

TRYBE

Modulo III – Back-end

Bloco 26 – NodeJS

1) Intro NodeJS

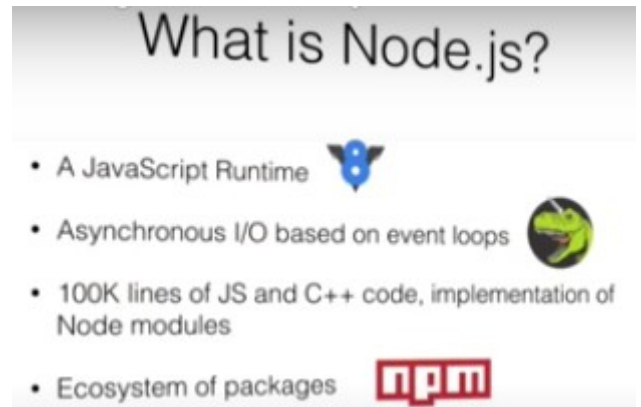
O que é e porque usar

O que é

Interpretador/engine/runtime de Javascript.

Porque usar

- **incontornabilidade:** abundância dos pacotes npm;
- **performance:** NodeJS permite escrever softwares servidores de requisições HTTP eficientemente, por conta de leitura e escritura não bloqueante e assíncrona
- **tempo real**, ex lib [socket](#)
- versatilidade do **Js**.



Sistema de módulos

O que é

Funcionalidade / conjunto de funcionalidades isolada do restante do código (escopo).

Módulos internos

Core modules, inclusos no NodeJs. [fs](#), [url](#), [querystring](#), [util](#).

Alguns core modules:

- HTTP: a module that acts as a server
- File System: a module that reads and modifies files
- Path: a module for working with directory and file paths
- Assertion Testing: a module that checks code against prescribed constraints

Módulos locais

Definidos juntamente à aplicação. Podem ser publicados com npm para uso de terceiros.

Módulos de terceiros

Criados por outras pessoas e disponibilizados para uso através do npm.

Importar módulos

Saber **importar módulo / pacote externo** para o contexto no qual estamos.

Módulos ES6 - Suportado via transpiladores (Babel) ou supersets (Typescript).

Importar

```
import meuModulo from './meuModulo';  
meuModulo.fazerAlgo();
```

OU

```
import { fazerAlgo } from './meuModulo';  
fazerAlgo();
```

Exportar

```
export function fazerAlgo () {  
  console.log('fazendo');  
}  
export default { fazerAlgo };
```

Módulos CommonJS - No NodeJS nativamente. Comando *require*.

Com core modules

Chamar **core modules como parâmetro do require**:

```
const fs = require('fs');  
fs.readFileSync('./meuArquivo.txt');
```

Com locais

O **segundo parâmetro é o caminho do arquivo ou da pasta** (precisando de index.js), ainda melhor para reaproveitar todos seus elementos:

No index.js: *module.exports = { funcionalidade1, funcionalidade2 };*

No app:

```
const meuModulo = require('./meuModulo');  
console.log(meuModulo); // { funcionalidade1: [Function: funcionalidade1], funcionalidade2:  
[Function: funcionalidade2] }  
meuModulo.funcionalidade1();
```

Com terceiros

Baixar pacote e depois usar module como parâmetro (procurado no node_modules).

Pode inclusive destruturando: *const { nomeComando } = require('comando');*

NPM

Node Package Manager: repositório oficial para publicação de pacotes NodeJS. Pacote é um conjunto de arquivos que exportam um ou mais módulos Node (nem sempre é biblioteca).

CLI: ferramenta de gerenciamento de pacotes (criar, instalar, remover, publicar...).

- **npm init**: criar novo pacote, cria package.json, bom fazer readme.md. Opcional: -y
- **npm install packagename**: Instala pacote. Dentro da node_modules aparecem todos, enquanto package-lock.json mostra o que foi preciso anteriormente.
- **npm publish**: gera um arquivo compactado com todos os arquivos do nosso pacote e o deixa acessível.
- **Propriedade main** do package.js determina o entry point do pacote, o que vai ser baixado dele, geralmente index.js mas pode ter outro nome.
- **npm rm packagename**: Desinstala pacote.

