Module

August 17, 2026

Module

Kind: Class

Source: `packages/core/injector/module.ts`

Part of: Core

`Module` is Nest's internal runtime representation of an application module within the dependency-injection container. It registers the module itself, `ModuleRef`, application configuration, injectable classes, and custom providers so dependencies can be resolved within the module scope.

Methods

Method	Signature	Returns
`addCoreProviders`	`addCoreProviders()`	`void`
`addModuleRef`	`addModuleRef()`	`void`
`addModuleAsProvider`	`addModuleAsProvider()`	`void`
`addApplicationConfig`	`addApplicationConfig()`	`void`
`addInjectable`	`addInjectable(injectable: Provider, enhancerSubtype: EnhancerSubtype, host: Type<T>)`	`void`
`addProvider`	`addProvider(provider: Provider)`	`InjectionToken`
`addProvider`	`addProvider(provider: Provider, enhancerSubtype: EnhancerSubtype)`	`InjectionToken`
`addProvider`	`addProvider(provider: Provider, enhancerSubtype: EnhancerSubtype)`	`void`
`isCustomProvider`	`isCustomProvider(provider: Provider)`	`provider is
`addCustomProvider`	`addCustomProvider(provider:	ClassProvider
`isCustomClass`	`isCustomClass(provider: any)`	`provider is ClassProvider`

Method	Signature	Returns
`isCustomValue`	`isCustomValue(provider: any)`	`provider is ValueProvider`
`isCustomFactory`	`isCustomFactory(provider: any)`	`provider is FactoryProvider`
`isCustomUseExisting`	`isCustomUseExisting(provider: any)`	`provider is ExistingProvider`
`isDynamicModule`	`isDynamicModule(exported: any)`	`exported is DynamicModule`
`addCustomClass`	`addCustomClass(provider: ClassProvider, collection: Map<InjectionToken, InstanceWrapper>, enhancerSubtype: EnhancerSubtype)`	`void`
`addCustomValue`	`addCustomValue(provider: ValueProvider, collection: Map<Function	string
`addCustomFactory`	`addCustomFactory(provider: FactoryProvider, collection: Map<Function	string
`addCustomUseExisting`	`addCustomUseExisting(provider: ExistingProvider, collection: Map<Function	string
`addExportedProviderOrModule`	`addExportedProviderOrModule(toExport: Provider	string
`addCustomExportedProvider`	`addCustomExportedProvider(provider:	FactoryProvider
`validateExportedProvider`	`validateExportedProvider(token: InjectionToken)`	`void`
`addController`	`addController(controller: Type<Controller>)`	`void`
`assignControllerUniqueId`	`assignControllerUniqueId(controller: Type<Controller>)`	`void`
`addImport`	`addImport(moduleRef: Module)`	`void`
`replace`	`replace(toReplace: InjectionToken, options: any)`	`void`
`hasProvider`	`hasProvider(token: InjectionToken)`	`boolean`
`hasInjectable`	`hasInjectable(token: InjectionToken)`	`boolean`
`getProviderByKey`	`getProviderByKey(name: InjectionToken<T>)`	`InstanceWrapper<T>`

Method	Signature	Returns
`getProviderById`	`getProviderById(id: string)`	`InstanceWrapper
`getControllerById`	`getControllerById(id: string)`	`InstanceWrapper
`getInjectableById`	`getInjectableById(id: string)`	`InstanceWrapper
`getMiddlewareById`	`getMiddlewareById(id: string)`	`InstanceWrapper
`getNonAliasProviders`	`getNonAliasProviders()`	`Array< [InjectionToken, InstanceWrapper<Injectable>] >`
`createModuleReferenceType`	`createModuleReferenceType()`	`Type<ModuleRef>`

Where it refuses work

- `Module` stops the work with `RuntimeException` when `!this._providers.has(this._metatype)`.
- `Module` stops the work with `InvalidClassException` when `!(type && isFunction(type) && type.prototype)`.
- `Module` stops the work with an early return when `this.isCustomProvider(injectable)`.
- `Module` stops the work with an early return when `(this.isTransientProvider(provider) || this.isRequestScopeProvider(provider)) && isAlread⋯`.
- `Module` stops the work with an early return when `isString(provide) || isSymbol(provide)`.
- `Module` stops the work with an early return when `this._providers.has(token)`.

