Request lifecycle

Nest applications handle requests and produce responses in a sequence we refer to as the **request lifecycle**. With the use of middleware, pipes, guards, and interceptors, it can be challenging to track down where a particular piece of code executes during the request lifecycle, especially as global, controller level, and route level components come into play. In general, a request flows through middleware to guards, then to interceptors, then to pipes and finally back to interceptors on the return path (as the response is generated).

Middleware

Middleware is executed in a particular sequence. First, Nest runs globally bound middleware (such as middleware bound with `app.use`) and then it runs module bound middleware, which are determined on paths. Middleware are run sequentially in the order they are bound, similar to the way middleware in Express works. In the case of middleware bound across different modules, the middleware bound to the root module will run first, and then middleware will run in the order that the modules are added to the imports array.

Guards

Guard execution starts with global guards, then proceeds to controller guards, and finally to route guards. As with middleware, guards run in the order in which they are bound. For example:

```
@UseGuards(Guard1, Guard2)
@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @UseGuards(Guard3)
  @Get()
  getCats(): Cats[] {
    return this.catsService.getCats();
  }
}
```

`Guard1` will execute before `Guard2` and both will execute before `Guard3`.

> info **Hint** When speaking about globally bound vs controller or locally bound, the difference is where the guard (or other component is bound). If you are using `app.useGlobalGuard()` or providing the component via a module, it is globally bound. Otherwise, it is bound to a controller if the decorator precedes a controller class, or to a route if the decorator proceeds a route declaration.

Interceptors

Interceptors, for the most part, follow the same pattern as guards, with one catch: as interceptors return RxJS Observables, the observables will be resolved in a first in last out manner. So inbound requests will go through the standard global, controller, route level resolution, but the response side of the request (i.e., after returning from the controller method handler) will be resolved from route to controller to global. Also,

any errors thrown by pipes, controllers, or services can be read in the `catchError` operator of an interceptor.

Pipes

Pipes follow the standard global to controller to route bound sequence, with the same first in first out in regards to the `@UsePipes()` parameters. However, at a route parameter level, if you have multiple pipes running, they will run in the order of the last parameter with a pipe to the first. This also applies to the route level and controller level pipes. For example, if we have the following controller:

```
@UsePipes(GeneralValidationPipe)
@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @UsePipes(RouteSpecificPipe)
  @Patch(':id')
  updateCat(
    @Body() body: UpdateCatDTO,
    @Param() params: UpdateCatParams,
    @Query() query: UpdateCatQuery,
  ) {
    return this.catsService.updateCat(body, params, query);
  }
}
```

then the `GeneralValidationPipe` will run for the `query`, then the `params`, and then the `body` objects before moving on to the `RouteSpecificPipe`, which follows the same order. If any parameter-specific pipes were in place, they would run (again, from the last to first parameter) after the controller and route level pipes.

Filters

Filters are the only component that do not resolve global first. Instead, filters resolve from the lowest level possible, meaning execution starts with any route bound filters and proceeding next to controller level, and finally to global filters. Note that exceptions cannot be passed from filter to filter; if a route level filter catches the exception, a controller or global level filter cannot catch the same exception. The only way to achieve an effect like this is to use inheritance between the filters.

> info **Hint** Filters are only executed if any uncaught exception occurs during the request process. Caught exceptions, such as those caught with a `try/catch` will not trigger Exception Filters to fire. As soon as an uncaught exception is encountered, the rest of the lifecycle is ignored and the request skips straight to the filter.

Summary

In general, the request lifecycle looks like the following:

1. Incoming request

2. Middleware
    - 2.1. Globally bound middleware
    - 2.2. Module bound middleware
3. Guards
    - 3.1 Global guards
    - 3.2 Controller guards
    - 3.3 Route guards
4. Interceptors (pre-controller)
    - 4.1 Global interceptors
    - 4.2 Controller interceptors
    - 4.3 Route interceptors
5. Pipes
    - 5.1 Global pipes
    - 5.2 Controller pipes
    - 5.3 Route pipes
    - 5.4 Route parameter pipes
6. Controller (method handler)
7. Service (if exists)
8. Interceptors (post-request)
    - 8.1 Route interceptor
    - 8.2 Controller interceptor
    - 8.3 Global interceptor
9. Exception filters
    - 9.1 route
    - 9.2 controller
    - 9.3 global
10. Server response