Day 6 — REST API & API testing with Postman

Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

- **Day 5:**
  - Connecting Node.js with SQL databases
  - Introduction to Sequelize ORM
  - Models, migrations, and associations in Sequelize
  - CRUD operations with Sequelize
  - CRUD with Nestjs
- Day 6:
  - REST principles & HTTP status codes
  - Designing resource-oriented APIs (URIs, nesting, filtering, pagination)
  - API versioning strategies
  - API documentation basics (Swagger / OpenAPI)
  - API testing with Postman (environments, requests, tests, collections)

---

1. REST principles — essential ideas

- **Resources (nouns):** endpoints represent resources (`/employees`), not actions.
- **Uniform interface:** standard HTTP verbs — GET, POST, PUT, PATCH, DELETE.
- **Statelessness:** each request contains everything server needs.
- **Representation:** JSON is the default for modern REST APIs.
- **Idempotency:** PUT should be idempotent; POST creates a new resource.

Flow Diagram:

```
Client --HTTP--> API (Controller) --calls--> Service --DB--> Data
```

---

2. HTTP Status Codes — concise guide

- **2xx Success**
  - `200 OK` — GET/PUT/PATCH success with body
  - `201 Created` — POST created resource; include `Location` header
  - `204 No Content` — successful DELETE or no-body responses
- **4xx Client errors**
  - `400 Bad Request` — validation errors

  - `401 Unauthorized` — missing/invalid auth
  - `403 Forbidden` — authenticated but not allowed
  - `404 Not Found` — resource missing
  - `409 Conflict` — unique constraint violation (duplicate email/email unique)
- **5xx Server errors**
  - `500 Internal Server Error` — unhandled exceptions

Consistent error shape recommendation:

```
{ "statusCode": 400, "message": "Validation failed", "errors": [...] }
```

---

3. Designing resource-oriented APIs — best practices

- Use **plural nouns**: `/employees`
- Nested routes for ownership: `/employees/{id}/tasks`
- Filtering, sorting, pagination:
  - Filtering: `/employees?department=hr`
  - Sorting: `/employees?sort=-salary,name`
  - Pagination: `/employees?page=2&limit=20`
- Use `Location` header on successful resource creation.
- Keep error shapes consistent.

---

4. API versioning — strategies & recommendation

- **URI versioning**: `/api/v1/employees` — visible and simple.
- **Header versioning**: `Accept: application/vnd.app.v1+json` — cleaner URLs, more complex clients.
- **Recommendation:** use **URI versioning** for clarity: `app.setGlobalPrefix('api/v1')`.

In NestJS:

```
// main.ts
app.setGlobalPrefix('api/v1');
```

---

5. API documentation basics — Swagger / OpenAPI

- Auto-generate interactive docs with `@nestjs/swagger`.
- Install:

```
npm install @nestjs/swagger swagger-ui-express
```

- Bootstrap Swagger:

```
// main.ts (inside bootstrap)
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

const config = new DocumentBuilder()
  .setTitle('Employees API')
  .setDescription('API docs for Employees service')
  .setVersion('1.0')
  .addBearerAuth()
  .build();

const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('api/docs', app, document);
```

- Annotate DTOs and controllers with `@ApiProperty` etc. to enrich docs.

---

6. API testing with Postman

Setup

1. Install Postman.
2. Create environment `Local` with `baseUrl = http://localhost:3000/api/v1`.

Collection

Create a Postman collection `Employees API` with requests:

- `POST {{baseUrl}}/employees` (create)
- `GET {{baseUrl}}/employees` (list)
- `GET {{baseUrl}}/employees/:id` (get)
- `PUT {{baseUrl}}/employees/:id` (update)
- `DELETE {{baseUrl}}/employees/:id` (delete)

Postman tests example (Create)

Tests tab:

```
pm.test("Status is 201", () => pm.response.to.have.status(201));
const json = pm.response.json();
pm.expect(json).to.have.property('id');
pm.environment.set('createdEmployeeId', json.id);
```

Pre-request script (generate test email)

```
pm.environment.set('testEmail',
`emp_${Math.floor(Math.random()*100000)}@example.com`);
```

Run collections in CI

Use Newman:

```
newman run Employees.postman_collection.json -e
Local.postman_environment.json --reporters cli,junit --reporter-junit-
export newman-report.xml
```

---

7. Sample code — Employees API (NestJS + Sequelize + Swagger + DTO + Versioning)

Note: code below uses `sequelize-typescript` decorators and NestJS controllers/services. Swap Oracle config if you already use Oracle (Day 5 covers Oracle configuration).

