O que é o Node.js?

O Node.js é uma plataforma de desenvolvimento de código aberto que permite a execução de JavaScript no lado do servidor. Essa tecnologia revolucionou a maneira como as aplicações web são construídas. Aqui estão algumas das principais características e vantagens do Node.js:

1. JavaScript Unificado: Com o Node.js, tanto o lado do servidor quanto o lado do cliente podem ser programados em JavaScript, proporcionando uma experiência de desenvolvimento mais consistente e eficiente.

2. Arquitetura Event-Driven e Não-Bloqueante: O Node.js é construído com uma arquitetura baseada em eventos que permite que os desenvolvedores criem servidores e aplicativos altamente escaláveis. É não bloqueante, o que significa que pode lidar com muitas conexões simultâneas sem esperar que uma operação seja concluída antes de prosseguir para a próxima.

3. Ecossistema de Pacotes NPM: O Node.js inclui o Node Package Manager (NPM), um gerenciador de pacotes que simplifica o processo de instalação, atualização e gerenciamento de bibliotecas e módulos de terceiros. A vasta comunidade de desenvolvedores contribui com uma enorme variedade de pacotes prontos para uso.

4. Alto Desempenho: O Node.js é construído sobre o motor V8 do Google Chrome, conhecido por sua velocidade e eficiência. Isso o torna ideal para

aplicações que precisam de alta performance, como aplicações em tempo real e APIs.

5. Grande Comunidade e Suporte: O Node.js tem uma comunidade ativa e crescente de desenvolvedores. Isso significa que há uma abundância de recursos, tutoriais e bibliotecas disponíveis para ajudar os desenvolvedores a enfrentar desafios comuns.

6. Facilita o Desenvolvimento em Tempo Real: Devido à sua natureza não bloqueante, o Node.js é amplamente utilizado em aplicações de tempo real, como chat ao vivo, jogos online e colaboração em tempo real.

7. Reutilização de Código: A capacidade de compartilhar código entre o lado do servidor e o lado do cliente facilita a reutilização de funções e objetos, economizando tempo e esforço de desenvolvimento.

8. Ampla Aceitação no Mercado: Muitas empresas e organizações líderes, como Netflix, Uber, LinkedIn e PayPal, adotaram o Node.js devido à sua eficiência e desempenho. Isso significa que o Node.js é uma escolha sólida para desenvolvedores que desejam adquirir habilidades valiosas no mercado.

Em resumo, o Node.js é uma plataforma de desenvolvimento que revolucionou a maneira como as aplicações web são construídas, oferecendo eficiência, escalabilidade e velocidade. Sua combinação única de JavaScript tanto no lado do servidor quanto no cliente, juntamente com sua arquitetura não bloqueante e uma comunidade ativa, o torna uma escolha poderosa para uma variedade de aplicações web modernas.

NPM e o arquivo package.json

A instalação de pacotes via Node Package Manager (NPM) é uma parte fundamental do desenvolvimento em Node.js. Ela permite que os desenvolvedores gerenciem e instalem bibliotecas e dependências de terceiros em seus projetos de forma eficaz. O arquivo *package.json* desempenha um papel central nesse processo, pois é onde todas as informações sobre as dependências do projeto são registradas.

Aqui está uma explicação básica de como a instalação de pacotes via NPM funciona e como o arquivo *package.json* gerencia esse processo:

1. Inicializando um Projeto com *npm init*: O primeiro passo ao iniciar um novo projeto Node.js é criar um arquivo package.json. Isso pode ser feito executando o comando *npm init* no terminal. Esse comando fará uma série de perguntas sobre o projeto, como nome, versão, descrição e ponto de entrada, e criará um arquivo *package.json* com base nas respostas.

2. Instalando Pacotes com *npm install*: Depois de ter o arquivo *package.json* configurado, você pode começar a instalar pacotes e dependências. Para fazer isso, use o comando *npm install nome-do-pacote*. Por exemplo, se você deseja instalar o pacote *express* para criar um servidor web, basta executar *npm install express*. O NPM irá baixar o pacote e suas dependências e instalá-los no diretório do seu projeto.

3. Registro no package.json: Quando você instala um pacote, o NPM automaticamente adiciona uma entrada ao seu arquivo *package.json*. Isso inclui o nome do pacote, a versão instalada e qualquer outra informação relevante.

