# Resumen: Sistemas Web

# Tema 3: HTML

(Por Tim Berners-Lee) (CERN, 1991)

**HTML** es un lenguaje de marcado (“Markup language”), que consiste en un sistema para “anotar documentos” (es una serie de reglas, sobre todo sintácticas, que sirven para formatear un texto), donde su contenido es distinguible del texto “normal” (contenido), donde el orden es fundamental y no se muestra cuando se procesa.

Otros ejemplos de lenguajes de marcado: LaTex, SGML, XML…

HTML: “Hyper Text Markup Language” / “Lenguaje de marcado de hipertexto (texto con referencias a otro texto.)”. Es el lenguaje por excelencia con el que se construyen páginas web.

**Compuesto por:**

Elementos: Todo lo que está comprendido entre una etiqueta de apertura y una de cierre incluyendo las etiquetas.

Contenido: Todo lo que está comprendido entre una etiqueta de apertura y una de cierre, sin incluir las etiquetas.

Atributos: una etiqueta tendrá nombre, y puede tener atributos o no. Algunos de ellos pueden no tener valores asociados, se les llama, atributos compactos o booleanos, cuya función es indicar si algo está presente o no en un elemento.

Algunos atributos compactos son:

* Hidden
* Autofocus
* Readonly
* Disable
* Checked

*Void* *Elements* o Elementos vacíos: son etiquetas que no tienen contenido, ni se corresponden con otra etiqueta de cierre (ni siquiera es necesario colocarles la “/” de cierre).

Algunos *void elements* y sus usos:

* <img> insertar imagen en el doc
* <br> insertar un salto de línea
* <hr> insertar una línea horizontal
* <input> representa un campo de entrada en formularios
* <meta> proporciona metadatos sobre el doc
* <link> víncula recursos externos como hojas CSS
* <source> especifica recurso multimedia para audio o video

Tabla

Descripción generada automáticamente

Por temas de normalización de estilos, se recomienda:

* Recomendable usar dos espacios (no tabs) para sangrado
* Especificar atributos alt, width y height de las imágenes



Atributos globales:

Estándar de accesibilidad ARIA

¿Qué es?

ARIA (Accessible Rich Internet Applications) es un conjunto de atributos diseñados para mejorar la accesibilidad web, especialmente para personas que utilizan tecnologías como lectores de pantalla.

Atributos clave:

role: Define el rol de un elemento para tecnologías de asistencia.

Ejemplo: <div role="button">Clic aquí</div> indica que un <div> actúa como un botón.

aria-\*: Atributos específicos que describen estados o propiedades.

Ejemplo: <button aria-disabled="true">No disponible</button> indica que el botón está desactivado.

Event Handlers

¿Qué son?

Son atributos usados para ejecutar código JavaScript cuando ocurre un evento en un elemento.

Ejemplos comunes:

onclick: Se ejecuta cuando el usuario hace clic en un elemento.

Ejemplo: <button onclick="alert('¡Hola!')">Clic</button>.

onfocus: Se ejecuta cuando un elemento gana el foco (por ejemplo, al tabular a un campo de formulario).

Ejemplo: <input onfocus="this.style.background='yellow'">.

onsubmit: Se ejecuta al enviar un formulario.

Ejemplo: <form onsubmit="return validarFormulario()">.

Archivos relacionados con las páginas web (en directorio raíz (o al especial)):

*robots.txt* 🡪 Indica a los web crawlers que URLs pueden acceder al sitio. Sirve para evitar sobrecargar el sitio con peticiones. No sirve para que un recurso no aparezca en los buscadores, en otras palabras, no protege contenido sensible.

*security.txt* 🡪 es un archivo que se puede incluir en el directorio especial del sitio, es un estándar y está pensado para ofrecer a los usuarios información sobre cómo contactar con el equipo de seguridad, cómo reportar alguna vulnerabilidad, políticas de seguridad, claves públicas…

*ai.txt* 🡪 no es un estándar todavía, pero es una propuesta sobre como controlar las imágenes asociadas a IA. El documento está dirigido a las inteligencias artificiales y algoritmos de machine learning. Propone delimitar que contenido de la web puede o no, ser usado por estos sistemas.

