**Administración del desarrollo de servicios web (ADSAW)**

**Guía de programación**

|  |  |
| --- | --- |
| Responsables: Christopher Rojano Jimenez |  |
| Cliente: |  |

Copyright © 2023

# Estándares de Programación Html y CSS

## Separación semántica y de presentación

Separa la semántica del elemento del efecto del elemento en la representación de su contenido

## Principio de selección de elementos

### Menos es más

La etiqueta solo debe usarse cuando el contenido requiera semántica consejos: pregúntese cómo piensa desempeñar el papel semántico de un elemento. Si no puede responder, no utilice este elemento.

### No uses mal los elementos

Solo use elementos para su propósito original, no cree su propia semántica. Si no puede encontrar uno adecuado, puede considerar elementos generales (como span o div) y usar la clase de atributo global para indicar su significado.

### Específico es mejor, siempre consistente

El uso del mismo elemento debe ser coherente en toda la página.

## Codificación

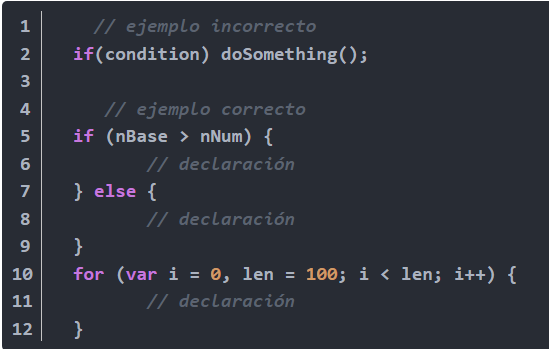
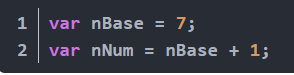
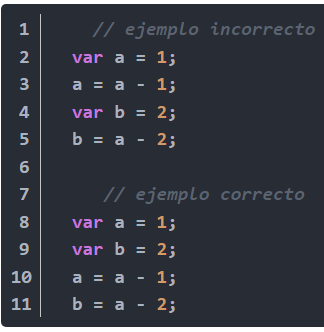
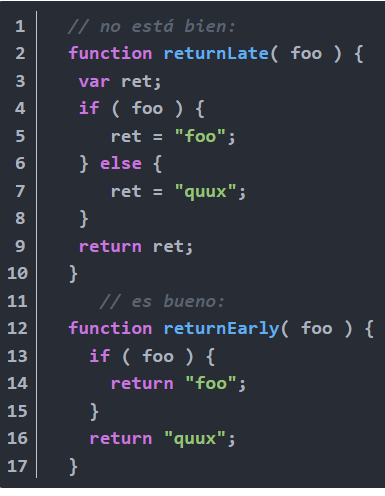
Utilice UTF-8

# Estándares de Programación JavaScript

## Sangría

-Nunca mezcle espacios y pestañas.

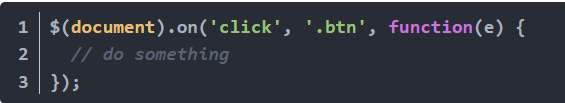
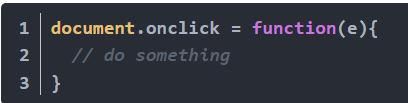
## Especificación de formato js

* + 1. Operador Antes y después Para mantener 1 espacio para mejorar la legibilidad del código, agregue un punto y coma al final de cada línea de código.
    2. Utilice paréntesis y espacios para mejorar la legibilidad
    3. Las declaraciones de juicio como if / for / try generalmente tienen paréntesis, llaves y varias líneas, y se recomienda ajustar
    4. En el mismo método o cierre, la definición de variable se unifica en la parte superior
    5. Las comillas simples (') son mejores que las comillas dobles ("), especialmente cuando crea una cadena que contiene código HTML, así que use las comillas de manera uniforme en su proyecto y manténgase consistentes.
    6. Devuelve los resultados de la función por adelantado para mejorar la legibilidad del código

## 

## 

## Vinculación de eventos

* + 1. Está prohibido vincular eventos directamente con onxxx, como:
    2. En proyectos que usan jQuery, se recomienda usar on () para implementar el enlace de eventos

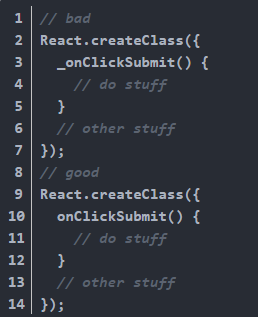
## 

# Estándares de Programación React / JSX y React Native

* 1. **Especificaciones básicas**
* Cada archivo tiene un Component de React, y debe ser el que indica el nombre.
  1. **Conversión de nomenclatura**
* nombre del archivo:

Los nombres de carpetas y archivos deben usar nombres de mayúsculas y minúsculas, como HomeIndex.js;

Los componentes comunes se colocan en la carpeta Component. El archivo de entrada debe ser claro, generalmente se puede usar index.js.

