ClientsProviderAsyncOptions

August 18, 2026

ClientsProviderAsyncOptions

Kind: Interface

Source: `packages/microservices/module/interfaces/clients-module.interface.ts`

Part of: Microservices

`ClientsProviderAsyncOptions` configures an asynchronously created microservice client provider. It supports resolving client options from an existing factory, instantiating a factory class, or invoking a factory function with injected dependencies. The optional provider metadata helps Nest register the resulting client under a string or symbol token.

Properties

Property	Type
`useExisting`	`Type<ClientsModuleOptionsFactory>`
`useClass`	`Type<ClientsModuleOptionsFactory>`
`useFactory`	`(...args: any[]) => Promise
`inject`	`any[]`
`extraProviders`	`Provider[]`
`name`	`string

Diagram

```
graph LR
  A[ClientsProviderAsyncOptions] --> B{Configuration strategy}
  B -->|useExisting| C[Existing ClientsModuleOptionsFactory]
  B -->|useClass| D[New ClientsModuleOptionsFactory]
  B -->|useFactory| E[Factory Function]
```

```
  F[inject dependencies] --> E
  G[extraProviders] --> D

  C --> H[ClientProvider]
  D --> H
  E --> H

  H --> I[name: string or symbol]
  I --> J[Injectable Microservice Client]
```

Usage

```
import { Injectable } from '@nestjs/common';
import {
  ClientProvider,
  ClientsModuleOptionsFactory,
} from '@nestjs/microservices';
import { ClientsProviderAsyncOptions } from '@nestjs/microservices/module/interfaces/clients-mo

@Injectable()
class MessagingClientConfigService implements ClientsModuleOptionsFactory {
  createClientOptions(): ClientProvider {
    return {
      name: 'MESSAGING_CLIENT',
      transport: 0, // Replace with the appropriate Transport enum value
      options: {
        host: 'localhost',
        port: 3001,
      },
    };
  }
}

const messagingClientOptions: ClientsProviderAsyncOptions = {
  name: 'MESSAGING_CLIENT',
  useClass: MessagingClientConfigService,
  extraProviders: [MessagingClientConfigService],
};

// Example registration:
// ClientsModule.registerAsync([messagingClientOptions]);
```

AI Coding Instructions

- Use exactly one configuration strategy: `useExisting`, `useClass`, or `useFactory`; do not combine them for the same client registration.

- Ensure the resolved factory returns a valid `ClientProvider`, either directly or through a `Promise<ClientProvider>`.
- Add dependencies required by `useFactory` to `inject`, and ensure those dependencies are available in the importing module.
- Use `extraProviders` when the async client factory or its supporting services must be registered alongside the client configuration.
- Keep the `name` token consistent with the token used when injecting the resulting microservice client.