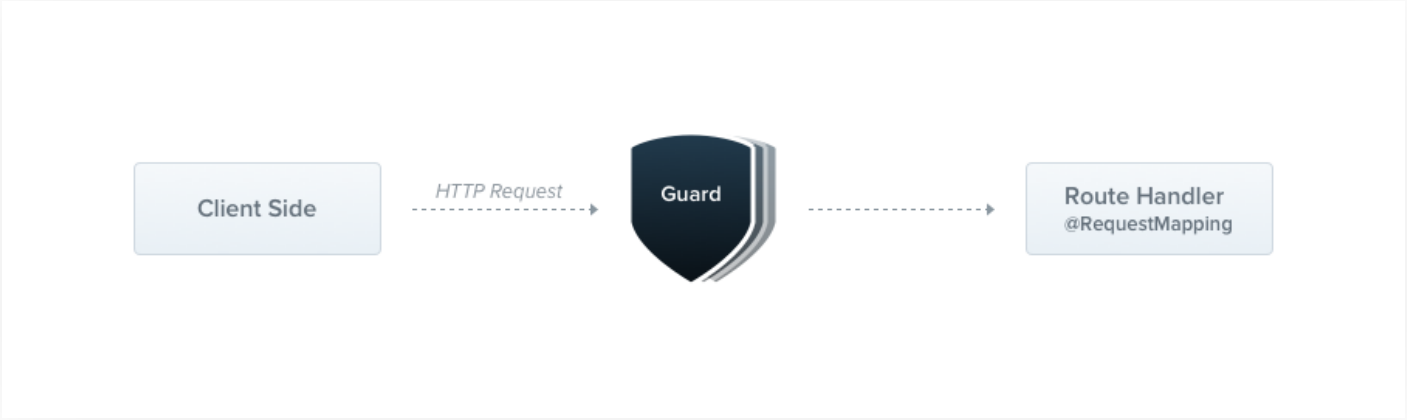
Guards

A guard is a class annotated with the `@Injectable()` decorator. Guards should implement the `CanActivate` interface.



Guards have a **single responsibility**. They determine whether a given request will be handled by the route handler or not, depending on certain conditions (like permissions, roles, ACLs, etc.) present at run-time. This is often referred to as **authorization**. Authorization (and its cousin, **authentication**, with which it usually collaborates) has typically been handled by **middleware** in traditional Express applications. Middleware is a fine choice for authentication, since things like token validation and attaching properties to the `request` object are not strongly connected with a particular route context (and its metadata).

But middleware, by its nature, is dumb. It doesn't know which handler will be executed after calling the `next()` function. On the other hand, **Guards** have access to the `ExecutionContext` instance, and thus know exactly what's going to be executed next. They're designed, much like exception filters, pipes, and interceptors, to let you interpose processing logic at exactly the right point in the request/response cycle, and to do so declaratively. This helps keep your code DRY and declarative.

> **HINT**
> Guards are executed **after** each middleware, but **before** any interceptor or pipe.

Authorization guard

As mentioned, **authorization** is a great use case for Guards because specific routes should be available only when the caller (usually a specific authenticated user) has sufficient permissions. The `AuthGuard` that we'll build now assumes an authenticated user (and that, therefore, a token is attached to the request headers). It will extract and validate the token, and use the extracted information to determine whether the request can proceed or not.

auth.guard.ts JS

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

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}
```

The logic inside the `validateRequest()` function can be as simple or sophisticated as needed. The main point of this example is to show how guards fit into the request/response cycle.

Every guard must implement a `canActivate()` function. This function should return a boolean, indicating whether the current request is allowed or not. It can return the response either synchronously or asynchronously (via a `Promise` or `Observable`). Nest uses the return value to control the next action:

- if it returns `true`, the request will be processed.
- if it returns `false`, Nest will deny the request.

The `canActivate()` function takes a single argument, the `ExecutionContext` instance. 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 chapter** 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 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).

Role-based authentication

Let's build a more functional guard that permits access only to users with a specific role. We'll start with a basic guard template, and build on it in the coming sections. For now, it allows all requests to proceed:

roles.guard.ts JS

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

@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    return true;
  }
}
```

Binding guards

Like pipes and exception filters, guards can be **controller-scoped**, method-scoped, or global-scoped. Below, we set up a controller-scoped guard using the `@UseGuards()` decorator. This decorator may take a single argument, or a comma-separated list of arguments. This lets you easily apply the appropriate set of guards with one declaration.