4. Gerenciamento de Versões: O arquivo *package.json* também é usado para gerenciar versões de pacotes. Você pode especificar intervalos de versões, ou usar operadores de comparação para definir versões específicas.

5. Reproduzibilidade do Projeto: Ter um arquivo *package.json* com todas as dependências listadas torna o projeto altamente reproduzível. Outros desenvolvedores podem clonar seu projeto e, executando *npm install*, obterão automaticamente todas as dependências necessárias, garantindo que todos trabalhem com as mesmas versões de pacotes.

6. Compartilhamento de Projetos: O arquivo *package.json* é essencial para compartilhar seu projeto em repositórios de código, como o GitHub, ou implantá-lo em serviços de hospedagem. Ele fornece a estrutura necessária para que outras pessoas possam entender e reproduzir seu ambiente de desenvolvimento.

Em resumo, a instalação de pacotes via NPM é uma parte vital do desenvolvimento em Node.js. O arquivo *package.json* não apenas gerencia as dependências do projeto, mas também facilita a colaboração, a manutenção e a reprodução consistente do ambiente de desenvolvimento. É uma ferramenta valiosa para o controle e gerenciamento de bibliotecas e dependências em seus projetos Node.js.

Módulos no JavaScript: Simplificando e Organizando seu Código

Os módulos no JavaScript são uma técnica fundamental para organizar e estruturar o código de maneira mais eficiente e reutilizável. Eles permitem dividir um programa em partes menores e independentes, chamadas módulos, que encapsulam funcionalidades específicas. Aqui está uma explicação básica do que são módulos, para que eles servem e as principais vantagens de modularizar um código:

O que são Módulos no JavaScript?

Módulos no JavaScript são unidades independentes de código que contêm variáveis, funções e até mesmo classes relacionadas a uma tarefa ou funcionalidade específica. Eles são uma maneira de dividir um grande programa em partes menores e gerenciáveis, tornando o código mais organizado e legível.

Para que servem os Módulos?

1. **Organização:** Os módulos ajudam a organizar o código, dividindo-o em partes mais gerenciáveis. Cada módulo pode se concentrar em uma única funcionalidade ou tarefa, facilitando a manutenção do código.

2. **Reutilização:** Os módulos são reutilizáveis. Uma vez definidos, você pode importar e usar o mesmo módulo em diferentes partes do seu programa, economizando tempo e esforço.

3. Abstração: Os módulos permitem abstrair a complexidade. Você pode ocultar detalhes de implementação em um módulo e apenas expor a interface pública, tornando seu código mais claro e menos propenso a erros.

4. Colaboração: Em projetos de equipe, os módulos facilitam a colaboração. Cada membro da equipe pode trabalhar em módulos separados sem afetar o código uns dos outros.

Principais Vantagens de Modularizar um Código:

- Manutenção Simples: Com o código dividido em módulos, a manutenção se torna mais simples. Você pode atualizar ou corrigir um módulo sem afetar o restante do programa.

- Reutilização de Código: Módulos reutilizáveis significam menos trabalho repetitivo. Se você escreveu um módulo para uma funcionalidade, pode usá-lo em vários projetos.

- Legibilidade Aprimorada: A modularização melhora a legibilidade do código. Cada módulo se concentra em uma tarefa específica, tornando mais fácil entender o que o código faz.

- Escopo Isolado: Módulos têm escopo isolado. Isso significa que variáveis e funções em um módulo não entram em conflito com nomes semelhantes em outros módulos, evitando colisões de variáveis.

- Encapsulamento de Funcionalidade: Módulos permitem encapsular funcionalidades em unidades independentes, protegendo o código interno e expondo apenas o que é necessário para o uso externo.

- Carregamento Sob Demanda: Com a modularização, você pode carregar apenas os módulos necessários quando são necessários, o que melhora o desempenho.

Em resumo, módulos no JavaScript são uma técnica importante para dividir e organizar o código de maneira eficiente. Eles simplificam a manutenção, promovem a reutilização e melhoram a legibilidade do código, tornando a programação mais eficaz e colaborativa. Quando usado adequadamente, os módulos são uma ferramenta poderosa para desenvolvedores que desejam criar código mais limpo e mais sustentável.

