Tratamento de Erros e Exceções no NestJS

PROF. DR.
RONI FABIO BANASZEWSKI

Objetivo

- Compreender como o NestJS lida com erros por padrão.
- Aprender a criar filtros de exceção para personalizar o tratamento de erros.
- Aplicar técnicas para melhorar a experiência do usuário e o monitoramento de erros em produção.
- Implementar soluções práticas em um projeto NestJS.

Por que o Tratamento de Erros é Essencial?

- Confiabilidade: Garante que a API responda adequadamente mesmo em cenários de falha.
- Melhor Experiência do Usuário:
  - Respostas claras ajudam os clientes a entender o problema.
  - Mensagens padronizadas reduzem a confusão.
- Segurança:
  - Oculta detalhes sensíveis do sistema.
  - Minimiza o risco de exposição de informações críticas.
- Facilidade de Depuração:
  - Logs e mensagens de erro estruturados ajudam na investigação de problemas.

O que são Exceções no NestJS?

- Definição:
  - Exceções são usadas para interromper o fluxo normal de execução quando ocorre um erro.
- Tratamento no NestJS:
  - Exceções são tratadas por uma camada de gerenciamento embutida.
  - Baseia-se no conceito de "lançar e capturar" erros (try-catch).

Classe Base – Error

- Todos os erros no JavaScript derivam da classe Error.
- Principais propriedades:
  - message: Descrição do erro.
  - stack: Caminho da execução no momento do erro.

```
try {
  throw new Error('Algo deu errado!');
} catch (error) {
  console.error(error.message); // Algo deu errado!
}
```

Como o NestJS Lida com Erros por Padrão?

- Camada de Exceções Global:
  - O NestJS possui uma camada de gerenciamento de exceções integrada.
  - Captura automaticamente erros não tratados lançados durante a execução de controladores.
- Exceções Capturadas:
  - Erros que derivam de HttpException.
  - Exceções padrão do JavaScript, como Error.
  - Erros inesperados, como exceções de bibliotecas externas.

Como o NestJS Lida com Erros por Padrão?

- Resposta Padrão:
  - Se a exceção não for reconhecida, o NestJS gera a seguinte resposta:

Como o NestJS Lida com Erros por Padrão?

- Exceções Reconhecidas (ex.: HttpException):
  - Quando uma exceção HttpException é lançada, o NestJS utiliza as informações fornecidas na exceção para gerar a resposta:
    - statusCode: Código de status HTTP fornecido.
    - message: Mensagem fornecida.

```
{
  "statusCode": 403,
  "message": "Recurso não permitido"
}
```

```
import { HttpException, HttpStatus } from '@nestjs/common';

@Get()
async getExample() {
  throw new HttpException('Recurso não permitido', HttpStatus.FORBIDDEN);
}
```

Como o NestJS Lida com Erros por Padrão?

- Stack Trace (Para Desenvolvimento):
  - Durante o desenvolvimento, o NestJS inclui detalhes como o stack trace no log de erros.
  - No ambiente de produção, esses detalhes são omitidos para segurança.
- Configurando Erros no Módulo Principal:
  - A configuração global de erros pode ser feita utilizando o app.useGlobalFilters().
  - Personalize o tratamento de erros com filtros de exceção (abordado em seções posteriores).

Exceções Padrão

Classe HttpException

- A classe HttpException é usada para lançar erros HTTP personalizados.
- Estrutura do Construtor:

```
new HttpException(response: string | object, status: number);
```

- Parâmetros:
  - response: Mensagem ou objeto JSON para a resposta.
  - status: Código de status HTTP.
  - options (opcional): Propriedades extras como a causa do erro.
- Disponível no pacote @nestjs/common.

Classe HttpException

- Exemplo Simples com String:

```
@Get()
async findAll() {
  throw new HttpException('Forbidden', HttpStatus.FORBIDDEN);
}
```

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

- Personalizando a Resposta com objeto:

```
throw new HttpException(
  { status: 403, error: 'Ação não permitida' },
  HttpStatus.FORBIDDEN,
);
```

```
{
  "status": 403,
  "error": "Ação não permitida"
}
```

Utilizando HttpStatus

- É um Enum embutido no NestJSque fornece valores constantes para os códigos de status HTTP (200, 404, 500, etc.).
- Principais Códigos Disponíveis:
- Informativos (1xx): HttpStatus.CONTINUE (100)
- Sucesso (2xx): HttpStatus.OK (200), HttpStatus.CREATED (201)
- Redirecionamento (3xx): HttpStatus.MOVED_PERMANENTLY (301)
- Erro do Cliente (4xx): HttpStatus.BAD_REQUEST (400), HttpStatus.NOT_FOUND (404), HttpStatus.UNAUTHORIZED (401)
- Erro do Servidor (5xx):
  - HttpStatus.INTERNAL_SERVER_ERROR (500)

Exceções HTTP Integradas

- Exceções integradas no NestJS para cobrir cenários comuns de erro.
- Todas herdam de HttpException.
- Exemplos de Exceções Prontas:
  - BadRequestException (400)
  - UnauthorizedException (401)
  - NotFoundException (404)
  - ConflictException (409)
  - InternalServerErrorException (500)
  - E muito mais...

