DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

Ramaiah Institute of Technology

College Timetable Management Backend

using Node.js, Express.js and AWS DynamoDB

Project Report

Submitted in partial fulfillment of the requirements for the award of the course on

NOSQL DATABASES

in

5th Semester , ISE

Submitted by:

Gagan R — 1MS23IS040
Bhargav B K — 1MS23IS028
Chirag B K — 1MS23IS035
SriHari — 1MS23IS039

Under the guidance of:

Savitha K Shetty
Associate Professor, Department of ISE

2024–2025

Abstract

Traditional timetable management in colleges is often handled using spreadsheets or manual methods, making it error-prone, difficult to update, and hard to integrate with other systems. This project presents a **RESTful backend system** for **College Timetable Management**, implemented using **Node.js**, **Express.js**, and **AWS DynamoDB**. The system is designed to be consumed entirely via **Postman** or any HTTP client; no frontend is required.

The backend supports **two user roles**: Admin and Student. Admin registration and login are performed using **email + password**, while students use **USN + password**. **bcrypt** is used to hash passwords, and **JWT (JSON Web Token)** is used for stateless authentication. This implementation supports **dual-authentication**: secure HTTP-only cookies (preferred) and a Bearer-token fallback via the `Authorization` header. Role-based access control is enforced using dedicated middlewares: `adminAuthMiddleware` and `studentAuthMiddleware`.

The data layer uses **AWS DynamoDB** with two main tables: `Users` and `TimeTable`. The `Users` table stores both admin and student accounts with a composite primary key of the form `userType#identifier`. The `TimeTable` table stores timetable slots using a `yearSection` as the partition key and a composite sort key `day#slot` to model 7 daily slots per section.

The backend exposes a clear set of APIs for:

- Admin registration and login.
- Student registration and login.
- Admin-protected timetable CRUD: adding single slots, batch inserting a day's slots, updating a slot, and deleting a slot.
- Student-protected timetable fetch APIs for weekly and per-day views.

All APIs return meaningful JSON responses and consistent error messages. Environment variables are used for secrets and configuration. The system is thoroughly tested using Postman, and sample test instructions are documented in the Appendix.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Timetable management is a routine but critical operation in any college. Departments must allocate subjects, faculty, rooms, and lab sessions across multiple sections and semesters. Traditional manual or spreadsheet-based approaches are error-prone, hard to update, and do not scale. A programmatic backend service simplifies updates, enables access control, allows conflict detection, and makes integration with clients straightforward.

1.2 Problem Statement

Design and implement a secure, scalable, and role-based **backend system** for **College Timetable Management** using Node.js, Express.js, and AWS DynamoDB with the following requirements:

- Admins must authenticate using **email + password**.
- Students must authenticate using **USN + password**.
- Passwords must be stored securely using hashing (bcrypt).
- Stateless authentication using JWT with role-based middlewares.
- Timetable operations for admins and read operations for students.
- Dual-auth: HTTP-only cookie (preferred) + Bearer token fallback.

1.3 Objectives

1. Implement a **RESTful backend** using Node.js and Express.js.

2. Design DynamoDB tables for **users** and **timetables** with efficient access patterns.
3. Implement secure authentication using **bcrypt** and **JWT**.
4. Implement **role-based authorization** via middlewares.
5. Expose clear APIs for timetable CRUD and fetch operations.
6. Validate duplicate timetable slots and return meaningful error messages.
7. Provide Postman-based testing instructions and sample requests.

1.4 Scope

- Backend-only implementation (no frontend UI).
- Testing and demonstration using Postman.
- Single college / single-department timetable management, easily extensible.
- 7 fixed slots per day with pre-defined time ranges.

1.5 Technology Stack

- **Runtime:** Node.js (v14+)
- **Web Framework:** Express.js
- **Database:** AWS DynamoDB (NoSQL)
- **AWS SDK:** `@aws-sdk/client-dynamodb`, `@aws-sdk/lib-dynamodb`
- **Authentication:** JWT (`jsonwebtoken`), bcrypt (`bcrypt`)
- **Environment config:** `dotenv`
- **API Testing:** Postman

1.6 High-level Architecture

The system follows a layered architecture:

- **Routes Layer**: Defines endpoints and maps to controllers.
- **Controller Layer**: Input validation response formatting.

