Injection scopes

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

However, there are edge-cases when request-based lifetime may be the desired behavior, for instance per-request caching in GraphQL applications, request tracking, and multi-tenancy. Injection scopes provide a mechanism to obtain the desired provider lifetime behavior.

Provider scope

A provider can have any of the following scopes:

`DEFAULT`	A single instance of the provider is shared across the entire application. The instance lifetime is tied directly to the application lifecycle. Once the application has bootstrapped, all singleton providers have been instantiated. Singleton scope is used by default.
`REQUEST`	A new instance of the provider is created exclusively for each incoming **request**. The instance is garbage-collected after the request has completed processing.
`TRANSIENT`	Transient providers are not shared across consumers. Each consumer that injects a transient provider will receive a new, dedicated instance.

> info **Hint** Using singleton scope is **recommended** for most use cases. Sharing providers across consumers and across requests means that an instance can be cached and its initialization occurs only once, during application startup.

Usage

Specify injection scope by passing the `scope` property to the `@Injectable()` decorator options object:

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

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

Similarly, for custom providers, set the `scope` property in the long-hand form for a provider registration:

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

> info **Hint** Import the `Scope` enum from `@nestjs/common`

Singleton scope is used by default, and need not be declared. If you do want to declare a provider as singleton scoped, use the `Scope.DEFAULT` value for the `scope` property.

> warning **Notice** Websocket Gateways should not use request-scoped providers because they must act as singletons. Each gateway encapsulates a real socket and cannot be instantiated multiple times. The limitation also applies to some other providers, like *Passport strategies* or *Cron controllers*.

Controller scope

Controllers can also have scope, which applies to all request method handlers declared in that controller. Like provider scope, the scope of a controller declares its lifetime. For a request-scoped controller, a new instance is created for each inbound request, and garbage-collected when the request has completed processing.

Declare controller scope with the `scope` property of the `ControllerOptions` object:

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

Scope hierarchy

The `REQUEST` scope bubbles up the injection chain. A controller that depends on a request-scoped provider will, itself, be request-scoped.

Imagine the following dependency graph: `CatsController <- CatsService <- CatsRepository`. If `CatsService` is request-scoped (and the others are default singletons), the `CatsController` will become request-scoped as it is dependent on the injected service. The `CatsRepository`, which is not dependent, would remain singleton-scoped.

Transient-scoped dependencies don't follow that pattern. If a singleton-scoped `DogsService` injects a transient `LoggerService` provider, it will receive a fresh instance of it. However, `DogsService` will stay singleton-scoped, so injecting it anywhere would *not* resolve to a new instance of `DogsService`. In case it's desired behavior, `DogsService` must be explicitly marked as `TRANSIENT` as well.

Request provider

In an HTTP server-based application (e.g., using `@nestjs/platform-express` or `@nestjs/platform-fastify`), you may want to access a reference to the original request object when using request-scoped providers. You can do this by injecting the `REQUEST` object.

```
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 request: Request) {}
}
```

Because of underlying platform/protocol differences, you access the inbound request slightly differently for Microservice or GraphQL applications. In GraphQL applications, you inject `CONTEXT` instead of `REQUEST`:

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

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

You then configure your `context` value (in the `GraphQLModule`) to contain `request` as its property.

Inquirer provider

If you want to get the class where a provider was constructed, for instance in logging or metrics providers, you can inject the `INQUIRER` token.

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

@Injectable({ scope: Scope.TRANSIENT })
export class HelloService {
  constructor(@Inject(INQUIRER) private parentClass: object) {}

  sayHello(message: string) {
    console.log(`${this.parentClass?.constructor?.name}: ${message}`);
  }
}
```

And then use it as follows:

```
import { Injectable } from '@nestjs/common';
import { HelloService } from './hello.service';

@Injectable()
export class AppService {
  constructor(private helloService: HelloService) {}

  getRoot(): string {
    this.helloService.sayHello('My name is getRoot');

