Logs no NestJS

PROF. DR.
RONI FABIO BANASZEWSKI

Objetivo

- Explorar como implementar, personalizar e utilizar logs no NestJS para melhorar o monitoramento, a depuração e a manutenção de aplicações.

O Que São Logs?

- Definição: Registros detalhados das atividades e eventos ocorridos em uma aplicação.
- Benefícios:
  - Diagnóstico de problemas e erros.
  - Monitoramento de desempenho.
  - Análise de comportamento da aplicação.
- Citação Inspiradora:
  - "Logs são os rastros deixados pela sua aplicação no caminho para a solução."

Logger no NestJS

- Disponibilidade:
  - O Logger é uma classe embutida no @nestjs/common
- Métodos Principais:
  - log(): Para mensagens gerais.
  - error(): Para erros.
  - warn(): Para avisos.
  - debug(): Para depuração.
  - verbose(): Para informações detalhadas.

Logger no NestJS

- Exemplo Prático:

```
import { Logger } from '@nestjs/common';

const logger = new Logger('AppService');
logger.log('Aplicação iniciada com sucesso!');
logger.error('Erro ao conectar ao banco de dados', 'DatabaseService');
logger.warn('Alerta: alta latência detectada');
```

Categorias de Logs no Logger Padrão

- log (Informações):
  - Usado para mensagens gerais, como inicialização de serviços.
  - Exemplo: "Servidor iniciado na porta 3000."
- error (Erros):
  - Para erros críticos que exigem atenção imediata.
  - Exemplo: "Falha ao conectar ao banco de dados."
- warn (Avisos):
  - Alertas sobre problemas que podem não ser críticos, mas requerem atenção.
  - Exemplo: "Requisição de alta latência detectada."

Categorias de Logs no Logger Padrão

- debug (Depuração):
  - Informações úteis durante o desenvolvimento.
  - Exemplo: "Consulta SQL gerada: SELECT * FROM users."
- verbose (Detalhado):
  - Informações adicionais úteis para depuração avançada.
  - Exemplo: "Middleware executado para a rota /users."

Categorias de Logs no Logger Padrão

- Exemplo Consolidado:

```
import { Logger } from '@nestjs/common';

const logger = new Logger('OrderService');
logger.log('Pedido criado com sucesso');
logger.warn('Estoque baixo para o produto ID 123');
logger.error('Falha ao processar pagamento', 'PaymentError');
logger.debug('Tempo de execução: 120ms');
logger.verbose('Payload recebido: {...}');
```

Categorias de Logs no Logger Padrão

- Exemplo Consolidado:

```
import { Logger } from '@nestjs/common';

const logger = new Logger('OrderService');
logger.log('Pedido criado com sucesso');
logger.warn('Estoque baixo para o produto ID 123');
logger.error('Falha ao processar pagamento', 'PaymentError');
logger.debug('Tempo de execução: 120ms');
logger.verbose('Payload recebido: {...}');
```

Habilitando Logs Globais

- Configuração de Logs Globais:
  - No arquivo main.ts, habilite o logger global ao criar a aplicação.

```
import { Logger, ValidationPipe, NestFactory } from '@nestjs/common';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  app.useLogger(new Logger());
  await app.listen(3000);
}
bootstrap();
```

Habilitando Logs Globais

- Vantagens do Logger Global:
  - Todos os logs são centralizados.
  - Possibilita a configuração única e reutilização em toda a aplicação.
- Nível de Logs:
  - Configure níveis globais de log para filtrar categorias de mensagens.

```
const app = await NestFactory.create(AppModule, {
  logger: ['log', 'warn', 'error'],
});
```

Configuração Global no Main.ts

- Filtrando Níveis de Log:
  - Ativando apenas logs importantes para produção.
- Logs Verbosos em Desenvolvimento:
  - Adicione debug e verbose para detalhes adicionais.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    logger: ['log', 'error', 'warn'],
  });
  await app.listen(3000);
}
```

Adicionando Logs Personalizados em Serviços

- Por Que Logar em Serviços?
  - Monitorar fluxos internos, como operações de banco de dados.
  - Diagnosticar falhas de lógica ou integração.
  - Identificar gargalos e pontos de melhoria.

Adicionando Logs Personalizados em Serviços

- Exemplo de Logs em um Serviço:

```
@Injectable()
export class UsersService {
  private readonly logger = new Logger(UsersService.name);

  async findAll() {
    this.logger.log('Iniciando busca de todos os usuários');
    try {
      const users = await this.fetchUsersFromDatabase();
      this.logger.log(`Encontrados ${users.length} usuários`);
      return users;
    } catch (error) {
      this.logger.error('Erro ao buscar usuários', error.stack);
      throw error;
    }
  }
```

Adicionando Logs Personalizados em Serviços

- Boas Práticas de Níveis de Log:
  - log para fluxos normais.
  - warn para eventos inesperados.
  - error para falhas.
- Detalhes Relevantes: Inclua o contexto, como IDs ou valores importantes.
- Evite Dados Sensíveis: Não exponha senhas, tokens ou informações confidenciais.

```
[UsersService] Iniciando busca de todos os usuários
[UsersService] Encontrados 1 usuários
[UsersService] Erro ao buscar usuários: Database not reachable
```

Logs em Middleware

- O que é Middleware?
  - Código executado antes de uma rota ser processada.
- Objetivo dos Logs em Middleware:
  - Registrar todas as requisições recebidas.
  - Capturar informações como método, URL e cabeçalhos.
- Benefícios:
  - Visibilidade de tráfego em tempo real.
  - Diagnóstico de latência e status de resposta.

Logs em Middleware

```ts
import { Injectable, NestMiddleware, Logger } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  private readonly logger = new Logger('HTTP');

  use(req: Request, res: Response, next: NextFunction) {
    const { method, originalUrl } = req;
    const start = Date.now();

    res.on('finish', () => {
      const duration = Date.now() - start;
      const statusCode = res.statusCode;
      this.logger.log(`${method} ${originalUrl} ${statusCode} - ${duration}ms`);
    });

    next();
  }
}
```

Logs em Interceptadores

- O que é um Interceptador?
  - Uma camada entre a requisição e a resposta, permitindo manipulações adicionais.
- Objetivo dos Logs em Interceptadores:
  - Registrar o tempo de execução de rotas.
  - Adicionar informações sobre o contexto da requisição.
- Benefícios:
  - Monitoramento detalhado da performance.
  - Identificação de rotas com maior tempo de execução.

Logs em Interceptadores

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler, Logger } from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  private readonly logger = new Logger('ExecutionTime');

  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const now = Date.now();
    const req = context.switchToHttp().getRequest();
    const method = req.method;
    const url = req.url;

    return next.handle().pipe(
      tap(() =>
        this.logger.log(`${method} ${url} - ${Date.now() - now}ms`)
      ),
    );
  }
}
```

Logs em Filtros de Exceção

- O que é um Filtro de Exceção?
  - Uma camada para capturar erros e gerar respostas apropriadas.
- Objetivo dos Logs em Filtros:
  - Capturar e registrar detalhes de erros.
  - Ajudar na investigação de exceções inesperadas.
- Benefícios:
  - Centralização do registro de erros.
  - Melhoria no rastreamento e correção de falhas.

Logs em Filtros de Exceção

```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException, Logger } from '@nestjs/common';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  private readonly logger = new Logger('HttpException');

  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const request = ctx.getRequest();
    const status = exception.getStatus();

    this.logger.error(
      `Error ${status}: ${exception.message} - ${request.method} ${request.url}`,
    );

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

Estruturação de Logs para Ambientes Diferentes

- Por Que Diferenciar?
  - Ambientes de desenvolvimento precisam de logs mais detalhados para depuração.
  - Ambientes de produção devem focar em informações críticas, reduzindo ruído e protegendo dados sensíveis.
  - Logs detalhados em produção podem impactar a performance e aumentar custos de armazenamento.

Estruturação de Logs para Ambientes Diferentes

- No main.ts, defina os níveis de log com base no ambiente:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    logger: process.env.NODE_ENV === 'production'
      ? ['error', 'warn']
      : ['log', 'debug', 'verbose', 'warn', 'error'],
  });

  await app.listen(3000);
}
bootstrap();
```

