**NODE.JS**

**23/08 - COMO O NAVEGADOR FUNCIONA?**

O navegador é como uma montadora de veículos: ele reúne pequenos itens (banco, volante, rodas, carcaça, câmbio) e junta tudo isso, formando o carro.

Podemos dizer que o navegador é formado por um conjunto de componentes montados. Dentre eles, temos o principal componente, que é o **motor de interpretação.** É ele que interpreta os nossos comandos quando pesquisamos algo no navegador e responde nossas solicitações. O motor do Google Chrome se chama V8, e ele foi criado em C++ para interpretar JavaScript.

O Node.js surgiu a partir dessa ideia: Ryan Dahl decidiu extrair o V8 do navegador e usá-lo em outro contexto, para que fosse possível utilizar a mesma linguagem de programação usada no lado do cliente também no lado do servidor, para aumento de performance.

O Node.js, então, é **um runtime, ou seja, um ambiente de execução, onde é possível utilizar o JavaScript no lado do servidor.** É claro que a linguagem de programação é a mesma, o que muda são os recursos que ela oferece para cada contexto. No lado do cliente, você não consegue ler, acessar ou modificar pastas, afinal não faz sentido fazer isso nesse contexto. Mas no lado do servidor, você também não consegue acessar métodos de manipulação do DOM ou até mesmo disparar eventos, pois essas coisas não se encaixam no contexto de servidor. Perceba que os recursos mudam a depender do contexto, mas a linguagem é a mesma.

Esse runtime é dividido em três áreas:

* **Core concepts:** conceitos básicos do Node como event loop, conexão não-bloqueante, etc.
* **APIs nativas:** APIs que já vem junto com o Node, como FileSystem e HTTP.
* **APIs de terceiros:** dependências de terceiros que podem ser usadas em algum projeto utilizando o gerenciador de pacotes do Node (npm).

**NPM**

O NPM (cuja sigla é Node Package Manager) é o gerenciador de pacotes do Node. Ele é como uma biblioteca, responsável por instalar dependências e módulos do Node de terceiros na nossa máquina. Quando queremos usar jQuery ou Angular, por exemplo, usamos o comando **npm install** no terminal. Ao fazer isso, estamos pegando aquele “livro” que queremos da prateleira e colocando em nosso projeto, por dizer assim.

Sempre ao instalar algum módulo de terceiros, o Node gera a pasta **node\_modules** e o arquivo **package.json.**

A pasta **node\_modules** contém todos os arquivos das dependências que instalamos em nossa máquina. Ela é uma pasta muito pesada, que geralmente não enviamos para o nosso repositório remoto. Nesse caso, criamos um arquivo chamado **.gitignore** e nele, incluímos os arquivos e pastas que não queremos enviar para o GitHub.

Já o arquivo **package.json** contém um registro de todas as dependências que o nosso projeto precisa ou já possui.

Para criar um arquivo package.json do zero, utilize o comando **npm init** e responda ás perguntas. Caso queira prosseguir imediamente, use a flag **-y.**

Caso tenhamos o registro dos módulos que precisamos no arquivo **package.json**, mas ainda não temos eles instalados na pasta node\_modules, basta executar o comando **npm install.** O Node irá ler o arquivo package.json e instalar os módulos contidos lá.

Um exemplo de módulo de terceiro é o **Nodemon**. Com ele, o nosso código será executado automaticamente, sempre quando fizermos uma alteração e salvarmos.

**NPM vs YARN**

O Yarn é outro gerenciador de pacotes alternativo ao NPM. Eles possuem algumas diferenças evidentes, que são as seguintes:

* O NPM não é do Node, mas já vem instalado com ele. Já com yarn, é necessário fazer a instalação manualmente.
* O yarn é mais rápido que o NPM e seus arquivos na pasta node\_modules ficam mais organizados.
* O NPM possui uma maior variedade de módulos de terceiros do que o yarn.
* O NPM se baseia no **package.json,** enquanto o yarn usa o **yarn.lock.**
* O comando para instalação dos módulos de terceiros usando npm é **npm install (-g)**. No yarn, é **yarn add (global).**

**24/08 – CONEXÃO BLOQUEANTE E NÃO BLOQUEANTE**

Esse é um conceito muito importante e que deve ser compreendido para o pleno uso do Node. Basicamente, uma linguagem de programação executa vários **processos**\*, e esses processos podem ser vários.

**\*Processo:** representa a execução de um programa. Toda vez que abrimos um software, iniciamos um processo.

Em uma conexão **bloqueante,** os processos são executados **um por vez.** Enquanto um processo não termina, o próximo não pode ser executado. Eles ficam aguardando em uma fila para serem executados. Usamos conexão bloqueante na maioria das vezes em códigos **síncronos,** ou seja, que possuem sincronia, que realizam uma tarefa após a outra.

Já na conexão **não bloqueante,** vários processos são executados **ao mesmo tempo**. Nesse cenário, enquanto um processo está sendo executado, outros também são executados simultaneamente. Dessa forma, ganha-se mais tempo, deixa o código com mais performance e proporciona rapidez á aplicação. Usamos conexão não bloqueante em códigos **assíncronos**, ou seja, que não estão em sincronia, que não precisam aguardar a finalização de uma tarefa para realizar outra.

(exemplo: usar promises para fazer requisição AJAX a uma API e obter dados)

**Concorrência**

Concorrência refere-se à capacidade de um sistema de lidar com múltiplas tarefas, processos ou threads simultaneamente. É a ideia de que várias atividades podem estar em andamento ao mesmo tempo. A concorrência é especialmente relevante em sistemas multi-core e ambientes de programação paralela.

No contexto de servidores e aplicações web, a concorrência está relacionada à capacidade de lidar com várias requisições ou tarefas concorrentemente. Em sistemas de servidor, a concorrência é importante para evitar gargalos e atrasos, permitindo que várias solicitações sejam processadas simultaneamente e de forma eficiente.

**Taxa de Transferência**

A taxa de transferência se refere à quantidade de dados que podem ser transmitidos ou processados durante um determinado período de tempo. É uma métrica que mede a eficiência e a capacidade de um sistema de mover dados de um lugar para outro.

No contexto de comunicação de rede, como a internet, a taxa de transferência é frequentemente medida em termos de bits por segundo (bps) ou suas múltiplas, como kilobits por segundo (kbps), megabits por segundo (Mbps) ou gigabits por segundo (Gbps). Uma taxa de transferência alta indica que o sistema é capaz de mover uma grande quantidade de dados rapidamente.

**Relação entre Concorrência e Taxa de Transferência**

A concorrência pode influenciar a taxa de transferência em sistemas que envolvem comunicação e processamento de dados. Quando um sistema é capaz de executar tarefas concorrentemente, isso pode aumentar a taxa de transferência, pois múltiplas tarefas podem ser realizadas ao mesmo tempo, permitindo o processamento mais rápido de dados.

