

## Guards

A guard is a class annotated with the `@Injectable()` decorator, which implements 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.

**info Hint** Guards are executed **after** all 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.

```
@@filename(auth.guard)
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);
  }
}

@@switch
import { Injectable } from '@nestjs/common';

@Injectable()
export class AuthGuard {
  async canActivate(context) {
```

```

    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
  }
}

```

**info Hint** If you are looking for a real-world example on how to implement an authentication mechanism in your application, visit [this chapter](#). Likewise, for more sophisticated authorization example, check [this page](#).

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.

## Execution context

The `canActivate()` function takes a single argument, the `ExecutionContext` instance. The `ExecutionContext` inherits from `ArgumentsHost`. We saw `ArgumentsHost` previously in the exception filters chapter. In the sample above, we are just using the same helper methods defined on `ArgumentsHost` that we used earlier, to get a reference to the `Request` object. You can refer back to the **Arguments host** section of the [exception filters](#) chapter for more on this topic.

By extending `ArgumentsHost`, `ExecutionContext` also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in building more generic guards that can work across a broad set of controllers, methods, and execution contexts. Learn more about `ExecutionContext` [here](#).

## 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:

```

@@filename(roles.guard)
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;
  }
}

```

```
}  
@@switch  
import { Injectable } from '@nestjs/common';  
  
@Injectable()  
export class RolesGuard {  
  canActivate(context) {  
    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.

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

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

Above, we passed the `RolesGuard` class (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:

```
@@filename()  
@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:

```
@@filename()  
const app = await NestFactory.create(AppModule);  
app.useGlobalGuards(new RolesGuard());
```

**warning Notice** In the case of hybrid apps the `useGlobalGuards()` method doesn't set up guards for gateways and micro services by default (see [Hybrid application](#) for information on how to change

this behavior). 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:

```
@@filename(app.module)
import { Module } from '@nestjs/common';
import { APP_GUARD } from '@nestjs/core';

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

info **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](#).

## Setting roles per handler

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 (learn more [here](#)). Nest provides the ability to attach custom **metadata** to route handlers through either decorators created via `Reflector#createDecorator` static method, or the built-in `@SetMetadata()` decorator.

For example, let's create a `@Roles()` decorator using the `Reflector#createDecorator` method that will attach the metadata to the handler. `Reflector` is provided out of the box by the framework and exposed from the `@nestjs/core` package.

```
@@filename(roles.decorator)
import { Reflector } from '@nestjs/core';
```

```
export const Roles = Reflector.createDecorator<string[]>();
```

The `Roles` decorator here is a function that takes a single argument of type `string[]`.

Now, to use this decorator, we simply annotate the handler with it:

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

@@switch
@Post()
@Roles(['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

Here we've attached the `Roles` decorator metadata to the `create()` method, indicating that only users with the `admin` role should be allowed to access this route.

Alternatively, instead of using the `Reflector#createDecorator` method, we could use the built-in `@SetMetadata()` decorator. Learn more about [here](#).

## 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 again, as follows:

```
@@filename(roles.guard)
import { Injectable, CanActivate, ExecutionContext } from
  '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Roles } from './roles.decorator';

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

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get(Roles, context.getHandler());
    if (!roles) {
```

```

        return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return matchRoles(roles, user.roles);
}
}

@switch
import { Injectable, Dependencies } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Roles } from './roles.decorator';

@Injectable()
@Dependencies(Reflector)
export class RolesGuard {
    constructor(reflector) {
        this.reflector = reflector;
    }

    canActivate(context) {
        const roles = this.reflector.get(Roles, context.getHandler());
        if (!roles) {
            return true;
        }
        const request = context.switchToHttp().getRequest();
        const user = request.user;
        return matchRoles(roles, user.roles);
    }
}

```

info **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). Check [this chapter](#) for more information on this topic.

warning **Warning** The logic inside the `matchRoles()` 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.

Refer to the [Reflection and metadata](#) section of the **Execution context** chapter for more details on utilizing `Reflector` in a context-sensitive way.

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

```

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

```

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).

**Hint** If you are looking for a real-world example on how to implement authorization, check [this chapter](#).