



PROGRAMAÇÃO COM NODE.JS

Allan Crasso Naia de Souza

PROGRAMAÇÃO COM NODE.JS

1ª edição

São Paulo
Platos Soluções Educacionais S.A
2022

© 2022 por Platos Soluções Educacionais S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Platos Soluções Educacionais S.A.

Head de Platos Soluções Educacionais S.A

Silvia Rodrigues Cima Bizatto

Conselho Acadêmico

Alessandra Cristina Fahl

Camila Braga de Oliveira Higa

Camila Turchetti Bacan Gabiatti

Giani Vendramel de Oliveira

Gislaine Denisale Ferreira

Henrique Salustiano Silva

Mariana Gerardi Mello

Nirse Ruscheinsky Breternitz

Priscila Pereira Silva

Tayra Carolina Nascimento Aleixo

Coordenador

Henrique Salustiano Silva

Revisor

Tatiele Martins Razera

Editorial

Beatriz Meloni Montefusco

Carolina Yaly

Márcia Regina Silva

Paola Andressa Machado Leal

Dados Internacionais de Catalogação na Publicação (CIP)

T454s Souza, Allan Crasso Naia de
Programação com Node.JS / Allan Crasso Naia de
Souza. – São Paulo: Platos Soluções Educacionais S.A.,
2022.
32 p.

ISBN 978-65-5356-203-5

1. JavaScript. 2. Tecnologia Node.Js. 3. APIs. I. Título.

CDD 005

Evelyn Moraes – CRB: 010289/O

2022

Platos Soluções Educacionais S.A

Alameda Santos, nº 960 – Cerqueira César

CEP: 01418-002— São Paulo — SP

Homepage: <https://www.platosedu.com.br/>

PROGRAMAÇÃO COM NODE.JS

SUMÁRIO

Apresentação da disciplina	05
Desenvolvimento Web Back-end	07
Desenvolvimento Web Back-end II	17
Desenvolvimento Web Back-end III	28
Desenvolvimento Web Back-end IV	37



Apresentação da disciplina

Seja bem-vindo à disciplina *Programação com Node.js*. Nosso objetivo é apresentar os princípios do desenvolvimento de aplicações web, a utilização da linguagem de programação de script, de acordo com o ECMAScript, a estrutura do Node.js e o processo de instalação e configuração da plataforma.

Vamos iniciar a disciplina abordando a linguagem JavaScript, que inicialmente surgiu para implementar interatividade em páginas web, a princípio somente voltada para o desenvolvimento *front-end*. Isso mudou com o surgimento do Node.js, que permite a utilização do JavaScript para a programação back-end. Podemos dizer que, com a chegada do Node.js, o JavaScript se tornou uma linguagem de programação *full stack*.

O Node.js é um framework que permite criar aplicação em JavaScript sem depender de um browser para execução. Isso é possível graças ao fato de ser uma aplicação *single-thread*, em que uma única *thread* executa o código JavaScript. Em outras linguagens, ele é *multi-thread*. Logo, essa característica traz uma performance para as aplicações Node.js, fazendo com que sejam indicadas para o desenvolvimento, apesar de não ser seu foco, de jogos e até aplicações para inteligência artificial. Isso porque são do tipo *non-blocking*, e, portanto, essas *threads* não são bloqueadas.

Ao longo da disciplina, serão abordadas ainda as formas que o Node.js usa para gerenciar as sessões e o controle de cache. Além disso,

entenderemos como a ferramenta utiliza o controle de fluxo e de que forma isso impacta no ciclo de vida da aplicação, chegando ao tópico de utilização de APIs para várias funcionalidades, como conexão com uma determinada base de dados.

Diante dessa explanação resumida da disciplina, nós o convidamos para ler os materiais teóricos, assistir às videoaulas e acompanhar os podcasts sobre nosso assunto. Bons estudos!

Desenvolvimento Web Back-end

Autoria: Allan Crasso Naia de Souza

Leitura crítica: Tatiele Razera



Objetivos

- Conhecer os princípios de desenvolvimento de aplicações web.
- Utilizar linguagem de programação de script de acordo com os padrões do ECMAScript.
- Conhecer a estrutura do Node.js e o processo de instalação e configuração da plataforma.

1. Node.js

Antes de iniciarmos o estudo sobre Node.js, precisamos abordar a linguagem de programação JavaScript, que foi desenvolvida para proporcionar maior interatividade às páginas web utilizando a manipulação do DOM (*Document Object Model*). Dessa forma, rodava os conteúdos, o *client-side*, com maior performance, tornando-os mais dinâmicos.

Um fato que precisa ser salientado é que o JavaScript nasceu em uma época em que a maioria das páginas era estática e sem interação com o usuário. Logo, ele veio para proporcionar uma interatividade voltada ao *front*. Como atendeu bem ao seu papel, caiu no gosto da comunidade e logo os desenvolvedores da época, como o Ryan Dahl, perguntavam-se sobre o motivo de não utilizarem a linguagem no lado do servidor também, proporcionando otimização de processos e serviços. Assim, o Node.js surge como uma alternativa plausível para o desenvolvimento *back-end* utilizando a linguagem JavaScript.

Segundo a sua documentação oficial, o Node.js foi escrito em C++ e em JavaScript. Com o intuito de impulsionar o Node, Dahl utilizou a *engine V8* do JavaScript, também utilizada no Google Chrome, o que permitiu aos desenvolvedores criar páginas maduras que antes eram desenvolvidas em Java ou PHP. A partir da criação do Node.js, o JavaScript passa a ser reconhecido como uma linguagem confiável rodando no lado do servidor.

No final de 2009, Ryan Dahl com a ajuda inicial de 14 colaboradores criou o Node.js. Esta tecnologia possui um modelo inovador, sua arquitetura é totalmente *non-blocking thread* (não-bloqueante), apresentando uma boa performance com consumo de memória e utilizando ao máximo e de forma eficiente o poder de processamento dos servidores, principalmente em sistemas que produzem uma alta carga de processamento. (PEREIRA, 2016, p. 2)

Figura 1 – Logotipo do Node.js



Fonte: <https://nodejs.org/en/about/resources/>. Acesso em: 28 mar. 2022.

O Node.js também é conhecido pela sua característica de ser escalável e de baixo nível, o que proporciona ao desenvolvedor programar diretamente com diversos protocolos de rede ou fazer uso das bibliotecas que acessam recursos do sistema operacional. Além dessas características, outro ponto forte, que será abordado com maior riqueza de detalhes a seguir, é ser assíncrono, trabalhando com uma única *thread*. Por ser *single-thread*, o Node.js não bloqueia o processo em questão e, dessa forma, pode atender a um volume absurdamente grande de requisições ao mesmo tempo.

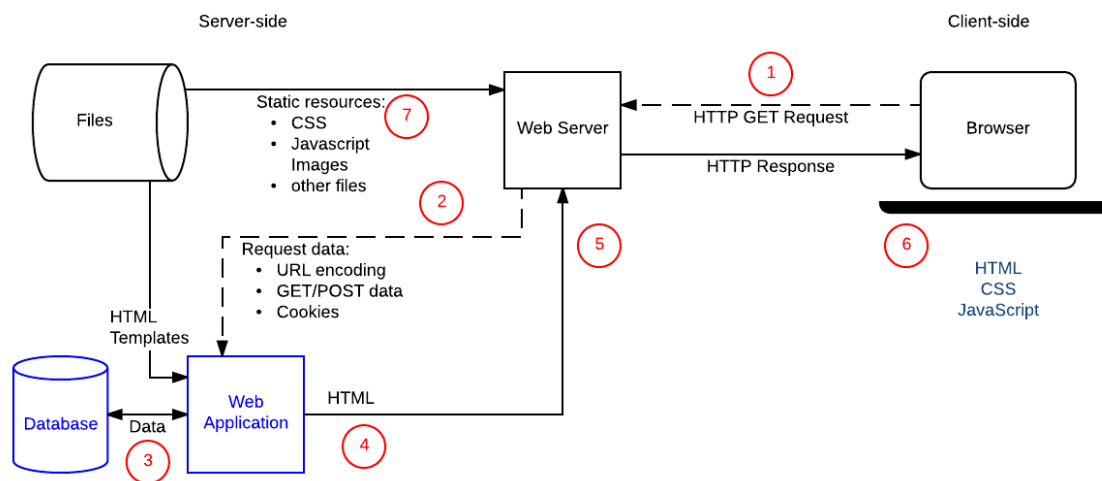
1.1 Scripts lado do servidor (*server side*)

O script executado no servidor web, ou *server side*, é uma técnica de desenvolvimento que visa executar o software no servidor, proporcionando que algumas operações, como a personalização de uma página web, a alteração dinâmica no conteúdo do site, a geração de respostas às consultas e o acesso ao banco de dados, rodem no servidor. De forma resumida, é o que conhecemos como *back-end*.

