upGrad ENTERPRISE

Day 8 — Unit Testing with Jest

Siddhesh Prabhugaonkar
Microsoft Certified Trainer
siddheshpg@azureauthority.in

- **Previous day:**
  - 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)
- Day 7
  - Explain what unit testing is and why it's important
  - Use the **AAA (Arrange-Act-Assert)** pattern to write tests
  - Set up **Jest** for a Node.js / NestJS project
  - Write tests for **pure functions**
  - Test **NestJS controllers, services, and middleware**
  - Mock database calls and external APIs
  - Generate and interpret **code coverage reports**

---

1. Understanding Unit Testing

Unit testing is the process of testing individual units or components of code to ensure they work as expected.

Purpose:

- Catch bugs early
- Enable safe refactoring
- Improve confidence during deployment
- Document expected behavior

Types of Tests:

Type	Description	Example
Unit	Tests single functions or services in isolation	Service method returning list of employees
Integration	Tests interaction between modules	Controller + Service + DB
E2E	Tests full flow (API to DB and back)	API endpoint test via HTTP request

---

2. The AAA Pattern (Arrange – Act – Assert)

The **AAA pattern** helps structure your test cases clearly.

Step	Description	Example (EmployeeService test)
Arrange	Prepare data, mocks, and environment	Mock Sequelize model
Act	Execute the code under test	Call `employeesService.findAll()`
Assert	Verify the result	Expect an array of employees

Visual Representation

```
+------------------------------+
|           ARRANGE            |
| (Setup mocks, inputs, stubs) |
+--------------↓---------------+
|              ACT             |
| (Call the function under test)|
+--------------↓---------------+
|            ASSERT            |
| (Verify the expected result) |
+------------------------------+
```

What is Jest?

Jest is a JavaScript testing framework commonly used with NestJS.
It allows you to write **unit tests**, **mock dependencies**, and **ensure code works as expected**.

---

1 `describe()` – Grouping Tests

What it is:
A function that groups related tests together.

Why we use it:

- Organizes tests into sections
- Helps readability
- Shows structured output in the test report

Syntax:

```
describe('CalculatorService', () => {
  // write tests here
});
```

Think of it as a **container** for tests.

---

2 `it()` – Individual Test Case

What it is:
Defines a **single test**.

Why we use it:

- Describes one behavior
- Should test only ONE thing

Syntax:

```
it('should add two numbers', () => {
  expect(2 + 2).toBe(4);
});
```

`it` is usually written as a sentence:
"It should do something…"

---

3 `test()` – Same as `it()`

`test()` is just an alias of `it()`.
Both are identical.

```
test('should subtract numbers', () => {
  expect(5 - 3).toBe(2);
});
```

You can use either.
Most developers prefer **it()** because it reads more naturally.

---

4 Matchers (`expect()` + matchers)

`expect()`

Starts an assertion — we tell Jest what we expect.

Example:

```
expect(value).toBe(10);
```

Common Matchers:

✔ **`toBe()` – strict equality (===)**

```
expect(3 + 2).toBe(5);
```

✔ **`toEqual()` – deep equality (objects, arrays)**

```
expect({ name: 'John' }).toEqual({ name: 'John' });
```

✔ **`toBeDefined()` / `toBeUndefined()`**

```
expect(value).toBeDefined();
```

✔ **`toBeNull()`**

```
expect(value).toBeNull();
```

✔ **`toBeTruthy()` / `toBeFalsy()`**

```
expect(true).toBeTruthy();
```

✔ **`toContain()`**

Works for arrays and strings:

```
expect([1, 2, 3]).toContain(2);
expect('hello').toContain('he');
```

✔ **`toThrow()` – exceptions**

```
expect(() => service.divide(5, 0)).toThrow();
```

5️⃣ **`beforeEach()`, `afterEach()`, `beforeAll()`, `afterAll()`**

These run automatically before/after tests.

Use them to:

- create mock data
- initialize services
- reset state

Example:

```
let service: CalculatorService;

beforeEach(() => {
  service = new CalculatorService();
});
```

---

6 Simple Example for NestJS Service

```
import { Test, TestingModule } from '@nestjs/testing';
import { CalculatorService } from './calculator.service';

describe('CalculatorService', () => {
  let service: CalculatorService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [CalculatorService],
    }).compile();

    service = module.get<CalculatorService>(CalculatorService);
  });

  it('should return sum of numbers', () => {
    expect(service.add(2, 3)).toBe(5);
  });

  test('should return difference of numbers', () => {
    expect(service.subtract(5, 2)).toBe(3);
  });
});
```

---

7 Test Output Example

When you run:

```
npm run test
```

You get something like:

```
CalculatorService
  ✓ should return sum of numbers
  ✓ should return difference of numbers
```

Example: Using AAA in a Simple Test

```
// employee.service.spec.ts
import { EmployeesService } from './employees.service';

describe('EmployeesService', () => {
  it('should return the correct employee count', () => {
    // Arrange
    const mockEmployees = [{ id: 1 }, { id: 2 }];
    const service = new EmployeesService();
    jest.spyOn(service, 'findAll').mockResolvedValue(mockEmployees as any);

    // Act
    const result = service.findAll();

    // Assert
    expect(result).resolves.toHaveLength(2);
  });
});
```

---

3. Setting up Jest for a Node.js / NestJS Project

Install Jest and Dependencies

```
npm install --save-dev jest @types/jest ts-jest
```

Initialize Jest Config

```
npx ts-jest config:init
```

It creates `jest.config.js`:

```
module.exports = {
  preset: 'ts-jest',
  testEnvironment: 'node',
  coverageDirectory: 'coverage',
  moduleFileExtensions: ['js', 'json', 'ts'],
  rootDir: 'src',
  testRegex: '.*\\.spec\\.ts$',
  transform: {
    '^.+\\.(t|j)s$': 'ts-jest',
  },
};
```

Add Script in package.json

```
"scripts": {
  "test": "jest",
  "test:watch": "jest --watch",
  "test:cov": "jest --coverage"
}
```

---

4. Writing Unit Tests for Pure Functions

Folder Layout for Tests

```
src/
├── modules/
│   ├── employees/
│   │   ├── employees.service.ts
│   │   ├── employees.service.spec.ts
│   │   └── employees.controller.spec.ts
│   ├── users/
│   │   ├── users.service.ts
│   │   ├── users.service.spec.ts
│   │   └── auth.service.spec.ts
├── utils/
│   ├── salary-calculator.util.ts
│   └── salary-calculator.util.spec.ts
└── test/
    └── employees.e2e-spec.ts
```

Example: salary-calculator.util.ts

```
// src/utils/salary-calculator.util.ts
export function calculateAnnualSalary(monthlySalary: number): number {
  if (monthlySalary < 0) throw new Error('Invalid salary');
  return monthlySalary * 12;
}
```

Test: salary-calculator.util.spec.ts

```
import { calculateAnnualSalary } from './salary-calculator.util';

describe('calculateAnnualSalary', () => {
  it('should calculate annual salary correctly', () => {
    // Arrange
    const monthly = 50000;

    // Act
    const annual = calculateAnnualSalary(monthly);

    // Assert
    expect(annual).toBe(600000);
  });

  it('should throw error for negative salary', () => {
    expect(() => calculateAnnualSalary(-100)).toThrow('Invalid salary');
  });
});
```

Demonstrates AAA pattern clearly for simple logic.

