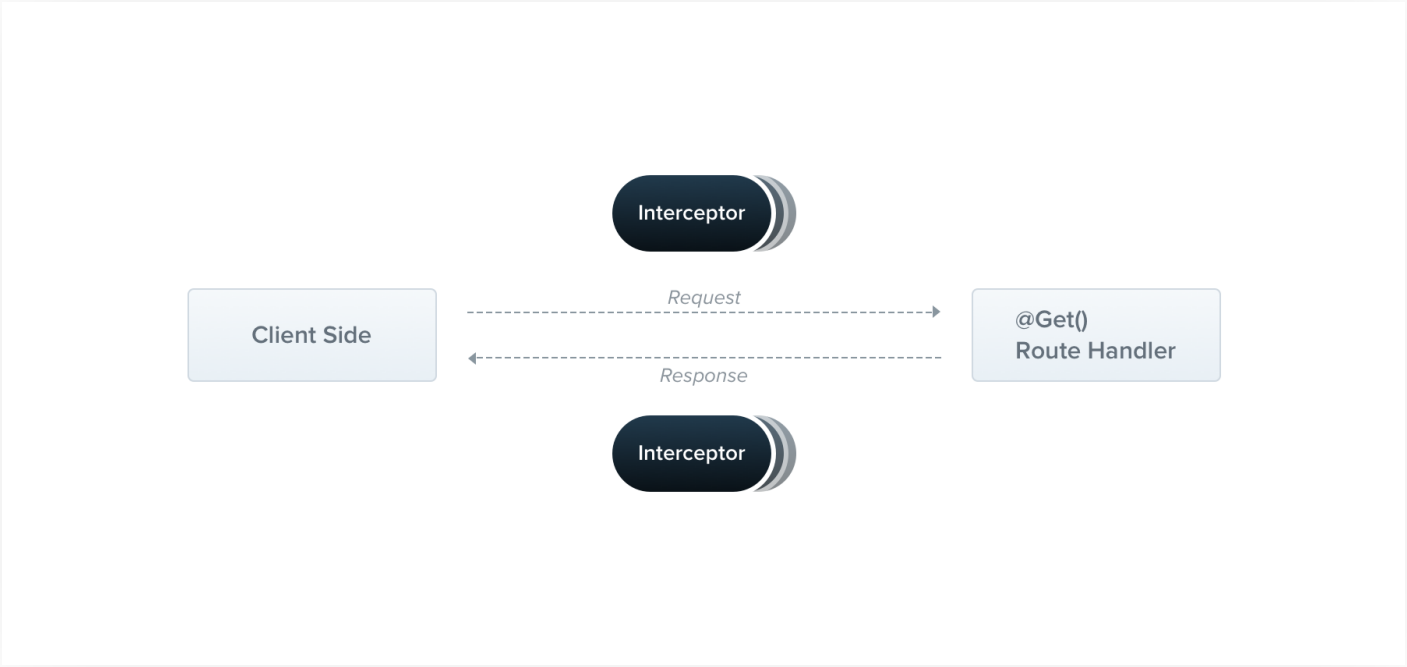
Interceptors

An interceptor is a class annotated with the `@Injectable()` decorator. Interceptors should implement the `NestInterceptor` interface.



Interceptors have a set of useful capabilities which are inspired by the **Aspect Oriented Programming** (AOP) technique. They make it possible to:

- bind **extra logic** before / after method execution
- **transform** the result returned from a function
- **transform** the exception thrown from a function
- **extend** the basic function behavior
- completely **override** a function depending on specific conditions (e.g., for caching purposes)

Basics

Each interceptor implements the `intercept()` method, which takes two arguments. The first one is the `ExecutionContext` instance (exactly the same object as for **guards**). The `ExecutionContext` inherits from `ArgumentsHost`. We saw `ArgumentsHost` before in the exception filters chapter. There, we saw that it's a wrapper around arguments that have been passed to the original handler, and contains different arguments arrays based on the type of the application. You can refer back to the **exception filters** for more on this topic.

