ECMAScript

Introdução a Módulos

Já vimos como incluir código JavaScript no HTML:
- Utilizando a tag <script>
- Todo conteudo do arquivo é incluído diretamente na página simultaneamente.

A medida que os sites foram evoluindo e as aplicações começaram a surgir, o JavaScript foi sendo cada vez mais utilizado.

Além disso o uso do JavaScript fora dos navegadores também passou a ser não apenas possível, como muito popular.
- Node.js é uma tecnologia que permite interpretar código JavaScript fora do navegador, foi criado em 2019.

Isso tudo contribuiu para o amadurecimento do JavaScript e a implementação de uma funcionalidade já existente em outras linguagens: módulos.

O que são Módulos?

Pensa em capítulos de um livro ou em seções de um supermercado. Os autores organizam suas histórias em capítulos, supermercados, seus produtos em seções, e bons programadores organizam seus códigos em módulos.

Ou seja, podemos entender que módulos são agrupamentos de códigos com funcionalidades distintas que podem ser compartilhados, adicionados ou até mesmo removidos de nossos softwares.

Por que utilizar Módulos?

Manutenibilidade: ou seja, um código com uma boa manutenibilidade é um código que é bom de dar manutenção, você não vai ter dificuldade em realizar manutenção;

Namespacing: é um conceito da programação que seria como um agrupamento de nomes. É como se fosse um agrupamento dos módulos. Os módulos nos permitem que tenhamos grupos de nomes isolados, ou seja, para que não haja divergência de nomes, é feito esse namespacing.

Reusabilidade: ou seja, com os módulos nós conseguimos criar pequenos agrupamentos específicos de códigos, e ai podemos reaproveitar esses códigos.

Módulos no JavaScript:

- CommonJS
- ES Modules

CommonJS

Há duas formulas de trabalharmos com módulos em JS, e a primeira forma que será apresentado é com o CommonJs, que é a forma mais tradicional de fazermos isso.

E para trabalharmos com o CommonJs nós usaremos uma nova ferramenta que é o NodeJS. O NodeJS é uma ferramenta que permite executarmos código JS fora do navegador, ou seja, ele trabalho do lado do back-end.

Abaixo nós podemos perceber que há duas funções, uma chamada render, que tem como objetivo aparecer no console uma mensagem informando que a interface esta sendo renderizada. E a outra chamada store, que tem como objetivo informar ao usuário que as informações estão sendo salvas no banco

```
function render() {
  console.log('Renderizando a interface da aplicação...')
}

function store() {
  console.log("Salvando as informações no banco de dados...")
}

console.log("Aplicação iniciada.")
render()
store()
console.log("Aplicação finalizada.")
```

Como é mostrado no console...

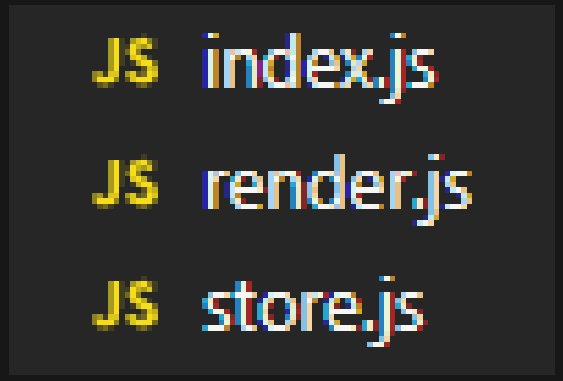
```
Aplicação iniciada.
Renderizando a interface da aplicação...
Salvando as informações no banco de dados...
Aplicação finalizada.
```

Porem fica a seguinte pergunta. Faz sentido essas duas funções estarem no mesmo arquivo?

A resposta, obvia, é que não faz sentido, pois são funções completamente divergentes!!

Ou seja, para que haja uma organização e um sentido, nós devemos separar os arquivos. Um arquivo para o render e outro para o store.

Separação dos arquivos



O que há no arquivo render:

```
function render() {
    console.log('Renderizando a interface da aplicação...')
}
```

O que há no arquivo store:

```
function store() {
    console.log("Salvando as informações no banco de dados...")
}
```

O que há no arquivo index:

```
console.log("Aplicação iniciada.")
render()
store()
console.log("Aplicação finalizada.")
```

Mas ainda há mais um detalhe muito importante!!

