

Resumen: Sistemas Web

Índice:

{	
	T1 Sistemas Web
	T2 HTTP
	T3 HTML
	T4 CSS
	T5 JS
	T6 NodeJS
	NPM
	Event Loop
	Express
	Loggin
	Socket.IO
	Passport
	T7 Accesibilidad
}	

Tema 1: Introducción a Sistemas Web

Tim Berners-Lee es el científico informático inventor de la World Wide Web. Implementó el Hypertext Transfer Protocol (HTTP), tecnología clave para el funcionamiento de la web. Es fundador y director del World Wide Web Consortium (W3C), encargado de su desarrollo continuo, y de la World Wide Web Foundation.

Aplicación para internet/web

Una aplicación web es un **programa interactivo** que se puede **acceder** a través de un **navegador web**. Estas aplicaciones pueden realizar procesos complejos en el cliente o en el servidor. Se ejecutan directamente en el navegador sin necesidad de ser instaladas en el dispositivo del usuario. Los datos y archivos son procesados y almacenados en la web o en una intranet. Se codifican en lenguajes soportados por los navegadores web, y su **ejecución** se lleva a cabo **dentro del navegador** o en el caso de las aplicaciones basadas en **servidor**, utilizan protocolos de Internet para recibir solicitudes del cliente, procesar el código asociado y **devolver los datos**; todo esto los convierte en programas **accesibles de forma remota**.

Características: Rápida evolución; Gran alcance y altamente distribuido; Multiusuario; Portable (diferentes puntos y medios de acceso); Gran volumen de información; Multiformato y Multiproceso.

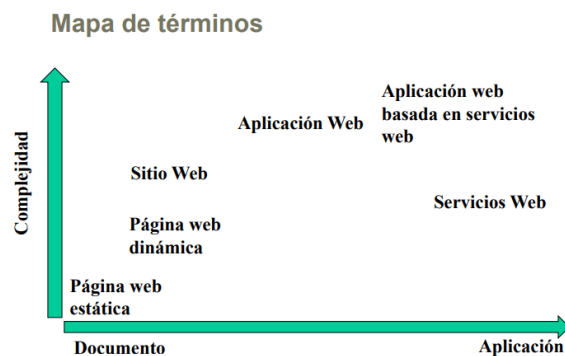
Las aplicaciones web ofrecen **ventajas** como ahorro de tiempo en instalación y despliegue, ausencia de problemas de compatibilidad, no ocupan espacio, permiten actualizaciones inmediatas, tienen bajo consumo de recursos, son multiplataforma, portátiles, de alta disponibilidad, difíciles de atacar por virus y son ideales para la colaboración. Sin embargo, presentan **desventajas** como menor funcionalidad comparado con aplicaciones de escritorio, limitaciones del navegador, dependencia del proveedor de red, alta dependencia del servidor, restricciones por el protocolo HTTP, cuello de botella en el ancho de banda con grandes datos, necesidad de encriptación y de software o versiones específicas de navegadores.

Modelo Cliente-Servidor

El modelo Cliente-Servidor se basa en servicios o funciones que requieren acceso a información difícil de distribuir, equipamiento especial o alta capacidad de cómputo. Los **servidores ejecutan** estos **servicios bajo petición**, mientras que los **clientes solicitan** la ejecución del servicio, **y presentan la respuesta al usuario**. Este modelo sigue un **protocolo de petición-respuesta** y se soporta a través de una red que facilita las interacciones entre cliente y servidor.



Comparativa de términos

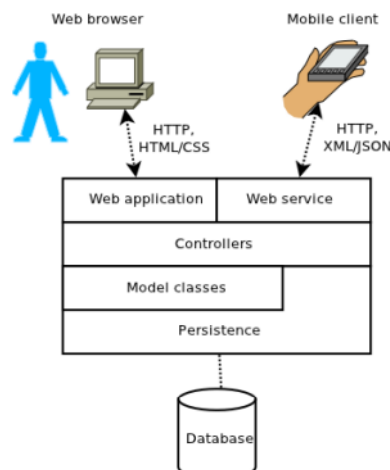


Sistema Web: es la **infraestructura** que permite que una aplicación web funcione correctamente, **proporcionando los recursos y servicios** necesarios para su **ejecución**.

Aplicación Web: es una **aplicación distribuida** y **plataforma interactiva** que realiza una función específica (como una tarea de negocio) y está basada en tecnologías de la web, utilizando recursos propios de la web para su operación.

Sitio web: es **un conjunto de páginas web** que están **agrupadas bajo un mismo dominio**, generalmente diseñadas para ofrecer contenido informativo (aunque sí que puede haber contenido dinámico, no suele ser ni complejo, ni pesado). Este tipo de sitios suele ser estático, es decir, su principal función es proporcionar información **sin requerir mucha** o ninguna **interacción del usuario**. Los sitios web se pueden crear fácilmente usando sistemas de gestión de contenido (CMS), que permiten la creación y administración de las páginas sin necesidad de tener conocimientos avanzados en programación.

