HTML5

- Tags, Best Practice y Validators -

Turno: Mañana
ADA ITW
- Grupo N°6 -

Integrantes:

- ★ Beccan, Constanza.
- ★ Busatto, María Florencia.
- ★ Aponte, Elifer.
- ★ Garcete, Erica.
- ★ Yusti, Zoe.

Introducción:

El presente trabajo pretende afianzar los conocimientos sobre HTML, obtenidos durante la cursada en ADA ITW, poniendo especial énfasis en las nuevas etiquetas incorporadas en la 5ta versión y sin dejar de lado el apartado de las buenas prácticas que debe observar todo *developer* que se precie de tal y el uso correcto de validators que nos permita trabajar con un código “limpio y depurado de errores”.

A lo largo de los distintos ejes temáticos de nuestro trabajo, el apartado de FAQ y nuestra presentación dinámica esperamos ayudar a reforzar conceptos de una forma amena y respetando un vocabulario técnico que colabore en la incorporación de dichos conceptos a nuestra tarea diaria.

¡Esperamos que lo disfruten!

Las autoras.

¿Qué es HTML?

Haciendo referencia a sus siglas en inglés***, el HyperText Markup Language*** (lenguaje de marcas de hipertexto) es una forma de embeber texto plano. Este nació en 1980 a partir de una propuesta de un nuevo sistema de *"hipertexto"* para compartir documentos. Antiguamente, este sistema se utilizaba en el ámbito informático para la visualización de documentos electrónicos. De cierta manera, los primitivos sistemas de *"hipertexto"* podrían asimilarse a los enlaces de las páginas web actuales.
El primer documento formal con la descripción de HTML se publicó en 1991 bajo el nombre ***HTML Tags*** *(Etiquetas HTML*) y todavía hoy puede ser consultado online a modo de *reliquia informática* en el siguiente link : **https://tinyurl.com/oldhtmltags**

¿Cómo funciona el HTML?

Todo HTML se basa en un único concepto, las etiquetas. Estas son una forma de hablarle al navegador. Al crear un archivo con etiquetas e información, básicamente le estamos dando instrucciones al navegador de como mostrar dicha información. Con esto, si creamos una etiqueta diciendo “esto es un título” el navegador interpretará la información dentro de la etiqueta y mostrará la información en forma de título.

Etiquetas agregadas para HTML5

Secciones:

<section>	Define una sección en un documento.
<nav>	Define una sección que solamente contiene enlaces de navegación.
<article>	Define contenido autónomo que podría existir independientemente del resto del contenido.
<aside>	Define algunos contenidos vagamente relacionados con el resto del contenido de la página. Si es removido, el contenido restante seguirá teniendo sentido.

<header>	Define la cabecera de una página o sección. Usualmente contiene un logotipo, el título del sitio Web y una tabla de navegación de contenidos.
<footer>	Define el pie de una página o sección. Usualmente contiene un mensaje de derechos de autoría, algunos enlaces a información legal o direcciones para dar información de retroalimentación.
<address>	Define una sección que contiene información de contacto.
<main>	Define el contenido principal o importante en el documento. Solamente existe un elemento <main> en el documento.

Agrupación de contenido:

<figure>	Representa una figura ilustrada como parte del documento.
<figcaption>	Representa la leyenda de una figura.

Semántica a nivel de texto:

<data>	Asocia un *equivalente legible por máquina* a sus contenidos. (Este elemento está sólamente en la versión de la WHATWG del estándar HTML, y no en la versión de la W3C de HTML5).
<time>	Representa un valor de *fecha* y *hora*; el equivalente legible por máquina puede ser representado en el atributo `datetime`.
<mark>	Representa texto resaltado con propósitos de *referencia*, es decir por su relevancia en otro contexto.

<ruby>	Representa contenidos a ser marcados con *anotaciones ruby*, recorridos cortos de texto presentados junto al texto. Estos son utilizados con regularidad en conjunto a lenguajes de Asia del Este, donde las anotaciones actúan como una guía para la pronunciación, como el *furigana* Japonés.
<rt>	Representa el *texto de una anotación ruby* .
<rp>	Representa los *paréntesis* alrededor de una anotación ruby, usada para mostrar la anotación de manera alterna por los navegadores que no soporten despliegue estandar para las anotaciones.
<bdi>	Representa un texto que debe ser *aislado* de sus alrededores para el formateado bidireccional del texto. Permite incrustar un fragmento de texto con una direccionalidad diferente o desconocida.
<wbr>	Representa una *oportunidad de salto de línea,* es decir, un punto sugerido de envoltura donde el texto de múltiples líneas puede ser dividido para mejorar su legibilidad.

Contenido Incrustado:

<embed>	Representa un *punto de integración* para una aplicación o contenido interactivo externo que por lo general no es HTML.
<svg>	Define una *imagen vectorial* embebida.
<math>	Define una *fórmula matemática*.

Formularios:

<datalist>	Representa un *conjunto de opciones predefinidas* para otros controles.

