Middleware

Middleware is a function which is called **before** the route handler. Middleware functions have access to the **request** and **response** objects, and the `next()` middleware function in the application's request-response cycle. The **next** middleware function is commonly denoted by a variable named `next`.

Nest middleware are, by default, equivalent to **express** middleware. The following description from the official express documentation describes the capabilities of middleware:

> Middleware functions can perform the following tasks:
>
> - execute any code.
> - make changes to the request and the response objects.
> - end the request-response cycle.
> - call the next middleware function in the stack.
> - if the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function. Otherwise, the request will be left hanging.

You implement custom Nest middleware in either a function, or in a class with an `@Injectable()` decorator. The class should implement the `NestMiddleware` interface, while the function does not have any special requirements. Let's start by implementing a simple middleware feature using the class method.

logger.middleware.ts JS

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: Function) {
```

```
    console.log('Request...');
    next();
  }
}
```

Dependency injection

Nest middleware fully supports Dependency Injection. Just as with providers and controllers, they are able to **inject dependencies** that are available within the same module. As usual, this is done through the `constructor`.

Applying middleware

There is no place for middleware in the `@Module()` decorator. Instead, we set them up using the `configure()` method of the module class. Modules that include middleware have to implement the `NestModule` interface. Let's set up the `LoggerMiddleware` at the `ApplicationModule` level.

app.module.ts JS

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('cats');
  }
}
```

In the above example we have set up the `LoggerMiddleware` for the `/cats` route handlers that were previously defined inside the `CatsController`. We may also further restrict a middleware to a particular request method by passing an object containing the route `path` and request `method` to the `forRoutes()` method when configuring the middleware. In the example below, notice that we import the `RequestMethod` enum to reference the desired request method type.

app.module.ts JS

```
import { Module, NestModule, RequestMethod, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
```

```
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET });
  }
}
```

Route wildcards

Pattern based routes are supported as well. For instance, the asterisk is used as a **wildcard**, and will match any combination of characters:

```
forRoutes({ path: 'ab*cd', method: RequestMethod.ALL });
```

The `'ab*cd'` route path will match `abcd`, `ab_cd`, `abecd`, and so on. The characters `?`, `+`, `*`, and `()` may be used in a route path, and are subsets of their regular expression counterparts. The hyphen (`-`) and the dot (`.`) are interpreted literally by string-based paths.

Middleware consumer

The `MiddlewareConsumer` is a helper class. It provides several built-in methods to manage middleware. All of them can be simply **chained** in the **fluent style**. The `forRoutes()` method can take a single string, multiple strings, a `RouteInfo` object, a controller class and even multiple controller classes. In most cases you'll probably just pass a list of **controllers** separated by commas. Below is an example with a single controller:

app.module.ts JS

```
import { Module, NestModule, MiddlewareConsumer } from '@nestjs/common';
import { LoggerMiddleware } from './common/middleware/logger.middleware';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class ApplicationModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
```

```
      .apply(LoggerMiddleware)
      .forRoutes(CatsController);
  }
}
```

> **HINT**
> The `apply()` method may either take a single middleware, or multiple arguments to specify **multiple middlewares**.

Quite often we might want to **exclude** certain routes from having the middleware applied. When defining middleware with a class (as we have been doing so far, as opposed to using the alternative **functional middleware**), we can easily exclude certain routes with the `exclude()` method. This method takes one or more objects identifying the `path` and `method` to be excluded, as shown below:

```
consumer
  .apply(LoggerMiddleware)
  .exclude(
    { path: 'cats', method: RequestMethod.GET },
    { path: 'cats', method: RequestMethod.POST }
  )
  .forRoutes(CatsController);
```

With the example above, `LoggerMiddleware` will be bound to all routes defined inside `CatsController` **except** the two passed to the `exclude()` method. Please note that the `exclude()` method **does not work** with functional middleware (middleware defined in a function rather than in a class; see below for more details). In addition, this method doesn't exclude paths from more generic routes (e.g., wildcards). If you need that level of control, you should put your paths-restriction logic directly into the middleware and, for example, access the request's URL to conditionally apply the middleware logic.

Functional middleware

The `LoggerMiddleware` class we've been using is quite simple. It has no members, no additional methods, and no dependencies. Why can't we just define it in a simple function instead of a class? In fact, we can. This type of middleware is called **functional middleware**. Let's transform the logger middleware from class-based into functional middleware to illustrate the difference:

logger.middleware.ts JS

```
export function logger(req, res, next) {
  console.log(`Request...`);
  next();
```

```
};
```

And use it within the `ApplicationModule`:

app.module.ts JS

```
consumer
  .apply(logger)
  .forRoutes(CatsController);
```

> **HINT**
> Consider using the simpler **functional middleware** alternative any time your middleware doesn't need any dependencies.

Multiple middleware

As mentioned above, in order to bind multiple middleware that are executed sequentially, simply provide a comma separated list inside the `apply()` method:

```
consumer.apply(cors(), helmet(), logger).forRoutes(CatsController);
```

Global middleware

If we want to bind middleware to every registered route at once, we can use the `use()` method that is supplied by the `INestApplication` instance:

```
const app = await NestFactory.create(ApplicationModule);
app.use(logger);
await app.listen(3000);
```

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

blueanchor.io

NOVOLOGIC Inc.

Designed by Jakub Staroń, hosted by Netlify