Injector

August 18, 2026

Injector

Kind: Class

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

Part of: Core

`Injector` is Nest's internal dependency-injection runtime responsible for creating class, factory, controller, middleware, and provider instances. It resolves constructor dependencies, tracks circular or asynchronous resolution through settlement signals, and stores resolved instances in their module collections and execution contexts.

Methods

Method	Signature	Returns
`loadPrototype`	`loadPrototype({ token }: InstanceWrapper<T>, collection: Map<InjectionToken, InstanceWrapper<T>>, contextId: undefined)`	`void`
`loadInstance`	`loadInstance(wrapper: InstanceWrapper<T>, collection: Map<InjectionToken, InstanceWrapper>, moduleRef: Module, resolutionContext: ResolutionContext)`	`void`
`loadMiddleware`	`loadMiddleware(wrapper: InstanceWrapper, collection: Map<InjectionToken, InstanceWrapper>, moduleRef: Module, contextId: undefined, inquirer: InstanceWrapper)`	`void`
`loadController`	`loadController(wrapper: InstanceWrapper<Controller>, moduleRef: Module, contextId: undefined)`	`void`
`loadInjectable`	`loadInjectable(wrapper: InstanceWrapper<T>, moduleRef: Module, contextId: undefined, inquirer: InstanceWrapper)`	`void`
`loadProvider`	`loadProvider(wrapper: InstanceWrapper<Injectable>, moduleRef: Module, resolutionContext: ResolutionContext)`	`void`
`applySettlementSignal`	`applySettlementSignal(instancePerContext: InstancePerContext<T>, host: InstanceWrapper<T>)`	`void`
`resolveConstructorParams`	`resolveConstructorParams(wrapper: InstanceWrapper, moduleRef: Module, inject: InjectorDependency[]	undefined, callback: (args: unknown[]) => void
`getClassDependencies`	`getClassDependencies(wrapper: InstanceWrapper<T>)`	`[InjectorDependency[], number[]]`

Method	Signature	Returns
`getFactoryProviderDependencies`	`getFactoryProviderDependencies(wrapper: InstanceWrapper<T>)`	`[InjectorDependency[], number[]]`
`reflectConstructorParams`	`reflectConstructorParams(type: Type	Function)`
`reflectOptionalParams`	`reflectOptionalParams(type: Type	Function)`
`reflectSelfParams`	`reflectSelfParams(type: Type	Function)`
`resolveSingleParam`	`resolveSingleParam(wrapper: InstanceWrapper, param: Type	string
`resolveParamToken`	`resolveParamToken(wrapper: InstanceWrapper, param: Type	string
`resolveComponentWrapper`	`resolveComponentWrapper(moduleRef: Module, token: InjectionToken, dependencyContext: InjectorDependencyContext, wrapper: InstanceWrapper, resolutionContext: ResolutionContext, keyOrIndex: symbol	string
`resolveComponentHost`	`resolveComponentHost(moduleRef: Module, instanceWrapper: InstanceWrapper<T	Promise>, resolutionContext: ResolutionContext)`
`lookupComponent`	`lookupComponent(providers: Map<Function	string
`lookupComponentInParentModules`	`lookupComponentInParentModules(dependencyContext: InjectorDependencyContext, moduleRef: Module, wrapper: InstanceWrapper, resolutionContext: ResolutionContext, keyOrIndex: symbol	string
`lookupComponentInImports`	`lookupComponentInImports(moduleRef: Module, name: InjectionToken, wrapper: InstanceWrapper, moduleRegistry: Set, resolutionContext: ResolutionContext, keyOrIndex: symbol	string
`resolveProperties`	`resolveProperties(wrapper: InstanceWrapper<T>, moduleRef: Module, inject: InjectorDependency[], resolutionContext: ResolutionContext, parentInquirer: InstanceWrapper)`	`Promise<PropertyDependency[]>`
`reflectProperties`	`reflectProperties(type: Type<T>)`	`PropertyDependency[]`
`applyProperties`	`applyProperties(instance: T, properties: PropertyDependency[])`	`void`
`instantiateClass`	`instantiateClass(instances: any[], wrapper: InstanceWrapper, targetMetatype: InstanceWrapper, resolutionContext: ResolutionContext)`	`Promise<T>`
`loadPerContext`	`loadPerContext(instance: T, moduleRef: Module, collection: Map<InjectionToken, InstanceWrapper>, ctx: ContextId, wrapper: InstanceWrapper)`	`Promise<T>`
`loadEnhancersPerContext`	`loadEnhancersPerContext(wrapper: InstanceWrapper, ctx: ContextId, inquirer: InstanceWrapper)`	`void`
`loadCtorMetadata`	`loadCtorMetadata(metadata: InstanceWrapper<any>[], contextId: ContextId, inquirer: InstanceWrapper, parentInquirer: InstanceWrapper)`	`Promise<any[]>`