Strings

Uma String em JavaScript é um tipo de dado que representa texto, como palavras, frases ou caracteres. Elas são cercadas por aspas simples (' '), aspas duplas (" "), ou acentos graves (` ` `). Strings são amplamente utilizadas para armazenar e manipular informações de texto em programas JavaScript.

Template Literals:

Os Template Literals, introduzidos no ECMAScript 6 (também conhecido como ES6) do JavaScript, são uma maneira moderna e mais flexível de criar strings. Eles são delimitados por acentos graves (` ` `) e permitem a incorporação de expressões JavaScript dentro de strings, usando a sintaxe `${expressão}`. Isso é útil quando você deseja criar strings que contenham variáveis ou expressões dinâmicas.

Concatenação de Strings:

A concatenação de strings é o processo de combinar duas ou mais strings em uma única string. No JavaScript, você pode realizar a concatenação de strings de várias maneiras:

Usando o operador de adição (+):

```
const nome = 'João';
const sobrenome = 'Silva';
const nomeCompleto = nome + ' ' + sobrenome;
console.log(nomeCompleto); // Saída: "João Silva"
```

Usando Template Literals:

```
const nome = 'João';
const sobrenome = 'Silva';
const nomeCompleto = `${nome} ${sobrenome}`;
console.log(nomeCompleto); // Saída: "João Silva"
```

A principal vantagem dos Template Literals é que eles oferecem uma sintaxe mais limpa e legível, além de permitir a incorporação direta de variáveis e expressões dentro da string, sem a necessidade de operadores de concatenação (+). Isso torna o código mais eficiente e fácil de manter, especialmente quando você precisa criar strings complexas ou que dependem de valores dinâmicos.

No entanto, é importante observar que os Template Literals não são suportados em versões mais antigas do JavaScript, então a concatenação de strings usando o operador de adição (+) ainda é amplamente usada em projetos que têm como alvo navegadores mais antigos.

Manipulação de Strings

A manipulação de *strings* em *JavaScript* é uma parte fundamental da programação, e a linguagem oferece uma variedade de métodos para ajudar a realizar várias operações com *strings*. Abaixo estão alguns dos métodos mais comuns usados para manipular *strings* em *JavaScript*:

1. `toUpperCase()` e `toLowerCase()`:
   - `toUpperCase()`: Este método converte todos os caracteres em uma *string* para letras maiúsculas.
   - `toLowerCase()`: Este método converte todos os caracteres em uma *string* para letras minúsculas.

2. `trim()`, `trimStart()`, e `trimEnd()`:
   - `trim()`: Remove espaços em branco no início e no final de uma *string*.
   - `trimStart()`: Remove espaços em branco no início de uma *string*.
   - `trimEnd()`: Remove espaços em branco no final de uma *string*.

3. `replace()`:
   - O método `replace()` permite substituir partes específicas de uma *string* por outra. Você pode usar uma *string* ou expressão regular para encontrar e substituir o texto desejado.

4. `endsWith()` e `startsWith()`:
   - `endsWith()`: Verifica se uma *string* termina com a *substring* especificada e retorna um valor booleano.
   - `startsWith()`: Verifica se uma *string* começa com a *substring* especificada e retorna um valor booleano.

5. `split()`:

- O método `split()` divide uma *string* em um *array* de *substrings*, com base em um separador especificado. Isso é útil para dividir uma *string* em partes menores.

6. `includes()`:

- `includes()`: Este método verifica se uma *string* contém a *substring* especificada e retorna um valor booleano.

Aqui estão alguns exemplos de uso desses métodos:

```
const texto = '   JavaScript é incrível!   ';

const emMaiusculas = texto.toUpperCase();
const emMinusculas = texto.toLowerCase();
const semEspacos = texto.trim();
const comSubstituicao = texto.replace('incrível', 'poderoso');
const terminaComExclamacao = texto.endsWith('!');
const começaComJavascript = texto.startsWith('JavaScript');
const partes = texto.split(' ');

console.log(emMaiusculas); // Saída: "   JAVASCRIPT É INCRÍVEL!   "
console.log(emMinusculas); // Saída: "   javascript é incrível!   "
console.log(semEspacos); // Saída: "JavaScript é incrível!"
console.log(comSubstituicao); // Saída: "   JavaScript é poderoso!   "
console.log(terminaComExclamacao); // Saída: true
console.log(começaComJavascript); // Saída: true
console.log(partes); // Saída: ["JavaScript", "é", "incrível!"]
```

Esses métodos são poderosos e versáteis, tornando a manipulação de *strings* em *JavaScript* mais eficiente e flexível. Eles são essenciais para tarefas comuns, como formatação, busca e extração de informações em *strings*.