<keygen>	Representa un control de *par generador de llaves.*
<output>	Representa el *resultado de un cálculo.*
<progress>	Representa el *progreso de finalización* de una tarea.
<meter>	Representa la *medida* escalar (o el valor fraccionario) dentro de un rango conocido.

Elementos Interactivos :

<details>	Representa un *widget* desde el que un usuario puede obtener información o controles adicionales.
<summary>	Representa un *resumen, título* o *leyenda* para un elemento <details> dado.
<command>	Representa un *comando* que un usuario puede invocar.
<menu>	Representa una *lista de comandos* .

¿De qué hablamos cuando nos referimos a Meta Tags?

Las **Meta Etiquetas** o **Meta Tags** son etiquetas de información que se añaden en el código html de cada página de una Web para aportar información relevante sobre la categorización de esa página web. Es una información que sólo podrá accederse a través del código fuente y que en realidad sólo resulta relevante para los buscadores, pero su finalidad es de gran utilidad ya que aportan gran información sobre la página como el autor, fecha, palabras clave, descripción, etc.

La calidad de una **meta tag** tiene dos aspectos importantes:

1. Como una persona evalúa la **meta tag**

2. Como un robot evalúa la **meta tag.**

Para una persona, la **meta tag** necesita ser llamativa, interesante, informativa, curiosa y con un toque de "*call-for-action"* (instrucción que provoca una reacción inmediata).

Para un robot, la **meta tag** necesita de una dosis de palabras clave para que el algoritmo clasifique de que trata la página web.

Lo esencial en el desarrollo de la **meta tags** es encontrar un término medio entre estos dos puntos.

La Meta Etiqueta Title: Estrictamente hablando, este no es ningún meta-tag, sino una etiqueta autónoma de HTML, aunque, debido a su significado a la hora de interactuar con los agentes de usuario es común que sea mencionada como parte de los metadatos. Actúa como título de la página en cuestión y debe estar formada por palabras clave presentes en la página web.

Codificación de caracteres: Si la fuente no fue previamente definida en el header del archivo HTTP, es necesario hacerlo usando HTML. Así se evita, por ejemplo, que la ñ o las tildes no se muestren correctamente.

La Meta Etiqueta Description: La etiqueta descripción o Meta Etiqueta Description actúa como una descripción de la página en cuestión y también debe estar formada por palabras clave y frases que resuman el contenido de la página web.

Esta información se muestra como snippet (una síntesis en dos líneas del tema de una página que aparece bajo la URL) en los buscadores de uso más generalizado como Google o Bing, por lo que se recomienda cuidar su redacción.

Palabras clave (Keywords): Con esta etiqueta meta los administradores tienen la posibilidad de definir palabras clave para el buscador. Las keywords son aquellos criterios a los que responde un buscador para ofrecerle al usuario páginas HTML como respuesta, donde tales palabras clave son parte de los meta tags.

Autor (author) y copyright: Estos dos meta tags, de uso opcional desde el punto legal, permiten hacer referencia al diseñador de una página web y al propietario de los derechos del código fuente de una página HTML.

VIEWPORT

Básicamente, sirve para definir qué área de pantalla está disponible al renderizar un documento, cuál es nivel de escalado que puede realizar el usuario, así como si el

navegador debe mostrarla con algún *zoom* inicial. Todo ello se indica a través de varios parámetros en la propia etiqueta META.

Lista completa de propiedades de viewport:

Atributo	Valores	Descripción
width	Valor integral (en pixeles) o constante device-width	Define el ancho del viewport.
height	Valor integral (en píxeles) o constante device-height	Define el alto del viewport.
initial-scale	Cualquier número real de 0.1 en adelante. 1 representa no escala.	Escala inicial del viewport.
minimum-scale	Cualquier número real de 0.1 en adelante. 1 representa no escala.	Escala máxima del viewport.
maximum-scale	Cualquier número real de 0.1 en adelante. 1 representa no escala.	Escala mínima del viewport.
user-scale	"yes" / "no"	Permiso para que el usuario pueda hacer zoom.

¿Cuales son las buenas prácticas en un Programador?

★ **Nunca perder el foco del proyecto:** para esto podemos hacer un documento, un diagrama de flujo, mock ups y todas las herramientas que nos ayudan a esquematizarlo, para que durante el desarrollo no perdamos el objetivo o los elementos principales del mismo.

★ **Usar comentarios:** todo programa debe ser previamente comentado, explicando el propósito, funcionamiento completo y el resultado esperado. Adicionalmente a los comentarios generales, siempre comenta antes de cada conjunto importante de código. **¿Qué evitar?** el código ***commented-out*****,** que corresponde al código comentado para que no se ejecute/no compile, ya que la lectura del código se vuelve engorrosa.

★ **Keep it simple:** escribe tus programas lo más simple y directo posible.

★ **Uso de compiladores:** si trabajas con un compilador, ajusta sus opciones para que arroje la mayor cantidad de errores y advertencias posibles al compilar, así tu aplicación tendrá menores chances de obtener errores aleatorios.