Não podemos falar de *server side* sem deixar de abordar tecnologias como servidor, banco de dados, API e o próprio produto, ou aplicação *web back-end*, desenvolvido utilizando uma linguagem de script *server-side*, nosso famoso JavaScript. Antes de falarmos sobre seu funcionamento, é importante que você estude os conceitos de cliente-servidor e o funcionamento do protocolo “http”. Assim, você poderá compreender melhor o que abordaremos a partir de agora.

Imagine um usuário utilizando um navegador web, ou browser. Quando o navegador envia uma solicitação ao servidor para uma página qualquer na internet, esse servidor irá processar o script antes de veicular a página no navegador. Esse processamento, inclusive, pode incluir a consulta em um banco de dados. Assim, o script fica sendo processado e a saída é enviada ao navegador. A seguir, a Figura 2 ilustra o que foi abordado.

Figura 2 – Esquema de requisição a uma aplicação *back-end*



Fonte: https://developer.mozilla.org/pt-BR/docs/Learn/Server-side/First_steps/Client-Server_overview. Acesso em: 28 mar. 2022.

Podemos dizer que o maior benefício do código executado do lado do servidor é personalizar o conteúdo visualizado pelos usuários de forma individual. Em outras palavras, o que o usuário A visualiza ou tem acesso é diferente do que o usuário B visualiza ou tem acesso, lembrando que ambos estão acessando o mesmo endereço web.

Imagine uma plataforma de biblioteca digital à qual duas pessoas diferentes têm acesso. Agora imagine que essa plataforma permite que cada usuário possa definir suas preferências de leituras destacando as categorias dos títulos disponíveis em seu acervo. Enquanto um usuário gosta de tecnologia e automobilismo, o outro pode gostar de gastronomia e moda. Dessa forma, por mais que os dois estejam acessando a mesma aplicação web, por ser *back-end, server-side*, ela entrega experiências diferentes para cada usuário. Logo, é uma aplicação dinâmica *back-end* que roda do lado do servidor.

1.2 Estrutura do Node.js

Já abordamos um ponto da estrutura do Node.js mais cedo, quando citamos que sua execução é *single-thread*, isto é, apenas uma *thread* executa o código JavaScript da aplicação. Cabe ressaltar que todas as outras linguagens são *multi-thread*. Agora vamos ver o que realmente isso quer dizer e se traz vantagens à execução.

Em um servidor web utilizando, por exemplo, o PHP, para cada requisição recebida, é criada uma nova *thread* para tratar essa requisição. Logo, para cada requisição recebida, serão demandados recursos computacionais, como memória e CPU. Porém, se esses recursos forem limitados, as *threads* não serão muitas, já que elas também serão limitadas. Dessa forma, as novas requisições terão que esperar na fila de processos.

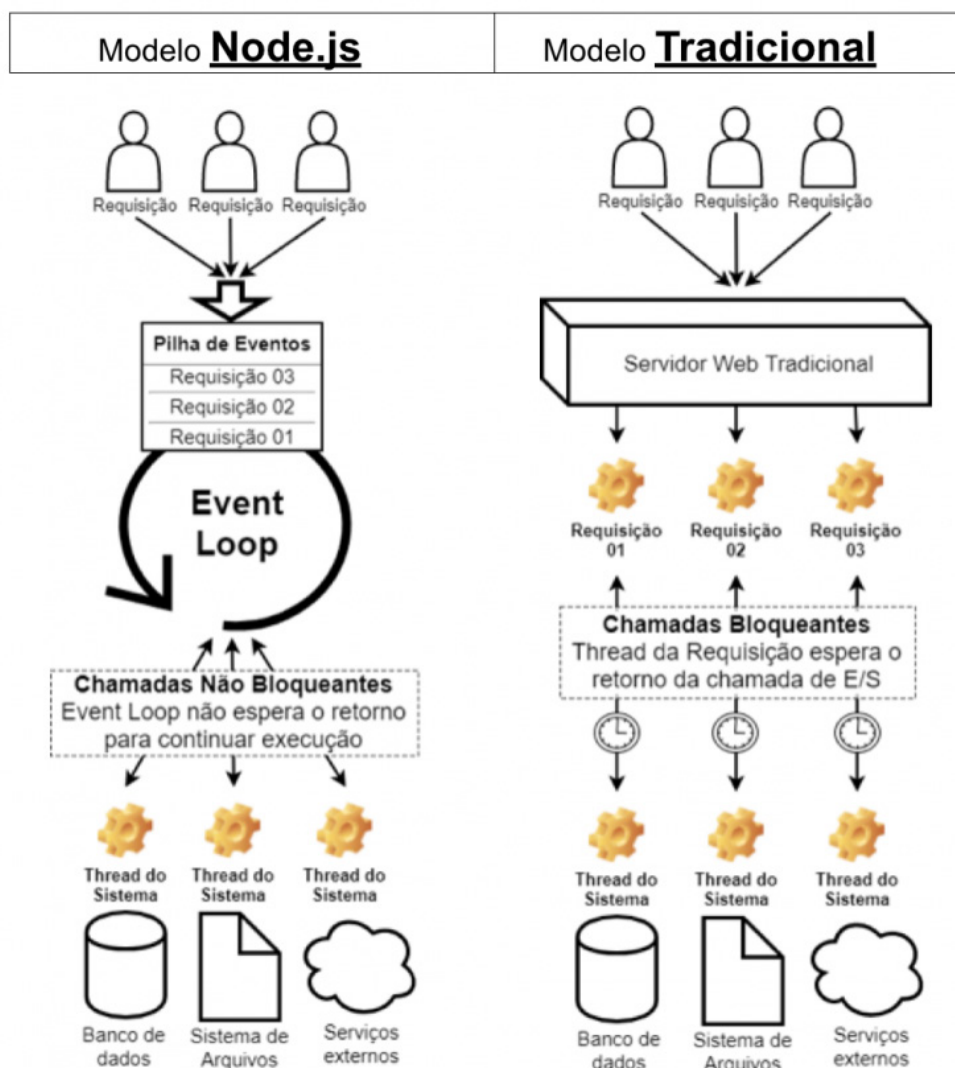
Como no Node uma *thread* é responsável por tratar essas requisições, então ela recebe o nome de *Event Loop*, já que trata as requisições como eventos. Assim, ela fica sempre em execução, esperando novos eventos para tratar, e, para cada requisição, um novo evento é criado.

O fato de o Node ser *single-thread* não o torna menos eficiente se comparado a outras tecnologias, já que sua arquitetura faz com que

um número maior de requisições concorrentes seja atendido. Em comparação às *multi-threads*, o Node se mostrou mais eficiente.

A Figura 3 mostra a diferença entre o modelo tradicional e o Node.js:

Figura 3 – Node.js versus servidor web tradicional



Fonte: <https://www.opus-software.com.br/node-js/>. Acesso em: 28 mar. 2022.

1.3 Instalando e configurando o Node.js

Cabe ressaltar que, para instalar e configurar o Node.js, os passos serão os mesmos, porque uma das características mais interessantes da

tecnologia é ser multiplataforma. Isso quer dizer que ele roda, inclusive, no sistema operacional que você está utilizando.

O primeiro passo a ser dado é acessar a página oficial do Node (<http://nodejs.org/>) e escolher a opção de download compatível com a sua máquina. A Figura 4 mostra a página oficial com as opções citadas. Uma observação importante é que, ao acessar a página citada para download do Node, ela já reconhece o seu sistema operacional.

Figura 4 – Página oficial do Node.js para download



Fonte: captura de tela de <https://nodejs.org/en/>. Acesso em: 28 mar. 2022.

Cabe ressaltar que o Node.js é disponibilizado em duas versões. A Current é a versão mais atualizada da plataforma. Já a LTS (*Long-term support*) é a de suporte a longo prazo e a versão mais estável do Node, recebendo, logo, suporte do time do Node até a nova versão ser lançada.

