

ΠΟΤΕΡΑΜΕΝΤΟ ΕΝ  
ΜΟΔΟΥΛΟ HTTP

# ROTEAMENTO EM APIS

O **roteamento** em *APIs* refere-se ao processo de mapear *URLs* para funções ou controladores específicos dentro de um **servidor**. Ele define como as **requisições HTTP** (**GET, POST, PUT, DELETE**, etc.) são tratadas e direcionadas para os recursos adequados.

# COMO FUNCIONA O ROTEAMENTO EM APIS?

Quando um **cliente** (navegador, frontend ou outra API) faz uma requisição a um **servidor**, o roteador analisa o caminho (*endpoint*) e o *método HTTP* utilizado, e então encaminha essa requisição para a função correspondente.

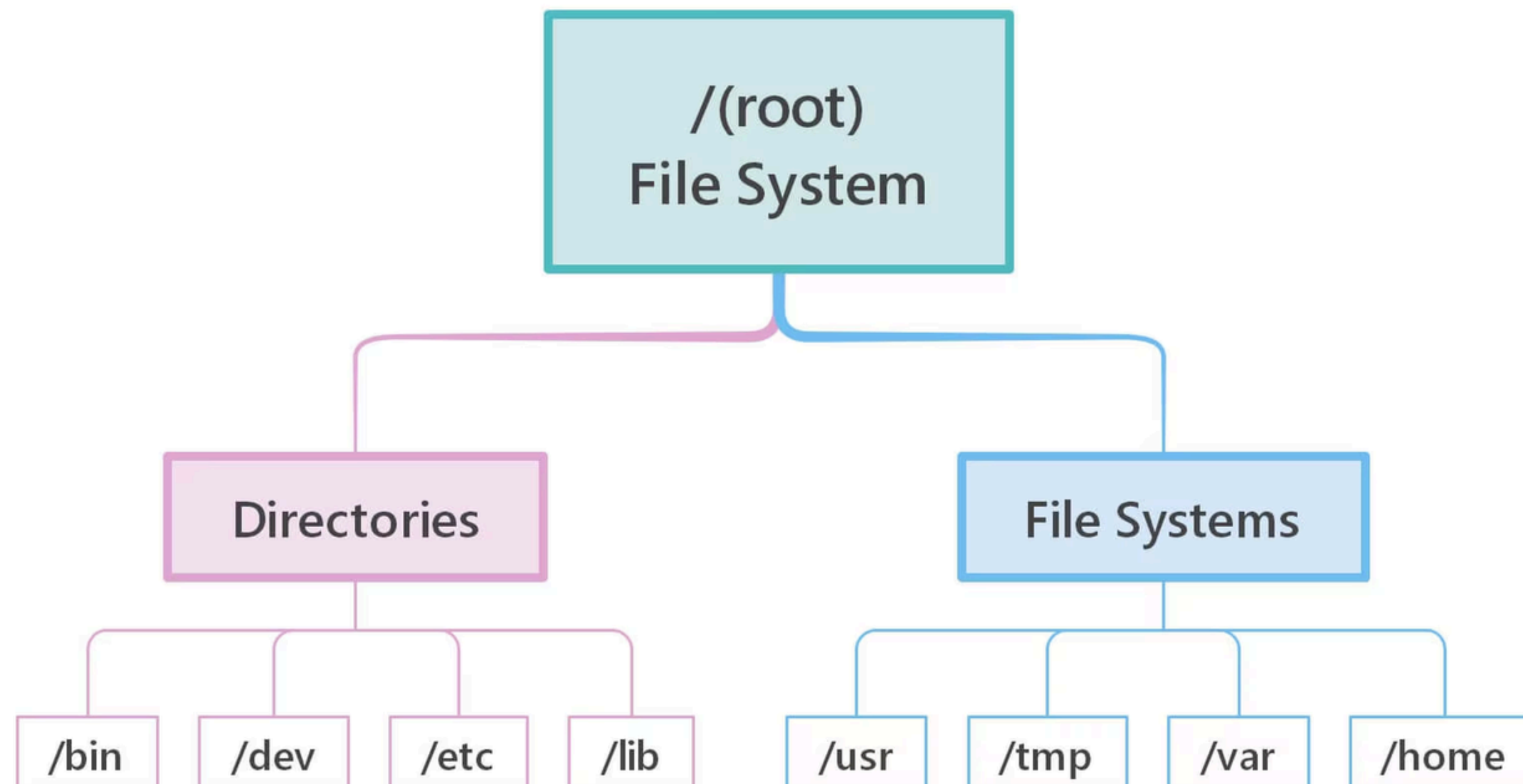
GET	/pet/{petId}	Find pet by ID
PUT	/pet	Update an existing pet
DELETE	/pet/{petId}	Deletes a pet
POST	/pet/{petId}/uploadImage	uploads an image