DTO: CreateEmployeeDto

```
// src/modules/employees/dto/create-employee.dto.ts
import { IsString, IsEmail, IsDateString, IsNumber, Min } from 'class-
validator';
import { ApiProperty } from '@nestjs/swagger';

export class CreateEmployeeDto {
  @ApiProperty({ example: 'Ravi Kumar' })
  @IsString() name!: string;

  @ApiProperty({ example: '2021-07-15' })
  @IsDateString() dateOfJoining!: string;

  @ApiProperty({ example: 'ravi.kumar@example.com' })
  @IsEmail() email!: string;

  @ApiProperty({ example: 55000 })
  @IsNumber() @Min(0) salary!: number;

  // Optional for employee create if admin sets password
  @ApiProperty({ example: 'secret123', required: false })
  @IsString()
  password?: string;
}
```

Employee model

```
// src/models/employee.model.ts
import { Table, Column, Model, DataType, PrimaryKey, AutoIncrement } from
'sequelize-typescript';

@Table({ tableName: 'EMPLOYEE', timestamps: true }) // timestamps true adds
createdAt/updatedAt
export class Employee extends Model<Employee> {
  @PrimaryKey
  @AutoIncrement
  @Column({ type: DataType.INTEGER })
  id!: number;

  @Column({ type: DataType.STRING(100), allowNull: false })
  name!: string;

  @Column({ type: DataType.DATE, allowNull: false })
  dateOfJoining!: Date;

  @Column({ type: DataType.STRING(100), allowNull: false, unique: true })
  email!: string;

  @Column({ type: DataType.FLOAT, allowNull: false })
  salary!: number;

  @Column({ type: DataType.STRING(255), allowNull: true })
  password?: string;
}
```

EmployeesService (key methods)

```
// src/modules/employees/employees.service.ts
import { Injectable, ConflictException } from '@nestjs/common';
import { InjectModel } from '@nestjs/sequelize';
import { Employee } from '../../models/employee.model';
import { CreateEmployeeDto } from './dto/create-employee.dto';
import * as bcrypt from 'bcrypt';

@Injectable()
export class EmployeesService {
  constructor(@InjectModel(Employee) private employeeModel: typeof
Employee) {}

  async create(dto: CreateEmployeeDto): Promise<Employee> {
    const exists = await this.employeeModel.findOne({ where: { email:
dto.email } });
    if (exists) throw new ConflictException('Email already exists');
    const hash = dto.password ? await bcrypt.hash(dto.password, 10) :
undefined;
    const created = await this.employeeModel.create({
      ...dto,
      dateOfJoining: new Date(dto.dateOfJoining),
      password: hash,
    });
    return created;
  }

  findAll(): Promise<Employee[]> {
    return this.employeeModel.findAll({ attributes: { exclude: ['password']
} as any });
  }

  findOne(id: number): Promise<Employee | null> {
    return this.employeeModel.findByPk(id, { attributes: { exclude:
['password'] } as any });
  }

  async update(id: number, dto: Partial<CreateEmployeeDto>):
Promise<Employee | null> {
    const emp = await this.employeeModel.findByPk(id);
    if (!emp) return null;
    if (dto.password) dto.password = await bcrypt.hash(dto.password, 10);
    if (dto.dateOfJoining) (dto as any).dateOfJoining = new
Date(dto.dateOfJoining);
    await emp.update(dto);
    const safe = await this.employeeModel.findByPk(id, { attributes: {
exclude: ['password'] } as any });
    return safe;
  }

  async remove(id: number): Promise<boolean> {
    const emp = await this.employeeModel.findByPk(id);
    if (!emp) return false;
    await emp.destroy();
    return true;
  }
}
```