Servicio web: está **diseñado para** ser utilizado por **máquinas**, no **por humanos**. **Estos servicios exponen APIs** (Interfaces de Programación de Aplicaciones) que permiten la comunicación entre sistemas. La interacción es **Machine to Machine** (M2M), lo que significa que un **sistema solicita datos o servicios** a otro sin la intervención de un usuario humano. Los servicios web proporcionan **información sin presentación**; la información se maneja en formatos como XML o JSON, que son adecuados para ser procesados por otras aplicaciones o sistemas.



Arquitectura

Arquitectura

- Capa de acceso (capa del navegador)
- Capa del servidor
- Capa de persistencia



Modelo básico: Consiste en un ordenador dedicado ejecutando un servidor que se conecta a una red para recibir y responder a las peticiones. Si la red es interna, se denomina intranet.

Modelo con separación de funciones: Este modelo también incluye un servidor dedicado y una conexión a una red, pero en este caso, el servidor tiene conectividad con otros servidores, como los de bases de datos, para distribuir las tareas y mejorar la eficiencia.

Modelo con separación de redes: Similar al anterior, pero con una separación de redes. Un servidor está conectado a una red externa para recibir peticiones, y otras redes internas (por ejemplo, de bases de datos) están protegidas por un cortafuegos.

Modelo de separación completa de funciones: En este modelo, las funciones del sistema están completamente divididas entre varios servidores. Unos sirven contenidos estáticos (como páginas HTML), otros generan contenidos dinámicos, y otros gestionan bases de datos, seguridad y otras tareas.

Modelo para un alto rendimiento: En este caso, se usan varios ordenadores dedicados para ejecutar el mismo servidor (nos referimos a un entorno distribuido diseñado para manejar un gran volumen de peticiones o usuarios simultáneamente). A través de un balanceador de carga, se distribuyen las peticiones y respuestas entre los servidores, que también se conectan a otros servidores, como los de bases de datos, para optimizar el rendimiento.

Modelo para una alta disponibilidad: Este modelo utiliza un repartidor de carga que distribuye las peticiones a múltiples servidores, asegurando que todos los elementos del sistema estén duplicados para evitar fallos y garantizar la disponibilidad continua del servicio.

Front-End

Tareas principales de un desarrollador Front-End:

Web Performance: Reducir tiempos de carga; Optimizar la usabilidad e interactividad del sitio; Garantizar un rendimiento fluido.

Responsive Web Design: Asegurar que la interfaz se adapte a diferentes dispositivos y tamaños de pantalla.

Compatibilidad entre navegadores (Cross-Browser Compatibility): Comprobar funcionalidad, rendimiento y accesibilidad en diversos navegadores.

Pruebas de extremo a extremo (End-to-End Testing): Simular escenarios reales; Validar la integración y la integridad de los datos.

Automatización de tareas (Build Automations): Procesos de empaquetado y minificación de código; Optimización de imágenes para reducir su tamaño.

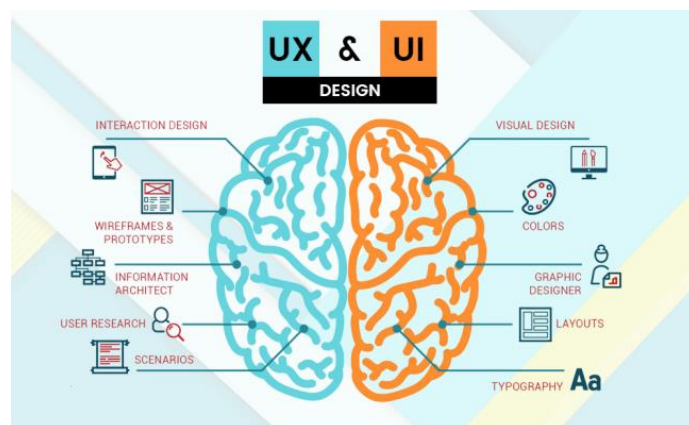
Accesibilidad: Diseñar y desarrollar interfaces accesibles para todos los usuarios.

Usabilidad: Mejorar la experiencia del usuario; Evitar ambigüedades y organizar bien los elementos en pantalla.

Herramientas de edición de imágenes: Editar y optimizar recursos gráficos.

Diseño de interfaces de usuario (User Interface): Crear interfaces atractivas y funcionales.

Optimización para motores de búsqueda (SEO): Implementar prácticas para mejorar el posicionamiento en buscadores.



Tema 2: HTTP

HTTP es el **protocolo** de **la capa de aplicación** usado para transmitir documentos de hipertexto como HTML. Facilita la comunicación cliente-servidor, principalmente a través **de transacciones de petición-respuesta**.

HTTP es un protocolo **sin estado** (**Stateless**), lo que significa que **no guarda información** sobre **conexiones anteriores**. La respuesta que da a un cliente previo es la misma que la que da a uno recién llegado.

Esto supone **ventajas** como:

Escalabilidad: Facilita manejar múltiples usuarios simultáneamente.

Menos complejidad: No necesita almacenar el estado de las sesiones.

Mayor rendimiento: Reduce la sobrecarga en el servidor.

Cacheo de recursos: Mejora la velocidad de acceso a recursos repetidos.