Referências diversas

Pesquisar pacotes existentes no npmjs.com, documentações oficiais [npm docs](https://npmjs.com/docs), [NodeJS doc](https://nodejs.org/doc).

Dicas diversas

Quais as principais diferenças entre o JS no browser e no Node?



Express: module frequentemente usado junto com Node.js

“Express runs between the server created by Node.js and the frontend pages of a web application. Also handles its routing.”

Package.json

Duas opções para o escrever: na mão / no terminal com npm init

```
package.json
1 {
2   "name": "fcc-learn-npm-package-json",
3   "author": "Juliette",
4   "description": "bla", // ex. Freecodecamp
5   "keywords": ["freecodecamp", "gotrybe"], // license mais comum MIT (replicável por todos), GPL mais
6   "license": "UNLICENSED", // complexa
7   "version": "1.0",
8   "dependencies": {
9     "express": "^4.14.0",
10    "moment": "~2.10.2"
11  },
```

Na propriedade *scripts*, possibilidade de escrever “start”: “node index.js” (a ser ativado por npm start no terminal) ou palavrainventada (a ser ativado por npm run palavrainventada).

Atalho Ctrl+J para abrir/minimizar terminal no VSCode.

Pasta src – boa prática de criar e incluir dentro index.js, a fins de organização.

Caso apagar node_modules – dar npm install ou npm i pois o package.json já relacionou.

Instalar node: já feito no bloco React com algo tipo sudo nvm install / sudo npm node. [NVM](https://nvmjs.org) para navegar entre versões.

2) Fluxo assíncrono

Para arrumar problemas de performance ligados com o Js ser single-threaded.

Lendo arquivos com métodos síncronos

Método **fs.readFileSync**

Dois parâmetros: *nome do arquivo e opcional encoding*.
Faz leitura síncrona e erros devem ser acertados manualmente com *try e catch* (feitos para gerar tratamento de erro com functions síncronas apenas).

```
const fs = require('fs');

const nomeDoArquivo = 'meu-arquivo.txt';

try {
  const data = fs.readFileSync('meu-arquivo.txt', 'utf8');
  console.log(data);
} catch (err) {
  console.error(`Erro ao ler o arquivo: ${err.path}`);
  console.log(err);
}
```

Lendo arquivos com métodos assíncronos

Método **fs.readFile**

Três parâmetros: *nome do arquivo, opcional encoding, callback* (params err e data).

Encoding padrão: raw buffer.

Cuidado: dados armazenados na memória RAM.

```
const fs = require('fs');

const nomeDoArquivo = 'meu-arquivo.txt';

fs.readFile(nomeDoArquivo, 'utf8', (err, data) => {
  if (err) {
    console.error(`Não foi possível ler o arquivo ${nomeDoArquivo}\n Erro: ${err}`);
    process.exit(1);
  }
  console.log(`Conteúdo do arquivo: ${data}`);
});
```

Callbacks

Callback com *dois parâmetros, um de error e outro de data*, é chamada de **node-style callback**.

O parâmetro que não corresponde com o cenário efetivo retorna *undefined*.

Lado ruim dos callbacks: escopo limitado criando **callback hell**, com indentações infinitas.