★ **Identación:** dentro de las funciones definidas, establece un **espaciado o indentación**, que resalte la estructura funcional de la aplicación y facilite la lectura al programador al que le corresponda analizar el código:

1. **Por lo general, se usa un nivel de indentación por cada bloque de código** (sentencias condicionales y bucles son consideradas como bloques de código embebido dentro de otro, por lo que se recomienda la indentación), ésta indentación corresponde a una sangría que comúnmente tiene el valor de una tabulación (tecla Tab) o bien tres o cuatro espacios.
2. **Es importante que el tamaño de las sangrías sean regulares (consistentes) y no varíen a lo largo del código**, es decir, si el primer bloque ocupa como indentación una tabulación, el resto de bloques deben ser indentados con una tabulación adicional por cada nivel, con eso se facilita la lectura en cualquier editor de código.

★ **Variables:** se recomienda declarar variables en líneas separadas, ya que se facilita la descripción de cada variable mediante comentarios:

1. **No uses variables cuyo nombre no posea significado descriptivo**, una variable con nombres significativos permite al lector entender el contexto del código y permite disminuir la cantidad de documentación asociada, puesto que con un código legible y nombres significativos, el código se ve *auto documentado*. Por ejemplo, una variable llamada cantidad_recursos, tiene más significado que una variable llamada c.
2. Es **altamente recomendada** la definición de variables locales al inicio de la implementación de cada función, como un bloque de código bien separado del bloque que contenga las instrucciones ejecutables, ésta separación puede consistir en una línea en blanco, o bien un comentario que denote la utilidad de cada bloque.

★ **Sobre las comas:** poner un espacio después de cada coma(,) facilita la legibilidad del código.

★ **Elección de nombres:** se consistente al usar un estándar para nombres largos, puedes usar el estándar usado en C ("nombre_largo"), o bien el de Java, llamado CamelCase("nombre","VariableNombreLargo","Clase","ClaseNombreLargo").

★ **Comentarios en las Funciones:**

1. Comenta cuando sea justo y necesario, usa los comentarios dentro de las funciones para describir las variables (sólo cuando su utilidad sea potencialmente dudosa) y cuando existan bloques de código difíciles de entender a primera vista; el exceso de comentarios vuelve ilegible el código.
2. Se recomienda añadir al inicio de cada función, un bloque de comentarios que expliquen el comportamiento general de la función, de modo que pueda entenderse a grosso modo que es lo que hace, o se espera que haga, así se facilita la búsqueda de errores, y se evita el análisis innecesario en una gran cantidad de casos.

★ **Operadores:**

1. En caso de usar operadores binarios (por ejemplo +, -, &&, ||, entre otros) se recomienda poner espacio a los extremos de cada operador, de modo que se resalte su presencia y se facilite la lectura del código.
2. Se recomienda en algunas operaciones complejas, hacer uso de paréntesis redundantes o innecesarios que sirven para poder agrupar expresiones dentro de tales operaciones.
3. Cuando escribas operaciones que hagan uso de muchos operadores, procura revisar que las operaciones se estén realizando en el orden que tu esperas que se realicen, muchas veces el lenguaje tiene otra forma de asimilar la precedencia, por lo que el resultado real varía con respecto al esperado, en general, se recomienda forzar la precedencia de operaciones haciendo uso de paréntesis.

★ **Evita la incorporación de más de una instrucción por línea:** esto reduce notoriamente la legibilidad del código, ya que el programador habitualmente está acostumbrado a leer una instrucción por línea.

1. Si el código soporta la separación de sentencias en varias líneas, procura realizar una separación coherente, en el que cada punto de ruptura tenga sentido.
2. Si una instrucción abarca más de una línea, recuerda realizar la indentación necesaria.

★ **Uso de llaves:** si el lenguaje soporta llaves({}) para la separación de bloques, es altamente recomendado usarlas, ello facilita el proceso de distinción de bloques de código en forma rápida, permitiendo identificar y reparar errores en el código con menos dificultad.Si deseas evitar omitir una llave, abre y cierra el bloque de código que deseas crear, y luego introduce código dentro del bloque, con eso te aseguras la victoria.

★ **Contadores: nunca** olvides inicializar los contadores y sumadores.

★ **Usar minúsculas:** no solo en las etiquetas, sino también en los atributos para facilitar la lectura de tu código.

Buenas prácticas en HTML5

Además de las reglas generales, veamos algunas buenas prácticas específicas de HTML5 que nos ayudarán a la hora de un correcto maquetado.

1) **Hacer uso del Doctype**: la declaración de un “Doctype” deberá localizarse al inicio del código HTML. **En la declaración del doctype NO debe haber NINGÚN espacio en blanco ni cambio de línea ANTES**, la mayoría de los navegadores saben arreglárselas, pero en IE provocará “misteriosos comportamientos”.

2) **Uso preciso de las etiquetas:** usar de la forma más precisa posible las etiquetas de HTML5 en lugar de abusar del uso del . A modo de ejemplo, lo correcto sería utilizar:**<footer>** **En lugar de: **

★ **Extra tip:** sino recordas cuales son las nuevas etiquetas incorporadas, podes emplear las hojas de trucos de HTML5 para verificar que etiqueta utilizar.Otra herramienta útil para verificar que el navegador soporta las etiquetas que quieras usar en tu maquetación es Caniuse aquí el enlace.