Y **desventajas** como:

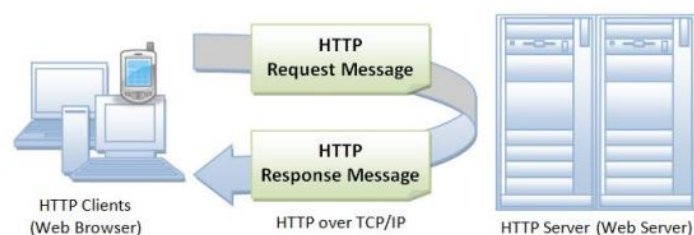
Interacción con el usuario: Se vuelve más complicada al no recordar estados previos.

Información adicional: Requiere mecanismos como cookies o tokens para mantener sesiones.

Vulnerabilidad a ataques: Es más susceptible a ataques como DDoS.

Funcionamiento básico:

1. El cliente abre una conexión (generalmente usando TCP; aunque es compatible con cualquier capa de transporte fiable (ej., RUDP)).
2. Inicia una petición HTTP (request).
3. El servidor responde con una respuesta HTTP (response).
4. La conexión se cierra.



Conceptos:

Un **User-Agent** es una cadena de texto, que **identifica** a un cliente que **realiza** una **solicitud HTTP**. Su función es dotar de identidad al cliente, describiendo el tipo de cliente, su versión, el SO, e información adicional. Trabajar con ellos, permite: Personalizar el contenido (los **navegadores** suelen **enviar esta** información, para que el **servidor** pueda **entregar una versión del contenido apropiada**, por ejemplo al dispositivo a través del cual se conectan) y analizar y controlar el tráfico.

Ejemplo de User-Agent:

Un navegador como Google Chrome podría enviar:

```
Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.110 Safari/537.36
```

Esto indica:

- **Sistema operativo:** Windows 10 (64 bits).
- **Motor de renderizado:** WebKit.
- **Navegador:** Google Chrome, versión 96.

Una **URL** (Localizador Uniforme de Recursos) es una referencia única que se utiliza para localizar un recurso en la web, como un documento HTML, imagen, vídeo, entre otros.

Su estructura:

```
scheme://[userinfo@]host[:port]/path[?query][#fragment]
```

Donde:

scheme (esquema): Es el protocolo utilizado para acceder al recurso. Como http/s (HyperText Transfer Protocol/Secure); ftp (File Transfer Protocol)...

userinfo@ (opcional): Información de usuario para autenticación, generalmente en el formato user:password. Este componente es raro en la mayoría de las URLs actuales, pero se usaba en algunos sistemas para incluir credenciales.

host: Es el nombre del servidor o la dirección IP del recurso, como por ejemplo www.ejemplo.com.

port (puerto): El puerto es el número de red que se usa para establecer la conexión. Es opcional y, si no se especifica, se usan puertos predeterminados según el protocolo.

path (ruta): Es la ruta del recurso dentro del servidor. La ruta indica la ubicación exacta del archivo en el sistema de archivos del servidor, y generalmente comienza desde el directorio raíz de la web. Ejemplo: /documentos/imagen.jpg

query (consulta) (opcional): Son parámetros adicionales enviados al servidor en formato "clave=valor", separados por & si hay varios. Ejemplo: ?q=busqueda&lang=es.

Se usan para especificar detalles adicionales en la solicitud (como filtros o términos de búsqueda)

fragmento (fragment) (opcional): Es una referencia a una sección específica dentro del recurso, como un ancla dentro de una página web. No es enviado al servidor, solo es utilizado para navegación dentro del documento. Ejemplo: #seccion1.

URL – Ejemplo

```
http://google.es:8080/ejemplo/index.html?q=text#something
```

- schema: http
- host: google.es
- port: 8080
- authority: google.es:8080
- path: ejemplo/index.html
- query: q=text
- fragment: something

URI (Uniform Resource Identifier): Es un **identificador genérico** utilizado para identificar un recurso en la web, **sin** especificar necesariamente **cómo acceder a él**.

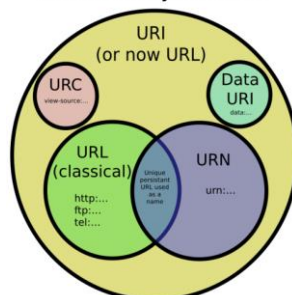
URL: Es un tipo de URI que no solo **identifica el recurso**, sino que **también** especifica **cómo localizarlo**, es decir, proporciona la dirección exacta para acceder al recurso en la web (por ejemplo, <http://www.ejemplo.com>).

URN (Uniform Resource Name): Es otro tipo de URI que identifica de manera única a un recurso sin proporcionar información sobre cómo acceder a él. Ejemplos:

- urn:isbn:0451450523: Identifica un libro de manera única por su número ISBN.
- urn:ISSN:0167-6423: Identifica una publicación periódica con su número ISSN.

URC (Uniform Resource Characteristic): Se refiere a los **metadatos de un recurso**, proporcionando información sobre el recurso en lugar de su localización o nombre. A menudo se usa para describir las características del recurso, como autor, fecha de creación, etc.

Venn diagram of URIs
as defined by the W3C



Petición HTTP

Una petición HTTP sigue este formato:

```
Método SP URL SP Versión HTTP CRLF (nombre-cabecera:valor-  
cabecera(,valor-cabecera)*CRLF)* CRLF Cuerpo del mensaje
```

SP (espacio en blanco): Separador entre los componentes.

CRLF (retorno de carro): Para finalizar una línea (carácter de nueva línea en HTTP).