Promises

```
const promiseName = new Promise((resolve, reject) => {
});
promiseName.then().catch()
```

Escrevendo dados em arquivos

Método **fs.writeFile**

```
const fs = require('fs');
fs.writeFile('./meu-arquivo.txt', 'Meu texto', (err) => {
  if (err) {
    throw err;
  }
  console.log('Arquivo salvo');
});
// o conteúdo do meu-arquivo.txt foi alterado para "Meu texto"
```

Terceiro parâmetro opcional [flag](#) para incluir regras de manipulação e abertura do arquivo. Disponível também para o método `writeFileSync`.
Padrão `w`, possibilidade `wr` para ter erro quando já existir (`w` sozinho cria ou reescreve).

```
const fs = require('fs');

fs.writeFile('./meu-arquivo.txt', 'Eu estive aqui :eyes:', { flag: 'wx' }, function (err) {
  // A flag wx abre o arquivo para escrita caso ele não exista
  /*
    Flag =>
    w: write
    x: exclusive
  */
  // Se o arquivo existir, um erro é retornado
  if (err) throw err;
  console.log('Arquivo salvo');
});
```

// fs.createFile(filePath, includedText, flag, asynchronousFunction);

Ferramentas do NodeJS para escrever promises

Método **fs.writeFile** + **async await**

```
const fs = require('fs');

const text = 'Texto teste 2';

async function writingFiles() {
  await fs.writeFile('./meu-arquivo.txt', text, (err) => {
    if (err) return console.log(err);
    console.log(text);
  })
}

writingFiles();
```

| function **fs.promises** transforma funções em promises

```
const fs = require('fs').promises;

const text = 'Texto teste 3';

function writingFiles() {
  fs.writeFile('./meu-arquivo.txt', text);
}

writingFiles();
```

| Propriedade **promisify do módulo util**, transforma parâmetro recebido em promise

```
const fs = require('fs');
const util = require('util');

const writeFile = util.promisify(fs.writeFile);

const text = 'Texto teste 4';

function writingFiles() {
  writeFile('./meu-arquivo.txt', text);
}

writingFiles();
```

*// em caso real, incluir tratamento de erros
// funciona apenas com node-style callback ou seja com erro passado antes de data*

Promise.all

Método da Promise que permite **passar um array de Promises e receber, de volta, uma única Promise**, que será resolvida com os resultados de todas Promises, assim que elas forem resolvidas.

```
const fs = require('fs');

function readFilePromise(fileName) {
  return new Promise((resolve, reject) => {
    fs.readFile(fileName, (err, content) => {
      if (err) return reject(err);
      resolve(content);
    });
  });
}

Promise.all([
  readFilePromise('file1.txt'),
  readFilePromise('file2.txt'),
  readFilePromise('file3.txt')
])
.then(([file1, file2, file3]) => {
  const fileSizeSum = file1.byteLength + file2.byteLength + file3.byteLength;
  console.log(`Lidos 3 arquivos totalizando ${fileSizeSum} bytes`);
})
.catch((err) => {
  console.error(`Erro ao ler arquivos: ${err.message}`);
})
```

Dicas diversas

Quotes

"Um callback é síncrono. Uma function que usa um callback não é síncrona" – ROZ

“Uma função espera um callback OU retorna uma promise” - ROZ

Erro do “pending” infinito – Ex: if dentro do qual o code entra e fica preso apesar de não dever entrar, talvez seja porque é condicionado com promise que usada assim vira booleano true ou false (e se for true, entra no if mesmo).

.finally - Cenário de then.catch.then : em caso de erro, vai executar a catch que faz tratamento de erro, que retorna promise e assim depois entra bem no then. Em caso de sucesso, para no catch. Pode resolver com .finally como ultima etapa, dentro do qual o code entra independentemente do cenário.

Três estados de promise: pending, resolve, reject. Quando cria promise.resolve ou promise.reject ela nasce, força com esse estado. Quando um then retorna promise rejeitada, pula até catch.

Async await: sintaxe mais simples do que Promise.

Await no lugar do return - aplicavel apenas para elementos que retornam promises - com async await consegue dar try catch – podemos combinar async await e promise!

Async mesmo sem await garante de retornar promise.

Saber quando usar await: porque quebrou / porque viu function anterior que por ex procura banco e por isso com certeza retorna promise.

Esses dois codes fazem o mesmo →



```
async function () {
  console.log()
}

function () {
  return new Promise((resolve, reject) => {
    console.log()
    resolve()
  })
}
```

Escrever Promise.reject ou reject sozinho? – depende de se estiver dentro de uma Promise com param reject. Promise.reject também para que seja uma rejeição e não resolução de promise.

