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, a method 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()`	`req`
`@Response(), @Res()`	`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]`
`@Ip()`	`req.ip`
`@HostParam()`	`req.hosts`

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.

```
@@filename(user.decorator)
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
```

```
  (data: unknown, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    return request.user;
  },
);
```

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

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

```
@@filename(user.decorator)
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const User = createParamDecorator(
  (data: string, ctx: ExecutionContext) => {
    const request = ctx.switchToHttp().getRequest();
    const user = request.user;
```

```
    return data ? user?.[data] : user;
  },
);
@@switch
import { createParamDecorator } from '@nestjs/common';

export const User = createParamDecorator((data, ctx) => {
  const request = ctx.switchToHttp().getRequest();
  const user = request.user;

  return data ? user && user[data] : user;
});
```

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

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

> info **Hint** For TypeScript users, note that `createParamDecorator<T>()` is a generic. This means you can explicitly enforce type safety, for example `createParamDecorator<string>((data, ctx) => ...)`. Alternatively, specify a parameter type in the factory function, for example `createParamDecorator((data: string, ctx) => ...)`. If you omit both, the type for `data` will be `any`.

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:

```
@@filename()
@Get()
async findOne(
  @User(new ValidationPipe({ validateCustomDecorators: true }))
  user: UserEntity,
```

```
) {
  console.log(user);
}
@@switch
@Get()
@Bind(User(new ValidationPipe({ validateCustomDecorators: true })))
async findOne(user) {
  console.log(user);
}
```

> info **Hint** Note that `validateCustomDecorators` option must be set to true. `ValidationPipe` does not validate arguments annotated with the custom decorators by default.

Decorator composition

Nest provides a helper method to compose multiple decorators. For example, suppose you want to combine all decorators related to authentication into a single decorator. This could be done with the following construction:

```
@@filename(auth.decorator)
import { applyDecorators } from '@nestjs/common';

export function Auth(...roles: Role[]) {
  return applyDecorators(
    SetMetadata('roles', roles),
    UseGuards(AuthGuard, RolesGuard),
    ApiBearerAuth(),
    ApiUnauthorizedResponse({ description: 'Unauthorized' }),
  );
}
@@switch
import { applyDecorators } from '@nestjs/common';

export function Auth(...roles) {
  return applyDecorators(
    SetMetadata('roles', roles),
    UseGuards(AuthGuard, RolesGuard),
    ApiBearerAuth(),
    ApiUnauthorizedResponse({ description: 'Unauthorized' }),
  );
}
```

You can then use this custom `@Auth()` decorator as follows:

```
@Get('users')
@Auth('admin')
findAllUsers() {}
```

This has the effect of applying all four decorators with a single declaration.

> warning **Warning** The `@ApiHideProperty()` decorator from the `@nestjs/swagger` package is not composable and won't work properly with the `applyDecorators` function.