JS

```
@Controller('cats')
@UseGuards(RolesGuard)
export class CatsController {}
```

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

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

JS

```
@Controller('cats')
```

```
@UseGuards(new RolesGuard())
export class CatsController {}
```

The construction above attaches the guard to every handler declared by this controller. If we wish the guard to apply only to a single method, we apply the `@UseGuards()` decorator at the **method level**.

In order to set up a global guard, use the `useGlobalGuards()` method of the Nest application instance:

JS

```
const app = await NestFactory.create(ApplicationModule);
app.useGlobalGuards(new RolesGuard());
```

NOTICE
In the case of hybrid apps the `useGlobalGuards()` method doesn't set up guards for gateways and micro services. For "standard" (non-hybrid) microservice apps, `useGlobalGuards()` does mount the guards globally.

Global guards are used across the whole application, for every controller and every route handler. In terms of dependency injection, global guards registered from outside of any module (with `useGlobalGuards()` 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 a guard directly from any module using the following construction:

app.module.ts JS

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

@Module({
  providers: [
    {
      provide: APP_GUARD,
      useClass: RolesGuard,
    },
  ],
})
export class ApplicationModule {}
```

HINT

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

Reflection

Our `RolesGuard` is working, but it's not very smart yet. We're not yet taking advantage of the most important guard feature - the **execution context**. It doesn't yet know about roles, or which roles are allowed for each handler. The `CatsController` , for example, could have different permission schemes for different routes. Some might be available only for an admin user, and others could be open for everyone. How can we match roles to routes in a flexible and reusable way?

This is where **custom metadata** comes into play. Nest provides the ability to attach custom **metadata** to route handlers through the `@SetMetadata()` decorator. This metadata supplies our missing `role` data, which a smart guard needs to make decisions. Let's take a look at using `@SetMetadata()` :

cats.controller.ts JS

```
@Post()
@SetMetadata('roles', ['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

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

With the construction above, we attached the `roles` metadata (`roles` is a key, while `['admin']` is a particular value) to the `create()` method. While this works, it's not good practice to use `@SetMetadata()` directly in your routes. Instead, create your own decorators, as shown below:

roles.decorator.ts JS

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

export const Roles = (...roles: string[]) => SetMetadata('roles', roles);
```

This approach is much cleaner and more readable, and is strongly typed. Now that we have a custom `@Roles()` decorator, we can use it to decorate the `create()` method.

cats.controller.ts JS

```
@Post()
@Roles('admin')
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Putting it all together

Let's now go back and tie this together with our `RolesGuard`. Currently, it simply returns `true` in all cases, allowing every request to proceed. We want to make the return value conditional based on the comparing the **roles assigned to the current user** to the actual roles required by the current route being processed. In order to access the route's role(s) (custom metadata), we'll use the `Reflector` helper class, which is provided out of the box by the framework and exposed from the `@nestjs/core` package.

roles.guard.ts JS

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs';
import { Reflector } from '@nestjs/core';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private readonly reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<string[]>('roles', context.getHandler());
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    const hasRole = () => user.roles.some((role) => roles.includes(role));
    return user && user.roles && hasRole();
  }
}
```

HINT

In the node.js world, it's common practice to attach the authorized user to the `request` object. Thus, in our sample code above, we are assuming that `request.user` contains the user instance and allowed roles. In your app, you

will probably make that association in your custom **authentication guard** (or middleware).

The `Reflector` class allows us to easily access the metadata by the specified **key** (in this case, the key is `'roles'`; refer back to the `roles.decorator.ts` file and the `SetMetadata()` call made there). In the example above, we passed `context.getHandler()` in order to extract the metadata for the currently processed request method. Remember, `getHandler()` gives us a **reference** to the route handler function.

We can make this guard more generic by extracting the **controller metadata** and using that to determine the current user role. To extract controller metadata, we pass `context.getClass()` instead of `context.getHandler()`:

```
JS
```

```
const roles = this.reflector.get<string[]>('roles', context.getClass());
```

When a user with insufficient privileges requests an endpoint, Nest automatically returns the following response:

```
{
  "statusCode": 403,
  "message": "Forbidden resource"
}
```

Note that behind the scenes, when a guard returns `false`, the framework throws a `ForbiddenException`. If you want to return a different error response, you should throw your own specific exception. For example:

```
throw new UnauthorizedException();
```

Any exception thrown by a guard will be handled by the **exceptions layer** (global exceptions filter and any exceptions filters that are applied to the current context).

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

blueanchor.io

NOVOLOGIC Inc.

Designed by Jakub Staroń, hosted by Netlify