---

5. Testing NestJS Controllers and Services

EmployeeService Test (Mocking DB calls)

```
// src/modules/employees/employees.service.spec.ts
import { EmployeesService } from './employees.service';
import { Employee } from '../../models/employee.model';

describe('EmployeesService', () => {
  let service: EmployeesService;
  let mockEmployeeModel: Partial<typeof Employee>;

  beforeEach(() => {
    mockEmployeeModel = {
      findAll: jest.fn().mockResolvedValue([{ id: 1, name: 'Ravi' }]),
      findByPk: jest.fn().mockResolvedValue({ id: 1, name: 'Ravi' }),
    };
    service = new EmployeesService(mockEmployeeModel as any);
  });

  it('should return all employees', async () => {
    const result = await service.findAll();
    expect(result).toHaveLength(1);
    expect(mockEmployeeModel.findAll).toHaveBeenCalled();
  });

  it('should return one employee by id', async () => {
    const result = await service.findOne(1);
    expect(result.name).toBe('Ravi');
  });
});
```

Testing Controller (Mocking Service)

```
// src/modules/employees/employees.controller.spec.ts
import { EmployeesController } from './employees.controller';
import { EmployeesService } from './employees.service';

describe('EmployeesController', () => {
  let controller: EmployeesController;
  let service: EmployeesService;

  beforeEach(() => {
    service = new EmployeesService({} as any);
    controller = new EmployeesController(service);
  });

  it('should return list of employees', async () => {
    jest.spyOn(service, 'findAll').mockResolvedValue([{ id: 1, name: 'Ravi'
}] as any);
    const result = await controller.findAll();
    expect(result).toEqual([{ id: 1, name: 'Ravi' }]);
  });
});
```

---

6. Mocking Database Calls & External APIs

In real projects, direct DB or API calls should not be made during unit tests.

Why mock?

- Avoid hitting real DBs or APIs
- Ensure deterministic and fast tests
- Test logic independently

Mock Example

```
jest.spyOn(employeeModel, 'findAll').mockResolvedValue([{ id: 1, name:
'Mocked' }]);
```

Mocking External API Call

```
import axios from 'axios';
jest.mock('axios');

it('should call external API', async () => {
  (axios.get as jest.Mock).mockResolvedValue({ data: { status: 'ok' } });
  const response = await axios.get('http://api.example.com');
  expect(response.data.status).toBe('ok');
});
```

---

7. Testing NestJS Routes and Middleware

Testing Routes with `@nestjs/testing`

employees.e2e-spec.ts

```
import { Test, TestingModule } from '@nestjs/testing';
import { INestApplication } from '@nestjs/common';
import * as request from 'supertest';
import { AppModule } from '../../app.module';

describe('Employees API (e2e)', () => {
  let app: INestApplication;

  beforeAll(async () => {
    const moduleFixture: TestingModule = await Test.createTestingModule({
      imports: [AppModule],
    }).compile();

    app = moduleFixture.createNestApplication();
    await app.init();
  });

  it('/employees (GET)', async () => {
    const res = await request(app.getHttpServer()).get('/employees');
    expect(res.status).toBe(200);
  });

  afterAll(async () => {
    await app.close();
  });
```

```
});
```

Note: Uses `supertest` for HTTP assertions.
Install:

```
npm install --save-dev supertest @types/supertest
```

Middleware Testing Example

```
import { LoggingMiddleware } from './logging.middleware';

describe('LoggingMiddleware', () => {
  it('should call next()', () => {
    const req = { url: '/test' } as any;
    const res = {} as any;
    const next = jest.fn();

    const middleware = new LoggingMiddleware();
    middleware.use(req, res, next);

    expect(next).toHaveBeenCalled();
  });
});
```

How It Differs for Express.js

Aspect	NestJS	Express.js
Test setup	Use `@nestjs/testing` utilities	Use `supertest` directly
Dependency injection	Can mock providers easily	Manual mock/stubs
Middleware testing	Class-based `use()` tested with Nest app	Function-based middleware tested directly
Config	Jest config included in Nest CLI	Must set up manually

However, Jest works identically in both — Arrange, Act, Assert pattern applies universally.

---

8. Code Coverage Reports

Run with:

```
npm run test:cov
```

Output sample:

```
------------------|---------|----------|---------|---------
File              | % Stmts | % Branch | % Funcs | % Lines
```

```
------------------|---------|----------|---------|---------
All files         |   92.85 |     90.0 |   91.66 |   92.50
 employees.service.ts |   95.00 |   90.00 |   92.00 |   94.00
------------------|---------|----------|---------|---------
```

Tips:

- Focus on testing core logic, not framework boilerplate.
- Aim for **>80% coverage**, but quality > quantity.
- Use `.coveragerc` or Jest config to exclude migration/config files.

---

9. Tips & Best Practices

- Always isolate business logic from database logic for easier testing.
- Use **mocks** or **stubs** for DB/API calls.
- Don't test libraries — test your logic.
- Follow the **AAA pattern** consistently.
- Run tests as part of CI pipeline.
- Keep test names descriptive and behavior-driven.
  → "should return 404 when employee not found" is better than "test1".