Em servidores web, por exemplo, a concorrência pode influenciar a taxa de transferência devido à capacidade de atender várias requisições simultaneamente. Um servidor altamente concorrente pode processar mais solicitações em paralelo, resultando em uma taxa de transferência mais alta para servir páginas da web, arquivos ou outros tipos de dados.

**CUIDADO**

**É importante seguir um mesmo padrão nos nossos projetos.** Se formos usar um método não bloqueante, devemos usá-lo em todo o restante do código. Também, se formos usar um método bloqueante, devemos usá-lo em todo o restante do código. Misturar métodos bloqueantes com métodos não bloqueantes pode ser catastrófico para a nossa aplicação.

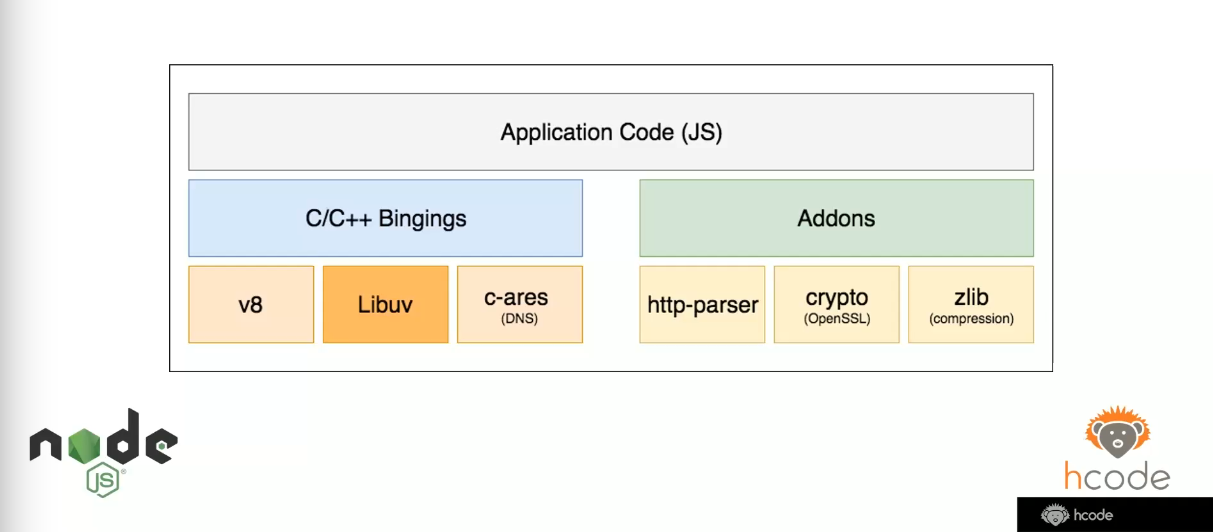
Considere o seguinte exemplo: usamos um método não bloqueante (assíncrono) para ler um arquivo. Em seguida, usamos um método bloqueante (síncrono) para deletar esse mesmo arquivo. Como resultado, o código executado leu parte do arquivo, mas foi interrompido no meio da leitura, porque o método bloqueante excluiu o arquivo.

O correto nesse caso seria usar nos dois métodos as conexões bloqueantes ou não bloqueantes. Dessa forma, o código atingiria seu resultado.

**TIPOS DE APLICAÇÃO NO NODE**

* Aplicações Web
* Interfaces de Linha de Comando – CLI (npm, webpack, backup e sync)
* IoT (Internet das Coisas – Conexão com hardware, dispositivos e sensores)
* Aplicações Desktop

**NÚCLEO DO NODE**

****

O coração do Node é formado por dois componentes principais: o V8 (motor de interpretação do Google Chrome) e a LIBUV.

A LIBUV é uma biblioteca de código aberto desenvolvida em C++ que implementa ao Node um sistema chamado **event loop.** O Node é baseado em um paradigma de programação reativo e orientado a eventos, ou seja, ele vai executando os processos á medida que eventos são disparados. O event loop possibilita ao Node realizar operações não bloqueantes e assíncronas, gerenciando eficientemente múltiplas solicitações.

Além disso, o Node pode ser composto por add-ons, que são módulos nativos que podem ser usados dependendo do contexto do projeto (http-parser para requisições HTTP, crypto para criptografar dados e zlib para comprimir arquivos).

**25/08 – THREAD**

Uma thread representa **uma sequência de instruções que faz parte de um processo.** As threads compartilham o mesmo espaço de endereço de memória e recursos do processo. Isso permite que múltiplas threads de um mesmo processo executem tarefas em paralelo, aproveitando eficientemente os recursos do sistema.

**MULTITHREAD**

Multithreading é uma abordagem em que **um único processo possui múltiplas threads de execução.** Cada thread executa partes independentes do programa, permitindo a execução concorrente de diferentes partes do código.

