Testing

Automatic tests are an essential part of the fully functional **software product**. That is very critical to cover at least the most sensitive parts of your system. In order to accomplish that goal, we produce a set of different tests like integration tests, unit tests, e2e tests, and so on. And Nest provides a bunch of test utilities that improves testing experience.

In general, you can use any **testing framework** that you enjoy working with. We don't enforce tooling, choose whatever fits your requirements. The main Nest application starter is integrated with Jest framework to reduce an amount of overhead when it comes to start writing your tests, but still, you can get rid of it and use any other tool easily.

Installation

Firstly, we need to install the required package:

```
$ npm i --save-dev @nestjs/testing
```

Unit testing

In the following example, we have two different classes, `CatsController` and `CatsService` respectively. As mentioned before, Jest is used as a fully-fledged testing framework. That framework behaves like a test-runner and also, provides assert functions and test-doubles utilities that helps with mocking, spying, etc. We have manually enforced `catsService.findAll()` method to return `result` variable, once it's called. Thanks to that, we can test whether `catsController.findAll()` returns expected result, or not.

cats.controller.spec.ts JS

```
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

  beforeEach(() => {
    catsService = new CatsService();
    catsController = new CatsController(catsService);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      expect(await catsController.findAll()).toBe(result);
    });
  });
});
```

HINT
Keep your test files nearby tested classes. The testing files should have a `.spec` or `.test` suffix.

We didn't make use of any existing Nest testing utility so far. Since we have manually taken care of instantiating tested classes, above test suite has nothing to do with Nest. This type of testing is called **isolated tests**.

Testing utilities

The `@nestjs/testing` package gives us a set of utilities that boost the testing process. Let's rewrite the previous example, but now, using exposed `Test` class.

cats.controller.spec.ts JS

```
import { Test } from '@nestjs/testing';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

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

    catsService = module.get<CatsService>(CatsService);
    catsController = module.get<CatsController>(CatsController);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);
      expect(await catsController.findAll()).toBe(result);
    });
  });
});
```

`Test` class has a `createTestingModule()` method that takes a module metadata (the same object as this one passed in `@Module()` decorator) as an argument. This method creates a `TestingModule` instance which in turn provides a few methods, but only one of them is useful when it comes to unit tests - the `compile()`. This function is **asynchronous**, therefore it has to be awaited. Once module is compiled, you can retrieve any instance using `get()` method.

In order to mock a real instance, you can override existing provider with a **custom provider**.

End-to-end testing

When the application grows, it is hard to manually test a behavior of each API endpoint. The end-to-end tests help us to make sure that everything is working correctly and fits project requirements. To perform e2e tests we use the same configuration as in the case of **unit testing**, but additionally, we take advantage of **supertest** library that allows simulating HTTP requests.

cats.e2e-spec.ts JS

```
import * as request from 'supertest';
import { Test } from '@nestjs/testing';
import { CatsModule } from '../../src/cats/cats.module';
import { CatsService } from '../../src/cats/cats.service';
import { INestApplication } from '@nestjs/common';

describe('Cats', () => {
  let app: INestApplication;
  let catsService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const module = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
      .compile();

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

  it(`/GET cats`, () => {
    return request(app.getHttpServer())
      .get('/cats')
      .expect(200)
      .expect({
        data: catsService.findAll(),
      });
  });

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

HINT
Keep your e2e test files inside the `e2e` directory. The testing files should have a `.e2e-spec` or `.e2e-test` suffix.

The `cats.e2e-spec.ts` test file contains a single HTTP endpoint test (`/cats`). We have used `app.getHttpServer()` method to pick up an underlying HTTP server that runs in the background of Nest application. Notice that `TestingModule` instance provides a `overrideProvider()` method, and thus we can **override** the existing provider which is declared by the imported module. Also, we can successively override the guards, interceptors, filters, and pipes using corresponding methods, `overrideGuard()` , `overrideInterceptor()` , `overrideFilter()` , and `overridePipe()` respectively.

The compiled module has several methods well described in the following table:

`createNestApplicaton()`	Creates a Nest instance based on a given module (returns `INestApplication`). Notice that it's necessary to manually initialize the application using `init()` method.
`createNestMicroservice()`	Creates a Nest microservice instance based on a given module (returns `INestMicroservice`).
`get()`	Retrieves an instance of either controller or provider (including guards, filters, and so on) available in the application context.
`select()`	Navigates through the modules graph, for example, to pull out a specific instance from the selected module (used along with enabled strict mode `strict: true` in `get()` method).

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more here.

Principal Sponsor

Valor SOFTWARE

Sponsors / Partners

Become a sponsor