**User Controller**

User Registration (signup):

Validates the uniqueness of the email.

Hashes the password using bcrypt.

Generates a verification code for email verification.

Sends a verification email.

Creates a new user with the provided information and sets the user's status to 'inactive'.

Creates a user profile associated with the user.

Resend Verification Code (resendCode):

Resends a verification code for users with 'inactive' status.

Updates the verification code in the user's document.

Activate Account (activateAccount):

Verifies the provided verification code.

Activates the user's account if the verification code is valid and not expired.

User Login (signin):

Validates user credentials (email and password).

Checks the status of the user's account (inactive, blocked, or active).

Generates a JWT token for authentication.

Google Sign-in (googleSignin):

Verifies the Google ID token.

Handles existing users and new users signing in with Google.

Generates a JWT token for authentication.

Get Single User (getSingleUser):

Retrieves a single user's details, including associated profile and courses, based on the JWT token.

Forgot Password (forgotPassword):

Initiates the password reset process by sending a reset email with a unique token.

Saves the hashed reset token and its expiration time in the user document.

Reset Password (resetPassword):

Handles the reset password functionality.

Verifies the reset token and its expiration time.

Updates the user's password.

Update User Password (updateUserPassword):

Allows the user to update their password.

Validates the old password.

Checks that the new password and confirm password match.

Hashes and updates the password.

Dependencies:

Requires the use of several Node.js packages, including dotenv, bcrypt, crypto, jsonwebtoken, and googleapis.

This code represents the backend logic for user authentication, registration, and password management in a web application.

**AdminController**

Update User Password by Admin (updateUserPasswordByAdmin):

Takes an admin request to update a user's password.

Validates the provided email and ensures it corresponds to an existing user.

Compares the new password with the old password to prevent similarity.

Hashes and updates the user's password.

Responds with a success message upon successful password update.

Store Environment Variables (storeEnvVariables):

Checks if environment variables have already been stored.

If not, stores the provided environment variables along with the editor's ID.

Calls a utility function (getEnvVar) to update the application's environment variables dynamically.

Responds with a success message upon successful storage.

Get Environment Variables (getEnvVariables):

Retrieves the stored environment variables.

Calls a utility function (getEnvVar) to update the application's environment variables dynamically.

Responds with the retrieved environment variables.

Update Environment Variables (updateEnvVariables):

Takes an admin request to update environment variables.

Validates the editor's ID and updates the specified fields.

Calls a utility function (getEnvVar) to update the application's environment variables dynamically.

Responds with the updated environment variables upon success.

Dependencies:

Requires the use of several Node.js packages, including bcrypt for password hashing and getEnvVar utility for updating environment variables dynamically.

These controller functions are responsible for managing user-related operations (password update) and environment variable operations (storage, retrieval, and update) in the web application.

**Assignment Controller**

Add Assignment (addAssignment):

Takes a request to add a new assignment and creates an entry in the "Assignment" model using the provided data from the request body.

Updates the corresponding course document by adding the newly created assignment's ID to the "assignments" array in the "Course" model.

Responds with the result of the course update and a success message indicating the assignment was added successfully.

Get All Assignments (getAllAssignments):

Retrieves all assignments from the "Assignment" model.

Uses populate to include details from the associated "Course" model for each assignment.

Responds with an array containing assignment data, including associated course details.

These controller functions are responsible for handling the addition of assignments and retrieval of all assignments, with associated course details, in the web application. The use of the Mongoose populate function ensures that course details are included when fetching assignments. Additionally, error handling is implemented to handle any unexpected issues during these operations.

**Auth Controller**

Functionality:

The module exports a function named sendPasswordResetEmail.

This function is intended to be a controller for handling the logic related to sending password reset emails.

Dependencies:

The controller relies on the User model, presumably representing user data.

It assumes the existence of functions generateResetToken and sendResetEmail, which are expected to generate a reset token and send a reset email, respectively. These functions are not provided in the snippet.

Process:

Accepts a request (req) and response (res) object, extracting the user's email from the request body.

Attempts to find a user with the provided email using the User model.