XHTML: es un lenguaje de marcado, *extendido* de HTML, y basado en XML; pretendía resolver ciertos problemas como inconsistencias presentadas por HTML, con el objetivo de mejorar la interoperabilidad, portabilidad y establecería una estricta estructura sintáctica, sin embargo, con la llegada de HTML5 (que ofrecía flexibilidad y estructura) perdió relevancia.

# Tema 5: JAVASCRIPT (JS)

Fue creado para que las páginas web “estuvieran vivas”. Es un lenguaje interpretado (aunque algunos navegadores usan JIT (just-in-time compilation; donde las partes que se ejecutan con mayor frecuencia *warm* se convierten a bytecode; todo esto con el objetivo de optimizar ciertas funciones), multiplataforma, orientado a objetos, de tipado débil y dinámico (es decir, que una variable puede cambiar de tipo durante la ejecución del programa), y completamente integrado con HTML y CSS. Los programas funcionan a través de *scripts* y es necesario in “JavaScript Engine” para ejecutarlo (la mayoría de los navegadores modernos están preparados para ello).

JS está basado en la especificación ECMA Script ES, que es un estándar para lenguajes de scripting, se limita a describir la sintaxis y semántica del core, y cada implementación es responsable de añadir otras funcionalidades.

Se usa en el desarrollo web, en el servidor de node.js, en el james web telescope, en proyectos relacionados con la programación de drones…

Cuando se ejecuta JS en el navegador este funciona dentro de un entorno muy controlado llamado *sandbox*. Esto se aplica para entre varias cosas proteger la integridad del SSOO de los usuarios; Respetar el principio de “Same Origin Policy” (que establece que cada pestaña/ventana del navegador, funciona en su propio espacio de memoria, y de como un programa independiente); No puede comunicarse con otros dominios, solo lo hará directamente con el servidor de donde viene, en caso de ser necesario.

Existen también lenguajes “*transpiled*” que tienen su propia sintaxis, pero pueden convertirse a JS a través de un proceso de transpilación. Existen porque están especializados en ciertas tareas, aunque muchas veces, solo aumentan la complejidad del proceso de desarrollo, y de la transpilación.

El código puede estar directamente incrustado en el HTML como:

<script>

alert('Hello, world!');

</script>

O en un archivo externo .js al que se referencia como:

<script src="/path/to/script.js"></script>

Actualmente conviene añadir los scripts en el <head>, agregando a las etiquetas del script los atributos de *defer (No se ejecutará hasta que se haya ccargado completamente el HTML, ya que se carga en paralelo con el resto de elementos)* o *async (Se carga en paralelo con otros scripts, y se ejecuta solo cuando está disponible)*

Una captura de pantalla de una computadora

Descripción generada automáticamente con confianza media

Esto significa que es conveniente usar *defer* cuando los scripts dependan de otros elementos del HTML, o necesiten ejecutarse en un orden específico; mientras que si los scripts son independientes, y no importa el orden en el que se ejecuten (con respecto a otros scripts) será útil *async*.

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String>

Tema 6: Node JS (2009)

No es un lenguaje de programación, no es un *framework…* Es un entorno de ejecución para código JS (*runtime environment for excuting JS code*), con una arquitectura no bloqueante (o asíncrona). Es un software *cross-platform* lo que significa que es altamente portable, ya que se puede ejecutar de la misma manera sobre diversos SSOO.

Node conserva el principio de proceso único, (que similar a JS) su ejecución está basada en un único hilo o proceso principal.

A pesar de tener una arquitectura/modelo de **I/O** no bloqueante (en donde un hilo es capaz de atender diferentes solicitudes sin detenerse); Node también permite ejecutar tareas de manera secuencial, siguiendo cierto flujo lógico (lo cual podría interpretarse como “síncrono”).