EmployeesController

```
// src/modules/employees/employees.controller.ts
import { Controller, Get, Post, Put, Delete, Param, Body, UsePipes,
ValidationPipe, Res } from '@nestjs/common';
import { ApiTags, ApiOperation } from '@nestjs/swagger';
import { EmployeesService } from './employees.service';
import { CreateEmployeeDto } from './dto/create-employee.dto';
import { Response } from 'express';

@ApiTags('Employees')
@Controller({ path: 'employees', version: '1' })
export class EmployeesController {
  constructor(private employeesService: EmployeesService) {}

  @Get()
  @ApiOperation({ summary: 'List employees' })
  findAll() {
    return this.employeesService.findAll();
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.employeesService.findOne(Number(id));
  }

  @Post()
  @UsePipes(new ValidationPipe({ transform: true }))
  async create(@Res() res: Response, @Body() dto: CreateEmployeeDto) {
    const emp = await this.employeesService.create(dto);
    // Set Location header pointing to the new resource
    res.location(`/api/v1/employees/${emp.id}`);
    // return safe payload excluding password (service returns full; map to
safe)
    const { password, ...safe } = emp.toJSON() as any;
    return res.status(201).json(safe);
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() dto: Partial<CreateEmployeeDto>)
{
    return this.employeesService.update(Number(id), dto);
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    return this.employeesService.remove(Number(id));
  }
}
```

8. Testing examples

Create employee (cURL)

```
curl -X POST http://localhost:3000/api/v1/employees \
 -H "Content-Type: application/json" \
 -d '{"name":"Ravi Kumar","dateOfJoining":"2021-07-
15","email":"ravi.kumar@example.com","salary":55000,"password":"secret"}'
```

Get All employees (cURL)

```
curl "http://localhost:3000/api/v1/employees"
```

Get by ID

```
curl "http://localhost:3000/api/v1/employees/1"
```

Update

```
curl -X PUT "http://localhost:3000/api/v1/employees/1" -H "Content-Type:
application/json" -d '{"salary":60000}'
```

Delete

```
curl -X DELETE "http://localhost:3000/api/v1/employees/1"
```

9. Tips & best practices

- Return consistent error shapes; map DB constraint errors to `409 Conflict`.
- Do **not** return `password` in API responses — exclude at query level or map before returning.
- Use DTOs + ValidationPipe to validate early.
- Use transactions for multi-step writes.
- Document with Swagger and run Postman tests in CI with Newman.

10. Summary

- REST APIs should be resource-oriented, stateless, and use proper HTTP verbs and status codes.
- Version APIs early (`/api/v1`) to allow non-breaking future changes.
- DTOs + validation keep the API robust.
- Swagger provides interactive API docs; Postman + Newman automate testing.
- For Employee data, never leak sensitive fields (password), and enforce DB constraints (unique email) with proper error mapping.

Step-by-step to implement & test (walkthrough)

1. **Install dependencies**

```
npm install @nestjs/sequelize sequelize sequelize-typescript reflect-
metadata bcrypt
# if using Oracle: npm install oracledb
npm install -D typescript ts-node @types/sequelize @types/bcrypt
```

2. **Ensure tsconfig has decorator settings**

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true
```

3. **Add SequelizeModule in `app.module.ts`** (Day 5 has Oracle config). Ensure `models: [Employee]` is registered.
4. **Create DTO, Model, Service, Controller** files above under `src/modules/employees/`.
5. **Start the app**

```
npm run dev  # ts-node / nodemon setup
```

6. **Test create endpoint with Postman**:

- POST `{{baseUrl}}/employees` with JSON body (see earlier sample).
- Expect `201 Created`, response body without `password`, and `Location` header set.

7. **Test GET list**:

- GET `{{baseUrl}}/employees` — see objects without password.

8. **Test GET by id** — check `Location` URL returns the created object.
9. **Test duplicate email** — create another with same email → expect `409 Conflict`.
10. **Test update & delete** — run PUT/DELETE and verify.