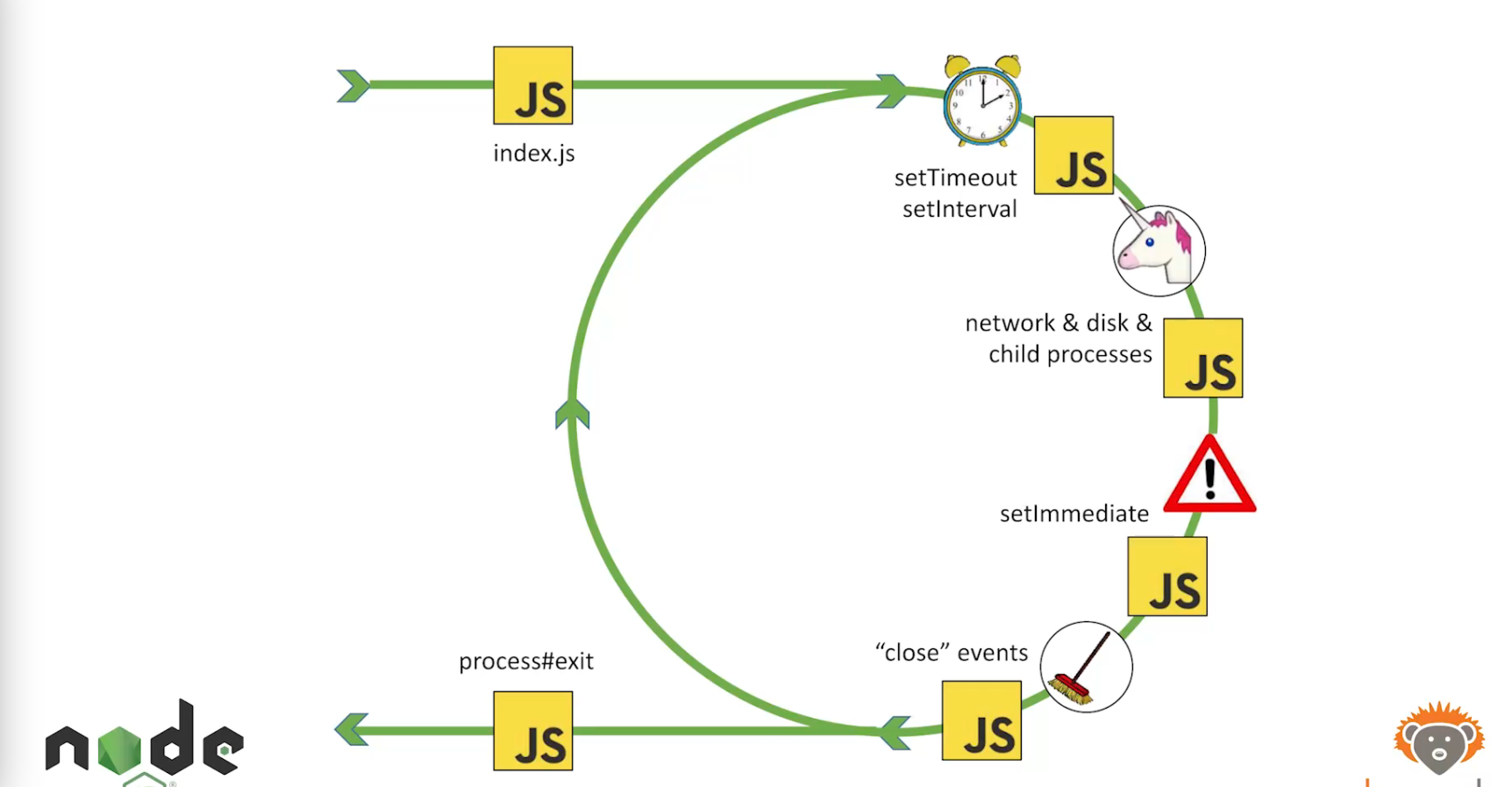
**THREADING POOL**

Um "threading pool" é um grupo de threads que está sempre pronto para realizar tarefas. O Node é singlethread, ou seja, possui apenas uma thread, um único caminho na memória. Mas caso ele tenha que executar um processo muito pesado, **ele designa esse processo para as 4 threads do processador,** para que elas trabalhem em conjunto e processem essas informações mais pesadas, fazendo o threading pool.

**EVENT LOOP**

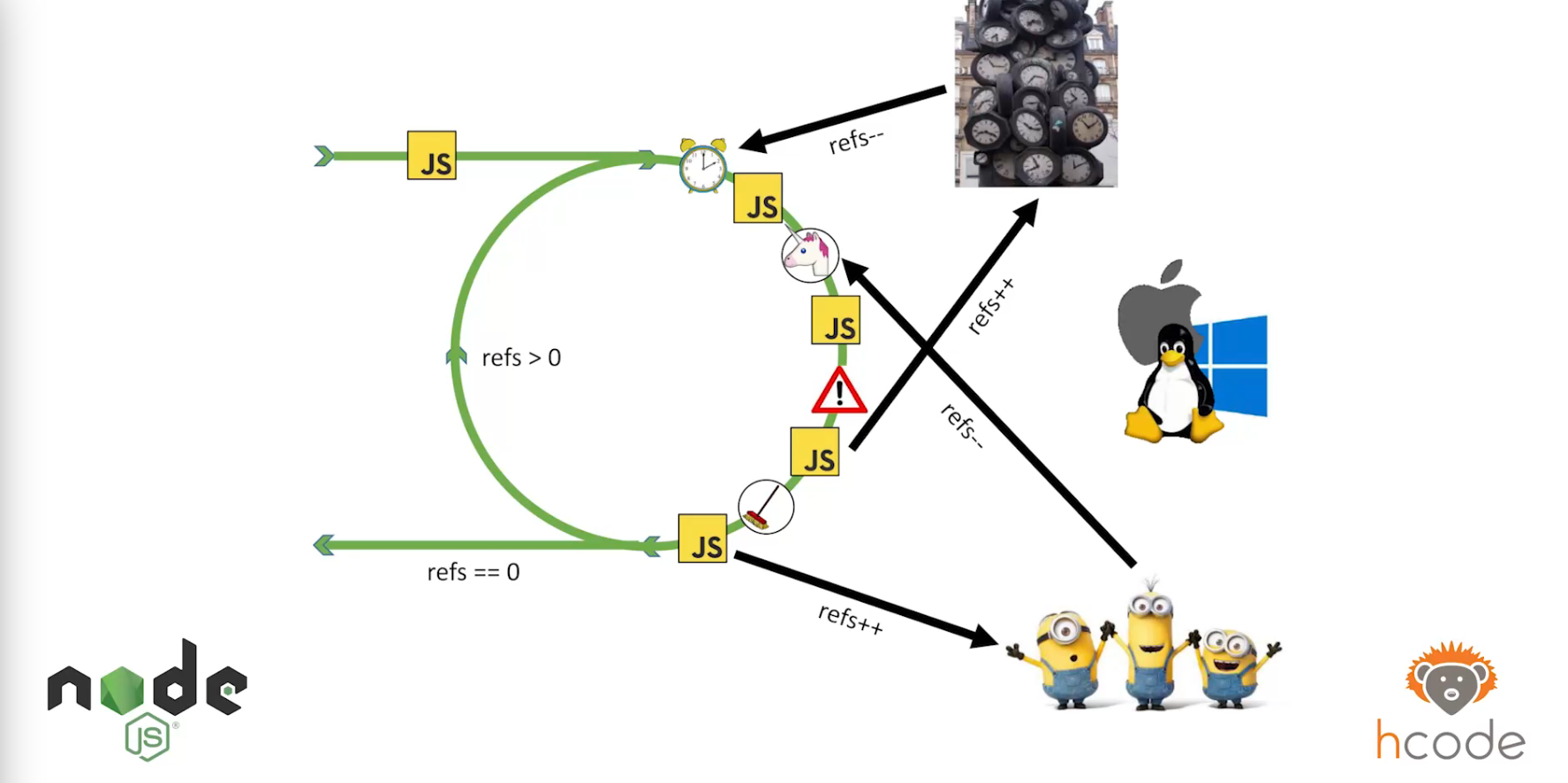
**O loop de eventos é uma sequência de execução de etapas** que acontece toda vez que rodamos um código JavaScript em Node.

Basicamente, um código possui várias etapas durante sua execução, e essas etapas precisam acontecer nos seus tempos determinados. Quando todas as etapas terminam, o Node encerra a execução e o loop de eventos.



Todo event loop possui um contador de referência. Ele que vai determinar se o loop de eventos deve parar ou não. Quando há algum setTimeout ou setInterval (primeira etapa), o contador de referência adiciona 1 e aguarda essa etapa terminar. A segunda etapa são funções unicórnio, usadas para consultar dados de rede, disco e processos (algo mais relacionado ao hardware). Assim por diante, o processo é realizado.

Quando o contador de referência fica em 0, o event loop é encerrado.



**NODEMON**

O Nodemon é um módulo de terceiros do Node que nos permite **monitorar as alterações de um arquivo.**

Por padrão, quando executamos um arquivo e depois fazemos alguma alteração nele, precisamos ir no terminal novamente para rodar o código atualizado.