3) Arquitetura

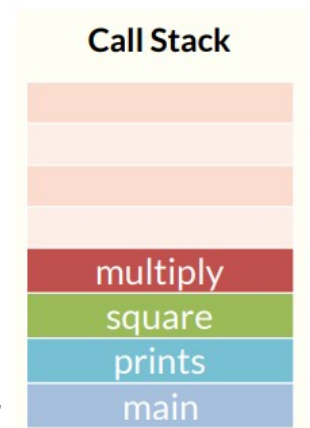
Para entender o Js e o executar com maior performance.

Call Stack

“Pilha de chamadas”: **estrutura de dados** utilizada por um programa para saber **em que ponto da execução** ele está, segundo o princípio LIFO (**Last In, First Out**). Code roda dentro de **chamada pai** (a que depende de todas as outras).

JavaScript é **single threat** ou seja possui apenas uma CS.

*// a function chamada no topo está sendo atualmente executada
// CS enche e desenche até o Js acabar de executar*



Event Loop

Task queue

Fila de tarefas, onde operações estão esperando para acontecer.

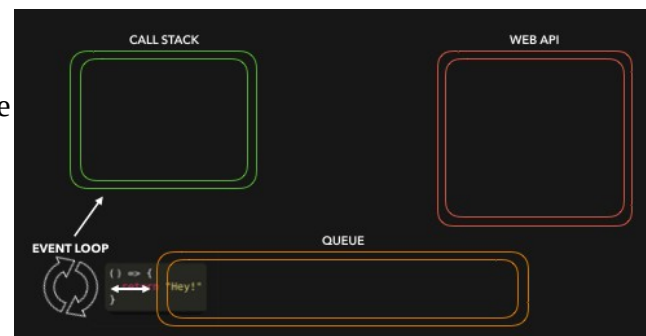
Micro Task Queue especial para promises.

Vendor APIs

// AKA Web APIs no browser e C++ APIs no NodeJS

Saindo do contexto Javascript, podemos operar fora da CS e tirar tempos de espera:

1. Chamada para Vendor API que passa via o CS
2. Vendor API recebe e executa o processamento pesado
3. Tarefa volta na fila e é executada no CS quando estiver vazio.



*// event loop fazendo a transição entre fila e pilha
// resumindo; funções assincronas vao para 1. vendor API, 2. task queue, 3. call stack*

V8

Engine que de fato executa nosso código JavaScript dentro do NodeJS (Call stack e fila nele). Responsável pelo **heap**, onde acontece a **alocação de memória** para nossas variáveis e funções.

// Compila – run – afecta como compila depois / Interpreta / Especula o melhor caminho, volta qdo errado

Compiler - JS engine - Virtual machine (VM)

- Google's open-source JavaScript engine used in Chrome
- Just-in-time compiler
- Speculative optimizations



- Written in C++
- Implements JavaScript according to EcmaScript specification
- New language features
- Garbage collection
- WebAssembly
- No DOM, no console, no file system access

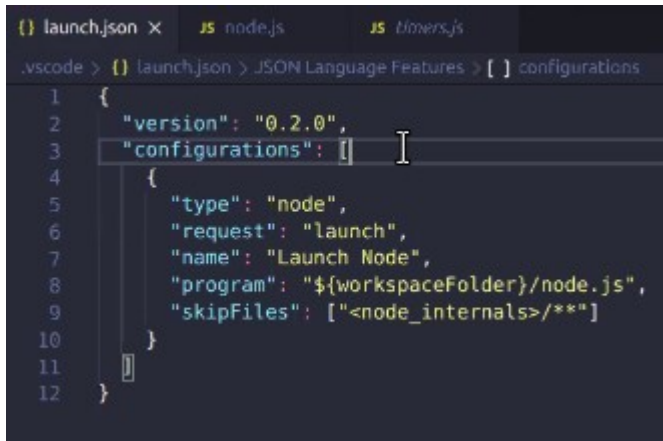


Dicas diversas

Dica para visualizar V8: Consultar [fluxo](#) de como funciona + [completo no app Trybe](#)

Visualizar também na [árvore](#) + [loupe](#).

