upGrad ENTERPRISE

Day 3— Express.js + NestJS

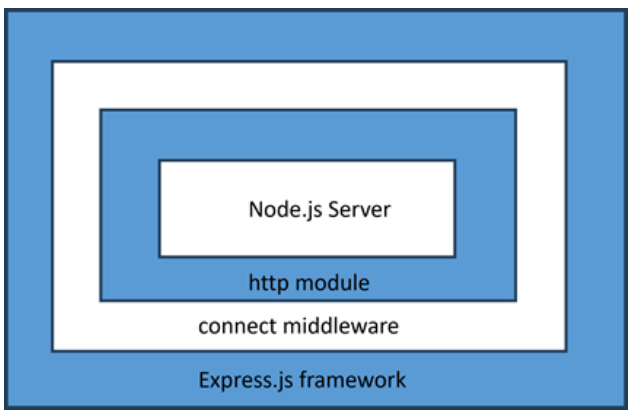
Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

- **Day 2 Recap:**
  - Modules & require/import
  - Callbacks, Promises, and async/await
  - Error handling in Node.js
  - Working with the filesystem (CRUD)
  - Streams & buffers basics
- **Day 3 Agenda:**
  - 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**.

Express.js Framework

1. Introduction to Express.js

Express.js is a **minimal, unopinionated Node.js framework** used for building web servers and REST APIs.
It's built around **routing** and **middleware** — small functions that process requests step by step.

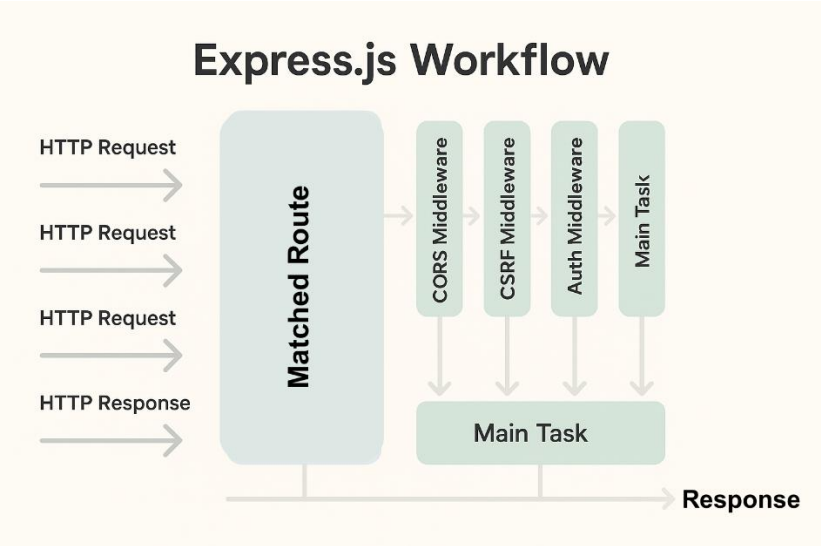


Key features

- Lightweight, fast, flexible.
- Works directly on top of Node.js HTTP module.
- Perfect for small to medium APIs.
- Extensible using middleware.

2. Express Architecture

```
Client ---> [Express App] ---> Middleware ---> Router ---> Controller --->
Service ---> Database
                     |                          |                ^
                     |                          |                |
                     +--> Static Files (public) +----------------+
```



Code Example – Without Express (pure Node.js):

```
const http = require("http"); //ESM?

const server = http.createServer((req, res) => {
  if (req.url === "/") {
    res.end("Hello World!");
  }
});

server.listen(3000, () => console.log("Server running..."));
```

Same using Express:

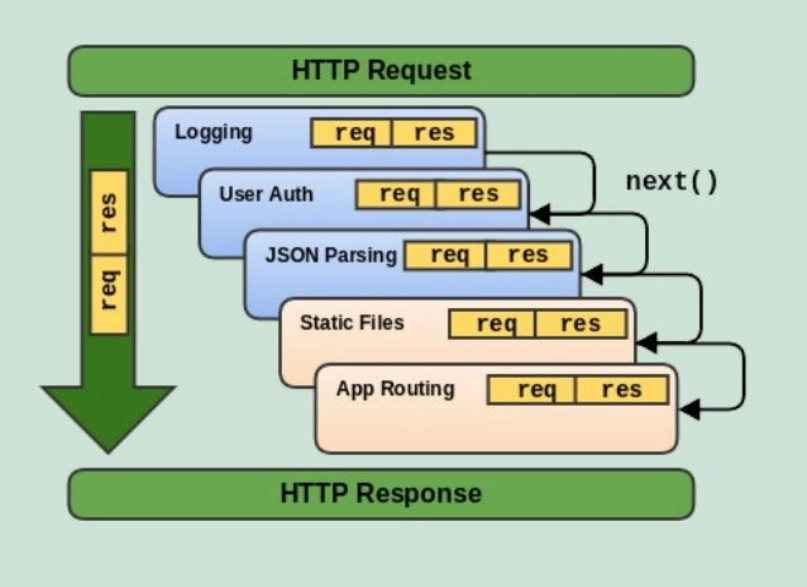
```
const express = require("express");
const app = express();

app.get("/", (req, res) => res.send("Hello World from Express!"));
app.listen(3000, () => console.log("Server started on port 3000"));
```

3. Middleware

Middleware = functions that run **between** the request and response.

Why middleware?

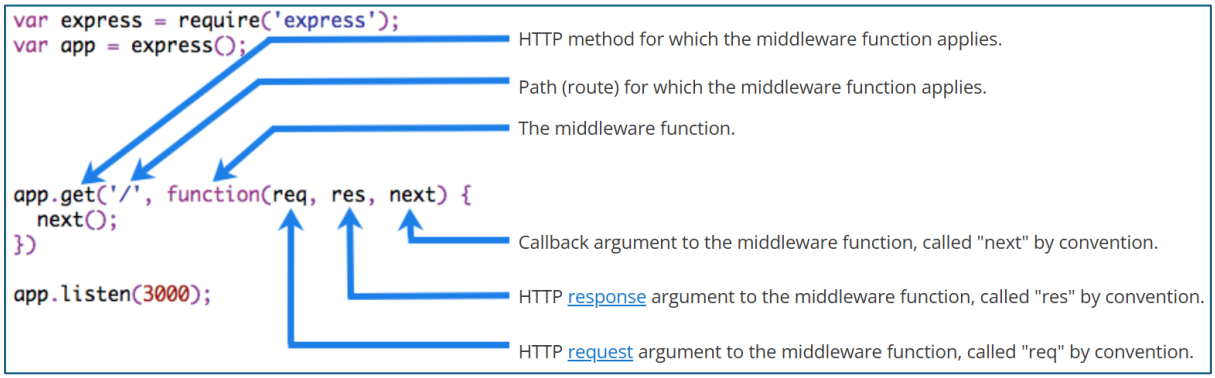
- Logging requests
- Parsing incoming data
- Handling errors
- Authentication
- Serving static files



Types of middleware:

Type	Example	Description
Application-level	`app.use(logger)`	Affects all routes
Router-level	`router.use(auth)`	Affects only that router
Built-in	`express.json()`	Parse JSON body
Error-handling	`(err, req, res, next)`	Handles exceptions
Third-party	`morgan`, `cors`	Logging, CORS, etc.

Execution order matters!
First registered middleware executes first.



Middleware Types:

- Application-level (`app.use()`)
- Route-level (specific routes)
- Built-in (`express.json()`, `express.static()`)
- Error-handling (with 4 parameters)

Example – Logger Middleware

```
const express = require("express");
const app = express();

// Middleware: logs every request
app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});

app.get("/", (req, res) => res.send("Home Page"));
app.get("/about", (req, res) => res.send("About Page"));

app.listen(3000, () => console.log("Server running..."));
```

---

4. Setting Up an Express Project (ESM)

Step 1: Initialize project

```
mkdir express-employee
cd express-employee
npm init -y
```

Step 2: Enable ESM

Add `"type": "module"` to `package.json`.