    return 'Hello world!';
  }
}
```

In the example above when `AppService#getRoot` is called, `"AppService: My name is getRoot"` will be logged to the console.

Performance

Using request-scoped providers will have an impact on application performance. While Nest tries 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. Unless a provider must be request-scoped, it is strongly recommended that you use the default singleton scope.

> info **Hint** Although it all sounds quite intimidating, a properly designed application that leverages request-scoped providers should not slow down by more than ~5% latency-wise.

Durable providers

Request-scoped providers, as mentioned in the section above, may lead to increased latency since having at least 1 request-scoped provider (injected into the controller instance, or deeper - injected into one of its providers) makes the controller request-scoped as well. That means, it must be recreated (instantiated) per each individual request (and garbage collected afterwards). Now, that also means, that for let's say 30k requests in parallel, there will be 30k ephemeral instances of the controller (and its request-scoped providers).

Having a common provider that most providers depend on (think of a database connection, or a logger service), automatically converts all those providers to request-scoped providers as well. This can pose a challenge in **multi-tenant applications**, especially for those that have a central request-scoped "data source" provider that grabs headers/token from the request object and based on its values, retrieves the corresponding database connection/schema (specific to that tenant).

For instance, let's say you have an application alternately used by 10 different customers. Each customer has its **own dedicated data source**, and you want to make sure customer A will never be able to reach customer B's database. One way to achieve this could be to declare a request-scoped "data source" provider that - based on the request object - determines what's the "current customer" and retrieves its corresponding database. With this approach, you can turn your application into a multi-tenant application in just a few minutes. But, a major downside to this approach is that since most likely a large chunk of your application' components rely on the "data source" provider, they will implicitly become "request-scoped", and therefore you will undoubtedly see an impact in your apps performance.

But what if we had a better solution? Since we only have 10 customers, couldn't we have 10 individual DI sub-trees per customer (instead of recreating each tree per request)? If your providers don't rely on any property that's truly unique for each consecutive request (e.g., request UUID) but instead there're some specific attributes that let us aggregate (classify) them, there's no reason to *recreate DI sub-tree* on every incoming request.

And that's exactly when the **durable providers** come in handy.

Before we start flagging providers as durable, we must first register a **strategy** that instructs Nest what are those "common request attributes", provide logic that groups requests - associates them with their corresponding DI sub-trees.

```
import {
  HostComponentInfo,
  ContextId,
  ContextIdFactory,
  ContextIdStrategy,
} from '@nestjs/core';
import { Request } from 'express';

const tenants = new Map<string, ContextId>();

export class AggregateByTenantContextIdStrategy implements
ContextIdStrategy {
  attach(contextId: ContextId, request: Request) {
    const tenantId = request.headers['x-tenant-id'] as string;
    let tenantSubTreeId: ContextId;

    if (tenants.has(tenantId)) {
      tenantSubTreeId = tenants.get(tenantId);
    } else {
      tenantSubTreeId = ContextIdFactory.create();
      tenants.set(tenantId, tenantSubTreeId);
    }

    // If tree is not durable, return the original "contextId" object
    return (info: HostComponentInfo) =>
      info.isTreeDurable ? tenantSubTreeId : contextId;
  }
}
```

> info **Hint** Similar to the request scope, durability bubbles up the injection chain. That means if A depends on B which is flagged as `durable`, A implicitly becomes durable too (unless `durable` is explicitly set to `false` for A provider).

> warning **Warning** Note this strategy is not ideal for applications operating with a large number of tenants.

The value returned from the `attach` method instructs Nest what context identifier should be used for a given host. In this case, we specified that the `tenantSubTreeId` should be used instead of the original, auto-generated `contextId` object, when the host component (e.g., request-scoped controller) is flagged as durable (you can learn how to mark providers as durable below). Also, in the above example, **no payload** would be registered (where payload = `REQUEST`/`CONTEXT` provider that represents the "root" - parent of the sub-tree).

If you want to register the payload for a durable tree, use the following construction instead:

```
// The return of `AggregateByTenantContextIdStrategy#attach` method:
return {
  resolve: (info: HostComponentInfo) =>
    info.isTreeDurable ? tenantSubTreeId : contextId,
  payload: { tenantId },
  }
```

Now whenever you inject the `REQUEST` provider (or `CONTEXT` for GraphQL applications) using the `@Inject(REQUEST)`/`@Inject(CONTEXT)`, the `payload` object would be injected (consisting of a single property - `tenantId` in this case).

Alright so with this strategy in place, you can register it somewhere in your code (as it applies globally anyway), so for example, you could place it in the `main.ts` file:

```
ContextIdFactory.apply(new AggregateByTenantContextIdStrategy());
```

> info **Hint** The `ContextIdFactory` class is imported from the `@nestjs/core` package.

As long as the registration occurs before any request hits your application, everything will work as intended.

Lastly, to turn a regular provider into a durable provider, simply set the `durable` flag to `true` and change its scope to `Scope.REQUEST` (not needed if the REQUEST scope is in the injection chain already):

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

@Injectable({ scope: Scope.REQUEST, durable: true })
export class CatsService {}
```

Similarly, for custom providers, set the `durable` property in the long-hand form for a provider registration:

```
{
  provide: 'foobar',
  useFactory: () => { ... },
  scope: Scope.REQUEST,
  durable: true,
}
```