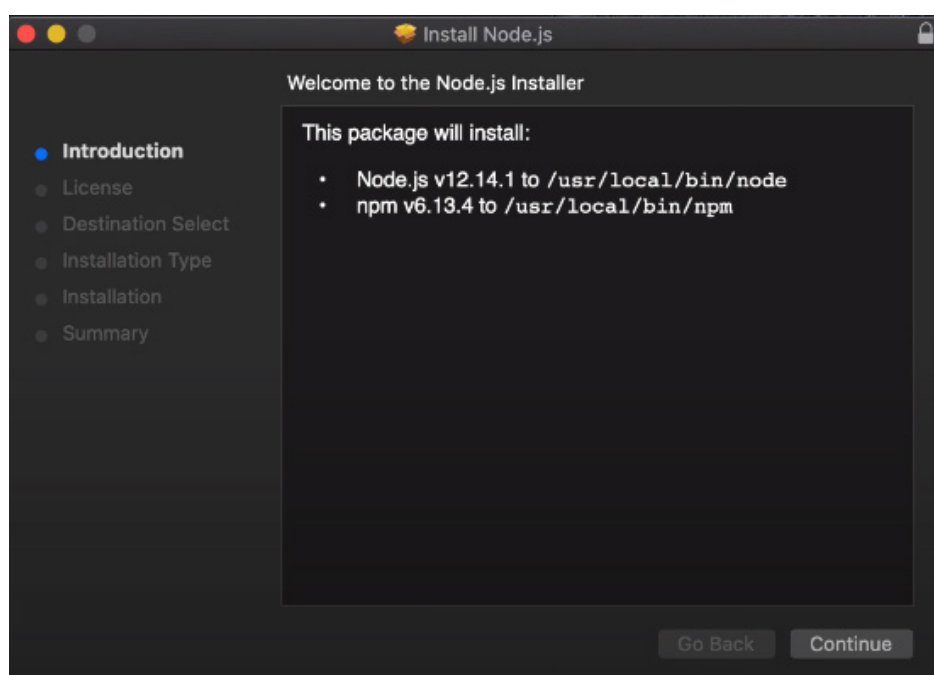
Outro fato importante de ser mencionado é que as versões do Node.js são retrocompatíveis, ou seja, você poderá instalar a última versão sem prejudicar o que implementou utilizando a versão anterior. Para manter o Node sempre atualizado é simples, basta instalá-lo novamente para realizar a atualização.

Falando em instalação, a seguir trazemos o processo de download e instalação do Node.js. Dependendo do sistema operacional utilizado, temos formas de instalação diferentes.

Para quem usa o SO Windows, basta realizar o download do executável pela página oficial do Node.js e executar em seu dispositivo. Dessa forma, o Node já estará no Path do Windows.

Já para a instalação no Mac, você poderá escolher entre dois caminhos. Um é pela página oficial do Node, que até certo ponto é parecido com o processo do Windows. Depois do download do executável, o arquivo será aberto e a instalação iniciará. A partir desse momento, basta ir clicando para continuar até finalizar a instalação. A Figura 5 ilustra esse processo.

Figura 5 – Interface Mac na instalação do Node.js

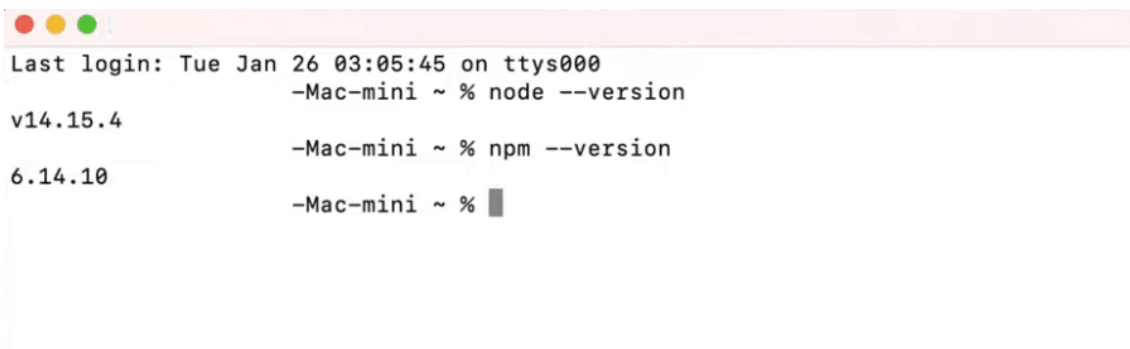


Fonte: captura de tela de <https://www.treinaweb.com.br/blog/instalacao-do-node-js-windows-mac-e-linux>. Acesso em: 28 mar. 2022.

Após a instalação do Node no Mac, você precisará testar para verificar se tudo ocorreu bem, sendo necessário, então, iniciar o Terminal. Para isso,

basta utilizar o comando *command + barra de espaço*, digitar “terminal” e pressionar “enter”. No “Terminal”, digite “node-v” e pressione “enter”. Se a versão do Node.js for exibida, é porque o processo deu certo. A Figura 6 ilustra esse resultado.

Figura 6 – Validando a instalação do Node.js no Mac



```

Last login: Tue Jan 26 03:05:45 on ttys000
-Mac-mini ~ % node --version
v14.15.4
-Mac-mini ~ % npm --version
6.14.10
-Mac-mini ~ % █
  
```

Fonte: <https://kinsta.com/pt/blog/como-instalar-o-node-js/>. Acesso em: 28 mar. 2022.

Para instalar o Node.js no Linux, basta utilizar o próprio gerenciador de pacotes do SO. Para isso, inicie o terminal pressionando CTRL + ALT + T. No Terminal, digite o comando de instalação do curl. Como mostrador a seguir:

```
sudo apt-get install curl
```

Em seguida, execute o script, mostrado a seguir, para adicionar o repositório do Node:

```
curl -fsSL https://deb.nodesource.com/setup_14.x | sudo -E bash -
```

Para finalizar, instale o Node utilizando o comando a seguir:

```
sudo apt-get install -y nodejs
```

Assim como no Mac, verifique se a instalação foi bem-sucedida utilizando o mesmo comando:

```
node -v
```

Se a versão for exibida, a instalação foi realizada com sucesso!

Finalizamos nosso primeiro Tema. Como boa prática, realize a instalação do Node.js na sua máquina. Porém, o mais importante é compreender que o Node.js é uma plataforma para programar em JavaScript e, por esse motivo, fizemos uma introdução sobre a linguagem. Assuntos inerentes à programação, como lógica de programação, algoritmos, HTML e CSS, são extremamente necessários também, mas outras áreas também têm sua relevância. Reunindo esses conhecimentos, você terá mais facilidades quando for tocar seus projetos. Como sugestão, não deixe de visitar a página oficial do Node.js e acompanhar as atualizações disponíveis para ter acesso à documentação completa da plataforma.

Bons estudos!



Referências

LECHETA, Ricardo. **Node.js Essencial**. São Paulo: Novatec, 2018.

NODEJS.ORG. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 28 mar. 2022.

PEREIRA, Caio. **Node.js**: Aplicações web real-time com Node.js. São Paulo: Casa do Código, 2016.

SHAH, Dhruti. **Node.js Guide Book**. Noida: BPB, 2019.

Desenvolvimento Web Back-end II

Autoria: Allan Crasso Naia de Souza

Leitura crítica: Tatiele Razera



Objetivos

- Conhecer os princípios de desenvolvimento de aplicações web.
- Utilizar linguagem de programação de script, de acordo com os padrões do ECMAScript.
- Conhecer a estrutura do Node.js e o processo de instalação e configuração da plataforma.



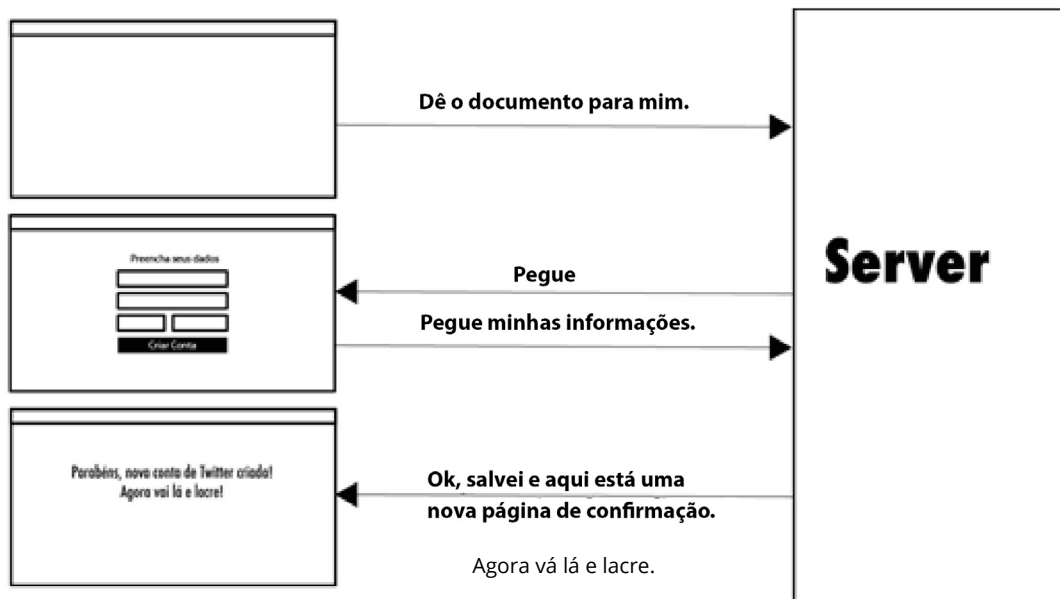
1. Node.js – Gerenciamento de Sessão e Controle de Cache

Para abordar o gerenciamento de sessão e controle de cache, primeiramente é importante abordar o protocolo HTTP. Esse protocolo clássico é uma ferramenta sem estado, ou seja, cada solicitação enviada por um cliente é interpretada pelo servidor da web de forma independente e não tem nada a ver com outra solicitação.