If the user is not found, it responds with a 404 status and a message indicating that the user was not found.

If the user is found, it generates a reset token, saves it to the user's document in the database, and sends a password reset email to the user with a link containing the reset token.

If any error occurs during the process, it logs the error and responds with a 500 status and a generic internal server error message.

Export:

The module exports the sendPasswordResetEmail function, making it available for use in other parts of the application.

**BillingAddress Controller**

Store Billing Address:

Function: storeBillingAddress

Route: POST /storeBillingAddress

Logic:

Extracts billing address data from the request body.

Checks if a billing address is already stored for the user (based on userId in the decoded token).

If a billing address is found, responds with a 400 status and an error message.

If not found, creates a new billing address document in the database.

Responds with a success status (200) and the ID of the stored billing address.

Get Billing Address:

Function: getBillingAddress

Route: GET /getBillingAddress

Logic:

Retrieves the billing address associated with the user (based on userId in the decoded token).

If a billing address is not found, responds with a 404 status and an error message.

If found, responds with a success status (200) and the billing address information.

Update Billing Address:

Function: updateBillingAddress

Route: PUT /updateBillingAddress

Logic:

Extracts the fields to be updated from the request body.

Updates the billing address associated with the user (based on userId in the decoded token).

If the billing address is not found, responds with a 404 status and an error message.

If found and updated, responds with a success status (200) and the updated billing address information.

Error Handling:

In case of errors during the execution of each function, a generic error message is sent with a 500 status.

**BookCounseling Controller**

Get All Booked Counseling Sessions:

Function: getAllBookCounselling

Route: GET /getAllBookCounselling

Logic:

Retrieves all records from the BookCounselling model.

Responds with a success status (200) and an array containing all booked counselling sessions.

Book a Free Counselling Session:

Function: bookAFreeCounselling

Route: POST /bookAFreeCounselling

Expected Request Body: { name, email, contactNumber, slots }

Logic:

Extracts information (name, email, contactNumber, and slots) from the request body.

Validates that name, email, and contactNumber are provided; if not, responds with a 403 status and an error message.

Creates a new BookCounselling record in the database with the provided information.

Responds with a success status (200) and a message indicating that the request has been submitted. The team will contact the requester shortly.

File Handling (Commented Out):

The code includes commented-out sections related to file handling (e.g., uploading a resume). This part seems to be disabled, and it's currently not in use.

Error Handling:

In case of errors during the execution of each function, the next function is called, passing the error to the next middleware in the stack.

**Certificate Controller**

Download Certificate:

Function: downloadCertificate

Route: GET /downloadCertificate/:id

Logic:

Extracts the id parameter from the request URL.

Fetches the course details using the Course model.

Loads the certificate template image.

Creates a canvas using the template image dimensions.

Draws the user's name and result on the canvas.

Converts the canvas to a buffer and sends it as a response with content type 'image/png'.

The customized certificate is dynamically generated with the user's name and result.

Upload Certificate Image:

Function: uploadCertificate

Route: POST /uploadCertificate

Expected Request Body: { courseId, certificate }

Logic:

Extracts the courseId and certificate from the request body.

Validates that certificate is provided; if not, responds with a 404 status and an error message.

Converts the base64-encoded certificate image to a buffer.

Resizes the image using the Jimp library (1600 width, maintaining aspect ratio).

Writes the resized image to the server in the /public/images/certificate/ directory.

Updates the Course model with the path to the uploaded certificate image.

Responds with a success status (200) and a message indicating that the certificate has been uploaded successfully.

Error Handling:

In case of errors during the execution of each function, the next function is called, passing the error to the next middleware in the stack.

**CouponCode controller**

Create Coupon Code:

Function: createCouponCode

Route: POST /createCouponCode

Logic:

Creates a new coupon code by inserting the request body into the CouponCode model.

Responds with a success status (200) and a message indicating that the coupon code has been created successfully.

Get All Coupon Codes:

Function: getAllCouponCode

Route: GET /getAllCouponCode

Logic:

Fetches all coupon codes from the CouponCode model and populates the "courseId" field.