- **Service Layer**: Business logic DynamoDB interactions.
- **Database Layer**: DynamoDB DocumentClient (AWS SDK v3).
- **Middleware**: Authentication authorization logic.

Figure 1.1: High-level architecture (Routes → Controllers → Services → DynamoDB)

1.7 User Roles and Use Cases

1.7.1 Admin

- Register with email + password.
- Login to receive a JWT (cookie fallback).
- Manage timetable: add single slot, add batch slots for a day, update slot, delete slot.

1.7.2 Student

- Register with USN + password.
- Login to receive a JWT (cookie fallback).
- Fetch weekly timetable for their year-section.
- Fetch timetable for a specific day.

1.8 API Overview

High-level endpoints:

- **Admin Auth**: POST /admin/register, POST /admin/login, POST /admin/logout
- **Student Auth**: POST /student/register, POST /student/login
- **Admin Timetable CRUD**: POST /admin/timetable/addSingleSlot, POST /admin/timetable/addBatchForDay, PUT /admin/timetable/updateSlot, DELETE /admin/timetable/deleteSlot
- **Student Timetable Fetch**: GET /student/timetable/weekly/:yearSection, GET /student/timetable/day/:yearSection/:day

Chapter 2

Database Design (DynamoDB)

2.1 Why DynamoDB?

AWS DynamoDB is chosen for:

- Low-latency single-digit millisecond performance.
- Fully managed, scales automatically.
- Simple key-value / document model matches our access patterns.

2.2 Tables

We use two tables: `Users` and `TimeTable`.

2.2.1 Users Table

Table Name: `Users`

Table 2.1: Users Table Schema

Attribute	**Description**
`PK`	Partition key: `userType#identifier` (e.g., ADMIN#admin@x.com)
`userType`	"ADMIN" or "STUDENT"
`email`	Admin email (only for ADMIN)
`usn`	Student USN (only for STUDENT)
`name`	Full name
`hashed_password`	bcrypt-hashed password
`createdAt`	ISO timestamp

Primary Key examples:

- Admin: `PK = "ADMIN#admin@college.edu"`

- Student: `PK = "STUDENT#1MS23IS040"`

2.2.2 TimeTable Table

Table Name: `TimeTable`

Table 2.2: TimeTable Table Schema

Attribute	**Description**
`PK`	Partition key: `yearSection` (e.g., ”5A”)
`SK`	Sort key: `day#slot` (e.g., ”MONDAY#1”)
`subject`	Subject name/code
`faculty`	Faculty identifier/name
`room`	Room identifier
`type`	”Theory” or ”LAB”
`createdAt`	ISO timestamp

2.2.3 Slot Timing Reference

There are exactly 7 slots per day:

Table 2.3: Daily Slot Timings

Slot	**Time**
1	9:00 – 9:55
2	9:55 – 10:50
3	11:05 – 12:00
4	12:00 – 12:55
5	1:45 – 2:40
6	2:40 – 3:35
7	3:35 – 4:30

2.3 Access Patterns and Queries

- **Get single user (Admin/Student):** GetItem with `PK`.
- **Add/update/delete single slot:** PutItem / UpdateItem / DeleteItem with `PK` and `SK`.
- **Get day timetable for section:** Query with `PK = yearSection` and filter or KeyCondition for `begins_with(SK, "MONDAY")`.
- **Get weekly timetable:** Query with `PK = yearSection` and return all items, then group by day sort by slot ascending.

2.4 Duplicate Slot Prevention

Before inserting a slot:

- Use GetItem on the composite key (`PK, SK`).
- If exists, return 409 Conflict with friendly message.
- For batch insert: validate duplicates inside the payload and then check existing items via batch/transaction operations. If any slot exists, abort and return list of conflicting slots.