O rastreamento de sessão dá a possibilidade de sabermos o progresso de um usuário em vários *servlets* ou páginas HTML, que não têm estado. E o que é uma sessão? Resumidamente, pode-se dizer que uma sessão é definida por um conjunto de solicitações que um determinado navegador faz por um determinado período.

Para exemplificar melhor, essas solicitações podem ser páginas que são partes de um todo. Pense em um aplicativo de restaurante: você acessa páginas em que pode escolher um determinado prato, depois uma determinada bebida e, para finalizar, uma sobremesa. Então, todas essas páginas são solicitações. Na Figura 1, é mostrado como funcionam as sessões web.

Figura 1 – Requisição ao servidor web



Fonte: adaptada de <https://medium.com/>. Acesso em: 2 abr. 2022.

Em Node.js, geralmente, utiliza-se *middleware* para gerenciar sessões. E como isso funciona?

O Node.js armazena a sessão no próprio servidor expresso (Express). Aqui chamamos a atenção para o fato de o armazenamento de sessão do lado servidor, que se chama Memorystore, de forma proposital, não ter sido desenvolvido para o ambiente de produção. Isso porque fatalmente ele vazará a memória na maioria das condições, não ultrapassará um único processo e será destinado à depuração e ao desenvolvimento. Dessa forma, como gerenciar várias sessões?

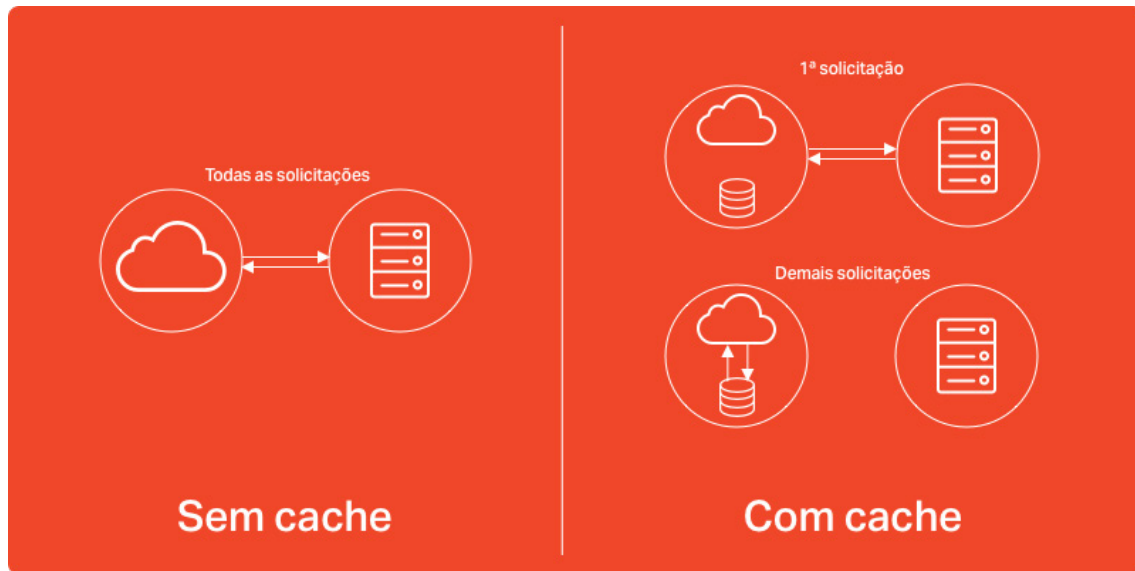
Para essa tarefa, é necessário criar um mapa global para que todas as sessões possam ser mantidas. Cabe ressaltar que variáveis globais em Node.js consomem memória e podem revelar falhas de segurança a nível de produção. Diante disso, pensou-se no armazenamento de sessão externo, em que todas as sessões são armazenadas na loja. Dessa forma, elas pertencerão a um único usuário.

Como somos desenvolvedores de aplicações Web, uma de nossas preocupações é otimizar o desempenho da aplicação, a fim de melhorar cada vez mais a experiência do usuário. Para que isso seja possível, é necessário identificar os gargalos que dificultam o desempenho das aplicações. Em se tratando de Node.js, por ser utilizado para desenvolvimento de aplicações *back-end*, é necessário armazenar dados ou informações dos usuários em um banco de dados para recuperação posterior.

Dessa forma, o cache pode ser utilizado como uma técnica para resolver os gargalos de desempenho em escala. O armazenamento em cache, de modo geral, é recomendado para casos em que o estado dos dados em um ponto específico da aplicação raramente irá ser alterado. A título de exemplo, caso seja necessário utilizar uma API, esse recurso pode ser de uma fonte externa que será chamada uma única vez e armazenada em cache.

Essa técnica é o que você fará na prática quando utilizar o Node.js e suas bibliotecas. Com isso, o desempenho da aplicação melhora de forma significativa, já que a resposta será rápida porque os dados estão armazenados no cache. A Figura 2 ilustra como é realizado esse controle de cache.

Figura 2 – Controle de cache



Fonte: <https://blog.impulso.network/armazenando-em-cache-javascripts-axios/>.
Acesso em: 2 abr. 2022.

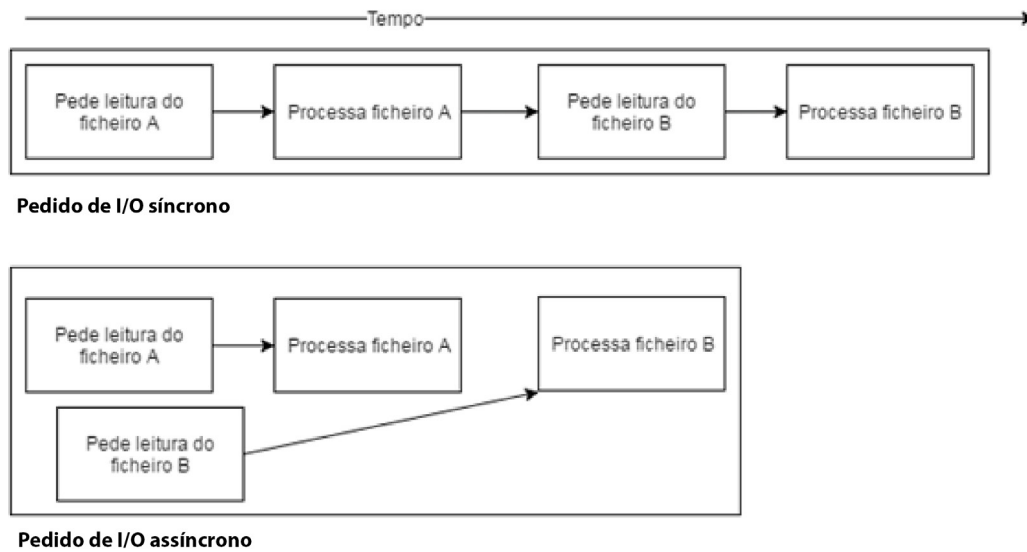
1.1 Blocking versus Non-blocking

Segundo o documento oficial do Node.js, a diferença entre chamadas com e sem bloqueio no Node.js está relacionada ao *loop* de eventos e *libuv*. Aqui nenhum conhecimento prévio desses tópicos é necessário, porque supõe-se que você já tenha o conhecimento sobre JavaScript e sobre o padrão de chamada Node.js.

Para que você comece o estudo do padrão de chamada, é necessário abordar execução síncrona e assíncrona. A execução síncrona é quando o código roda em sequência. Já na programação assíncrona, o programa não espera que a tarefa seja concluída e pode passar para a próxima tarefa. No geral, uma operação assíncrona está relacionada com E/S, porque as operações de E/S são muito lentas e, portanto, se fossem síncronas, iriam bloquear a execução adicional do código.

A Figura 3 ilustra o fluxo de execução síncrona e assíncrona mencionado.

Figura 3 – Pedido de E/S Assíncrono



Fonte: <https://pt.stackoverflow.com/questions/124283/o-que-%C3%A9-o-assincronismo>.
Acesso em: 2 abr. 2022.

O bloqueio irá acontecer sempre que a execução de JavaScript, ocorrendo pelo Node.js, tiver que esperar até que uma outra operação seja concluída. Isso acontece devido ao *loop* de eventos não poder continuar rodando o JavaScript durante a execução de uma AS, que é uma operação de bloqueio.