Responds with a status of 200 and the array of coupon codes.

Change Coupon Code Status:

Function: updateCouponStatus

Route: PUT /updateCouponStatus/:id

Logic:

Updates the status of a specific coupon code identified by the provided id.

Responds with a success status (200) and a message indicating that the status has been updated successfully.

Delete Coupon Code By ID:

Function: deleteCouponCodeById

Route: DELETE /deleteCouponCodeById/:id

Logic:

Deletes a specific coupon code identified by the provided id.

Responds with a success status (200) and a message indicating that the coupon code has been deleted successfully.

Apply Coupon Code:

Function: applyCouponCode

Route: POST /applyCouponCode

Logic:

Extracts courseId and couponCode from the request body.

Checks if the provided courseId exists.

Checks if the provided couponCode is valid for the given courseId.

Responds with the coupon details if valid; otherwise, responds with an error message.

Error Handling:

Each function includes a try-catch block to handle errors. If an error occurs, it calls the next function, passing the error to the next middleware in the stack.

**Course Controller**

Get Course by ID:

Function: getCourseById

Route: GET /getCourseById/:id

Logic:

Fetches a specific course by its ID.

Responds with the course details if found; otherwise, responds with an error.

Delete Course by ID:

Function: deleteCourseById

Route: DELETE /deleteCourseById/:id

Logic:

Removes the course ID from associated users.

Deletes associated coupon codes, certificates, and assignments.

Deletes the course itself.

Responds with a success status and the deleted course details if successful; otherwise, responds with an error.

Add Course Title:

Function: addCourseTitle

Route: POST /addCourseTitle

Logic:

Processes the cover photo, resizing it and saving it to a specified directory.

Creates a new course with the provided title, price, source path, and processed cover photo.

Responds with the created course details if successful; otherwise, responds with an error.

Upload Course Video:

Function: uploadCourseVideo

Route: POST /uploadCourseVideo

Logic:

Finds the course by ID.

Adds a video to an existing module or creates a new module if it doesn't exist.

Responds with a success message if the video is uploaded successfully; otherwise, responds with an error.

Get All Courses:

Function: getAllCourses

Route: GET /getAllCourses

Logic:

Fetches all courses.

Responds with an array of course details.

Get Top Selling Courses:

Function: getTopSalesCourse

Route: GET /getTopSalesCourse

Logic:

Fetches the top-selling courses based on sales volume.

Responds with an array of course details.

Error Handling:

Each function includes a try-catch block to handle errors. If an error occurs, it calls the next function, passing the error to the next middleware in the stack.

**Course Enroll Controller**

Get Enrolled Course:

Function: getEnrolledCourse

Route: GET /getEnrolledCourse

Logic: Fetches courses that a student has enrolled in.

Response: An array of enrolled courses.

Get Enrolled and Not Refunded Course:

Function: getEnrolledAndNotRefundCourse

Route: GET /getEnrolledAndNotRefundCourse

Logic: Fetches courses that a student has enrolled in and not requested a refund for.

Response: An array of enrolled courses without refund requests.

Get Recent Orders:

Function: getRecentOrders

Route: GET /getRecentOrders

Logic: Fetches the most recent course enrollment orders.

Response: An array of recent course enrollment orders.

Enroll Course by USD (Stripe):

Function: enrollCourseByUSD

Route: POST /enrollCourseByUSD

Logic: Initiates the enrollment process using Stripe for USD transactions.

Response: Redirect URL for Stripe checkout.

Verify Stripe Web-Hook for Course Enrollment:

Function: postStripeWebHook

Route: POST /postStripeWebHook

Logic: Verifies and processes Stripe web-hook events, specifically handling successful checkout session completions.

Response: Success message if the verification is successful.

Enroll Course by INR (Razorpay):

Function: enrollCourseByINR

Route: POST /enrollCourseByINR

Logic: Initiates the enrollment process using Razorpay for INR transactions.

Response: Razorpay order details.