---

10. Summary

Concept	Key Takeaway
AAA Pattern	Arrange, Act, Assert ensures clear test logic
Jest	Simple, fast testing framework for Node.js & NestJS
Mocking	Replaces DB/API dependencies for isolated tests
Coverage	Ensures critical parts of code are verified
NestJS Testing	Uses `@nestjs/testing` and `supertest` for API-level checks
Express vs NestJS	Jest works similarly; NestJS offers structured testing utilities

---

Employee service unit testing by mocking repository (TypeORM)

Employee service Test File Structure

Imports

```
import { Test, TestingModule } from '@nestjs/testing';
import { EmployeesService } from './employees.service';
import { Employee } from './entities/employee.entity';
import { getRepositoryToken } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { ConflictException, NotFoundException } from '@nestjs/common';
import { CreateEmployeeDto } from './dto/create-employee.dto';
import { UpdateEmployeeDto } from './dto/update-employee.dto';
```

Key Points:

- `Test` and `TestingModule` - NestJS testing utilities for creating test modules
- `getRepositoryToken` - Critical for mocking TypeORM repositories
- Import the service, entity, DTOs, and exceptions you'll be testing

---

Setting Up Mocks

Creating a Mock Repository

```
let service: EmployeesService;
let mockRepository: Partial<Record<keyof Repository<Employee>, jest.Mock>>;

beforeEach(async () => {
  mockRepository = {
    find: jest.fn(),
    findOneBy: jest.fn(),
    insert: jest.fn(),
    update: jest.fn(),
    delete: jest.fn(),
  };
```

```
  const module: TestingModule = await Test.createTestingModule({
    providers: [
      EmployeesService,
      {
        provide: getRepositoryToken(Employee),
        useValue: mockRepository,
      },
    ],
  }).compile();

  service = module.get<EmployeesService>(EmployeesService);
});
```

Key Concepts:

1. **Mock Repository Type**: `Partial<Record<keyof Repository<Employee>, jest.Mock>>`
    - Creates a type-safe mock with Jest mock functions
    - Only includes methods we actually use in the service
2. **`getRepositoryToken(Employee)`**:
    - Returns the injection token that NestJS uses for the Employee repository
    - This is what tells NestJS to inject our mock instead of a real repository
3. **`useValue: mockRepository`**:
    - Provides our mock as the repository implementation
    - All repository calls in the service will use this mock
4. **Why Mock?**
    - Unit tests should test business logic, not database operations
    - Mocks are fast and don't require database setup
    - Tests are isolated and predictable

Cleanup

```
afterEach(() => {
  jest.clearAllMocks();
});
```

- Clears all mock call history between tests
- Ensures tests don't interfere with each other

Testing CRUD Operations

1. Read Operations - `findAll()`

```
describe('findAll', () => {
  it('should return all employees', async () => {
    const mockEmployees: Employee[] = [
      { id: 1, name: 'Ravi', email: 'ravi@test.com', salary: 50000,
        dateOfBirth: new Date('1990-01-01'), mobileNumber: 1234567890,
departmentId: 1 },
      { id: 2, name: 'Priya', email: 'priya@test.com', salary: 60000,
        dateOfBirth: new Date('1992-02-02'), mobileNumber: 9876543210,
departmentId: 2 },
    ];
    mockRepository.find.mockResolvedValue(mockEmployees);

    const result = await service.findAll();

    expect(result).toEqual(mockEmployees);
    expect(result).toHaveLength(2);
    expect(mockRepository.find).toHaveBeenCalled();
  });
});
```

What's Happening:

1. **Setup**: Define mock data that represents what the database would return
2. **Mock Configuration**: `mockResolvedValue()` makes the mock return our test data
3. **Execution**: Call the service method
4. **Assertions**:
    - Verify the result matches expected data
    - Check the result length
    - Confirm the repository method was called

2. Read Operations - `findOne(id)`

```
describe('findOne', () => {
  it('should return one employee by id', async () => {
    const mockEmployee: Employee = {
      id: 1, name: 'Ravi', email: 'ravi@test.com', salary: 50000,
```

```
      dateOfBirth: new Date('1990-01-01'), mobileNumber: 1234567890,
departmentId: 1,
    };
    mockRepository.findOneBy.mockResolvedValue(mockEmployee);

    const result = await service.findOne(1);

    expect(result).toEqual(mockEmployee);
    expect(result.name).toBe('Ravi');
    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 1 });
  });

  it('should throw NotFoundException when employee not found', async () => {
    mockRepository.findOneBy.mockResolvedValue(null);

    await expect(service.findOne(999)).rejects.toThrow(NotFoundException);
    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 999 });
  });
});
```

Key Testing Patterns:

1. **Happy Path Testing**: Test when data is found
    - `toHaveBeenCalledWith({ id: 1 })` verifies correct parameters were passed
2. **Error Path Testing**: Test when data doesn't exist
    - Mock returns `null` to simulate not found
    - `rejects.toThrow(NotFoundException)` verifies proper exception handling

3. Create Operations - `create(dto)`

```
describe('create', () => {
  const createDto: CreateEmployeeDto = {
    name: 'New Employee',
    email: 'new@test.com',
    salary: 55000,
    dateOfBirth: '1995-05-05',
    mobileNumber: 5555555555,
    departmentId: 1,
  };
```

```
  it('should create a new employee', async () => {
    mockRepository.findOneBy.mockResolvedValue(null);
    mockRepository.insert.mockResolvedValue({ generatedMaps: [], identifiers:
[], raw: [] });

    const result = await service.create(createDto);

    expect(result.name).toBe(createDto.name);
    expect(result.email).toBe(createDto.email);
    expect(result.salary).toBe(createDto.salary);
    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ email:
createDto.email });
    expect(mockRepository.insert).toHaveBeenCalled();
  });

  it('should throw ConflictException when email already exists', async () => {
    const existingEmployee: Employee = {
      id: 1, name: 'Existing', email: 'new@test.com', salary: 50000,
      dateOfBirth: new Date('1990-01-01'), mobileNumber: 1234567890,
departmentId: 1,
    };
    mockRepository.findOneBy.mockResolvedValue(existingEmployee);

    await
expect(service.create(createDto)).rejects.toThrow(ConflictException);
    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ email:
createDto.email });
    expect(mockRepository.insert).not.toHaveBeenCalled();
  });
});
```

Important Concepts:

1. **Testing Business Logic**:
    - Service checks for duplicate email before inserting
    - First test: no duplicate exists (null) → successful creation
    - Second test: duplicate exists → ConflictException thrown
2. **Multiple Mock Calls**:
    - `findOneBy` is called first (duplicate check)
    - `insert` is called only if no duplicate
3. **Negative Assertions**:

- `.not.toHaveBeenCalled()` verifies method wasn't called when it shouldn't be

4. Update Operations - `update(id, dto)`

```
describe('update', () => {
  const updateDto: UpdateEmployeeDto = {
    name: 'Updated Name',
    salary: 65000,
  };

  it('should update an employee', async () => {
    const existingEmployee: Employee = {
      id: 1, name: 'Old Name', email: 'test@test.com', salary: 50000,
      dateOfBirth: new Date('1990-01-01'), mobileNumber: 1234567890,
departmentId: 1,
    };
    mockRepository.findOneBy.mockResolvedValue(existingEmployee);
    mockRepository.update.mockResolvedValue({ affected: 1, generatedMaps: [],
raw: [] });

    const result = await service.update(1, updateDto);

    expect(result.name).toBe(updateDto.name);
    expect(result.salary).toBe(updateDto.salary);
    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 1 });
    expect(mockRepository.update).toHaveBeenCalledWith(
      { id: 1 },
      expect.objectContaining(updateDto)
    );
  });

  it('should throw NotFoundException when employee not found', async () => {
    mockRepository.findOneBy.mockResolvedValue(null);

    await expect(service.update(999,
updateDto)).rejects.toThrow(NotFoundException);
    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 999 });
    expect(mockRepository.update).not.toHaveBeenCalled();
  });
```

```
});
```

Testing Strategy:

1. **Partial Updates**: `UpdateEmployeeDto` uses `PartialType`
    - Only specified fields need to be updated
    - `expect.objectContaining(updateDto)` checks for subset match
2. **Object Mutation**: Service uses `Object.assign()` to merge changes
    - Test verifies the returned object has updated values

5. Delete Operations - `remove(id)`

```
describe('remove', () => {
  it('should delete an employee', async () => {
    const existingEmployee: Employee = {
      id: 1, name: 'To Delete', email: 'delete@test.com', salary: 50000,
      dateOfBirth: new Date('1990-01-01'), mobileNumber: 1234567890,
departmentId: 1,
    };
    mockRepository.findOneBy.mockResolvedValue(existingEmployee);
    mockRepository.delete.mockResolvedValue({ affected: 1, raw: [] });

    await service.remove(1);

    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 1 });
    expect(mockRepository.delete).toHaveBeenCalledWith(1);
  });

  it('should throw NotFoundException when employee not found', async () => {
    mockRepository.findOneBy.mockResolvedValue(null);

    await expect(service.remove(999)).rejects.toThrow(NotFoundException);
    expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 999 });
    expect(mockRepository.delete).not.toHaveBeenCalled();
  });
});
```

Exception Handling Tests

Why Test Exceptions?

Exception handling is critical business logic that needs testing:

1. **NotFoundException**: Resource doesn't exist (404)
2. **ConflictException**: Duplicate/constraint violation (409)

Pattern for Testing Exceptions

```
await expect(service.methodName(params)).rejects.toThrow(ExceptionType);
```

Components:

- `expect()` - Jest assertion
- `await` - Wait for async operation
- `.rejects` - Indicates we expect a rejection/exception
- `.toThrow(ExceptionType)` - Specific exception class to match

Key Takeaways

1. AAA Pattern (Arrange-Act-Assert)

```
// Arrange: Setup mock data and behavior
mockRepository.findOneBy.mockResolvedValue(mockEmployee);

// Act: Execute the method under test
const result = await service.findOne(1);

// Assert: Verify results and behaviors
expect(result).toEqual(mockEmployee);
expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 1 });
```

2. Test Both Happy and Error Paths

- Happy path: Everything works as expected
- Error path: Handle invalid input, missing data, conflicts

3. Mock Return Values Match Real Data

- Mock data should have the same structure as entity
- Include all required fields
- Use realistic test data

4. Verify Method Calls

```
expect(mockRepository.find).toHaveBeenCalled();
expect(mockRepository.findOneBy).toHaveBeenCalledWith({ id: 1 });
expect(mockRepository.insert).not.toHaveBeenCalled();
```

5. Isolation is Key

- Each test is independent
- `beforeEach` sets up fresh state
- `afterEach` cleans up mocks
- Tests don't share state

6. Mock Configuration Methods

```
jest.fn()                          // Create a mock function
.mockResolvedValue(data)           // Async success with data
.mockRejectedValue(error)          // Async failure with error
.mockReturnValue(data)             // Sync return
.mockImplementation((args) => {})  // Custom implementation
```

7. Common Jest Matchers

```
expect(value).toBe(expected)              // Strict equality
expect(value).toEqual(expected)           // Deep equality
expect(value).toHaveLength(n)             // Array/string length
expect(fn).toHaveBeenCalled()             // Function was called
expect(fn).toHaveBeenCalledWith(args)     // Called with specific args
expect(fn).not.toHaveBeenCalled()         // Function wasn't called
expect(promise).rejects.toThrow(Error)    // Async exception
expect(obj).toMatchObject(partial)        // Partial object match
expect(obj).objectContaining(partial)     // Subset matcher
```

Running the Tests

```
# Run all tests
npm test

# Run tests in watch mode
npm test -- --watch

# Run tests with coverage
npm test -- --coverage

# Run specific test file
npm test employees.service.spec.ts
```

Best Practices Summary

☑ **DO:**

- Mock external dependencies (databases, APIs)
- Test one unit of functionality at a time
- Test both success and failure scenarios
- Use descriptive test names
- Keep tests simple and readable
- Clean up after each test

❌ **DON'T:**

- Connect to real databases in unit tests
- Test multiple things in one test
- Make tests depend on each other
- Use hard-coded IDs from databases
- Skip error case testing

Additional Resources

- NestJS Testing Documentation
- Jest Documentation
- TypeORM Repository API

upGrad ENTERPRISE

Controller Unit Testing - Mocking Services

Controllers handle HTTP requests and delegate business logic to services, so our tests focus on request/response handling.

Controller vs Service Testing

What Controllers Do

- Handle HTTP requests (GET, POST, PUT/PATCH, DELETE)
- Extract parameters from routes, query strings, and request bodies
- Call service methods with proper parameters
- Return responses to clients
- Apply guards, interceptors, and pipes

What We Test in Controllers

☑ Correct service methods are called with correct parameters

☑ Controller returns what the service returns

☑ Exceptions from service are properly propagated

☑ Route parameters are correctly parsed and passed

❌ We DON'T test business logic (that's in the service)

❌ We DON'T test database operations (that's in the service)

Test File Structure

Imports

```
import { Test, TestingModule } from '@nestjs/testing';
import { EmployeesController } from './employees.controller';
import { EmployeesService } from './employees.service';
```

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

Key Points:

- Import both the controller and the service (to mock it)
- Import DTOs for request body types
- Import entity for response types
- Import exceptions that might be thrown