Um ponto relevante a ser observado é que todos os métodos de E/S na biblioteca do Node.js fornecem versões assíncronas, que por sua vez não são bloqueantes e aceitam funções de retorno de chamada. Outro ponto sobre *blocking* e *non-blocking* é que os métodos de bloqueio são executados de forma síncrona e os métodos sem bloqueio são executados de forma assíncrona.

1.2 Node.js – Entendendo o Núcleo

Esse tópico é importante para se ter uma ideia sobre a arquitetura do Node.js, passando-se a compreender como essa tecnologia funciona no que tange a suas relações com processos enviados à CPU. Um fato que

abordaremos é que o Node.js realiza solicitações simultâneas, seguindo o padrão *single-thread*. Sendo uma plataforma escalável, de baixo nível, você poderá programar diretamente com diversos protocolos, os mais utilizados em redes e internet.

1.2.1 Node.js *Single-thread*

As aplicações do Node.js serão *single-thread*, mas o que isso significa? Entenda que uma *thread* é uma porção do software que trabalha como um subsistema, como se fosse uma parte do todo, ou seja, do processo que se divide em uma ou mais tarefas. Assim, o Node.js utiliza o conceito de *thread* única para fazer a gestão de pilhas de eventos ou pilha de chamada (*Call Stack*), que por sua vez adota um comportamento do tipo LIFO (última entrada, primeira saída).

Porém, existe uma discussão entre o Node ser *single* ou *multi-thread*. Isso se deve ao fato de as operações em *background* do Node serem gerenciadas por *works*, que por sua vez podem conter operações *multi-threads*. Os *works* são processos que rodam em *background*, de E/S assíncronos, ou seja, não são bloqueantes e são gerenciados pela biblioteca *libuv*.

Observação importante: esse modelo de *thread* única para a manipulação de *Call Stack* é o que garante a performance dessa plataforma. Vamos falar mais sobre isso a seguir.

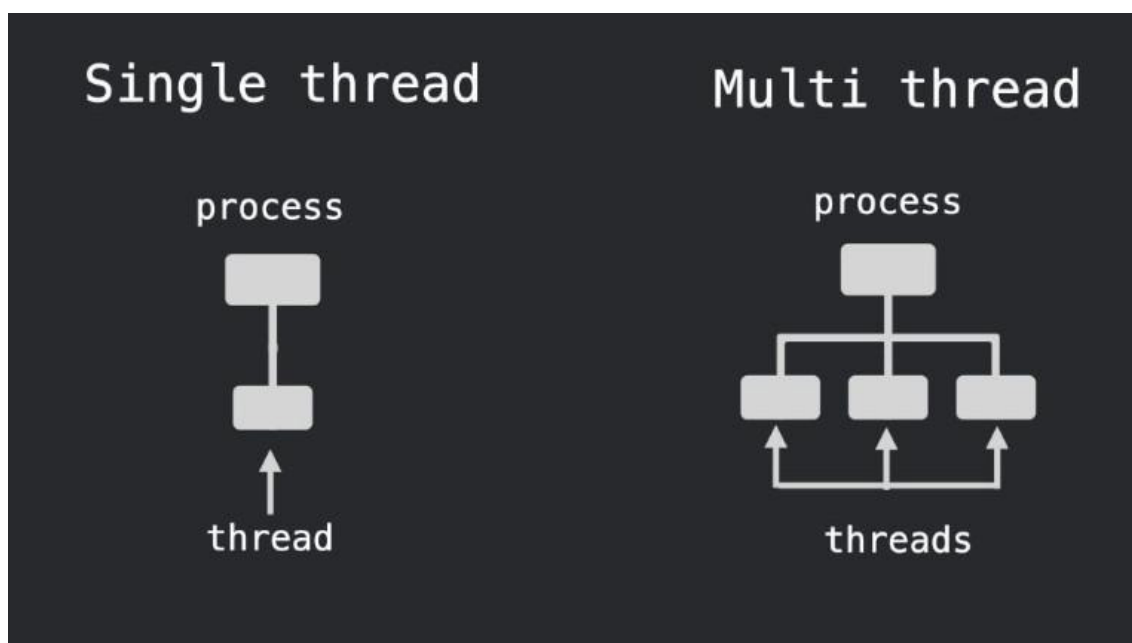
1.2.2 *Multi-thread*

Nesse ponto, você já está ciente de que o Node.js é *single-thread*. Essa característica, a princípio, parece não ser eficiente, o que leva a pensarmos que o desempenho será ruim. Porém, o que ocorre na verdade é o contrário, pois o fato de o Node.js ser *single-thread* o torna

mais eficiente e escalável em comparação a outras alternativas *multi-thread*, como a linguagem JAVA.

Falando em *multi-thread*, essa execução envolve o aproveitamento de vários núcleos de um sistema. Diante disso, se uma *thread* estiver na fila para execução, as outras *threads* ainda estão em execução. Na teoria, o modelo de *multi-thread* parece ser o ideal, porém o que não é levado em consideração é que nele uma *thread* poderá ser bloqueada mesmo tendo outras disponíveis.

Figura 4 – Comparação entre *single-thread* e *multi-thread*



Fonte: <https://blog.debugeverything.com/pt/introducao-node-js/>. Acesso em: 2 abr. 2022.

1.2.3 Call Stack

Call Stack, ou pilha de chamadas, é um mecanismo para rastrear as chamadas de função. Essa característica vem da própria linguagem JavaScript, que já usa a pilha de chamadas para gerir as execuções de funções.

Para entender melhor, quando um script é executado, o motor JavaScript cria um contexto de execução global e coloca o processo no topo da pilha de chamadas, princípio conhecido como LIFO. Assim que a execução termina, como seu endereço está armazenado em memória, ela é retirada da pilha e retoma a execução de onde havia parado.

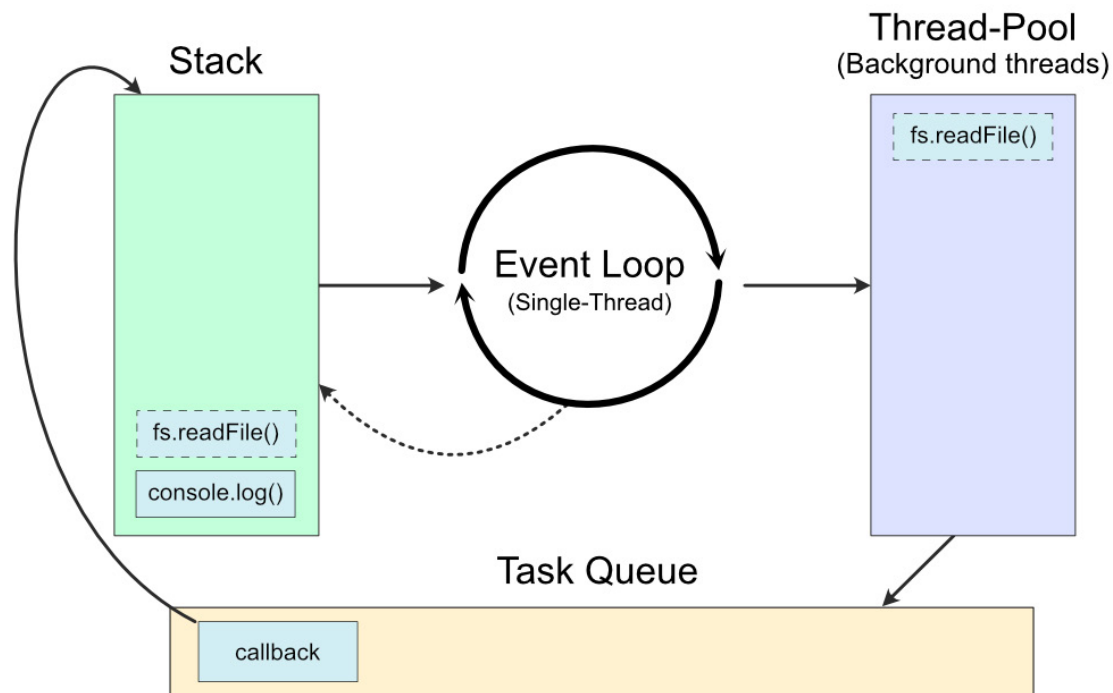
1.2.4 Event Loop

De uma forma mais simples, um *Event Loop* é responsável por monitorar a *Stack*, a pilha, com o intuito de buscar eventos a serem processados em espera. Quando um evento é encontrado, logo ele é executado e o *Event Loop* é liberado para buscar outros eventos prontos para serem executados.

Cabe ressaltar que isso não representa uma operação longa. Se fosse o caso, como acontece na leitura de arquivos em disco, o Node.js não teria a performance que tem.

