Curso Superior de Tecnologia em Sistemas para Internet
Desenvolvimento front-end I (front 1)
Rafael Cunha Cardoso
rafaelcardoso@ifsul.edu.br

Desestruturação

JavaScript

Desestruturação

Funcionalidade do JavaScript introduzida no ECMAScript 2015 (ES6) que permite **extrair valores de arrays ou propriedades de objetos** em variáveis distintas de forma muito concisa.

Objetivo: tornar o código mais limpo, fácil de entender e menos repetitivo.

Antes da introdução do conceito de desestruturação, acessar propriedades específicas de um objeto ou valores de um array poderia resultar em código verboso e difícil de manter.

Com a desestruturação, é possível extrair rapidamente apenas as partes necessárias, reduzindo a complexidade e a quantidade de linhas de código.

Desestruturação de Arrays

A desestruturação de arrays permite **extrair valores de um array e os atribuir a variáveis usando uma sintaxe curta**.

```
const numeros = [10, 20, 30, 40, 50];

const primeiro = numeros[0];
const segundo = numeros[1];
const terceiro = numeros[2];

console.log(primeiro);  // 10
console.log(segundo);   // 20
console.log(terceiro);  // 30
```

Nesse código, é preciso **acessar cada elemento do array individualmente**, o que pode ser verboso, especialmente se o **array** for grande ou se for preciso acessar vários valores.

```
const numeros = [10, 20, 30, 40, 50];

const [primeiro, segundo, terceiro] = numeros;

console.log(primeiro);  // 10
console.log(segundo);   // 20
console.log(terceiro);  // 30
```

Com a desestruturação, é possível extrair diretamente os valores desejados do **array** em uma única linha de código:

Ela torna o código mais conciso e fácil de entender, eliminando a necessidade de acessar manualmente cada índice do **array**.

Sintaxe básica

```
const [var1, var2, var3] = array;
```

1. **Array Original:** começamos com um array que contém valores, por exemplo, [10, 20, 30].

2. **Chaves Colchetes []:** Dentro dos colchetes, serão listadas as variáveis que irão armazenar os valores extraídos.

3. **Atribuição:** O array à direita do sinal de igual = é desestruturado, e seus valores são atribuídos às variáveis correspondentes à esquerda.

```
const numeros = [10, 20, 30];

const [primeiro, segundo, terceiro] = numeros;

console.log(primeiro);  // 10
console.log(segundo);   // 20
console.log(terceiro);  // 30
```

Pulando (ignorando) elementos

É possível pular elementos do array simplesmente deixando **posições vazias** no lado esquerdo.

```
const numeros = [10, 20, 30, 40];

const [primeiro, , terceiro] = numeros;



console.log(primeiro);  // 10
console.log(terceiro);  // 30
```

Capturando o resto dos valores

É possível usar o operador de **resto (...)** para capturar todos os valores restantes do array em uma variável.

```
const numeros = [10, 20, 30, 40, 50];

const [primeiro, segundo, ...restante] = numeros;



console.log(primeiro);  // 10
console.log(segundo);   // 20
console.log(restante);  // [30, 40, 50]
```

Valor padrão

É possível definir **valores padrão** para os elementos desestruturados, caso o array não tenha valor suficiente

```
const numeros = [10];

const [primeiro, segundo = 20] = numeros;


console.log(primeiro);  // 10
console.log(segundo);   // 20 (valor padrão, já que não havia um segundo valor no array)
```

```
const numeros = [10, 13, 7];

const [primeiro, segundo = 20] = numeros;

console.log(primeiro);  // 10
console.log(segundo);   // 13 – se existir o elemento, não se usa o valor padrão
```

Desestruturação de arrays aninhados

Segue uma lógica semelhante à desestruturação de arrays simples.

Permite extrair valores de arrays que estão dentro de outros arrays (arrays aninhados) de forma organizada.

```
const numeros = [1, [2, 3], [4, 5]];

const [um, [dois, tres], [quatro, cinco]] = numeros;

console.log(um);     // 1
console.log(dois);   // 2
console.log(tres);   // 3
console.log(quatro); // 4
console.log(cinco);  // 5
```

Detalhando

1. **Primeiro nível:** **`const [um, [dois, tres], [quatro, cinco]]`**
   - No array **`numeros`**, o primeiro elemento é 1, que é atribuído à variável **`um`**.
2. **Segundo nível:** O segundo elemento do array **`numeros`** é **`[2, 3]`**, que é outro array.
   - Este array é desestruturado em **`dois`** e **`tres`**.
3. **Terceiro nível:** O terceiro elemento é **`[4, 5]`**, que é desestruturado em **`quatro`** e **`cinco`**.

