### LAMBDA FUNCTION -> **dashbordQuizzinfo**

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

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

Stage -> singleview

### 1. Role of the API

This API handles the retrieval of a quiz by its \_id from the database and returns its details along with formatted fields.

### 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 Retrieval**: Retrieves the quiz by its \_id and formats the response.
* **Response Handling**: Returns the quiz details or appropriate error messages

### 3. Request Body

**Required:**

{

"headers": {

"Authorization": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2NjcxYWU1NTZhNWY0YTRjNWNhMzMzYjUiLCJlbWFpbCI6InNhaW5pcHM5NDE0NjZAZ21haWwuY29tIiwiaWF0IjoxNzIxMTgwOTE2LCJleHAiOjE3MjExOTg5MTZ9.zhtMlw5sHYBNkG8NQCANzPZ6tg-I709KpEvSvF93dyA",

"Content-Type": "application/json"

},

"body": "{\"\_id\":\"66966d24afdfc5ae21562500\"}"

}

### 4. Response

**Success Response:**

* **Status Code**: 200

Data retrieve successfully

**Error Responses:**

* **Status Code**: 401
  + **Body**: {"error": "Unauthorized: No token provided"}
  + **Body**: {"error": "Unauthorized: Invalid token"}
* **Status Code**: 400
  + **Body**: {"error": "Bad Request: Empty payload"}
  + **Body**: {"error": "Bad Request: Quiz \_id is required"}
* **Status Code**: 404
  + **Body**: {"error": "Not Found: 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 Retrieval**: Retrieves the quiz by its \_id and formats the response, including changing \_id to mcqQuestion\_id and descriptiveQuestion\_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'); // Adjust the path to your Quiz model file

// MongoDB connection URI

const mongoUri = process.env.MONGODB\_URI; // Replace with your MongoDB connection string

// JWT secret key

const jwtSecret = process.env.JWT\_SECRET\_KEY; // Replace with your JWT secret key

// MongoDB connection

let conn = null;

async function connectToDatabase(uri) {

if (conn == null) {

conn = await mongoose.connect(uri);

}

return conn;

}

exports.handler = async (event) => {

// Define CORS headers

const corsHeaders = {

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

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

};

try {

// Connect to MongoDB

await connectToDatabase(mongoUri);

// Validate if the Authorization header is present

const token = event.headers.Authorization;

if (!token) {

return {

statusCode: 401,

headers: corsHeaders,

body: { error: 'Unauthorized: No token provided' },

};

}

// Verify JWT

let decoded;

try {

decoded = jwt.verify(token, jwtSecret);

} catch (err) {

return {

statusCode: 401,

headers: corsHeaders,

body: { error: 'Unauthorized: Invalid token' },

};

}

// Validate the payload

if (!event.body) {

return {

statusCode: 400,

headers: corsHeaders,

body: { error: 'Bad Request: Empty payload' },

};

}

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

if (!\_id) {

return {

statusCode: 400,

headers: corsHeaders,

body: { error: 'Bad Request: Quiz \_id is required' },

};

}

// Find the quiz by \_id

const quiz = await Quiz.findById(\_id).populate('mcqQuizz').populate('descriptiveQuizz').exec();

if (!quiz) {

return {

statusCode: 404,

headers: corsHeaders,

body: { error: 'Not Found: Quiz not found' },

};

}

// Convert the Mongoose document to a plain object and format isCompleted and status

const quizObject = quiz.toObject();

quizObject.isCompleted = quiz.isCompleted ? "Completed" : "Incomplete";

quizObject.status = quiz.status ? "Active" : "Inactive";

// Append mcqQuestion\_id and descriptiveQuestion\_id and remove the original \_id

quizObject.mcqQuizz = quizObject.mcqQuizz.map(mcq => {

const { \_id, ...rest } = mcq;

return {

...rest,

mcqQuestion\_id: \_id,

};

});

quizObject.descriptiveQuizz = quizObject.descriptiveQuizz.map(descriptive => {

const { \_id, ...rest } = descriptive;

return {

...rest,

descriptiveQuestion\_id: \_id,

};

});

// Return the quiz details

return {

statusCode: 200,

headers: corsHeaders,

body: quizObject,

};

} catch (error) {

console.error(error);

return {

statusCode: 500,

headers: corsHeaders,

body: { 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;