Estruturação de Logs para Ambientes Diferentes

- Centralize a configuração com arquivos .env ou bibliotecas como @nestjs/config:
  - .ENV

```
NODE_ENV=production
LOG_LEVELS=error,warn
```

  - main.ts:

```
const logLevels = process.env.LOG_LEVELS?.split(',') || ['log', 'warn', 'error'];
const app = await NestFactory.create(AppModule, { logger: logLevels });
```

Ferramentas para Centralização e Análise de Logs

- Por Que Integrar Ferramentas de Logs?
  - Centralização: Consolida logs de várias instâncias ou serviços.
  - Visualização: Gráficos e dashboards para monitoramento em tempo real.
  - Alertas: Notificações automáticas para falhas ou anomalias.
- Ferramentas Populares de Logs:
  - ELK Stack (Elasticsearch, Logstash, Kibana):
    - Ideal para busca, análise e visualização de logs.
  - Datadog:
    - Monitoramento de performance e integração com logs.
  - Sentry:
    - Focado em rastreamento de erros e exceções.
  - Graylog:
    - Processamento de logs em tempo real.


Sentry

Integrando Logs com Sentry

- Instalação do SDK Sentry:

```
npm install --save @sentry/node
npm install --save @sentry/tracing
```

Integrando Logs com Sentry

- Configuração Básica no main.ts:

```
import * as Sentry from '@sentry/node';
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  Sentry.init({
    dsn: 'SEU_DSN_AQUI',
    tracesSampleRate: 1.0,
  });

  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

Integrando Logs com Sentry

- Exemplo de Captura de Erros: Personalize capturas em interceptadores ou serviços.

```
import * as Sentry from '@sentry/node';

try {
  // Código que pode lançar erro
} catch (error) {
  Sentry.captureException(error);
}
```

Integrando Logs com Sentry

- Visualização e Alertas:
- Configure dashboards no Sentry ou Kibana para acompanhar:
  - Tendências de erros.
  - Distribuição de logs por serviço.
  - Alertas automáticos por e-mail ou Slack.
- Benefícios:
  - Diagnóstico Rápido: Identifique problemas antes que impactem os usuários.
  - Rastreamento de Exceções: Contexto detalhado para cada erro.
  - Colaboração: Equipes recebem notificações em tempo real.

Boas Práticas

Boas Práticas de Logging

- Definição de Objetivos:
  - Determine o propósito do log: Monitoramento, auditoria ou diagnóstico.
  - Evite excesso de logs que podem obscurecer informações importantes.
- Organização dos Logs:
  - Use categorias claras para identificar a origem do log.
  - Exemplo: Logger(UsersService.name) ou Logger(DatabaseService.name).
  - Inclua contextos úteis, como IDs de requisição ou parâmetros chave.

Boas Práticas de Logging

- Níveis de Log Apropriados:
  - log (Informativo): Processos normais.
  - warn (Aviso): Problemas potencialmente críticos.
  - error (Erro): Falhas que requerem ação imediata.
  - debug e verbose: Detalhes técnicos, usados em ambientes de desenvolvimento.
- Performance e Segurança:
  - Minimize Logs Desnecessários: Use logs detalhados apenas em ambiente de desenvolvimento.
  - Proteja Informações Sensíveis: Não registre senhas, tokens ou dados pessoais.

Boas Práticas de Logging

- Logs Centralizados:
  - Configure logs globais com níveis específicos para cada ambiente (desenvolvimento, produção, etc.).
- Exemplo para produção:

```
const app = await NestFactory.create(AppModule, {
  logger: ['error', 'warn'],
});
```

THE END!