Tratamento de exceções

O tratamento de exceções no *JavaScript* é uma técnica fundamental para lidar com erros e exceções que podem ocorrer durante a execução de um programa. Erros, como tentar acessar uma variável indefinida ou dividir por zero, podem causar a interrupção inesperada de um programa, mas o tratamento de exceções oferece uma maneira de gerenciar essas situações de forma controlada.

Blocos `try` e `catch`:

A base do tratamento de exceções em *JavaScript* são os blocos *try* e *catch*. Aqui está uma explicação simples de como eles funcionam:

- O bloco `*try*` é usado para envolver o código que pode gerar uma exceção. Dentro do bloco `*try*`, você coloca o código que deseja monitorar em busca de erros. Se ocorrer uma exceção dentro desse bloco, a execução do código será interrompida nesse ponto, e o controle será transferido para o bloco `*catch*`.

- O bloco `*catch*` é usado para definir o que fazer quando uma exceção é lançada dentro do bloco `*try*`. Ele contém um conjunto de instruções que tratam o erro ou exceção específica que ocorreu. É dentro do bloco `*catch*` que você pode criar mensagens de erro personalizadas, registrar informações de erro ou tomar medidas corretivas.

Aqui está um exemplo simples:

```
try {
  // Código que pode gerar uma exceção
  const resultado = 10 / 0;
} catch (erro) {
  // Código para tratar a exceção
  console.error('Ocorreu um erro:', erro.message);
}
```

Neste exemplo, o bloco `*try*` tenta realizar uma divisão por zero, o que causaria uma exceção. O controle é transferido para o bloco `*catch*`, que registra uma mensagem de erro informando que ocorreu uma divisão por zero.

Funções dos Blocos `try` e `catch`:

Os blocos `*try*` e `*catch*` desempenham as seguintes funções:

1. Prevenção de Interrupções Inesperadas: Eles evitam que erros inesperados causem a interrupção completa do programa, permitindo que o código continue a ser executado mesmo quando ocorrem exceções.

2. Identificação de Erros: Eles fornecem informações sobre a natureza do erro, incluindo mensagens de erro que podem ajudar os desenvolvedores a diagnosticar e corrigir problemas.

3. Tomada de Medidas Corretivas: Eles permitem que você tome medidas corretivas apropriadas, como fornecer feedback ao usuário, registrar erros para depuração ou implementar estratégias alternativas.

Em resumo, o tratamento de exceções no *JavaScript*, com a utilização dos blocos `*try*` e `*catch*`, é uma técnica essencial para tornar os programas mais robustos e confiáveis. Ela permite que você lide com erros de forma controlada e continue a execução do programa, garantindo uma melhor experiência do usuário e facilitando a depuração e o diagnóstico de problemas.

Arquivos JSON

JSON, ou *JavaScript Object Notation*, é um formato de dados amplamente utilizado na programação para armazenar e trocar informações estruturadas. É uma forma de representar dados que é legível tanto para humanos quanto para máquinas.

O que é um Arquivo JSON?

- Um arquivo JSON é um arquivo de texto que contém dados organizados em uma estrutura de pares de chave e valor. Os dados são armazenados em um formato que se assemelha a objetos e *arrays* em *JavaScript*. O JSON é independente da linguagem de programação, o que significa que pode ser utilizado em várias linguagens.

- Um arquivo JSON é composto por objetos e *arrays*, onde os objetos são delimitados por chaves `{ }` e contêm pares de chave-valor, e os *arrays* são delimitados por colchetes `[ ]` e contêm uma lista ordenada de valores. Os valores podem ser *strings*, números, booleanos, outros objetos JSON, *arrays* ou até mesmo `*null*`.

Para que Serve um Arquivo JSON?

- Comunicação de Dados: JSON é frequentemente usado para trocar dados entre um servidor e um cliente, tornando-o uma escolha comum em serviços web, APIs e aplicativos da *web*. A simplicidade e a legibilidade do JSON o tornam eficaz para transmitir informações estruturadas.

