🧠 What is an Interceptor? (In plain English)

Think of an **interceptor as a wrapper around your API method**.

```
Request → Interceptor (before) → Controller Method → Interceptor (after) → Response
```

It **sits between**:

- the incoming request
- and the outgoing response

Just like:

- **Middleware** → runs before request (but dumb)
- **Guard** → allows or blocks request
- **Pipe** → transforms input
- ✅ **Interceptor** → can touch *both request AND response*

🎯 What Interceptors Can Do (Memory Trick: B-T-T-E-O)

Interceptors allow you to:

1. **B**efore logic → run code before controller method
2. **T**ransform response → modify returned data
3. **T**ransform exceptions → catch & replace errors
4. **E**xtend behavior → logging, metrics, timing
5. **O**verride handler → skip controller (cache, mock data)

🏗️ Basic Structure of an Interceptor

```
@Injectable()
export class MyInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler):
Observable<any> {
    // before logic
    return next.handle().pipe(
      // after logic
    );
  }
}
```

Key things:

- `@Injectable()` → enables DI
- `ExecutionContext` → info about request
- `CallHandler` → controls controller execution
- `handle()` → actually calls the controller method

❗ **If you don't call `next.handle()` → controller never runs**

🔍 ExecutionContext (Very Important)

`ExecutionContext` tells you:

- Which controller
- Which method
- HTTP / GraphQL / RPC context
- Request & Response objects

Example:

```
const req = context.switchToHttp().getRequest();
const res = context.switchToHttp().getResponse();
```

📌 Same object used in **Guards** & **Filters**

🔁 Why RxJS is Used Here?

`next.handle()` returns an **Observable**.

This lets you:

- modify response
- catch errors
- timeout requests
- short-circuit execution

Think:

> **Observable = response stream**

📝 Example 1: Logging Interceptor (Before + After)

```
@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler):
Observable<any> {
    console.log('Before...');

    const now = Date.now();
    return next.handle().pipe(
      tap(() => console.log(`After... ${Date.now() - now}ms`)),
    );
  }
}
```

What happens?

1. Logs **before controller**

2. Calls controller
3. Logs **after response**
4. Measures execution time

🧠 `tap()` → observe without changing data

🔗 Binding Interceptors

1️⃣ Method level

```
@UseInterceptors(LoggingInterceptor)
@Get()
findAll() {}
```

2️⃣ Controller level

```
@UseInterceptors(LoggingInterceptor)
@Controller('cats')
export class CatsController {}
```

3️⃣ Global (no DI)

```
app.useGlobalInterceptors(new LoggingInterceptor());
```

❌ Cannot inject services here

4️⃣ Global (with DI ✅ BEST PRACTICE)

```
providers: [
  {
    provide: APP_INTERCEPTOR,
    useClass: LoggingInterceptor,
  },
]
```

📌 Even if placed in a module → interceptor is **global**

🔄 Response Mapping (Transform Output)

Controller returns:

```
[]
```

Interceptor converts it to:

```
{ "data": [] }
```

Transform Interceptor

```
@Injectable()
export class TransformInterceptor<T>
  implements NestInterceptor<T, { data: T }> {

  intercept(context: ExecutionContext, next: CallHandler) {
    return next.handle().pipe(
      map(data => ({ data })),
    );
  }
}
```

📌 Used for:

- standard API responses
- wrapping metadata
- pagination format

⚠️ Important Rule

❌ **Does NOT work with `@Res()`**
Because Nest loses control of response stream

🚫 Null Value Transformation

```
.pipe(
  map(value => value === null ? '' : value)
)
```

Use case:

- frontend compatibility
- clean API contracts

❌ Exception Mapping (Error Replacement)

```
@Injectable()
export class ErrorsInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler) {
    return next.handle().pipe(
      catchError(() =>
        throwError(() => new BadGatewayException()),
      ),
    );
  }
}
```

Meaning:

- Any error → replace with `502 Bad Gateway`

🧠 Used when:

- hiding internal errors
- microservice failures

🚀 Stream Overriding (Skip Controller)

```
if (isCached) {
  return of([]);
```

```
}
return next.handle();
```

Result:

- Controller **never runs**
- Cached response returned instantly

📌 Perfect for:

- caching
- feature flags
- mock APIs

⏱️ Timeout Interceptor (Very Common in Prod)

```
timeout(5000),
catchError(err => {
  if (err instanceof TimeoutError) {
    throw new RequestTimeoutException();
  }
})
```

What happens?

- If API takes > 5 seconds → auto cancel
- Prevents stuck requests
- Protects server resources

🧩 Mental Model (EXAM FRIENDLY)

Component	Purpose
Middleware	Raw request handling
Guard	Allow / Deny
Pipe	Validate / Transform input
Interceptor	Wrap request-response
Filter	Handle exceptions

✅ When Should YOU Use Interceptors?

Use Interceptors for:

- Logging
- Response formatting
- Performance metrics
- Caching
- Timeouts
- Error masking
- Auditing
- Tracing (APM)