# BOAS PRÁTICAS PARA ROTEAMENTO

1. Organização por módulos: Separe as rotas em arquivos específicos para cada recurso (/users, /products etc.).
2. Uso de Middleware: Autenticação, logs e tratamento de erros devem ser aplicados nas rotas corretamente.
3. Rotas RESTful: Utilize os métodos HTTP de forma semântica (GET para leitura, POST para criação, etc.).
4. Roteamento hierárquico: Estruture suas rotas de forma lógica, como:
  - GET /users -> Lista todos os usuários
  - GET /users/{id} -> Busca um usuário específico
  - POST /users -> Cria um novo usuário
  - PUT /users/{id} -> Atualiza um usuário
  - DELETE /users/{id} -> Remove um usuário

# MÓDULO FILE SYSTEM

O módulo **File System (fs)** do Node.js permite **ler**, **criar**, **modificar** e **excluir** arquivos e diretórios no sistema operacional. Ele pode ser usado tanto de forma síncrona quanto assíncrona.



# IMPORTANDO O MÓDULO FS

Antes de usar, precisamos importar o módulo:



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

Se quiser usar **Promises** e **async/await**, use:



```
1 const fs = require('fs').promises;
```

# OPERAÇÕES COM ARQUIVOS

## Criar ou Escrever em um Arquivo

- O método **writeFile** cria um novo arquivo ou sobrescreve um existente.
- **Forma assíncrona** (recomendada para evitar travamento da aplicação):

```
1 fs.writeFile('arquivo.txt', 'Hello, Node.js!', (err) => {  
2   if (err) throw err;  
3   console.log('Arquivo criado com sucesso!');  
4 });
```

- **Forma síncrona** (trava o código até concluir a escrita):

```
1 fs.writeFileSync('arquivo.txt', 'Hello, Node.js!');  
2 console.log('Arquivo criado!');
```

# OPERAÇÕES COM ARQUIVOS

## Adicionar Conteúdo a um Arquivo

- O método **appendFile** adiciona conteúdo sem apagar o que já existe:

```
1 fs.appendFile('arquivo.txt', '\nNova linha adicionada!', (err) => {  
2   if (err) throw err;  
3   console.log('Conteúdo adicionado!');  
4 });
```



# OPERAÇÕES COM ARQUIVOS

## Ler um Arquivo

- O método **readFile** permite ler um arquivo de forma assíncrona:

```
1 fs.readFile('arquivo.txt', 'utf8', (err, data) => {  
2   if (err) throw err;  
3   console.log('Conteúdo do arquivo:', data);  
4 });
```

## Ler um Arquivo

- O método **readFileSync** permite ler um arquivo de forma assíncrona:

```
1 const data = fs.readFileSync('arquivo.txt', 'utf8');  
2 console.log('Conteúdo do arquivo:', data);
```

# OPERAÇÕES COM ARQUIVOS

## Renomear um Arquivo

- O método **rename** altera o nome do arquivo:

```
1 fs.rename('arquivo.txt', 'novo-nome.txt', (err) => {  
2   if (err) throw err;  
3   console.log('Arquivo renomeado!');  
4 });
```

## Excluir um Arquivo

- Para deletar um arquivo, usamos **unlink**:

```
1 fs.unlink('novo-nome.txt', (err) => {  
2   if (err) throw err;  
3   console.log('Arquivo deletado!');  
4 });
```

# OPERAÇÕES COM DIRETÓRIOS

## Criar um Diretório

- Criamos pastas com **mkdir**:

```
1 fs.mkdir('novaPasta', (err) => {  
2   if (err) throw err;  
3   console.log('Diretório criado!');  
4 });
```

