### LAMBDA FUNCTION -> **DashbordEdit**

API URL -> <https://7efwp1v3ed.execute-api.us-east-1.amazonaws.com/dashedit/DashbordEdit>

Resource -> **DashbordAPI's ->** [**DashbordEdit**](https://us-east-1.console.aws.amazon.com/apigateway/main/apis/7efwp1v3ed/resources?api=7efwp1v3ed&region=us-east-1)

Stage -> dashedit

### 1. Role of the API

This API handles updating a quiz's title and completion status in the database.

### 2. Functioning

* **Token Verification**: Validates the JWT token to authenticate the request.
* **CORS Handling**: Sets appropriate headers to handle Cross-Origin Resource Sharing (CORS).
* **Database Connection**: Ensures a consistent MongoDB connection.
* **Quiz Update**: Updates the quiz title and/or completion status based on the provided \_id.
* **Response Handling**: Returns success or error messages based on the operation's outcome.

### 3. Request Body

**Required:**

{

"body": "{\"\_id\":\"66966d24afdfc5ae21562500\",\"quizTitle\":\"LastTesting\",\"isCompleted\":false}",

"headers": {

"Authorization": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2NjcxYWU1NTZhNWY0YTRjNWNhMzMzYjUiLCJlbWFpbCI6InNhaW5pcHM5NDE0NjZAZ21haWwuY29tIiwiaWF0IjoxNzIxMTk0NTEyLCJleHAiOjE3MjEyMTI1MTJ9.L1xBwgXYsXitQrcPy8mok2OT5E\_\_UfWzFM4WH63yuAI",

"Content-Type": "application/json"

}

}

quizTitle -> Optional

isCompleted -> Optional

### 4. Response

**Success Response:**

* **Status Code**: 200

{

"statusCode": 200,

"headers": {

"Access-Control-Allow-Origin": "\*",

"Access-Control-Allow-Methods": "OPTIONS,POST",

"Access-Control-Allow-Headers": "Content-Type,Authorization"

},

"body": "{\"message\":\"Data edited successfully\"}"

}

**Error Responses:**

* **Status Code**: 401
  + **Body**: {"error": "Authorization token missing"}
  + **Body**: {"error": "Invalid or expired token"}
* **Status Code**: 400
  + **Body**: {"error": "Invalid request payload"}
* **Status Code**: 404
  + **Body**: {"error": "Quiz not found"}
* **Status Code**: 500
  + **Body**: {"error": "Internal Server Error"}

### 5. Logic

1. **Token Verification**: Validates the JWT token to ensure the user is authorized to perform the operation.
2. **CORS Headers**: Sets headers to allow CORS requests from any origin.
3. **Database Connection**: Ensures a consistent connection to MongoDB using Mongoose.
4. **Request Validation**: Ensures required fields like \_id are present and valid.
5. **Quiz Update**: Updates the quiz title and/or completion status based on the provided \_id.
6. **Response Creation**: Generates and returns appropriate responses based on the success or failure of the operation.

### 6. Dependencies

* **mongoose**: MongoDB object modeling tool.
* **jsonwebtoken**: Library for generating and verifying JSON Web Tokens (JWT).

CODE ->

const mongoose = require('mongoose');

const jwt = require('jsonwebtoken');

const Quiz = require('./question.js'); // Ensure this path is correct for your Quiz model

// Environment Variables

const uri = process.env.MONGODB\_URI; // MongoDB connection string from environment variable

const jwtSecret = process.env.JWT\_SECRET\_KEY; // JWT secret key from environment variable

let cachedDb = null;

// Function to connect to MongoDB

async function connectToDatabase() {

if (cachedDb && mongoose.connection.readyState === 1) {

return cachedDb;

}

try {

cachedDb = await mongoose.connect(uri, { useNewUrlParser: true, useUnifiedTopology: true });

console.log('MongoDB connected');

return cachedDb;

} catch (error) {

console.error('MongoDB connection error:', error);

throw new Error('Could not connect to the database');

}

}

// JWT verification function

function verifyToken(token) {

try {

return jwt.verify(token, jwtSecret);

} catch (error) {

throw new Error('Invalid or expired token');

}

}

exports.handler = async (event) => {

// CORS headers

const headers = {

'Access-Control-Allow-Origin': '\*',

'Access-Control-Allow-Methods': 'OPTIONS,POST',

'Access-Control-Allow-Headers': 'Content-Type,Authorization'

};

if (event.httpMethod === 'OPTIONS') {

// Handle CORS preflight request

return {

statusCode: 200,

headers,

body: JSON.stringify({ message: 'CORS preflight' })

};

}

try {

// Extract JWT token from headers and validate

const authHeader = event.headers.Authorization || event.headers.authorization;

if (!authHeader) {

return {

statusCode: 401,

headers,

body: JSON.stringify({ error: 'Authorization token missing' }),

};

