**Um guia para importação e exportação de módulos com JavaScript**





**Juliana Amoasei**

10/08/2022

**COMPARTILHE**

**Introdução**

Neste artigo vamos revisar um conceito importante no dia a dia de [desenvolvimento com JavaScript](https://www.alura.com.br/artigos/javascript): o que é a **importação e exportação de módulos** e como funcionam as duas formas de fazer isso: usando **CJS** (*Common JavaScript*) ou **ESM** (EcmaScript Modules).

Mas por que escolhemos falar sobre esse assunto? Os conceitos de módulo e modularidade são parte importante do desenvolvimento com JavaScript, e quando temos mais de uma forma de fazer algo em programação - como é o caso aqui - é ainda mais importante que a gente entenda melhor os detalhes.

Para facilitar sua leitura, fiz esse sumário dos conteúdos que serão explorados ao longo deste artigo:

* O que são os módulos do JavaScript
* Por que utilizamos módulos?
* Utilização de módulos no JavaScript
  + CJS
  + ESM
  + A extensão .mjs
  + Pontos importantes sobre o uso do ESM
* Qual devo usar, afinal?
* Conclusão

O foco aqui são as importações e exportações de módulos com [Node.JS](https://www.alura.com.br/artigos/node-js), mas se você trabalha ou estuda frontend também pode encontrar algumas informações interessantes.

*O que é mesmo o Node.js? Se precisar, pode conferir*[*aqui*](https://youtu.be/8VSTrZY8vwI)*.*

Vamos nessa?!



**O que são os módulos do JavaScript?**

O conceito de modularidade é “esconder” detalhes de implementação das partes que compõem uma aplicação, e assim organizar e separar melhor o código. O objetivo de modularizar o código é permitir que aplicações maiores sejam montadas de forma “modular”; ou seja, através de várias partes independentes.

A primeira forma adotada pelo JavaScript para permitir a modularidade foi o chamado **CJS** (*Common JS*, ou JavaScript comum), baseado na função require().

O CJS, embora fosse conhecido como o “padrão” do Node.js para módulos, nunca foi definido como ferramenta de modularidade oficial da linguagem, pois basicamente cria as exportações a partir da alteração do objeto global exports do Node.js (ao invés de utilizar métodos próprios para isso).

Achou muita coisa? Calma, vamos ver com mais detalhes o que isso significa no decorrer do artigo.

Por não existir esse método próprio, o ES6 especificou, aí sim oficialmente, o chamado **EcmaScript Modules** (que chamamos de **ESM** ou, em algumas documentações, de *ES6 Modules*), baseado nas palavras-chave import e export.

Porém, existe um “hiato” de tempo entre as especificações definidas para cada versão do JavaScript e a implementação de cada uma delas, tanto nos navegadores quanto no Node.js (e todo o seu ecossistema de bibliotecas); e por esse motivo até hoje é possível encontrar alguns bugs e *workarounds* (termo usado em programação para o que chamamos em bom português de “gambiarras”) para utilizar a sintaxe ESM com Node.js.

Assim, ainda é muito comum ver o uso do CJS e do require() em Node.js. E mesmo após a implementação do ESM e a adoção desta nova sintaxe pelas bibliotecas, boa parte das documentações ainda utiliza a forma anterior.

*Então, o que é o tal ECMA e por que o JavaScript depende que suas funcionalidades sejam implementadas nos navegadores e no Node.js? Respondemos essas e algumas outras dúvidas de JavaScript na websérie*[*Memes do Js*](https://www.youtube.com/watch?v=VHxoyduIt18&list=PLh2Y_pKOa4UeJOI2P-N6J8nGyV3aCuO3O&index=1&t=0s&ab_channel=AluraCursosOnline)*.*

**Por que utilizamos módulos?**

Vimos que o JavaScript utiliza duas formas de trabalhar com módulos.

Mas, afinal, o que são módulos em programação e para que são utilizados?

Vamos lá: Lembra que a definição mais básica de módulo é uma unidade, uma parte que pode ser combinada com outras mas que tem, por si mesma, uma funcionalidade. Podemos pensar em *móveis modulares*, que podem ser montados em conjunto de acordo com a necessidade, embora cada parte também funcione sozinha (um armário, por exemplo). Certo?!