A Figura 5 mostra a representação de funcionamento de um *Event Loop*.

Figura 5 – Execução do *Event Loop*



Fonte: <https://fabiojanio.medium.com/introdu%C3%A7%C3%A3o-ao-node-js-single-thread-event-loop-e-mercado-46edd82c1faf>. Acesso em: 2 abr. 2022.

A imagem ilustra dois eventos na *Stack*. Levando em consideração que a *Stack* é do tipo LIFO, o "`fs.readFile()`" será executado primeiro.

Resumindo, o *Event Loop* é um mecanismo responsável por executar e emitir eventos. Na prática, ele é basicamente um *loop* infinito que, a cada ciclo, verifica em sua fila de eventos se tem algum para ser disparado. Quando isso acontece, o *Event Loop* executa e envia para a fila de executados.

Para finalizar este Tema, até aqui foi abordada uma teoria necessária para quem deseja utilizar o Node.js como sua plataforma de desenvolvimento *back-end* com JavaScript. Saber os motivos que fazem do Node.js uma plataforma com performance ágil é importante quando você entrar no dilema de optar por outras plataformas.

Como já mencionado, existem alguns conhecimentos que são inerentes a outros. Os conceitos abordados são nativos de banco de dados. Por estarmos abordando uma plataforma que utiliza uma linguagem de programação, que por sua vez vai utilizar processos, é necessário entender como isso ocorre no core das tecnologias e o que faz essas tecnologias serem as mais utilizadas do mercado.

Como dica de estudo, sugerimos que você leia a documentação oficial do Node.js para aprofundar o estudo sobre o core Node. Busque entender a biblioteca *libuv*.

Bons estudos!

Referências

LECHETA, Ricardo. **Node.js Essencial**. São Paulo: Novatec, 2018.

NODEJS.ORG. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 28 mar. 2022.

PEREIRA, Caio. **Node.js**: Aplicações web real-time com Node.js. São Paulo: Casa do Código, 2016.

POWERS, Shelley. **Aprendendo NODE**: Usando JavaScript no Servidor. São Paulo: Novatec, 2017.

SHAH, Dhruti. **Node.js Guide Book**. Noida: BPB, 2019.

Desenvolvimento Web

Back-end III

Autoria: Allan Crasso Naia de Souza

Leitura crítica: Tatiele Razelo



Objetivos

- Conhecer os princípios de desenvolvimento de aplicações web.
- Utilizar linguagem de programação de script, de acordo com os padrões do ECMAScript.
- Conhecer a estrutura do Node.js e o processo de instalação e configuração da plataforma.



1. Node.js – Gerenciamento de Controle de Fluxo de uma Aplicação Node.js

Antes de iniciarmos o tema propriamente dito, vamos esclarecer o que é controle de fluxo em uma aplicação Node.js. Cabe ressaltar que controle de fluxo é a capacidade que uma linguagem tem de desviar o fluxo do programa de acordo com um determinado teste. A exemplo, temos as estruturas condicionais, as quais verificam um determinado bloco de comandos e depois escolhem um caminho a seguir.

O Node.js segue o modelo de programação assíncrono. E o que isso significa?

Esses dois modelos, programação síncrona e assíncrona, são modelos de desenvolvimento intimamente relacionados ao fluxo de execução. No modelo síncrono, uma operação precisa ser finalizada para que outra seja executada. Pode-se dizer que é um modelo linear, previsível, cujas execuções acontecem de maneira cíclica, uma após a outra. Esse mesmo modelo é seguido por outras linguagens de programação, como o PHP.

Em comparação com o modelo síncrono, tem-se o modelo assíncrono, em que uma operação não precisa esperar outra terminar. Em vez disso, elas alternam o controle da execução entre si. Dessa forma, se torna um modelo imprevisível, que não garante a ordem da execução, favorecendo a concorrência.

Em Node.js, é utilizada a função de *callback* como mecanismo para a escrita de código. Essa técnica foi por muito tempo a forma mais comum para escrita de códigos assíncronos. Cabe ressaltar que hoje em dia estes são considerados obsoletos, já que tornam os códigos confusos no momento da leitura.

Como já visto, o JavaScript, e por conseguinte o Node.js, tenta simplificar os problemas de códigos simultâneos fornecendo um *loop* de eventos de *threads*. A seguir trazemos as responsabilidades do *loop* de eventos:

- Executar o código.
- Coletar o processamento de eventos.
- Executar as subtarefas em espera.

Entender o fluxo de controle do Node.js é muito importante para criar aplicações com grandes escalas que precisam ser confiáveis, fornecendo valor comercial previsível.

1.1 Ciclo de vida: aplicações Node.js

Para entender o ciclo de eventos do Node.js, você deve recordar o *Event Loop*, que é um mecanismo que torna as tarefas mais rápidas, podendo ser multitarefa. Ele permite que o Node.js execute operações E/S sem bloqueio.

Quando um arquivo Node.js é executado, com uma extensão `app.js`, o script será analisado pelo software responsável por interpretá-lo para que a máquina seja capaz de processá-lo. Isso quer dizer que todas as variáveis e funções são registradas na memória. Depois de ser analisado, o script, ou se preferir programa, vai mudar de estado, ficando pronto para execução e, por conseguinte, esperando o *loop* de eventos para que ele possa executar.

Como exemplo, imagine um banco de dados de uma aplicação que executa alguns comandos de consulta a essa base. De forma quase imperceptível, até que seja retornado algum resultado, existe um tempo muito pequeno, mas existe. Depois que o resultado é mostrado, o usuário aplica outra consulta e assim por diante. De forma resumida,

o papel do *loop* de eventos é esperar as solicitações, que são *scripts* prontos para executar.

Basicamente, o ciclo de vida de uma aplicação Node.js funciona assim. Ainda temos outros pontos a considerar, como os temporizadores.

1.2 Node.js – Funções de retorno

As funções de retorno, ou *callbacks*, são métodos passados como argumentos dentro de outras funções com a finalidade de serem chamadas em outro momento, predefinido no código. Essa técnica é utilizada, na maioria das vezes, para rodar uma rotina logo após uma execução assíncrona. No entanto, não existem impedimentos para utilização na programação síncrona.

A seguir, as Figuras 1 e 2 ilustram os retornos em programação síncrona e assíncrona.

Figura 1 – Função de retorno em programação síncrona

```
function processData () {  
  var data = fetchData ();  
  data += 1;  
  return data;  
}
```

Fonte: <https://nodejs.org/en/knowledge/getting-started/control-flow/what-are-callbacks/>.
Acesso em: 13 abr. 2022.

Figura 2 – Função de retorno em programação assíncrona

```
function processData (callback) {  
  fetchData(function (err, data) {  
    if (err) {  
      console.log("An error has occurred. Abort everything!");  
      return callback(err);  
    }  
  })  
}
```

Fonte: <https://nodejs.org/en/knowledge/getting-started/control-flow/what-are-callbacks/>.
Acesso em: 13 abr. 2022.

1.2.1 Node.js – Funções de retorno anônimas

Foi dito no tópico geral sobre funções de retorno que elas são passadas como argumentos dentro de outras funções. Porém, temos as *callbacks* anônimas, que são funções que não precisam estar ligadas a outras funções. A seguir, trazemos um exemplo dessas funções. Na Figura 3, são ilustradas a sintaxe básica e a aplicação de uma função de retorno anônima.

Figura 3 – Exemplificando a utilização de uma função de retorno anônima

```
function nonAnonymousFunction ( a ) {  
  return function ( b ) {  
    console . log ( "Parâmetro não anônimo: "  
      + a + " Parâmetro anônimo: " + b ) ;  
  } ;  
}  
  
nonAnonymousFunction ( "Olá" ) ( "Mundo" ) ; // Parâmetro não anônimo: Olá Parâmetro anônimo: Mundo
```

Fonte: <https://simple-task.com/news/anonymous-functions-callback-chaining-essentials-js/>.
Acesso em: 13 abr. 2022.

1.2.2 Métodos que utilizam as funções de retorno (*callback*)

Retomando o que foi abordado sobre as funções de retorno estarem como argumentos em outros métodos, os principais métodos que utilizam as *call-backs* são:

- **map():** esse método chama a função de retorno em cada elemento do vetor e, então, traz um novo vetor com os resultados. A Figura 4 mostra como a função **map()** utiliza as funções de *callback*.

Figura 4 – Método map() utilizando função de *callback*

```
let array = [1,2,3,4,5];

let novoArray = array.map(function(x){ //Utiliza uma função anônima de callback para
  calcular o quadrado de cada elemento
    return x * x;
});

console.log(novoArray);
// exibe [1,4,9,16,25] no console
```

Fonte: <https://www.mundojs.com.br/2018/08/15/funcoes-de-retorno-callback/>.
Acesso em: 13 abr. 2022.

