### LAMBDA FUNCTION -> **submitQuestionValidation**

API URL ->

<https://ybkfar4y6i.execute-api.us-east-1.amazonaws.com/s2/submitQuestionValidation>

Resource -> **StudentAPI  ->** [**submitQuestionValidation**](https://us-east-1.console.aws.amazon.com/apigateway/main/apis/ybkfar4y6i/resources?api=ybkfar4y6i&region=us-east-1)

Stage -> s2

**Role of the API**:

* This API endpoint processes and updates student responses to both MCQ and descriptive questions.
* It verifies JWT tokens for authorization, connects to MongoDB to fetch and update student records, and calculates not attempted and correct questions.
* It handles CORS pre-flight requests and separates received questions into MCQs and descriptive categories for processing.

**Functioning**:

* Verifies the JWT token provided in the request headers for authorization using jsonwebtoken.
* Connects to MongoDB using the URI provided in the MONGODB\_URI environment variable and ensures the connection state is maintained.
* Handles CORS pre-flight requests (OPTIONS method) to allow CORS headers.
* Parses the request body JSON to extract applicationNumber and allQuestions.
* Finds the student document in MongoDB using Student.findOne.
* Separates allQuestions into MCQs (questions with optionId) and descriptive questions (questions with answer).
* Updates the AttemptedQuestion, NotAttemptedQuestion, and CorrectQuestion fields within the MCQ and descriptiveQuestions sections of the student document based on the received questions and their status.
* Marks modified fields (MCQ.AttemptedQuestion, MCQ.NotAttemptedQuestion, MCQ.CorrectQuestion, descriptiveQuestions.AttemptedQuestion, descriptiveQuestions.NotAttemptedQuestion) before saving the changes.
* Saves the modified Student document back to MongoDB and returns a response indicating whether changes were detected and updated or if no changes were detected.

**Request Body**:

* The API expects a JSON object in the request body with the following fields:
  + applicationNumber: Application number of the student.
  + allQuestions: Array of objects representing questions attempted by the student, with each object containing questionId, optionId (for MCQs), or answer (for descriptive questions).

 **Response**:

* **200 OK**: If changes were detected and successfully updated, returns a success message along with details of changes, not attempted MCQs and descriptive questions, and correct MCQs.
* **200 OK**: If no changes were detected, returns a message indicating no changes along with details of not attempted MCQs and descriptive questions, and correct MCQs.
* **401 Unauthorized**: If the JWT token is missing or invalid.
* **404 Not Found**: If the student with the specified applicationNumber is not found in the database.
* **500 Internal Server Error**: If there's any server-side issue during database operations.

**Dependencies**:

* jsonwebtoken: For generating and verifying JSON Web Tokens (JWT).
* mongoose: MongoDB object modeling tool for Node.js.
* Student: Mongoose model representing the Student collection in MongoDB.
* ObjectId: From mongoose.Types to handle MongoDB ObjectIDs.
* process.env.MONGODB\_URI: MongoDB connection URI stored in environment variables.
* process.env.JWT\_SECREAT: JWT secret stored in environment variables for token verification.

CODE ->

const mongoose = require('mongoose');

const ObjectId = mongoose.Types.ObjectId;

const jwt = require('jsonwebtoken');

const Student = require('student.js'); // Adjust the path as necessary

// MongoDB connection URL and options

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

let isConnected;

async function connectToDatabase() {

if (!isConnected) {

await mongoose.connect(uri, {

useNewUrlParser: true,

useUnifiedTopology: true,

});

isConnected = mongoose.connection.readyState;

}

}

function verifyToken(token) {

const secret = process.env.JWT\_SECREAT; // Replace with your JWT secret

return jwt.verify(token, secret);

}

exports.handler = async (event) => {

// Handle CORS preflight request

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

return {

statusCode: 200,

headers: {

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

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

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

},

body: ''

};

}

// Set CORS headers

const headers = {

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

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

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

};

// Verify JWT token

const token = event.headers.Authorization;

if (!token) {

return {

statusCode: 401,

headers,

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

};

}

try {

verifyToken(token);

} catch (error) {

return {

statusCode: 401,

headers,

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

};

}

await connectToDatabase();

const { applicationNumber, allQuestions } = JSON.parse(event.body);

// Fetch the student record

const student = await Student.findOne({ applicationNumber });

if (!student) {

return {

statusCode: 404,

headers,

body: JSON.stringify({ message: 'Student not found' }),

};

}

const changes = {

MCQ: {

AttemptedQuestion: [],

NotAttemptedQuestion: [],

CorrectQuestion: []

},

descriptiveQuestions: {

AttemptedQuestion: [],

NotAttemptedQuestion: []

}

};

let isChanged = false;

// Separate MCQ and descriptive questions from the allQuestions payload

const mcqQuestions = allQuestions.filter(q => q.optionId);

const descriptiveQuestions = allQuestions.filter(q => q.answer);

// Process MCQ questions

const newMCQAttemptedQuestions = mcqQuestions.map(receivedQuestion => ({

\_id: receivedQuestion.questionId.toString(),

optionId: receivedQuestion.optionId.toString()