Step 3: Install dependencies

```
npm install express morgan uuid
npm install --save-dev nodemon
```

Step 4: Add scripts to package.json

```
"scripts": {
  "start": "node src/index.js",
  "dev": "nodemon --watch src --exec node src/index.js"
}
```

---

5. Simple CRUD API (Generic Example)

src/index.js

```
import express from 'express';
import morgan from 'morgan';
import { v4 as uuid } from 'uuid';

const app = express();

// Built-in middleware
app.use(express.json()); // replaces body-parser
app.use(morgan('dev'));

// In-memory store
const items = new Map();

// Routes
app.get('/api/items', (req, res) => {
  res.json(Array.from(items.values()));
});

app.get('/api/items/:id', (req, res) => {
  const item = items.get(req.params.id);
  if (!item) return res.status(404).json({ error: 'Item not found' });
  res.json(item);
});

app.post('/api/items', (req, res) => {
  const { name } = req.body;
  if (!name) return res.status(400).json({ error: 'name required' });
  const id = uuid();
  const item = { id, name };
  items.set(id, item);
  res.status(201).json(item);
});

app.put('/api/items/:id', (req, res) => {
  const existing = items.get(req.params.id);
  if (!existing) return res.status(404).json({ error: 'Item not found' });
  const updated = { ...existing, ...req.body };
  items.set(req.params.id, updated);
  res.json(updated);
});

app.delete('/api/items/:id', (req, res) => {
  if (!items.has(req.params.id)) return res.status(404).json({ error: 'Item
not found' });
  items.delete(req.params.id);
```

```
  res.status(204).send();
});

app.listen(3000, () => console.log('Express server running at
http://localhost:3000'));
```

Test using Bruno or curl

Bruno

- Create a new request → Method: `POST`, URL: `http://localhost:3000/api/items`
- Set body → JSON → `{ "name": "toy" }`
- Send

curl

```
curl -X POST http://localhost:3000/api/items \
  -H "Content-Type: application/json" \
  -d '{"name":"toy"}'
```

---

6. Employee CRUD API (Express)

src/employees.js

```
import express from 'express';

const router = express.Router();

// In-memory array to simulate database
let employees = [
  {
    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'
  }
];

// Helper to generate next numeric ID
const getNextId = () => {
  return employees.length > 0 ? Math.max(...employees.map(e => e.id)) + 1 :
1;
};

// CREATE
router.post('/', (req, res) => {
```

```
  const { name, email, dateOfBirth, mobile } = req.body;

  // Simple validation
  if (!name || !email || !dateOfBirth || !mobile) {
    return res.status(400).json({ error: 'name, email, dateOfBirth and
mobile are required' });
  }

  const newEmp = {
    id: getNextId(),
    name,
    email,
    dateOfBirth,
    mobile
  };

  employees.push(newEmp);
  res.status(201).json(newEmp);
});

// READ ALL
router.get('/', (req, res) => {
  res.json(employees);
});

// READ ONE
router.get('/:id', (req, res) => {
  const id = Number(req.params.id);
  const emp = employees.find(e => e.id === id);

  if (!emp) return res.status(404).json({ error: 'Employee not found' });
  res.json(emp);
});

// UPDATE
router.put('/:id', (req, res) => {
  const id = Number(req.params.id);
  const index = employees.findIndex(e => e.id === id);

  if (index === -1) return res.status(404).json({ error: 'Employee not
found' });

  const { name, email, dateOfBirth, mobile } = req.body;
  const existing = employees[index];

  // Update only provided fields
  employees[index] = {
    ...existing,
    name: name ?? existing.name,
    email: email ?? existing.email,
    dateOfBirth: dateOfBirth ?? existing.dateOfBirth,
    mobile: mobile ?? existing.mobile
  };

  res.json(employees[index]);
});

// DELETE
router.delete('/:id', (req, res) => {
```

```
  const id = Number(req.params.id);
  const index = employees.findIndex(e => e.id === id);

  if (index === -1) return res.status(404).json({ error: 'Employee not
found' });

  employees.splice(index, 1);
  res.status(204).send();
});

export default router;
```