Basado en la estructura de *event-loop*, un mecanimos de escucha pasiva, basado en un bucle de eventos más bien reactivos. Ejecuta las funciones asociadas cuando eventos (como solicitudes HTTP o respuestas de BBDD...) ocurren.

Implementado por diversas empresas como LinkedIn, Netflix, y Paypal.

A diferencia de JS, no se trabaja con el DOM (*Document Object Model*: que es la representación en forma de árbol de la estructura HTML de la página web; le permite a JS interactuar con los distintos elementos. El documento representa la estructura en sí del HTML, y los objetos son todos aquellos elementos dentro, que se pueden manipular (como objetos) para darle dinamismo a la página), para interactuar con la interfaz gráfica, si no que se enfoca en la lógica y ejecución del servidor.

Node JS, por tanto, tampoco tiene acceso a variables del navegador tales como, *document* o *window*. Tendrá acceso a otras que sean propias del entorno del servidor, como *global, process* o *require…*

Está además adaptado para leer, escribir y manipular archivos del sistema directamente.

Finalmente, no tiene problemas de compatibilidad entre navegadores, ni versiones.

Parte un poco más práctica:

Npm (Node package Manager) es el gestor de paquetes de Node JS, y es el encargado de gestionar las dependencias del proyecto (como su descarga, actualización, versionado…), también permite ejecutar y definir tareas.

Para iniciar:

$ npm init

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Esto generará un archivo “*package.json*” que contiene la metadata del proyecto (datos como: licencias, dependencias, versiones, descripción, configuración…)

En *main*: indicamos manualmente cuál queremos que sea el módulo de exportación del paquete, si no existe un valor por defecto, será el “*index.js*”

Dentro del apartado de *scripts* se definirán los scripts ejecutables, tales como por ejemplo, los archivos de tests unitarios. Que más adelante se podrán ejecutar con:

$ npm run <*script name*>

Solo puede haber un autor, al que se le puede asociar un nombre, email y URL, mientras que puede haber varios contribuidores.

Con respecto al apartado de *dependencies*, se añaden automáticamente cuando se instala un paquete con:

$ npm install <*package name*>

Un detalle sobre esto: Es posible no asignar un número de versión concreta a las dependencias, esto se hace en principio con la intención de dejar abierto un rango de versiones para usar, sin embargo, representa un problema mayor del que resuelve, ya que puede generar inconsistencias y hace que el paquete no sea reproducible.

También se puede agregar un apartado para las dependencias solamente de desarrollo (es decir que no serán necesarias en producción), y estás también se añaden automáticamente en el apartado de *devDependencies* con la instalación:

$ npm install --save-dev <*package name*>

Con siguiente comando, generamos el “*package-lock.json*”, que especificará de manera exacta la versión de cada dependencia, resolviendo el problema acerca de la reproducibilidad del paquete anterior. Este archivo también se genera y actualiza de manera automática:

$ npm install

\*Extra: node\_modules es la carpeta donde Node.js guarda todas las dependencias que tu proyecto necesita para funcionar. Pero no se recomienda subirla al repositorio, ya que su función ya está cubierta con los json’s anteriores, y en comparación, este otro archivo ocupa demasiado espacio innecesario. \*

Express JS

Es un *web* *framework* diseñado específicamente para Node JS, que simplifica la creación de aplicaciones web y API’s.

Entre sus características principales:

Express permite manejar solicitudes HTTP (POST, GET, PUT, DELETE) y asociarlas a rutas específicas.

Soporta el *middleware*, entendiendo a este como las funciones que se ejecutan entre la solicitud del cliente y la respuesta del servidor (esto permite realizar tareas como validación, autentificación, manejo de errores... Y será necesario ejecutar un “*next()*” para que continúe la cadena de tareas).

