nest

Injection scopes

For the people coming from different languages, it might be awkward that in Nest almost everything is shared across the incoming requests. We have a connection pool to the database, singleton services with a global state etc. Generally, Node.js doesn't follow request/response Multi-Threaded Stateless Model in which every request is being processed by the separate thread. Hence, using singleton instances is fully **safe** for our applications.

However, there are edge-cases when request-based lifetime of the controller may be an intentional behavior, for instance per-request cache in GraphQL applications, request tracking or multi-tenancy. How can we handle them?

Scopes

Basically, every provider can act as a singleton, be request-scoped, and be switched to the transient mode. See the following table to get familiar with the differences between them.

`SINGLETON`	Each provider can be **shared** across multiple classes. The provider lifetime is strictly tied to the application lifecycle. Once the application has bootstrapped, all providers are already instantiated. The singleton scope is being used by default.
`REQUEST`	A new instance of the provider is going to be exclusively created for every incoming **request** and garbage collected after the request processing is completed.
`TRANSIENT`	Transient providers cannot be shared between providers. Every time when another provider asks the Nest container for particular transient provider, the container will create a new, dedicated instance.

HINT
Using a singleton scope is always the **recommended** way. Sharing providers among requests leads to lower memory consumption and thus to better performance of your application (no requirement to instantiate class every time).

Usage

In order to switch to another injection scope, you have to pass an argument to the `@Injectable()` decorator:

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

@Injectable({ scope: Scope.REQUEST })
```

```
export class CatsService {}
```

In the case of **custom providers**, you have to set an extra `scope` property:

```
{
  provide: 'CACHE_MANAGER',
  useClass: CacheManager,
  scope: Scope.TRANSIENT,
}
```

And when it comes to controllers, pass the `ControllerOptions` object:

```
@Controller({
  path: 'cats',
  scope: Scope.REQUEST,
})
export class CatsController {}
```

NOTICE
Gateways should never rely on request-scoped providers because they act as singletons. One gateway encapsulates a real socket inside and cannot be instantiated multiple times.

Per-request injection

The request-scoped providers have to be used very carefully. Keep in mind that the scope actually bubbles up in the **injection chain**. If your controller depends on a provider which is request-scoped, it means that your controller is actually request-scoped as well.

Imagine the following chain: `CatsController <- CatsService <- CatsRepository`. If your `CatsService` is request-scoped (and the rest are, theoretically, singletons), the `CatsController` would become request-scoped too (because request-scoped instance have to be injected into a newly created controller), whereas `CatsRepository` would remain as a singleton.

WARNING
The circular dependencies in this case will lead to very painful side-effects and thus, you should certainly avoid creating them.

Request provider

In the HTTP application, using request-scoped providers gives you a capability to inject an original request reference.

```
import { Injectable, Scope, Inject } from '@nestjs/common';
import { REQUEST } from '@nestjs/core';
import { Request } from 'express';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(REQUEST) private readonly request: Request) {}
}
```

However, this functionality doesn't work with either micro services or GraphQL applications. In **GraphQL** applications, you can inject `CONTEXT` instead.

```
import { Injectable, Scope, Inject } from '@nestjs/common';
import { CONTEXT } from '@nestjs/graphql';

@Injectable({ scope: Scope.REQUEST })
export class CatsService {
  constructor(@Inject(CONTEXT) private readonly context) {}
}
```

Afterwards, you can configure your `context` value (in the `GraphQLModule`) to contain `request` as its property.

Performance

Using request-scoped providers will obviously affect application performance. Even though Nest is trying to cache as much metadata as possible, it will still have to create an instance of your class on each request. Hence, it will slow down your average response time and overall benchmarking result. If your provider doesn't necessarily need to be request-scoped, you should rather stick with the singleton scope.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more **here**.

Principal Sponsor

Sponsors / Partners

Become a sponsor

Copyright © 2017-2019 MIT by Kamil Mysliwiec | design by Jakub Staron
Official NestJS Consulting Trilon.io | hosted by Netlify