Day 4 — CRUD in NestJS

Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

- **Day 3 Recap:**
  - Explain what Express.js and NestJS are, and when to use them.
  - Create a simple CRUD API in Express using modern ESM syntax.
  - Understand and use middleware and static file serving.
  - Explain what a DTO is, why it's used, and apply validation using decorators.
  - Build Employee CRUD APIs using Express and NestJS.
  - Implement request validation and file upload handling in NestJS.
  - Test APIs using **Bruno** or **curl**.
- **Day 4**
  - Use the Nest CLI to scaffold modules, controllers, services or an entire resource.
  - Implement an Employees module with in-memory CRUD (GET, POST, PUT, DELETE).
  - Create and use DTOs with `class-validator` and `ValidationPipe`.
  - Handle exceptions with the Nest built-in exceptions and an optional global exception filter.
  - Follow feature-based folder structure conventions.
  - Test endpoints with Bruno and `curl`.

Quick primers

- **NestJS**: a TypeScript Node.js framework using decorators and dependency injection.
- **Nest CLI**: `nest` command line tool to scaffold projects and code.
- **DTO**: Data Transfer Object — a class describing request/response shapes + validation rules.
- **ValidationPipe**: a global pipe that validates incoming DTOs using `class-validator`.
- **ExceptionFilter**: catches exceptions and formats responses.

Install core packages (if not already):

```
# globally (optional)
npm i -g @nestjs/cli

# inside project
nest new nest-employee        # scaffold project via CLI (choose npm/yarn)
cd nest-employee
```

```
npm install class-validator class-transformer @nestjs/mapped-types
```

High-level architecture

```
Client (Bruno/curl)
   |
   v
Nest App (main.ts)
   |
   +--> Global Pipes (ValidationPipe)
   +--> Global Filters (AllExceptionsFilter)
   |
   +--> Modules
         └─ EmployeesModule
             ├─ EmployeesController
             └─ EmployeesService
```

Nest CLI: generate commands — explanation + usage

The Nest CLI speeds up scaffolding. Use `nest g` (or `nest generate`) with resource, module, controller, or service.

1) Generate a module

```
nest generate module employees
# or short
nest g mo employees
```

What it creates

- `src/employees/employees.module.ts` with a `@Module` skeleton.

When to use

- When adding a new feature group (employees, auth, products).

2) Generate a service

```
nest generate service employees/employees --no-spec
# or
nest g s employees/employees --no-spec
```

What it creates

- `src/employees/employees.service.ts` (and by default a `.spec.ts` test file unless `--no-spec` is used).

Why

- Service holds business logic and data access (keep controllers thin).

3) Generate a controller

```
nest generate controller employees/employees --no-spec
# or
nest g co employees/employees --no-spec
```

What it creates

- `src/employees/employees.controller.ts` with a basic controller class.

Notes

- Add route path in decorator `@Controller('employees')` (CLI may prompt).

4) Generate a resource (module + controller + service + DTOs)

```
nest generate resource employees
# or short
nest g resource employees
```

The `resource` command is interactive (or can accept flags). CLI will ask:

- What transport layer? (choose `REST API`)
- Would you like CRUD entry points? (yes)
- Which path? (default `employees`)

You can run non-interactively:

```
nest g resource employees --type rest --no-spec
```

What it creates

- `employees.module.ts`, `employees.controller.ts`, `employees.service.ts`,
- DTOs (`create-employee.dto.ts`, `update-employee.dto.ts`) if `--crud`/interactive chose CRUD,
- Optional `.spec.ts` test files (use `--no-spec` to skip tests).

When to use

- To scaffold a standard REST resource quickly — you'll still replace generated code with your actual implementation.

How to use the generated files (recommended workflow)

1. `nest g resource employees --type rest --no-spec`
2. Open `src/employees/` — CLI created module/controller/service and DTO skeletons.
3. Replace generated controller/service logic with the in-memory code below (or adapt it).
4. Add DTO validation rules.
5. Register global `ValidationPipe` (in `main.ts`) and optional exception filter.
6. Run via `npm run start:dev`.

Employee API CRUD

Place these files under `src/` in your Nest project.