Chapter 3

Implementation Details

3.1 Project Structure

```
src/
 config/
    dynamoClient.js       # DynamoDB client configuration
 controllers/
    adminAuthController.js
    studentAuthController.js
    timetableController.js
 middleware/
    adminAuth.js
    studentAuth.js
 services/
    userService.js
    timetableService.js
 routes/
    adminRoutes.js
    studentRoutes.js
 app.js                   # Express app configuration
 server.js                # Server entry point
```

3.2 Environment Variables (`.env`)

- `PORT=3000`
- `NODE`$_{E}NV = development$
- `JWT`$_{S}ECRET = your_{s}uper_{s}ecret_{j}wt_{k}ey$

- AWS$_{R}EGION = ap - south - 1$
- AWS$_{A}CCESS_{K}EY_{I}D = your_{a}ccess_{k}ey$
- AWS$_{S}ECRET_{A}CCESS_{K}EY = your_{s}ecret_{k}ey$
- USERS$_{T}ABLE_{N}AME = Users$
- TIMETABLE$_{T}ABLE_{N}AME = TimeTable$

3.3 DynamoDB Client (Representative snippet)

```
// src/config/dynamoClient.js
const { DynamoDBClient } = require("@aws-sdk/client-dynamodb");
const { DynamoDBDocumentClient } = require("@aws-sdk/lib-dynamodb");

const client = new DynamoDBClient({
  region: process.env.AWS_REGION
});

const docClient = DynamoDBDocumentClient.from(client);

module.exports = { docClient };
```

3.4 User Registration — Representative Logic

- Validate required fields (email or usn, name, password).
- Hash password using bcrypt.
- Build PK: `ADMIN#email` or `STUDENT#usn`.
- Use PutItem with ConditionExpression to avoid duplicates (conditional write).

```
// Representative: create user
const bcrypt = require("bcrypt");
const { PutCommand } = require("@aws-sdk/lib-dynamodb");

async function registerAdmin({ name, email, password }) {
  const salt = await bcrypt.genSalt(10);
  const hashed = await bcrypt.hash(password, salt);
  const pk = `ADMIN#${email.toLowerCase()}`;
  await docClient.send(new PutCommand({
    TableName: process.env.USERS_TABLE_NAME,
    Item: {
      PK: pk,
      userType: "ADMIN",
      email,
      name,
      hashed_password: hashed,
      createdAt: new Date().toISOString()
    },
    ConditionExpression: "attribute_not_exists(PK)"
  }));
  return { message: "Admin registered" };
}
```

3.5 Login, JWT Generation, and Cookie Setup (Representative snippet)

We support dual-auth:

- On login, server signs JWT and sets it as an HTTP-only cookie (adminToken / studentToken) with SameSite and Secure flags.
- The token is also returned in response body so Postman or clients can store it as a fallback.

```
// Representative: login and issue cookie + token
const jwt = require("jsonwebtoken");

function issueToken(res, payload, cookieName) {
  const token = jwt.sign(payload, process.env.JWT_SECRET, {
     expiresIn: "24h" });
  // Set HTTP-only cookie
  res.cookie(cookieName, token, {
    httpOnly: true,
    secure: process.env.NODE_ENV === "production",
    sameSite: "strict",
    maxAge: 24 * 60 * 60 * 1000
  });
  // Also return token in JSON for fallback
  return token;
}
```

3.6 Auth Middlewares (Representative snippet)

- `adminAuthMiddleware` checks cookie `adminToken` first, then `Authorization` header fallback.
- Verifies JWT, ensures `userType` matches.

```
// src/middleware/adminAuth.js (representative)
const jwt = require("jsonwebtoken");

function extractToken(req) {
  const authHeader = req.headers["authorization"];
  if (authHeader && authHeader.startsWith("Bearer ")) return
     authHeader.split(" ")[1];
  if (req.cookies && req.cookies.adminToken) return req.cookies.
     adminToken;
  return null;
}

async function adminAuthMiddleware(req, res, next) {
  try {
    const token = extractToken(req);
    if (!token) return res.status(401).json({ success:false,
       message:"Token missing" });
    const payload = jwt.verify(token, process.env.JWT_SECRET);
    if (payload.userType !== "ADMIN") return res.status(403).json
       ({ success:false, message:"Forbidden - Admins only" });
    req.user = payload;
    next();
  } catch (err) {
    return res.status(401).json({ success:false, message:"Invalid
        or expired token" });
  }
}
```

3.7 Timetable CRUD — Representative Logic

3.7.1 Add Single Slot

- Build `PK = year_section.toUpperCase()`.
- Build `SK = ‘`*day*`slot‘` where day is uppercase (MONDAY).
- Check existing item with GetItem. If exists → 409 Conflict.
- Else put item.