- Configurações e Metadados: JSON é usado para armazenar configurações e metadados em muitos aplicativos. Isso inclui configurações de aplicativos, preferências do usuário, informações de perfil e muito mais.

- Armazenamento de Dados: JSON também é usado para armazenar dados em arquivos ou bancos de dados. Sua estrutura aninhada é útil para representar dados complexos, como listas de tarefas, registros de eventos ou informações geoespaciais.

- Interoperabilidade: Como o JSON é suportado por várias linguagens de programação, ele é usado para facilitar a interoperabilidade entre diferentes sistemas. Isso significa que dados podem ser facilmente compartilhados e processados em diversos ambientes de desenvolvimento.

- Legibilidade: JSON é fácil de ler e escrever para humanos, tornando-o uma escolha popular para configurações e dados legíveis por humanos. Sua estrutura clara e simples facilita a depuração e a manutenção de arquivos de configuração e dados.

Basicamente, um arquivo JSON é uma forma eficaz de estruturar e armazenar dados em um formato legível por humanos e máquinas. Sua versatilidade o torna uma escolha comum para a comunicação de dados, configurações de aplicativos, armazenamento de informações e interoperabilidade entre sistemas de diferentes linguagens de programação.

Leitura e Escrita em Arquivos em *JavaScript*

A leitura e escrita em arquivos são operações essenciais em muitos aplicativos *JavaScript*, seja para manipular configurações, salvar dados do usuário ou processar informações de arquivos. Vamos explorar como realizar essas operações de forma simples e direta.

Leitura de Arquivos:

Para ler um arquivo em *JavaScript*, você pode usar o módulo `fs` (sistema de arquivos). Aqui está um exemplo básico de leitura de um arquivo:

```
const fs = require('fs'); // Importe o módulo 'fs'

// Leitura síncrona (bloqueante)
try {
  const dados = fs.readFileSync('arquivo.txt', 'utf-8');
  console.log('Conteúdo do arquivo:', dados);
} catch (erro) {
  console.error('Erro ao ler o arquivo:', erro.message);
}
```

O código acima usa `fs.readFileSync()` para ler o arquivo "arquivo.txt" de forma síncrona, o que significa que ele bloqueará a execução do programa até que a leitura seja concluída. Você pode usar `fs.readFile()` para leitura assíncrona, que é preferível para aplicativos em tempo real e não bloqueia o programa enquanto o arquivo é lido.

Escrita em Arquivos:

Para escrever em um arquivo, você também pode usar o módulo `fs`. Veja um exemplo simples:

```
const fs = require('fs'); // Importe o módulo 'fs'

// Escrita síncrona (bloqueante)
try {
  fs.writeFileSync('arquivo.txt', 'Este é o conteúdo a ser escrito.', 'utf-8');
  console.log('Arquivo escrito com sucesso!');
} catch (erro) {
  console.error('Erro ao escrever no arquivo:', erro.message);
}
```

Da mesma forma que na leitura, você pode usar `fs.writeFile()` para escrita assíncrona, que é preferível para aplicativos em tempo real, evitando bloqueios.

Ao realizar operações de leitura e escrita em arquivos, lembre-se de lidar com exceções e erros. A manipulação adequada de exceções é fundamental para garantir a estabilidade de seu aplicativo.

Esteja ciente de que a leitura e escrita em arquivos podem ser operações intensivas e, em ambientes de produção, geralmente é preferível usar operações assíncronas para evitar bloqueios.

Certifique-se de ter as permissões necessárias para acessar os arquivos no sistema de arquivos em que seu aplicativo está sendo executado.

Leitura e escrita em arquivos são tarefas comuns em muitos aplicativos JavaScript, e o módulo `fs` oferece as ferramentas necessárias para realizar essas operações com eficiência. A prática e o entendimento das operações de arquivo são importantes para lidar com dados persistentes em seus aplicativos.

Programação assíncrona - *Callbacks*

O que é programação assíncrona?

Os códigos e funções assíncronas no JavaScript, quando são executados, não bloqueiam a execução das linhas de código seguintes. Uma função assíncrona é executada em segundo plano enquanto o programa continua interpretando as linhas de código seguintes.