Setting Up Service Mocks

Creating a Mock Service

```
let controller: EmployeesController;
let mockEmployeesService: Partial<Record<keyof EmployeesService, jest.Mock>>;

beforeEach(async () => {
  mockEmployeesService = {
    create: jest.fn(),
    findAll: jest.fn(),
    findOne: jest.fn(),
    update: jest.fn(),
    remove: jest.fn(),
  };

  const module: TestingModule = await Test.createTestingModule({
    controllers: [EmployeesController],
    providers: [
      {
        provide: EmployeesService,
        useValue: mockEmployeesService,
      },
    ],
  }).compile();

  controller = module.get<EmployeesController>(EmployeesController);
```

```
});
```

Key Concepts:

1. **Mock Service Type**: `Partial<Record<keyof EmployeesService, jest.Mock>>`
    - Type-safe mock containing all service methods
    - Each method is a Jest mock function
2. **Service as Provider**:

```
providers: [
  {
    provide: EmployeesService,
    useValue: mockEmployeesService,
  },
]
```

    - Tells NestJS to inject our mock instead of real service
    - Controller receives the mock in its constructor
9. **Why Mock the Service?**
    - Controller tests should be fast and isolated
    - Business logic testing belongs in service tests
    - We only verify controller delegates correctly

Cleanup

```
afterEach(() => {
  jest.clearAllMocks();
});
```

Testing HTTP Endpoints

1. POST Request - Create Employee

```
describe('create', () => {
  const createDto: CreateEmployeeDto = {
    name: 'John Doe',
    email: 'john@test.com',
    salary: 50000,
```

```
    dateOfBirth: '1990-01-15',
    mobileNumber: 1234567890,
    departmentId: 1,
  };

  const createdEmployee: Employee = {
    id: 1,
    name: 'John Doe',
    email: 'john@test.com',
    salary: 50000,
    dateOfBirth: new Date('1990-01-15'),
    mobileNumber: 1234567890,
    departmentId: 1,
  };

  it('should create a new employee', async () => {
    mockEmployeesService.create.mockResolvedValue(createdEmployee);

    const result = await controller.create(createDto);

    expect(result).toEqual(createdEmployee);
    expect(mockEmployeesService.create).toHaveBeenCalledWith(createDto);
    expect(mockEmployeesService.create).toHaveBeenCalledTimes(1);
  });

  it('should throw ConflictException when email already exists', async () => {
    mockEmployeesService.create.mockRejectedValue(new ConflictException());

    await
expect(controller.create(createDto)).rejects.toThrow(ConflictException);
    expect(mockEmployeesService.create).toHaveBeenCalledWith(createDto);
  });
});
```

What We're Testing:

1. **Request Body Handling** (@Body() decorator)
    - DTO is passed correctly to service
    - Controller receives and forwards the DTO
2. **Success Response**
    - Controller returns what service returns

- No transformation or modification

3. **Error Handling**
   - Exceptions from service propagate through controller
   - `mockRejectedValue()` simulates service throwing exception

Controller Code Being Tested:

```
@Post()
async create(@Body() createEmployeeDto: CreateEmployeeDto) {
  return await this.employeesService.create(createEmployeeDto);
}
```

2. GET Request - Retrieve All Employees

```
describe('findAll', () => {
  it('should return an array of employees', async () => {
    const employees: Employee[] = [
      {
        id: 1,
        name: 'John Doe',
        email: 'john@test.com',
        salary: 50000,
        dateOfBirth: new Date('1990-01-15'),
        mobileNumber: 1234567890,
        departmentId: 1,
      },
      {
        id: 2,
        name: 'Jane Smith',
        email: 'jane@test.com',
        salary: 60000,
        dateOfBirth: new Date('1992-05-20'),
        mobileNumber: 9876543210,
        departmentId: 2,
      },
    ];
    mockEmployeesService.findAll.mockResolvedValue(employees);

    const result = await controller.findAll();
```

```
    expect(result).toEqual(employees);
    expect(result).toHaveLength(2);
    expect(mockEmployeesService.findAll).toHaveBeenCalled();
    expect(mockEmployeesService.findAll).toHaveBeenCalledTimes(1);
  });

  it('should return an empty array when no employees exist', async () => {
    mockEmployeesService.findAll.mockResolvedValue([]);

    const result = await controller.findAll();

    expect(result).toEqual([]);
    expect(result).toHaveLength(0);
    expect(mockEmployeesService.findAll).toHaveBeenCalled();
  });
});
```

Testing Strategy:

1. **Happy Path**: Service returns data
    - Verify array is returned unchanged
    - Check service was called exactly once
2. **Edge Case**: No data exists
    - Empty array is valid response
    - Not an error condition

Controller Code Being Tested:

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

3. GET Request with Route Parameter - Find One Employee

```
describe('findOne', () => {
  const employee: Employee = {
    id: 1,
    name: 'John Doe',
```

```
    email: 'john@test.com',
    salary: 50000,
    dateOfBirth: new Date('1990-01-15'),
    mobileNumber: 1234567890,
    departmentId: 1,
  };

  it('should return a single employee by id', async () => {
    mockEmployeesService.findOne.mockResolvedValue(employee);

    const result = await controller.findOne(1);

    expect(result).toEqual(employee);
    expect(mockEmployeesService.findOne).toHaveBeenCalledWith(1);
    expect(mockEmployeesService.findOne).toHaveBeenCalledTimes(1);
  });

  it('should throw NotFoundException when employee not found', async () => {
    mockEmployeesService.findOne.mockRejectedValue(new NotFoundException());

    await expect(controller.findOne(999)).rejects.toThrow(NotFoundException);
    expect(mockEmployeesService.findOne).toHaveBeenCalledWith(999);
  });
});
```

Important Details:

1. **Route Parameter Extraction** (`@Param('id')`)
    - Controller receives `id` from URL path
    - `+id` converts string to number in actual controller
    - Test calls with number directly
2. **Not Found Scenario**
    - Service throws `NotFoundException`
    - Controller doesn't catch it (propagates to NestJS)
    - Client receives 404 response

Controller Code Being Tested:

```
@Get(':id')
async findOne(@Param('id') id: number) {
  return await this.employeesService.findOne(+id);
```

```
}
```