Diagram

```
graph LR
  A[Module Metadata] --> B[Module Instance]
  B --> C[Core Providers]
  C --> C1[Module Class Provider]
  C --> C2[ModuleRef]
  C --> C3[ApplicationConfig]
  B --> D[addProvider]
  B --> E[addInjectable]
  D --> F{Custom Provider?}
  F -->|No| G[Class Provider Wrapper]
  F -->|Yes| H[Factory / Value / Existing / Class Provider]
  G --> I[Provider Registry]
  H --> I
  E --> J[Injectable Registry]
```

Usage

```
import { Module as RuntimeModule } from '@nestjs/core/injector/module';
import { NestContainer } from '@nestjs/core/injector/container';
import { ApplicationConfig } from '@nestjs/core/application-config';

class UsersService {
  findAll() {
    return ['Ada', 'Grace'];
  }
}

class AppModule {}

const applicationConfig = new ApplicationConfig();
const container = new NestContainer(applicationConfig);

// RuntimeModule is typically created by Nest internally during scanning.
const moduleRef = new RuntimeModule(AppModule, container);

// Register a standard class provider.
moduleRef.addProvider(UsersService);

// Register a custom value provider.
moduleRef.addProvider({
  provide: 'API_URL',
  useValue: 'https://api.example.com',
});

// Register an injectable used by framework features such as guards or pipes.
moduleRef.addInjectable(UsersService);
```

AI Coding Instructions

- Treat `Module` as an internal Nest container type; application code should usually register providers through the `@Module()` decorator instead.
- Use `addProvider()` for both class providers and custom provider objects; custom providers are detected through `isCustomProvider()`.
- Preserve provider tokens when adding or modifying registrations, since tokens are the keys used for dependency resolution.
- Ensure new providers are associated with the correct module instance so module-scoped resolution and exports continue to work.
- Do not remove the core-provider setup path: module classes, `ModuleRef`, and application configuration must remain registered for Nest runtime behavior.

How it works

`Module` is a mutable runtime record for one module metatype within the dependency-injection container. It stores the module class, container reference, an ID, imports, exports, providers, injectables, middleware wrappers, controller wrappers, and entry-provider tokens. [packages/core/injector/module.ts:44-77]

The container creates it from a compiled module type, assigns its container token afterward, stores it in the container's module map, and may mark it global, set its distance to `Number.MAX_VALUE`, and register it among global modules. [packages/core/injector/container.ts:163-184]

Relationships

- IMPORTS → `EnhancerSubtype`
- IMPORTS → `ENTRY_PROVIDER_WATERMARK`
- IMPORTS → `ClassProvider`
- IMPORTS → `Controller`
- IMPORTS → `DynamicModule`
- IMPORTS → `ExistingProvider`
- IMPORTS → `FactoryProvider`
- IMPORTS → `Injectable`
- IMPORTS → `InjectionToken`
- IMPORTS → `NestModule`
- IMPORTS → `Provider`
- IMPORTS → `Scope`
- IMPORTS → `Type`
- IMPORTS → `ValueProvider`
- IMPORTS → `randomStringGenerator`
- IMPORTS → `isFunction`
- IMPORTS → `isNil`
- IMPORTS → `isObject`
- IMPORTS → `isString`
- IMPORTS → `isSymbol`
- IMPORTS → `isUndefined`

Used by

4 references from 4 files. Each is a place in this repository where the symbol is actually used — go read one rather than trusting an example.

Imported by (4)

- `ListenersController` — `packages/microservices/listeners-controller.ts`:45
- `TestingInjector` — `packages/testing/testing-injector.ts`:23
- `TestingInstanceLoader` — `packages/testing/testing-instance-loader.ts`:6
- `TestingModule` — `packages/testing/testing-module.ts`:26