3.7.2 Batch Add (Day)

- Validate there are no duplicate slot numbers in the payload.
- Check existing items for collisions (either with BatchGet or multiple Gets).
- Use TransactWrite (optional) or BatchWrite to insert; on conflict, return clear error with conflicting slots.

3.7.3 Update Slot

- Use UpdateCommand with Key PK, SK .
- If item not found, return 404.

3.7.4 Delete Slot

- Use DeleteCommand with Key PK, SK .
- Return success message if deletion acknowledged.

3.8 Student Timetable Fetch

- **GET /student/timetable/weekly/:yearSection**: Query by PK, parse SK into day and slot, group by day, sort ascending by slot.
- **GET /student/timetable/day/:yearSection/:day**: Query by PK with begins$_{w}ithSKfilterf$

3.9 Representative JSON Responses

- Success:

```
{
  "success": true,
  "message": "Slot added",
  "data": { ... }
}
```

- Error:

```
{
  "success": false,
  "message": "Slot already exists",
  "errorCode": "DUPLICATE_SLOT"
}
```

3.10 Security Considerations

- Passwords stored as bcrypt hashes (salt rounds = 10).
- JWT expiry 24 hours (tunable).
- Cookies: `HttpOnly, SameSite=strict, Secure` in production.
- Role-based middlewares ensure separation of privileges.
- Input validation on all endpoints (e.g., slot number 1-7, day uppercase, year section sanitized).

Chapter 4

Testing with Postman and README (Merged)

4.1 Prerequisites

- Node.js (v14+)
- AWS account with DynamoDB access
- AWS credentials (Access Key ID Secret)
- `.env` configured
- Tables created: `Users`, `TimeTable`

4.2 Installation (README steps)

```
git clone <repo-url>
cd project
npm install
cp .env.example .env
# fill .env with AWS creds and JWT secret
npm start
```

4.3 DynamoDB Setup (Quick summary)

1. Create IAM user with DynamoDB permissions.
2. Create two tables:
    - Users: Partition key PK (String)

- TimeTable: Partition key PK (String), Sort key SK (String)

3. Alternatively, use local DynamoDB for offline testing.

4.4 Postman Environment Setup

Create a Postman Environment (e.g., `Timetable Management`) with these variables:

- `base_url` = http://localhost:3000
- `admin_token` = (empty)
- `student_token` = (empty)

Enable **Send cookies** in Postman Settings so cookie-based auth works automatically.

4.5 Postman Testing Workflow

4.5.1 Admin Flow

1. **Register Admin**
   - POST base_url/admin/register
   - Body:

```
{
  "email": "admin@college.edu",
  "password": "admin123",
  "name": "Admin Name"
}
```

2. **Login Admin**
   - POST base_url/admin/login
   - Body:

```
{
  "email": "admin@college.edu",
  "password": "admin123"
}
```

   - The server sets an HTTP-only cookie `adminToken` automatically (ensure Postman is set to Send Cookies). Response also includes `token` in JSON as fallback.
   - (Optional) Put a Test script in Postman to extract token to environment:

```
if (pm.response.code === 200) {
  var jsonData = pm.response.json();
  pm.environment.set("admin_token", jsonData.token || "");
}
```

3. **Use Admin Endpoints** — for each request: either rely on cookie, or include header:

```
Authorization: Bearer {{admin_token}}
```

4.5.2 Student Flow

1. **Register Student**
   - POST base_url/student/register
   - Body:

   ```
   {
     "usn": "1MS23IS040",
     "password": "student123",
     "name": "Gagan R"
   }
   ```

2. **Login Student**
   - POST base_url/student/login
   - Body:

   ```
   {
     "usn": "1MS23IS040",
     "password": "student123"
   }
   ```

   - Server sets HTTP-only cookie `studentToken` and returns token in JSON for fallback.
   - (Optional) Save token to environment:

   ```
   if (pm.response.code === 200) {
     var jsonData = pm.response.json();
     pm.environment.set("student_token", jsonData.token || "");
   }
   ```