Um exemplo de uma função assíncrona no JavaScript é a função `setTimeout()`. Com essa função nós podemos programar para que um determinado código seja executado depois que o tempo em milisegundos, passados como parâmetro, se esgotar. Por exemplo:

```
setTimeout(() => {
  console.log('Será executado após 2 segundos');
}, 2000);
```

Note que o primeiro parâmetro da função `setTimeout` é uma outra função que possui um `console.log` internamente. O segundo parâmetro é o tempo em milissegundos que deverá ser aguardado até que a função do primeiro parâmetro seja executada.

Funções assíncronas são amplamente utilizadas em sistemas web. Isso porque muitos dos dados que são necessários nesses sistemas geralmente estão em um servidor, e o acesso a esse servidor se dá pela internet. Como o tempo referente ao acesso ao servidor, ao processamento e ao retorno de dados pode demorar muito, usa-se a programação assíncrona para que o sistema não fique bloqueado enquanto esse processo está sendo feito.

Obviamente nós não vamos implementar um servidor para fazermos todos os processos. Para a nossa aula, nós vamos utilizar a função `setTimeout` para simular um tempo de resposta de um processo envolvendo um servidor na internet.

Callbacks

Uma função *callback* no JavaScript é uma função que é passada como parâmetro para outra função, que pode chamar a função *callback* para completar algum tipo de rotina ou ação.

Exemplo:

```
function obterNome(callback) {
  var nome = prompt('Entre com o seu nome: ');
  callback(nome);
}

obterNome((nome) => {
  console.log(nome);
});
```

No exemplo acima, a função `obterNome` recebe uma outra função *callback* como parâmetro. Essa função *callback* é chamada após o usuário entrar com o nome dele, passando a variável `nome` como parâmetro.

Promises

As *Promises* no JavaScript estão presentes em praticamente todo o ecossistema da linguagem. Elas são um padrão de desenvolvimento que visam representar a conclusão de operações assíncronas.

Uma Promise possui diferentes estados, sendo alguns deles:

- Pendente (*Pending*);
- Resolvida (*Resolved*);
- Rejeitada (*Rejected*);
- Realizada (*Fulfilled*);
- Estabelecida (*Settled*).

Geralmente os estados mais utilizados são: Resolvida e Rejeitada.

Ao criar uma *Promise*, ela se inicia como pendente (*pending*), assim, os estados em que ela pode se apresentar são aqueles informados anteriormente. Se ela estiver no estado de resolvida (*resolved*) é porque tudo deu certo, ou seja, a *Promise* foi criada e processada com sucesso, porém, em casos de falhas, ela estará no estado de rejeitada (*rejected*).

Uma das maneiras de fazer esse tratamento é por meio das funções *then* e *catch*, para sucesso ou falha, respectivamente.

Para resolver uma *Promise*, podemos utilizar a função `resolve` , passando como parâmetro um valor que será acessível através de nossa *Promise* resolvida:

```
function cadastrarUsuario() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Usuário cadastrado");
        }, 5000);
    });
}
```

Para pegar o valor passado para a função *resolve*, utilizanos o operador `then()`. Exemplo:

```
cadastrarUsuario().then((mensagem) => {
  console.log(mensagem);
  //será escrito na tela a mensagem Usuário Cadastrado
});
```

Para rejeitar uma *Promise*, usamos a função `reject` (da mesma forma que a *resolve*). Exemplo:

```
function cadastrarUsuario() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject("Erro ao cadastrar usuário");
        }, 5000);
    });
}
```

Assim como temos o `then()` para sucesso, também temos o `catch()` para erros:

```
cadastrarUsuario()
  .then((mensagem) => {
    console.log(mensagem);
  })
  .catch((mensagem) => {
    console.log(mensagem); //será escrito na tela a mensagem "Erro ao cadast
  });
```

Tanto o `resolve` com o `then` ou `reject` com `catch` funcionam em conjunto da mesma maneira, porém cada um tendo seu propósito: o primeiro para sucesso e o segundo para erro.

Referência

CASTIGLIONI, M. **Trabalhando Com Promises Em Javascript** . Disponível em: <https://blog.matheuscastiglioni.com.br/trabalhando-com-promises-em-javascript/>. Acesso em: 10 mar. 2023.