Mount it in `src/index.js`

```
import employeesRouter from './employees.js';
app.use('/api/employees', employeesRouter);
```

Static Files

Serve static assets:

```
app.use(express.static('public'));
```

`public/index.html` can be loaded at http://localhost:3000/.

Serving Static Files

- Express can serve static HTML, CSS, JS, or image files from a folder.
- Use `express.static()` middleware.

Why Static Files Are Served Separately in Node.js

- Static files (HTML, CSS, JS, images) don't change often — can be **cached and reused**.
- Improves **performance** — no need for server-side processing on each request.
- Keeps the **Node.js event loop free** for dynamic API or database requests.
- Enables **browser and CDN caching** → faster load times.
- Provides **security isolation** — avoids exposing backend code or sensitive files.
- Allows **independent scaling** of frontend (static) and backend (API) servers.
- Supports **clean architecture** — separation of concerns between UI and logic.
- Middleware like express.static() can serve files **faster than custom routes**.
- Static servers (CDN/Nginx) can **compress and optimize** delivery automatically.
- Considered a **best practice in production** — Node.js focuses on APIs, not asset delivery.

Testing (Bruno + curl)

curl examples:

```
# Create
curl -X POST http://localhost:3000/api/employees \
  -H "Content-Type: application/json" \
  -d '{"name":"Bruno Tester","email":"bruno@example.com","role":"HR"}'

# Get all
curl http://localhost:3000/api/employees

# Update
curl -X PUT http://localhost:3000/api/employees/<id> \
  -H "Content-Type: application/json" \
  -d '{"role":"Manager"}'

# Delete
curl -X DELETE http://localhost:3000/api/employees/<id>
```

NestJS

1. Why NestJS?

NestJS is a **progressive Node.js framework** built with **TypeScript** and inspired by Angular architecture.

Key advantages

- Modular architecture (Modules, Controllers, Services)
- Built-in dependency injection
- Type safety with decorators
- Powerful middleware, guards, pipes, and interceptors
- Integrated validation, file upload, and static serving support

Visual overview

```
Client
  ↓
Controller → Service → Repository (DB)
  ↑
  └── Middleware / Pipes / Interceptors
```

✅ Use Express if you need:
A quick API
Maximum flexibility
Minimal abstraction

✅ Use NestJS if you want:
Scalable architecture
Maintainable codebases
Built-in tools for modern apps (Dependency Injection, GraphQL, WebSockets, Microservices)

NestJS vs Express

Feature	express	nestjs
Architecture	Minimal, unopinionated	Modular, Angular-style architecture
TypeScript	Optional	Native & bulit-in
Dependency Injection	Manual or external	Built-in, scalable Di system
Routing	Middleware-based	Decorator-based (clean & readable)
Testing	Manual setup	Built-in Jest support, mocks, testing
WebSockets	Needs integration	Out-of-the-box support
GraphQL	Add via Apollo	First-class support via @nestis/graphql
Microservices	Requires custom logic	Built-in support (Karka, Redis, MQTT,etc.)
CLI Tooling	Basic (e.g.. express-generator)	Powerful CLI (generate modules, services, etc.)
Middleware & Interceptors	app.use(), custom	Middeware + guards, pipes, interceptors
Performance	Lightweight	Slightly heavier, but scalable
Learning Curve Community & Ecosytern	Shallow – quick to start	Moderate – but structured and enterprise-ready