*/): Indican que algunos elementos son opcionales o pueden repetirse.

Componentes de la Petición HTTP:

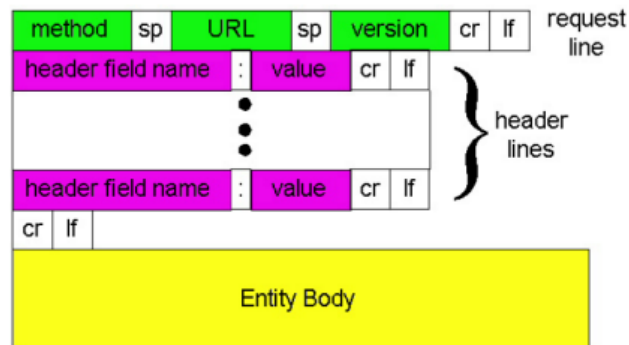
Método: Es la acción que se va a realizar sobre el recurso (ej. GET, POST).

URL: Es la dirección del recurso que se está solicitando.

Versión de HTTP: Especifica qué versión del protocolo HTTP se está utilizando (ej. HTTP/1.1).

Cabeceras: Se incluyen información adicional sobre la petición, como el tipo de contenido, tamaño, etc. Se escriben en formato nombre-cabecera: valor-cabecera.

Cuerpo del mensaje: Contiene los datos (solo si es necesario, como en el caso de POST o PUT).



Métodos HTTP: También llamados verbos, son las acciones que se realizan sobre un recurso. Algunos de los más comunes son:

GET:

Obtiene un recurso. No modifica nada en el servidor. Método obligatorio.

HEAD:

Similar a GET, pero solo obtiene las cabeceras, sin el contenido del recurso. Usado para obtener meta-información como el tamaño o la versión del recurso.

POST:

Envía datos al servidor para ser procesados. Los datos se incluyen en el cuerpo de la petición. Puede crear o actualizar un recurso en el servidor.

PUT:

Envía un recurso completo al servidor. Crea una nueva conexión para transmitir el recurso, lo que es más eficiente que enviarlo dentro del cuerpo de la petición.

DELETE:

Elimina el recurso especificado en la URL.

TRACE:

Solicita que el servidor devuelva un mensaje de respuesta. Usado para diagnosticar problemas en la conexión.

OPTIONS:

Solicita al servidor los métodos HTTP que soporta para un recurso específico.

CONNECT:

Utilizado para convertir una conexión existente en una conexión encriptada (HTTPS).

PATCH:

Modifica parcialmente un recurso existente en el servidor.

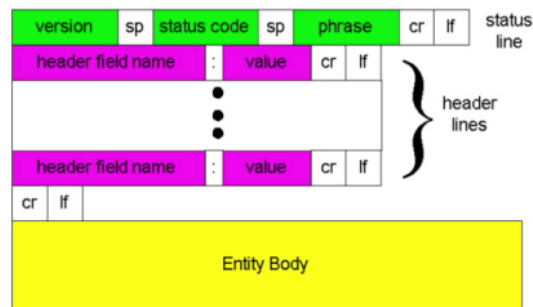
Categorías de Métodos HTTP:

Métodos seguros (Safe Methods): Son aquellos que **no** deberían **cambiar el estado** del **servidor**, solo recuperan información. GET es un ejemplo, aunque puede usarse para enviar comandos, lo que no es recomendable.

Métodos idempotentes (Idempotent Methods): Hacen que **múltiples peticiones** del mismo tipo **tengan el mismo resultado**. Los safe methods deberían ser también idempotentes, aunque una mala implementación podría violar esta regla.

HTTP method	RFC	Request has Body	Response has Body	Safe	Idempotent	Cacheable
GET	RFC 7231	Optional	Yes	Yes	Yes	Yes
HEAD	RFC 7231	Optional	No	Yes	Yes	Yes
POST	RFC 7231	Yes	Yes	No	No	Yes
PUT	RFC 7231	Yes	Yes	No	Yes	No
DELETE	RFC 7231	Optional	Yes	No	Yes	No
CONNECT	RFC 7231	Optional	Yes	No	No	No
OPTIONS	RFC 7231	Optional	Yes	Yes	Yes	No
TRACE	RFC 7231	No	Yes	Yes	Yes	No
PATCH	RFC 5789	Yes	Yes	No	No	No

Respuesta HTTP



Códigos de estado: (Y frase que acompaña/explica el código)

Son 3 dígitos que forman parte de las respuestas HTTP; explican qué ha sucedido al intentar llevar a cabo una petición. Y hay varias categorías/niveles:

- 1XX → Mensajes/Información
- 2XX → Operación realizada con éxito
- 3XX → Redirección
- 4XX → Error del cliente
- 5XX → Error del servidor

Algunos ejemplos:

- 400 Bad Request: Petición malformada o inválida
- 401 Unauthorized: El cliente tiene que registrarse
- 403 Forbidden: El cliente no tiene los derechos de acceso
- 404 Not Found: No existe el recurso indicado
- 301 Moved Permanently: contiene la nueva URL
- 302 Found: El cambio es temporal
- 502 Bad Gateway
- 503 Service Unavailable: pej. sobrecargado o en mantenimiento
- 504 Gateway Timeout
- 201 Created: Petición OK y se ha creado un nuevo recurso
- 202 Accepted:
- 100 Continue: Conexión aceptada, continua la petición
- 101 Switching Protocols: Cambiando protocolos
- 102 Processing: Procesando todavía la petición

HTTPS (Hypertext Transfer Protocol Secure) es una **extensión segura de HTTP**, destinada a garantizar la transferencia segura de datos entre el cliente y el servidor. Utiliza **cifrado basado en SSL/TLS** para proteger la privacidad e integridad de la información transmitida y, generalmente, opera a través del puerto **443**.