Com o Nodemon, isso não é mais necessário. Após instalado de maneira global por meio do comando **npm i -g nodemon**, usamos o comando **nodemon file.js** e pronto! O Nodemon fica sempre monitorando aquele arquivo, e qualquer alteração feita nele, o código será executado novamente.

**CÓDIGO ASSÍNCRONO**

Em Node, as APIs são assíncronas. Isso significa que elas executam seus processos de maneira não bloqueante, ou seja, enquanto um processo está sendo executado, ele executa os outros processos restantes.

Quando fazemos uma requisição a uma API, por exemplo, enquanto o Node faz essa requisição, ele vai executando os outros processos do código em paralelo. Isso é o **assincronismo.** Você não precisa esperar a resposta da requisição para então executar os processos restantes. Enquanto a requisição acontece, o Node executa outros processos paralelamente.

Existem três maneiras de tratar código assíncrono em Node, as quais estão listadas a seguir:

* **Funções de callback:** as callbacks são funções de retorno, ou seja, funções passadas como parâmetro de outras funções. Elas só serão executadas quando a função anterior for executada.

Essas funções sempre possuem dois parâmetros: o primeiro é o erro, caso ele aconteça. O segundo é o resultado, caso ele tenha sido obtido.

Com as funções de callback, é possível deixar o código assíncrono, mas não elegante. Quando você tem muitas callbacks, o código vai ficando maior e mais indentado, o que chamamos de Inferno de Callbacks (Callback Hell). Para solucionar isso, vieram as Promises.

* **Promises:** as Promises são um modo mais elegante que temos de tratar código assíncrono em JavaScript. Elas são literalmente promessas, onde o JavaScript promete que irá retornar os dados de uma requisição (exemplo), mas que pode dar certo ou errado. Por isso, uma Promise recebe uma função de callback com dois parâmetros: **resolve** (com os dados da requisição, caso tenha dado certo) e **reject** (com as informações do erro, caso tenha dado errado).

Após executada a promessa, nós podemos decidir o que fazer com os dados da requisição por meio do método **then()**, executado quando a promessa dá certo. O parâmetro de then() é o que foi colocado como parâmetro do resolve(), lá dentro da Promise.

Além disso, caso a requisição tenha falhado, podemos exibir o erro usando o método **catch()**, executado quando a promessa dá errado. O parâmetro de catch() é o que foi colocado como parâmetro do reject(), lá dentro da Promise.

* **Async/Await:** o async/await é um modo mais elegante ainda de trabalharmos com código assíncrono. Fazendo uso desse modo, o código fica parecendo síncrono.

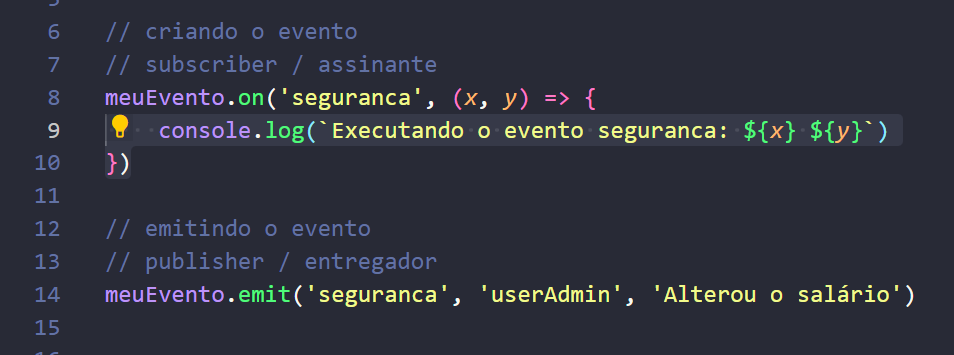
Funciona da seguinte maneira (exemplo da requisição): a função que vai exibir os dados de uma requisição deve possuir a palavra reservada **async**, para que seja possível usar o **await.** Ao executarmos a função que retorna uma Promise com os dados da requisição, precisamos usar o await antes, para determinar que é necessário esperar aquele retorno para prosseguir com o código.

Geralmente, usamos blocos de try/catch para deixar o código mais legível e versátil.

**EVENTOS NO NODE**

Assim como no front-end, é possível criar e emitir eventos no back-end também. Eventos são gatilhos disparados por algum motivo. O bom de criarmos eventos é que eles trabalham o tempo inteiro na nossa aplicação. O Node escuta esses eventos o tempo todo, e quando acontecem, ele executa algo. Podemos criar um evento em Node da seguinte maneira:

* 1. Fazer o require do módulo ‘events’.
* 2. Armazenar isso em uma variável (exemplo: EventEmiiter).
* 3. Criar uma classe (exemplo: Evento) que extende da classe EventEmiiter.
* 4. Instanciar essa classe em alguma variável, para poder fazer uso de seus métodos.
* 5. Usar o método **on()** dessa classe. Esse método possui dois parâmetros: o nome do evento e a função de callback que será executada quando ele for disparado.
* 6. Para emitir esse evento, basta usar o método **emit()** também dessa classe. Deve-se passar como parâmetro o nome do evento e os parâmetros passados na função de callback (se houver).