Execution context

By extending `ArgumentsHost`, `ExecutionContext` provides additional details about the current execution process. Here's what it looks like:

```
export interface ExecutionContext extends ArgumentsHost {
  getClass<T = any>(): Type<T>;
  getHandler(): Function;
}
```

The `getHandler()` method returns a reference to the route handler about to be invoked. The `getClass()` method returns the type of the `Controller` class which this particular handler belongs to. For example, if the currently processed request is a `POST` request, destined for the `create()` method on the `CatsController`, `getHandler()` will return a reference to the `create()` method and `getClass()` will return a `CatsController` **type** (not instance).

Call handler

The second argument is a `CallHandler`. The `CallHandler` interface implements the `handle()` method, which you can use to invoke the route handler method at some point in your interceptor. If you don't call the `handle()` method in your implementation of the `intercept()` method, the route handler method won't be executed at all.

This approach means that the `intercept()` method effectively **wraps** the request/response stream. As a result, you may implement custom logic **both before and after** the execution of the final route handler. It's clear that you can write code in your `intercept()` method that executes **before** calling `handle()`, but how do you affect what happens afterward? Because the `handle()` method returns an `Observable`, we can use powerful **RxJS** operators to further manipulate the response. Using Aspect Oriented Programming terminology, the invocation of the route handler (i.e., calling `handle()`) is called a **Pointcut**, indicating that it's the point at which our additional logic is inserted.

Consider, for example, an incoming `POST /cats` request. This request is destined for the `create()` handler defined inside the `CatsController`. If an interceptor which does not call the `handle()` method is called anywhere along the way, the `create()` method won't be executed. Once `handle()` is called (and its `Observable` has been returned), the `create()` handler will be triggered. And once the response stream is received via the `Observable`, additional operations can be performed on the stream, and a final result returned to the caller.

Aspect interception

The first use case we'll look at is to use an interceptor to log user interaction (e.g., storing user calls, asynchronously dispatching events or calculating a timestamp). We show a simple `LoggingInterceptor` below:

logging.interceptor.ts JS

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
```

```
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}
```

HINT
The `NestInterceptor<T, R>` is a generic interface in which `T` indicates the type of an `Observable<T>` (supporting the response stream), and `R` is the type of the value wrapped by `Observable<R>`.

NOTICE
Interceptors, like controllers, providers, guards, and so on, can **inject dependencies** through their `constructor`.

Since `handle()` returns an RxJS `Observable`, we have a wide choice of operators we can use to manipulate the stream. In the example above, we used the `tap()` operator, which invokes our anonymous logging function upon graceful or exceptional termination of the observable stream, but doesn't otherwise interfere with the response cycle.

Binding interceptors

In order to set up the interceptor, we use the `@UseInterceptors()` decorator imported from the `@nestjs/common` package. Like **pipes** and **guards**, interceptors can be controller-scoped, method-scoped, or global-scoped.

cats.controller.ts JS

```
@UseInterceptors(LoggingInterceptor)
export class CatsController {}
```

HINT
The `@UseInterceptors()` decorator is imported from the `@nestjs/common` package.

Using the above construction, each route handler defined in `CatsController` will use `LoggingInterceptor`. When someone calls the `GET /cats` endpoint, you'll see the following output in your standard output:

```
Before...
After... 1ms
```

Note that we passed the `LoggingInterceptor` type (instead of an instance), leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes, guards, and exception filters, we can also pass an in-place instance:

cats.controller.ts JS

```
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

As mentioned, the construction above attaches the interceptor to every handler declared by this controller. If we want to restrict the interceptor's scope to a single method, we simply apply the decorator at the **method level**.