Debug no VSCode – configurar launch.json criado dentro de pasta .vscode:



```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Node",
      "program": "${workspaceFolder}/node.js",
      "skipFiles": ["<node_internals>/**"]
    }
  ]
}
```

Executar no VSCode: F5

4) Express: HTTP com Node.js

“Express é um framework para aplicativos web escritos em Node.js, mínimo e flexível, que fornece um conjunto robusto de recursos.” - definição dos seus criadores.

Outros elementos sobre express:

- framework na construção de **APIs** com **Node**;
- Responsável por abstrair várias funções que precisaríamos utilizar caso fôssemos trabalhar com requisições **HTTP** diretamente no Node;
- é **unopiniated**: não impõe padrão de desenvolvimento da aplicação;
- Oferece sistema de **rotas e middlewares**.

HTTP

1. Requisição feita pelo navegador

Método de requisição / Caminho / Versão do protocolo

Endereço do host (que estamos acessando) / Headers (informações adicionais)

// Métodos sendo : [GET, PUT, POST, DELETE, PATCH, OPTIONS](#)

GET / HTTP/1.1 //

Host: developer.mozilla.org

2. Resposta do servidor para o cliente - Depois, fecha a conexão TCP.

Versão do protocolo

Código do status

Headers

Body (opcional)

APIs

Application Programming Interface: interface entre apps (web, mobile... clientes) e servidor.
Qualquer coisa que permita a comunicação programática com uma aplicação.
Conversa de sistema para sistema.

Api REST

Usa a Web, usa *protocolo HTTP 1.1*. APIs mais comuns são deste tipo.
Mas APIs podem não ser REST e atuarem em qualquer tipo de sistema, ex. *sistema operacional*.

Endpoints

O que o serviço expõe. Contem propriedades **ABC**: *Address Binding Contract* (onde serviço está hospedado, como serviço está acessado, o que pode ser visto no serviço)

Gerir APIs

Via *Google Cloud Endpoints*. Especificar tipo openAPI ou gRPCAPI.

Segurança

Comunicação - SSL – HTTPs
Autenticação – Tokens

Retorno

Que informação volta? Tipo de informação é inerente a cada API.
JSON e XML são os formatos usados para retornar informações das APIS de web.

Contextualizando as APIs no nosso contexto presente

APIs recebem requisições e devolvem dados, passando por validações, regras de negócio, acesso ao banco de dados, manipulação de dados.

! Métodos, rotas e headers definem o destino das requisições mas podem vir incompletos ou com defeito. Aqui entra o Express.

Introdução ao Express

```
const express = require('express');  
/* Chama a função express para instanciar a aplicação do framework  
   e armazenar na variável app para ser utilizada no código */  
const app = express();  
  
/* Ouve por requisições, utilizando o método GET, no caminho '/' */  
app.get('/', function (req, res) {  
  /* Retorna a resposta */  
  res.send('Hello World!');  
});  
  
/* Ouve a porta 3000 */  
app.listen(3000, function () {  
  console.log('Example app listening on port 3000!');  
});
```

Roteamento

Uma rota (ou endpoint) é definida pelo caminho e pelo método HTTP.

1. Declarar rota

app.METODO(caminho, callback)

Sendo que *callback(request, response, next)*

e Next é opcional, função que diz para o express que aquele callback terminou de ser executado.