Exemplo com valores padrão

Caso algum dos arrays aninhados esteja ausente ou seja menor do que o esperado, é possível utilizar valores padrão:

```
const numeros = [1, [2], []];

const [um, [dois, tres = 3], [quatro = 4, cinco = 5]] = numeros;

console.log(um);      // 1
console.log(dois);    // 2
console.log(tres);    // 3 (valor padrão)
console.log(quatro); // 4 (valor padrão)
console.log(cinco);  // 5 (valor padrão)
```

Técnica é útil quando você trabalha com estruturas de dados complexas que envolvem arrays aninhados, como resultados de consultas, dados JSON ou quando é necessário manipular grupos de valores de uma só vez.

A desestruturação de arrays aninhados **permite extrair valores de arrays dentro de arrays**, mantendo o código limpo e organizado.

Assim, é possível trabalhar de forma eficiente com dados hierárquicos sem recorrer a acessos repetitivos aos índices.

desestruturação de objetos

Desestruturação de objetos

De forma análoga, a desestruturação de objetos permite que **extrair propriedades específicas de um objeto e atribuí-las a variáveis**.

```
const { property1, property2 } = objectName;
```

```
const pessoa = {
  nome: 'João Simões Lopes Neto',
  idade: 51,
  cidade: 'Pelotas'
};

// Desestruturação
const { nome, idade, cidade } = pessoa;

console.log(nome);  // João Simões Lopes Neto
console.log(idade); // 51
console.log(cidade); // Pelotas
```

O que acontece?

- O JavaScript pega as propriedades **nome**, **idade** e **cidade** do objeto **pessoa**.
- Cria três variáveis chamadas **nome**, **idade** e **cidade**.
- Atribui a elas os valores correspondentes.

Renomeando propriedades

É possível atribuir um **identificador diferente do nome da propriedade**, utilizando a seguinte sintaxe:

```
const { nomePropriedade: novoNomeDeVariavel } = idObjeto;
```

Exemplo:

```
const usuario = { nome: "Alice", idade: 25 };

const { nome: userName, idade: age } = usuario;
```

O que aconteceu?

- A propriedade **nome** do objeto **usuario** é atribuída à variável **userName**.
- A propriedade **age** do objeto **usuario** é atribuída à variável **userAge**.

Resultado:

- **userName** agora é "Alice".
- **userAge** agora é 25.

Utilizando valores padrão

Se uma propriedade **não existir no objeto**, é possível definir um valor padrão para a variável.

```
const { property = defaultValue } = objectName;
```

Exemplo:

```
const user = { name: "João" };

const { name, age = 30 } = user;
```

O que aconteceu?

- O JavaScript verifica se `age` existe no objeto `user`.
- Como `age` não existe, a variável `age` recebe o valor padrão de `30`.

Resultado:

- `name` agora é `"João"`.
- `age` agora é `30`.

Desestruturação Aninhada

Para **desestruturar objetos dentro de outros objetos**, a sintaxe é:

```
const {
  propriedadeExterna: {
    propriedadeInterna
  }
} = nomeDoObjeto;
```

Nesse tipo de desestruturação "navegamos" pelos objetos internos até chegar às propriedades desejadas.

Exemplo:

```
const pet = {
  nome: “Candy",
  idade: 9,
  endereco: {
    rua: “Av. Adolfo Fetter",
    cidade: "Pelotas",
    cep: “96090000"
  }
};

const { endereco: { cidade, cep } } = pet;

console.log(cidade);     // "Pelotas"
console.log(cep);        // "9609000"
```

O que aconteceu?

- **`endereco: {cidade, cep}`** significa que estamos desestruturando o objeto **`endereco`** dentro do objeto **`pet`**.
- **`cidade`** e **`cep`** são extraídos de **`endereco`** e se tornam variáveis.