**VARIÁVEIS INTERNAS DO NODE**

Nome do arquivo: **\_\_filename**

Diretório do arquivo: **\_\_dirname** // de onde veio

Diretório em que foi invocado: **process.cwd()** // de onde chamou

Parâmetros de execução: **process.argv**

Ambiente do servidor: **process.platform**

Variáveis de ambiente: **process.env**

Sistema operacional: **process.env.OS**

Login da máquina: **process.env.USERNAME**

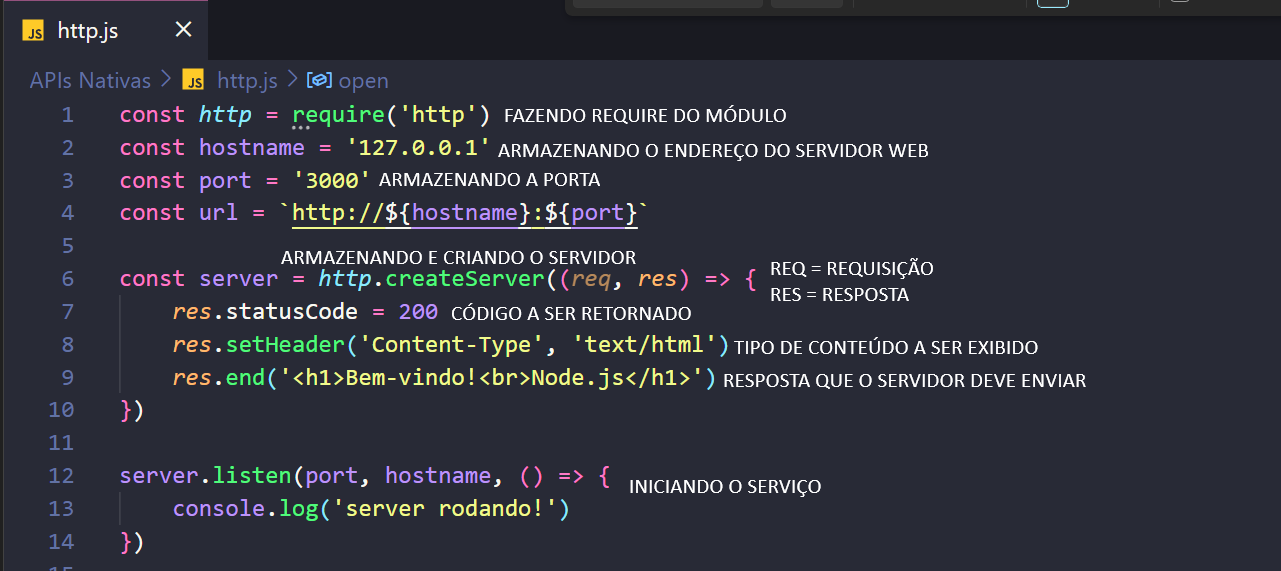
Idioma: **process.env.LANG**

Nome do servidor: **process.env.COMPUTERNAME**

Encerrar execução do código: **process.exit()**

**30/08 – APIs NATIVAS DO NODE**

**HTTP:** esse módulo nativo do Node nos permite criar um servidor Web. Ele é semelhante ao Express, que é bem mais simples.



**FS (FileSystem):** esse módulo nativo do Node nos permite manipular arquivos e pastas de um sistema operacional.

O módulo FileSystem possui alguns métodos para manipulação de pastas e arquivos. São estes:

**fs.readdir():** leitura do conteúdo de diretórios (pastas)

**fs.readFile():** leitura do conteúdo de arquivos.

**fs.mkdir():** criação de diretórios (pastas).

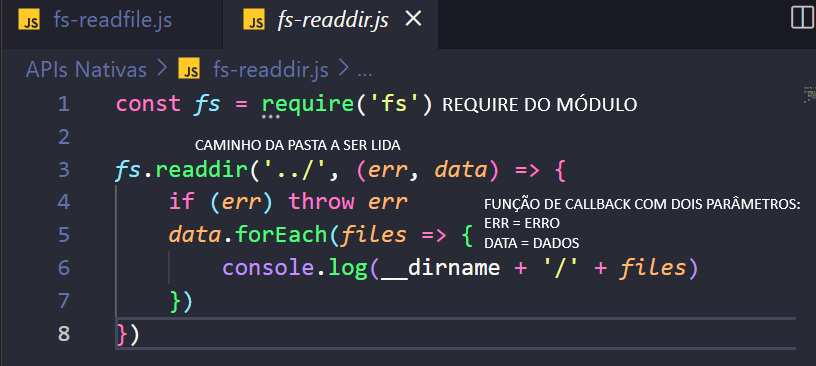
**fs.writeFile():** criação de arquivos.

**fs.appendFile():** atualizar o conteúdo de um arquivo.

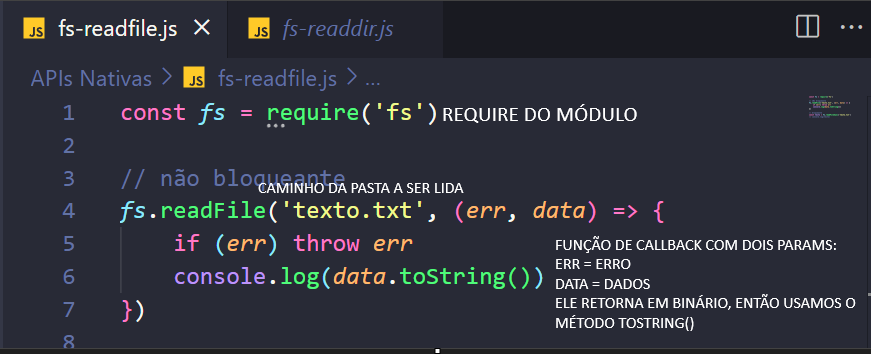
**fs.unlink():** excluir um diretório ou arquivo.