Postman es una plataforma de desarrollo para APIs que permite enviar peticiones HTTP, realizar pruebas (testing) y simular endpoints. Originalmente era una extensión del navegador, pero ahora funciona como una aplicación independiente (standalone), aunque también se puede utilizar desde el navegador.

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

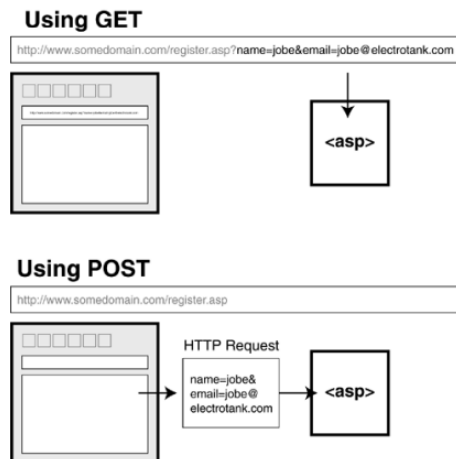
Void Elements o Elementos vacíos: son etiquetas que no tienen contenido, ni se corresponden con otra etiqueta de cierre (ni siquiera es necesario colocarles la “/” de cierre).

Algunos *void elements* y sus usos:

- insertar imagen en el doc
-
 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> vincula recursos externos como hojas CSS
- <source> especifica recurso multimedia para audio o video

Métodos GET y POST:

Ambos métodos de comunicación con el servidor, que se usan esencialmente para enviar y recibir datos. Por un lado el **GET** es útil cuando se necesita enviar más información (límite de aprox. 2000 caracteres) y que **no** sea **sensible** (ya que los **datos** son **visibles** en la **URL**) (comunmente usado para búsquedas, filtros, navegación...); mientras que por otro lado el **POST** ofrece una **capa** un poco más decente de **seguridad** (ya que los datos no se ven en la URL, van “ocultos” en el cuerpo de la solicitud), aunque **no** puede enviar **grandes cantidades** de información (comúnmente usado para formularios, inicios de sesión, carga de archivos...).



Atributos y Estilos

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**

Caracteres especiales

Secuencia de escape	Caracteres
<	<
>	>
&	&
"	"
á é í ó ú	á, é, í, ó, ú
ñ	ñ
¿	¿
 	(Espacio no separador)

Estándar de accesibilidad ARIA

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 globales:

Son atributos comunes a todo el HTML. Como:

Event Handlers → Son atributos usados para ejecutar código JavaScript cuando ocurre un evento en un elemento. Como:

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()">`.

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.

Otros 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 4: CSS (Cascading Style Sheets)

Es un **lenguaje** utilizado para definir la presentación de un documento web, incluyendo su disposición (*layout*), colores, fuentes, y ajustes específicos. Permite que múltiples documentos compartan el mismo estilo y utiliza un **sistema** de **prioridad** para decidir qué regla aplicar, se le conoce como **cascading**. Es gestionado por la W3C y su última versión el CSS3, una versión modular, que constituye un “superset” de sus versiones anteriores.

Funcionamiento:

Hay un **selector** que indica los elementos del HTML que se verán afectados. (Ejemplo: h1)

Y la **declaración** comprendida entre “{ }” que contiene una serie de elementos propiedad-valor, separados “;”. (Ejemplo: color: red; text-align: left)

(Ejemplo) h1 {color: red; text-align: left}

Tipos de selectores:

Selectores Simples:

Basados en el **nombre** de la etiqueta/elemento:

\$ p {color: red; text-align: left} → se aplica a todos los elementos p (párrafos)

Basados en el atributo **ID**: Ya que es un identificar único, afectará solo a ese elemento

\$ #paragraph1 {color: red; text-align: left} → donde → <p id=“paragraph1”>Hello World</p>

Basados en el atributo **Class**: para comprender a varios elementos

\$.center {color: red; text-align: left} → donde → <p class=“center”>Hello World</p>

También es configurable para que afecte solo a un tipo de elemento de la clase (colocando el nombre de la etiqueta, antes del “.”).

\$ p.center {color: red; text-align: left} → donde → <p class=“center”>Hello World</p> → pero no afectaría a → <h1 class=“center”>This</h1>

Selector Universal *: que afecta a todos los elementos del HTML.

\$ * {color: red; text-align: left}

Este tipo de selectores simples, también admite la agrupación de los mismos, por elementos, si hay varios que comparten diseño:

\$ h1, h2, p {color: red; text-align: left}

Selectores Combinados: Permiten relacionar elementos en función de su jerarquía, o posición en el árbol del DOM.

