Name :- VALLEPU RAKESH

Phone Number :- 7207974348

Email:- rakeshvallepu0518@gmail.com

1. *Answer*

1. Set Up AWS Resources

a. Create a Public S3 Bucket

- Go to the **S3 service** in the AWS Management Console.
- Create a new S3 bucket, naming it appropriately (e.g., json-storage-bucket).
- Ensure the bucket is public so that your Lambda functions can access it.
- Configure bucket permissions and policies to allow PUT, GET, and LIST operations for your Lambda function (using an S3 bucket policy).

b. Set Up API Gateway

- Go to the **API Gateway** service in AWS.
- Create a **REST API**.
- Define two resources/endpoints:
  - /store (for the POST request to store JSON data).
  - /retrieve (for the GET request to retrieve all stored JSON data).
- Set up methods:
  - POST method for /store.
  - GET method for /retrieve.

---

2. Implement the POST Endpoint (Node.js Lambda Function)

Create a Lambda function named storeJsonData for handling the POST request to store JSON data.

a. Code for the storeJsonData Lambda Function

In AWS Lambda, choose **Node.js** as the runtime and write the following code:

```
const AWS = require('aws-sdk');
const s3 = new AWS.S3();
const BUCKET_NAME = 'json-storage-bucket';

exports.handler = async (event) => {
 try {
   const requestBody = JSON.parse(event.body);
```

```
    // Generate a unique filename for each JSON file
    const fileName = `json_data_${Date.now()}.json`;

    // Upload JSON data to the S3 bucket
    const s3Response = await s3.putObject({
      Bucket: BUCKET_NAME,
      Key: fileName,
      Body: JSON.stringify(requestBody),
      ContentType: 'application/json'
    }).promise();

    // Return e_tag and s3 link
    const response = {
      e_tag: s3Response.ETag,
      s3_link: `https://${BUCKET_NAME}.s3.amazonaws.com/${fileName}`
    };

    return {
      statusCode: 200,
      body: JSON.stringify(response),
    };
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ message: 'Error storing JSON data', error: error.message }),
    };
  }
};
```

b. Configure API Gateway to Use the POST Lambda Function

- In API Gateway, under the /store resource, link the POST method to the storeJsonData Lambda function.
- Set the **Lambda Proxy Integration** to handle the event structure from API Gateway.
- Deploy the API to a stage (e.g., dev).

---

3. Implement the GET Endpoint (Node.js Lambda Function)

Create another Lambda function named retrieveJsonData for handling the GET request to retrieve all stored JSON data.

a. Code for the retrieveJsonData Lambda Function

```
const AWS = require('aws-sdk');
const s3 = new AWS.S3();
const BUCKET_NAME = 'json-storage-bucket';

exports.handler = async (event) => {
  try {
```

```
    // List all objects in the S3 bucket
    const listResponse = await s3.listObjectsV2({
      Bucket: BUCKET_NAME,
    }).promise();

    const allJsonData = [];

    // Fetch each JSON file and compile contents
    for (const item of listResponse.Contents) {
      const fileData = await s3.getObject({
        Bucket: BUCKET_NAME,
        Key: item.Key
      }).promise();

      const jsonData = JSON.parse(fileData.Body.toString());
      allJsonData.push(jsonData);
    }

    return {
      statusCode: 200,
      body: JSON.stringify(allJsonData),
    };
  } catch (error) {
    return {
      statusCode: 500,
      body: JSON.stringify({ message: 'Error retrieving JSON data', error: error.message }),
    };
  }
};
```

b. Configure API Gateway to Use the GET Lambda Function

- In API Gateway, under the /retrieve resource, link the GET method to the retrieveJsonData Lambda function.
- Set the **Lambda Proxy Integration** for compatibility with API Gateway.
- Deploy the API to the same stage (e.g., dev).

---

4. Testing with Angular Frontend

In your Angular project, create a service to handle HTTP requests to the API Gateway endpoints.

a. Angular Service (DataService)

In data.service.ts, define the Angular service:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
```

```
@Injectable({
  providedIn: 'root'
})
export class DataService {
  private apiUrl = 'https://your-api-id.execute-api.region.amazonaws.com/dev';

  constructor(private http: HttpClient) {}

  // Method to store JSON data
  storeJsonData(data: any): Observable<any> {
    return this.http.post(`${this.apiUrl}/store`, data, {
      headers: new HttpHeaders({ 'Content-Type': 'application/json' })
    });
  }

  // Method to retrieve all JSON data
  retrieveJsonData(): Observable<any> {
    return this.http.get(`${this.apiUrl}/retrieve`);
  }
}
```

b. Usage in Angular Component

In app.component.ts, use the service to store and retrieve JSON data.