- **filter():** esse método remove elementos do vetor que não passam por um critério específico, o qual ainda será definido na função de retorno. Para elucidar, veja o exemplo na Figura 5.

Figura 5 – Método filter() é utilizado para remover elementos de um vetor que não sejam pares

```
let array = [1,2,3,4,5];

function isEven(x){ //Confere se o valor é par.
  return x % 2 == 0;
}

let novoArray = array.filter(isEven); //Utiliza a função como callback para saber se
o valor é par

console.log(novoArray);
// exibe [2,4] no console.
```

Fonte: <https://www.mundojs.com.br/2018/08/15/funcoes-de-retorno-callback/>. Acesso em:
13 abr. 2022.

Esse foi um resumo sobre funções de retorno. A seguir, entenderemos como funciona a programação assíncrona com Node.js.

1.3 Node.js – Programação assíncrona

Neste ponto do material, é perceptível a importância dos conceitos vistos até aqui. Isso porque esse paradigma foca no uso das chamadas assíncronas, ou seja, como visto anteriormente, não deixa a CPU ociosa quando realiza uma operação I/O.

O código do tipo síncrono realiza carregamento sequencial de arquivos, fazendo leitura, uma após a outra, na ordem de execução. Diferentemente do síncrono, o desenvolvimento assíncrono realiza um carregamento em paralelo, e, assim, o uso da CPU é otimizado enquanto estiver ocorrendo uma execução de E/S.

É de suma importância que o código Node.js desenvolvido invoque o mínimo possível de funções bloqueantes, ou seja, funções síncronas. Essa boa prática tem o objetivo de obter a melhor performance na execução do código, otimizando o uso da CPU.

1.4 Node.js – *Async* e *Await*

Antes de falarmos do operador ***Await***, será necessário entender o que é ***Promise***. Como o próprio nome sugere, *Promise* quer dizer promessa e representa um valor que pode estar ou não disponível. Dessa forma, permite que métodos assíncronos retornem valores como métodos síncronos, ou seja, em vez do valor final, o método retorna uma promessa ao valor em algum momento no futuro.

Para fazer um resumo sobre *Promises*, iniciamos falando sobre *Await*. Esse operador foi desenvolvido justamente para esperar uma *Promise*, ou seja, ele faz a execução de uma função *Async* pausar, esperando o

retorno de uma *Promise*, e, depois disso, resume a execução da função *Async* quando o valor da *Promise* for resolvido.

No resumo sobre *Promise*, foi dito que ela pode ou não retornar algo. Nesses casos, o *Await* terá diferentes abordagens. Caso a *Promise* não seja retornada como valor final, ele será convertido para uma *Promise* resolvida. Ao mesmo tempo, se a *Promise* for rejeitada, o *Await* invocará uma *Exception* com valor rejeitado.

A Figura 6 ilustra o processo de execução de uma *Promise* em programação assíncrona.

Figura 6 – Execução da *Promise* em script assíncrono

```
async function pegarDadosProcessados(url) {  
  let v;  
  try {  
    v = await baixarDados(url);  
  } catch(e) {  
    v = await baixarDadosReservas(url);  
  }  
  return processarDadosNoWorker(v);  
}
```

Fonte: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Statements/async_function. Acesso em: 13 abr. 2022.

Para finalizar, cabe ressaltar que somente é possível utilizar uma *Await* se definirmos a função como *Async*. Dessa forma, a *Await* poderá receber a *Promise*.

1.5 Node.js – Programação orientada a eventos

Resumindo tudo que foi abordado até aqui, pode-se dizer que a programação com o Node.js é orientada a eventos. Isso significa que, em

geral, o cliente não espera pela ação executada no servidor, ou seja, é uma interação ***non-blocking*** (**assíncrona**).

Logo, o paradigma orientado a eventos é um fluxo de programação dirigido por eventos específicos, com execução de rotinas que são responsáveis por disparar esses eventos.



Referências

LECHETA, Ricardo. **Node.js Essencial**. São Paulo: Novatec, 2018.

NODEJS.ORG. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 13 abr. 2022.

PEREIRA, Caio. **Node.js**: Aplicações web real-time com Node.js. São Paulo: Casa do Código, 2016.

POWERS, Shelley. **Aprendendo NODE**: Usando JavaScript no Servidor. São Paulo: Novatec, 2017.

SHAH, Dhruti. **Node.js Guide Book**. Noida: BPB, 2019.

Desenvolvimento Web

Back-end IV

Autoria: Allan Crasso Naia de Souza

Leitura crítica: Tatiele Razelo



Objetivos

- Conhecer os princípios de desenvolvimento de aplicações web.
- Utilizar linguagem de programação de script, de acordo com os padrões do ECMAScript.
- Conhecer a estrutura do Node.js e o processo de instalação e configuração do *framework*.



1. APIs Nativas do Node.js

Agora que já sabemos como o Node.js trabalha para comunicar e entregar o conteúdo processado, abordaremos a utilização das APIs pelo Node.js. Estudaremos sobre termos, conceitos, a proposta de utilização e como essas APIs realizam a comunicação entre *client-side* com o *back-end*.

Nos dias de hoje, é muito comum criar aplicações, sejam web ou mobile, que consomem dados de uma API (*Application Programming Interface*), os quais fornecem dados do *back-end* a essas aplicações *client-side*. Então, pode-se dizer que uma API são aplicações *back-end* focadas apenas no processamento e no envio de dados entre aplicações *front* e *back-end*. De forma centralizada, são desenvolvidas aplicações clientes com interfaces específicas, dependendo da aplicação a que estejam dando suporte, que são responsáveis por fornecer algum tipo de informação para uma aplicação desktop, mobile ou web.

Segundo a documentação do Node.js, existem várias APIs nativas em Node.js para facilitar a vida do programador. Essas APIs podem ser para gerenciamento de memória, retorno de chamada, manipulação de erros etc. Para uso seguro dessas APIs, existe um índice de estabilidade das APIs nativas, o qual está dividido em:

- Estabilidade 0: representada pela cor vermelho, que indica que a API está obsoleta.
- Estabilidade 1: representada pela cor laranja, que indica que a API é experimental.
- Estabilidade 2: representada pela cor verde, que indica que a API é estável.

- Estabilidade 3: representada pela cor azul, que indica que a API é legada.

Segundo a documentação do Node.js, as APIs são marcadas como legado sempre que algum bug é encontrado. Elas ainda podem ser utilizadas, principalmente nos pacotes NPM, pois os bugs serão corrigidos. No entanto, é feita uma chamada de atenção para o uso de recursos experimentais, pois os bugs podem ter comportamentos que surpreendem os usuários quando ocorrerem modificações nas APIs experimentais. Portanto, é interessante evitar o uso de dessas APIs.

1.1 Node.js – Módulos

Os módulos em Node.js são funcionalidades que podem ser simples ou complexas, organizadas em um ou vários arquivos JavaScript, os quais podem ser reutilizados em todo o aplicativo Node.js. Cada módulo tem seu próprio contexto e, por isso, não pode interferir em outros módulos ou prejudicar o escopo global. Como dica de organização para evitar quaisquer problemas, cada módulo pode ser colocado em um arquivo “.js” separado em pastas.

Segundo a documentação do Node.js, existe um padrão para implementação dos módulos, chamado CommonJS. Esse padrão funciona como um grupo de voluntários que define outros padrões JavaScript para servidores web. A seguir, trazemos os três tipos de módulos Node.js:

- Módulos Principais.
- Módulos Locais.
- Módulos de Terceiros.

Abordaremos cada um dos módulos.

Módulos Principais incluem funcionalidades mínimas do Node.js. Eles são compilados em sua distribuição binária e carregados automaticamente quando o processo Node.js é iniciado. Cabe ressaltar que é necessário primeiro importar o módulo principal para que depois ele possa ser usado em seu aplicativo. Segue uma lista de alguns dos módulos principais utilizados no Node.js:

- HTTP: inclui classes, métodos e eventos para criar o servidor HTTP Node.js.
- URL: inclui métodos para resolução e análise de URL.
- QueryString: inclui métodos para lidar com a *string* de consulta.
- Path: inclui métodos para lidar com caminhos de arquivos.
- fs: inclui classes, métodos e eventos para trabalhar com E/S de arquivo.
- util: inclui funções utilitárias úteis para programadores.

Sabendo sobre os módulos disponíveis, agora vem a pergunta: como utilizá-los?

