DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Front End II

Boas práticas

Em nossa vida profissional como desenvolvedores, nem sempre estaremos escrevendo código. Em muitas ocasiões, teremos que ler códigos escritos por outros desenvolvedores e até por nós mesmos.

Vamos pensar, por exemplo, que desenvolvemos uma funcionalidade e, alguns meses após seu lançamento, o departamento de UX decide que devemos fazer uma mudança nela. Como fomos nós que trabalhamos em sua criação, desta vez nos é atribuída a tarefa de fazer os ajustes necessários. Você pode imaginar como seria encontrar um bloco de código escrito pelo seu "eu" do passado?

Situações como essa acontecem com muita frequência. Se trabalhamos em uma equipe de desenvolvimento, será comum realizarmos “code review” (revisão de código) de alguma funcionalidade implementada por um de nossos colegas. Em determinados momentos, podemos passar mais tempo revisando o código do que escrevendo.

Diante dessas situações, poder contar com um código claro e legível será de extrema importância para podermos entender o que o código que estamos lendo diz e faz. Isso facilitará nosso processo de revisão e também nos permitirá ter uma ideia de como nosso aplicativo funcionará, antes mesmo de testá-lo.

Nesse contexto, desta vez vamos nos concentrar em comentar alguns princípios básicos a serem lembrados ao escrever código. Não se preocupe, você não precisa decorá-los. Em geral, é algo que aprendemos à medida que interagimos com o código, portanto, quanto mais código escrevermos e revisarmos, mais nos acostumaremos com esses princípios. De qualquer forma, queremos contar rapidamente sobre cada um deles.

Princípio de responsabilidade única

Esse princípio indica que cada classe, função ou módulo deve ser responsável por uma tarefa específica, e essa tarefa deve ser encapsulada nessa classe, função ou módulo.

Considere, por exemplo, a seguinte função:

```
function calcularMedia(numero1, numero 2) {
    const soma = numero1 + numero2;
    const media = soma / 2;
    return media;
}
```

Como podemos ver, esta função está realizando duas tarefas:

1. Somar dois números.
2. Dividir o resultado por 2.

Se pensarmos no princípio da responsabilidade única, poderíamos extrair a soma para uma função separada, que é exclusivamente responsável por realizar essa tarefa e retornar o resultado. A função **calcularMedia** apenas se encarrega de pegar o resultado e dividir por dois. Então, como poderíamos refatorar nosso código? Deixamos a resposta:

```
function somar(numero1, numero2) {
    return numero1 + numero2;
}

function calcularMedia(numero1, numero2) {
    const soma = somar(numero1, numero2);
    return soma / 2;
}
```

Como podemos ver neste caso, cada função é responsável por uma única tarefa, e juntas contribuem para o resultado esperado. Este exemplo pode parecer muito simples, mas quando nos deparamos com funcionalidades mais complexas, este princípio nos ajudará a distribuir melhor as tarefas entre os diferentes blocos do nosso código, dando-nos a possibilidade de ter um código mais legível e estruturado.

Keep it simple, stupid (KISS)

Este princípio é tão simples quanto o nome indica. Basicamente, a ideia é que qualquer sistema ou programa funciona melhor se for mantido simples em vez de complexo. Podemos evitar adicionar qualquer camada de complexidade que não seja estritamente necessária para o bom funcionamento do sistema.

Vejamos o seguinte exemplo:

```
if (nome === null || nome === undefined || nome === '') {
    console.log("Nome inválido!");
} else {
    console.log(nome);
}
```

```
const usuario = nome ? nome : "Nome inválido!";
console.log(usuario );
```

No caso acima, temos duas maneiras de realizar a mesma operação. Embora seja verdade que o primeiro (**if/else**) seja o mais declarativo, ao usá-lo estamos adicionando complexidade desnecessária ao nosso sistema, que está realizando uma tarefa simples. Portanto, se formos guiados pelo princípio KISS, a segunda opção seria a alternativa a escolher. Se quiséssemos dar um passo adiante, poderíamos simplificar ainda mais a operação anterior da seguinte forma:

```
const usuario = nome || "Nome inválido!";
console.log(usuario);
```

Olhando para este bloco de código, você pode estar se perguntando: por que estamos usando o operador **OR** (**||**) se não estamos avaliando uma condição, mas atribuindo um valor a uma variável? Esta sintaxe funciona como um "atalho" para atribuir um valor condicionalmente. Por fim, o JavaScript avaliará o valor de "nome". Se o mesmo é um valor de tipo **truthy** (verdadeiro), esse valor será atribuído à variável "usuario". Caso contrário, será atribuído o que estiver à direita do **OR** (neste caso, "Nome inválido!").

Don’t repeat yourself (DRY)

Este princípio nos convida a pensar em nosso código para evitar repetições desnecessárias. Para isso, devemos pensar de forma abstrata, com especial ênfase nas funções que ele deve cumprir independentemente de um caso específico. Além disso, na medida do possível, devemos considerar a padronização das informações, para evitar redundâncias.

Vejamos um exemplo:

```
const clientes = [‘João’, ‘Pedro’, ‘Marcelo’];
const fornecedores = [Paulo, Luiz, José];

console.log(clientes[0]);
console.log(clientes[1]);
console.log(clientes[2]);

console.log(fornecedores[0]);
console.log(fornecedores[1]);
console.log(fornecedores[2]);
```

Nesse caso, podemos ver que existem várias linhas de código que realizam a mesma tarefa (exibe um nome no console). Da mesma forma, vemos que em cada linha estamos acessando um elemento dentro de um array, que é basicamente o que queremos exibir.

Se pensarmos nisso de forma abstrata (sem prestar atenção nos dados concretos), podemos ver que cada linha de código realiza a mesma operação: acessar um elemento dentro de um array e exibi-lo no console. Dessa forma, podemos repensar nosso código da seguinte forma:

```
function logItens(array) {
    array.forEach((item) => console.log(item));
}

logItens(clientes);
logItens(fornecedores);
```

Como podemos ver, desta vez abstraímos a lógica do nosso código em uma única função responsável por realizar as tarefas repetidas a cada oportunidade. Dessa forma, podemos reutilizar essa função toda vez que precisarmos percorrer um array e exibir cada um de seus elementos, sem precisar repetir o mesmo código todas as vezes.

Até agora, apresentamos alguns dos princípios e práticas mais relevantes recomendadas para escrever código legível que seus colegas e, acima de tudo, você possa entender agora e no futuro.

Convidamos você a colocar esses princípios em prática ao realizar qualquer atividade a partir de agora. A praticar!