Transactional
  public String jwtLogin(User user) {
    try {
      authManager.authenticate(new UsernamePasswordAuthenticationToken(user.getUsername(),
user.getPassword()));
      UserDetails userDetails = userAuthService.loadUserByUsername(user.getUsername());
      System.out.println(userDetails);
      String token = jwt.generateToken(userDetails.getUsername(), userDetails.getAuthorities().toString());
      return token;
    } catch (Exception e) {
      throw new RuntimeException("Authentication failed: " + e.getMessage(), e);
    }
  }

  @Transactional
  public void updatePassword(String username, String newPassword) {
```

```
    // Find user via DAO, then update and save via DAO
    User user = userRepository.findByUsername(username)
        .orElseThrow(() -> new RuntimeException("User not found with the given username: " + username));

    user.setPassword(encoder.encode(newPassword));
    userRepository.save(user); // Delegate save for update
  }

  public User updateUser(int id, User updatedUser) throws UserNameNotFoundException {
    // Retrieve the existing user from the database
    User existingUser = userRepository.findById(id)
        .orElseThrow(() -> new UserNameNotFoundException("User not found with ID: " + id));

    // Only update mutable fields (immutable fields like 'username', 'email', etc., are excluded from form and
ignored)
    if (updatedUser.getFullname() != null && !updatedUser.getFullname().isEmpty()) {
      existingUser.setFullname(updatedUser.getFullname());
    }
    if (updatedUser.getPhone() != null && !updatedUser.getPhone().isEmpty()) {
      existingUser.setPhone(updatedUser.getPhone());
    }
    if (updatedUser.getRegion() != null && !updatedUser.getRegion().isEmpty()) {
      existingUser.setRegion(updatedUser.getRegion());
    }
    if (updatedUser.getSkills() != null && !updatedUser.getSkills().isEmpty()) {
      existingUser.setSkills(updatedUser.getSkills());
    }
    if (updatedUser.getBio() != null && !updatedUser.getBio().isEmpty()) {
      existingUser.setBio(updatedUser.getBio());
    }
    if (updatedUser.getResponseTime() > 0) {
      existingUser.setResponseTime(updatedUser.getResponseTime());
    }



    // Save and return the updated user entity
    return userRepository.save(existingUser);
  }


  public boolean changePassword(int userId,String currentPassword,String newPassword){
    User user= userRepository.findById(userId)
        .orElseThrow(() -> new UserNameNotFoundException("User not found with ID: " + userId));
    if (!encoder.matches(currentPassword,user.getPassword())){
      throw new InvalidPasswordException("Current password is incorrect");
    }
```

```
    user.setPassword(encoder.encode(newPassword));
    userRepository.save(user);
    return true;
  }

  @Override
  @Transactional
  public void deleteUser(int userId){
    User user= userRepository.findById(userId)
        .orElseThrow(() -> new UserNameNotFoundException("User not found with ID: " + userId));

    paymentRepository.deleteByUserId(userId);

    List<Course> courses = courseRepository.findByCreatorId(userId);
    for(Course course:courses){
      courseService.deleteCourse(course.getId());
    }
    transactionRepository.deleteByUser(user);



    userRepository.deleteById(userId);

  }
}
```

```
package com.bartr;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class BartrApplication {

  public static void main(String[] args) {
    SpringApplication.run(BartrApplication.class, args);
  }

}
```

```
#spring.application.name=bartr
#
## Server port
#server.port=8085
#
```

```
## ===============================
## = Data Source Configuration   =
## ===============================
#spring.datasource.url=postgresql://neondb_owner:npg_6McmqxB4azXS@ep-summer-credit-afy8iyxd-
pooler.c-2.us-west-2.aws.neon.tech/neondb?sslmode=require&channel_binding=require
#spring.datasource.username=hydtufiyg.iy7t6r5dhyjtfukgiuogyftydkugio
#spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.datasource.hikari.connection-timeout=60000
#
##logging.level.org.springframework=DEBUG
#
#
## ===============================
## = JPA Configuration         =
## ===============================
#spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
#spring.jpa.show-sql=true
#spring.jpa.hibernate.ddl-auto=update

spring.application.name=bartr

# Server port
server.port=8085

# ===============================
# = Data Source Configuration   =
# ===============================
spring.datasource.url=jdbc:postgresql://ep-summer-credit-afy8iyxd-pooler.c-2.us-west-
2.aws.neon.tech/neondb?sslmode=require&channel_binding=require
spring.datasource.username=*********
spring.datasource.password=**********
spring.datasource.driver-class-name=*********
spring.datasource.hikari.connection-timeout=60000

#logging.level.org.springframework=DEBUG



# ===============================
# = JPA Configuration         =
# ===============================
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```