🧠 What is a Pipe in NestJS? (Plain English)

A **Pipe** is a **function/class that runs BEFORE your controller method**.

It sits **between the incoming request and your controller** and is used to:

1. **Transform data** (change format/type)
2. **Validate data** (reject bad input early)

If a pipe fails → **controller method NEVER runs**

Think of a pipe as a **security + data-cleaning gate** 🚪

📍 Where Pipes Run in Request Flow

```
Client Request
   ↓
Guards
   ↓
Pipes   👈 (THIS TOPIC)
   ↓
Controller Method
   ↓
Service
```

So pipes protect your **system boundary** (best practice).

🎯 Two Main Use Cases of Pipes

1️⃣ Transformation (change data)

Example:

- `"123"` → `123`
- `"true"` → `true`

- `"2026-01-01"` → `Date`

2️⃣ Validation (reject bad data)

Example:

- Missing required fields
- Wrong type
- Invalid UUID
- Invalid enum value

✅ Built-in Pipes (Most Important Ones)

NestJS already gives you many pipes from `@nestjs/common`:

Pipe	What it does
`ParseIntPipe`	string → number
`ParseBoolPipe`	string → boolean
`ParseUUIDPipe`	validates UUID
`ParseEnumPipe`	validates enum
`DefaultValuePipe`	supplies default
`ValidationPipe`	DTO validation

> You use **built-in pipes 80% of the time** in real projects.

🔹 Example 1: `ParseIntPipe` (MOST COMMON)

❌ Without pipe

```
@Get(':id')
```

```
findOne(@Param('id') id: number) {
  // id is actually a STRING 😬
}
```

✅ With pipe

```
@Get(':id')
findOne(@Param('id', ParseIntPipe) id: number) {
  // id is guaranteed to be NUMBER
}
```

If user calls:

```
GET /cats/abc
```

Nest returns:

```
{
  "statusCode": 400,
  "message": "Validation failed (numeric string is expected)"
}
```

👉 Controller never runs
👉 Safe code
👉 No manual checks

🔧 Passing Options to a Pipe

```
@Get(':id')
findOne(
  @Param('id', new ParseIntPipe({
    errorHttpStatusCode: 406
  })) id: number
) {}
```

Here:

- You create a **pipe instance**

- You customize its behavior

🔹 Pipes Work on:

Source	Decorator
URL params	`@Param()`
Query params	`@Query()`
Body	`@Body()`

Example with query:

```
@Get()
find(@Query('page', ParseIntPipe) page: number) {}
```

🧪 Validation Pipes (Big Concept)

❓ Why Validation Pipe Exists

You **should NOT** validate inside controller:

`if (!body.name || !body.age)` ❌

Why?

- Violates **Single Responsibility Principle**
- Repeated code
- Hard to maintain

👉 Pipes solve this cleanly.

🧱 DTO + Validation (Real-World Standard)

DTO with decorators

```
export class CreateCatDto {
  @IsString()
  name: string;

  @IsInt()
  age: number;

  @IsString()
  breed: string;
}
```

Controller

```
@Post()
create(@Body() dto: CreateCatDto) {}
```

Global Validation Pipe (BEST PRACTICE)

```
app.useGlobalPipes(new ValidationPipe());
```

Now:

- ❌ Invalid body → rejected
- ✅ Valid body → controller runs

🌍 Pipe Scopes (VERY IMPORTANT)

Scope	Applies To
Parameter	One parameter
Method	One route
Controller	All routes in controller
Global	Entire app

Parameter-level pipe

```
@Param('id', ParseIntPipe)
```

Method-level pipe

```
@UsePipes(new ValidationPipe())
@Post()
create() {}
```

Global pipe

```
app.useGlobalPipes(new ValidationPipe());
```

⚠️ Important Global Pipe DI Rule

❌ This cannot inject services:

```
app.useGlobalPipes(new MyPipe());
```

✅ Correct way (with DI):

```
providers: [
  {
    provide: APP_PIPE,
    useClass: ValidationPipe,
  },
]
```

🔄 Custom Pipe (Simple Mental Model)

Every pipe:

```
class MyPipe implements PipeTransform {
  transform(value, metadata) {
    // change or validate value
    return value; // or throw error
  }
}
```

If you throw error → request stops

If you return value → controller runs

🔁 Transformation Example (Custom ParseInt)

```
transform(value: string) {
  const num = parseInt(value);
  if (isNaN(num)) throw new BadRequestException();
  return num;
}
```

🎁 DefaultValuePipe (Hidden Gem ⭐)

Problem:

```
/cats?page=
```

Solution:

```
@Query('page',
  new DefaultValuePipe(0),
  ParseIntPipe
) page: number
```

Flow:

```
undefined → 0 → parsed to number
```

🧠 One-Line Summary (Remember This)

> **Pipes are the gatekeepers of NestJS — they clean, validate, and transform incoming data before it touches your business logic.**