`src/main.ts` — bootstrap, global ValidationPipe and filter

```
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { AllExceptionsFilter } from './common/filters/http-
exception.filter';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

    // Add this to enable validation globally
  app.useGlobalPipes(new ValidationPipe({
    whitelist: true, // Strip properties that don't have decorators
    forbidNonWhitelisted: true, // Throw error if non-whitelisted
properties are present
    transform: true, // Automatically transform payloads to DTO instances
  }));

  app.useGlobalFilters(new AllExceptionsFilter());
  await app.listen(3000);
  console.log('Nest server running at http://localhost:3000');
}
bootstrap();
```

`src/app.module.ts`

```
import { Module } from '@nestjs/common';
import { EmployeesModule } from './employees/employees.module';

@Module({
  imports: [EmployeesModule],
})
```

```
export class AppModule {}
```

---

src/employees/employees.module.ts

```
import { Module } from '@nestjs/common';
import { EmployeesController } from './employees.controller';
import { EmployeesService } from './employees.service';

@Module({
  controllers: [EmployeesController],
  providers: [EmployeesService],
})
export class EmployeesModule {}
```

---

src/employees/entities/employee.entity.ts

```
export class Employee {
  id: number;
  name: string;
  email: string;
  dateOfBirth: string; // 'YYYY-MM-DD'
  mobile: string;      // 10 digits
}
```

---

src/employees/dto/create-employee.dto.ts

```
import { IsEmail, IsNotEmpty, IsOptional, IsString, Matches } from 'class-
validator';

export class CreateEmployeeDto {
  @IsNotEmpty()
  @IsString()
  name: string;

  @IsNotEmpty()
  @IsEmail()
  email: string;

  @IsNotEmpty()
  @Matches(/^\d{4}-\d{2}-\d{2}$/, { message: 'dateOfBirth must be in YYYY-
MM-DD format' })
  dateOfBirth: string;

  @IsNotEmpty()
  @Matches(/^\d{10}$/, { message: 'mobile must be 10 digits' })
  mobile: string;

  @IsOptional()
  @IsString()
  role?: string;
}
```

---

src/employees/dto/update-employee.dto.ts

```
import { PartialType } from '@nestjs/mapped-types';
import { CreateEmployeeDto } from './create-employee.dto';

export class UpdateEmployeeDto extends PartialType(CreateEmployeeDto) {}
```

`src/employees/employees.service.ts` (in-memory CRUD)

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { CreateEmployeeDto } from './dto/create-employee.dto';
import { UpdateEmployeeDto } from './dto/update-employee.dto';
import { Employee } from './entities/employee.entity';

@Injectable()
export class EmployeesService {
  private employees: Employee[] = [
    { id: 1, name: 'Alice Johnson', email: 'alice@example.com',
dateOfBirth: '1998-05-15', mobile: '9876543210' },
    { id: 2, name: 'Bob Smith', email: 'bob@example.com', dateOfBirth:
'1995-10-20', mobile: '9123456780' },
  ];

  private getNextId(): number {
    return this.employees.length > 0 ? Math.max(...this.employees.map(e =>
e.id)) + 1 : 1;
  }

  findAll(): Employee[] {
    return this.employees;
  }

  findOne(id: number): Employee {
    const emp = this.employees.find(e => e.id === id);
    if (!emp) throw new NotFoundException(`Employee with id ${id} not
found`);
    return emp;
  }

  create(dto: CreateEmployeeDto): Employee {
    const newEmp: Employee = { id: this.getNextId(), ...dto };
    this.employees.push(newEmp);
    return newEmp;
  }

  update(id: number, dto: UpdateEmployeeDto): Employee {
    const idx = this.employees.findIndex(e => e.id === id);
    if (idx === -1) throw new NotFoundException(`Employee with id ${id} not
found`);
    this.employees[idx] = { ...this.employees[idx], ...dto };
    return this.employees[idx];
  }

  remove(id: number): void {
    const idx = this.employees.findIndex(e => e.id === id);
    if (idx === -1) throw new NotFoundException(`Employee with id ${id} not
found`);
    this.employees.splice(idx, 1);
  }
}
```

`src/employees/employees.controller.ts`

```
import { Body, Controller, Delete, Get, Param, Post, Put } from
'@nestjs/common';
import { CreateEmployeeDto } from './dto/create-employee.dto';
import { UpdateEmployeeDto } from './dto/update-employee.dto';
import { EmployeesService } from './employees.service';

@Controller('employees')
export class EmployeesController {
  constructor(private readonly employeesService: EmployeesService) {}

  @Get()
  findAll() {
    return this.employeesService.findAll();
  }

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

  @Post()
  create(@Body() dto: CreateEmployeeDto) {
    return this.employeesService.create(dto);
  }

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

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

`src/common/filters/http-exception.filter.ts` (global exception filter)

```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException, HttpStatus }
from '@nestjs/common';
import { Request, Response } from 'express';

@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();

    let status = HttpStatus.INTERNAL_SERVER_ERROR;
    let message: any = 'Internal server error';

    if (exception instanceof HttpException) {
      status = exception.getStatus();
      const resBody = exception.getResponse();
      message = (resBody && typeof resBody === 'object' && (resBody as
any).message) ? (resBody as any).message : resBody;
    }

    response.status(status).json({
      statusCode: status,
      timestamp: new Date().toISOString(),
      path: request.url,
      error: message,
    });
  }
}
```

Notes about the code

- Numeric `ids` (auto-increment via `getNextId`) — simpler and deterministic for teaching.
- DTOs validate `name`, `email`, `dateOfBirth` (`YYYY-MM-DD`) and `mobile` (10 digits) using `class-validator`.
- `ValidationPipe` with `{ whitelist: true, transform: true }` strips unknown props and converts types where possible.
- `NotFoundException` thrown in service maps to 404 automatically; the global filter wraps responses into a consistent JSON shape.