Selector Descendiente:

```
div p {  
  background-color: yellow;  
}
```

Esto aplica un fondo amarillo a todos los `<p>` que se encuentren dentro de un `<div>`, sin importar cuán anidados estén.

Selector Hijo:

```
div > p {  
  background-color: yellow;  
}
```

Aquí, solo los `<p>` que sean hijos directos de un `<div>` tendrán el fondo amarillo. No afecta a los `<p>` más profundamente anidados.

Selector Hermano Adyacente:

```
div + p {  
  background-color: yellow;  
}
```

Esto afecta solo al primer `<p>` que aparezca inmediatamente después de un `<div>`.

Selector Hermano General:

```
div ~ p {  
  background-color: yellow;  
}
```

Esto aplica el fondo amarillo a todos los `<p>` que sean hermanos del `<div>` y que aparezcan después de él.

(Jerarquías)

```
<div>Este es un div</div>  
<p>Este párrafo será amarillo.</p> <!-- Hermano adyacente del div -->  
<p>Este no será amarillo.</p> <!-- No es adyacente al div -->  
<div>  
  <p>Este párrafo no será amarillo.</p> <!-- Está dentro del div -->  
</div>
```


Selectores pseudoclasses: Aplican **estilos en estados** específicos de los elementos.

Con sintaxis: `selector:pseudo-class { }`

→ Sea la *pseudo-class* alguna como: “:hover”: Estilo para el elemento cuando el ratón está encima. “:focus”: Aplica estilo al elemento que tiene el foco (como un campo de entrada activo). “:first-child”: Selecciona el primer hijo dentro de su padre. “:last-child”: Selecciona el último hijo dentro de su padre.

Selectores pseudoelementos: Seleccionan partes específicas de un elemento.

Sintaxis: `selector::pseudo-element { }`

Ejemplos:

- `::after`: Inserta contenido después del elemento (requiere content).
- `::before`: Inserta contenido antes del elemento.
- `::first-letter`: Selecciona la primera letra del texto.
- `::first-line`: Selecciona la primera línea de texto.
- `::selection`: Aplica estilo a la parte seleccionada por el usuario.

Selectores de Atributo: Estilos basados en atributos de los elementos HTML.

Sintaxis básica: `selector[attribute] { }`

- Ejemplo: `a[target] { background-color: yellow; }` (aplica a enlaces con el atributo target).

Selectores con valores específicos:

Sintaxis avanzada: `selector[attribute="value"] { }`

- Ejemplo: `a[target="_blank"] { background-color: yellow; }` (aplica a enlaces que abren en nueva pestaña).

Declaraciones Se refiere al modelo de caja:

- **Contenido:** El contenido de la caja como texto o imágenes
- **Padding:** Área vacía alrededor del contenido. Es transparente
- **Borde:** Borde alrededor del padding y del contenido
- **Margen:** Área vacía fuera del borde. Es transparente



Más detallitos acerca de valores en las diapos

Especificidad (*Specificity*): Cuando múltiples selectores afectan a un mismo elemento, se utiliza la especificidad para decidir cuál regla tiene prioridad. Cuanto más específico sea el selector, mayor será su prioridad.

! > last > Id > classes > tags

El *!important* sobreescribe todas las reglas sobre un valor.

Last se refiere, a que si al final coincide que un elemento tiene más de una etiqueta, y ambas tienen la misma especificidad, se le aplicará la propiedad del último estilo (entre las coincidencias) que aparezca en la lista.

Agregar el CSS al proyecto:

Para insertar CSS como un archivo externo:

```
$ <head> <link rel="stylesheet" href="mystyles.css"> </head>
```

Para insertarlo directamente: (Es mejor trabajar con un archivo externo)

```
$ <head> <style> ... </style> </head>
```

Para insertarlo en la línea: (NO se debería usar)

```
$ <p style="color:red;"> Párrafo </p>
```

Frameworks de CSS:

Son librerías que facilitan el desarrollo web al proporcionar un conjunto de herramientas predefinidas basadas en CSS, que a menudo incluyen también JS. Sus características incluyen el reseteo de la hoja de estilos, la disposición en forma de *grid* (para que sea *responsive*), un enfoque *mobile first*, soporte, tipografía, fuentes e íconos.

Aunque son útiles, requieren de tiempo para aprender a trabajar con el framework; contienen mucho código adicional que no se termina usando; las webs pueden parecerse entre todas si coinciden; le genera a la lógica de negocio y función de la aplicación dependencia al framework.

Algunos frameworks como Ulkit, Foundation, Bulma, Materialize, Semantic UI, tailwindcss, bootstrap.

Recomendaciones de diseño:

Usar 12 columnas: Se puede dividir en columnas de 1, 2, 3, 4 y 6.

Usar líneas de unos 70 caracteres para mejorar la legibilidad.

Usar paletas de colores accesibles (4% de daltónicos).

Pre-Procesadores de CSS:

Less y **Sass** son preprocesadores que facilitan el desarrollo de estilos en proyectos grandes y complejos, que se convierten en CSS estándar mediante herramientas de compilación. Ambos ofrecen características avanzadas como:

- Variables: Reutilizar valores como colores o tamaños.
- Nesting: Anidar selectores para reflejar la jerarquía HTML.
- Módulos: Dividir estilos en archivos reutilizables.
- Mixins: (Solo en Sass) Reutilizar fragmentos de código con parámetros.