3. **Use Student Endpoints** — rely on cookie or send:

```
        Authorization: Bearer {{student_token}}
```

4.6 Representative Postman Requests

4.6.1 Add Single Slot (Admin)

```
POST {{base_url}}/admin/timetable/addSingleSlot
Headers:
 Authorization: Bearer {{admin_token}}   # optional if cookie used
Body (JSON):
{
  "year_section": "5A",
  "day": "MONDAY",
  "slot": 2,
  "subject": "DBMS",
  "faculty": "RSH",
  "room": "LHC-315",
  "type": "Theory"
}
```

4.6.2 Add Batch Slots (Admin)

```
POST {{base_url}}/admin/timetable/addBatchForDay
Headers: Authorization: Bearer {{admin_token}}
Body:
{
  "year_section": "5A",
  "day": "MONDAY",
  "slots": [
    { "slot": 1, "subject": "OS", "faculty": "JDS",
    "room": "LHC-315", "type": "Theory"},
    { "slot": 2, "subject": "DBMS", "faculty": "RSH",
    "room": "LHC-315", "type": "Theory"}
  ]
}
```

4.6.3 Weekly Timetable (Student)

```
GET {{base_url}}/student/timetable/weekly/5A
```

```
Headers: Authorization: Bearer {{student_token}}
```

4.6.4 Day Timetable (Student)

```
GET {{base_url}}/student/timetable/day/5A/MONDAY
Headers: Authorization: Bearer {{student_token}}
```

4.7 Test Cases and Expected Responses

- **Duplicate user registration** → 409 Conflict with message "User already exists".
- **Wrong login credential** → 401 Unauthorized.
- **Adding duplicate slot** → 409 Conflict with "Slot already exists" and slot details.
- **Accessing admin route with student token** → 403 Forbidden.
- **Accessing protected route without token** → 401 Unauthorized.

4.8 Troubleshooting

- DynamoDB connection issues: verify AWS credentials, region, and table names.
- Token issues: check `JWT_SECRET` and token expiry.
- Cookie not sent in Postman: enable "Send cookies" in settings.
- Ensure day names are uppercase and slot numbers are 1-7.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The backend achieves:

- Secure role-based authentication with bcrypt and JWT.
- Dual-auth support (HTTP-only cookies + Bearer fallback) for flexibility.
- CRUD operations for timetable with duplicate detection.
- Student-focused read APIs for weekly and daily timetables.
- Clean layered architecture suitable for extension.

5.2 Future Enhancements

- Conflict detection across rooms faculties (global constraint checks).
- Notifications or email alerts for timetable changes.
- Multi-tenant support for multiple departments and years.
- Frontend integration with role-based UI.
- Audit logs for modifications and admin activity tracking.

Appendix A

API Reference (Detailed)

A.1 Authentication APIs

Endpoint	Method	Description
`/admin/register`	POST	Admin registration using email+password.
`/admin/login`	POST	Admin login; sets HTTP-only cookie ”adminToken”; returns token in JSON.
`/admin/logout`	POST	Clears admin cookie.
`/student/register`	POST	Student registration using USN+password.
`/student/login`	POST	Student login; sets HTTP-only cookie ”studentToken”; returns token in JSON.

A.2 Admin Timetable APIs

Endpoint	Method	Description
`/admin/timetable /addSingleSlot`	POST	Add a single slot.
`/admin/timetable /addBatchForDay`	POST	Add multiple slots for a given day.
`/admin/timetable /updateSlot`	PUT	Update slot identified by year_section, day, slot.
`/admin/timetable /deleteSlot`	DELETE	Delete a slot using query params (year_section, day, slot).

A.3 Student Timetable APIs

Endpoint	**Method**	**Description**
`/student/timetable/weekly/:yearSection`	GET	Return timetable grouped by day.
`/student/timetable/day/:yearSection/:day`	GET	Return slots sorted by slot number.

Appendix B

Representative Error Codes

- DUPLICATE$_{U}SER|userregistrationconflict.$

- INVALID$_{C}REDENTIALS|wronglogindetails.$

- DUPLICATE$_{S}LOT|slotalreadyexists.$

- UNAUTHORIZED | missing or invalid token.
- FORBIDDEN | role mismatch.