Verify Razorpay Payment:

Function: razorpayVerify

Route: POST /razorpayVerify

Logic: Verifies and processes Razorpay payments, updating course enrollment details accordingly.

Response: Success message if the payment is verified.

Update Refund Request:

Function: updateRefundRequest

Route: PATCH /updateRefundRequest/:id

Logic: Updates the refund request status for a course enrollment.

Response: Success message if the refund request is updated.

Note:

The module also includes imports for various models (e.g., Course, CourseEnroll, Profile, User, Reward, RefundTerms) and external libraries (crypto, stripe, Razorpay).

**Profile Controller**

Get Profile:

Function: getProfile

Route: GET /getProfile

Logic: Retrieves the profile of the currently authenticated user.

Response: The user's profile data.

Update Profile By Id:

Function: updateProfileById

Route: PATCH /updateProfileById/:id

Logic: Updates various aspects of the user profile, such as personal information, address, and avatar.

Response: Success message and the updated profile data.

Get All Students:

Function: getAllStudents

Route: GET /getAllStudents

Logic: Retrieves a list of all students (course enrollees) with optional pagination and search functionality.

Response: An array of student data and the total count.

Get All Users:

Function: getAllUsers

Route: GET /getAllUsers

Logic: Retrieves a list of all users (profiles) with optional pagination and search functionality.

Response: An array of user data and the total count.

Update User Status By Id:

Function: updateUserStatusById

Route: PATCH /updateUserStatusById/:id

Logic: Updates the status (e.g., active, inactive) of a user by their ID.

Response: Success message and the updated user data.

Delete User By Id:

Function: deleteUserById

Route: DELETE /deleteUserById/:id

Logic: Deletes a user and related data (profile, course enrollments, certificates) by their ID.

Response: Success message indicating that the user has been deleted.

Note:

The module includes imports for various models (e.g., Profile, User, CourseEnroll, Course, Certificate) and external libraries (Jimp, path).

**RefundTerms Controller**

Create Refund Terms:

Function: createRefundTerms

Route: POST /createRefundTerms

Logic: Creates new refund terms with specified registration fees and return window.

Response: The created refund terms and a success message.

Get Refund Terms:

Function: getRefundTerms

Route: GET /getRefundTerms

Logic: Retrieves the existing refund terms.

Response: The existing refund terms.

Update Registration Fees:

Function: updateRegistrationFees

Route: PATCH /updateRegistrationFees

Logic: Updates the registration fees in the refund terms. If no refund terms exist, it creates a new document with default values.

Response: Success message indicating that the registration fees have been updated.

Update Return Window:

Function: updateReturnWindow

Route: PATCH /updateReturnWindow

Logic: Updates the return window in the refund terms. If no refund terms exist, it creates a new document with default values.

Response: Success message indicating that the return window has been updated.

Note:

The module includes an import for the RefundTerms model.

It handles scenarios where there may not be an existing document in the RefundTerms collection by creating one with default values.

**Reward COntroller**

Create or Update Reward Points:

Function: createReward

Route: POST /createReward

Logic: Calculates and updates the user's reward points based on the dollars spent. Ensures the dollars spent is a valid positive number.

Response: The updated/created reward data and a success message.

Get All Rewards:

Function: getAllRewards

Route: GET /getAllRewards

Logic: Retrieves all reward information for the authenticated user.

Response: The user's reward data.

Redeem Reward Points:

Function: reedemRewardPoints

Route: POST /reedemRewardPoints

Logic: Redeems reward points for a purchase, calculating the discount and updating the user's reward points.

Response: Success message, reward discount amount, and the final price after applying the discount.

Delete Reward By ID:

Function: deleteRewardById

Route: DELETE /deleteRewardById/:id

Logic: Deletes a user's reward by ID.

Response: The deleted reward data and a success message.

Reward Functions:

The module imports createOrUpdateRewardPoints and redeemRewardPointsDeduction from a utility file (rewardFunctions.js), suggesting that the actual reward-related calculations and interactions with the database are implemented in that file.

Note:

The module includes an import for the Reward model.

There are commented-out console log statements and code, indicating possible debugging or development remnants.