- Para criar um diretório **dentro de outro diretório**, usamos **{ recursive: true }**:

```
1 fs.mkdir('pasta1/pasta2', { recursive: true }, (err) => {  
2   if (err) throw err;  
3   console.log('Diretórios criados!');  
4 });
```

# OPERAÇÕES COM DIRETÓRIOS

## Ler o Conteúdo de um Diretório

- O método **readdir** lista os arquivos dentro de uma pasta:

```
1 fs.readdir('.', (err, files) => {  
2   if (err) throw err;  
3   console.log('Arquivos na pasta:', files);  
4 });
```

# OPERAÇÕES COM DIRETÓRIOS

## Remover um Diretório

- Para apagar um diretório vazio, usamos **rmdir**:

```
1 fs.rmdir('novaPasta', (err) => {  
2   if (err) throw err;  
3   console.log('Diretório removido!');  
4 });
```

- Para apagar um **diretório com arquivos dentro**, usamos **{ recursive: true }**:

```
1 fs.rm('pasta1', { recursive: true, force: true }, (err) => {  
2   if (err) throw err;  
3   console.log('Diretório e arquivos removidos!');  
4 });
```

# USANDO FS.PROMISES COM ASYNC/AWAIT

Se preferir uma abordagem mais moderna e limpa, podemos usar fs.promises:

*Exemplo: Criando e Lendo um Arquivo*

```
1 const fs = require('fs').promises;
2
3 async function manipularArquivo() {
4     try {
5         await fs.writeFile('asyncFile.txt', 'Conteúdo assíncrono');
6         console.log('Arquivo criado!');
7
8         const data = await fs.readFile('asyncFile.txt', 'utf8');
9         console.log('Conteúdo:', data);
10    } catch (err) {
11        console.error('Erro:', err);
12    }
13 }
14
15 manipularArquivo();
```

# CONCLUSÃO

O módulo *fs* é essencial para manipulação de arquivos e diretórios no Node.js. Ele oferece duas abordagens:

- **Callback** (*fs.writeFile, fs.readFile*) → Código tradicional, mas pode ser verboso.
- **Promises** (*fs.promises*) + *async/await* → Código mais limpo e moderno.

Se precisar lidar com muitos arquivos e diretórios, é recomendável usar *fs.promises* para evitar o problema de *callback hell*.

# O QUE É O MÓDULO PATH?


O módulo *path* no Node.js é um módulo **nativo** que permite manipular e resolver caminhos de arquivos e diretórios de forma **segura** e **multiplataforma**. Ele evita problemas causados por diferenças entre sistemas operacionais, como o uso de **barras normais (/)** no **Linux/macOS** e **barras invertidas (\)** no **Windows**.



# PRINCIPAIS MÉTODOS DO PATH

## 1. path.join() → Junta segmentos de caminho

Cria um caminho correto, independentemente do sistema operacional.



```
1 const path = require('path');  
2  
3 const caminho = path.join('pasta', 'subpasta', 'arquivo.txt');  
4 console.log(caminho);  
5 // No Windows: "pasta\subpasta\arquivo.txt"  
6 // No Linux/macOS: "pasta/subpasta/arquivo.txt"
```

# PRINCIPAIS MÉTODOS DO PATH

## 2. `path.resolve()` → Resolve um caminho absoluto

Cria um caminho absoluto baseado no diretório atual.



```
1 const caminhoAbsoluto = path.resolve('pasta', 'subpasta', 'arquivo.txt');  
2 console.log(caminhoAbsoluto);  
3 // Exemplo de saída: "/home/usuario/projeto/pasta/subpasta/arquivo.txt" (Linux/macOS)  
4 // Ou "C:\Users\Usuario\projeto\pasta\subpasta\arquivo.txt" (Windows)
```

## 3. `path.basename()` → Retorna o nome do arquivo



```
1 const arquivo = path.basename('/caminho/para/arquivo.txt');  
2 console.log(arquivo); // "arquivo.txt"
```

# PRINCIPAIS MÉTODOS DO PATH

## 4. path.dirname() → Retorna o diretório do arquivo



```
1 const diretorio = path.dirname('/caminho/para/arquivo.txt');  
2 console.log(diretorio); // "/caminho/para"
```

## 5. path.extname() → Retorna a extensão do arquivo



```
1 const extensao = path.extname('/caminho/para/arquivo.txt');  
2 console.log(extensao); // ".txt"
```