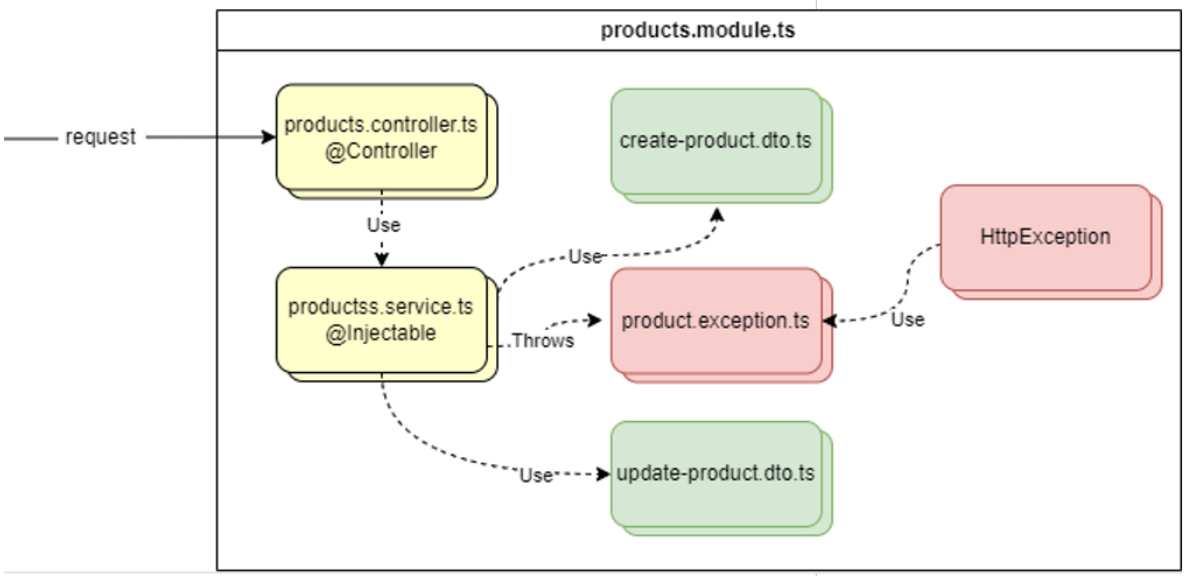
2. Setting Up a Nest Project

```
npm i -g @nestjs/cli
nest new nest-employee
cd nest-employee
npm i class-validator class-transformer @nestjs/mapped-types multer
@nestjs/platform-express @nestjs/serve-static
```