3) **HTML5 semántico:** hace referencia a **usar las etiquetas de encabezados “<h1>…<h6>” de manera jerárquica** para otorgar importancia a nuestro contenido y que éste sea clasificado acertadamente en **SEO.**

4) **CSS Reset:** es válido usar un **CSS Reset** para resetear desde cero ciertos estilos que por default traen algunos elementos, también puede usarse Normalize.css.

5) **Ficheros externos:** siempre usar archivos externos para colocar nuestro JavaScript y CSS.

6) **Enlace de archivos en HTML:** enlazar **los estilos CSS** que vamos a utilizar al inicio de nuestro HTML. Los archivos externos JavaScript deben enlazarse al final de nuestro HTML antes del cierre de la etiqueta </body>

7) **Usar los atributos "alt" y "title" para las imágenes:** el atributo **"alt"** significa "texto alternativo", y quiere decir, que si no se encuentra la imagen se debe mostrar ese texto, entonces un error común es introducir toda una oración, lo cuál rompería horriblemente la página si le pedimos al navegador que no muestre las imágenes. Por otra parte los buscadores comprenden el significado de la imagen por su **"title",** el título sí puede llevar una oración explicativa.

8) **Cierre de etiquetas:** prestar atención al cierre de etiquetas. Muchas de las veces veremos que algunos desarrolladores no cierran por ejemplo **< li >** pero la buena práctica es cerrarla así: **< / li >.**

9) **Valida tu código:** Hacer validación del código que estamos escribiendo y eso lo podemos hacer por ejemplo, con W3C Validator.

10) **Encoding**: La razón por la que se hace uso de los "&aacute" es que la página no define su encoding, debe estar en UTF-8, y especificar el meta para el encoding, de modo que no haya ninguna necesidad de utilizar HTML entities para las tildes y caracteres raros.

11) **Correcto uso de " ": sin caer en el abuso de esta tag, es necesario y útil emplearla** para maquetar y dividir tu contenido en zonas o secciones. El primer paso es dividir el contenido de la página en secciones principales para organizar la información de la misma. Con esto garantizarás un contenido ordenado y con una buena arquitectura de la información.

12) **NO utilizar tablas, A MENOS que se trate de información tabulada:** si la naturaleza de la información es una relación de datos tabulados, entonces nada mejor que una tabla para representarla (ej: tablas comparativas, desgloses, una relación de elementos con sus respectivas propiedades, etc). A continuación unos ejemplos de uso correcto e incorrecto.

- ★ **incorrecto**: que el marco del sitio sea una tabla que lo envuelve todo para darle estructura.
- ★ **correcto**: que una tabla con “info tabulada” forme “parte” del cuerpo de un contenido.

13) Separa el contenido de la página HTML del estilo con que se muestra: el código HTML contendrá la información, el código CSS el estilo y la manera en que se presenta. Siempre hay que usar estilos separados y no dentro de la página HTML, así el código será más limpio y permitirá modificaciones de manera más eficaz.

14) Unificar todos los estilos en una sola hoja de estilos: cada archivo implica una solicitud HTTP, lo cual hace que el tiempo de carga de la página sea más lento. Por eso es mucho mejor usar un framework que permita “agregar” los ficheros CSS y JS en un único fichero para CSS y uno para JS (y resuelve el problema de las descargas paralelas).

15) No abusar de float: esto es muy importante, para garantizar la compatibilidad entre los diferentes navegadores, **los divs no deben ser flotados masivamente** porque en muchos browsers (IE por ejemplo) el sitio se rompe de mala manera, los float se deben usar solo en los casos que ya están identificados como válidos y multi-browser.

**16) Para maquetar los menús te recomendamos utilizar una lista desordenada **

17) Utilizar etiquetas <fieldset> y <label> en los formularios: es útil usar la etiqueta <label> para dar nombre a los elementos de un formulario y eliminar el uso de la etiqueta párrafo <p> o para evitar introducir texto sin etiqueta alguna. Igualmente es importante dividir los sets dentro de los formularios mediante la etiqueta <fieldset> y cuando sea necesario nombrar los set mediante la etiqueta <legend>.

Estructurar un proyecto en Programación

Convertirse en un programador no solo es aprender la sintaxis y los conceptos de un determinado lenguaje de programación: se trata de emplear ese conocimiento para hacer programas. Por eso hablaremos del proceso en general y en particular para el desarrollo exitoso de un programa.