}));

if (newMCQAttemptedQuestions.length > 0 || student.MCQ.AttemptedQuestion.length > 0) {

student.MCQ.AttemptedQuestion = newMCQAttemptedQuestions;

changes.MCQ.AttemptedQuestion = newMCQAttemptedQuestions;

isChanged = true;

}

// Process descriptive questions

const newDescriptiveAttemptedQuestions = descriptiveQuestions.map(receivedQuestion => ({

\_id: receivedQuestion.questionId.toString(),

answer: receivedQuestion.answer

}));

if (newDescriptiveAttemptedQuestions.length > 0 || student.descriptiveQuestions.AttemptedQuestion.length > 0) {

student.descriptiveQuestions.AttemptedQuestion = newDescriptiveAttemptedQuestions;

changes.descriptiveQuestions.AttemptedQuestion = newDescriptiveAttemptedQuestions;

isChanged = true;

}

// Function to find not attempted questions

function findNotAttempted(generatedQuestions, attemptedQuestions) {

const attemptedIds = new Set(attemptedQuestions.map(aq => aq.\_id.toString()));

const notAttempted = generatedQuestions.filter(gq =>

!attemptedIds.has(gq.\_id.toString())

);

return notAttempted;

}

// Function to find correct MCQ questions

function findCorrectMCQQuestions(generatedQuestions, attemptedQuestions) {

const correctQuestions = [];

attemptedQuestions.forEach(attempted => {

const generatedQuestion = generatedQuestions.find(gq => gq.\_id.toString() === attempted.\_id);

if (generatedQuestion && generatedQuestion.correctAnswer === attempted.optionId) {

correctQuestions.push(attempted);

}

});

return correctQuestions;

}

// Update NotAttemptedQuestion with potentially missing fields

student.MCQ.NotAttemptedQuestion = findNotAttempted(

student.generatedShuffledQuestion.mcqQuizz,

student.MCQ.AttemptedQuestion

).map(q => ({ \_id: q.\_id.toString() })); // Save as object with \_id as String

student.descriptiveQuestions.NotAttemptedQuestion = findNotAttempted(

student.generatedShuffledQuestion.descriptiveQuizz,

student.descriptiveQuestions.AttemptedQuestion

).map(q => ({ \_id: q.\_id.toString() })); // Save as object with \_id as String

// Update CorrectQuestion for MCQ

student.MCQ.CorrectQuestion = findCorrectMCQQuestions(

student.generatedShuffledQuestion.mcqQuizz,

student.MCQ.AttemptedQuestion

).map(q => ({ \_id: q.\_id.toString(), optionId: q.optionId.toString() })); // Save as object with \_id and optionId as String

// Mark modified fields before saving

student.markModified('MCQ.AttemptedQuestion');

student.markModified('MCQ.NotAttemptedQuestion');

student.markModified('MCQ.CorrectQuestion');

student.markModified('descriptiveQuestions.AttemptedQuestion');

student.markModified('descriptiveQuestions.NotAttemptedQuestion');

// Save changes to the database

if (isChanged) {

await student.save();

// Prepare response

return {

statusCode: 200,

headers,

body: JSON.stringify({

message: 'Changes detected and updated',

changes,

notAttemptedMCQDetails: student.MCQ.NotAttemptedQuestion,

notAttemptedDescriptiveDetails: student.descriptiveQuestions.NotAttemptedQuestion,

correctMCQDetails: student.MCQ.CorrectQuestion

})

};

} else {

return {

statusCode: 200,

headers,

body: JSON.stringify({

message: 'No changes detected',

notAttemptedMCQDetails: student.MCQ.NotAttemptedQuestion,

notAttemptedDescriptiveDetails: student.descriptiveQuestions.NotAttemptedQuestion,

correctMCQDetails: student.MCQ.CorrectQuestion

})

};

}

};

MODEL ->

Student.js

const mongoose = require('mongoose');

const studentSchema = new mongoose.Schema({

fullName: {

type: String,

required: true,

trim: true

},

applicationNumber: {

type: String,

required: true,

unique: true,

trim: true

},

DOB: {

type: Date,

required: true

},

AllquestionId: {

type: String,

required: true,

trim: true

},

generatedShuffledQuestion: {

type: mongoose.Schema.Types.Mixed, // Using Mixed type to allow flexibility in the structure

required: false

},

MCQ: {

AttemptedQuestion: [{

type: mongoose.Schema.Types.Mixed,

required: false

}],

NotAttemptedQuestion: [{

type: mongoose.Schema.Types.Mixed,

required: false

}],

CorrectQuestion: [{

type: mongoose.Schema.Types.Mixed,

required: false

}]

},

descriptiveQuestions: {

AttemptedQuestion: [{

type: mongoose.Schema.Types.Mixed,

required: false

}],

NotAttemptedQuestion: [{

type: mongoose.Schema.Types.Mixed,

required: false

}],

}

},

{

timestamps: true

});

// Setting applicationNumber as the primary key

studentSchema.index({ applicationNumber: 1 }, { unique: true });

const Student = mongoose.model('StudentDetails', studentSchema);

module.exports = Student;