### LAMBDA FUNCTION -> **quizzQuestionService**

API URL -> <https://ee4pmf8ys1.execute-api.us-east-1.amazonaws.com/add/Question>

Resource -> **quizzQuestionService ->** [**Question**](https://us-east-1.console.aws.amazon.com/apigateway/main/apis/ee4pmf8ys1/resources?api=ee4pmf8ys1&region=us-east-1)

Stage -> add

### 1. Role of the API

The primary role of this API is to add new MCQ questions to an existing quiz in the database. It validates the JWT token, checks for the presence of required fields, processes the MCQ questions, and updates the existing quiz document.

### 2. Functioning

* **Authentication**: Validates the JWT token from the request header to ensure the user is authorized.
* **Database Connection**: Connects to MongoDB using Mongoose.
* **Request Validation**: Validates the presence of required fields (quizTitle and mcqQuizz) and the correctness of the MCQ objects.
* **Quiz Retrieval and Update**: Finds the existing quiz by quizTitle and creatorEmail, processes the MCQ objects, and updates the quiz document in the database.

**3. Request Body**

**{**

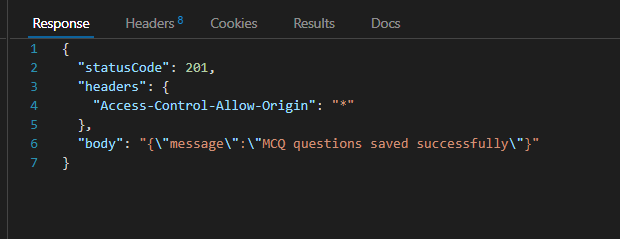
**"headers": {**

**"Authorization": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2NjcxYWU1NTZhNWY0YTRjNWNhMzMzYjUiLCJlbWFpbCI6InNhaW5pcHM5NDE0NjZAZ21haWwuY29tIiwiaWF0IjoxNzIxMTMwODY2LCJleHAiOjE3MjExNDg4NjZ9.5gpUxGLTlmt8ClRtwTv-ePX8sH2c6ScaTIrLc9bQhBM"**

**},**

**"body": "{\"quizTitle\":\"PrashantTesffft\",\"mcqQuizz\":[{\"question\":\"What is the capital of France?\",\"questionImageLink\":\"www.google.com\",\"options\":[{\"answer\":\"Paris\",\"answerImageLink\":\"www.google.com\"},{\"answer\":\"London\",\"answerImageLink\":\"www.google.com\"}],\"correctAnswer\":2,\"description\":\"The capital of France is Paris.\"},{\"question\":\"Which planet is known as the Red Planet?\",\"questionImageLink\":\"\",\"options\":[{\"answer\":\"Mars\",\"answerImageLink\":\"www.google.com\"},{\"answer\":\"Jupiter\",\"answerImageLink\":\"www.google.com\"}],\"correctAnswer\":2,\"description\":\"Mars is often called the 'Red Planet'.\"}]}"**

**}**

****

### 4. Response

**Success Response:**

{

"statusCode": 201,

"headers": {

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

},

"body": "{\"message\":\"MCQ questions saved successfully\"}"

}

**Status:401**

{

"statusCode": 401,

"error": "Unauthorized: No token provided"

}

{

"statusCode": 401,

"error": "Unauthorized: Token expired or invalid"