Exceções HTTP Integradas

- Exemplo:

Uso no Código:

```
@Get(':id')
async findOne(@Param('id') id: string) {
  if (!id.match(/^\d+$/)) {
    throw new BadRequestException('ID inválido');
  }
}
```

Resposta Gerada:

```
{
  "statusCode": 400,
  "message": "ID inválido"
}
```

Personalizando Exceções

- Por Que Criar Exceções Personalizadas?
  - Fornecer mensagens claras e úteis para desenvolvedores e consumidores.
- Opções de como criar
  - Crie uma classe que herda de HttpException.
  - Envie um Objeto JSON
  - Customize Exceções HTTP Integradas

Personalizando Exceções

- Crie uma classe que herda de HttpException.

Exemplo de Exceção Personalizada:

```
import { HttpException, HttpStatus } from '@nestjs/common';

export class ForbiddenActionException extends HttpException {
  constructor() {
    super('Ação proibida', HttpStatus.FORBIDDEN);
  }
}
```

Uso no Controlador:

```
@Delete(':id')
async delete(@Param('id') id: string) {
  if (!this.canDelete(id)) {
    throw new ForbiddenActionException();
  }
}
```

Personalizando Exceções

- Envie objetos JSON no message para adicionar detalhes.

Exemplo de Exceção Personalizada:

```
throw new HttpException(
  { status: 400, error: 'Campo obrigatório ausente' },
  HttpStatus.BAD_REQUEST,
);
```

Resposta gerada:

```
{
  "status": 400,
  "error": "Campo obrigatório ausente"
}
```

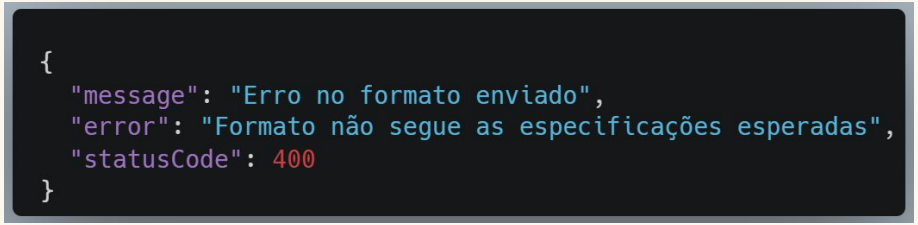
Personalizando Exceções

- Use Exceções HTTP Integradas:

```
throw new BadRequestException('Entrada inválida', {
  cause: new Error('Falta de campo obrigatório'),
  description: 'Certifique-se de enviar todos os campos necessários',
});
```

Adicionando Detalhes Extras no Corpo da Resposta

- Propriedades Adicionais:
  - timestamp: Hora do erro.
  - path: Endpoint acessado.
  - cause: Detalhe interno da causa do erro.



```
{
  "message": "Erro no formato enviado",
  "error": "Formato não segue as especificações esperadas",
  "statusCode": 400
}
```

```
throw new BadRequestException('Erro no formato enviado', {
  cause: new Error('Campo inválido'),
  description: 'Formato não segue as especificações esperadas',
});
```

Boas Práticas

- Use Classes Integradas Sempre Que Possível:
  - Evite criar exceções customizadas para cenários comuns.
- Seja Claro e Preciso:
  - Mensagens devem explicar o problema e como corrigi-lo.
- Adicione Detalhes Relevantes:
  - Inclua informações como o campo inválido, o formato esperado e a causa do erro.
- Padronize o Formato:
  - Mantenha uma estrutura uniforme para todas as respostas de erro.

Por Que Usar Filtros de Exceção?

- Limitações das Exceções Padrão:
  - Reutilização Limitada: Difícil aplicar a mesma lógica em várias rotas.
  - Sem Controle Global: Exceções padrão precisam ser lançadas manualmente em cada método.
  - Sem Logging: Não registram erros automaticamente.
  - Formato Fixo: Não permitem customizar profundamente o JSON de resposta.

Filtros de Exceção

O que são Filtros de Exceção?

- Definição:
  - Filtros de exceção permitem capturar e tratar erros de forma personalizada, controlando a resposta enviada ao cliente.
- Por Que Usar?
  - Personalizar mensagens de erro e estrutura da resposta.
  - Adicionar informações adicionais (ex.: timestamp, URL).
  - Centralizar a lógica de tratamento de erros.
- Fluxo de Execução:
  - A exceção ocorre.
  - O filtro de exceção captura o erro.
  - O filtro gera a resposta customizada.

O que são Filtros de Exceção?

