Desenvolvimento Backend

Aula 12 - Depuração e manipulação de erros

Material Didático do Instituto Metrópole Digital - IMD

Termo de uso

Os materiais didáticos aqui disponibilizados estão licenciados através de Creative Commons **Atribuição-SemDerivações-SemDerivados CC BY-NC-ND**. Você possui a permissão para realizar o download e compartilhar, desde que atribua os créditos do autor. Não poderá alterá-los e nem utiliza-los para fins comerciais.

Atribuição-SemDerivações-SemDerivados

CC BY-NC-ND

https://creativecommons.org/licenses/by-nc-nd/4.0/

Apresentação

Nesta aula, serão apresentados conceitos e práticas sobre depuração, tratamento de erros e *logs*.

Objetivos

- Conhecer o conceito e a utilização da depuração.
- Conhecer como fazer o tratamento de erros.
- Conhecer como fazer um bom *log*.

Depuração: o que é e como fazer

Link do video da aula: https://youtu.be/4z0SFGTE6ac

Depuração é a ação realizada pelo desenvolvedor para ver, linha a linha, as transformações que estão acontecendo no código.

Configurando o depurador

Inicialmente, no menu lateral do *vscode*, encontre a ferramenta *Run and Debug* e clique nela. Após essa ação, ficará visível um botão *Run and Debug* no qual você deve clicar para que seja feita a configuração.

Primeiro é necessário dizer ao depurador qual o ambiente estamos utilizando, no nosso caso é o *node.js*. Essa ação irá gerar um documento '*launch.js*' que contém a configuração inicial do depurador. No entanto, nesse arquivo precisamos alterar a linha que corresponde ao caminho do arquivo inicial da aplicação.

O arquivo de configuração deve ficar conforme o modelo abaixo:

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit:
https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "type": "pwa-node",
            "request": "launch",
            "name": "Depuração",
            "skipFiles": [
```

```
                "<node_internals>/**"
            ],
            "program": "${workspaceFolder}/src/index.js"
        }
    ]
}
```

Por fim, inicie a depuração e acompanhe a aula para melhor compreensão da ferramenta.

Notas:

- **BREAKPOINTS**: *é um ponto de interrupção do código durante a depuração, fazendo o código ficar parado esperando um sinal de prosseguimento do programador*.

- **WATCH**: *é uma expressão que o programador deseja observar a todo momento*.

- **CALL STACK**: *é a pilha de chamadas de funções*.

Tratamento de erros com try/catch

Link do video da aula: https://youtu.be/n74PHHi5eM0

Agora será visto como tratar erros no *express*. Então, no código serão analisados e tratados possíveis erros.

Tratando erro de usuário inexistente

Considere que o usuário da aplicação deseje adicionar um novo post, no entanto, ao inserir o *id*, coloque um que não exista no banco de dados dos usuários. Nesse caso, o *sequelize* irá colocar um erro no terminal, porém, nada será retornado ao usuário que fez a tentativa e a requisição ficará esperando alguma resposta para sempre.

Para tratar esse erro, usaremos o *try/catch*, que possui essa estrutura:

```
    try{
        const post = await Post.create(data)
        res.json({msg: "Post adicionado com sucesso!"})
```

```
    }catch (err){
        res.status(500).json({msg: "Falha ao adicionar post"})
    }
```

O bloco *try* vai tentar realizar as instruções normalmente, mas ao encontrar um erro ou exceção. Ao acontecer isso, o bloco *catch* vai capturar o erro e fazer algum tratamento. Então, utilizando esses blocos, o usuário agora terá alguma informação sendo retornada.

Personalizando erros

Para não ter que fazer tratamento de erros em toda função, iremos personalizar esse erro deixando-o mais geral. Então, na pasta */src*, crie uma pasta de utilitários chamada *utils* e dentro dela um arquivo *ErrorHandler.js*, implementando o código abaixo:

```
class ErrorHandler extends Error {

    constructor(statusCode, msg){
        super()
        this.statusCode = statusCode
        this.msg = msg
    }

}

module.exports = ErrorHandler
```

Fazendo isso, ao acontecer uma exceção é possível informar ao *express* que ocorreu um erro e capturar isso em apenas um local, através de um *middleware*, e aplicar um único tratamento.

Agora, é preciso mudar o bloco que adiciona um novo post ao banco de dados, tratando o erro de adicionar com um *id* de usuário inexistente. Primeiramente, importe o *ErrorHandler.js* no arquivo *post.rota.js* e altere sua rota para tratar esse erro.

O código deve ficar conforme se vê a seguir:

```
    router.post('/', async (req, res, next) => {
            const data = req.body
            if (req.file){
                    data.foto =
`/static/uploads/${req.file.filename}`
```

```
            }
            try{
                    const post = await Post.create(data)
                    res.json({msg: "Post adicionado com sucesso!"})
            }catch (err){
                    next(new ErrorHandler(500, 'Falha interna ao
adicionar postagem'))
            }

    })
```

Criando o *middleware*

Para criar o *middleware*, abra o arquivo *index.js* e implemente o código abaixo:

```
app.use((err, req, res, next) => {
    const { statusCode, msg } = err
    res.status(statusCode).json({msg: msg})
})
```

Logs: o que é e como fazer

Link do video da aula: https://youtu.be/ztV6JF2SqG0

Log é um registro de tudo que aconteceu na execução do programa, que serve para análise de erros e compreensão do funcionamento da aplicação.

Utilizando o *winston*

A biblioteca winston auxiliará no uso de *logs*. Para usá-la, é preciso fazer a instalação executando, no terminal, o comando:

```
 npm install winston
```

Após isso, é preciso configura-lá. Então, crie um arquivo, nomeando-o de '*logger.js*' na pasta *utils*.

Com o arquivo já criado, insira o código abaixo:

```
const { createLogger, format, transports } = require('winston');
const { combine, timestamp, label, printf } = format;
```

```
const myFormat = printf(({ level, message, label, timestamp }) => {
  return `${timestamp} [${label}] ${level}: ${message}`;
});

const logger = createLogger({
  format: combine(
    label({ label: 'right meow!' }),
    timestamp(),
    myFormat
  ),
  transports: [new transports.Console()]
});

module.exports = logger
```

Agora, abra o arquivo '*index.js*', importe *logger* e troque todos os *console.log()* por *logger.info()*. O arquivo ficará conforme o modelo abaixo:

```
app.listen(8080, () => {
    logger.info(`Iniciando no ambiente ${process.env.NODE_ENV}`)
    logger.info('Servidor pronto na porta 8080')
})
```

Configurando novos transportes

Com essa configuração acima, o *log* vai ser escrito apenas no console. Porém, queremos salvá-lo em um arquivo e para fazer isso é preciso configurar, em '*logger.js*', um novo transporte, se vê abaixo:

```
new transports.File({
    filename: "logs/app-log.log",
    level: 'debug'
})]
```

Com isso, as informações serão salvas em um arquivo que será criado automaticamente. Mas, ainda assim, seria interessante criar um *log* no nível *debug*. Para isso, no entanto, é necessário antes criar um *middleware*.

Criando um *middleware*

Crie um arquivo chamado '*log.mid.js*' na pasta *middleware* e implemente o código abaixo:

```
    const logger = require("../utils/logger");

    function logar(req, res, next){
            logger.debug('Requisição %s na rota %s', req.method,
req.path)
            next()
    }

    module.exports = logar
```

Também é preciso registrar esse novo *middleware* no '*index.js*'.

Atualizando o arquivo de *logger*

Acompanhando a aula, serão feitas algumas pequenas alterações, explicadas no vídeo, no arquivo de '*logger.js*'. Ao final, o arquivo deverá ficar conforme o modelo abaixo:

```
const { createLogger, format, transports } = require('winston');
const { combine, timestamp, label, printf } = format;

const myFormat = printf(({ level, message, label, timestamp }) => {
  return `${timestamp} [${level}] ${message}`;
});

const logger = createLogger({
  format: combine(
    format.splat(),
    timestamp(),
    myFormat
  ),
  transports: [new transports.Console({
      level: 'debug'
  }), new transports.File({
    filename: "logs/app-log.log",
    level: 'debug'
  })]
});

module.exports = logger
```

Resumo

Nesta aula foram vistos conceitos de depuração e *logs*. Além disso, foram explanadas as práticas de como fazer uma depuração, como tratar erros e como criar um bom *log* para o sistema. Ademais, foi destacada a importância dessas práticas para evitar problemas futuros com a aplicação em produção.