### LAMBDA FUNCTION -> **userDashboard**

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

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

Stage -> userdetails

### 1. Role of the API

This API handles the retrieval of quizzes based on various filters and pagination, returning quiz details along with pagination information.

### 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 quizzes based on filters like userEmail, isCompleted, and status, and handles pagination.
* **Response Handling**: Returns the list of quizzes along with pagination details or appropriate error messages.

**3. Request Body**

{

"body": "{\"userEmail\":\"sainips941466@gmail.com\",\"page\":3,\"limit\":10}",

"headers": {

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

"Content-Type": "application/json"

}

}

### 4. Response

**Success Response:**

* **Status Code**: 200

**Error Responses:**

* **Status Code**: 401
  + **Body**: {"error": "Unauthorized: Missing token"}
  + **Body**: {"error": "Unauthorized: Invalid token"}
* **Status Code**: 400
  + **Body**: {"message": "Empty request body."}
  + **Body**: {"error": "Bad Request: Invalid JSON"}
  + **Body**: {"error": "Bad Request: Missing userEmail in request body"}
  + **Body**: {"error": "Bad Request: Invalid pagination parameters"}
* **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 userEmail, page, and limit are present and valid.
5. **Quiz Retrieval**: Retrieves quizzes based on filters and pagination parameters.
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 jwt = require('jsonwebtoken');

const mongoose = require('mongoose');

const Quiz = require('./question'); // Import your Quiz model

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

const MONGODB\_URI = process.env.MONGODB\_URI;

if (!JWT\_SECRET\_KEY || !MONGODB\_URI) {

throw new Error('Environment variables JWT\_SECRET\_KEY and MONGODB\_URI must be set.');

}

mongoose.connect(MONGODB\_URI)

mongoose.connection.on('error', err => {

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

});

mongoose.connection.once('open', () => {

console.log('MongoDB connected');

});

const headers = {

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

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

'Access-Control-Allow-Credentials': true,

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

};

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

if (!event.body) {

return {

statusCode: 400,

headers,

body: JSON.stringify({ message: "Empty request body." })

};

}

const token = event.headers.Authorization;

if (!token) {

return {

statusCode: 401,

headers,

body: JSON.stringify({ error: 'Unauthorized: Missing token' }),

};

}

let decoded;

try {

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

} catch (error) {

console.error('Error verifying token:', error);

return {

statusCode: 401,

headers,

body: JSON.stringify({ error: 'Unauthorized: Invalid token' }),

};

}

let requestBody;

try {

requestBody = JSON.parse(event.body);

} catch (error) {

console.error('Error parsing request body:', error);

return {

statusCode: 400,

headers,

body: JSON.stringify({ error: 'Bad Request: Invalid JSON' }),

};

}

const userEmail = requestBody.userEmail && requestBody.userEmail.toLowerCase();

const page = parseInt(requestBody.page, 10);

const limit = parseInt(requestBody.limit, 10);

const isCompleted = requestBody.isCompleted; // Expecting "Completed" or "Incomplete"

const status = requestBody.status; // Expecting "Active" or "Inactive"

if (!userEmail) {

return {

statusCode: 400,

headers,

body: JSON.stringify({ error: 'Bad Request: Missing userEmail in request body' }),

};

}

if (!Number.isInteger(page) || page <= 0 || !Number.isInteger(limit) || limit <= 0) {

return {

statusCode: 400,

headers,

body: JSON.stringify({ error: 'Bad Request: Invalid pagination parameters' }),

};

}

const skip = (page - 1) \* limit;

let filter = { creatorEmail: userEmail };

if (isCompleted) {

filter.isCompleted = isCompleted === "Completed";

}

if (status) {

filter.status = status === "Active";

}

try {

const totalQuizzes = await Quiz.countDocuments(filter).exec();

const quizzes = await Quiz.find(filter)

.select('\_id quizTitle creatorName createdAt updatedAt mcqQuizz descriptiveQuizz isCompleted status')

.skip(skip)

.limit(limit)

.exec();

const response = {

quizzes: quizzes.map(quiz => ({

quizzId: quiz.\_id,

quizTitle: quiz.quizTitle,

creatorName: quiz.creatorName,

createdAt: quiz.createdAt,

updatedAt: quiz.updatedAt,

totalQuestions: quiz.mcqQuizz.length + quiz.descriptiveQuizz.length,

isCompleted: quiz.isCompleted ? "Completed" : "Incomplete",

status: quiz.status ? "Active" : "Inactive"

})),

pagination: {

totalQuizzes,

totalPages: Math.ceil(totalQuizzes / limit),

currentPage: page

}

};

return {

statusCode: 200,

headers,

body: response

};

} catch (err) {

console.error('Database query error:', err);

return {

statusCode: 500,

headers,

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

};

}

};

MOODEL ->

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;