RESTful API in Node.js

1 What is REST Architecture?

REST (**Representational State Transfer**) is an architectural style used to design networked applications. It follows the **client-server model** and is based on HTTP methods.

◇ Key Principles of REST:

1. **Stateless** – Each request from a client must contain all the necessary information; no session is stored on the server.
2. **Client-Server** – The client and server are independent of each other.
3. **Cacheable** – Responses should be cacheable to optimize performance.
4. **Uniform Interface** – A consistent and well-defined API structure (e.g., URLs, methods).
5. **Layered System** – The client does not need to know about intermediary layers.

2 HTTP Methods in RESTful API

Method	Description	Example Endpoint
`GET`	Retrieve resources	`/users` (Get all users)
`POST`	Create a new resource	`/users` (Add a new user)
`PUT`	Update an existing resource	`/users/:id` (Update user details)
`PATCH`	Partially update a resource	`/users/:id` (Update a specific field)
`DELETE`	Remove a resource	`/users/:id` (Delete a user)

3 RESTful Web Services

A **RESTful web service** is an API that follows REST principles. It allows clients to interact with the server using HTTP methods.

◇ Key Features of RESTful APIs

- Uses **JSON** or **XML** for data exchange.
- Uses **URLs** to access resources.
- Stateless architecture.

What is Layered Architecture?

A **Layered Architecture** separates the logic into different layers:

- **Routes**: Handles API requests and responses.
- **Controller** : It receives HTTP requests, processes input data, and calls the appropriate service methods.
- **Services (Business Logic Layer)**: Processes the request and contains business logic.
- **Repositories (Data Access Layer)**: Interacts with the database.
- **Models (Entity Layer)**: Defines data structures.

Creating a RESTful API for a User using Layer Architecture

Project Structure

```
/node-layered-sql
├── /src
│   ├── /config
│   │   ├── database.js
│   ├── /repositories
│   │   ├── userRepository.js
│   ├── /services
│   │   ├── userService.js
│   ├── /controllers
│   │   ├── userController.js
│   ├── /routes
│   │   ├── userRoutes.js
│   ├── app.js
├── .env
├── package.json
├── nodemon.json
```

Configure SQL Server Connection (`database.js`)

Create **`src/config/database.js`** to connect to SQL Server.

```
const sql = require("mssql");
const config = {
user: "root",
password: "root",
server: "localhost", // or your server name
database: "testingDB",
options: {
encrypt: false, // Disable encryption for local servers
trustServerCertificate: true,
```

```
}
};
const poolPromise = new sql.ConnectionPool(config)
.connect()
.then(pool => {
console.log("Connected to SQL Server");
return pool;
})
.catch(err => console.error("Database Connection Failed! ", err));
module.exports = { sql, poolPromise };
```

Connects to SQL Server using `mssql`.

Create Model

```
class User {
constructor(id, name, email, isActive = true) {
this.id = id;
this.name = name;
this.email = email;
this.isActive = isActive;
}
}

module.exports = User;
```

Create User Repository

Create **`src/repositories/userRepository.js`** to handle database queries.

```
const { poolPromise } = require("../config/database");

class UserRepository {
  async getAllUsers() {
    const pool = await poolPromise;
    const result = await pool.request().query("SELECT * FROM Users");
    return result.recordset;
  }

  async getUserById(id) {
```

```
    const pool = await poolPromise;
    const result = await pool.request().input("id", id).query("SELECT *
FROM Users WHERE id = @id");
    return result.recordset[0];
  }

  async createUser(name, email, isActive) {
    const pool = await poolPromise;
    await pool.request()
      .input("name", name)
      .input("email", email)
      .input("isActive", isActive)
      .query("INSERT INTO Users (name, email, isActive) VALUES (@name,
@email, @isActive)");
    return { message: "User added successfully" };
  }

  async deleteUser(id) {
    const pool = await poolPromise;
    await pool.request().input("id", id).query("DELETE FROM Users WHERE id
= @id");
    return { message: "User deleted successfully" };
  }
  async updateUser(id, name, email, isActive) {
    const pool = await poolPromise;
    await pool.request()
      .input("id", id)
      .input("name", name)
      .input("email", email)
      .input("isActive", isActive)
      .query("UPDATE Users SET name = @name, email = @email, isActive =
@isActive WHERE id = @id");
    return { message: "User updated successfully" };
  }

}

module.exports = new UserRepository();
```

Create User Service

Create **`src/services/userService.js`** to handle business logic.

```
const userRepository = require("../repository/userRepository");
```

```
class UserService {
  async getAllUsers() {
    return await userRepository.getAllUsers();
  }

  async getUserById(id) {
    return await userRepository.getUserById(id);
  }

  async createUser(name, email, isActive) {
    return await userRepository.createUser(name, email, isActive);
  }

  async deleteUser(id) {
    return await userRepository.deleteUser(id);
  }
  async updateUser(id, name, email, isActive) {
    return await userRepository.updateUser(id, name, email, isActive);
  }

}

module.exports = new UserService();
```

Calls repository functions to interact with the database.