Agora, vamos traduzir essa ideia para a programação: podemos chamar de módulo todo código que tem uma funcionalidade específica e que está implementado de forma independente - podendo assim ser *modularizado*- ou seja, utilizado em conjunto com outras partes do código para desenvolver uma aplicação completa.

Em aplicações desenvolvidas para Node.js, já é padrão a divisão das funcionalidades em quantos diretórios e arquivos forem necessários para manter o código separado e organizado.

Os navegadores têm que baixar e carregar todos os arquivos necessários para que a aplicação funcione corretamente - daí a utilidade dos *bundlers* ou empacotadores (como o [webpack](https://webpack.js.org/) que otimizam os arquivos para transferência e carregamento rápido pelos navegadores. O que não é o caso com aplicações executadas no ambiente do Node.js, que pode dispensar então esse processo de empacotamento/otimização para um único arquivo que é comum ao frontend.

O Node.js considera cada arquivo como um módulo separado e independente, com seu próprio *namespace*. Ou seja, todos os códigos definidos em um arquivo, sejam variáveis, funções ou classes, ficam restritos - ou seja, **privados** - e são acessados apenas pelo próprio arquivo em que foram criados, a não ser que sejam explicitamente **exportados** e **importados** em outro arquivo/módulo.

Também são considerados módulos bibliotecas externas que importamos em nossos projetos conforme a necessidade.

Por exemplo, construímos um módulo que valida a sequência numérica de cartões de crédito, antes que os cartões sejam utilizados pelo restante da aplicação:

*// arquivo validacoes/validacaoCartao.js*

function validaCartao(cartao) {

*// lógica interna da validação*

return cartaoEhValido

}

export default validaCartao;

Acima, a função validaCartao() está sendo explicitamente exportada através da linha export default validaCartao.

Qualquer outro módulo na aplicação que queira usar o validador deve importá-lo:

*// arquivo index.js*

import validaCartao from 'validacoes/validacaoCartao.js';

function enviaDadosCliente(dadosCliente) {

*// lógica interna*

const cartaoEhValido = validaCartao(dadosCliente.cartao);

*// lógica interna continua*

}

Acima, o arquivo/módulo index.js só consegue acessar validaCartao() e usar sua lógica interna a partir do momento em que o módulo é **importado** de seu arquivo original.

O mesmo acontece para bibliotecas e dependências externas. Por exemplo, se quisermos trabalhar com a biblioteca de validação [express-validator](https://express-validator.github.io/docs/), podemos importar somente os módulos que interessam e apenas no arquivo onde serão usados:

*// arquivo validacoes/index.js*

import { body } from 'express-validator';

A modularização em JavaScript também permite que sejam definidas as funções, classes, objetos ou variáveis de um módulo que serão exportados, mantendo o restante da implementação com acesso apenas no módulo onde foram definidos:

*// arquivo validacoes/validacaoCartao.js*

function validaCartao(cartaoRecebido) {

*// lógica interna da validação*

const resultado = funcaoAuxiliar(cartaoRecebido)

*// lógica interna da validação continua*

return cartaoEhValido

}

function funcaoAuxiliar(dado) {

*// lógica interna da função*

return resultado

}

export default validaCartao;

No exemplo acima, apenas a função validaCartao() está sendo exportada, e a função funcaoAuxiliar() está sendo usada apenas internamente pelo módulo validacaoCartao(), não ficando disponível para ser acessada pelo restante do código.

Ou seja, temos apenas uma “exportação padrão” neste módulo, da função que outras partes da aplicação precisam acessar. O restante da lógica (exemplificado aqui por funcaoAuxiliar) fica restrito ao módulo e não é acessado.

**Utilização de módulos no JavaScript**

Como vimos no início, existem duas formas de se trabalhar com módulos em JavaScript, usando a forma CJS ou ESM.

Vamos ver com mais detalhes como funciona cada uma dessas formas e quais as diferenças entre elas?! Bora!

**CJS**

O CJS, sintaxe adotada pelo Node.js desde antes da implementação do ESM, utiliza o objeto global exports para gerenciar as exportações de módulos e a função require() para gerenciar as importações.