```
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-root',
  template: `
    <div>
      <h2>Store JSON Data</h2>
      <button (click)="storeData()">Store Sample JSON</button>

      <h2>Retrieve JSON Data</h2>
      <button (click)="retrieveData()">Get All JSON</button>
      <pre>{{ jsonData | json }}</pre>
    </div>
  `
})
export class AppComponent {
  jsonData: any;

  constructor(private dataService: DataService) {}

  storeData() {
```

```
    const sampleData = { name: "Test User", age: 30 };
    this.dataService.storeJsonData(sampleData).subscribe(response => {
      console.log('Stored data response:', response);
    });
  }

  retrieveData() {
    this.dataService.retrieveJsonData().subscribe(data => {
      this.jsonData = data;
      console.log('Retrieved data:', data);
    });
  }
}
```

5. Testing

- **POST Endpoint**: Click the "Store Sample JSON" button in the Angular app to store JSON data. Verify the response for the e_tag and s3_link.
- **GET Endpoint**: Click the "Get All JSON" button to retrieve all stored JSON data from S3 and display it in the Angular app.

Summary

This setup allows you to:

1. **Store** JSON data using a POST request to an S3 bucket.
2. **Retrieve** all stored JSON data with a GET request.
3. **Interact** with these endpoints through an Angular frontend.

2nd Answer

1. Implement the Scheduler Class in Node.js

Create a file scheduler.js to define the Scheduler class:

```
class Scheduler {
  constructor() {
    // Stores events as an array of objects with start_time and end_time properties
    this.events = [];
  }
```

```
  // Method to add an event after checking for overlaps
  addEvent({ start_time, end_time }) {
    // Check if start_time and end_time are valid
    if (start_time < 0 || end_time > 23 || start_time >= end_time) {
      throw new Error('Invalid time range');
    }

    // Check for overlapping events
    for (const event of this.events) {
      if (
        (start_time < event.end_time && end_time > event.start_time) // Overlap condition
      ) {
        return false; // Overlap found
      }
    }

    // No overlaps, so add the event
    this.events.push({ start_time, end_time });
    return true;
  }

  // Method to get all scheduled events
  getEvents() {
    return this.events;
  }
}

// Export Scheduler class
module.exports = Scheduler;
```

2. **Create an Express Server to Use the Scheduler Class**

```
const express = require('express');
const Scheduler = require('./scheduler'); // Import the Scheduler class
const app = express();
const scheduler = new Scheduler(); // Create an instance of Scheduler

app.use(express.json());

// Endpoint to add an event
app.post('/addEvent', (req, res) => {
  const { start_time, end_time } = req.body;

  try {
    const success = scheduler.addEvent({ start_time, end_time });
    if (success) {
      res.json({ success: true, message: 'Event added successfully' });
    } else {
      res.json({ success: false, message: 'Event overlaps with an existing event' });
    }
  } catch (error) {
    res.status(400).json({ success: false, message: error.message });
  }
});

// Endpoint to retrieve all events
app.get('/getEvents', (req, res) => {
  res.json({ events: scheduler.getEvents() });
});

// Start the server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

3. Angular Frontend to Interact with the Scheduler

In your Angular project, create a service to interact with the Node.js backend.

a. Angular Service (scheduler.service.ts)

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
```

```
export class SchedulerService {
  private apiUrl = 'http://localhost:3000';

  constructor(private http: HttpClient) {}

  // Method to add an event
  addEvent(start_time: number, end_time: number): Observable<any> {
    return this.http.post(`${this.apiUrl}/addEvent`, { start_time, end_time });
  }

  // Method to retrieve all events
  getEvents(): Observable<any> {
    return this.http.get(`${this.apiUrl}/getEvents`);
  }
}
```

b. Angular Component (app.component.ts)

In app.component.ts, create the logic for handling form submission and displaying events.

```
import { Component, OnInit } from '@angular/core';
import { SchedulerService } from './scheduler.service';

@Component({
  selector: 'app-root',
  template: `
    <div>
      <h2>Daily Event Scheduler</h2>

      <!-- Form to add new events -->
      <form (submit)="scheduleEvent()">
        <label for="start">Start Time (0-23):</label>
        <input type="number" [(ngModel)]="start_time" name="start" min="0" max="23"
required>

        <label for="end">End Time (0-23):</label>
        <input type="number" [(ngModel)]="end_time" name="end" min="0" max="23"
required>