Resultado:

- Agora temos duas variáveis, **`cidade`** e **`cep`**, que contêm os valores "Pelotas" e "96090000".

Renomeando variáveis na desestruturação aninhada


```
const pet = {
  nome: "Candy",
  idade: 9,
  endereco: {
    rua: "Av. Adolfo Fetter",
    cidade: "Pelotas",
    cep: "96090000",
  },
};
```

O que aconteceu?

- **rua** é renomeado para **street**.
- **cidade** é renomeado para **city**.

Resultado:

- Agora as variáveis **street** e **city** contêm os valores **"Av. Adolfo Fetter"** e **"Pelotas"**.

```
const { endereco: { rua: street, cidade: city } } = pet;

console.log(street);  // "Av. Adolfo Fetter"
console.log(city);    // "Pelotas"
```

Desestruturação aninhada com valores padrão

Se uma propriedade aninhada pode não existir, é possível fornecer **valores padrão**.

Suponha que cep pode não estar presente no objeto endereco.

É possível definir um valor padrão para ele:

```
const pet = {
  nome: "Candy",
  idade: 9,
  endereco: {
    rua: "Av. Adolfo Fetter",
    cidade: "Pelotas",
    //cep: "96090000",
  },
};

const { endereco: { rua, cep = "00000" } } = pet;

console.log(rua);  // "Av. Adolfo Fetter"
console.log(cep);    // “00000"
```

O que aconteceu?

- Se zip não existir no objeto address, ele será definido como "00000"

Resultado:

- Agora as variáveis **`street`** e **`city`** contêm os valores **`"Av. Adolfo Fetter"`** e **`"Pelotas"`**.

A desestruturação em JavaScript, tanto para arrays quanto para objetos, é uma ferramenta poderosa que simplifica e torna mais legível o código, especialmente em projetos complexos.

Ao permitir a extração de dados de estruturas compostas de maneira concisa, ela ajuda a evitar redundâncias e facilita o trabalho com dados estruturados.

Vantagens

Código Mais Limpo e Legível:

- Desestruturar arrays e objetos permite acessar valores diretamente, sem a necessidade de múltiplas linhas de código para acessar propriedades ou índices.
  - Isso melhora a clareza do código, tornando-o mais fácil de entender e manter, tanto para você quanto para outros desenvolvedores que venham a trabalhar no projeto.

Simplificação de Funções:

- Funções que recebem objetos como parâmetros podem usar a desestruturação para extrair apenas as propriedades necessárias, tornando as funções mais claras e reduzindo o acoplamento.

```
function displayUser({ name, age }) {
  console.log(`${name} is ${age} years old.`);
}
```

Trabalho eficiente com dados aninhados:

- Quando lidamos com objetos aninhados, a desestruturação permite acessar rapidamente propriedades internas sem a necessidade de múltiplas referências ao objeto pai.

Valores Padrão:

- A capacidade de definir valores padrão ao desestruturar dados evita erros comuns, como o acesso a propriedades indefinidas, o que pode tornar o código mais robusto e resiliente.

Renomeação de Variáveis:

- A desestruturação permite renomear variáveis de forma simples, o que é especialmente útil ao trabalhar com APIs ou objetos cujas propriedades possam ter nomes pouco descritivos ou conflitantes com variáveis já existentes.

Desestruturação com Rest Operator:

- Tanto em arrays quanto em objetos, o uso do operador ... (rest) facilita a captura do restante dos itens ou propriedades, permitindo operações mais flexíveis e adaptáveis às mudanças nos dados.

Conclusões

A desestruturação é uma técnica essencial em JavaScript moderno, melhorando a **eficiência do desenvolvimento e a qualidade do código**.

Ao utilizar essa técnica, os desenvolvedores podem criar aplicações mais limpas, legíveis e fáceis de manter, além de reduzir significativamente a complexidade ao manipular dados estruturados.

Seja para extrair dados de objetos retornados por APIs, trabalhar com arrays ou simplificar a assinatura de funções, a desestruturação é uma prática recomendada e amplamente adotada na comunidade de desenvolvimento JavaScript.

Curso Superior de Tecnologia em Sistemas para Internet
Desenvolvimento front-end I (front 1)
Rafael Cunha Cardoso
rafaelcardoso@ifsul.edu.br

Desestruturação

JavaScript