* 1. **Propiedades**
* Las propiedades del componente utilizan la nomenclatura de la pequeña caja camel.
* En React, className reemplaza el atributo de clase y htmlFor reemplaza el atributo for.
* Cuando se utilizan estilos fuera de línea, el nombre del atributo debe tener preferiblemente la palabra clave Style. Por ejemplo: flexStyle.js
  1. **Uso de comillas**
* Los atributos HTML / JSX utilizan comillas dobles ";
* JS usa comillas simples ';
  1. **Organización del código interno del Component**
* No use el prefijo de subrayado para nombrar los métodos del componente React;

# Estándares para comentar apropiadamente

## 4.1. Tipos de comentarios:

### 4.1.1. Una sola línea

Los comentarios de una sola línea se utilizan generalmente para comentar una parte de la línea o una línea completa de código.

En JavaScript: //

En HTML: <!-- comentario →

En CSS: //

En react: {//}

En python: #

### 4.1.2. Múltiples líneas

Si quisiéramos agregar una nota que se distribuye en varias líneas, podemos optar por comentarios de varias líneas o comentarios a nivel de bloque.

En JavaScript: /\* \*/

En HTML: <!-- comentario →

En CSS: /\* \* /

En react: {/\* \*/}

En python: #

## 4.2. Buenas prácticas

**4.2.1. Los comentarios tienen que ser útiles:**

No hace falta explicar cosas obvias. El código fuente en el 98% de los casos lo va a leer otro programador. O al menos alguien con conocimientos de programación. Por ejemplo:

// Comprobamos si el contador es 5

if (contador == 5)

Ese comentario no aporta nada. Mejor si lo eliminamos. Mejor todavía si no llegamos ni a escribirlo.

**4.2.2. Piensa si un comentario es necesario antes de añadirlo**

Si tenemos que añadir un comentario es mejor pensarlo un poco antes de hacerlo. ¿Por qué voy a añadir este comentario? ¿Es realmente necesario? ¿Puedo refactorizar mi código para evitarlo? Quizá el código que hemos escrito es demasiado complejo y por eso necesita ser explicado. En ese caso podremos separarlo en métodos más sencillos.

**4.2.3. Evita comentarios dentro de métodos o funciones**

Puede ser útil comentar lo que hace un método antes de su declaración. Incluso podemos comentar sus parámetros. Pero hay que tener en cuenta que debemos comentar el "qué" y no el "cómo". Es decir, debemos explicar qué hace nuestro método. El "cómo" hace el método su función ya va explicado en el código.

**4.2.4. No comentes código eliminado**

A veces nos vemos obligados a eliminar parte del código. Y a veces, en un arranque de "por si acaso", comentamos el código eliminado, para no perderlo. Esto es algo que no tiene sentido en el siglo XXI. Ya deberíamos tener un sistema de control de código fuente para estas cosas. Y si no lo tenemos, tenemos problemas más graves que los comentarios.

**4.2.5. Sigue siempre un mismo estilo**

Es recomendable comentar el código siempre de la misma manera. Un estilo definido ayuda a comprender el código a quién lo está leyendo. Si siempre comentamos un método con una introducción y una descripción de los parámetros, tendremos que ser constantes y hacerlo en todos los métodos. Si no la persona que lo lea puede acabar confundido por la disparidad de criterios.

**4.2.6. Usar comentarios para depurar**

Además de tomar notas, los comentarios también se pueden utilizar para evitar rápidamente la ejecución de código con fines de depuración. Esto es posible porque los motores JavaScript no interpretan el código comentado. Esto se llama como comentando el código.

Si hay una línea errónea que está causando problemas, simplemente podemos «comentarla» para deshabilitarla, sin borrar la línea. Esto se puede combinar con depuradores reales para ayudarlo a evaluar lo que está sucediendo.

# Buenas prácticas al trabajar con Git

## 5.1. Sigue el GitFlow Workflow.

El GitFlow Workflow es una metodología de trabajo basada en el división de las distintas etapas de producción de software en distintas ramas del repositorio:

**master**: En la rama máster se encuentran las releases estables de nuestro software. Esta es la rama que un usuario típico se descargará para usar nuestro software, por lo que todo lo que hay en esta rama debería ser funcional. Sin embargo, puede que las últimas mejoras introducidas en el software no estén disponibles todavía en esta rama.

**dev**: En esta rama surge de la última release de master. En ella se van integrando todas las nuevas características hasta la siguiente release.

**feature-X**: Cada nueva mejora o característica que vayamos a introducir en nuestro software tendrá una rama que contendrá su desarrollo. Las ramas de feature salen de la rama dev y una vez completado el desarrollo de la mejora, se vuelven a integrar en dev.

