JS-12: JavaScript Moderno

Aula 3: Destructuring e ES6 Modules

Destructuring: extraindo informações de objetos e arrays

Com certa frequência precisamos atribuir dados advindos de uma lista ou de uma propriedade de objeto a uma ou mais variáveis. Embora essa tarefa não seja particularmente difícil de ser feita com os recursos que temos hoje, em especial quando precisamos atribuir valores a várias variáveis, esta ainda assim, pode ser no mínimo monótona.

A *Destructuring Assignment (Atribuição via Desestruturação)* é uma maneira bem legal e concisa de realizar essa extração de informações e atribuí-las a variáveis e/ou constantes quando se é necessário.

Vejamos alguns exemplos desse recurso:

```
const usuario = {
        username: “marcio.oliveira”,
        id: 5265,
        nome: “Marcio”
        sobrenome: “Oliveira”
        email: “marciooliveira99@gmail.com”,
        isAtivo: true,
        idade: 27
        endereco: {
                logradouro: “Rua da Cruz”,
                numero: 54
                complemento: “N/A”,
                cidade: “Osasco”,
                estado: “SP”
        }
}
```

No código anterior temos um objeto chamado “usuario” com várias propriedades. Vamos imaginar que precisamos guardar certas informações desse usuário em variáveis. Para conseguir esse resultado tínhamos o seguinte:

```
let id = usuario.id
let email = usuario.email
```

let isAtivo = usuario.isAtivo

Aqui estamos extraindo 3 informações do objeto “usuario” para armazenar em variáveis distintas: id, email e isAtivo.

Agora veja como ficaria usando a *Destructuring Assignment*. O operador de desestruturação para **OBJETOS** são as chaves **{ }**. Confira:

let { id, email, isAtivo **}** = usuario

Pronto!

Aqui estamos pedindo pro JS criar três variáveis com o mesmo nome das propriedades do objeto “usuario”, ou seja, o JavaScript vai criar a variável *id* em memória, vai procurar no objeto “usuario” uma propriedade chamada *id* e vai atribuir à variável *id* criada o valor dessa propriedade. Também fará o mesmo processo com *email* e *isAtivo*.

Bem simples, não?!

Mas e se quiséssemos atribuir os valores das mesmas propriedades que usamos no exemplo acima só que em variáveis com nomes diferentes (e não os mesmos nomes das propriedades do objeto)?

Também é bastante simples, veja:

let { id: **idUsuario**, email: **emailUsuario**, isAtivo: **usuarioAtivo }** = usuario
console.log(idUsuario) *// 5265*
console.log(emailUsuario) *// marciooliveira99@gmail.com*
console.log(usuarioAtivo) *// true*

Aqui estamos informando ao JavaScript para atribuir o valor de um propriedade no objeto para uma variável específica (e não com o mesmo nome) com o comando *{ propriedade: variável }*. Sendo assim, para a propriedade *id* do objeto foi criada a variável *idUsuario*; para *email* foi criada a variável emailUsuario e por fim, para a propriedade *isAtivo* a variável *usuarioAtivo*.

É possível inclusive a extração de valores de subpropriedades (propriedades presentes em objetos dentro de outros objetos). Por exemplo:

let { email: emailUsuario, endereco: **{** cidade **}**, isAtivo **}** = usuario
console.log(emailUsuario) *// marciooliveira99@gmail.com*
console.log(cidade) *// Osasco*
console.log(isAtivo) *// true*

Aqui extraímos a informação *cidade* que se encontra dentro do objeto da propriedade *endereco*. Assim, sinalizamos para o JavaScript que do objeto existente na propriedade *endereco* queremos só a *cidade* e que ele crie uma variável com esse nome mesmo em memória (endereco: { cidade }).

Bom… até o momento vimos como extrair informações de um **OBJETO**. Agora vejamos como podemos extrair informações de um **ARRAY** e atribuir esses dados a variáveis distintas.