*Importante: o CJS que veremos agora utiliza funções e objetos nativos do Node.js, como o objeto global exports, e não vai funcionar da mesma forma nos navegadores.*

Porém, era comum o uso de exports e require() em aplicações frontend através de *bundlers* como o [webpack](https://webpack.js.org/) que permitiam o uso deste recurso e “traduziam” o código para um formato de JavaScript que os navegadores (que não têm o objeto global exports e não entendem o CJS) pudessem interpretar.

Com o lançamento do ES6 e do ESM, os navegadores passaram a adotar esta que é a sintaxe “oficial” de importação e exportação com import e export (que veremos mais abaixo neste artigo).

O objeto global exports é sempre definido internamente pelo Node.js. Assim, quando queremos exportar vários módulos, atribuímos estes módulos como propriedades do objeto exports:

*// arquivo 'operacoes.js'*

exports.soma = function(num1, num2) {

return num1 + num2;

}

exports.multiplica = function(num1, num2) {

return num1 \* num2;

}

Ou, alternadamente:

*// arquivo 'operacoes.js'*

module.exports = {

soma(num1, num2) {

return num1 + num2;

},

multiplica(num1, num2) {

return num1 \* num2;

},

};

Caso você queira definir um módulo que exporte apenas uma função ou classe, ao invés de várias, a forma é bem parecida:

*// arquivo 'operacoes.js'*

function soma(num1, num2) {

return num1 + num2;

}

function multiplica(num1, num2) {

return num1 \* num2;

}

function subtrai(num1, num2) {

return num1 - num2;

}

module.exports = soma;

Na forma acima, não é necessário criar as funções dentro do objeto que será exportado ou declarar exports em todas as funções.

Podemos também exportar apenas parte das funções usando esta mesma forma:

module.exports = { multiplica, subtrai };

*O exemplo acima utiliza o recurso de desestruturação de objetos do JavaScript. Caso precise, você pode*[*conferir aqui*](https://youtu.be/f8a-qwKC5yk)*o que é desestruturação e ver alguns exemplos de uso.*

Agora que já exportamos nossos módulos, hora de importá-los para que possam ser usados em outras partes da aplicação. Para isso, usamos a função require().

Vamos primeiro exportar a função soma():

*// arquivo 'operacoes.js'*

function soma(num1, num2) {

return num1 + num2;

}

module.exports = soma;

Agora podemos importar soma() para uso em outro módulo:

*// arquivo 'index.js'*

const soma = require('./operacoes.js');

No exemplo acima, usamos require() para importar módulos do nosso próprio código, passando como parâmetro uma string com o caminho relativo do arquivo onde se encontram os módulos que queremos importar. O retorno da função require() é normalmente a função, classe ou objeto importado, que guardamos na variável soma.

*A utilização do mesmo nome na importação não é obrigatória, mas sim um padrão da linguagem.*

Alternadamente, é possível usar o recurso de desestruturação de objetos para importar apenas os módulos necessários:

*// arquivo 'operacoes.js'*

module.exports = {

soma(num1, num2) {

return num1 + num2;

},

multiplica(num1, num2) {

return num1 \* num2;

},

subtrai(num1, num2) {

return num1 - num2;

}

};

*// arquivo 'index.js'*

const { soma, multiplica } = require('./operacoes.js');

Ou, ainda, é possível importar o objeto completo e utilizar seus métodos da seguinte forma:

*// arquivo 'index.js'*

const operacoes = require('./operacoes.js');

const numerosSomados = operacoes.soma(4, 2) *//6*

O mesmo princípio vale para módulos externos ao nosso código, sejam os nativos do Node.js ou os que fazem partes de bibliotecas ou frameworks; mas nesse caso não é necessário passar o caminho do arquivo, somente o nome do módulo:

const { soma, multiplica } = require('./operacoes.js');

const fs = require('fs');

const express = require('express');

No exemplo acima, fs é um [módulo nativo do Node.js](https://nodejs.org/api/fs.html) utilizado para o Node.js interagir com o sistema de arquivos do computador. Já express é um [framework web](https://expressjs.com/) muito utilizado em desenvolvimento de APIs com Node.js; não é um módulo nativo e deve ser instalado manualmente como uma dependência de um projeto Node.js para que possa ser importado da forma acima e utilizado pelo código.