- Mecanismo:
  - Substituem a camada padrão de exceções do NestJS para casos específicos.
  - Capturam exceções lançadas em controladores ou serviços.
- Vantagens:
  - Melhoram a consistência nas respostas de erro.
  - Simplificam o código de controladores ao centralizar a lógica de tratamento de erros.
- Exemplo de Resposta Padrão Personalizada:

Estrutura de um Filtro de Exceção

- Classe Decorada com @Catch:
  - Identifica o tipo de exceção a ser capturado.
- Implementação de ExceptionFilter:
  - Obrigatório implementar o método catch(exception: any, host: ArgumentsHost).
- Uso do ArgumentsHost:
  - Acesso a detalhes da requisição e resposta.
  - Pode acessar o contexto HTTP, WebSocket ou Microservices.

Criando Filtros de Exceção

- Adapte a resposta conforme sua necessidade.
- O @Catch pode capturar um ou mais tipos de exceção.

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

    response.status(status).json({
      statusCode: status,
      message: exception.message,
      timestamp: new Date().toISOString(),
      path: request.url,
    });
  }
}
```

Criando Filtros de Exceção

- Crie uma classe que implemente ExceptionFilter<T>.
- Use o decorador @Catch() para capturar tipos específicos de exceção.
- Utilize o método catch() para personalizar a resposta.

```
@Catch(HttpException)
export class ForbiddenExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();

    response.status(403).json({
      message: 'Acesso negado.',
      statusCode: 403,
    });
  }
}
```

Acessando Detalhes com ArgumentsHost

- O que é ArgumentsHost?
  - Um objeto que fornece acesso ao contexto da execução atual.
  - Permite acessar dados da requisição, resposta, e outros detalhes.
  - Com ArgumentsHost, é possível criar respostas ricas em informações sobre erros, incluindo a URL acessada e o método HTTP usado.
- Métodos Importantes:
  - switchToHttp(): Acessa o contexto HTTP.
  - switchToRpc(): Acessa o contexto de chamadas remotas (RPC).
  - switchToWs(): Acessa o contexto de WebSockets.

Acessando o Contexto HTTP

- Requisição:

```
const request = ctx.getRequest<Request>();
console.log(request.url);
```

- Resposta:

```
const response = ctx.getResponse<Response>();
response.status(404).send('Not Found');
```

Usando o Decorador @Catch

- O que é @Catch?
  - Define quais tipos de exceção o filtro deve capturar.
  - Pode ser usado com uma classe específica ou com várias exceções.
- Uso Comum:
  - Capturar uma exceção específica:
    - @Catch(HttpException)
  - Capturar múltiplos tipos de exceção:
    - @Catch(BadRequestException, NotFoundException)
  - Captura de Todas as Exceções. Sem parâmetros, o filtro captura qualquer tipo de exceção:
    - @Catch()

Aplicando Filtros de Exceção

- Níveis de Escopo para Filtros:
  - Escopo de Método
  - Escopo de Controlador
  - Escopo Global
- Dicas:
  - Prefira filtros globais para erros comuns, como NotFoundException.
  - Use filtros por método para casos específicos.

Aplicando Filtros de Exceção

- Escopo de Método: Aplica o filtro apenas a um único método.

```
@Post()
@UseFilters(new HttpExceptionFilter())
async create(@Body() createDto: CreateDto) {
  throw new Error('Erro simulado');
}
```

Aplicando Filtros de Exceção

- Escopo de Controlador: Aplica o filtro a todos os métodos de um controlador.

```
@Controller('users')
@UseFilters(HttpExceptionFilter)
export class UsersController {}
```

Aplicando Filtros de Exceção

- Escopo Global: Aplica o filtro a toda a aplicação.
  - Use filtros globais para erros comuns (e.g., NotFoundException).

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new HttpExceptionFilter());
  await app.listen(3000);
}
```

Capturando Erros Genéricos com Filtros

- Por Que Capturar Erros Genéricos?
  - Cobrir cenários inesperados que não são tratados por filtros específicos.
  - Garantir que a aplicação nunca quebre sem uma resposta controlada.
- Como Fazer?
  - Use o @Catch() sem parâmetros para capturar todos os tipos de exceção.

Boas Práticas

- Centralize o Tratamento de Erros:
  - Use filtros globais para capturar erros comuns.
  - Evite duplicar lógica de tratamento em múltiplos locais.
- Personalize Mensagens de Erro:
  - Torne as mensagens úteis e claras para os desenvolvedores e usuários finais.
  - Use o idioma do público-alvo, quando apropriado.
- Utilize Filtros Específicos:
  - Aplique filtros por controlador ou método para lidar com cenários pontuais.
  - Exemplo: Logging detalhado em endpoints críticos.

Boas Práticas

- Evite Divulgar Informações Sensíveis:
  - Não exponha detalhes internos no erro (e.g., stack trace ou nomes de serviços).
  - Substitua por mensagens genéricas como "Erro interno do servidor".
- Teste Todos os Cenários:
  - Simule diferentes tipos de erros e garanta que cada um seja tratado adequadamente.
  - Inclua cenários de falha como IDs inválidos, campos ausentes e erros de banco de dados.
- Mantenha a Consistência:
  - Padronize o formato das respostas de erro em toda a aplicação.
  - Utilize propriedades comuns como statusCode, message e timestamp.

THE END!