Practical steps — generate + replace (example workflow)

1. Scaffold project (if not done):
2. `nest new nest-employee`
3. `cd nest-employee`
4. `npm install class-validator class-transformer @nestjs/mapped-types`
5. Use CLI to scaffold resource:
6. `nest g resource employees --type rest --no-spec`
    - CLI creates `employees.module.ts`, `employees.controller.ts`, `employees.service.ts`, and DTOs (depending on options).
    - Open the generated files and replace the contents with the code provided above.
7. Add the `common/filters/http-exception.filter.ts` file.
8. Update `src/main.ts` as shown to enable global validation and the filter.
9. Start dev server:
10. `npm run start:dev`

11. Test endpoints with Bruno or `curl`.

Bruno testing (step-by-step)

Create (POST)

- URL: `http://localhost:3000/employees`
- Method: POST
- Headers: `Content-Type: application/json`
- Body:

```
{
  "name": "Carol Adams",
  "email": "carol@example.com",
  "dateOfBirth": "2000-01-01",
  "mobile": "9988776655"
}
```

- Expected: `201 Created` with new employee object including numeric `id`.

Get all (GET)

- URL: `http://localhost:3000/employees` → should return array.

Get one (GET)

- URL: `http://localhost:3000/employees/1` → returns single employee or 404.

Update (PUT)

- URL: `http://localhost:3000/employees/1`
- Body (JSON): `{"mobile":"9001122334"}` → returns updated object.

Delete (DELETE)

- URL: `http://localhost:3000/employees/1` → returns `{ status: 'deleted' }` or 204 depending on controller.

Invalid payload

- Omit `email` on create → validation error (400) returned by ValidationPipe and formatted by filter.

---

curl quick references

Create

```
curl -s -X POST http://localhost:3000/employees \
  -H "Content-Type: application/json" \
  -d '{"name":"Carol
Adams","email":"carol@example.com","dateOfBirth":"2000-01-
01","mobile":"9988776655"}' | jq
```

Get all

```
curl -s http://localhost:3000/employees | jq
```

Get one

```
curl -s http://localhost:3000/employees/1 | jq
```

Update

```
curl -s -X PUT http://localhost:3000/employees/1 \
  -H "Content-Type: application/json" \
  -d '{"mobile":"9001122334"}' | jq
```

Delete

```
curl -s -X DELETE http://localhost:3000/employees/1
```

Practice activity

Goal: add a Department CRUD API and update Employee API to accept `departmentId`, verifying existence.

Steps

1. In existing Nest project, add department:
   - `nest g resource departments --type rest --no-spec`
2. Add code to the generated `departments.*` files.
3. Update Employees DTOs (`create-employee.dto.ts`, `update-employee.dto.ts`) to include `departmentId`.
4. Update `employees.service.ts` to inject `DepartmentsService` and validate `departmentId` in `create/update`.
5. Update `employees.module.ts` to `imports: [DepartmentsModule]`.
6. Ensure `departments.module.ts` exports `DepartmentsService` (code above does this).
7. Start server
8. Test in Bruno:
   - POST `/departments` to create one (note id).

- POST `/employees` with `departmentId` set to that id — expect `201`.
- POST `/employees` with `departmentId` missing or invalid — expect `404` if invalid (department not found) or success if missing (department optional).
- PUT `/employees/:id` to change `departmentId`, check validation.

Bonus tasks

- Add a simple referential integrity rule: prevent deleting a department with existing employees. (Hint: `DepartmentsService.remove` can call a function on `EmployeesService` — requires arranging module imports/export carefully and using `forwardRef` if circular.)
- Add `GET /departments/:id/employees` to list employees in a department — implement in `EmployeesService` a method to filter by departmentId; expose route in DepartmentsController.

Best practices

- **Module per feature**: group controller/service/dto/entity under `employees/`.
- **Thin controllers, fat services**: controllers call services only.
- **DTOs first**: design DTOs before writing controller/service.
- **ValidationPipe globally**: provides consistent validation and transformation.
- **Exception handling**: use built-in exceptions; add a filter for consistent error payloads.
- **No in-memory for prod**: use a DB (Prisma/TypeORM) in real apps.
- **Use Nest CLI**: speeds scaffolding and enforces convention; always review generated code.