**ESM**

A partir do ES6 o JavaScript implementou o suporte real à modularidade como parte da linguagem, usando as palavras-chave import e export.

Conceitualmente, o princípio de modularidade permanece o mesmo do utilizado pelo Node.js:

* Cada arquivo é considerado um módulo independente;
* Funções, classes e variáveis definidos dentro de um arquivo não podem ser acessados por fora dele, a não ser que sejam explicitamente exportados e importados.

Vamos ver alguns exemplos de importação e exportação de módulos com essa sintaxe. **O ESM utiliza a palavra-chave export antes da declaração da função, variável ou classe**.

***Importante****: Para utilizar o ESM em aplicações Node.js, é necessário adicionar a propriedade type: module no arquivo package.json, conforme*[*este exemplo*](https://nodejs.org/docs/latest-v13.x/api/esm.html#esm_enabling)*.*

*// exportação: arquivo 'operacoes.js'*

export function soma(num1, num2) {

return num1 + num2;

}

export function multiplica(num1, num2) {

return num1 \* num2;

}

export function subtrai(num1, num2) {

return num1 - num2;

}

*// importação: arquivo 'index.js'*

import { soma, multiplica, subtrai } from './operacoes.js';

No exemplo acima, exportamos cada uma das funções com a palavra-chave export e utilizamos a desestruturação de objetos para importar cada uma delas em um outro arquivo.

É possível variar a chamada da exportação. Por exemplo, ao concentrar as exportações de módulos no fim do arquivo export { soma, multiplica, subtrai }; e ao usar a desestruturação para exportar e importar cada função em separado:

*// exportação: arquivo 'operacoes.js'*

function soma(num1, num2) {

return num1 + num2;

}

function multiplica(num1, num2) {

return num1 \* num2;

}

function subtrai(num1, num2) {

return num1 - num2;

}

export { soma, multiplica, subtrai };

*// importação: arquivo 'index.js'*

import { soma, multiplica, subtrai } from './operacoes.js';

Dessa forma, também é possível importar apenas as funções (ou variáveis, ou classes) de um módulo que serão utilizadas em cada parte da aplicação; ou seja, algumas funções exportadas de operacoes.js podem ser importadas por index.js, outras por app.js:

*// exportação: arquivo 'index.js'*

import { soma } from './operacoes.js';

*// importação: arquivo 'app.js' (outro arquivo da aplicação)*

import { multiplica, subtrai } from './operacoes.js';

Também é muito comum o cenário onde apenas uma função ou classe deve ser exportada, mesmo que existam outras funções, classes ou variáveis no arquivo. Neste caso, utiliza-se export default (exportação padrão, em tradução livre) ao invés de apenas export:

function validaCartao(cartao) {

*// lógica interna da validação*

const resultado = funcaoAuxiliar(algumDado)

*// lógica interna da validação continua*

return cartaoEhValido

}

function funcaoAuxiliar(dado) {

*// lógica interna da função*

return resultadoDaLogica

}

export default validaCartao;

Com o export default, não é utilizada a desestruturação para fazer a importação do módulo, pois o que está sendo exportado é o objeto completo:

*// importação: arquivo 'index.js'*

import validaCartao from './validacoes.js';

Enquanto a exportação com export só pode ser usada em funções nomeadas, a exportação padrão com export default pode ser feita em funções anônimas e também em objetos literais:

*// exportação: arquivo 'operacao.js'*

export default function(num1, num2) {

return num1 + num2;

}

*// importação: arquivo 'index.js'*

import operacao from './operacao.js';

No exemplo acima, podemos criar o identificador operacao na importação da função anônima; os identificadores dos imports se comportam como constantes.

