ConfigurableModuleAsyncOptions

August 18, 2026

ConfigurableModuleAsyncOptions

Kind: Interface

Source: `packages/common/module-utils/interfaces/configurable-module-async-options.interface.ts`

Part of: Common

Interface that represents the module async options object
Factory method name varies depending on the "FactoryClassMethodKey" type argument.

`ConfigurableModuleAsyncOptions` defines how a configurable module resolves its options asynchronously during module registration. It supports options provided by an existing factory, a newly instantiated factory class, or an inline factory function, with optional dependency injection for factory arguments.

Properties

Property	Type
`useExisting`	`Type< ConfigurableModuleOptionsFactory<ModuleOptions, FactoryClassMethodKey> >`
`useClass`	`Type< ConfigurableModuleOptionsFactory<ModuleOptions, FactoryClassMethodKey> >`
`useFactory`	`(...args: any[]) => Promise
`inject`	`FactoryProvider['inject']`
`provideInjectionTokensFrom`	`Provider[]`

Diagram

```
graph LR
  A[ConfigurableModuleAsyncOptions] --> B{Options source}
  B --> C[useExisting]
  B --> D[useClass]
  B --> E[useFactory]

  C --> F[ConfigurableModuleOptionsFactory]
  D --> F
  E --> G[ModuleOptions]

  H[inject] --> E
  I[provideInjectionTokensFrom] --> H
  F --> G
```

Usage

```
import { Injectable } from '@nestjs/common';
import { ConfigurableModuleAsyncOptions } from '@nestjs/common';

interface DatabaseModuleOptions {
  host: string;
  port: number;
}

@Injectable()
class DatabaseConfigService {
  getDatabaseOptions(): DatabaseModuleOptions {
    return {
      host: 'localhost',
      port: 5432,
    };
  }
}

const asyncOptions: ConfigurableModuleAsyncOptions<DatabaseModuleOptions> = {
  inject: [DatabaseConfigService],
  useFactory: (configService: DatabaseConfigService) => {
    return configService.getDatabaseOptions();
  },
};

// Example module registration:
// DatabaseModule.registerAsync(asyncOptions);
```

AI Coding Instructions

- Use exactly one options strategy: `useFactory`, `useClass`, or `useExisting`; avoid combining factory sources unless the consuming module explicitly supports it.
- Add all dependencies required by `useFactory` to `inject`, preserving the same parameter order as the factory function.
- Use `useExisting` when a compatible options factory is already registered as a provider; use `useClass` when the module should create the factory provider.
- Ensure factory methods return `ModuleOptions` or `Promise<ModuleOptions>` and match the method key expected by `FactoryClassMethodKey`.
- Use `provideInjectionTokensFrom` when injection tokens must be collected from additional providers before resolving factory dependencies.