}

const token = authHeader.replace('Bearer ', '');

if (!token) {

return {

statusCode: 401,

headers,

body: JSON.stringify({ error: 'Authorization token missing' }),

};

}

// Verify JWT token

const decoded = verifyToken(token);

// Extract quiz ID and update details from the event body

const { \_id, quizTitle, isCompleted } = JSON.parse(event.body);

// Validate required fields

if (!\_id || (quizTitle === undefined && isCompleted === undefined)) {

return {

statusCode: 400,

headers,

body: JSON.stringify({ error: 'Invalid request payload' }),

};

}

// Connect to MongoDB

await connectToDatabase();

// Find the quiz by ID

const quiz = await Quiz.findById(\_id);

if (!quiz) {

return {

statusCode: 404,

headers,

body: JSON.stringify({ error: 'Quiz not found' }),

};

}

// Update quiz only if necessary

if (quizTitle !== undefined && quiz.quizTitle !== quizTitle) {

quiz.quizTitle = quizTitle;

}

if (isCompleted !== undefined && quiz.isCompleted !== isCompleted) {

quiz.isCompleted = isCompleted;

}

// Save the updated quiz

await quiz.save();

return {

statusCode: 200,

headers,

body: JSON.stringify({ message: 'Data edited successfully' })

};

} catch (error) {

console.error('Error:', error.message);

return {

statusCode: 500,

headers,

body: JSON.stringify({ error: 'Internal Server Error' }),

};

}

};

MODEL -> question.js

const mongoose = require('mongoose');

// Define schema for multiple choice questions

const MCQSchema = new mongoose.Schema({

question: {

type: String,

required: false,

},

questionImageLink: {

type: String,

required: false,

},

options: [{

answer: {

type: String,

required: false,

},

answerImageLink: {

type: String,

required: false,

}

}],

correctAnswer: {

type: String,

required: true,

},

description: {

type: String,

required: false

},

version: {

type: Number,

default: 1

}

});

// Define schema for descriptive questions

const DescriptiveSchema = new mongoose.Schema({

question: {

type: String,

required: false,

},

questionImageLink: {

type: String,

required: false,

},

answer: {

type: String,

required: false

},

answerImageLink: {

type: String,

required: false,

},

version: {

type: Number,

default: 1

}

});

// Define main quiz schema

const QuizSchema = new mongoose.Schema({

quizTitle: {

type: String,

required: true,

},

creatorName: {

type: String,

required: false

},

creatorEmail: {

type: String,

required: true

},

isCompleted: {

type: Boolean,

required: false,

default: false

},

status: {

type: Boolean,

required: false,

default: true

},

preVersionID: {

type: [mongoose.Schema.Types.ObjectId],

default: []

},

mcqQuizz: [MCQSchema], // Array of multiple choice questions

descriptiveQuizz: [DescriptiveSchema], // Array of descriptive questions

quizDuration: {

type: Number,

required: false

}

}, { timestamps: { createdAt: 'createdAt' } });

// Pre-save middleware to set creatorName from User2's fullname

QuizSchema.pre('save', async function(next) {

if (this.isNew || this.isModified('creatorEmail')) {

const User2 = mongoose.model('User2');

const user = await User2.findOne({ email: this.creatorEmail }).exec();

if (user) {

this.creatorName = user.fullname;

} else {

const error = new Error('User not found');

error.statusCode = 404;

return next(error);

}

}

next();

});

// Middleware to handle versioning of questions in mcqQuizz and descriptiveQuizz

QuizSchema.pre('save', async function(next) {

if (!this.isNew) {

const originalQuiz = await mongoose.model('Quiz').findById(this.\_id).exec();

if (originalQuiz) {

this.mcqQuizz.forEach(mcq => {

const originalMCQ = originalQuiz.mcqQuizz.id(mcq.\_id);

if (originalMCQ && !mcq.\_id.equals(originalMCQ.\_id)) {

mcq.version = originalMCQ.version + 1;

}

});

this.descriptiveQuizz.forEach(dq => {

const originalDQ = originalQuiz.descriptiveQuizz.id(dq.\_id);

if (originalDQ && !dq.\_id.equals(originalDQ.\_id)) {

dq.version = originalDQ.version + 1;

}

});

// Handle deleted questions by adding their IDs to preVersionID

originalQuiz.mcqQuizz.forEach(originalMCQ => {

if (!this.mcqQuizz.id(originalMCQ.\_id)) {

this.preVersionID.push(originalMCQ.\_id);

}

});

originalQuiz.descriptiveQuizz.forEach(originalDQ => {

if (!this.descriptiveQuizz.id(originalDQ.\_id)) {

this.preVersionID.push(originalDQ.\_id);

}

});

}

}

next();

});

// Create a model using the schema

const Quiz = mongoose.model('Quiz', QuizSchema);

module.exports = Quiz;