**hotfix-X**: Si nuestro código contiene bugs críticos que es necesario parchear de manera inmediata, es posible crear una rama hotfix a partir de la publicación correspondiente en la rama master. Esta rama contendrá únicamente los cambios que haya que realizar para parchear el bug. Una vez arreglado, se integrará en master, con su etiqueta de versión correspondiente y en dev.

## 5.2. Prueba tus cambios antes de hacer commit.

Siempre es bueno probar tus cambios antes de enviarlos a tu repositorio local. Las pruebas son esencialmente una capa adicional de seguridad antes de comprometerse. Te permite identificar errores y fallas más rápido antes de que vayan a su servidor de producción.

## 5.3. Realiza commit frecuentemente

Un commit temprano ayuda a reducir el riesgo de conflictos entre dos cambios simultáneos. Además, tener puntos de control periódicos significa que puedes comprender cuando algo dejó de funcionar. Puedes hacer commit localmente “todo el tiempo” y push cuando todo esté funcionando.

## 5.4. No integres dependencias

Tu repositorio en Git debe ser exclusivamente para administrar tu código fuente, no para almacenar las dependencias. Además, la presencia de estas aumentará significativamente el tamaño del repositorio del proyecto.

Nota: Se sancionará al que suba la carpeta *node\_modules* al repositorio.

## 5.5. Crea un archivo .gitignore significativo

Un archivo .gitignore es imprescindible en cada repositorio para ignorar archivos y directorios predefinidos. Te ayudará a evitar dependencias y muchas otras posibles discrepancias en tu código. Recuerda que algunos archivos no se pueden ignorar por lo que pon atención al siguiente punto.

## 5.6. Añade solo los archivos en los que estuviste trabajando

Usar git add . definitivamente no es la mejor práctica, lo mejor en estos casos es añadir solo los archivos de los que estuviste trabajando, esto evita que se ingresen muchos archivos “trash” que pueden traer una repercusión grave en la dimensión del repositorio.

Añadir los archivos individualmente:

git add src/Carrito/component/CarritoComponent.js

Nota: Se sancionará al que suba archivos basura (como eslintcache o py.cache)

## 5.7. Sobre los commits.

### 5.7.1. Usa el verbo imperativo (Add, Change, Fix, Remove, …)

Aunque el mensaje puede sonar un poco borde, el verbo presente es una forma de expresar la acción que se realiza en el commit. Por ejemplo, Add significa que se añade un nuevo archivo, Change significa que se modifica un archivo existente y Fix significa que se arregla un bug.

### 5.7.2. No uses punto final ni puntos suspensivos en tus mensajes

Usar puntuación, más allá de las comas, es innecesario a la hora de crear un buen mensaje de commit. Cada carácter cuenta a la hora de crear un buen mensaje de commit así que no lo desperdicies con puntos innecesarios.

### 5.7.3. Usa como máximo 80 carácteres para tu mensaje de commit

Sé corto y conciso. Si tienes mucho qué explicar… seguramente es que tu commit hace demasiadas cosas. ¿Puedes separarlo en diferentes commits? Pues hazlo.

Haz que el mensaje sea claro, directo y que realmente refleje los cambios que lleva.

### 5.7.4. Añade todo el contexto que sea necesario en el cuerpo del mensaje de commit

A veces necesitas proveer de más contexto a tu commit. Para ello, en lugar de saturar el sumario del commit, añade información que sea necesaria en el cuerpo del mensaje.

Puedes lograrlo usando

git commit -m "Cosas que agregue" -m "Dando contexto"

pero en estos casos lo mejor es que uses directamente

git commit

## 5.8. Crea pull request adecuadamente.

Al finalizar los cambios realizados, es importante no solo hacer push de tu rama, si no también crear un pull a la rama de dev, esto para ver reflejados los cambios.

Este proceso se puede realizar mediante comandos, sin embargo se le permitirá al equipo de trabajo utilizar la página de git para realizarlo.

Un pull request bien estructurado considera lo siguiente:

La base SIEMPRE debe ser dev y después colocar la rama de tus cambios.

En la parte de “reviewer” recuerda poner el usuario del revisor

Espera a que tus cambios sean aprobados entonces, Y SOLO ENTONCES, manda tus cambios a dev.

## 5.9. Baja los cambios apropiadamente.

Si alguien más realizó cambios que se aprobaron antes de los tuyos, y tu rama se creó a la par de la suya, es importante que bajes los cambios realizados desde dev a tu rama. Esto evitará que cuando quieras cubrir tus cambios, haya algún conflicto entre ramas.

Por último, recuerda que los cambios que bajes solo pueden ser de una rama de otro integrante (en casos muy específicos) o desde la rama de desarrollo de dev.

Nota: Se sancionará a quien baje cambios desde master