Create Controller

Create **`src/controllers/userController.js`** to handle API requests.

```
const userService = require("../services/userService");

class UserController {
  async getAllUsers(req, res) {
    try {
      const users = await userService.getAllUsers();
      res.json(users);
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  }

  async getUserById(req, res) {
    try {
      const user = await userService.getUserById(req.params.id);
```

```
      if (!user) return res.status(404).json({ message: "User not found"
});
      res.json(user);
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  }

  async createUser(req, res) {
    try {
      const { name, email, isActive } = req.body;
      const result = await userService.createUser(name, email, isActive);
      res.status(201).json(result);
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  }

  async deleteUser(req, res) {
    try {
      const result = await userService.deleteUser(req.params.id);
      res.json(result);
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  }
  async updateUser(req, res) {
    try {
      const { name, email, isActive } = req.body;
      const result = await userService.updateUser(req.params.id, name,
email, isActive);
      res.json(result);
    } catch (err) {
      res.status(500).json({ error: err.message });
    }
  }

}

module.exports = new UserController();
```

Handles HTTP requests and responses.

Create Routes

Create **`src/routes/userRoutes.js`**.

```
const express = require("express");
const userController = require("../controllers/userController");

const router = express.Router();

router.get("/", userController.getAllUsers);
router.get("/:id", userController.getUserById);
router.post("/", userController.createUser);
router.delete("/:id", userController.deleteUser);
router.put("/:id", userController.updateUser);


module.exports = router;
```

Defines API endpoints.

Create Express App

Create **`src/app.js`**.

```
const express = require("express");
const userRoutes = require("./routes/userRoutes");

const app = express();
app.use(express.json());

// Routes
app.use("/users", userRoutes);

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

Loads database and routes.

Running the Project

```
npx nodemon src/app.js
```

- Nodemon will automatically restart on changes.

Assignment: Employee Leave Management System (ELMS) Using REST API & Layered Architecture (Without Authentication)

Problem Statement

Design and implement a Leave Management System for a company where employees can apply for leave, and managers can approve or reject leave requests. The system should be built using **Node.js, Express, SQL Server**, and a **layered architecture**.

Layers in the Application

1. **Controller Layer** - Handles API requests and responses.
2. **Service Layer** - Implements business logic and validation.
3. **Repository Layer** - Handles database operations (CRUD).
4. **Database Layer** - Manages SQL Server connection.
5. **Routing Layer** - Defines API endpoints.

Requirements

The system should support the following operations:

Employee Features

✔ Apply for leave (Start Date, End Date, Leave Type, Reason).
✔ View leave history and current leave status.

Manager Features

✔ View pending leave requests.
✔ Approve or reject leave requests.

Admin Features

✔ View reports of all leaves taken by employees.

API Endpoints

Employee APIs

1. **`POST /leaves`** - Apply for a new leave.
2. **`GET /leaves/:employeeId`** - Get leave history of an employee.

Manager APIs

3. **`GET /leaves/pending`** - Get all pending leave requests.
4. **`PUT /leaves/:leaveId/approve`** - Approve a leave request.
5. **`PUT /leaves/:leaveId/reject`** - Reject a leave request.

Admin APIs

6. **`GET /leaves/report`** - Get reports of all leaves taken by employees.

Database Schema (SQL Server)

Employee Table

id name email role (Employee/Manager/Admin)

Leave Table

id employee_id start_date end_date	**leave_type (Sick/Vacation)**	**status (Pending/Approved/Rejected)**	**reason**

Expected Outcome

Apply Layered Architecture to separate concerns.
Use REST API best practices for request handling.
Store leave requests in SQL Server
Handle errors properly in all layers.

Submission Guidelines

Code should be well-structured using folders:
`controllers/`, `services/`, `repositories/`, `routes/`, `config/`.
Include **Postman collection** for testing APIs.
Provide **SQL script** for creating tables.