In order to set up a global interceptor, we use the `useGlobalInterceptors()` method of the Nest application instance:

```
const app = await NestFactory.create(ApplicationModule);
app.useGlobalInterceptors(new LoggingInterceptor());
```

Global interceptors are used across the whole application, for every controller and every route handler. In terms of dependency injection, global interceptors registered from outside of any module (with `useGlobalInterceptors()`, as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can set up an interceptor **directly from any module** using the following construction:

app.module.ts JS

```
import { Module } from '@nestjs/common';
import { APP_INTERCEPTOR } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_INTERCEPTOR,
      useClass: LoggingInterceptor,
    },
  ],
})
```

```
export class ApplicationModule {}
```

> **HINT**
> When using this approach to perform dependency injection for the interceptor, note that regardless of the module where this construction is employed, the interceptor is, in fact, global. Where should this be done? Choose the module where the interceptor (`LoggingInterceptor` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more **here**.

Response mapping

We already know that `handle()` returns an `Observable`. The stream contains the value **returned** from the route handler, and thus we can easily mutate it using RxJS's `map()` operator.

> **WARNING**
> The response mapping feature doesn't work with the library-specific response strategy (using the `@Res()` object directly is forbidden).

Let's create the `TransformInterceptor`, which will modify each response in a trivial way to demonstrate the process. It will use RxJS's `map()` operator to assign the response object to the `data` property of a newly created object, returning the new object to the client.

transform.interceptor.ts JS

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

export interface Response<T> {
  data: T;
}

@Injectable()
export class TransformInterceptor<T> implements NestInterceptor<T, Response<T>> {
  intercept(context: ExecutionContext, next: CallHandler): Observable<Response<T>> {
    return next.handle().pipe(map(data => ({ data })));
  }
}
```

> **HINT**

Nest interceptors work with both synchronous and asynchronous `intercept()` methods. You can simply switch the method to `async` if necessary.

With the above construction, when someone calls the `GET /cats` endpoint, the response would look like the following (assuming that route handler returns an empty array `[]`):

```
{
  "data": []
}
```

Interceptors have great value in creating re-usable solutions to requirements that occur across an entire application. For example, imagine we need to transform each occurrence of a `null` value to an empty string `''`. We can do it using one line of code and bind the interceptor globally so that it will automatically be used by each registered handler.

JS

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable()
export class ExcludeNullInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next
      .handle()
      .pipe(map(value => value === null ? '' : value ));
  }
}
```

Exception mapping

Another interesting use-case is to take advantage of RxJS's `catchError()` operator to override thrown exceptions:

errors.interceptor.ts JS

```
import {
  Injectable,
  NestInterceptor,
  ExecutionContext,
  BadGatewayException,
```

```
  CallHandler,
} from '@nestjs/common';
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';

@Injectable()
export class ErrorsInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next
      .handle()
      .pipe(
        catchError(err => throwError(new BadGatewayException())),
      );
  }
}
```

Stream overriding

There are several reasons why we may sometimes want to completely prevent calling the handler and return a different value instead. An obvious example is to implement a cache to improve response time. Let's take a look at a simple **cache interceptor** that returns its response from a cache. In a realistic example, we'd want to consider other factors like TTL, cache invalidation, cache size, etc., but that's beyond the scope of this discussion. Here we'll provide a basic example that demonstrates the main concept.

cache.interceptor.ts JS

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable, of } from 'rxjs';

@Injectable()
export class CacheInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const isCached = true;
    if (isCached) {
      return of([]);
    }
    return next.handle();
  }
}
```

Our `CacheInterceptor` has a hardcoded `isCached` variable and a hardcoded response `[]` as well. The key point to note is that we return a new stream here, created by the RxJS `of()` operator, therefore the route handler **won't be called** at all. When someone calls an endpoint that makes use of `CacheInterceptor`, the response (a hardcoded, empty array) will be returned immediately. In order to create a generic solution, you can take advantage of `Reflector` and create a

custom decorator. The `Reflector` is well described in the **guards** chapter.

More operators

The possibility of manipulating the stream using RxJS operators gives us many capabilities. Let's consider another common use case. Imagine you would like to handle **timeouts** on route requests. When your endpoint doesn't return anything after a period of time, you want to terminate with an error response. The following construction enables this:

timeout.interceptor.ts JS

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { timeout } from 'rxjs/operators';

@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next.handle().pipe(timeout(5000))
  }
}
```

After 5 seconds, request processing will be canceled.

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