# PRINCIPAIS MÉTODOS DO PATH

6. `path.parse()` → Transforma um caminho em um objeto



```
1  const info = path.parse('/caminho/para/arquivo.txt');
2  console.log(info);
3  /*
4  {
5    root: '/',
6    dir: '/caminho/para',
7    base: 'arquivo.txt',
8    ext: '.txt',
9    name: 'arquivo'
10 }
11 */
```

# PRINCIPAIS MÉTODOS DO PATH

7. `path.format()` → Monta um caminho a partir de um objeto



```
1 const caminhoMontado = path.format({  
2   dir: '/caminho/para',  
3   base: 'arquivo.txt'  
4 });  
5 console.log(caminhoMontado); // "/caminho/para/arquivo.txt"
```

# POR QUE USAR PATH?

- Evita erros ao criar caminhos manualmente.
- Funciona corretamente em Windows, Linux e macOS.
- Facilita a manipulação de arquivos e diretórios.

*Exemplo prático:*

```
1 const path = require('path');  
2  
3 const arquivo = 'dados.json';  
4 const caminho = path.join(__dirname, 'arquivos', arquivo);  
5  
6 console.log(caminho);  
7 // "/home/usuario/projeto/arquivos/dados.json" (Linux/macOS)  
8 // "C:\\Users\\Usuario\\projeto\\arquivos\\dados.json" (Windows)
```

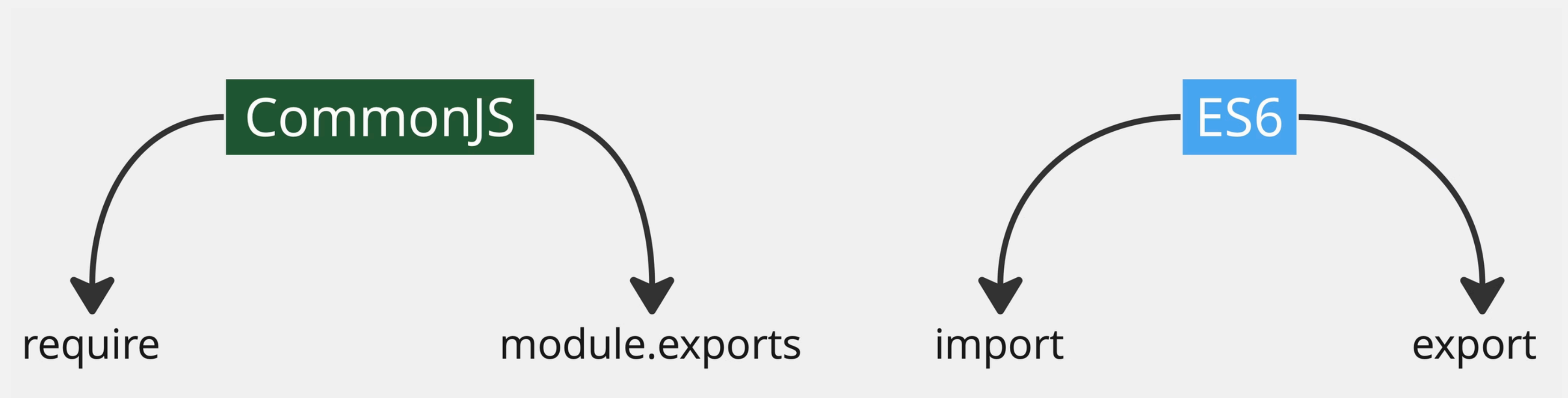
**\_\_dirname** → Obtém o diretório atual do arquivo.

# CONCLUSÃO

O módulo *path* é essencial para trabalhar com **arquivos e diretórios** no Node.js de forma segura, especialmente quando o código precisa rodar em **diferentes sistemas operacionais**.

# A EVOLUÇÃO DO JAVASCRIPT

Antes do ES6, o Javascript possuía sistemas de controles de módulos como o RequireJS, CommonJS e o sistema de injeção de dependências do Angular. Outras ferramentas como o Browserify e o próprio Webpack resolveram muitos problemas nesse contexto. Em 2015 tivemos uma primeira implementação do sistema de módulos no Javascript vanilla para Node.js utilizando o `require` que todos conhecemos. Mas isto ainda não chegou para os browsers.