Pautas Generales

Para lograr un buen desarrollo, es conveniente poner en práctica las siguientes pautas:

- ★ **1 - Seleccionar el talento y los recursos apropiados:** realizar un proceso de selección del talento con las destrezas necesarias y experiencia relevante es vital para garantizar el éxito del proyecto. Es importante asignar el trabajo apropiado a la persona indicada. Por otro lado, no hay que escatimar a la hora de invertir en herramientas que aumentan la productividad y eficiencia del equipo de desarrollo: hardware moderno, software y plataformas de desarrollo y de pruebas actualizado, y herramientas automatizadas ayudan a que el equipo pueda imprimir todo su conocimiento para garantizar un producto sólido, fiable y robusto.
- ★ **2 - Escoger el proceso de desarrollo apropiado:** el ciclo de vida del desarrollo del software tiene una fuerte dependencia del proceso elegido. El modelo en cascada, la metodología ágil, el enfoque iterativo en espiral, son todas formas contrastadas de alcanzar el éxito. La dificultad está en elegir bien qué metodología le conviene más a cada tipo de proyecto.
- ★ **3 - Presupuestos y estimaciones razonables:** muchos proyectos fracasan o se prolongan en el tiempo por hacer estimaciones poco realistas. Una planificación razonable depende de fijar bien los tiempos, el presupuesto, los recursos y los esfuerzos. Lo mejor es usar técnicas de estimación y presupuestarias contrastadas.
- ★ **4 - Fijar hitos más pequeños:** los grandes proyectos deben complementarse con mini-hitos para poder hacer mejor seguimiento y mejor gestión de riesgos, y en general para mitigar incidencias de forma controlada.
- ★ **5 - Definir bien los requisitos:** documentar de forma efectiva los requisitos es la columna vertebral para poder alinear el producto final con los objetivos empresariales. Es imperativo que se reúnan todas las partes (clientes, responsables de empresa y los líderes de los equipos) para documentar los requisitos de forma clara, sin dejar lugar a dudas o a la improvisación.Es necesario definir los requisitos básicos, los derivados y los implícitos, tanto funcionales como no funcionales. La funcionalidad se puede obtener mediante escenarios de casos de uso. Los requisitos en torno al rendimiento, funcionamiento a prueba de fallos, de sistema, diseño y arquitectura, todos deben estar bien documentados y tenidos en cuenta.

★ **6 - Implementar el código de manera efectiva:** el uso de módulos más pequeños que están auto-probados, probados unitariamente y que se integran continuamente es una buena práctica muy extendida. La automatización de herramientas *build* y la ejecución automatizada de pruebas de regresión para cada funcionalidad incluida se recomienda para garantizar que la funcionalidad ya implementada no rompe.

★ **7 - Pruebas rigurosas y validación:** la planificación de pruebas, la creación de conjuntos de pruebas y la ejecución de las mismas son muy importantes con el fin de validar la funcionalidad desarrollada. De hecho, la planificación de las pruebas debe hacerse en paralelo a la fase de desarrollo. Igual de importante es la documentación que hagamos de las pruebas, informar de forma efectiva los errores, el rastreo de los errores y la corrección de los mismos. El uso de herramientas automatizadas al igual que procesos contrastados que aseguren que los errores se identifiquen en la fase más temprana posible y resueltos con el menor coste. Las pruebas unitarias, las de integración, las de funcionalidades, las del sistema y las del rendimiento son algunos tipos de pruebas. Cada nivel de prueba requiere su pericia, planificación y ejecución.

★ **8 - Documentación:** al igual que el propio software, es importante toda la documentación sobre el que se apoya - el plan del proyecto, requisitos y especificaciones, Diseño de Alto Nivel (HLD), Diseño de Bajo Nivel (LLD), planes de pruebas, informes de las pruebas, informes de estado y la documentación para los usuarios. Estos documentos ayudan a garantizar el entendimiento del software, trazabilidad y eliminar la dependencia del equipo de desarrollo original. Pueden usarse como referencia en el futuro por otras personas que necesites mantener, mejorar o usar el software.

★ **9 - Planificar sesiones de revisión de código:** las revisiones de código muchas veces son más efectivas, y sin duda menos costoso, para encontrar errores que si solo hacemos pruebas. Las revisiones de todos los entregables, del código y de la documentación es algo que siempre se debe hacer.

★ **10 - Garantizar la gestión del control de las fuentes del software:** el uso de una gestión efectiva del código fuente y la documentación, para que estén controlados según la versión, es fundamental para poder mantener la trazabilidad y la reversión controlada del código si fuera necesario.

★ **11 - Control de calidad:** ayuda a sacar adelante los proyectos de desarrollo sin graves trastornos y de forma más rápida. Desde la detección de fallos hasta el establecimiento de métricas claves, las mejores prácticas en este terreno han

demostrado ser un éxito a la hora de determinar si un proyecto está en condiciones de pasar a una nueva fase, si está listo para ser lanzado o entregado al cliente. Se deben fijar métricas y objetivos para asegurar que los requisitos, el diseño, el código, las pruebas y otras tareas vayan coordinadas.

- ★ **12 - Instalación y despliegue eficaz:** en muchas ocasiones cuando ya hemos probado el software y todo va bien, de repente el proyecto fracasa en casa del cliente o cuando estamos en fase de implementación y despliegue. Es muy importante tener un buen plan de despliegue y hacer una lista a modo de “checklist” para evitar desastres.
- ★ **13 - Soporte y mantenimiento:** Incluso cuando el software desplegado está funcionando, debe haber un proceso de soporte y mantenimiento, previamente diseñado y pensado, para poder informar y reportar errores y mejoras al equipo de desarrollo de forma eficaz.