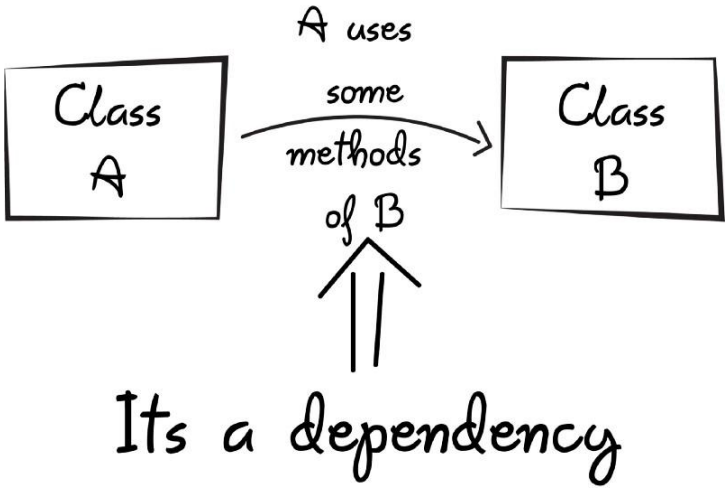
---

3. NestJS Folder Structure

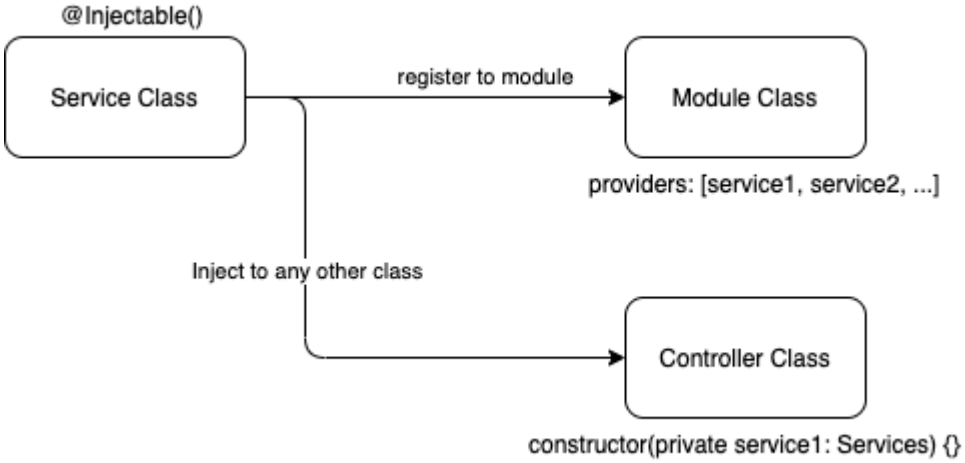
```
nest-employee/
├── src/
│   ├── main.ts
│   ├── app.module.ts
│   ├── employees/
│   │   ├── employees.controller.ts
│   │   ├── employees.service.ts
│   │   ├── employees.module.ts
│   │   ├── dto/
│   │   │   ├── create-employee.dto.ts
│   │   │   └── update-employee.dto.ts
│   │   └── entities/
│   │       └── employee.entity.ts
│   └── common/
│       └── middleware/
│           └── logger.middleware.ts
```

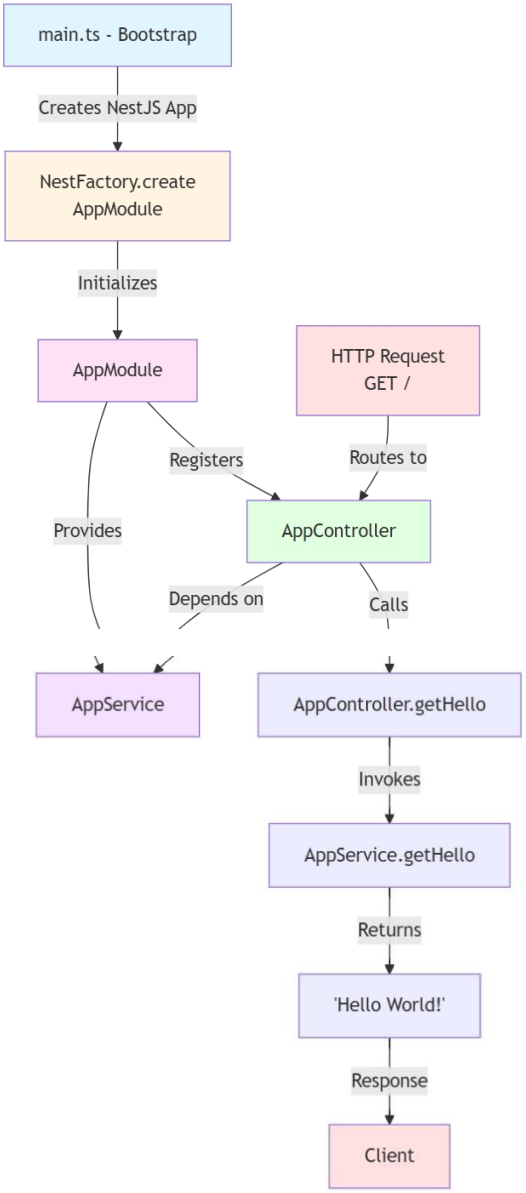


Dependency Injection



Read: *First steps | NestJS - A progressive Node.js framework*





4. Simple CRUD Example

[Controllers | NestJS - A progressive Node.js framework]