Entretanto, antes vamos ver como faríamos isso SEM a *Destructuring Assignment*.

```
const numeros = [12, 5, 78, 32, 9]
let cinco = numeros[1] // posição 1 = valor 5
let doze = numeros[0] // posição 0 = valor 12
let nove = numeros[4] // posição 4 = valor 9
```

Com o uso da *Destructuring Assignment*, vejamos como ficaria. O operador de desestruturação para **ARRAYS** são os colchetes **[ ]**.

```
const numeros = [12, 5, 78, 32, 9]
let [doze, cinco, , , nove] = numeros
console.log(doze, cinco, nove) // exibe: 12  5  9
```

Perceba que colocamos os nomes das variáveis exatamente nas posições correspondentes aos valores que gostaríamos de extrair, por isso deixamos algumas posições vazias (**[** doze, cinco, (vazio)**,** (vazio)**,** nove**]**) isso informa ao JavaScript para desconsiderar os valores dessas posições.

Também, se quiséssemos, poderíamos atribuir uma parte dos valores à variáveis específicas e o restante a uma variável única. Podemos realizar isso com o *Spread Operator*.
Veja:

```
let [doze, cinco, ...outros] = numeros
console.log(doze, cinco, outros) // exibe: 12  5  [78,32,9]
```

ES6 Modules: trabalhando com a modularização do ES6

À medida em que uma aplicação cresce, também aumenta a quantidade de código escrita e, se não bem organizado, a manutenção de toda essa estrutura de código pode se tornar completamente inviável. É por isso mesmo que é considerada uma boa prática a divisão de todo nosso código em unidades especializadas, cada uma sendo responsável por uma ação específica. Dependendo da metodologia aplicada em um determinado projeto, o nome dado a

esse processo separatório pode variar, porém duas nomenclaturas comumente utilizadas são: componentização ou **modularização**.

Sendo assim, um **módulo** seria simplesmente *um conjunto de códigos que tem por responsabilidade a resolução de um problema preciso dentro do ecossistema da aplicação que está sendo desenvolvida*.

Outra característica do **módulo** é a possibilidade deste *se comunicar ou ser utilizado* dentro de outros **módulos**. Desta forma, o **módulo** deve *expor (ou seja, tornar público) apenas aquilo que de fato é necessário para que este seja utilizado por quem precisar dentro da aplicação*.

Com esse conceito em mente, o ECMAScript implementou uma solução bastante sofisticada que atende a esses pontos mencionados acima que de forma resumida seria:

- **Isolamento de escopo:** tudo que for declarado/escrito dentro de um módulo só existe dentro do módulo (são privados). Isto é, não ficam disponíveis e visíveis. Só será visível aquilo que for EXPLICITAMENTE tornado público pelo módulo. Isso é possível através do atributo **type="module"** da tag **<script>**.

- **Importação de módulos:** possibilidade de importar os recursos de um módulo dentro de outro que precisa utilizá-lo dentro da aplicação. Isso é possível através do novo comando **import** do ES6.

- **Tornar recursos do módulo disponíveis publicamente:** para que o módulo criado possa ser utilizado onde necessário, os recursos necessários para seu uso precisam estar acessíveis quando estes módulos são importados. A exposição dos recursos de um módulo se dá através do novo comando **export** do ES6.

Antes dos *Módulos do ES6* fazíamos o uso das chamadas **IIFE's** *(Immediately Invoked Function Expressions)* para criarmos nossos módulos. Essas funções, normalmente, retornavam um objeto contendo os métodos e propriedades necessários para o uso de um módulo em questão.

Como exemplo, podemos pensar num módulo que administra produtos dentro de uma aplicação. Esse módulo teria, no mínimo, quatro métodos para administração de produtos: inserir, atualizar, excluir e selecionar.