Pautas específicas:

A la hora de iniciar nuestro desarrollo podemos recurrir a distintos modelos para esquematizar las etapas de nuestro proyecto. Los más conocidos son : **el modelo de desarrollo en cascada y el modelo de desarrollo en espiral.**

Por su sencillez explicamos brevemente cómo estructurar los pasos de nuestro desarrollo bajo el modelo de cascada, el cual **define que las siguientes fases deben cumplirse de forma sucesiva:**

1. **Especificación de requisitos.**
2. **Diseño del software.**
3. **Construcción o Implementación del software.**
4. **Integración.**
5. **Pruebas (o validación)**
6. **Despliegue (o instalación)**
7. **Mantenimiento.**

Código redundante: ¿Como evitarlo?

¿Qué es el código redundante? En programación, se conoce como código redundante a cualquier parte del código fuente que tenga algún tipo de redundancia tales como recalcular un valor que ha sido calculado previamente y todavía está disponible.

Estándares de Programación

Es muy importante para un programador definir el "estilo" de programación que va a utilizar. Algunos, usan nombres de sus seres queridos para nombrar objetos y variables en el programa, otros, usan nombres aleatorios para sus variables de código, y así podemos seguir con una infinidad de ejemplos. La pregunta es: **¿Cuál es el estilo adecuado?**: a decir verdad, no existe un "estilo" mejor que otro, pero para sortear el inconveniente de no saber qué estilo elegir suelen seguirse los denominados **"estándares de programación"**. Un estándar de programación es: **“una forma de *normalizar* la programación de forma tal que al trabajar en un proyecto, cualquier persona involucrada en el mismo tenga acceso y comprenda el código”.**Esto nos permite:

- ★ Definir la escritura y organización del código fuente de un programa.
- ★ Facilita a otro programador la modificación de tu propio código fuente aunque no estés trabajando en el equipo.
- ★ Definir la forma en que deben ser declaradas las variables, las clases, los comentarios.
- ★ Especificar qué datos deben incluirse acerca del programador y de los cambios realizados al código, etc.

Criterios de un buen estándar

Si bien hay muchos estándares que podemos usar, debemos elegir aquel que se adecue más a nuestro estilo de programación. Un buen estándar considerará los siguientes factores:

- ★ **Factor mnemotécnico:** Para que el programador pueda recordar el nombre de una variable fácilmente
- ★ **Factor sugestivo:**Para que otros programadores puedan leer y entender rápidamente nuestro código.
- ★ **Consistencia:**Tiene que ver con usar las mismas convenciones de nomenclatura en todo el programa y hacer que el texto del código sea ***"legible"***,

A continuación, compartimos algunas convenciones para organizar mejor nuestro trabajo.

Nombres de variables y programas

Variables

- ★ Los nombres que se usen deben ser significativos.
- ★ Los nombres deben estar en minúsculas, excepto la primera letra de cada palabra a partir de la segunda.
- ★ Una variable $aa o $a1 no significan nada. No hay problema en usarlo si es una variable temporal que va a ser empleada en las líneas siguientes, pero si va a ser utilizada más lejos en el programa, debe tener un nombre significativo.

Ejemplo:

SI	NO
$nbeEmpleado	Nombre_Empleado NOMBRE NbeEmpleado

Nombres de registros en tablas.

Cuando se lee un registro de una tabla, el nombre del registro, debe empezar por $row y luego tener el nombre de la tabla.
Ejemplo:

SI	NO
$rowPER	$registroDeCliente

Nombres de programa.

Todo en minúscula excepto la primera letra de cada palabra a partir de la segunda.Todos los programas deben tener la extensión PHP preferiblemente. Cuando un programa es llamado directamente desde rec.php o desde otro programa de N2C, usar el nombre de la tabla, seguido de una indicación de cómo está siendo llamado.
Ejemplo:

SI	NO
PERhtml.htm PERformula.php PERpre.php PERpost.php PERextendido.php PERjsvalidacion.js PERjs.js	formulaPER.php PER_pre.php extendidoPER.php CLIENTESformula .php

Constantes y Variables globales

Constantes

Se deben evitar constantes numéricas sin mucho significado,también es conveniente definir las constantes en el programa. Todos los caracteres deben estar en mayúsculas y las palabras separadas por "_".
Ejemplo:

SI	NO

define("EDAD_VOTACION", "18");
...
if (EDAD_VOTACION <= $edad)
	if (18<$edad)

Variables globales

Se debe evitar el uso de variables globales ya que pueden ser modificadas erróneamente y pueden causar errores muy difíciles de identificar. Si se usan, para poder identificarlas, deben estar en mayúsculas.
Ejemplo:

$BD
$EEE
$USUARIO

Corchetes e indentación

La indentación es algo que ayuda a darle claridad a un programa y es necesario que se haga bien. Debe hacerse con "tabs" y no con espacios en blanco. **Los corchetes de un bloque if, o switch, o for, deben ir en la misma línea de la cláusula.**
Ejemplo:

```
if ($edadCliente<$edadExigida){
        instruccion1;
        instruccion2;
} else {
        instruccion3;
};
```

Ejemplo de indentación apropiada:

```
    function verificarCondicion() {
      if (condicion1) {
            if (condicion2) {
```

```
                    while (condicion3) {
                        instruccion1;
                    };
            };
            instruccion2;
    }else{
            instruccion3;
    };
};
```

Claridad de los programas

Es importante que los programas y rutinas que se escriban sean claros y fáciles de entender. Por eso, además de dar la explicación de que hace cada programa o función al principio, hay que tratar que las funciones quepan en una sóla página y que antes de cada sección se explique qué es lo que se está haciendo. Sobre todo, cuando se usan "truquitos", es muy importante que se explique lo que se está haciendo.

Inclusión de funciones y rutinas

Muchas veces se incluye un archivo que tiene muchas funciones. Es muy importante, al hacer el **require** del archivo, que se indiquen los nombres de las funciones que se están utilizando. De forma que cuando se quiera saber de donde viene una función se pueda, al buscar la primera ocurrencia del nombre.

Ejemplo:

```
require("procesos/caseanexos.php");   //funciones VerAnexos,EditarAnexos
require("rutinas/campoenlace.php");   //Rutinas CampoEnlace,MostrarLinks
require("rutinas/impresion.php");     //Variable ParPosibles, rutina Desestacar
```

Ofuscación de código CSS

¿cómo funciona?

Es una forma única de proteger sus hojas de estilo, transformandolas de una manera que nadie querrá modificarlas. Un ofuscador también se encarga de minimizar y hacer modificaciones al programa, fusionando los nombres de variables, funciones y miembros. Hace que el programa sea mucho más difícil de entender y reduciendo su tamaño en la negociación. Algunos ofuscadores son bastante agresivos en sus transformaciones. Algunos requieren anotaciones especiales en el programa de origen para indicar qué cambios pueden ser seguros o no.

Cualquier transformación conlleva el riesgo de introducir un error. Incluso si el ofuscador no causó el error, el hecho de que podría tener es una distracción que ralentizará el proceso de depuración. Las modificaciones al programa también aumentan significativamente la dificultad de depuración.

Se pueden encontrar dos tipos de ofuscación de código: la ofuscación superficial y la ofuscación profunda. La primera siendo solamente un reacomodo de la sintaxis del programa, ya sea cambiando nombres de variables o algún otro método similar, y la segunda que intenta cambiar la estructura actual del programa, cambiando su control de flujo o modo de referenciar sus datos, esta última teniendo menos efecto en el proceso de estar oculto a la ingeniería inversa, ya que no se ocupa de disfrazar la sintaxis del código.

Ejemplo de un código sin ofuscar

```
var a="Hello World!";
function MsgBox(msg)
{
    alert(msg+"\n"+a);
}
MsgBox("OK");
```

Código ofuscado

```
var
_0x55ae=["\x48\x65\x6C\x6C\x6F\x20\x57\x6F\x72\x6C\x64\x21","\
x0A","\x4F\x4B"];var                    a=_0x55ae[0];function
MsgBox(_0x9e43x3){alert(_0x9e43x3+_0x55ae[1]+a);}
;MsgBox(_0x55ae[2]);
```

Ventajas y desventajas de ofuscar código

Ventajas

- Evita que tu código fuente sea obtenido por alguien más.
- Genera múltiples versiones del mismo código.
- A pesar de que los antivirus son cada día más potentes, nace una nueva técnica que logra poner en dificultad a estos sistemas actuales de defensa, llamada ofuscación de código dinámica, y se basa en la transformación o enmascaramiento del código para evitar que se pueda determinar la forma que tenía originalmente.

Desventajas

- La ofuscación del código puedes ser más difícil de reconstruir.
- Si es utilizada como una arma para lograr burlar los actuales mecanismos de defensa puede ser muy peligrosa. Aunque se podría detectar su peligrosidad mediante algunos de los análisis mencionados en este artículo, consumiría mucho tiempo y lo más probable es que mientras se realizara este tipo de análisis, se descuidara algún otro aspecto de la seguridad.
- El mantenimiento y la solución de problemas de una aplicación son tareas más difíciles.

HTML semántico

Usar HTML semántico significa aplicar a cada parte del contenido la etiqueta más adecuada a su tipo.

Por ejemplo: para los párrafos se usa `<p>` y para las listas, `<ol>` o `<ul>`. Adoptar esta práctica implica entender que HTML tiene que ver cada vez más con la estructura que con la presentación, por lo cual etiquetas como `<i>` deben dejar de utilizarse.

De manera similar, la etiqueta `<table>` puede permitir diseñar una plantilla tan bien como la etiqueta `<div>`. A pesar de que el resultado visual sea el mismo, `<table>`

solo es adecuada para presentar información tabular. En cambio, etiquetas como: `<div>`, `<section>`, `<header>` y `<footer>` se crearon específicamente para separar secciones dentro de una página, por lo que en este caso aportan semántica.