Code com diferentes métodos

// Rota com caminho '/', utilizando o método GET|POST|qualquer método HTTP

```
app.get|post|all('/', function (req, res) {  
  res.send('hello world');  
});
```

// Encadear as requisições para evitar repetir o caminho

```
/* Ou podemos encadear as requisições para evitar repetir o caminho */  
app  
  .route('/')  
  .get(function (req, res) {  
    res.send('hello world get');  
  })  
  .post(function (req, res) {  
    res.send('hello world post');  
  });  
  
app  
  .route('/')  
  .get(function (req, res) {  
    res.send('hello world get');  
  })  
  .post(function (req, res) {  
    res.send('hello world post');  
  });
```

// N callbacks para a mesma rota

```
app.get(  
  '/ping',  
  function (req, res, next) {  
    console.log('fiz alguma coisa');  
    /* Chama a próxima callback */  
    next();  
  },  
  function (req, res) {  
    /* A segunda (e última) callback envia a resposta para o cliente */  
    res.send('pong!');  
  }  
);
```

Caminhos e rodas dinâmicas com :, req.params e opcionalmente regex

//id vira um atributo dentro do objeto params, que por sua vez está dentro do objeto req

```
app.get('/api/people/:id', function (req, res) {  
  res.send(req.params.id);  
});
```

QueryString com ?, & e req.query

Caso de receber uma informação na URL que não faz parte do caminho, ex. pesquisa:

https://minha-api.com/endpoint/1?name=exemplo&number=10 - aqui sintaxe ? e &.

Acessar esses valores via req.query:

```
const express = require('express');
const app = express();

app.get('/hello', (req, res) => {
  const name = req.query.name;

  res.status(200)
    .json({ message: `Hello, ${name}` });
})
```

// sempre recebe uma string, usar parseInt quando precisar.

2. Resposta das rotas

Respondem a requisições que satisfaçam a condição método HTTP + caminho .

Middlewares Pattern

* Conhecer as Middlewares

Casos de uso: rotas ou headers incompletos, autenticação, necessidade de functions usadas em várias rotas...

O que é: uma ou mais **funções** que vão rodar antes ou depois da callback/controller da sua rota e depois do servidor receber a requisição. Ou seja, **entre a requisição HTTP e a resposta final**. Middleware pode fazer alterações tanto no request quanto na response.

Estrutura: mesma do que callback de rota

function (req, res, next) {}

Principais usos:

Registrar um middleware com app.use - para aplicar middlewares a todas as rotas, chamados de forma sequencial conforme o momento de registro delas

```
app.use(function (req, res, next) {
  console.log(`${req.method} ${req.path}`)
  /* Termina a operação no middleware e
  chama o próximo middleware ou rota */
  next();
});
```

Passar dados de um middleware para outro ou para a callback da rota →

```
const nameMiddleware = function (req, res,
next) {
  blablamodificationcode;
  next();
};
```

```
const express = require('express');
const app = express();

const requestTimeMiddleware = function (req, res, next) {
  /* Modificamos o objeto req, adicionando o campo requestTime */
  req.requestTime = Date.now();
  /* Chamamos a próxima função */
  next();
};

/* Registramos nosso middleware */
app.use(requestTimeMiddleware);

app.get('/', function (req, res) {
  const responseText = `Request feito às ${req.requestTime}`;
  res.send(responseText);
});

app.listen(3000);
```

* Organizar rotas com Router – Middleware de Rota

Router = Menor app onde declarar apenas rotas e middlewares, associada depois ao app principal.

Middleware de rota = responsável por alternar os destinos de acordo com a rota presente na embalagem do pacote.

```
const express =  
require('express');
```

```
const router = express.Router();
```

Code

```
module.exports = router;
```

```
// simpsons.js  
const express = require('express');  
const router = express.Router();  
  
router.get('/', function (req, res) {  
  res.send('Hello World!');  
});  
  
router.get('/homer', function (req, res) {  
  res.send('Hello Homer!');  
});  
  
module.exports = router;
```

```
// index.js  
const simpsons = require('./simpsons');  
  
/* Todas as rotas com /simpsons/<alguma-coisa> entram aqui e vão para o roteador. */  
app.use('/simpsons', simpsons);
```

* Níveis de Middleware

Middlewares em nível de aplicação / com escopos definidos (por exemplo apenas uma rota) / misturados com Routers.