Essas funções não vão funcionar no nosso arquivo index.js, pois elas não foram chamadas de maneira correta. Para que elas funcionem elas precisam ser exportadas para o próprio arquivo delas, e após feito isso nós conseguimos chama-las no arquivo index.js. Veja como fazer essa ação abaixo:

```
function store() {
    console.log("Salvando as informações no banco de dados...")
}

module.exports = store
```

Arquivo store

```
function render() {
    console.log('Renderizando a interface da aplicação...')
}

module.exports = render
```

Arquivo render

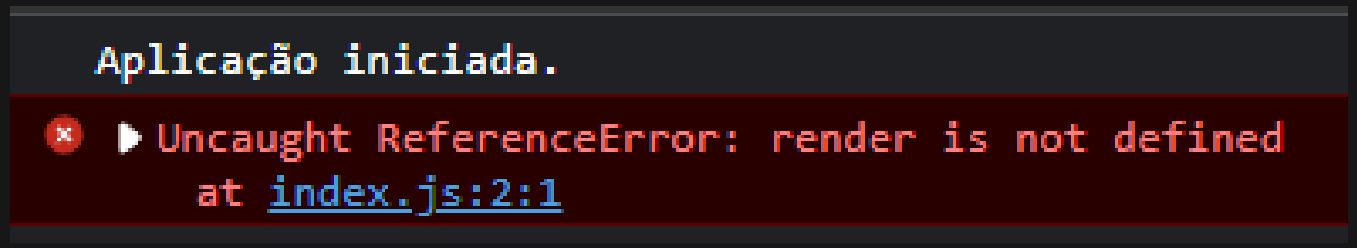
```
const render = require('./render')
const store = require('./store')

console.log("Aplicação iniciada.")
render()
store()
console.log("Aplicação finalizada.")
```

Arquivo index

Mas um detalhe...

A partir do momento que nós separamos os arquivos, vai dar erro no console.



E mesmo nós tendo feito as importações citadas anteriormente, ainda sim vai dar erro no console, pois os métodos de importação não é suportado pelo navegador, ou seja, não irá aparecer no console.

OBS:

O CommonJS é um módulo mais utilizado no NodeJS,.

ES Modules:

ES Modules é o sistema de módulo “mais novo” e deve ser um substituto para o sistema de módulo Node.js atual, que é CommonJS (CJS para abreviar), embora o CommonJS provavelmente ainda estará conosco por muito, muito tempo.

Exemplos:

```
//Funções para construir elementos:

//Função para criar um elemento label
function label(attributes) {
    const element = document.createElement('label') //Criação do elemento
    Object.assign(element, attributes) //Esta atribuindo ao elemento os atributos que passarmos
    return element //Retorna o elemento
}

//Função para criar um elemento input
function input(attributes) {
    const element = document.createElement('input')
    Object.assign(element, attributes)
    return element
}

//Função para criar um elemento br
function br() {
    const element = document.createElement('br')
    return element
}

//No console log esta sendo chamada as funções e esta sendo adicionado os atributos
console.log(label({ for: 'fullname', textContent: 'Nome Completo' }))
console.log(input({ id: 'fullname', name: 'fullname', placeholder: 'Digite seu nome completo...' }))
console.log(br())
```

index.js

O método object.assign() irá copiar todas as propriedades definidas em um objeto para outro objeto, ou seja, ele copia todas as propriedades de uma ou mais fontes para os objetos de destino. Fazendo isso, estamos acrescentando um elemento ao objeto.

Agora vamos para a parte que de fato interessa, que é realizar a separação do código.

Imagine que essa aplicação fosse muito maior, com muito mais linhas de código, como que faríamos para modularizar o código, ou seja, separar cada parte dele em diferentes módulos?

Nós já vimos como podemos fazer isso com o CommonJS, porem o CommonJS não roda do lado do navegador, então vamos ver agora como podemos fazer isso com os ES Modules.

Primeiro devemos criar um arquivo novo, o nome que foi dado para ele é function.js.

Modularizando as funções, ou seja, separando as funções:

```
console.log(label({ for: 'fullname', textContent: 'Nome Completo' }))
console.log(input({ id: 'fullname', name: 'fullname', placeholder: 'Digite seu nome completo...' }))
console.log(br())
```

index.js

```
function label(attributes) {
    const element = document.createElement('label')
    Object.assign(element, attributes)
    return element
}

function input(attributes) {
    const element = document.createElement('input')
    Object.assign(element, attributes)
    return element
}

function br() {
    const element = document.createElement('br')
    return element
}
```

function.js

Mas como já vimos antes, só fazer isso não faz com que o nosso código funcione, a não ser que chamemos o arquivo no HTML, mas ao invés de fazermos isso vamos utilizar o ES Modules

A forma de fazermos isso com o ES Modules é utilizando a palavra reservada EXPORT:

```
export function label(attributes) {
    const element = document.createElement('label')
    Object.assign(element, attributes)
    return element
}

export function input(attributes) {
    const element = document.createElement('input')
    Object.assign(element, attributes)
    return element
}

export function br() {
    const element = document.createElement('br')
    return element
}
```

O export é uma palavra reservada do JS que tem como objetivo realizar a exportação de uma função, ou até mesmo uma variável.

Essas funções que foram exportadas podem ser utilizadas por outros arquivos JavaScript. Segue o exemplo abaixo de como isso deve ser feito:

```
import { label, input, br } from './function.js'

console.log(label({ for: 'fullname', textContent: 'Nome Completo' }))
console.log(input({ id: 'fullname', name: 'fullname', placeholder: 'Digite seu nome completo...' }))
console.log(br())
```