4. PATCH Request - Update Employee

```
describe('update', () => {
  const updateDto: UpdateEmployeeDto = {
    name: 'John Updated',
    salary: 55000,
  };

  const updatedEmployee: Employee = {
    id: 1,
    name: 'John Updated',
    email: 'john@test.com',
    salary: 55000,
    dateOfBirth: new Date('1990-01-15'),
    mobileNumber: 1234567890,
    departmentId: 1,
  };

  it('should update an employee', async () => {
    mockEmployeesService.update.mockResolvedValue(updatedEmployee);

    const result = await controller.update(1, updateDto);

    expect(result).toEqual(updatedEmployee);
    expect(result.name).toBe('John Updated');
    expect(result.salary).toBe(55000);
    expect(mockEmployeesService.update).toHaveBeenCalledWith(1, updateDto);
    expect(mockEmployeesService.update).toHaveBeenCalledTimes(1);
  });

  it('should throw NotFoundException when employee not found', async () => {
    mockEmployeesService.update.mockRejectedValue(new NotFoundException());

    await expect(controller.update(999,
updateDto)).rejects.toThrow(NotFoundException);
    expect(mockEmployeesService.update).toHaveBeenCalledWith(999, updateDto);
  });
```

```
});
```

Testing Focus:

1. **Multiple Parameters**
    - Route parameter: `id`
    - Request body: `updateDto`
    - Both passed correctly to service
2. **Partial Updates**
    - `UpdateEmployeeDto` can have subset of fields
    - Only specified fields are updated
    - Service handles the logic

Controller Code Being Tested:

```
@Patch(':id')
async update(@Param('id') id: number, @Body() updateEmployeeDto:
UpdateEmployeeDto) {
  return await this.employeesService.update(+id, updateEmployeeDto);
}
```

5. DELETE Request - Remove Employee

```
describe('remove', () => {
  it('should delete an employee', async () => {
    mockEmployeesService.remove.mockResolvedValue(undefined);

    const result = await controller.remove(1);

    expect(result).toBeUndefined();
    expect(mockEmployeesService.remove).toHaveBeenCalledWith(1);
    expect(mockEmployeesService.remove).toHaveBeenCalledTimes(1);
  });

  it('should throw NotFoundException when employee not found', async () => {
    mockEmployeesService.remove.mockRejectedValue(new NotFoundException());

    await expect(controller.remove(999)).rejects.toThrow(NotFoundException);
    expect(mockEmployeesService.remove).toHaveBeenCalledWith(999);
  });
```

```
});
```

Key Points:

1. **No Return Value**
    - Delete operations typically return `void` or `undefined`
    - HTTP 200 OK with no body (or 204 No Content)
2. **Error Handling**
    - Cannot delete non-existent resource
    - Service throws exception, controller propagates

Controller Code Being Tested:

```
@Delete(':id')
async remove(@Param('id') id: number) {
  return await this.employeesService.remove(+id);
}
```

Parameter Handling

Types of Parameters in NestJS Controllers

1. **Route Parameters** (`@Param()`)

```
2. @Get(':id')
3. findOne(@Param('id') id: number) // URL: /employees/123
```

4. **Request Body** (`@Body()`)

```
5. @Post()
6. create(@Body() dto: CreateEmployeeDto) // JSON in request body
```

7. **Query Parameters** (`@Query()`)

```
8. @Get()
9. findAll(@Query('dept') deptId: number) // URL: /employees?dept=5
```

Testing Parameter Extraction

Route Parameters:

```
// Controller receives from URL path
await controller.findOne(1);
expect(mockService.findOne).toHaveBeenCalledWith(1);
```

Request Body:

```
// Controller receives from HTTP body
const dto = { name: 'John', email: 'john@test.com', ... };
await controller.create(dto);
expect(mockService.create).toHaveBeenCalledWith(dto);
```

Multiple Parameters:

```
// Combination of route param and body
await controller.update(1, updateDto);
expect(mockService.update).toHaveBeenCalledWith(1, updateDto);
```

Key Takeaways

1. Controller Tests Are About Delegation

```
// We test that controller...
☑ Calls the right service method
☑ Passes the right parameters
☑ Returns what service returns
☑ Doesn't catch exceptions (lets them propagate)

// We DON'T test...
❌ Business logic (service's job)
❌ Database operations (service's job)
❌ Data validation (pipes' job, tested separately)
```

2. Mock Service, Not Repository

```
// ☑ Correct: Mock the direct dependency
providers: [
  {
    provide: EmployeesService,
    useValue: mockEmployeesService,
  },
]

// ✗ Wrong: Don't mock repository in controller tests
// Repository is service's dependency, not controller's
```

3. Test Exception Propagation

```
// Service throws exception
mockEmployeesService.findOne.mockRejectedValue(new NotFoundException());

// Controller doesn't catch it
await expect(controller.findOne(999)).rejects.toThrow(NotFoundException);

// NestJS handles it and returns proper HTTP status
```

4. Verify Call Parameters

```
expect(mockService.method).toHaveBeenCalledWith(expectedParams);
expect(mockService.method).toHaveBeenCalledTimes(1);
```

5. AAA Pattern in Controller Tests

```
it('should return employee by id', async () => {
  // Arrange: Setup mock service response
  mockEmployeesService.findOne.mockResolvedValue(employee);

  // Act: Call controller method
  const result = await controller.findOne(1);
```

```
  // Assert: Verify behavior
  expect(result).toEqual(employee);
  expect(mockEmployeesService.findOne).toHaveBeenCalledWith(1);
});
```

6. Testing HTTP Methods

HTTP Method	Controller Decorator	Typical Return
GET	`@Get()`	Entity or array
POST	`@Post()`	Created entity
PATCH/PUT	`@Patch()` / `@Put()`	Updated entity
DELETE	`@Delete()`	void or deleted entity

7. Common Assertions for Controllers

```
// Return value matches
expect(result).toEqual(expectedData);

// Service was called correctly
expect(mockService.method).toHaveBeenCalled();
expect(mockService.method).toHaveBeenCalledWith(params);
expect(mockService.method).toHaveBeenCalledTimes(1);

// Exceptions propagate
await expect(controller.method()).rejects.toThrow(ExceptionType);
```

Comparison: Controller vs Service Tests

Controller Tests Focus On:

```
☑ HTTP request handling
☑ Parameter extraction (@Param, @Body, @Query)
```

- ☑ Calling correct service method
- ☑ Returning service response
- ☑ Exception propagation
- ☑ Guards and interceptors (integration level)

Service Tests Focus On:

- ☑ Business logic
- ☑ Data validation
- ☑ Database operations (mocked)
- ☑ Exception throwing
- ☑ Data transformation
- ☑ Complex calculations

Testing Guards (Advanced)

Your controller has commented-out guards:

```
@Post()
//@UseGuards(OktaAuthGuard, AdminGuard)
async create(@Body() createEmployeeDto: CreateEmployeeDto) {
```

How to Test with Guards

```
it('should be protected by guards', () => {
  const guards = Reflect.getMetadata('__guards__', controller.create);
  expect(guards).toContain(OktaAuthGuard);
  expect(guards).toContain(AdminGuard);
});
```