Minimal Controller + Service Example (TypeScript)

Nest generates boilerplate. Example of a simple `Cats` module:

Create cats module `nest g module cats`

1. Cat Data Transfer Object (DTO)
defines the shape of the data you expect from a POST request body.

```
// src/cats/dto/create-cat.dto.ts
export class CreateCatDto {
```

```
  readonly name: string;
  readonly age: number;
  readonly breed: string;
}
```

---

2. Missing File: Cat Interface

It's a best practice to create an interface that defines the shape of your Cat object.

```
// src/cats/interfaces/cat.interface.ts
export interface Cat {
  name: string;
  age: number;
  breed: string;
}
```

---

3. Updated: Cats Service (With Data)

```
// src/cats/cats.service.ts
import { Injectable } from '@nestjs/common';
import { CreateCatDto } from './dto/create-cat.dto';
import { Cat } from './interfaces/cat.interface'; // Import the interface

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [
    { name: 'Whiskers', age: 5, breed: 'Siamese' },
    { name: 'Smokey', age: 2, breed: 'Maine Coon' },
  ];

  create(catDto: CreateCatDto): Cat { // Use DTO for input
    const newCat = { ...catDto }; // In a real app, you'd map DTO to an
entity
    this.cats.push(newCat);
    return newCat;
  }

  findAll(): Cat[] { // Specify the return type
    return this.cats;
  }
}
```

---

4. Cats Controller

```
// src/cats/cats.controller.ts
import { Controller, Get, Post, Body } from '@nestjs/common';
import { CatsService } from './cats.service';
import { CreateCatDto } from './dto/create-cat.dto';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Post()
  create(@Body() createCatDto: CreateCatDto): Cat { // Added return type
    return this.catsService.create(createCatDto);
  }
```

```
  @Get()
  findAll(): Cat[] { // Added return type
    // This method was already correct!
    return this.catsService.findAll();
  }
}
```

CRUD generator | NestJS - A progressive Node.js framework

---

5. What is a DTO and Why It's Required

DTO: a simple class/object that defines the shape of data sent to / from your API (fields and types).

Why use DTOs

- **Validation & safety**: ensure incoming data has expected fields and types before business logic runs.
- **Separation of concerns**: controllers accept typed DTOs, services operate on validated data.
- **Sanitization**: tools (e.g., Nest's `ValidationPipe`) can remove properties not defined in DTOs (`whitelist`).
- **Self-documentation**: DTOs express your API contract and help generate docs.

---

6. Employee API (NestJS)

src/main.ts

```
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({ whitelist: true, transform: true
}));
  await app.listen(3000);
}
bootstrap();
```

DTOs

```
create-employee.dto.ts

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

export class CreateEmployeeDto {
  @IsNotEmpty()
  @IsString()
```

```
  name: string;

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

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

update-employee.dto.ts

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

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

---

7. Understanding Validation Decorators

Decorator	Purpose
`@IsNotEmpty()`	Ensures the field is present and not empty
`@IsEmail()`	Checks if field is valid email
`@IsString()`	Validates data type
`@IsOptional()`	Skips validation if field is missing

When DTO validation fails under Nest's ValidationPipe, the client receives 400 Bad Request with a readable error message.

Example validation failure response:

```
{
  "statusCode": 400,
  "message": [
    "name should not be empty",
    "email must be an email"
  ],
  "error": "Bad Request"
}
```

---

8. Controller + File Upload

employees.controller.ts

```
import { Controller, Get, Post, Body, Param, Put, Delete, UseInterceptors,
UploadedFile } from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express';
import { diskStorage } from 'multer';
import { extname } from 'path';
```