**fs.rename():** renomear um arquivo.

**Leitura do conteúdo de pastas (readdir):**

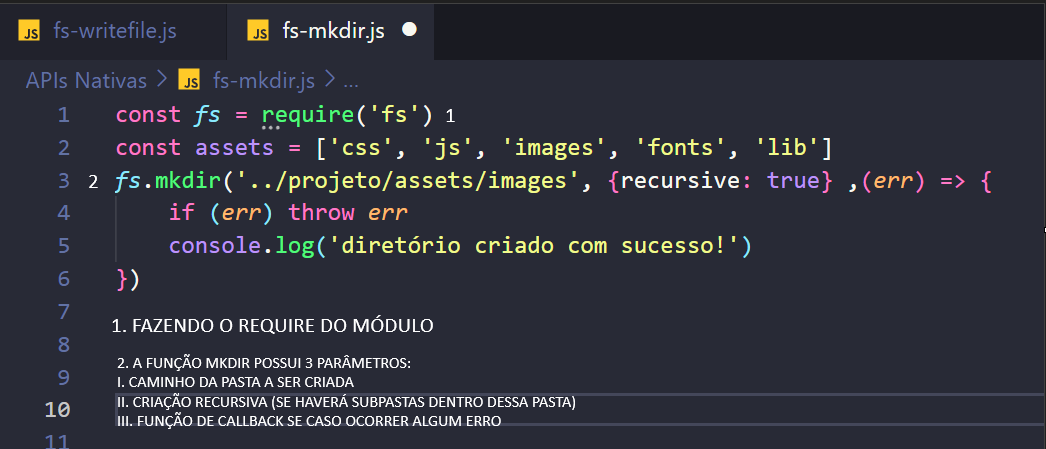


**Leitura do conteúdo de arquivos (readFile):**

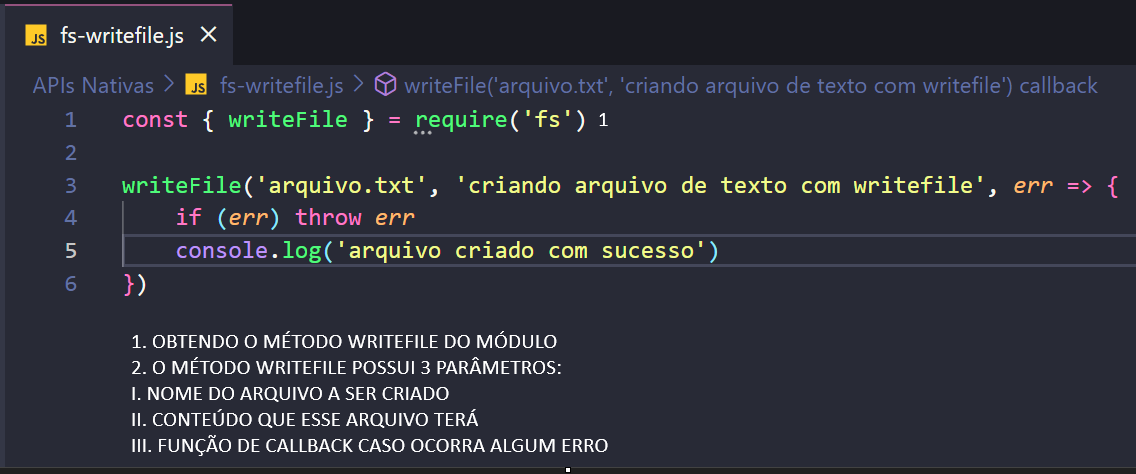


**Criação de pastas (mkdir):**

**\*** É possível usar o forEach no array de assets e criar várias pastas de um vez.

****

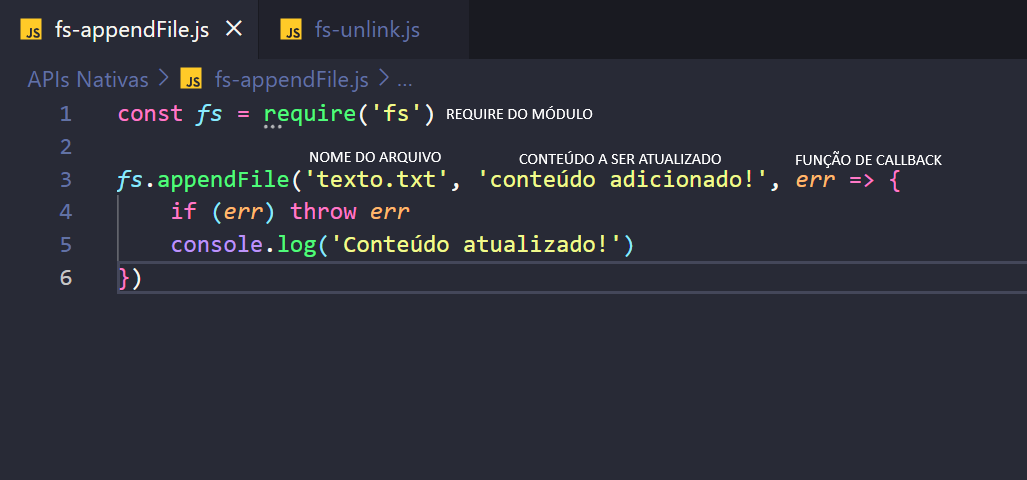
**Criação de arquivos (writefile):**

****

**Criação de arquivos utilizando Promises:**

****

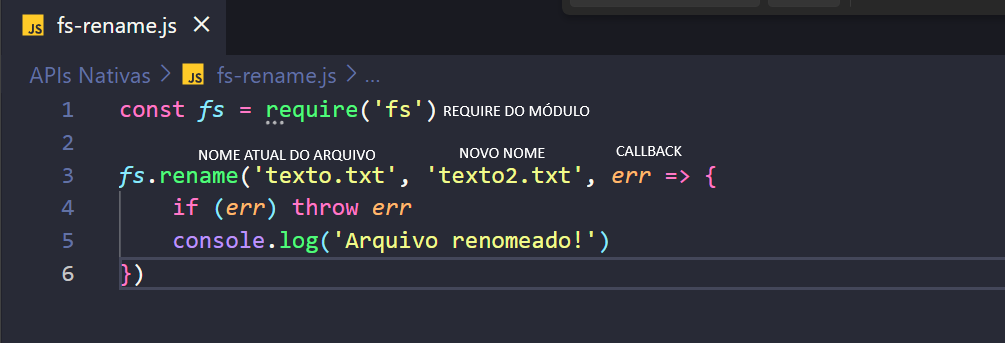
**Atualizar o conteúdo de arquivos (appendFile):**

****

**Excluir um arquivo (unlink):**

****

**Renomear arquivos (rename):**

****

**API de Console:** essa API nativa do Node nos permite exibir algum conteúdo no terminal, enquanto rodamos a nossa aplicação.

**console.log():** exibe uma mensagem no console.

**console.error():** exibe uma mensagem de erro no console.

**console.table():** exibe o conteúdo de um array ou de um objeto de maneira tabular (tabela).

**console.count(‘tarefa’):** cria um contador para determinar quantas vezes uma tarefa é executada no código.

**console.countReset(‘tarefa’):** reseta o contador criado.

**console.time(‘tarefa’):** começa a contar o tempo que leva para uma tarefa ser concluída.

**console.timeEnd(‘tarefa’):** termina de contar o tempo de uma tarefa e exibe o tempo dela no console.

**console.assert(‘condição’, ‘mensagem’):** exibe uma mensagem no console apenas se uma condição for falsa.

**console.clear():** limpa as mensagens do terminal.

**Child process:** essa API nativa do Node nos ajuda a executar processos em segundo plano.

**spawn():** inicia um novo processo com o comando recebido. Podemos armazenar a execução dessa função em uma variável e então usar os atributos **stdin** (entrada de dados), **stdout** (saída de dados) e **stderr** (mensagens de erro).

**Path:** essa API do Node serve para manipular caminhos de arquivos e de pastas.

**basename():** retorna a última parte do caminho de um diretório.

**resolve():** retorna o caminho absoluto de um arquivo.

**join():** junta os caminhos especificados em um único caminho.

**extname():** retorna apenas a extensão de um arquivo.

**Error:** API nativa para manipulação de erros.

**throw new Error(‘message’):** dispara um erro no console, exibindo a mensagem passada como parâmetro

**Buffer:** API nativa para manipulação de dados binários.

**Buffer.from(‘string’):** recebe uma string e transforma ela em binário.

**buf.toString():** transforma um elemento binário em string.

**Buffer.isBuffer(buf):** retorna um booleano se determinado elemento é binário.

**02/09 - SISTEMA DE MÓDULOS**

Em Node, cada arquivo é um **MÓDULO**. Podemos escrever vários códigos dentro de um arquivo, mas esses códigos ficam visíveis apenas dentro do arquivo. Mas nós podemos tornar isso visível para outros arquivos, usando o conceito de **importação e exportação.**

A forma mais comum de **exportarmos** um arquivo, ou seja, permitirmos que os métodos e atributos dele sejam usados em outro módulo, é usando o **module.exports = { },** onde tudo que vier dentro desse objeto poderá ficar visível para outros módulos.

E quando nós quisermos fazer a **importação** de um arquivo, ou seja, a permissão para usar os métodos e atributos de um arquivo em outro, usamos o **require(‘./arquivo’).** Assim, se esse arquivo requerido tiver um module.exports, poderemos usar os recursos dele em nosso outro arquivo.

**04/09 – EXPRESS**

O Express é um framework muito utilizado no Node para simplificar o desenvolvimento backend de uma aplicação. Com esse framework, podemos criar servidores web, APIs REST e micro serviços.