Grande parte dos guias de estilo (linter) de JavaScript vão recomendar que todos os módulos utilizados em um arquivo sejam explicitamente exportados, com a sintaxe import { funcao1, funcao2, funcao3 } from ‘modulo’. Porém, ao importarmos um módulo que exporta várias declarações, é possível escrever de forma reduzida:

import \* as operacoes from './operacoes.js';

Na forma acima, todos os export (desde que não sejam export default) do módulo são importados como propriedades do objeto operacoes. Uma vez que todas as exportações não-padrão (não default) são necessariamente nomeadas, o identificador da função, classe ou variável se torna uma propriedade deste objeto. Por exemplo:

*// arquivo 'operacoes.js'*

function soma(num1, num2) {

return num1 + num2;

}

function multiplica(num1, num2) {

return num1 \* num2;

}

function subtrai(num1, num2) {

return num1 - num2;

}

export { soma, multiplica, subtrai };

*/////////////////////////*

*// arquivo 'index.js'*

import \* as operacoes from './operacoes.js';

operacoes.soma(1, 1) *//2*

operacoes.multiplica(1, 1) *//1*

operacoes.subtrai(1, 1) *//0*

Também é possível, embora não muito usual, declarar export e export default em um mesmo arquivo. Neste caso, a importação fica da seguinte forma:

*// arquivo 'operacoes.js'*

export default function(num1, num2) {

return num1 + num2;

}

export function multiplica(num1, num2) {

return num1 \* num2;

}

export function subtrai(num1, num2) {

return num1 - num2;

}

*/////////////////////////*

*// arquivo 'index.js'*

import soma, { multiplica, subtrai } from './operacoes.js';

soma(1, 1) *//2*

multiplica(1, 1) *//1*

subtrai(1, 1) *//0*

**A extensão .mjs**

Há alguns padrões que o JavaScript adota em suas extensões de arquivo para indicar os diferentes “tipos” de código.

Por exemplo: aplicações frontend que utilizam [React](https://www.alura.com.br/artigos/react-js) podem adotar o padrão .js para arquivos escritos em JavaScript “padrão” e .jsx para arquivos que utilizem os recursos da [extensão JSX](https://reactjs.org/docs/introducing-jsx.html), embora a funcionalidade do código e dos arquivos permaneça a mesma.

Da mesma forma, é possível utilizar os padrões .mjs para assinalar arquivos em JavaScript que sejam módulos ESM e diferenciá-los de arquivos JavaScript que não utilizem módulos (mantendo a extensão normal .js).

Na prática, usar ou não este padrão não traz diferença para o desenvolvimento da aplicação, porém o Node.js, “por baixo dos panos”, vai identificar as diferentes extensões ao carregar os programas e indexar os arquivos. Então, você pode utilizá-lo, se quiser, embora este tipo de decisão muitas vezes fique a cargo da convenção utilizada por cada projeto/empresa.

Os arquivos com extensão .cjs seguem o mesmo princípio, porém para CJS. Caso seja necessário utilizar a sintaxe CJS em aplicações que já usam o ESM (e que, por consequência, têm definido o ”type”: “module”; como propriedade no arquivo package.json, será necessário utilizar a extensão .cjs, embora não seja recomendável misturar as duas formas de importação em um mesmo projeto.

**Pontos importantes sobre o uso do ESM:**

* Na sintaxe ESM é necessário incluir o nome completo do arquivo no caminho, incluindo a extensão .js. Já o CJS não requer a extensão do arquivo, é possível utilizar const soma = require('./operacoes');, por exemplo. Nos exemplos de código que usamos neste artigo, decidimos manter o .js mesmo nas importações feitas com require() para facilitar a compreensão.
* Assim como no CJS, todas as importações devem estar declaradas no topo dos arquivos onde os módulos serão utilizados (\*), e não devem ser feitas dentro de funções, classes, loops ou outros blocos de código;
* Já as exportações podem seguir as formas indicadas nos exemplos, como:
  + Adicionando export antes de cada declaração;
  + Montando o objeto que será exportado com export { funcao1, funcao2 } na última linha do arquivo;
  + Declarando export default antes da declaração de uma função anônima ou
  + Declarando export default funcao na última linha do arquivo.

*(\*) Declarar os módulos importados no topo do arquivo é uma****convenção****seguida pela comunidade, mas não é decisiva para o funcionamento do código. Módulos importados passam pelo mesmo processo de*[*hoisting*](https://www.alura.com.br/artigos/hoisting-no-javascript)*de declarações de função e variáveis e são “içados” para o topo dos arquivos.*

**Qual devo usar, afinal?**

Uma vez que sabemos que o CJS não é, digamos assim, uma **implementação oficial** da linguagem para importação e exportação de módulos, o ideal é migrarmos nosso código para a sintaxe ESM, essa sim uma funcionalidade implementada a partir do ES6 especialmente para lidar com modularidade.