Diferencias clave:

- Less: Basado en JavaScript, más simple y ligero. Inspirado en Sass.
- Sass: Basado en Ruby, ideal para proyectos grandes, con más funcionalidades avanzadas.

Responsive Web Design:

Consiste en ese diseño que responde de manera cómoda y se adapta al dispositivo que la renderiza. Depende en gran medida de la etiqueta <meta> y su atributo de nombre "viewport":

```
$ <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Media Queries:

Son elementos/etiquetas/características de CSS que permiten aplicar estilos, dependiendo de las características (como resolución, orientación, ancho-alto) del dispositivo o de las ventanas que renderizan la página. Esto es parte de crear diseños "responsivos".

Son etiquetas que van al final del CSS, y se ven como:

Mediatype: para las pantallas (screen)

Mediafeature: para una característica (max-width o min-width)

```
$ @media not | only mediatype and (mediafeature and | or | not mediafeature) {  
  CSS-code;}
```

Normalmente se enlazarán como 2 links diferentes de CSS:

```
<link rel="stylesheet" media="screen and (min-width: 900px)"  
  href="widescreen.css">  
<link rel="stylesheet" media="screen and (max-width: 600px)"  
  href="smallscreen.css">
```


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 cargado 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)



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*.

Callback:

Son **funciones**, que **se pasan como argumentos a otras funciones**. Se invocan cuando se complete alguna acción o rutina (como una llamada a un API, o el uso de temporizadores). Habitualmente se usan de forma asíncrona, pero, pueden ser síncronas.

Strict Mode:

Es un modo de trabajo/de desarrollo voluntario, en el que JS impone de manera estricta reglas de sintaxis y semántica para garantizar que el programador trabaje de forma limpia, esto promueve la robustez del código y la minimización de errores.

Se activa al inicio del documento JS con: (o dentro de bloques/funciones)

```
"use strict";
```

Sí se utiliza de forma automática al trabajar con módulos y clases.

Sloppy Mode:

Es la "contraparte" del modo estricto de trabajo. Es el modo predeterminado, que suele ser más permisivo.

Módulos:

Trabajar con módulos en JS, significa trabajar (en *strict mode*) separando en el código bloques que se puedan importar y reutilizar. Utilizan declaraciones de *import* y *export*. Algunas bibliotecas y frameworks se basan en módulos (que son archivos .js o .mjs).

Los módulos se caracterizan por ser de única ejecución, y de importación limitada, es decir, no se pueden importar módulos en scripts regulares. Con respecto a su alcance, las características que se importan no están disponibles para el ámbito global, aunque el módulo, sí tiene acceso a variables globales.

Se pueden integrar al HTML como:

```
$ <script type="module" src="main.js"> </script>
```

No es necesario agregar *defer*, ya que los módulos manejan automáticamente la carga del HTML, aunque el orden de ejecución dependerá de las dependencias entre los módulos.

Ejemplos de exportación:

```
// Exportación directa
export const name = "square";
export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);
  return { length, x, y, color };
}
```

O agrupadas al final del archivo:

```
javascript

const name = "square";
function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);
  return { length, x, y, color };
}

export { name, draw };
```

Ejemplos de importación:

1. Importación específica:

Puedes importar elementos individuales:


```
javascript  Copiar código

import { name, draw } from './modules/square.js'
console.log(name);
```

2. Renombrar elementos:

Es posible renombrar elementos importados usando

as :

```
javascript  Copiar código

import { name as nombre, draw } from './modules/square.js'
console.log(nombre);
```

3. Importación completa:

También puedes importar todo el contenido de un módulo dentro de un objeto:

```
javascript  Copiar código

import * as Square from './modules/square.js';
console.log(Square.name);
```


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
```

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 éstas 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. *

Variables de entorno son valores globales que afectan a toda la aplicación. Son útiles para manejar configuraciones que pueden variar según el entorno de ejecución (como producción, desarrollo o pruebas).

Node permite acceder a las variables de entorno a través de un módulo *process* que está disponible por defecto.

```
console.log(process.env.USER_ID);
```

Se usan comúnmente para tareas, como establecer el puerto de ejecución.

Aunque se pueden definir en el programa o cargar desde un archivo “.env” con el módulo *dotenv* (que es ideal para configuraciones sensibles como claves de API):

```
$ npm install dotenv
$ vim .env
  USER_ID="239482"
  USER_KEY="foobar"
  NODE_ENV="development"
→ En un formato clave valor.
```

```
require('dotenv').config();
console.log(process.env.USER_ID); // "239482"
console.log(process.env.NODE_ENV); // "development"
```

También se pueden definir y poner en acción, al ejecutar el programa:

```
USER_ID=239482 USER_KEY=foobar node app.js
```

Es decir pasarlos como **argumentos**. Estos se almacenan en un array accesible a través de *process.argv*. es un array que sigue el “convenio UNIX” (donde, p[0]= ruta completa del ejecutable node; p[1]= ruta completa del archivo en ejecución; p[2-n]= argumentos proporcionados por el usuario):

```
node index.js joe smith

javascript Copiar código

console.log(process.argv);
// Salida: ['/ruta/a/node', '/ruta/a/index.js', 'joe', 'smith']
```