        <button type="submit">Add Event</button>
      </form>

      <!-- Display feedback message -->
      <p *ngIf="message">{{ message }}</p>

      <!-- Display list of events -->
```

```
      <h3>Scheduled Events:</h3>
      <ul>
        <li *ngFor="let event of events">
          {{ event.start_time }} - {{ event.end_time }}
        </li>
      </ul>
    </div>
  `
})
export class AppComponent implements OnInit {
  start_time!: number;
  end_time!: number;
  events: { start_time: number, end_time: number }[] = [];
  message: string = '';

  constructor(private schedulerService: SchedulerService) {}

  ngOnInit() {
    this.loadEvents();
  }

  // Load events from the server
  loadEvents() {
    this.schedulerService.getEvents().subscribe((data) => {
      this.events = data.events;
    });
  }

  // Schedule a new event
  scheduleEvent() {
    this.schedulerService.addEvent(this.start_time, this.end_time).subscribe((response) => {
      this.message = response.message;
      if (response.success) {
        this.loadEvents(); // Reload events after adding a new one
      }
    });
  }
}
```

This component:

- Provides a form to input start_time and end_time for scheduling events.
- Displays a message after attempting to schedule an event, showing whether it was successful or if it overlapped.
- Lists all scheduled events on the page.

c. Angular Module Setup

Ensure you have imported the HttpClientModule and FormsModule in app.module.ts.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';
import { SchedulerService } from './scheduler.service';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    FormsModule
  ],
  providers: [SchedulerService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

4. Testing the Application

1. **Start the Node.js Server**:
    - Run the server by executing node server.js.
2. **Run the Angular Application**:
    - Start the Angular app using ng serve.
3. **Access the UI**:
    - Open a browser and go to http://localhost:4200.

4. **Test Event Scheduling**:
    - Add various events and observe if they appear in the scheduled list.
    - Try adding overlapping events to verify that the app prevents overlaps and displays an appropriate message.

Summary

This setup provides:

1. **Node.js Backend**: A scheduler class with methods to add and retrieve events, and an Express server to expose these functionalities.
2. **Angular Frontend**: A simple UI for scheduling events, with input validation and feedback on overlaps.
3. **Validation and Overlap Prevention**: Ensures only non-overlapping events are scheduled.

3nd Answer

1. Set Up the Node.js Server with Colyseus

a. Install Dependencies

Start by setting up a Node.js project with Colyseus:

mkdir multiplayer-shape-extrusion

cd multiplayer-shape-extrusion

npm init -y

npm install express colyseus @colyseus/schema

b. Create the Colyseus Room

In Colyseus, rooms are instances where clients can connect and synchronize their state. Create a ShapeRoom.js file:

```
// ShapeRoom.js

const { Room } = require("colyseus");

const { Schema, type, ArraySchema } = require("@colyseus/schema");
```

```
class Shape extends Schema {
  @type("string") id;
  @type("number") x = 0;
  @type("number") y = 0;
  @type("number") z = 0;
  @type("number") rotation = 0;
  @type("string") color = "#FFFFFF";

  constructor(id, color) {
    super();
    this.id = id;
    this.color = color;
  }
}

class ShapeRoomState extends Schema {
  @type([Shape]) shapes = new ArraySchema();
}

class ShapeRoom extends Room {
  onCreate(options) {
    this.setState(new ShapeRoomState());

    this.onMessage("create_shape", (client, data) => {
      const shape = new Shape(client.sessionId, data.color);
      this.state.shapes.push(shape);
      this.broadcast("shape_created", shape);
    });

    this.onMessage("move_shape", (client, data) => {
```

```
    const shape = this.state.shapes.find((s) => s.id === client.sessionId);
    if (shape) {
      shape.x = data.x;
      shape.y = data.y;
      shape.z = data.z;
      shape.rotation = data.rotation;
    }
    this.broadcast("shape_moved", shape);
  });
 }

 onJoin(client) {
  console.log(`${client.sessionId} joined`);
 }

 onLeave(client) {
  console.log(`${client.sessionId} left`);
  this.state.shapes = this.state.shapes.filter(shape => shape.id !== client.sessionId);
  this.broadcast("shape_removed", { id: client.sessionId });
 }
}

module.exports = ShapeRoom;
```

This code defines:

- **ShapeRoom**: Handles creating shapes and moving shapes for each player.
- **onMessage**: Listens for "create_shape" and "move_shape" messages, updating the shared state and broadcasting to all clients.
- **ShapeRoomState**: Holds an array of Shape instances for all players.

c. Set Up the Server

Create server.js to configure the Colyseus server with an Express instance:

```
const express = require("express");
const http = require("http");
const { Server } = require("colyseus");
const ShapeRoom = require("./ShapeRoom");

const app = express();
const port = 3000;
const server = http.createServer(app);
const gameServer = new Server({ server });

gameServer.define("shape_room", ShapeRoom);

server.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

2. Angular Frontend with BabylonJS

a. Install Angular and BabylonJS

Set up a new Angular project:

```
ng new multiplayer-game
cd multiplayer-game
npm install babylonjs colyseus.js
```

b. Set Up Colyseus Client

Create a service in Angular to interact with the Colyseus server.

src/app/services/colyseus.service.ts

```
import { Injectable } from '@angular/core';
import { Client, Room } from "colyseus.js";
```

```
@Injectable({
  providedIn: 'root'
})
export class ColyseusService {
  private client: Client;
  public room: Room;

  constructor() {
    this.client = new Client("ws://localhost:3000");
  }

  async joinRoom() {
    this.room = await this.client.joinOrCreate("shape_room");
  }

  sendCreateShape(color: string) {
    this.room.send("create_shape", { color });
  }

  sendMoveShape(x: number, y: number, z: number, rotation: number) {
    this.room.send("move_shape", { x, y, z, rotation });
  }
}
```

c. Create the BabylonJS Component

Use BabylonJS to render the 3D shapes and handle user input.

src/app/components/game/game.component.ts

```
import { Component, ElementRef, NgZone, OnInit, AfterViewInit } from '@angular/core';
import * as BABYLON from 'babylonjs';
import { ColyseusService } from '../../services/colyseus.service';
```

```
@Component({
  selector: 'app-game',
  template: '<canvas #gameCanvas></canvas>',
  styles: ['canvas { width: 100%; height: 100% }']
})
export class GameComponent implements OnInit, AfterViewInit {
  private engine: BABYLON.Engine;
  private scene: BABYLON.Scene;
  private shapes = new Map<string, BABYLON.Mesh>();

  constructor(private el: ElementRef, private colyseusService: ColyseusService, private ngZone:
NgZone) {}

  ngOnInit(): void {
    this.colyseusService.joinRoom().then(() => {
      this.colyseusService.room.onMessage("shape_created", (shape) => this.createShape(shape));
      this.colyseusService.room.onMessage("shape_moved", (shape) => this.moveShape(shape));
    });
  }

  ngAfterViewInit(): void {
    const canvas = this.el.nativeElement.querySelector("canvas");
    this.engine = new BABYLON.Engine(canvas, true);
    this.scene = new BABYLON.Scene(this.engine);

    const camera = new BABYLON.ArcRotateCamera("camera", Math.PI / 2, Math.PI / 2, 10, new
BABYLON.Vector3(0, 0, 0), this.scene);
    camera.attachControl(canvas, true);

    const light = new BABYLON.HemisphericLight("light", new BABYLON.Vector3(1, 1, 0), this.scene);
```

```
    this.engine.runRenderLoop(() => {
      this.scene.render();
    });
  }

  private createShape(shapeData: any) {
    const shape = BABYLON.MeshBuilder.CreateBox(shapeData.id, { height: 1, depth: 1, width: 1 },
this.scene);
    shape.position.x = shapeData.x;
    shape.position.y = shapeData.y;
    shape.position.z = shapeData.z;
    this.shapes.set(shapeData.id, shape);
  }

  private moveShape(shapeData: any) {
    const shape = this.shapes.get(shapeData.id);
    if (shape) {
      shape.position.x = shapeData.x;
      shape.position.y = shapeData.y;
      shape.position.z = shapeData.z;
      shape.rotation.y = shapeData.rotation;
    }
  }
}
```

This component:

1. **Initializes the BabylonJS Scene**.
2. **Joins the Colyseus room** and listens for shape creation and movement messages.
3. **Creates shapes** when receiving shape_created messages and **updates positions** when ss

d. UI to Create and Move Shapes

Add basic controls for creating and moving shapes in the component's HTML.

src/app/components/game/game.component.html

```
<div>
  <button (click)="createShape()">Create Shape</button>
</div>
<canvas #gameCanvas></canvas>
```

Add a method in game.component.ts to handle shape creation:

```
createShape() {
  const color = "#"+((1<<24)*Math.random()|0).toString(16); // Random color
  this.colyseusService.sendCreateShape(color);
}
```

Summary

This setup provides:

- **Backend**: A Node.js server with Colyseus for managing multiplayer state, allowing users to create and move shapes in real time.
- **Frontend**: An Angular application with BabylonJS for rendering 3D shapes, where users can interact with shapes and see updates from other players in real-time.

With this setup, users can join the game, create shapes, and move them, seeing each other's actions in real time.