# STRICT MODE

O strict mode faz várias mudanças nas semânticas normais do JavaScript. Primeiro, o strict mode elimina alguns erros silenciosos do JavaScript fazendo-os lançar exceções. Segundo, o strict mode evita equívocos que dificultam que motores JavaScript realizem otimizações: código strict mode pode às vezes ser feito para executar mais rápido que código idêntico não-strict mode. Terceiro, strict mode proíbe algumas sintaxes que provavelmente serão definidas em versões futuras do ECMAScript.

```
> (function() {  
  'use strict'  
  variable = 'hey'  
})();  
  
✖ ▶ Uncaught ReferenceError: variable is not defined  
   at <anonymous>:3:12  
   at <anonymous>:4:3  
  
> (() => {  
  'use strict'  
  myname = 'Flavio'  
})();  
  
✖ ▶ Uncaught ReferenceError: myname is not defined  
   at <anonymous>:3:10  
   at <anonymous>:4:3
```

# MODULE EXPORT

No modelo CommonJS podemos exportar valores atribuindo eles ao `module.exports` como no snippet abaixo:

```
1  module.exports = 1
2  module.exports = NaN
3  module.exports = 'foo'
4  module.exports = { foo: 'bar' }
5  module.exports = [ 'foo', 'bar' ]
6  module.exports = function foo () {}
7  module.exports = () => {}
```

No ES6, os módulos são arquivos que dão export uma API (basicamente igual ao CommonJS). As declarações, variáveis, funções e qualquer coisa daquele módulo existem apenas nos escopos daquele módulo, o que significa que qualquer variável declarada dentro de um módulo não está disponível para outros módulos (a não ser que eles sejam exportados explicitamente como parte da API, e importados posteriormente no módulo que as usa).

# EXPORT PADRÃO

Podemos simular o comportamento do CommonJS basicamente trocando o `module.exports` por `export default`:

```
1  export default = 1
2  export default = NaN
3  export default = 'foo'
4  export default = { foo: 'bar' }
5  export default = [ 'foo', 'bar' ]
6  export default = function foo () {}
7  export default = () => {}
```

Ao contrário dos módulos no CommonJS, declarações `export` só podem ser colocadas no top level do código, e não em qualquer parte dele. Presumimos que essa limitação existe para tornar mais fácil a vida dos interpretadores quando vão identificar os módulos, mas, olhando bem, é uma limitação bem válida, porque não há muitas boas razões para que possamos definir exports dinâmicos dentro das funções da nossa API.

# MELHORES PRÁTICAS COM EXPORT

Todas essas possibilidades de exportar um módulo vão introduzir um pouco de confusão na cabeça das pessoas. Na maioria dos casos é encorajado utilizar apenas um export default (e fazer isso só no final do módulo). Então você pode importar a API como o nome do próprio módulo.

```
1  var api = {  
2      foo: 'bar',  
3      baz: 'fooz'  
4  }  
5  
6  export default api
```

# MELHORES PRÁTICAS COM EXPORT

Os benefícios são:

- A interface que é exportada se torna bem óbvia, ao invés de termos que ficar procurando aonde exportamos a interface dentro do módulo.
- Você não cria a confusão sobre onde usar um export default ou um export nomeado (ou uma lista de exports nomeados). Tente se manter no export default
- Consistência. No CommonJS é normal usar um único método em um módulo. Fazer isso com exports nomeados é impossível porque você vai estar expondo um objeto com um método dentro, a não ser que você use o as default no export de lista. Já o export default é mais versátil porque você pode exportar só uma coisa
- Usar export default torna o import mais rápido

# IMPORTANDO EXPORTS NOMEADOS

É muito parecido com o destructuring assignment que temos na nova especificação.

```
import { map, reduce } from 'lodash'
```

Uma outra maneira que podemos também importar os export nomeados, é dar um alias para cada um, ou então apenas para um deles:

```
import { cloneDeep as clone, map } from 'lodash'
```

Você pode misturar os named imports com os exports padrões sem usar as chaves (mas aí você não vai poder dar um alias para eles):

```
1  import { default, map } from 'lodash'
2  import { default as _, map } from 'lodash'
3  import _, { map } from 'lodash'
```

THANK  
YOU

@wallace027dev