Se considera a sí mismo como “sin opinión” (*unopinionated*), esto significa que Express no impone ni una arquitectura, ni define motores de plantilla, ni bases de datos, ni ningún tipo de herramienta. Esto permite entonces, que sea flexible y extensible (gracias también a los numerosos paquetes disponibles con npm)

Después de crear nuestra instancia *app* de Express, podemos:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

<https://expressjs.com/en/4x/api.html#app.METHOD>

Un temp

Loggin

Es el proceso de registrar información relevante sobre el funcionamiento de un programa o sistema, para posteriormente analizarlo. Se utiliza principalmente para monitorizar el comportamiento de la aplicación, identificar problemas y resolver *bugs*.

Los objetivos principales para conservar los LOGS del sistema:

Comprobar que el programa se comporta como se espera 🡪 que todo funciona correctamente.

Identificar errores, fallos o vulnerabilidades, tras registrar que ha sucedido en un periodo de tiempo que abarque, antes-durante-después del fallo 🡪 solucionar bugs.

Registrar parámetros de entrada o resultados, para análisis (en cualquier nivel (administrativo, técnico optimizador...)) posterior 🡪 análisis de parámetros.

Información que SÍ interesa loggear:

* Cambios en el estado del programa (cambios en la base de datos, interacción del usuario)
* Interacción con el usuario (como inicios y cierres de sesión, formularios enviados…)
* Interacción con otros programas (solicitudes HTTP, respuestas, mensajes de servicios…)
* Interacción con ficheros (lectura y escritura, tiempos de acceso y modificación…)
* Comunicaciones (comunicación de red, solicitudes HTTP, respuesta de API, mensajes entre sistemas si la aplicación interactúa con otros servicios).
* Entrada y Salida de un método (para saber cuándo empieza un proceso, cuando termina, cuánto tiempo se ha tardado, y para seguir el flujo del programa correctamente)
* Errores y Excepciones (manejado o no, el error siempre debe ser registrado, con todos sus detalles (preferiblemente el stack trace), lo mismo para las excepciones).

Información que NO interesa loggear:

* Información sensible (Información personal, médica, financiera, contraseñas, direcciones IP…)
* Registrar URL’s en los logs, puede a veces llevar consigo información sensible, como emails, contraseñas o tokens, ID’s o claves privadas…

Debido a que el tamaño de los LOGS puede crecer rápida y desenfrenadamente se recomienda:

1. Rotación de los Logs: Rotar los logs después de cierto tamaño o edad.
2. Logs limitados por nivel/tipo: Cada tipo de aporta información clave en momentos diferentes del ciclo de vida del proyecto, por ejemplo, los logs de debug, pueden ser útiles durante el desarrollo. Esto permite no registrar demás.
3. Compresión de Logs
4. Almacenamiento adecuado (sistemas distribuidos o centralizados, dependiendo del alcance y riesgo)

Tipos de Logging:

* Logs de información (info) 🡪 Recogen data sobre el flujo regular del programa
* Logs de advertencia (warn) 🡪 Informan sobre condiciones/eventos no deseados (que no son precisamente errores, pero indican un posible problema futuro)
* Logs de error (error) 🡪 Captura las situaciones de fallo/errores en el sistema, que detienen una operación, pero no todo el sistema.
* Logs de depuración (debug) 🡪 Brindan información de diagnóstico.
* Logs de situación fatal (fatal) 🡪 Informan sobre una situación catastrófica de la que no hay recuperación.
* Logs de rastro (trace) 🡪 Registra todos los posibles detalles del comportamiento de la aplicación.

Y… ¿Por qué no usar console.log() instead?

Tabla

Descripción generada automáticamente

El objeto *console* en Node JS es un objeto global, que permite escribir mensajes de LOG a consola, cuenta con métodos bastante sencillos, que pueden ser útiles en proyectos pequeños. Entre ellos tenemos algunos como:

$ console.log()

$ console.error()

$ console.warn()

$ console.debug()

Sin embargo, una vez más, es mucho más recomendable emplear una librería de logging para trabajar, ya que aportan ligereza y eficiencia (frente al uso de console). Establecen un formato estructurado y normalizado para los LOGS; e incluso mejoran su distribución, accesibilidad y gestión de los registros.