* Lidar com erros com Middlewares de erro

Padrão error first ou seja como primeiro parâmetro: *function (err, req, res, next) {}* .

Importante: devem vir depois de rotas e outros middlewares e sempre ter 4 parâmetros.

// next(err) indica para o express express que ele não deve continuar executando nenhum middleware ou rota que não seja de erro.

```
app.use(function logErrors(err, req, res, next) {  
  console.error(err.stack);  
  // passa o erro para o próximo middleware  
  next(err);  
});  
  
app.use(function (err, req, res, next) {  
  res.status(500);  
  res.send({ error: err });  
});
```

Pacote express-rescue para garantir tratamento de erros antes do next

```
const rescue = require('express-rescue')  
// const fs = require('fs').promises  
  
app.get('/:fileName', rescue(async (req, res) => {  
  // const file = await fs.readFile('./fileName')  
}));
```

```
const rescue = require('express-  
rescue');
```

e logo usar como callback.

Dicas diversas

Existem ____ métodos **HTTP** diferentes!

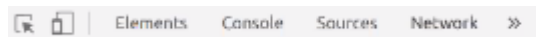
Quiz interface for HTTP methods. It shows a grid of 12 options with numbers 4, 11, 3, and 4. A 'Show media' button is below. Below the grid are two rows of colored boxes with 'X' marks and numbers 39 and 49.

Qual código abaixo faz o servidor escutar na porta 3001

Quiz interface for server listening port. It shows a grid of 12 options with numbers 4, 8, 5, and 8. A 'Show media' button is below. Below the grid are two rows of colored boxes with 'X' marks and a checkmark. The correct answer is 'app.listen(3001, () => console.log('Run'))'.

<http://webconcepts.info/concepts/http-method/>

Para visualizar requisições no browser: **Network**



Para ver objetos de api no terminal: *curl apiurl*

Ferramenta: **Postman**

Postman interface showing a POST request to `http://localhost:3000/hello`. The Headers tab is selected, showing a table with headers.

KEY	VALUE
<input checked="" type="checkbox"/> Content-Type	application/json
Key	Value

Nodemon para assistir alterações, como dependência de desenvolvimento:

instalar *npm i nodemon -D* / ativar *nodemon filename*

app.listen é escutado sempre primeiro

```
app.listen(3000, () => console.log("App ouvindo a porta 3000!"));
```

Para ter url inicial antes de outras

```
app.use("/secure", router);
```

5) Dia de prática

Opções de req: *req.params*, *req.query*, *req.headers*, *req.body* //body transporta parâmetros

Opções de res: *res.status*, *res.json*, *res.send*, *res.end*

// *res.send('lalala')* vira lalala enquanto *res.json('lalala')* vira "lalala"

MIDDLEWARES DE ERRO

```
const errorMiddlewareFunction = (err, req, res, next) => {  
  // req.params  
  // req.query  
  // req.headers  
  // req.body  
  // res.status  
  // res.json  
  // res.send  
  // res.end  
  // next  
};
```

Nos middlewares de erro:

next(err) é como um *reject()* numa promise, vai continuar mas pular middlewares de sucesso para ir para próximos middlewares de erro;

next() é para seguir.

REGISTRANDO MIDDLEWARES

```
// Executado em qualquer requisição  
app.use(middlewareFunction);  
// Executado em qualquer requisição para /path*  
app.use('/path', middlewareFunction);  
  
// Executado em requisições do tipo GET para /path  
app.get('/path', middlewareFunction);  
// Executado em requisições do tipo POST para /path  
app.post('/path', middlewareFunction);  
// Executado em requisições do tipo PUT para /path  
app.put('/path', middlewareFunction);  
// Executado em requisições do tipo DELETE para /path  
app.delete('/path', middlewareFunction);  
  
// Executado em requisições de qualquer tipo para /path  
app.all('/path', middlewareFunction);
```

Status de resposta HTTP:

- [lista](#);
- *res.status(num)* opcional (padrão 200), mas melhor prática escrever, para legibilidade e garantias.