Um módulo com essas características utilizando a sintaxe de uma IIFE ficaria assim:

```
const moduloProduto = (function() {

    /* Todo o código necessário para o funcionamento do módulo */
```

```
        // objeto retornado pelo módulo que será público
        return {
            inserir: inserirProduto, // função interna chamada inserirProduto
            excluir: excluirProduto, // função interna excluirProduto
            atualizar: atualizarProduto, // função interna atualizarProduto
            selecionar: selecionarProduto // função interna selecionarProduto
        }
}();
```

Aqui temos um módulo publico (global) chamado *moduloProduto*. Quando importamos o arquivo .js na nossa aplicação que contém esse módulo, todos os demais arquivos .js importados na aplicação podem utilizá-lo em seu código interno (porque *moduloProduto* aqui existe globalmente) e seu uso se dá através dos métodos expostos pelo objeto retornado pela IIFE, ou seja, os métodos: inserir, excluir, selecionar e atualizar.

De certa forma o uso das IIFEs resolvia o problema pois todo o código NÃO PÚBLICO ficaria dentro do escopo dela (isto é, privado) e apenas as funções expostas através do objeto que representa o módulo público é que podem ser acessadas pelos demais códigos que pretendem utilizar esse módulo. Porém, havia um problema: a ordem de importação dos arquivos .js interferia na execução do código.

Por exemplo: se tivermos um módulo chamado **moduloCarrinho** num arquivo chamado *carrinho.js*, e esse módulo precisa do objeto **moduloProduto** (que estaria num arquivo chamado *produto.js*) assim que a aplicação carrega (onload), temos que ter certeza que a importação do arquivo *produto.js* vem antes do *carrinho.js*, se não teremos um erro de referência (ReferenceError) dizendo que *moduloProduto não está definido*.

Veja:

```
<script src="js/carrinho.js"></script> // ReferenceError: moduloProduto is not defined
<script src="js/produto.js"></script>
```

PORÉM se estruturado assim…

```
<script src="js/produto.js"></script>
<script src="js/carrinho.js"></script> // OK moduloProduto foi definido no script anterior
```

Ficar monitorando a ordem da importação dos arquivos de script é bastante cansativo e inviável, principalmente em aplicações grandes onde diversos módulos podem ser usados em diversos momentos e de diversas formas.

Os módulos do ES6 resolvem esse problema pois temos apenas uma importação de arquivo via tag <script> (que representaria a aplicação como um todo) e dentro dela teríamos a

importação dos scripts iniciais necessários. Além disso, cada script pode internamente definir quais outros scripts deseja importar, ou seja, cada script/módulo tem autonomia para explicitamente dizer quais outros módulos deseja importar para dentro de si, não sendo mais preciso se preocupar com várias importações de arquivos via <script> e tampouco com a ordem em que são importados.

Vejamos:

NO HTML:

```
<script src= “js/app.js” type= “module”></script>
```

NO *js/app.js*:

```
/* Importamos os scripts necessários iniciais */
import “./modulo1.js”
import “./modulo2.js”
import “./modulo3.js”
```

NO *js/modulo3.js* poderíamos ter por exemplo:

```
export function funcao1() {
    /* faz alguma coisa */
}

export function funcao2() {
    /* faz alguma coisa */
}

export function funcao3() {
    /* faz alguma coisa */
}
```

NOS MÓDULOS INDIVIDUAIS (exemplo *js/modulo2.js*):

```
/* Importação de um módulo dentro de outro */
import * as modulo4 from “./modulo4.js”;
import { funcao1, funcao2 } from “./modulo3.js”;
```

Como podemos ver, no *modulo2.js* importamos os módulos e recursos que são necessários para o funcionamento dele. Observe que importamos o *modulo3.js* e especificamos que queremos dele apenas a *funcao1* e a *funcao2*, ainda que este módulo tenha mais funções e recursos disponíveis, apenas estes são necessários no *modulo2.js.*

Detalhes sobre o uso de Módulos ES6

1. Em um módulo podemos exportar *variáveis, constantes, funções, arrow functions e classes*. Não há um limite para o número de itens que podemos tornar acessíveis em nossos módulos, sendo assim, podemos usar o comando **export** quantas vezes quisermos/precisarmos. Contudo, só podemos ter UM ÚNICO item exportado como padrão em nosso módulo, isto é, só podemos ter UM **export default** por módulo.

