Test

August 17, 2026

Test

Kind: Class

Source: `packages/testing/test.ts`

Part of: Testing

`Test` is the entry point for creating isolated NestJS testing modules. Its `createTestingModule()` method accepts module metadata and returns a builder that can configure providers, overrides, and dependencies before compiling a testable application context.

Methods

Method	Signature	Returns
`createTestingModule`	`createTestingModule(metadata: ModuleMetadata, options: TestingModuleOptions)`	`void`

Diagram

```
graph LR
  A[Test.createTestingModule] --> B[TestingModuleBuilder]
  B --> C[Configure imports, providers, controllers]
  B --> D[Override providers or guards]
  C --> E[compile()]
  D --> E
  E --> F[TestingModule]
  F --> G[Resolve and test dependencies]
```

Usage

```
import { Test } from '@nestjs/testing';
import { UsersService } from './users.service';
```

```ts
import { UsersController } from './users.controller';

describe('UsersController', () => {
  it('returns a user', async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [UsersController],
      providers: [
        {
          provide: UsersService,
          useValue: {
            findOne: jest.fn().mockResolvedValue({ id: '1', name: 'Ada' }),
          },
        },
      ],
    }).compile();

    const controller = moduleRef.get(UsersController);

    await expect(controller.findOne('1')).resolves.toEqual({
      id: '1',
      name: 'Ada',
    });
  });
});
```

AI Coding Instructions

- Use `Test.createTestingModule()` as the starting point for unit and integration test dependency setup.
- Include only the controllers, providers, and imports required by the test to keep modules isolated and fast.
- Replace external dependencies, such as databases, HTTP clients, and queues, with `useValue` or `useFactory` mocks.
- Call `.compile()` before retrieving dependencies with `moduleRef.get()` or creating an application instance.
- Use builder override methods when testing modules that already register concrete providers or framework dependencies.

How it works

`Test` is an exported class whose role is to start construction of a testing module. It holds one private static `MetadataScanner` instance and passes that same instance to every builder it creates. [packages/testing/test.ts:8-15]

- `Test.createTestingModule(metadata, options?)` is a static factory method. It accepts `ModuleMetadata` and an optional `TestingModuleOptions`, then returns a new `TestingModuleBuilder`; it does not compile the module itself. [packages/testing/test.ts:11-16]
- The supplied `metadata` and optional `options` are forwarded unchanged to the builder constructor. [packages/testing/test.ts:12-15]
- During builder construction, a new `ApplicationConfig` and `NestContainer` are created, and the metadata is applied with `@Module(...)` to a locally declared `RootTestModule` class. [packages/testing/testing-module.builder.ts:38-56] [packages/testing/testing-module.builder.ts:195-199]
- The returned builder exposes configuration methods for logger selection, mocks, and provider/module overrides, followed by `compile()` to scan dependencies, create instances, and return a `TestingModule`. [packages/testing/testing-module.builder.ts:58-95] [packages/testing/testing-module.builder.ts:97-132]
- `TestingModuleOptions` currently contains only `moduleIdGeneratorAlgorithm`, inherited as a `Pick` from application-context options. [packages/testing/testing-module.builder.ts:29-32]
- `Test` itself contains no input validation and explicitly throws no errors. [packages/testing/test.ts:8-16]

Relationships

- IMPORTS → `ModuleMetadata`
- IMPORTS → `MetadataScanner`