nest

Custom route decorators

Nest is built around a language feature called **decorators**. It's a well-known concept in a lot of commonly used programming languages, but in the JavaScript world, it's still relatively new. In order to better understand how the decorators work, you should take a look at **this** article. Here's a simple definition:

> An ES2016 decorator is an expression which returns a function and can take a target, name and property descriptor as arguments. You apply it by prefixing the decorator with an `@` character and placing this at the very top of what you are trying to decorate. Decorators can be defined for either a class or a property.

Param decorators

Nest provides a set of useful **param decorators** that you can use together with the HTTP route handlers. Below is a comparison of the decorators with the plain express objects.

`@Request()`	`req`
`@Response()`	`res`
`@Next()`	`next`
`@Session()`	`req.session`
`@Param(param?: string)`	`req.params` / `req.params[param]`
`@Body(param?: string)`	`req.body` / `req.body[param]`
`@Query(param?: string)`	`req.query` / `req.query[param]`
`@Headers(param?: string)`	`req.headers` / `req.headers[param]`

Additionally, you can create your own, **custom decorator**. Why it is useful?

In the node.js world, it's a common practice to attach properties to the **request** object. Then you have to manually grab them every time in the route handlers, for example, using following construction:

```
const user = req.user;
```

In order to make it more readable and transparent, we can create a `@User()` decorator and reuse it across all existing controllers.

user.decorator.ts JS

```
import { createParamDecorator } from '@nestjs/common';

export const User = createParamDecorator((data, req) => {
  return req.user;
});
```

Then, you can simply use it wherever it fits your requirements.

JS

```
@Get()
async findOne(@User() user: UserEntity) {
  console.log(user);
}
```

Passing data

When the behavior of your decorator depends on some conditions, you may use the `data` param to pass an argument to the decorator's factory function. For example, the construction below:

JS

```
@Get()
async findOne(@User('test') user: UserEntity) {
  console.log(user);
}
```

Will make possible to access the `test` string via the `data` argument:

user.decorator.ts JS

```
import { createParamDecorator } from '@nestjs/common';

export const User = createParamDecorator((data: string, req) => {
  console.log(data); // test
  return req.user;
});
```

Working with pipes

Nest treats custom param decorators in the same fashion as the built-in ones (`@Body()`, `@Param()` and `@Query()`). It means that pipes are executed for the custom annotated parameters as well (in this case, for the `user` argument). Moreover, you can apply the pipe directly to the custom decorator:

JS

```
@Get()
async findOne(@User(new ValidationPipe()) user: UserEntity) {
  console.log(user);
}
```

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