O Express nos possibilita trabalhar com os seguintes conceitos:

* **Rotas:** caminhos que mudam o comportamento da aplicação.
* **Middlewares:** conceito utilizado pelo Express onde as funções são executadas em cadeia, uma atrás da outra. Esses middlewares interferem na requisição que o usuário faz e podem executar tarefas específicas, como validação de dados, autenticação de usuários, entre outros. **\***
* **Templates:** código HTML otimizado para conversar melhor com o backend da aplicação.
* **MVC:** arquitetura de projeto onde ele se divide em camadas: o **Model** é onde se encontra a manipulação dos dados da aplicação. **View** é toda a parte visual (HTML, CSS) e **controller** é onde são determinadas as regras de negócio.

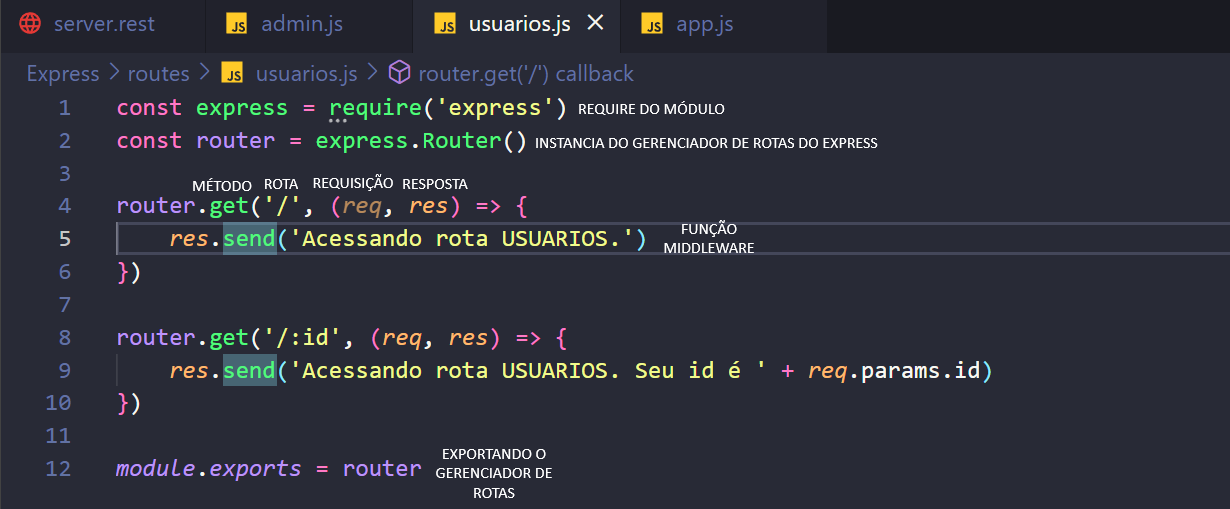
\* **Tipos de middlewares:**

* **Em nível de aplicação:** funções que queremos que sejam executadas enquanto nossa aplicação está rodando. Todos os middlewares que se encontram dentro de **app.use()** são considerados middlewares em nível de aplicação.
* **Em nível de rota:** funções que manipulam rotas específicas.
* **Erros:** middlewares para manipulação de erros. Geralmente, esses middlewares ficam no final do código e seu primeiro parâmetro é o erro, caso ele ocorra.
* **De terceiros:** funções middlewares que não são nativas do Node, mas nos ajudam a executar tarefas específicas na aplicação.

**\* Alguns middlewares de terceiros, mas que são mantidos pelo Express:**

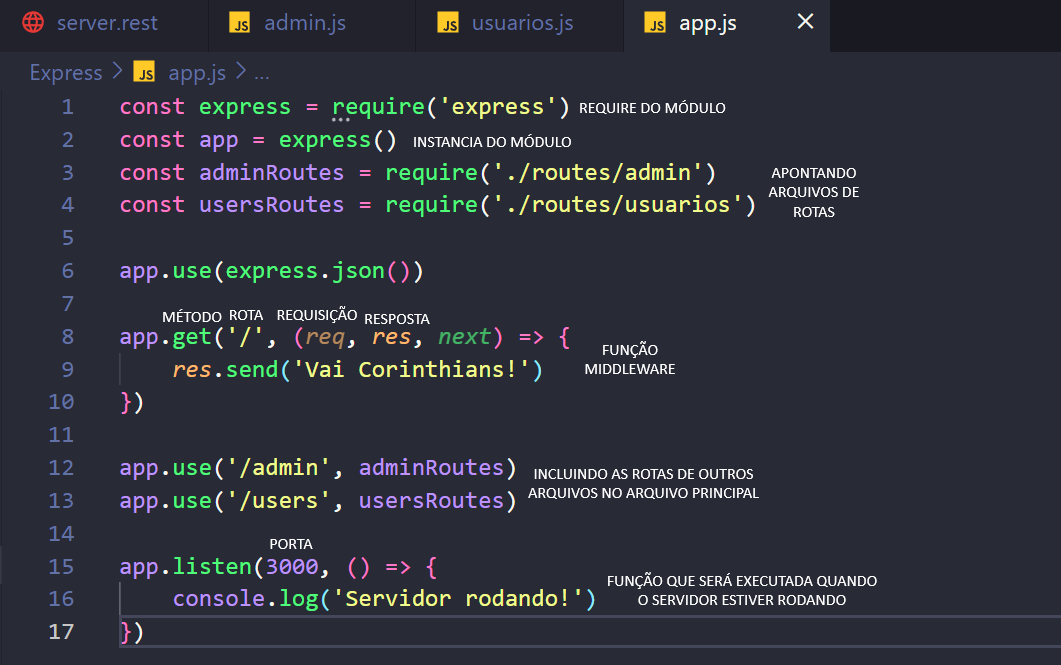
* **express.json():** interpreta dados em JSON que vem na requisição.
* **express.static():** renderiza arquivos estáticos (HTML, CSS). O parâmetro da função é o diretório da pasta public, onde geralmente ficam os arquivos estáticos.
* **body-parser:** interpreta os dados que vem no corpo da requisição (muito usado para solicitações do tipo POST).
* **cors:** permite realizar requisições via AJAX.
* **cookie-parser:** cria e configura cookies.
* **cookie-session:** armazena informações de cookies.

**CRIANDO UM SERVIDOR WEB COM EXPRESS**

****

Nessa primeira captura, observamos as rotas da página de usuários serem armazenadas em um arquivo separado, chamado **users.js.** Além de instanciarmos o Express, instanciamos também o gerenciador de rotas dele, o **express.Router().**

A partir daí, criamos as rotas normalmente, sem a necessidade de incluirmos a raiz **/users.** E por fim, exportamos o router para usarmos ele no arquivo principal.

****

Agora no arquivo principal, fazemos o require do router que se encontra lá no arquivo das rotas de **/users** (linha 4), e criamos uma função middleware com **app.use()** (linha 13) para determinar que quando o usuário acessar a rota **/users** (primeiro parâmetro), serão aplicadas as rotas que se encontram em **usersRoutes** (router da página de usuários – segundo parâmetro.)

