### LAMBDA FUNCTION -> **submitAllstudentquestion**

API URL -> <https://ybkfar4y6i.execute-api.us-east-1.amazonaws.com/S1/submitAllstudentquestion>

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

Stage -> S1

**Role of the API**:

* This API endpoint handles CRUD operations (create, read, update, delete) for student responses to MCQ and descriptive questions.
* It verifies JWT tokens for authorization, validates the payload structure, and interacts with MongoDB to update or delete student responses based on the action provided (update, delete).

**Functioning**:

* Validates the JWT token provided in the request headers for authorization.
* Connects to MongoDB using the URI provided in the MONGODB\_URI environment variable.
* Handles CORS pre-flight requests (OPTIONS method) to allow CORS headers.
* Parses the request body JSON to extract applicationNumber, questionId, optionId, answer, and action.
* Validates the structure of the payload to ensure required fields (applicationNumber, questionId) are present.
* Finds the student document in MongoDB using Student.findOne.
* Handles delete action by removing the specified question from AttemptedQuestion array within MCQ or descriptiveQuestions based on the type of question.
* Handles update or add action by updating the optionId for MCQs or answer for descriptive questions within AttemptedQuestion.
* Saves the updated Student document back to MongoDB and returns appropriate success or error responses.

**Request Body**:

* The API expects a JSON object in the request body with the following fields:
  + applicationNumber: Application number of the student.
  + questionId: ID of the question to update or delete.
  + optionId (optional): ID of the selected option for MCQs.
  + answer (optional): Answer provided for descriptive questions.
  + action: Action to perform (update, delete).

**Response**:

* **200 OK**: If the operation (update or delete) is successful, returns a success message.
* **400 Bad Request**: If the request payload is missing required fields or has an invalid structure.
* **401 Unauthorized**: If the JWT token is missing or invalid.
* **404 Not Found**: If the student or attempted question is not found in the database.
* **500 Internal Server Error**: If there's any server-side issue during database operations.

**Logic**:

* Handles CORS headers (Access-Control-Allow-Origin, Access-Control-Allow-Methods, Access-Control-Allow-Headers) for cross-origin requests.
* Verifies the JWT token for authorization using jsonwebtoken.
* Connects to MongoDB using Mongoose for database operations.
* Validates the incoming payload to ensure all required fields are present and in the correct format.
* Finds and updates or deletes the specified question in the Student document based on the action (update or delete).
* Saves the modified Student document back to MongoDB and returns appropriate status codes and messages based on the success or failure of these 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: Destructured from mongoose.Types to handle MongoDB ObjectIDs.

Code ->

const jwt = require('jsonwebtoken');

const mongoose = require('mongoose');

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

const { Types: { ObjectId } } = mongoose; // Destructure ObjectId from mongoose.Types

// MongoDB connection

mongoose.connect(process.env.MONGODB\_URI);

exports.handler = async (event) => {

// Consolidated CORS headers

const headers = {

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

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

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

};

// Handle CORS pre-flight request

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

return {

statusCode: 200,

headers,

body: JSON.stringify({ message: 'CORS preflight' })

};

}

const token = event.headers.Authorization || event.headers.authorization;

if (!token) {

return {

statusCode: 401,

headers,

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

};

}

let decoded;

try {

decoded = jwt.verify(token, process.env.JWT\_SECREAT);

} catch (error) {

return {

statusCode: 401,

headers,

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

};

}

const { applicationNumber, questionId, optionId, answer, action } = JSON.parse(event.body);

if (!applicationNumber || !questionId || (!optionId && !answer && action !== 'delete')) {

return {

statusCode: 400,

headers,

body: JSON.stringify({ message: 'Invalid payload structure' })

};

}

try {

// Find student by applicationNumber

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

if (!student) {

return {

statusCode: 404,

headers,

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

};

}

// Handle delete action

if (action === 'delete') {

// Check if it's an MCQ or descriptive question and remove it from AttemptedQuestion

const mcqIndex = student.MCQ.AttemptedQuestion.findIndex(q => q.\_id.toString() === questionId);

const descriptiveIndex = student.descriptiveQuestions.AttemptedQuestion.findIndex(q => q.\_id.toString() === questionId);

if (mcqIndex !== -1) {

student.MCQ.AttemptedQuestion.splice(mcqIndex, 1);

student.markModified('MCQ.AttemptedQuestion');

} else if (descriptiveIndex !== -1) {

student.descriptiveQuestions.AttemptedQuestion.splice(descriptiveIndex, 1);

student.markModified('descriptiveQuestions.AttemptedQuestion');

} else {

return {

statusCode: 404,

headers,

body: JSON.stringify({ message: 'Attempted question not found for delete' })

};

}

await student.save();

return {

statusCode: 200,

headers,

body: JSON.stringify({ message: 'Question deleted successfully' })

};

}

// Handle update or add action for MCQ or descriptive question

if (optionId !== undefined) {

// MCQ question

// Update or add optionId in AttemptedQuestion

const attemptedQuestionIndex = student.MCQ.AttemptedQuestion.findIndex(q => q.\_id.toString() === questionId);

if (attemptedQuestionIndex !== -1) {

student.MCQ.AttemptedQuestion[attemptedQuestionIndex].optionId = optionId;

} else {

student.MCQ.AttemptedQuestion.push({ \_id: questionId, optionId });

}

student.markModified('MCQ.AttemptedQuestion');

await student.save();

return {

statusCode: 200,

headers,

body: JSON.stringify({ message: 'MCQ Question updated successfully' })

};

} else if (answer !== undefined) {

// Descriptive question

// Update or add answer in AttemptedQuestion

const attemptedQuestionIndex = student.descriptiveQuestions.AttemptedQuestion.findIndex(q => q.\_id.toString() === questionId);

if (attemptedQuestionIndex !== -1) {

student.descriptiveQuestions.AttemptedQuestion[attemptedQuestionIndex].answer = answer;

} else {

student.descriptiveQuestions.AttemptedQuestion.push({ \_id: questionId, answer });

}

student.markModified('descriptiveQuestions.AttemptedQuestion');

await student.save();

return {

statusCode: 200,

headers,

body: JSON.stringify({ message: 'Descriptive Question updated successfully' })

};

} else {

return {

statusCode: 400,

headers,

body: JSON.stringify({ message: 'Invalid payload structure' })

};

}

} catch (error) {

console.error(error);

return {

statusCode: 500,

headers,

body: JSON.stringify({ message: 'Internal server error' })

};

}

};

**MODLE -> 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;