Dias de projeto – Crush Manager

Ver [CRUD na prática](#).

Libs recomendadas

Instalando body-parser, [rescue](#), [@hapi/boom](#) e [joi](#):

```
npm i express body-parser express-rescue @hapi/boom joi
```

Body parser para garantir de poder receber objetos JSON

```
const bodyParser = require('body-parser'); // equivalente ao app.use(express.json());
const middleware = require('./middlewares'); // porém express.json é atalho para o bodyparser, o
const app = express(); // BP é uma melhor pratica para ficar mais
app.use(bodyParser.json()); atualizado diretamente
app.use(middleware.logger);
```

[express rescue](#)

```
app.get(
  '/cats',
  rescue(async (_, res, next) => {
    const { data } = await axios.get(
      'https://api.thecatapi.com/v1/images/search'
    );
    res.status(200).json(data);
  })
);
```

Function que a gente chama, por ela a gente passa um middle async. Retorno é um middleware que chama dentro de um try catch. Rescue basicamente dá try catch por trás dos panos.

// axios em vez de fetch

[hapi.boom](#)

```
if (personIndex === -1) {
  return next(boom.notFound('pessoa não encontrada'));
}
```

```
module.exports = (err, req, res, next) => {
  if (boom.isBoom(err)) {
    return res.status(err.output.statusCode).json(err.output.payload);
  }
}
```

Vantagens: padronizar estatutos HTTP – não precisar montar objeto de erro na mão – se usar lib e tiver status como palavra-chave naquela lib, o boom não vai se confundir.

Underline

```
js index.js > app.get('/ping') callback
1 const express = require('express');
2
3 const app = express();
4
5 app.get('/ping', ([, res]) => {
6   res.json({ message: 'pong!' });
7 });
8
```

// underline _ se não for usar o parâmetro, pode ser _req e _next também mas res tem que sempre ser o 2º

Ferramenta HTTPIE <https://httpie.io/> - Mesmo papel que Postman mas com a praticidade de poder usar no Terminal do VSCode e ganhar eficiência

Boa prática de organizar middlewares

Arquivo middlewares, inclui index.js:

```
const mwname = require('mwname');
```

```
module.exports = { mwname };
```

No mwname.js: `module.exports = (req, res, next) => {}`

No index.js principal, chamar assim:

```
const middlewares = require('./middlewares');
```

```
app.use(middlewares.mwname);
```

```
const express = require('express');
const middlewares = require('./middlewares');

const app = express();

app.use(middlewares.logger);

app.get('/ping', (_, res) => {
  res.json({ message: 'pong!' });
});
```

Token com req.headers.authorization

```
J5 index.js      J5 auth.js      X
middlewares > J5 auth.js > <unknown> > module.exports
1  module.exports = (req, res, next) => {
2    if (!req.headers.authorization) {
3      return next({ status: 401, message: 'no auth token' });
4    }
5  };
6
```

Bônus: escrever middleware auth token com boom para definir se é requerido ou não

```
module.exports = () => {
  // Retorna um middleware
  return (req, res, next) => {
    if (!req.headers.authorization) {
      return next(boom.unauthorized('no auth token'));
    }

    return next();
  };
};
```

```
const boom = require('@hapi/boom');

module.exports = (required = false) => {
  // Retorna um middleware
  return (req, res, next) => {
    if (!req.headers.authorization && required) {
      return next(boom.unauthorized('no auth token'));
    }

    return next();
  };
};
```

```
app.delete('/people/:id', middlewares.auth(true), (req, res) => {
  const { id: stringId } = req.params;
  const id = parseInt(stringId);

  people = people.filter((person) => person.id !== id);

  return res.status(204).end();
});
```

// Parâmetro true para dizer que auth é required antes de poder deletar.

Projeto – para testar localmente: derrubar nodemon, rodar npm start e npm run test.

Mais pacotes usados: [rand-token](#), [moment](#) .