Para usar esses módulos ou módulos NPM, como são conhecidos também, é necessário primeiro importá-los usando a função `require()`, conforme mostrado a seguir:

```
var module = require ('module_name');
```

De acordo com a sintaxe mostrada anteriormente, é necessário especificar o nome do módulo na função `require ()`. Essa função

retornará um objeto JavaScript, dependendo do que o módulo especificado retornar.

```
var http = require('http');var server = http.createServer(function(req, res) {  
  //write code here  
});  
server.listen(5000);
```

Por sua vez, os Módulos Locais são módulos criados localmente em seu aplicativo Node.js. Eles incluem diferentes funcionalidades do seu aplicativo em arquivos e pastas separados. Você também pode empacotá-los e distribuí-los via NPM para que a comunidade Node.js possa usá-los. Por exemplo, se você precisar se conectar ao MongoDB e buscar dados, poderá criar um módulo para ele, que pode ser reutilizado em seu aplicativo. Para demonstrar seu uso, vamos escrever um módulo de log simples que registra informações, avisos ou erros no console (que é uma função de retorno).

Em se tratando de módulo local, ele deve ser colocado em um arquivo JavaScript separado. Sendo assim, crie um arquivo “log.js” e escreva o código a seguir:

```
var log = {  
  info: function (info) {  
    console.log('Info: ' + info);  
  },  
  warning: function (warning) {  
    console.log('Warning: ' + warning);  
  },  
  error: function (error) {  
    console.log('Error: ' + error);  
  }  
};  
  
module.exports = log
```

Dessa forma, é desenvolvido um módulo *logging* com três funções:

- info().
- warning().
- error().

Ao final, o objeto criado é atribuído ao `module.exports()`. Por sua vez, esse módulo expõe um objeto de log como módulo.

O `module.exports` é um objeto especial que está incluído em cada arquivo JS no aplicativo Node.js por padrão. Use `module.exports` ou `export` para expor uma função, objeto ou variável como um módulo no Node.js.

Até aqui o módulo de log foi criado, vamos ver agora como utilizá-lo em nosso aplicativo.

Para usar módulos locais nos aplicativos, é necessário carregá-los usando a função `require()` da mesma forma que foi utilizada no módulo principal. Porém, é importante especificar o caminho do arquivo JavaScript do módulo. A seguir é demonstrado como utilizar o módulo de registro desenvolvido em “Log.js”:

```
var myLogModule = require('./Log.js');  
myLogModule.info('Node.js started');
```

No código mostrado, é usado o módulo de log. Ele primeiro carrega o módulo de registro usando a função `require()` e o caminho especificado onde o módulo de registro está salvo. O módulo de registro está dentro do arquivo “Log.js” em alguma pasta – no exemplo, é a pasta raiz.

A função `require()` retorna um objeto de log porque o módulo de log expõe um objeto em Log.js usando `module.exports`. Então agora você pode usar o módulo de log como um objeto e chamar qualquer uma de suas funções usando notação de ponto, como `myLogModule.info()` ou `myLogModule.warning()` ou `myLogModule.error()`

No próximo tópico, vamos abordar sobre a utilização do módulo **FFmpeg**.

1.2 Node.js – FFmpeg

Toda página que trabalha com *streaming* de vídeo precisa mostrar, de alguma forma, uma pequena visualização do vídeo sem reproduzi-lo. A plataforma YouTube, por exemplo, reproduz um trecho, de 3 a 4 segundos, de um determinado vídeo sempre que o usuário passa o mouse sobre sua miniatura. Outra forma de fazer isso é utilizar quadros de um vídeo e fazer uma apresentação, em carrossel por exemplo. Para

implementar isso em Node.js, é utilizada uma ferramenta chamada FFmpeg. Veremos como fazer isso.

O FFmpeg é a ferramenta líder para codificar e decodificar mídias, visto que suporta os formatos antigos e novos de mídias. Além disso, é multiplataforma, o que contribuiu para ser tão popular e eficaz.

Pelos motivos citados, o FFmpeg foi a escolha para a manipulação de vídeo realizada dentro do aplicativo desenvolvido em Node.js. Para usar essa biblioteca, é necessário que o FFmpeg já esteja instalado (incluindo todas as bibliotecas de codificação necessárias, como libmp3lame ou libx264). A seguir, trazemos o comando utilizado para instalar o módulo usando o NPM:

```
npm install ffmpeg
```

Para começar a usar o FFmpeg, é necessário inclui-lo em seu projeto e então poderá utilizar a sua função por meio de uma *callback* ou de uma biblioteca *Promise*. A seguir, mostramos como utilizar a ferramenta em sua aplicação:

```
var ffmpeg = require('ffmpeg');
```

Agora usando a função de *callback*:

```
try {  
    new ffmpeg('/path/to/your_movie.avi', function (err, video) {  
        if (!err) {  
            console.log('The video is ready to be processed');  
        } else {  
            console.log('Error: ' + err);  
        }  
    });  
} catch (e) {  
    console.log(e.code);  
    console.log(e.msg);  
}
```

Como mencionado, podemos realizar a chamada dessa ferramenta utilizando uma *Promise*. Cabe ressaltar que cada vez que é criada uma instância, essa biblioteca fornece um novo objeto para recuperar as informações do vídeo. A seguir, trazemos o código com a configuração do FFmpeg e todos os métodos para realizar as conversões necessárias.

```
try {  
    var process = new ffmpeg('/path/to/your_movie.avi');  
    process.then(function (video) {  
        // Video metadata  
        console.log(video.metadata);  
        // FFmpeg configuration  
        console.log(video.info_configuration);  
    }, function (err) {  
        console.log('Error: ' + err);  
    });  
} catch (e) {  
    console.log(e.code);  
    console.log(e.msg);  
}
```

1.3 Node.js – APIs de Terceiros

Os Módulos de Terceiros, ou APIs de terceiros, são módulos disponíveis on-line para serem carregados utilizando o NPM. Podem ser instalados no projeto, localmente ou de forma global. A seguir listamos alguns módulos mais acessados:

- `npm install express.`
- `npm install mongoose.`
- `npm install -g @angular /cli.`

1.4 Node.js – MongoDB

Antes de abordarmos a utilização dessa base de dados, é importante mencionar a que ela se propõe. Segundo a sua página oficial, o MongoDB é um banco de dados desenvolvido em C++ e *open source* que utiliza o paradigma de documentos – um paradigma diferente para quem está acostumado com os bancos relacionais.

Para acessar o MongoDB, é necessário instalar os drivers. Para isso, em Node.js, utiliza-se o NPM. Segue a linha de comando que realiza essa tarefa:

```
npm install mongodb —save
```

Isso incluirá a pasta `mongodb` dentro da pasta `node_modules`.

Agora, inicie o servidor MongoDB usando o seguinte comando (supondo que seu banco de dados MongoDB esteja na pasta `C:\MyNodeJSConsoleApp\MyMongoDB`):

```
mongod -dbpath C:\MyNodeJSConsoleApp\MyMongoDB
```

Feito isso, é necessário realizar a conexão da aplicação com a base de dados:

```
var MongoClient = require('mongodb').MongoClient;

// Connect to the db

MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {
    if(err) throw err;
    //Write database Insert/Update/Query code here.
});
```

No código, foi importado o módulo `mongodb` (drivers nativos) e obtida a referência do objeto `MongoClient`. Em seguida, foi utilizado o método `MongoClient.connect()` para obter a referência do banco de dados MongoDB especificado. A URL especificada “`mongodb://localhost:27017/MyDb`” aponta para seu banco de dados MongoDB local criado na pasta `MyMongoDB`. O método `connect()` retorna a referência do banco de dados se ele existir; do contrário, cria um novo banco de dados.

A partir de agora, as operações básicas esperadas com a integração de uma aplicação com um banco, como inserir, atualizar e consultar, já podem ser realizadas. Para isso, basta utilizar a função de retorno `connect()` usando o parâmetro `db`.

Com a conexão com o MongoDB, finalizamos este Tema sobre o uso de APIs em Node.js. Perceba que, para agilizar o processo de desenvolvimento de aplicações web, utilizar esse recurso se faz extremamente necessário, pois o uso de APIs traz integração com outras aplicações, o que faz de sua aplicação uma solução eficiente.



Referências

HOWS, David; MEMBREY, Peter; PLUGGE, Eelco. **Introdução ao Mongo DB**. São Paulo: Novatec, 2019.

LECHETA, Ricardo. **Node.JS Essencial**. São Paulo: Novatec, 2018.

NODEJS.ORG. Disponível em: <https://nodejs.org/en/about/>. Acesso em: 13 abr. 2022.

PEREIRA, Caio Ribeiro. **Construindo APIs REST com Node.JS**. São Paulo: Casa do Código, 2019.

PEREIRA, Caio Ribeiro. **Node.js: Aplicações web real-time com Node.js**. São Paulo: Casa do Código, 2016.

POWERS, Shelley. **Aprendendo NODE: Usando JavaScript no Servidor**. São Paulo: Novatec, 2017.

SHAH, Dhruti. **Node.JS Guide Book**. Noida: BPB, 2019.



BONS ESTUDOS!