Existen un par de librerías de Logging para Node JS:

*Morgan* (Desarrollada por ExpressJS)

Es un middleware de logging, es decir, que se coloca entre el servidor y las solicitudes (captándolas y registrándolas de manera automática). Es especialmente útil en aplicaciones construidas con Express. Su función principal es gestionar el registro de las solicitudes HTTP que llegan al servidor (guardando detalles como: URL solicitada, código de estado HTTP, tipo de método, tiempo de respuesta…).

Es de fácil configuración, ya que solo se necesita integrar el middleware y seleccionar el formato adecuado (ya que funciona para un estado de desarrollo y otro de producción), lo que lo hace aún más flexible.

$ npm install morgan

Texto

Descripción generada automáticamente

*Winston*

Proporciona flexibilidad, escalabilidad y soporte para múltiples destinos llamados también “transportes”. Permite registrar los LOGS con diferentes formatos (desde JSON hasta texto plano), con niveles personalizables, y en diferentes ubicaciones, como en archivos, en bases de datos, o incluso en servicios externos. Se caracteriza por su desacoplamiento, ya que cada *transporte* puede tener su propio nivel y formato; esto proporciona una configuración modular del sistema de logging.

Es apropiado para proyectos complejos que requieran registros detallados y a la medida.

$ npm intall winston

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

*Pino …*

Hacer para el EXAMEN:

**Simulacro:**

1. Codificar que funcione Npm start:

=

Imagen que contiene texto, exterior, firmar, azul

Descripción generada automáticamenteInterfaz de usuario gráfica, Texto

Descripción generada automáticamente

1. Que un usuario pueda iniciar sesión

npm install ejs

npm install express

npm install express body-parser

1. Que se pueda registrar un usuario

**Examen Ordinario:**

Codificar:

Con un HTML y CSS predeterminados hacer cuadro para aceptar las cookies. Si se rechazan que te manden a abrir otra página. Si se aceptan, guardarlo en la sesión, para que si la persona se mueve entre páginas no le vuelva a aparecer el anuncio.

Teoría:

Diferencias entre servicios web y aplicaciones web

Nombra códigos de estado (400)

HTML busca 10 errores en el html

Json entero

Anexos:

El formato de numeración X.Y.Z es el estándar utilizado en Semantic Versioning (SemVer), que es una convención para la asignación de números de versión en software. La numeración se desglosa de la siguiente manera:

MAJOR (X):

Se incrementa cuando hay cambios incompatibles en la API o el comportamiento del software, lo que podría romper la compatibilidad con versiones anteriores.

MINOR (Y):

Se incrementa cuando se añaden funcionalidades nuevas de manera retrocompatible, es decir, sin romper el funcionamiento de las versiones anteriores.

PATCH (Z):

Se incrementa cuando se realizan correcciones de errores de manera retrocompatible, es decir, que no afectan a las funcionalidades existentes ni introducen nuevas características.

Ejemplo: Si tienes una versión 2.3.4:

2 es la versión MAJOR, indicando que podría haber cambios incompatibles con versiones anteriores.

3 es la versión MINOR, lo que indica que se han añadido nuevas funcionalidades de manera compatible.

4 es la versión PATCH, señalando que se ha corregido un error sin afectar las funcionalidades existentes.

Este sistema ayuda a los desarrolladores y usuarios a entender rápidamente el tipo de cambios que ha habido entre versiones.

JSON (JS Object Notation) (1999)

Basado en el intercambio de formato ligero, en el estándae ECMA-262. Soporta objetos, arrays y valores

REPL • Read Evaluate Print Loop • Entorno de Node.js en forma de consola • Para ejecutarlo: node