Certified Tech Developer
The Ultimate Degree

Buenas prácticas

En nuestra vida profesional como desarrolladores, no siempre estaremos escribiendo código. En muchas ocasiones, nos tocará leer código escrito por otros desarrolladores e inclusive, por nosotros mismos.

Pensemos por ejemplo, que hemos desarrollado una funcionalidad y, un par de meses después de su lanzamiento, el departamento de UX decide que debemos realizar un cambio en la misma. Ya que hemos sido nosotros quienes hemos trabajado en su creación, en esta oportunidad se nos asigna la tarea de realizar los ajustes necesarios. ¿Se imaginan cómo sería encontrarse con un bloque de código escrito por nuestros "yo" del pasado?.

Situaciones como esta, se dan muy a menudo. Si trabajamos en un equipo de desarrollo, será habitual que debamos realizar "code review" (revisión de código) de alguna funcionalidad implementada por alguno de nuestros compañeros. En ciertos momentos, puede que pasemos más tiempo revisando código que escribiendo.

Frente a estas situaciones, poder contar con un código claro y legible, será de suma importancia para que podamos entender que dice y hace el código que estamos leyendo. Esto facilitará nuestra tarea de revisión y, además, nos permitirá hacernos una idea de cómo funcionará nuestra aplicación, inclusive antes de probarla.

En ese contexto, en esta oportunidad nos enfocaremos en comentarte algunos principios básicos a tener en cuenta al momento de escribir código. No te preocupes, no tienes que aprenderlos de memoria; en general, es algo que vamos aprendiendo a

medida que interactuamos con el código por lo que, mientras más código escribamos y revisemos, más iremos habituandonos a dichos principios. De todas maneras, queremos contarte rápidamente de que se trata cada uno de ellos. Veamos...

Principio de responsabilidad única

Como su nombre lo indica, este principio indica que cada clase, función o módulo debe ser responsable de una tarea específica, y dicha tarea debe estar encapsulada dentro de dicha clase, función o módulo.

Pensemos por ejemplo en la siguiente función:

```
function calcularPromedio(numero1, numero2) {
  const suma = numero1 + numero2;

  const promedio = suma / 2;

  return promedio;
}
```

Como puede verse, dicha función esta realizando dos tareas: 1) Sumar dos números y 2) Dividir el resultado por 2.

Si pensamos en el principio de responsabilidad única, podríamos pensar en extraer la suma a una función aparte, que se encargue exclusivamente de realizar dicha tarea y devolver el resultado. A su vez, la función calcularPromedio únicamente se ocuparía de tomar el resultado y dividirlo por dos. Entonces, ¿cómo podríamos refactorizar nuestro código?. Aquí te dejamos la respuesta:

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

function calcularPromedio(numero1, numero2) {
  const suma = sumar(numero1, numero2);

  return suma / 2;
}
```

Como puedes ver, en este caso cada función se encarga de una única tarea, y entre todas contribuyen al resultado final esperado. Quizás este ejemplo parezca muy simple, pero cuando nos enfrentemos a funcionalidades más complejas, este principio nos ayudará a distribuir mejor las tareas entre los distintos bloques de nuestro código, dándonos la posibilidad de tener un código más legible y estructurado.

Keep it simple Stupid! (KisS)

Este principio, que en español podemos traducir como “mantenlo simple, estúpido!” es tan simple como su nombre lo indica 😉. Básicamente, la idea detrás del mismo es que cualquier sistema o programa, funciona mejor si se mantiene simple que si se hace complejo. Por ello, la idea es evitar agregar cualquier capa de complejidad que no sea estrictamente necesaria para el correcto funcionamiento del sistema.

Veamos el siguiente ejemplo:

```
if (nombre === null || nombre === undefined || nombre === '') {
  console.log('NOMBRE INVÁLIDO!');
} else {
  console.log(nombre);
}
```

```
const detalleUsuario = nombre ? nombre : 'NOMBRE INVÁLIDO!';
console.log(detalleUsuario);
```

En el caso anterior, tenemos dos formas de realizar la misma operación. Si bien es cierto que la primera de ellas (if/else) es la más declarativa, al utilizarla estamos agregando una complejidad innecesaria a nuestro sistema, que está realizando una tarea simple. Por ello, si nos guíamos por el principio KIsS, en este caso la segunda opción sería la alternativa a escoger.

Inclusive, si quisiéramos ir un paso más allá, podríamos simplificar más aún la operación anterior de la siguiente manera:

```
const detalleUsuario = nombre || 'NOMBRE INVÁLIDO!';
console.log(detalleUsuario);
```

Puede que al ver este bloque de código te preguntes, ¿por qué estamos usando el operador OR (||) si no estamos evaluando una condición sino asignando un valor a una variable?. Bueno, esta sintaxis actúa como un “atajo” para asignar un valor de forma condicional. En definitiva, Javascript evaluará el valor de “nombre”. Si el mismo es un valor de tipo *truthy* (verdadero), se asignará dicho valor a la variable

"detalleUsuario". Caso contrario, se asignará lo que se encuentre a la derecha del OR (en este caso, "NOMBRE INVALIDO").

Don't repeat yourself (DRY).

Este principio, nos invita a pensar nuestro código de forma tal de evitar repeticiones innecesarias de código. Para ello, debemos pensar nuestro código de manera abstracta, con especial énfasis en las funciones que el mismo debe cumplir independientemente de un caso concreto. Además, en la medida de lo posible, debemos tener en cuenta la normalización de la información, de manera de evitar redundancia.

Veamos un ejemplo:

```
const clientes = ['Juan', 'Pedro', 'Marcelo'];
const proveedores = ['Esteban', 'Luis', 'José'];

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

console.log(proveedores[0])
console.log(proveedores[1])
console.log(proveedores[2])
```

En este caso, podemos ver que existen varias líneas de código que realizan la misma tarea (mostrar un nombre en consola). De la misma manera, vemos que en cada línea estamos accediendo a un elemento dentro de un array, que es en definitiva lo que deseamos mostrar.

Si pensamos esto de manera abstracta (sin prestar atención al dato concreto), podemos ver que cada linea de código realiza la misma operación: acceder a un elemento dentro de un array y mostrarlo en la consola. De esta manera, podemos repensar nuestro código de la siguiente manera:

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

logItems(clientes);
logItems(proveedores);
```

Como podemos observar, en esta oportunidad hemos abstraído la lógica de nuestro código en una única función que se ocupa de realizar las tareas que se repiten en cada oportunidad. De esta manera, podemos reutilizar dicha función cada vez que necesitemos recorrer un array y mostrar cada uno de sus elementos, sin necesidad de repetir el mismo código todo el tiempo.

Hasta aquí, te hemos presentado algunos de los principios y buenas prácticas más relevantes para escribir un código legible que tus compañeros y, sobre todo tu puedan entender en el presente y en el futuro. Te invitamos a poner en práctica dichos principios al momento de realizar cualquier actividad de aquí en adelante. ¡A practicar!.