1️⃣ What is an Exception Filter (in simple words)?

👉 **An Exception Filter is NestJS's error handler**

- Whenever **something goes wrong (an error is thrown)**
- NestJS **catches it automatically**
- Converts it into a **proper HTTP response**
- Sends it back to the client

Think of it like a **safety net** 🪢 for your application.

2️⃣ Built-in Exception Handling (Default Behavior)

NestJS already has a **global exception filter** built in.

✅ If you throw an `HttpException`

NestJS knows how to handle it:

```
throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
```

Client gets:

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

❌ If you throw an unknown error

NestJS sends a default response:

```
{
  "statusCode": 500,
  "message": "Internal server error"
}
```

👉 This is why **you should always throw proper HTTP exceptions**

3️⃣ Why HttpException exists?

`HttpException` lets you:

- Control **status code**
- Control **error message**
- Control **response body**

Constructor

```
new HttpException(response, status, options?)
```

- **response** → string or object (JSON body)
- **status** → HTTP status code
- **options** → extra info (like error cause, for logging)

4️⃣ Custom Response Body

Override only message

```
throw new HttpException('Not Allowed', HttpStatus.FORBIDDEN);
```

Override full response

```
throw new HttpException(
  {
    status: HttpStatus.FORBIDDEN,
    error: 'Custom message',
  },
  HttpStatus.FORBIDDEN
);
```

Response:

```
{
  "status": 403,
  "error": "Custom message"
}
```

5️⃣ Built-in HTTP Exceptions (Use These!)

NestJS already gives you **ready-made exceptions** 👌

Examples:

```
throw new BadRequestException();
throw new UnauthorizedException();
throw new NotFoundException();
throw new ForbiddenException();
```

👉 **Best practice**:
Use these instead of `HttpException` directly

6️⃣ Why exceptions are NOT logged by default?

NestJS treats these as **normal application flow**, not bugs.

- `HttpException`
- `WsException`
- `RpcException`

All extend **IntrinsicException**

👉 If you want **logging**, you need a **custom exception filter**

7️⃣ Custom Exception (Your own error class)

When you want **domain-specific errors**

```
export class ForbiddenException extends HttpException {
  constructor() {
    super('Forbidden', HttpStatus.FORBIDDEN);
  }
}
```

Usage:

```
throw new ForbiddenException();
```

✔ Clean
✔ Reusable
✔ Professional

8️⃣ What is a Custom Exception Filter?

👉 It lets you **fully control**:

- Error response format
- Logging
- Extra fields (timestamp, path, request info)

9️⃣ Simple Custom Exception Filter (Core Idea)

```
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const request = ctx.getRequest();

    const status = exception.getStatus();

    response.status(status).json({
      statusCode: status,
```

```
      timestamp: new Date().toISOString(),
      path: request.url,
    });
  }
}
```

What happens here?

- `@Catch(HttpException)` → catch only HTTP errors
- `ArgumentsHost` → gives access to request & response
- You **manually send the response**

🔟 What is ArgumentsHost?

👉 A **wrapper** that works in:

- HTTP
- WebSockets
- Microservices

It helps you write **platform-independent code**

```
host.switchToHttp().getRequest()
host.switchToHttp().getResponse()
```

1️⃣1️⃣ Where can we apply exception filters?

1. Method-level

```
@UseFilters(HttpExceptionFilter)
@Post()
create() {}
```

2. Controller-level

```
@UseFilters(HttpExceptionFilter)
@Controller()
export class CatsController {}
```

3. Global-level (Entire App)

```
app.useGlobalFilters(new HttpExceptionFilter());
```

⚠️ Global filters registered this way **cannot use DI**

1️⃣2️⃣ Global Filter with Dependency Injection (Correct Way)

```
providers: [
  {
    provide: APP_FILTER,
    useClass: HttpExceptionFilter,
  },
]
```

✔ Supports DI
✔ Best practice for real apps

1️⃣3️⃣ Catch EVERYTHING Filter

```
@Catch()
export class CatchEverythingFilter implements ExceptionFilter {}
```

- Catches **all errors**
- Even non-HTTP ones

⚠️ Must be declared **before specific filters**

1️⃣4️⃣ Extending Default NestJS Filter

When you want **default behavior + customization**

```
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception, host) {
    super.catch(exception, host);
  }
}
```

👉 Use this when:

- You want default responses
- But add logging / metrics / monitoring

🧠 Final Mental Model (VERY IMPORTANT)

```
Controller/Service throws error
        ↓
NestJS Exception Layer
        ↓
Exception Filter decides:
  - status code
  - response format
  - logging
        ↓
Client receives response
```

✅ Best Practices Summary

✔ Use **built-in exceptions**
✔ Create **custom exceptions** for business logic
✔ Use **global exception filters** for consistency
✔ Use `APP_FILTER` for dependency injection
✔ Never return raw errors to client