Utilizar código semántico ofrece varias ventajas:

- **Posicionamiento en buscadores**. Los motores de búsqueda analizan el código para saber qué clase de contenido muestran y como deberían ser mostrados. La semántica del código todavía puede ser un factor secundario, pero indudablemente es cada vez más importante a la hora de posicionar nuestro sitio web.

- **Accesibilidad**. Los lectores de pantalla para usuarios ciegos organizan la lectura del contenido de acuerdo con la estructura del código.

- **Practicidad**. El código semántico es más fácil de entender y mantener.

- **Reusabilidad**. La separación del contenido de la presentación permite que una página sea rediseñada cambiando sólo el CSS.

En la medida en que los desarrolladores mejoren el significado de su código avanzaremos hacia la Web Semántica, una era en la que el contenido esté tan bien etiquetado que los buscadores puedan encontrar exactamente lo que el usuario desea sin estar navegando de página en página.

¿Qué es un validador?

Se trata de servicios normalmente gratuitos que evalúan el código (tanto de la contenido HTML como de la presentación CSS), e informan si tienen errores o están bien escritos. Cumplen una función similar al corrector de ortografía y gramática de Microsoft Word. Pero no solo se encargan que no tengan errores,si no que algunos te dan el código ya limpio.

¿Para qué validar?

En general, si tu sitio no es válido, es posible que tenga errores. Los errores pueden hacer que no se vea bien en algunos navegadores Web, y si es válido no debería haber problemas.

Además, una página válida y bien estructurada funciona a la perfección para navegadores solo texto, navegadores redactores para ciegos, y también para que los bots sepan de qué trata tu web.

Es muy popular el dicho que ronda “Google es ciego y gusta de las webs bien estructuradas.”,en otras palabras,si tu sitio es válido, tus SEO aumentarán.

¿Cómo los uso?

Como todo software, ninguno es idéntico a otro, pero en la página del validador de CSS de la w3c hay un tutorial de cómo usarlo, y como es de la misma organización, tiene la misma interfaz y por lo tanto sabrás usar el validador de HTML:
http://jigsaw.w3.org/css-validator/manual.html.es

¿Según cuáles validadores deben ser válidas mis páginas web?

Tus archivos HTML y CSS deben ser validados por lo menos, con estos validadores:
Validador HTML y XML:http://validator.w3.org/
Validador CSS: http://jigsaw.w3.org/css-validator/
Validador de accesibilidad: http://www.tawdis.net/

Bibliografía Web:

HTML5 y su historia:

- https://developer.mozilla.org/es/docs/HTML/HTML5
- https://developer.mozilla.org/es/docs/HTML/HTML5/HTML5_lista_elementos
- http://librosweb.es/libro/xhtml/capitulo_1/breve_historia_de_html.html

Buenas Prácticas

Prácticas generales:

- http://wiki.inf.utfsm.cl/index.php?title=Buenas_Practicas_de_Programaci%C3%B3n

De HTML5:

- https://www.codejobs.biz/es/blog/2016/01/19/buenas-practicas-en-html5
- http://www.danielgorostiaga.com/buenas-prcticas-para-la-maquetacin-en-html/

Cómo estructurar un proyecto de desarrollo.

Pautas generales:

- https://velneo.es/15-buenas-practicas-proyectos-desarrollo-software/

Pautas específicas: Ejemplos de modelos:

- https://sistemasvd.wordpress.com/2008/07/05/fases-del-proceso-de-desarrollo-del-software/
- https://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software

Código redundante

- https://es.wikipedia.org/wiki/C%C3%B3digo_redundante

Estándares de Programación

- https://es.scribd.com/document/36991779/Estandares-basicos-de-programacion
- http://yolopuedohacer.blogspot.com.ar/2010/06/estandares-de-programacion-manana-hoy-y.html

- http://www.net2client.net/manual/nuevomanual/Estandares_y_normas_de_programacion.htm
-

Ofuscación de Código:

- https://yuiblog.com/blog/2006/03/06/minification-v-obfuscation/
- https://blog.reaccionestudio.com/como-ofuscar-codigo-javascript-generador-online/

HTML sémantica:

- http://www.4rsoluciones.com/blog/html-la-importancia-del-codigo-semantico-2/

Meta-tags:

- https://www.1and1.es/digitalguide/paginas-web/desarrollo-web/los-meta-tags-mas-importantes-y-su-funcion/
- https://quiwiq.com/posicionamiento/importancia-meta-etiquetas-seo/952
- http://www.alhsis.com/las-meta-tags-y-su-importancia-en-el-seo/
- https://desarrolloweb.com/articulos/etiqueta-meta-viewport.html
- https://www.netvoluciona.es/blog/La-web-movil-y-la-utilidad-de-la-etiqueta--viewport--80

Validator

https://josewebmasterlibre.wordpress.com/2010/08/30/losvalidadores/