### LAMBDA FUNCTION NAME -> **quizzIdentity**

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

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

Stage name -> info

### 1. Role of API

This API handles the creation of a new quiz. It authenticates the user using a JWT token, ensures that the quiz title is unique for the user, and then saves the quiz to the MongoDB database.

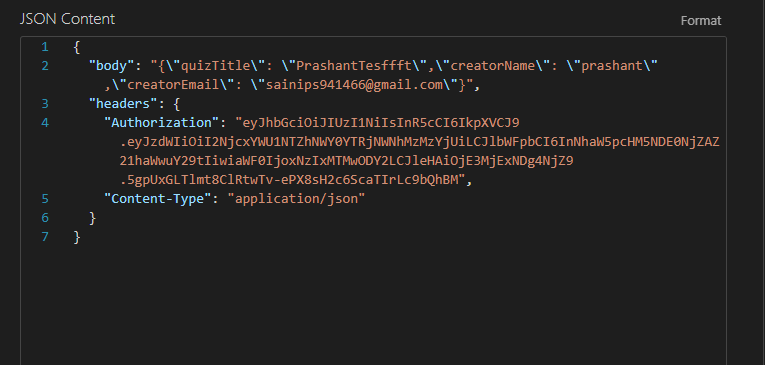
### 2. Functioning

The AWS Lambda function connects to a MongoDB database, validates the incoming request, checks the user's credentials via a JWT token, ensures the uniqueness of the quiz title for the user, and then saves the quiz to the database.

### 3. Request Body

The request body must be a JSON object with the following fields:

* quizTitle (String, required): The title of the quiz.
* mcqQuizz (Array, required): An array of multiple-choice questions.



### 4. Response

The API provides the following responses:

* **201 Created**: The quiz is successfully created.

A screenshot of a computer

Description automatically generated

**204 No Content**: Preflight request for CORS. No body is returned for this status code.

**400 Bad Request**: The request is missing required fields or contains invalid data.

json

**401 Unauthorized**: The JWT token is expired or invalid.

**500 Internal Server Error**: An error occurred on the server while processing the request.

### 5. Logic

1. **Connect to Database**: The function ensures a consistent MongoDB connection across Lambda invocations.
2. **Handle Preflight Requests**: The function handles preflight CORS requests by returning a 204 status code.
3. **Authenticate User**: The function verifies the JWT token from the Authorization header. If invalid or expired, it returns a 401 error.
4. **Parse Request Body**: The request body is parsed to extract the quizTitle and mcqQuizz fields.
5. **Ensure Unique Quiz Title**: The function checks if a quiz with the same title exists for the user and modifies the title if necessary to ensure uniqueness.
6. **Create Quiz**: The function creates a new quiz instance with the provided data and the user's email.
7. **Save Quiz**: The function saves the new quiz to the MongoDB database.
8. **Return Success Response**: The function returns a 201 status code with the new quiz title.
9. **Handle Errors**: If any errors occur during processing, appropriate error responses are returned.

### 6. Dependencies

* **AWS Lambda**: The function is designed to run as an AWS Lambda function.
* **MongoDB**: The function interacts with a MongoDB database to store quiz information.
* **Mongoose**: The function uses Mongoose for MongoDB object modeling and database interaction.
* **jsonwebtoken**: The function uses the jsonwebtoken library to verify JWT tokens.
* **Node.js**: The function is written in Node.js and requires the appropriate Node.js runtime.

CODE ->

//updated code

const mongoose = require('mongoose');

const Quiz = require('./question.js'); // Adjust the path as necessary

const User2 = require('./User'); // Adjust the path as necessary

const jwt = require('jsonwebtoken');

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

// Ensure consistent MongoDB connection across Lambda invocations

let cachedDb = null;

const connectToDatabase = async () => {

if (cachedDb) {

return Promise.resolve(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; // Ensure Lambda doesn't wait for event loop to be empty

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

// Handle preflight requests

return {

statusCode: 204,

headers: {

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

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

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

},

};

}

try {

await connectToDatabase(); // Connect to MongoDB

const token = event.headers.Authorization;

let decoded;

try {

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

// If token is valid, proceed with processing the request

} catch (error) {

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

return {

statusCode: 401, // Unauthorized

headers: {

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

},

body: JSON.stringify({ error: 'Unauthorized: token expired or invalid' }),

};

}

// Ensure the token contains an email

if (!decoded.email) {

console.error('Token does not contain an email');

return {

statusCode: 400, // Bad Request

headers: {

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

},

body: JSON.stringify({ error: 'Bad Request: Token does not contain an email' }),

};

}

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

let { quizTitle, mcqQuizz } = requestBody;

const creatorEmail = decoded.email.toLowerCase();

// Check if a quiz with the same title exists for the user

let newQuizTitle = quizTitle;

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

let suffix = 1;

while (existingQuiz) {

newQuizTitle = `${quizTitle}${suffix}`;

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

suffix++;

}

// Create a new quiz instance with provided data

const newQuiz = new Quiz({

quizTitle: newQuizTitle,

creatorEmail, // Use the lowercase email for storing

mcqQuizz

});

// Save the new quiz to MongoDB

const savedQuiz = await newQuiz.save();

console.log('Quiz created successfully:', savedQuiz);

return {

statusCode: 201, // Created

headers: {

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

},

body: JSON.stringify({ message: 'Quiz created successfully', quizTitle: newQuizTitle }),

};

} catch (error) {

console.error('Error creating quiz:', error);

// Check for validation errors

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

return {

statusCode: 400, // Bad Request

headers: {

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

},

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

};

}

return {

statusCode: 500, // Internal Server Error

headers: {

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

},

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

};

}

};

We use two model here -> User.js

const mongoose = require('mongoose');

const crypto = require('crypto');

const userSchema = new mongoose.Schema({

fullname: { type: String, required: true, unique: false },

email: { type: String, required: true, unique: true },

// contact: { type: String, required: false, unique: true },

password: { type: String, required: true, unique: false },

InstituteName: { type: String, required: true, unique: false },

createdAt: { type: Date, default: Date.now }

});

// Middleware to convert email to lowercase before saving

userSchema.pre('save', function(next) {

const user = this;

// Convert email to lowercase (if it exists and is modified)

if (user.email && user.isModified('email')) {

user.email = user.email.toLowerCase();

}

// Check if password field is modified or new

if (!user.isModified('password')) {

return next();

}

// Hash the password using MD5 (not recommended for real-world use)

const md5 = crypto.createHash('md5');

md5.update(user.password);

user.password = md5.digest('hex');

next();

});

module.exports = mongoose.model('User2', userSchema);

**question.js ->**

**//updated to latest quiz schem with updated options array and quiz Duration field.**

**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;**