**06/09 – MONGODB**

O MongoDB é um banco de dados não relacional orientado a documentos, ou seja, os dados são armazenados em documentos no formato BSON (Binary JSON). Cada documento possui coleções, e cada coleção possui registros.

Esse banco de dados não relacional possui esquemas altamente flexíveis. Os registros podem ter campos diferentes e a estrutura dos dados não é previamente definida. Além disso, o MongoDB é altamente escalável e pode ser usado para lidar com grandes volumes de dados.

A seguir, alguns comandos do MongoDB para manipularmos os dados desse banco:

**use dbname:** cria uma coleção/esquema no banco de dados. A partir disso, é possível inserir, encontrar, atualizar ou deletar os registros dessa coleção.

**insertOne() e insertMany():** criam um registro na coleção. O método insertOne() insere apenas um registro, enquanto o insertMany() insere vários registros, todos dentro de um array. [ ]

**find():** retorna todos os registros de uma coleção. Pode incluir um filtro como parâmetro opcional.

**updateOne() e updateMany():** edita o registro de uma coleção.

Possui dois parâmetros:

1. um filtro para determinar qual registro será editado,

2. o que será editado nesse registro (usa-se a flag $set).

O método updateOne() edita apenas um registro, enquanto o updateMany() atualiza vários registros, incluindo-os em um array. [ ]

**deleteOne() e deleteMany():** delete o registro de uma coleção. Possui como parâmetro um filtro para determinar o(s) registro(s) que será(ão) deletado(s).

O método deleteOne() deleta apenas um registro, enquanto o deleteMany() deleta vários registros, incluindo-os em um array. [ ]

**08/09 – API**

A sigla API deriva da expressão inglesa Application Programming Interface que, traduzida para o português, pode ser compreendida como uma interface de programação de aplicação. Ou seja, API é um **conjunto de normas que possibilita a comunicação entre plataformas por meio de uma série de padrões e protocolos.**

Por meio de APIs, desenvolvedores podem criar novos softwares e aplicativos capazes de se comunicar com outras plataformas. Por exemplo: caso um desenvolvedor queira criar um aplicativo de fotos para Android, ele poderá ter acesso à câmera do celular através da API do sistema operacional, sem ter a necessidade de criar uma nova interface de câmera do zero.

O mesmo acontece com aplicativos que utilizam os serviços de mapas por meio da API do Google Maps ou, ainda, nas integrações entre apps, como o Spotify e o Instagram, que possibilita compartilhar faixas nos Stories.

Exemplo: Quando você vai a um restaurante, você não entra na cozinha para fazer o seu próprio prato, você olha o menu, escolhe o que deseja e o garçom leva o seu pedido para a cozinha. A cozinha prepara a comida e o garçom entrega a você.

Da mesma forma, uma API é como o garçom: um menu que permite que um programa de computador (ou aplicativo) peça informações ou ações de outro programa, serviço ou sistema, sem precisar entender como tudo funciona nos bastidores.

**API REST**

REST é a sigla para Representational State Transfer e representa um conjunto de restrições para uma API.

Para uma API ser considerada do tipo RESTful, ela precisa está em conformidade com os seguintes critérios:

* Ter uma arquitetura cliente/servidor formada por clientes, servidores e recursos, com solicitações gerenciadas por HTTP.
* Estabelecer uma comunicação stateless entre cliente e servidor. Isso significa que nenhuma informação do cliente é armazenada entre solicitações GET e todas as solicitações são separadas e desconectadas.
* Armazenar dados em cache para otimizar as interações entre cliente e servidor.
* Ter uma interface uniforme entre os componentes para as informações serem transferidas em um formato padronizado. Para tanto, é necessário que:

1. Os recursos solicitados sejam identificáveis e estejam separados das representações enviadas ao cliente;

2. Os recursos possam ser manipulados pelo cliente por meio da representação recebida com informações suficientes para tais ações;

3. As mensagens autodescritivas retornadas ao cliente contenham informações suficientes para descrever como processá-las;

4, Hipertexto e hipermídia estão disponíveis. Isso significa que após acessar um recurso, o cliente pode usar hiperlinks para encontrar as demais ações disponíveis para ele no momento.

* Ter um sistema em camadas que organiza os tipos de servidores (responsáveis pela segurança, pelo carregamento de carga e assim por diante) envolvidos na recuperação das informações solicitadas em hierarquias que o cliente não pode ver.

**MÉTODOS HTTP**

Métodos HTTP são formas de acesso. Por meio deles, nós dizemos ao servidor que tipo de solicitação nós queremos fazer para ele. A seguir, os métodos HTTP mais conhecidos são:

**C ---------------** create ----------- **POST:** enviar/inserir dados

**R** ---------read ------------ **GET:** obter dados

**U** --------------- update ---------- **PUT:** editar dados

**D** -------------- delete ------------ **DELETE:** excluir dados

**SEGURANÇA**

É necessário garantir a segurança da nossa API, para que pessoas indesejadas não a acessem. Para isso, podemos usar o middleware do módulo **Passport** para verificar a integridade dos dados da pessoa quando ela acessa uma rota e então autorizá-la a prosseguir na aplicação ou recusar seu acesso.

Caso essa pessoa tenha sido autorizada, podemos usar o **JWT** (JSON Web Token) e gerar um token de autenticação para que essa pessoa possa navegar normalmente na aplicação. Além disso, para evitar que hackers convertam o token e o roubem, podemos usar o módulo **crypto** para criptografar o token, e assim garantir a segurança da REST API.

**LIDANDO COM DADOS SENSÍVEIS**

Quando nossa aplicação possui dados sensíveis, como tokens de configuração, senhas e endereços de conexão, não deixamos essas informações públicas. Incluímos elas nas variáveis de ambiente para garantir a segurança dessas informações. Para fazer isso, podemos:

1. Instalar o módulo **dotenv**
2. Executar no arquivo do nosso servidor: **require(‘dotenv’).config()**
3. Criar um arquivo chamado **.env** e ali dentro, armazenar em variáveis os dados sensíveis.

Suponhamos que eu tenha criado uma variável chamada PASSWORD para armazenar uma senha secreta. Fazendo todos os processos acima, podemos acessar esse dado utilizando **process.env.PASSWORD**. Dessa forma, a senha não estará visível.

**TESTAR ROTAS**

Existem softwares que nos possibilitam testar uma requisição a determinada rota da nossa aplicação. Os principais são:

**REST CLIENT:** extensão do VSCode. Em um arquivo com a extensão **.rest**, podemos digitar algo como GET <http://localhost:3000> e clicar no botão “Send Request”. Isso irá fazer uma requisição do tipo GET a esse endereço.

**POSTMAN:** uma plataforma mais interativa e completa, possibilitando ver uma maior amplitude dos dados da requisição.

**INSOMNIA:** outra plataforma parruda e completa assim como o Postman, que também nos dá uma ampla visão dos dados da requisição.