}

**Status:400**

**{**

**"statusCode": 400,**

**"error": "Token does not contain an email"**

**}**

**{**

**"statusCode": 400,**

**"error": "quizTitle and mcqQuizz are required fields"**

**}**

**{**

**"statusCode": 400,**

**"error": "Each MCQ object must have either question or questionImageLink defined"**

**}**

**Status: 404**

**{**

**"statusCode": 404,**

**"error": "Quiz with the provided title does not exist. Please provide a valid quiz title."**

**}**

**Status:500**

**{**

**"statusCode": 500,**

**"error": "Internal Server Error: Error message"**

**}**

### 5. Logic

1. **Database Connection**: Checks if there's a cached connection to the database. If not, establishes a new connection and caches it.
2. **CORS Handling**: For OPTIONS requests, returns appropriate headers to handle CORS preflight requests.
3. **Token Validation**: Checks for the presence of the JWT token and validates it.
4. **Request Parsing and Validation**: Parses the request body and validates required fields (quizTitle and mcqQuizz).
5. **Quiz Retrieval**: Searches for an existing quiz with the provided quizTitle and creatorEmail.
6. **MCQ Validation**: Validates each MCQ object for the presence of either question or questionImageLink, at least two options, and a valid correctAnswer index.
7. **Option ID Assignment**: Assigns \_id to each option if not already present.
8. **Correct Answer Mapping**: Converts 1-based correct answer index to 0-based and assigns the option ID.
9. **Quiz Update**: Adds the new MCQ questions to the existing quiz and saves it to the database.
10. **Response Creation**: Creates and returns appropriate success or error responses based on the outcome of the operations.

### 6. Dependencies

* **mongoose**: For MongoDB object modeling.
* **jsonwebtoken**: For JWT token validation.
* **question.js**: Quiz schema model.

CODE ->

const mongoose = require('mongoose');

const Quiz = require('./question.js');

const jwt = require('jsonwebtoken');

const JWT\_SECRET\_KEY = process.env.JWT\_SECRET\_KEY;

let cachedDb = null;

const connectToDatabase = async () => {

if (cachedDb) {

return cachedDb;

}

try {

const connection = await mongoose.connect(process.env.MONGODB\_URI);

cachedDb = connection.connection.db;

return cachedDb;

} catch (error) {

throw new Error('Error connecting to MongoDB: ' + error.message);

}

};

exports.handler = async (event, context) => {

context.callbackWaitsForEmptyEventLoop = false;

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

return {

statusCode: 204,

headers: {

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

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

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

},

};

}

try {

await connectToDatabase();

const token = event.headers.Authorization;

if (!token) {

return createErrorResponse(401, 'Unauthorized: No token provided');

}

let decoded;

try {

decoded = jwt.verify(token, JWT\_SECRET\_KEY);

} catch (error) {

return createErrorResponse(401, 'Unauthorized: Token expired or invalid');

}

if (!decoded.email) {

return createErrorResponse(400, 'Token does not contain an email');

}

const creatorEmail = decoded.email.toLowerCase();

const requestBody = JSON.parse(event.body);

const { quizTitle, mcqQuizz } = requestBody;

if (!quizTitle || !mcqQuizz) {

return createErrorResponse(400, 'quizTitle and mcqQuizz are required fields');

}

let existingQuiz = await Quiz.findOne({ quizTitle, creatorEmail });

if (!existingQuiz) {

return createErrorResponse(404, 'Quiz with the provided title does not exist. Please provide a valid quiz title.');

}

// Validate and process MCQ objects before saving

for (let mcq of mcqQuizz) {

if (!mcq.question && !mcq.questionImageLink) {

return createErrorResponse(400, 'Each MCQ object must have either question or questionImageLink defined');

}

if (!mcq.options || mcq.options.length < 2) {

return createErrorResponse(400, 'Each MCQ object must have at least two options');

}

if (typeof mcq.correctAnswer !== 'number' || mcq.correctAnswer < 1 || mcq.correctAnswer > mcq.options.length) {

return createErrorResponse(400, 'Correct answer must be a valid option index');

}

// Assign \_id to each option if not already present

mcq.options.forEach(option => {

if (!option.\_id) {

option.\_id = new mongoose.Types.ObjectId();

}

});

// Convert 1-based index to 0-based and assign option ID to correctAnswer

const correctAnswerIndex = mcq.correctAnswer - 1;

mcq.correctAnswer = mcq.options[correctAnswerIndex].\_id;

}

existingQuiz.mcqQuizz.push(...mcqQuizz);

await existingQuiz.save();

return createSuccessResponse(201, 'MCQ questions saved successfully');

} catch (error) {

if (error.name === 'ValidationError') {

return createErrorResponse(400, 'Bad Request: ' + error.message);

}

return createErrorResponse(500, 'Internal Server Error: ' + error.message);

}

};

function createErrorResponse(statusCode, errorMessage) {

return {

statusCode,

headers: {

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

},

body: JSON.stringify({ error: errorMessage })

};

}

function createSuccessResponse(statusCode, message) {

return {

statusCode,

headers: {

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

},

body: JSON.stringify({ message })

};

}

**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

}, { 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;