Porém, na prática, a forma CJS se consolidou como a “forma do Node.js” para trabalhar com modularidade, pois o objeto global exports e a função require() são *built-in* (embutidas) no Node.js, mas não nos navegadores.

O CJS permitiu que o Node.js desenvolvesse um padrão de organização de arquivos modular, naturalmente dividindo uma aplicação em vários arquivos/módulos diferentes conforme a necessidade, antes mesmo que isso fosse possível nos navegadores - inclusive permitindo aos navegadores utilizarem este recurso através de *bundlers*.

A partir da versão 13 o Node.js passou a ter suporte a ESM (no momento em que escrevemos este artigo a versão mais atual é a 18), o que não resolveu totalmente o problema, pois a partir desse momento o Node.js passou a ter, obrigatoriamente, suporte a duas formas diferentes (e não totalmente compatíveis) de se trabalhar com módulos.

*Além das diferenças nas palavras-chave, funções e objetos utilizados para importar e exportar módulos, o CJS e o ESM têm outras diferenças estruturais: uma diferença importante é que o CJS funciona de forma****síncrona****, enquanto o ESM funciona de forma****assíncrona****, o que dificulta a compatibilidade entre as duas formas.*

Você pode conferir este assunto com mais detalhes [neste artigo sobre async/await](https://www.alura.com.br/artigos/async-await-no-javascript-o-que-e-e-quando-usar).

Na prática, até agora grande parte das aplicações, bibliotecas e frameworks Node.js ainda utiliza o CJS em seus códigos. Embora essas aplicações, bibliotecas e etc. estejam desde a versão 13 sendo atualizadas para dar suporte ao ESM, ainda é possível encontrar alguns *bugs* de compatibilidade entre bibliotecas, além da maioria da documentação de bibliotecas e frameworks ainda utilizar a forma anterior (CJS).

Nossa recomendação: utilize a forma ESM em projetos com Node.js, uma vez que:

* O CJS está sendo (embora um pouco lentamente) substituído e
* O ESM foi implementado para ser a ferramenta de modularidade do JavaScript.

Caso você encontre *bugs* em bibliotecas e dependências (o que ainda tem acontecido com alguma frequência), sempre vale a pena procurar na documentação de cada uma a respeito de uso de módulos ou na parte de *issues* do repositório da ferramenta no GitHub (se estiver disponível).

Caso não encontre resposta, você sempre pode acessar um fórum ou postar um novo *issue* em caso de ferramentas de código aberto.

**E no frontend?**

Após a especificação do ESM no ES6, os navegadores começaram a implementar as novas funcionalidades em suas *engines*. O Firefox, por exemplo, implementou o suporte total ao ESM a partir da versão 60 (2018) (você pode checar a tabela de compatibilidade dos navegadores com o ESM [neste link do MDN](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules#browser_compatibility).

Antes da implementação total, eram utilizados *bundlers* de código para trabalhar com modularidade, mas hoje em dia todos os navegadores (com exceção do Internet Explorer) já têm o ESM nativo. Já o CJS nunca foi compatível com navegadores justamente por utilizar elementos que são nativos apenas do Node.js como o objeto global exports.

*Embora o ESM seja hoje implementado nativamente nos navegadores, é sempre interessante considerar o uso de bundlers como o webpack para gerar a versão de “produção” de uma aplicação frontend, pois estas ferramentas têm recursos para aumentar a performance do código que é executado no navegador.*

Existem outros detalhes no uso de módulos pelo JavaScript no frontend, como o uso da tag de HTML <script type=”module”>. Você pode conferir mais sobre o uso de JavaScript pelo frontend na formação da Alura [JavaScript para frontend](https://cursos.alura.com.br/formacao-javascript-front-end).

**Conclusão**

Neste artigo abordamos as duas formas de trabalhar com importação e exportação de módulos com JavaScript, dando destaque ao uso com Node.js.

E você pode aprofundar ainda mais o conteúdo sobre as APIs nativas do Node.js, como foram implementadas e como funcionam! Para isso, indico consultar a [documentação oficial da última versão](https://nodejs.org/docs/latest-v18.x/api/) (nesse momento, a versão 18) e conferir sempre os nossos conteúdos.