Or use integration tests:

```
// In e2e test
it('should return 401 without auth token', () => {
  return request(app.getHttpServer())
```

```
    .post('/employees')
    .send(createDto)
    .expect(401);
});
```

Best Practices Summary

☑ DO:

- Mock service dependencies
- Test parameter passing
- Test both success and error cases
- Verify service methods are called correctly
- Keep tests simple and focused
- Test one endpoint per describe block

❌ DON'T:

- Test business logic in controller tests
- Mock the repository in controller tests
- Test database operations
- Test validation (pipes handle this)
- Make controller tests complex
- Test HTTP framework behavior (NestJS handles this)

Running the Tests

```
# Run all tests
npm test

# Run only controller tests
npm test employees.controller.spec.ts

# Run tests in watch mode
npm test -- --watch

# Run with coverage
```

```
npm test -- --coverage
```

Common Patterns

Testing Async Controllers

```
// Controllers are usually async
it('should return data', async () => {
  mockService.method.mockResolvedValue(data);
  const result = await controller.method();
  expect(result).toEqual(data);
});
```

Testing Empty Results

```
it('should return empty array', async () => {
  mockService.findAll.mockResolvedValue([]);
  const result = await controller.findAll();
  expect(result).toEqual([]);
  expect(result).toHaveLength(0);
});
```

Testing Type Conversion

```
// Controller: +id converts string to number
@Get(':id')
findOne(@Param('id') id: number) {
  return this.service.findOne(+id);
}

// Test: Pass number directly
it('should convert id to number', async () => {
  await controller.findOne(1);
  expect(mockService.findOne).toHaveBeenCalledWith(1);
```

```
});
```

Integration vs Unit Tests

Unit Tests (What We're Doing)

- Mock all dependencies
- Test controller in isolation
- Fast execution
- Focus on logic flow

Integration Tests (E2E)

- Use real HTTP requests
- Test entire request pipeline
- Include guards, pipes, interceptors
- Use test database

```
// e2e test example
describe('/employees (e2e)', () => {
  it('POST /employees should create employee', () => {
    return request(app.getHttpServer())
      .post('/employees')
      .send(createDto)
      .expect(201)
      .expect((res) => {
        expect(res.body.name).toBe(createDto.name);
      });
  });
});
```

Additional Resources

- NestJS Controllers Documentation
- NestJS Testing Documentation
- Jest Mock Functions
- NestJS Guards Testing

upGrad ENTERPRISE

NestJS Guard Testing

This guide explains how to unit test NestJS guards for **authentication** and **authorization**.

Why Test Guards Separately?

☑ **Single Responsibility**: Each test focuses on one security concern
☑ **Faster Execution**: No need to set up entire request pipeline
☑ **Clear Failures**: Easy to identify security vs. business logic issues
☑ **Reusability**: Guards tested once, used across multiple controllers

---

Understanding Guards

A guard implements the `CanActivate` interface and returns `true` (allow) or throws an exception (deny).

Guard Responsibilities:

- **Authentication**: Verify user identity (OktaAuthGuard)
- **Authorization**: Verify user permissions (AdminGuard)

Testing Strategy

Test Type	What to Test	Mock Guards?
Unit - Guards	Authorization logic	❌ No
Unit - Controllers	Business logic	☑ Yes
E2E	Complete flow	❌ No

OktaAuthGuard Testing

File: `okta-auth.guard.spec.ts`

Test Setup with ConfigService

```
describe('OktaAuthGuard', () => {
  let guard: OktaAuthGuard;
  let mockConfigService: jest.Mocked<ConfigService>;

  beforeEach(() => {
    mockConfigService = {
      get: jest.fn((key: string, defaultValue?: any) => {
        const config: Record<string, string> = {
          'OKTA_ISSUER': 'https://dev-12345.okta.com/oauth2/default',
          'OKTA_CLIENT_ID': 'test-client-id',
          'OKTA_AUDIENCE': 'api://default',
        };
        return config[key] || defaultValue;
      }),
    } as any;

    guard = new OktaAuthGuard(mockConfigService);
  });
```

Key points:

- Mock `ConfigService` to provide Okta configuration
- Guard needs issuer, client ID, and audience for JWT verification

Mock ExecutionContext

```
const createMockExecutionContext = (headers: Record<string, string>):
ExecutionContext => {
  return {
    switchToHttp: jest.fn().mockReturnValue({
      getRequest: jest.fn().mockReturnValue({
        headers,
      }),
      getResponse: jest.fn(),
      getNext: jest.fn(),
    }),
    getClass: jest.fn(),
    getHandler: jest.fn(),
```

```
    getArgs: jest.fn(),
    getArgByIndex: jest.fn(),
    switchToRpc: jest.fn(),
    switchToWs: jest.fn(),
    getType: jest.fn(),
  } as unknown as ExecutionContext;
};
```

Why mock ExecutionContext?

- Guards receive `ExecutionContext`, not raw request
- `switchToHttp()` accesses HTTP-specific request
- `getRequest()` returns mock request with headers

Testing Valid Token

```
it('should return true for valid Bearer token', async () => {
  const mockContext = createMockExecutionContext({
    authorization: 'Bearer valid-okta-token-12345',
  });

  const result = await guard.canActivate(mockContext);

  expect(result).toBe(true);
});
```

Testing Missing Authorization

```
it('should throw UnauthorizedException when authorization header is missing',
async () => {
  const mockContext = createMockExecutionContext({});

  await expect(guard.canActivate(mockContext)).rejects.toThrow(
    UnauthorizedException,
  );
});
```

Testing Edge Cases

```
it('should throw UnauthorizedException for invalid token format', async () =>
{
  const mockContext = createMockExecutionContext({
    authorization: 'invalid-token-format', // No "Bearer" prefix
  });

  await expect(guard.canActivate(mockContext)).rejects.toThrow(
    UnauthorizedException,
  );
});

it('should throw UnauthorizedException for empty Bearer token', async () => {
  const mockContext = createMockExecutionContext({
    authorization: 'Bearer ',
  });

  await expect(guard.canActivate(mockContext)).rejects.toThrow(
    UnauthorizedException,
  );
});
```

Why test edge cases?