Async/await

As palavras-chave *async* e *await* são uma sintaxe que simplifica a programação assíncrona, facilitando o fluxo de escrita e leitura do código. Assim é possível escrever código que funciona de forma assíncrona, porém lido e estruturadode forma síncrona.

Definindo uma função como `async`, podemos utilizar a palavra-chave `await` antes de qualquer expressão que retorne uma *Promise*. Dessa forma, a execução da função externa (a função async) será pausada até que a *Promise* seja resolvida.

Uma função declarada como *async* significa que o valor de retorno da função será, "por baixo dos panos", uma *Promise*. Se a *Promise* se resolver normalmente, o objeto-Promise retornará o valor. Caso lance uma exceção, podemos usar o *try/catch* como estamos acostumados em programas síncronos.

O *async/await* surgiu como uma opção mais "legível" ao `then()`. É possível usar os dois métodos em um mesmo código e, a partir da versão 10 do Node.js, ambas as formas são equivalentes em termos de performance. O *async/await* simplifica a escrita e a interpretação do código, mas não é tão flexível e só funciona com uma *Promise* por vez.

Referência

AMOASEI, Juliana. **Async/await no JavaScript:** o que é e quando usar a programação assíncrona? Disponível em: <https://www.alura.com.br/artigos/async-await-no-javascript-o-que-e-e-quando-usar>. Acesso em: 10 mar. 2023. Com adaptações.

Requisições HTTP usando o axios

Requisições HTTP são um dos principais recursos usados no desenvolvimento web, já que são necessárias para acessar o *backend* de nossas aplicações, banco de dados, entre outros. Por isso, foi criado um dos pacotes mais famosos (se nãoo mais) de requisições JavaScript, o axios.

Características:

- Faz requisições HTTP;
- Transforma respostas em JSON automaticamente
- Criação de instâncias
- Suporte a requisições assíncronas

Requisições

Get

Uma requisição do tipo *get* pedirá todos os dados para o *backend*. Pelo fato de a requisição demorar algum tempo para ser respondida, será usada uma `promise then` para isso. E, dentro do `then`, trabalhamos com o resultado que retornou do servidor.

Exemplo:

```
axios.get('http://meuservidor.com/dados')
    .then((dados) => {
      //o parâmetro 'dados' é a variável que contém o retorno dos dados do s
    })
```

Post

Diferente de requisições *get*, o *post* já necessita de um corpo. Requisições do tipo *post* são usadas para gravar alguma informação dentro do servidor, como dados de usuário, imagens, vídeos, etc.

Em uma função do tipo post é preciso passar um objeto com os dados que queremos gravar dentro do corpo da requisição. Exemplo:

```
axios.post("http://meuservidor.com/dados", {
    id: 1,
    nome: 'Filipe'
  })
  .then(() => {
    console.log('Dados gravados com sucesso');
  });
```

Put

Requisições do tipo *put* são usadas para atualizar algum tipo de informação dentro do servidor, como usuário, produto, postagem etc. Assim como o *post*, este recebe um corpo dentro da requisição e o valor mandado sobreporá o valor atual do servidor.

```
//aqui passamos o id 1 como parâmetro para que o registro com esse id tenha
axios.put('http://meuservidor.com/dados/1', {
    nome: 'Bernardo'
  })
  .then(() => {
    console.log('Nome alterado com sucesso');
  })
```

Delete

A requisição *delete*, assim como o próprio nome diz, serve para deletar um dado do servidor. Assim como o *get*, ele não tem corpo, e também, não retorna nada, apenas um estado de sucesso. A função precisará apenas do *id* do dado que será deletado.

```
axios.delete('https:///meuservidor.com/dados/1')
.then(() => {
  console.log('Dados deletados com sucesso');
})
```

Concluindo

Esses são os principais recursos do axios, sendo muito úteis para diversos projetos que precisam se conectar a um *backend*. Por fim, caso queira aprender mais sobre os outros recursos do axios, recomendo o repositório oficial (https://github.com/axios/axios) do axios para isso.

Referências

RIBEIRO, L. **Requisições usando o axios.** Disponível em: <https://lucassr.medium.com/requisi%C3%A7%C3%B5es-usando-o-axios-c3eb75855a1e >. Acesso em: 15 fev. 2023. Com adaptações.