Method	Signature	Returns
`loadPropertiesMetadata`	`loadPropertiesMetadata(metadata: PropertyMetadata[], contextId: ContextId, inquirer: InstanceWrapper)`	`Promise<PropertyDependency[]>`
`addDependencyMetadata`	`addDependencyMetadata(keyOrIndex: symbol`	string

Where it refuses work

- `Injector` stops the work with `CircularDependencyException` when `resolutionContext.inquirer && settlementSignal?.isCycle(resolutionContext.inquirer.id)`.
- `Injector` stops the work with `RuntimeException` when `isUndefined(targetWrapper)`.
- `Injector` stops the work with `UnknownDependenciesException` when `wrapper && token === name`.
- `Injector` stops the work with `UnknownDependenciesException` when `isNil(instanceWrapper)`.
- `Injector` stops the work with an early return when `!this.isDebugMode()`, in 3 places.
- `Injector` stops the work with an early return when `!collection`.

When something fails

- `Injector` handles failure in 3 places: it lets it reach the caller in all 3.

Diagram

```
graph LR
  A[Module collections] --> B[Injector]
  B --> C[loadProvider / loadInjectable]
  B --> D[loadController / loadMiddleware]
  B --> E[loadPrototype]
  C --> F[resolveConstructorParams]
  D --> F
  F --> G[getClassDependencies]
  F --> H[getFactoryProviderDependencies]
  G --> I[Instantiate class or factory]
  H --> I
  I --> J[applySettlementSignal]
  J --> K[Resolved instance in InstanceWrapper]
```

Usage

```
import { Injector } from '@nestjs/core/injector/injector';
import { Module } from '@nestjs/core/injector/module';
import { InstanceWrapper } from '@nestjs/core/injector/instance-wrapper';

// This is typically performed by Nest internals during application bootstrap.
async function loadCustomProvider(
  moduleRef: Module,
  providerWrapper: InstanceWrapper,
) {
  const injector = new Injector();

  // Resolves constructor dependencies and creates the provider instance.
  await injector.loadProvider(providerWrapper, moduleRef);
```

```
    return providerWrapper.instance;
  }
```

AI Coding Instructions

- Treat `Injector` as framework-internal infrastructure; prefer Nest's public module and provider APIs in application code.
- Use `loadProvider`, `loadInjectable`, `loadController`, and `loadMiddleware` with the matching `InstanceWrapper` collection owned by a `Module`.
- Preserve the `contextId` and `inquirer` arguments when working with request-scoped or transient providers.
- Do not manually bypass `resolveConstructorParams` or `applySettlementSignal`; they handle dependency ordering, async resolution, and circular dependency coordination.
- When adding provider types, ensure both class-based and factory-provider dependency paths are represented in `getClassDependencies` or `getFactoryProviderDependencies`.

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
- `MicroservicesModule` — `packages/microservices/microservices-module.ts`:23
- `NestMicroservice` — `packages/microservices/nest-microservice.ts`:35
- `TestingInjector` — `packages/testing/testing-injector.ts`:23