- Prevents security bypasses
- Ensures robust validation
- Handles malformed inputs

AdminGuard Testing

File: `admin.guard.spec.ts`

Test Setup

```
describe('AdminGuard', () => {
  let guard: AdminGuard;

  beforeEach(() => {
    guard = new AdminGuard();
```

```
  });
```

Mock Context with User

```
const createMockExecutionContext = (user: any): ExecutionContext => {
  return {
    switchToHttp: jest.fn().mockReturnValue({
      getRequest: jest.fn().mockReturnValue({
        user,  // User object attached by OktaAuthGuard
      }),
      getResponse: jest.fn(),
      getNext: jest.fn(),
    }),
    getClass: jest.fn(),
    getHandler: jest.fn(),
    getArgs: jest.fn(),
    getArgByIndex: jest.fn(),
    switchToRpc: jest.fn(),
    switchToWs: jest.fn(),
    getType: jest.fn(),
  } as unknown as ExecutionContext;
};
```

Key difference from OktaAuthGuard:

- Reads `user` object (not `headers`)
- Runs AFTER OktaAuthGuard sets user

Testing Admin Access

```
it('should return true for user with admin in groups', () => {
  const mockContext = createMockExecutionContext({
    id: 1,
    email: 'admin@test.com',
    groups: ['admin', 'users'],
  });

  const result = guard.canActivate(mockContext);
```

```
  expect(result).toBe(true);
});
```

Note: AdminGuard checks `groups` array, not `role` property.

Testing Non-Admin Users

```
it('should throw ForbiddenException for user without admin in groups', () => {
  const mockContext = createMockExecutionContext({
    id: 2,
    email: 'user@test.com',
    groups: ['users'],
  });

  expect(() => guard.canActivate(mockContext)).toThrow(
    ForbiddenException,
  );
});
```

Exception Types:

- `UnauthorizedException` (401): Authentication failure
- `ForbiddenException` (403): Authorization failure

Testing Missing User

```
it('should throw ForbiddenException when user is undefined', () => {
  const mockContext = createMockExecutionContext(undefined);

  expect(() => guard.canActivate(mockContext)).toThrow(
    ForbiddenException,
  );
});

it('should throw ForbiddenException when user has no groups property', () => {
  const mockContext = createMockExecutionContext({
    id: 3,
    email: 'nogroups@test.com',
  });
```

```
  expect(() => guard.canActivate(mockContext)).toThrow(
    ForbiddenException,
  );
});
```

Testing Case Sensitivity

```
it('should be case-sensitive for admin group check', () => {
  const mockContext = createMockExecutionContext({
    groups: ['Admin'], // uppercase A
  });

  // 'Admin' !== 'admin', should fail
  expect(() => guard.canActivate(mockContext)).toThrow(
    ForbiddenException,
  );
});
```

Testing Controllers with Guards

Override Guards in Controller Tests

```
describe('EmployeesController - with Guards', () => {
  let controller: EmployeesController;
  let service: EmployeesService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      controllers: [EmployeesController],
      providers: [
        {
          provide: EmployeesService,
          useValue: {
            create: jest.fn(),
            findAll: jest.fn(),
          },
```

```
        },
      ],
    })
      .overrideGuard(OktaAuthGuard)
      .useValue({ canActivate: () => true })  // Always allow
      .overrideGuard(AdminGuard)
      .useValue({ canActivate: () => true })  // Always allow
      .compile();

    controller = module.get<EmployeesController>(EmployeesController);
    service = module.get<EmployeesService>(EmployeesService);
  });

  it('should call service.create when guards pass', async () => {
    const createDto = { name: 'Test', email: 'test@test.com', ... };
    jest.spyOn(service, 'create').mockResolvedValue({ id: 1, ...createDto } as
any);

    await controller.create(createDto);

    expect(service.create).toHaveBeenCalledWith(createDto);
  });
});
```

Why override guards?

- Controller tests focus on business logic
- Guard logic already tested in guard specs
- Makes tests faster and simpler

Best Practices

☑ DO

1. **Test guards in isolation**

```
2. describe('MyGuard', () => {
3.   let guard: MyGuard;
4.   beforeEach(() => {
```

```
    guard = new MyGuard(); // Direct instantiation
  });
});
```

8. **Test all edge cases**
    - Missing data
    - Invalid formats
    - Null/undefined values
    - Empty strings/arrays
9. **Use correct exception types**

```
// Authentication failure
throw new UnauthorizedException();  // 401

// Authorization failure
throw new ForbiddenException();  // 403
```

15. **Mock ExecutionContext properly**
    - Use `jest.fn().mockReturnValue()` for method chains
    - Cast as `unknown as ExecutionContext` for type safety
16. **Mock dependencies**
    - ConfigService for guards needing configuration
    - Any external services

❌ DON'T

1. **Don't test guards in controller unit tests**

```
// BAD - Testing guard logic in controller test
it('should block non-admin users', () => { ... });

// GOOD - Override guards in controller tests
.overrideGuard(AdminGuard).useValue({ canActivate: () => true })
```

7. **Don't skip edge cases**
    - Always test failure scenarios
    - Test malformed inputs
    - Test missing/null values
8. **Don't use async for synchronous guards**

```
// BAD - AdminGuard.canActivate is synchronous
```

```
10. await expect(guard.canActivate(context)).rejects.toThrow(...);
11.
12. // GOOD
13. expect(() => guard.canActivate(context)).toThrow(...);
```

Common Patterns

Pattern 1: Testing Multiple Roles/Groups

```
const allowedGroups = ['admin', 'superadmin', 'moderator'];

allowedGroups.forEach(group => {
  it(`should allow access for ${group} group`, () => {
    const mockContext = createMockExecutionContext({
      user: { groups: [group] },
    });

    const result = guard.canActivate(mockContext);
    expect(result).toBe(true);
  });
});
```

Pattern 2: Async vs Sync Guards

```
// Async guard (OktaAuthGuard)
await
expect(guard.canActivate(context)).rejects.toThrow(UnauthorizedException);

// Sync guard (AdminGuard)
expect(() => guard.canActivate(context)).toThrow(ForbiddenException);
```

Pattern 3: Testing with ConfigService

```
mockConfigService = {
  get: jest.fn((key: string, defaultValue?: any) => {
```

```
    const config: Record<string, string> = {
      'API_KEY': 'test-key',
      'FEATURE_FLAG': 'true',
    };
    return config[key] || defaultValue;
  }),
} as any;
```

Running Tests

```
# Run guard tests
npm test okta-auth.guard.spec.ts
npm test admin.guard.spec.ts

# Run with coverage
npm test -- --coverage

# Watch mode
npm test -- --watch
```

Why Test Guards Separately?

- ☑ **Single Responsibility**: Each test focuses on one security concern
- ☑ **Faster Execution**: No need to set up entire request pipeline
- ☑ **Clear Failures**: Easy to identify security vs. business logic issues
- ☑ **Reusability**: Guards tested once, used across multiple controllers