Testing

Automated testing is considered an essential part of any serious software development effort. Automation makes it easy to repeat individual tests or test suites quickly and easily during development. This helps ensure that releases meet quality and performance goals. Automation helps increase coverage and provides a faster feedback loop to developers. Automation both increases the productivity of individual developers and ensures that tests are run at critical development lifecycle junctures, such as source code control check-in, feature integration, and version release.

Such tests often span a variety of types, including unit tests, end-to-end (e2e) tests, integration tests, and so on. While the benefits are unquestionable, it can be tedious to set them up. Nest strives to promote development best practices, including effective testing, so it includes features such as the following to help developers and teams build and automate tests. Nest:

- automatically scaffolds default unit tests for components and e2e tests for applications
- provides default tooling (such as a test runner that builds an isolated module/application loader)
- provides integration with **Jest** and **Supertest** out-of-the-box, while remaining agnostic to testing tools
- makes the Nest dependency injection system available in the testing environment for easily mocking components

As mentioned, you can use any **testing framework** that you like, as Nest doesn't force any specific tooling. Simply replace the elements needed (such as the test runner), and you will still enjoy the benefits of Nest's ready-made testing facilities.

Installation

To get started, first install the required package:

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

Unit testing

In the following example, we test two classes: `CatsController` and `CatsService`. As mentioned, **Jest** is provided as the default testing framework. It serves as a test-runner and also provides assert functions and test-double utilities that help with mocking, spying, etc. In the following basic test, we manually instantiate these classes, and ensure that the controller and service fulfill their API contract.

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 located near the classes they test. Testing files should have a `.spec` or `.test` suffix.

Because the above sample is trivial, we aren't really testing anything Nest-specific. Indeed, we aren't even using dependency injection (notice that we pass an instance of `CatsService` to our `catsController`). This form of testing - where we manually instantiate the classes being tested - is often called **isolated testing** as it is independent from the framework. Let's introduce some more advanced capabilities that help you test applications that make more extensive use of Nest features.

Testing utilities

The `@nestjs/testing` package provides a set of utilities that enable a more robust testing process. Let's rewrite the previous example using the built-in `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 moduleRef = await Test.createTestingModule({
        controllers: [CatsController],
        providers: [CatsService],
      }).compile();

    catsService = moduleRef.get<CatsService>(CatsService);
    catsController = moduleRef.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);
    });
  });
});
```

The `Test` class is useful for providing an application execution context that essentially mocks the full Nest runtime, but gives you hooks that make it easy to manage class instances, including mocking and overriding. The `Test` class has a `createTestingModule()` method that takes a module metadata object as its argument (the same object you pass to the `@Module()` decorator). This method returns a `TestingModule` instance which in turn provides a few methods. For unit tests, the important one is the `compile()` method. This method bootstraps a module with its dependencies (similar to the way an application is bootstrapped in the conventional `main.ts` file using `NestFactory.create()`), and returns a module that is ready for testing.

> **HINT**
> The `compile()` method is **asynchronous** and therefore has to be awaited. Once the module is compiled you can retrieve any **static** instance it declares (controllers and providers) using the `get()` method.

`TestingModule` inherits from the **module reference** class, and therefore its ability to dynamically resolve scoped providers (transient or request-scoped). Do this with the `resolve()` method (the `get()` method can only retrieve static instances).

```
const moduleRef = await Test.createTestingModule({
  controllers: [CatsController],
  providers: [CatsService],
}).compile();

catsService = await moduleRef.resolve(CatsService);
```

WARNING

The `resolve()` method returns a unique instance of the provider, from its own **DI container sub-tree**. Each sub-tree has a unique context identifier. Thus, if you call this method more than once and compare instance references, you will see that they are not equal.

HINT

Learn more about the module reference features here.

Instead of using the production version of any provider, you can override it with a **custom provider** for testing purposes. For example, you can mock a database service instead of connecting to a live database. We'll cover overrides in the next section, but they're available for unit tests as well.

End-to-end testing

Unlike unit testing, which focuses on individual modules and classes, end-to-end (e2e) testing covers the interaction of classes and modules at a more aggregate level -- closer to the kind of interaction that end-users will have with the production system. As an application grows, it becomes hard to manually test the end-to-end behavior of each API endpoint. Automated end-to-end tests help us ensure that the overall behavior of the

system is correct and meets project requirements. To perform e2e tests we use a similar configuration to the one we just covered in **unit testing**. In addition, Nest makes it easy to use the **Supertest** library to simulate 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 moduleRef = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
      .compile();

    app = moduleRef.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

If you're using **Fastify** as your HTTP adapter, it requires a slightly different configuration, and has built-in testing capabilities:

```
let app: NestFastifyApplication;

beforeAll(async () => {
  app = moduleRef.createNestApplication<NestFastifyApplication>(
    new FastifyAdapter(),
  );

  await app.init();
  await app.getHttpAdapter().getInstance().ready();
})

it(`/GET cats`, () => {
  return app
    .inject({
      method: 'GET',
      url: '/cats'
    }).then(result => {
      expect(result.statusCode).toEqual(200)
      expect(result.payload).toEqual(/* expectedPayload */)
    });
})
```

In this example, we build on some of the concepts described earlier. In addition to the `compile()` method we used earlier, we now use the `createNestApplication()` method to instantiate a full Nest runtime environment. We save a reference to the running app in our `app` variable so we can use it to simulate HTTP requests.

We simulate HTTP tests using the `request()` function from Supertest. We want these HTTP requests to route to our running Nest app, so we pass the `request()` function a reference to the HTTP listener that underlies Nest (which, in turn, may be provided by the Express platform). Hence the construction `request(app.getHttpServer())` . The call to `request()` hands us a wrapped HTTP Server, now connected to the Nest app, which exposes methods to simulate an actual HTTP request. For example, using `request(...).get('/cats')` will initiate a request to the Nest app that is identical to an **actual** HTTP request like `get '/cats'` coming in over the network.

In this example, we also provide an alternate (test-double) implementation of the `CatsService` which simply

returns a hard-coded value that we can test for. Use `overrideProvider()` to provide such an alternate implementation. Similarly, Nest provides methods to override guards, interceptors, filters and pipes with the `overrideGuard()`, `overrideInterceptor()`, `overrideFilter()`, and `overridePipe()` methods respectively.

Each of the override methods returns an object with 3 different methods that mirror those described for **custom providers**:

- `useClass`: you supply a class that will be instantiated to provide the instance to override the object (provider, guard, etc.).
- `useValue`: you supply an instance that will override the object.
- `useFactory`: you supply a function that returns an instance that will override the object.

Each of the override method types, in turn, returns the `TestingModule` instance, and can thus be chained with other methods in the **fluent style**. You should use `compile()` at the end of such a chain to cause Nest to instantiate and initialize the module.

Also, sometimes you may want to provide a custom logger e.g. when the tests are run (for example, on a CI server). Use the `setLogger()` method and pass an object that fulfills the `LoggerService` interface to instruct the `TestModuleBuilder` how to log during tests (by default, only "error" logs will be logged to the console).

The compiled module has several useful methods, as described in the following table:

`createNestApplication()`	Creates and returns a Nest application (`INestApplication` instance) based on the given module. Note that you must manually initialize the application using the `init()` method.
`createNestMicroservice()`	Creates and returns a Nest microservice (`INestMicroservice` instance) based on the given module.
`get()`	Retrieves a static instance of a controller or provider (including guards, filters, etc.) available in the application context. Inherited from the **module reference** class.
`resolve()`	Retrieves a dynamically created scoped instance (request or transient) of a controller or provider (including guards, filters, etc.) available in the application context. Inherited from the **module reference** class.
`select()`	Navigates through the module's dependency graph; can be used to retrieve a specific instance from the selected module (used along with strict mode (`strict: true`) in `get()`

method).

HINT

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

Overriding globally registered enhancers

If you have a globally registered guard (or pipe, interceptor, or filter), you need to take a few more steps to override that enhancer. To recap the original registration looks like this:

```
providers: [
  {
    provide: APP_GUARD,
    useClass: JwtAuthGuard,
  },
],
```

This is registering the guard as a "multi"-provider through the `APP_*` token. To be able to replace the `JwtAuthGuard` here, the registration needs to use an existing provider in this slot:

```
providers: [
  {
    provide: APP_GUARD,
    useExisting: JwtAuthGuard,
    // ^^^^^^^^ notice the use of 'useExisting' instead of 'useClass'
  },
  JwtAuthGuard,
],
```

HINT

Change the `useClass` to `useExisting` to reference a registered provider instead of having Nest instantiate it behind the token.

Now the `JwtAuthGuard` is visible to Nest as a regular provider that can be overridden when creating the

`TestingModule` :

```
const moduleRef = await Test.createTestingModule({
  imports: [AppModule],
})
  .overrideProvider(JwtAuthGuard)
  .useClass(MockAuthGuard)
  .compile();
```

Now all your tests will use the `MockAuthGuard` on every request.

Testing request-scoped instances

Request-scoped providers are created uniquely for each incoming **request**. The instance is garbage-collected after the request has completed processing. This poses a problem, because we can't access a dependency injection sub-tree generated specifically for a tested request.

We know (based on the sections above) that the `resolve()` method can be used to retrieve a dynamically instantiated class. Also, as described **here**, we know we can pass a unique context identifier to control the lifecycle of a DI container sub-tree. How do we leverage this in a testing context?

The strategy is to generate a context identifier beforehand and force Nest to use this particular ID to create a sub-tree for all incoming requests. In this way we'll be able to retrieve instances created for a tested request.

To accomplish this, use `jest.spyOn()` on the `ContextIdFactory` :

```
const contextId = ContextIdFactory.create();
jest
  .spyOn(ContextIdFactory, 'getByRequest')
  .mockImplementation(() => contextId);
```

Now we can use the `contextId` to access a single generated DI container sub-tree for any subsequent request.

```
catsService = await moduleRef.resolve(CatsService, contextId);
```

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 Sponsors

Sponsors / Partners

Become a sponsor

Join our Newsletter

Subscribe to stay up to date with the latest Nest updates, features, and videos!

Email address..