Automock

Automock is a standalone library for unit testing. Using TypeScript Reflection API (`reflect-metadata`) internally to produce mock objects, Automock streamlines test development by automatically mocking class external dependencies.

> info **info** `Automock` is a third party package and is not managed by the NestJS core team. Please, report any issues found with the library in the appropriate repository

Introduction

The dependency injection (DI) container is an essential component of the Nest module system. This container is utilized both during testing, and the application execution. Unit tests vary from other types of tests, such as integration tests, in that they must fully override providers/services within the DI container. External class dependencies (providers) of the so-called "unit", have to be totally isolated. That is, all dependencies within the DI container should be replaced by mock objects. As a result, loading the target module and replacing the providers inside it is a process that loops back on itself. Automock tackles this issue by automatically mocking all the class external providers, resulting in total isolation of the unit under test.

Installation

```
$ npm i -D @automock/jest
```

Automock does not require any additional setup.

> info **info** Jest is the only test framework currently supported by Automock. Sinon will shortly be released.

Example

Consider the following cats service, which takes three constructor parameters:

```
@@filename(cats.service)
import { Injectable } from '@nestjs/core';

@Injectable()
export class CatsService {
  constructor(
    private logger: Logger,
    private httpService: HttpService,
    private catsDal: CatsDal,
  ) {}

  async getAllCats() {
    const cats = await
this.httpService.get('http://localhost:3000/api/cats');
```

```
    this.logger.log('Successfully fetched all cats');

    this.catsDal.saveCats(cats);
  }
}
```

The service contains one public method, `getAllCats`, which is the method we use an example for the following unit test:

```
@@filename(cats.service.spec)
import { TestBed } from '@automock/jest';
import { CatsService } from './cats.service';

describe('CatsService unit spec', () => {
  let underTest: CatsService;
  let logger: jest.Mocked<Logger>;
  let httpService: jest.Mocked<HttpService>;
  let catsDal: jest.Mocked<CatsDal>;

  beforeAll(() => {
    const { unit, unitRef } = TestBed.create(CatsService)
      .mock(HttpService)
      .using({ get: jest.fn() })
      .mock(Logger)
      .using({ log: jest.fn() })
      .mock(CatsDal)
      .using({ saveCats: jest.fn() })
      .compile();

    underTest = unit;

    logger = unitRef.get(Logger);
    httpService = unitRef.get(HttpService);
    catsDal = unitRef.get(CatsDal);
  });

  describe('when getting all the cats', () => {
    test('then meet some expectations', async () => {
      httpService.get.mockResolvedValueOnce([{ id: 1, name: 'Catty' }]);
      await catsService.getAllCats();

      expect(logger.log).toBeCalled();
      expect(catsDal).toBeCalledWith([{ id: 1, name: 'Catty' }]);
    });
  });
});
```

> info **info** The jest.Mocked utility type returns the Source type wrapped with type definitions of Jest mock function. (reference)

About `unit` and `unitRef`

Let's examine the following code:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();
```

Calling `.compile()` returns an object with two properties, `unit`, and `unitRef`.

`unit` is the unit under test, it is an actual instance of class being tested.

`unitRef` is the "unit reference", where the mocked dependencies of the tested class are stored, in a small container. The container's `.get()` method returns the mocked dependency with all of its methods automatically stubbed (using `jest.fn()`):

```
const { unit, unitRef } = TestBed.create(CatsService).compile();

let httpServiceMock: jest.Mocked<HttpService> = unitRef.get(HttpService);
```

> info **info** The `.get()` method can accept either a `string` or an actual class (`Type`) as its argument. This essentially depends on how the provider is being injected to the class under test.

Working with different providers

Providers are one of the most important elements in Nest. You can think of many of the default Nest classes as providers, including services, repositories, factories, helpers, and so on. A provider's primary function is to take the form of an `Injectable` dependency.

Consider the following `CatsService`, it takes one parameter, which is an instance of the following `Logger` interface:

```
export interface Logger {
  log(message: string): void;
}

export class CatsService {
  constructor(private logger: Logger) {}
}
```

TypeScript's Reflection API does not support interface reflection yet. Nest solves this issue with string-based injection tokens (see Custom Providers):

```
export const MyLoggerProvider = {
  provide: 'MY_LOGGER_TOKEN',
  useValue: { ... },
}
```

```
export class CatsService {
  constructor(@Inject('MY_LOGGER_TOKEN') private readonly logger: Logger)
{}
}
```

Automock follows this practice and lets you provide a string-based token instead of providing the actual class in the `unitRef.get()` method:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();

let loggerMock: jest.Mocked<Logger> = unitRef.get('MY_LOGGER_TOKEN');
```

More Information

Visit Automock GitHub repository for more information.