```
import { CreateEmployeeDto } from './dto/create-employee.dto';
import { UpdateEmployeeDto } from './dto/update-employee.dto';

@Controller('employees')
export class EmployeesController {
  private employees = new Map<string, any>();

  @Post()
  create(@Body() dto: CreateEmployeeDto) {
    const id = Date.now().toString();
    const emp = { id, ...dto };
    this.employees.set(id, emp);
    return emp;
  }

  @Get()
  findAll() {
    return Array.from(this.employees.values());
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    const emp = this.employees.get(id);
    if (!emp) throw new Error('Not found');
    return emp;
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() dto: UpdateEmployeeDto) {
    const emp = this.employees.get(id);
    if (!emp) throw new Error('Not found');
    const updated = { ...emp, ...dto };
    this.employees.set(id, updated);
    return updated;
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    this.employees.delete(id);
    return { status: 'deleted' };
  }

  @Post(':id/photo')
  @UseInterceptors(FileInterceptor('file', {
    storage: diskStorage({
      destination: './uploads',
      filename: (req, file, cb) => cb(null,
`${Date.now()}${extname(file.originalname)}`)
    }),
    limits: { fileSize: 5 * 1024 * 1024 }
  }))
  uploadPhoto(@Param('id') id: string, @UploadedFile() file:
Express.Multer.File) {
    const emp = this.employees.get(id);
    if (!emp) throw new Error('Not found');
    emp.photo = file.filename;
    this.employees.set(id, emp);
    return emp;
  }
```

```
}
```

---

9. Middleware and Static Files in NestJS

Middleware Example

```
logger.middleware.ts

import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log(`[${new Date().toISOString()}] ${req.method}
${req.originalUrl}`);
    next();
  }
}
```

Register in `employees.module.ts`:

```
import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';
import { LoggerMiddleware } from '../common/middleware/logger.middleware';

@Module({...})
export class EmployeesModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(LoggerMiddleware).forRoutes('employees');
  }
}
```

Static Files

```
import { ServeStaticModule } from '@nestjs/serve-static';
import { join } from 'path';

@Module({
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public'),
      exclude: ['/api*'],
    }),
  ],
})
export class AppModule {}
```

---

10. Testing (Bruno + curl)

Bruno

1. Open Bruno → new request.

2. URL: `http://localhost:3000/employees`
3. Choose method (POST/GET/PUT/DELETE).
4. Add headers: `Content-Type: application/json`.
5. Add JSON body (for POST/PUT).

curl

```
# Create employee
curl -X POST http://localhost:3000/employees \
  -H "Content-Type: application/json" \
  -d '{"name":"Bruno Tester","email":"bruno@example.com","role":"HR"}'

# Upload photo
curl -X POST http://localhost:3000/employees/<id>/photo \
  -F "file=@/path/to/photo.jpg"
```

Summary

Concept	Express.js	NestJS
Framework Type	Unopinionated	Opinionated, modular
Language	JavaScript / ESM	TypeScript
Middleware	Function-based	Class-based (NestMiddleware)
Routing	`app.get`, `app.post` etc.	Decorators (`@Get()`, `@Post()`)
Validation	Manual	DTOs + ValidationPipe
Static Files	`express.static()`	ServeStaticModule
File Uploads	`multer`	`@UseInterceptors(FileInterceptor)`

Practice Activities

Activity 1 — Express CRUD Practice

Goal: Build a simple Express CRUD app and test using Bruno or curl.

1. Create `/api/items` routes (use sample above).
2. Add custom middleware for logging.
3. Serve a static `index.html` from `public/`.
4. Test with Bruno or curl.

Activity 2 — NestJS Department API

Goal: Use DTOs, ValidationPipe, and file uploads.

1. Scaffold a Nest project.
2. Implement `departments.controller.ts` and DTOs as above.
3. Test create, get, update, delete, and upload.
4. Observe validation errors for missing fields.

Coverage

- 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**.