Custom route decorators

Nest is built around a language feature called **decorators**. Decorators are a well-known concept in a lot of commonly used programming languages, but in the JavaScript world, they're still relatively new. In order to better understand how decorators work, we recommend reading 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 list of the provided decorators and the plain Express (or Fastify) objects they represent

`@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 decorators**. Why is this useful?

In the node.js world, it's common practice to attach properties to the **request** object. Then you manually extract them in each route handler, using code like the following:

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

In order to make your code more readable and transparent, you can create a `@User()` decorator and reuse it across all of your 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 can use the `data` parameter to pass an argument to the decorator's factory function. One use case for this is a custom decorator that extracts properties from the request object by key. Let's assume, for example, that our **authentication layer** validates requests and attaches a user entity to the request object. The user entity for an authenticated request might look like:

```
{
  "id": 101,
  "firstName": "Alan",
  "lastName": "Turing",
  "email": "alan@email.com",
  "roles": ["admin"]
}
```

Let's define a decorator that takes a property name as key, and returns the associated value if it exists (or undefined if it doesn't exist, or if the `user` object has not been created).

user.decorator.ts JS

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

export const User = createParamDecorator((data: string, req) => {
  return data ? req.user && req.user[data] : req.user;
});
```

Here's how you could then access a particular property via the `@User()` decorator in the controller:

JS

```
@Get()
async findOne(@User('firstName') firstName: string) {
  console.log(`Hello ${firstName}`);
}
```

You can use this same decorator with different keys to access different properties. If the `user` object is deep or complex, this can make for easier and more readable request handler implementations.

Working with pipes

Nest treats custom param decorators in the same fashion as the built-in ones (`@Body()` , `@Param()` and `@Query()`). This means that pipes are executed for the custom annotated parameters as well (in our examples, 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

Become a sponsor

Copyright © 2017-2019 MIT by Kamil Mysliwiec | design by Jakub Staron
Official NestJS Consulting Trilon.io | hosted by Netlify