

What is a Guard (in simple words)?

Think of a **Guard** as a security gatekeeper .

Before a request reaches your **controller method**, the guard decides:

 “Should this request be allowed to go inside or not?”

If **YES** → controller runs

If **NO** → request is blocked (403 / 401)

Why Guards exist (and middleware is not enough)

Middleware (old way)

- Runs **before everything**
- Does **not know** which route or controller is being called
- Good for:
 - parsing tokens
 - logging
 - attaching `req.user`

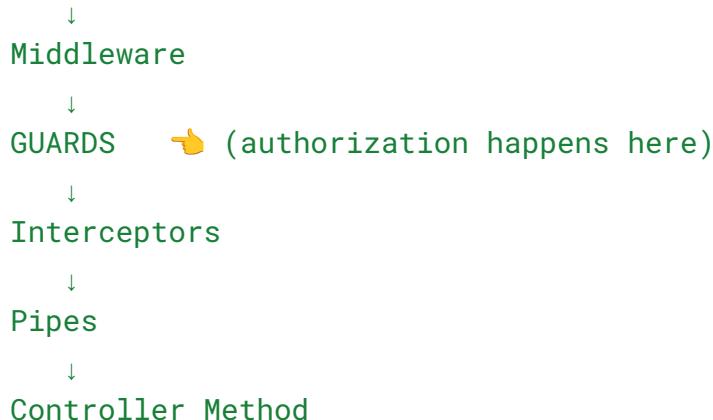
Guards (NestJS way)

- Runs **after middleware**
- Knows:
 - which controller
 - which route
 - which method
 - what metadata (roles, permissions) is attached

📌 Guards are smart. Middleware is dumb.

➡ Request Flow (IMPORTANT)

Client Request



🧱 Guard Structure (Minimum Requirement)

Every guard:

- Is a **class**
- Uses **@Injectable()**
- Implements **CanActivate**
- Has **one method** → **canActivate()**

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    return true;
}
```

Meaning:

- `true` → allow request
 - `false` → block request
-



ExecutionContext (Why Guards are powerful)

`ExecutionContext` tells the guard:

- Which **route** is being called
- Which **controller method**
- What **metadata** is attached
- Gives access to `request`, `response`

```
const request = context.switchToHttp().getRequest();
```

Now you can:

- Read headers
 - Read `request.user`
 - Check roles
 - Check permissions
-



Auth Guard (Simple Explanation)

Purpose:

“Is this user logged in?”

```
canActivate(context: ExecutionContext) {
  const request = context.switchToHttp().getRequest();
  return validateRequest(request);
}
```

- Token exists?
- Token valid?
- User attached to request?

✓ Yes → allow
✗ No → deny

🎭 Role-Based Guard (Real-world thinking)

Scenario:

- `/admin` → only admin
- `/profile` → any logged-in user
- `/posts` → public

We need:

- A way to **define roles per route**
 - A guard to **check those roles**
-

🏷️ Roles Decorator (Metadata)

This line means:

“This route requires ADMIN role”

```
@Roles(['admin'])  
@Post()  
create() {}
```

Internally:

- This stores metadata on the route
 - Guard can read it later
-



RolesGuard – How it works (Plain English)

```
const roles = this.reflector.get(Roles, context.getHandler());
```

Means:

“Give me the roles required for this route”

If no roles:

```
return true; // open route
```

If roles exist:

```
const user = request.user;
return matchRoles(roles, user.roles);
```

Meaning:

“Does the logged-in user have at least one required role?”



Full Mental Model (Very Important)

Think like this:

1. Middleware

- Extract token
- Validate token
- Attach `request.user`

2. Guard

- Read required roles from route
 - Compare with `request.user.roles`
 - Allow or deny
-

Where can you apply Guards?

1 Method level

```
@UseGuards(RolesGuard)  
@Get()  
findAll() {}
```

2 Controller level

```
@UseGuards(RolesGuard)  
@Controller('cats')
```

3 Global level

```
{  
  provide: APP_GUARD,  
  useClass: RolesGuard,  
}
```

Best practice:

Use `APP_GUARD` for role/auth guards → supports DI

What happens when Guard blocks request?

Nest automatically throws:

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

You can override it:

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



One-Line Summary (Remember This)

Guards answer one question only:

? “Should this request reach the controller or not?”



Real Project Analogy

Think of your office:

Concept	NestJS
ID Card check	Auth Guard
Role check (Admin / Intern)	Roles Guard
Entry allowed	Controller
Entry denied	403 / 401



When YOU should use Guards

Use Guards for:

- Authentication (JWT, session)
- Role-based access
- Permissions (RBAC)
- Feature flags
- Subscription checks



Don't use Guards for:

- Request validation → Pipes
- Response shaping → Interceptors
- Error formatting → Exception Filters