2. O comando export possui algumas regras para ser utilizado. Ele pode ser usado diretamente na frente do item a ser tornado acessível. Ex:

   export const dados = **[ ]**; *// exportação de variável*

   // exportação de função
   export function mostrarMensagem() **{ }**

   // exportação de classe
   export class MinhaClasse **{}**

   Porém, só podemos usar o comando **export default** junto de declaração de funções e classes. No caso de variáveis, constantes e arrow functions, só depois de declaradas. Ex:

   export default const dados = **[ ]**; *// GERA ERRO*

   const dados = **[ ]**; *// Declaração da constante: OK*
   export default dados; *// OK*

   No caso de classes e funções, podemos usar o export default sem problemas:

   export default function mostrarMensagem() **{ }** // sem problemas

   export default class MinhaClasse **{ }** // sem problemas

   OBS.: *Lembre-se que um módulo só pode ter UM ÚNICO item exportado como padrão.*

3. Para importar recursos de um módulo precisamos ficar atentos ao tipo de item que será importado. Caso queiramos importar recursos que NÃO SEJAM definidos como padrão

no módulo (**export default**), utilizamos as **chaves** (**{ }**) para dizer o que iremos importar. Ex:

```
import { adicionarCategoria, removerCategoria } from "./moduloCategoria.js";
```

Aqui estamos informando ao JavaScript que importe do "moduloCategoria.js" apenas os recursos "adicionarCategoria" e "removerCategoria".

4. Se quisermos importar apenas um recurso que está definido como padrão dentro de um módulo, não precisamos usar as chaves no processo de importação. Veja:

```
import recursoPadraoDoModulo from "./moduloCategoria.js";
```

Como pode notar, como estamos importando um recurso padrão (**export default**), não é necessário definir as chaves, basta usar o nome dele diretamente.

5. Se queremos fazer ambas as coisas, isto é, importar o recurso padrão e também os que não forem padrão, usamos um mix das duas abordagens:

```
import recursoPadraoDoModulo, { recurso1, recurso2 } from "./meuModulo.js";
```

6. Ao usamos o **operador asterisco (*)** numa importação, dizemos ao JavaScript que queremos que ele importe tudo o que há no módulo (seja padrão ou não). Só que para isso, somos obrigados também a usar o comando **as** para definir um apelido para o módulo. Esse apelido será um objeto contendo todos os recursos do módulo. Vejamos como funciona:

```
// Categorias vai ser o objeto que conterá tudo o que o moduloCategoria tem
import * as Categorias from "./moduloCategoria.js";

Categorias.recurso1;
Categorias.funcao1();
Categorias.funcao2();
```

7. Nos demais tipos de importação também temos a possibilidade de usar o comando **as** para trocar o nome dos recursos que importamos. Isso ajuda no caso de importamos módulos diferentes que possuem recursos com nomes iguais (o que daria conflito). Com o comando **as** podemos resolver esse problema renomeando o recurso importado.

```
import { adicionar as adicionarCategoria } from "./moduloCategoria.js";
import { adicionar as adicionarProduto } from "./moduloProduto.js";
```

No caso acima, tanto o "moduloCategoria.js" quanto o "moduloProduto.js" tem uma

função chamada “adicionar”. Ainda que elas façam coisas diferentes, o problema na importação dessas duas funções num mesmo módulo é que elas possuem o mesmo nome. Ou seja, ao chamar **adicionar()** não teria como o JavaScript saber se você está chamando o “adicionar” do módulo de produto ou o do módulo de categoria.

Com o comando de renomeação **“as”** esse conflito é resolvido, pois o *adicionar() de Categoria* será tratado aqui com o apelido de **adicionarCategoria()** e o *adicionar() de Produto* será tratado aqui como **adicionarProduto()**.