Aunque también se pueden pasar argumentos, con respectiva clave-valor (solo que habrá que parsearlos manualmente):

```
bash Copiar código

node index.js name=joe surname=smith

javascript Copiar código

const args = process.argv.slice(2); // ['name=joe', 'surname=smith']
const parsedArgs = Object.fromEntries(args.map(arg => arg.split('=')));
console.log(parsedArgs);
// Salida: { name: 'joe', surname: 'smith' }
```

Otra última opción para el tratamiento de argumentos más complejos, es el módulo *minimist* que permite:

\$ npm install minimist

```
bash Copiar código

node index.js --name=joe --surname=smith -x 1 -y 2 -abc

javascript Copiar código

const minimist = require('minimist');
const args = minimist(process.argv.slice(2));

console.log(args);

// Salida:
// {
//   _: [],           // Argumentos posicionales
//   name: 'joe',     // Clave-valor
//   surname: 'smith',
//   x: 1,            // Opciones abreviadas
//   y: 2,
//   a: true,         // Flags individuales
//   b: true,
//   c: true
// }
```

Process es el objeto global que permite interactuar y capturar la información general de la ejecución de Node.

```
process.memoryUsage() // Return an object with memory usage details.
process.nextTick() // Invoke a function soon.
process.pid // The process id of the current process.
process.platform // The OS: "linux", "darwin", or "win32", for example.
process.resourceUsage() // Return an object with resource usage details.
process.uptime() // Return Node's uptime in seconds.
process.version // Node's version string.
process.versions // Version strings for the libraries Node depends on.
```

```
process.argv // An array of command-line arguments.
process.arch // The CPU architecture: "x64", for example.
process.cwd() // Returns the current working directory.
process.cpuUsage() // Reports CPU usage.
process.env // An object of environment variables.
process.execPath // The absolute filesystem path to the node executable.
process.exit() // Terminates the program.
process.exitCode // An integer code to be reported when the program exits.
process.kill() // Send a signal to another process.
```

OS es el módulo que da acceso a bajo nivel sobre el SO, debe ser cargado:

```
const os = require("os");
os.cpus() // Data about system CPU cores, including usage times.
os.freemem() // Returns the amount of free RAM in bytes.
os.homedir() // Returns the current user's home directory.
os.hostname() // Returns the hostname of the computer.
os.loadavg() // Returns the 1, 5, and 15-minute load averages.
os.networkInterfaces() // Returns details about available network. connections.
os.release() // Returns the version number of the OS.
os.totalmem() // Returns the total amount of RAM in bytes.
os.uptime() // Returns the system uptime in seconds.
```

REPL (read-evaluate-print-loop): Es un entorno interactivo incluido en NodeJS que permite ejecutar código JS en una consola. Se caracteriza por su sencillez que con *read* lee el input del usuario, *evaluate* procesa el código ingresado, *print* muestra el resultado y *loop* permite repetir el proceso. Se accede con “\$node” en la terminal.

Ficheros

Path: módulo con utilidades de ficheros y rutas, que hay que importar.

```
$ const path = require("path");
```

HTTP es uno de los módulos/librerías que primero vamos a importar:

```
$ const http = require('http');
```

Ya que nos permite trabajar con funcionalidades diversas, como operaciones de I/O en archivos (fs), permite parsear y trabajar con *paths* y URL'S (path y url).

Event Loop

Es un componente fundamental de ejecución de Node.js, ya que le permite manejar múltiples operaciones de manera **asíncrona en un solo hilo**, haciendo que sea eficiente para aplicaciones que requieren manejar muchas conexiones o tareas concurrentes.

Características clave del Event Loop:

Un Único hilo: Node.JS utiliza un único hilo para ejecutar JS, esto significa que no existen múltiples hilos ejecutando código en paralelo. Aunque hay operaciones asíncronas como I/O, timers, o conexiones de red, que se delegan a hilos “de fondo”, que se gestiona con la biblioteca **libuv**.

Repetitivo (Loop): Como su nombre lo indica consiste en un bucle/ciclo que se ejecuta de manera continua siempre y cuando haya trabajo pendiente (callbacks, eventos, tareas programadas).

Cada iteración es un Ticks: cada iteración del Event Loop es un Tick; y en cada Tick Node.JS procesa las tareas pendientes en las colas correspondiente.

Una cola FIFO de callbacks: Los callbacks (también interpretados como “tareas pendientes” se colocan en una FIFO. En cada Tick en Event Loop procesa estos callbacks hasta que la cola queda vacía o alcance un límite de tiempo.

Gestionado por Libuv: es la biblioteca que implementa el Event Loop de Node.JS, proporciona abstracciones para operaciones de E/S no bloqueantes y mmultiprocesamiento.

Funcionamiento del Event Loop:

Un Event Loop cuenta con varias fases, y cada una, maneja un conjunto específico de tareas. Node utiliza este mecanismo para gestionar las operaciones asíncronas y no bloqueantes.

1. Timers: En esta fase se ejecutan funciones asociadas a temporizadores creados con:
\$ setTimeout()
\$ setInterval()
2. Pending Callbacks: Aquí se procesan los callbacks de operaciones que hayan sido delegadas al sistema operativo (como operaciones de E/S).
3. Poll: Procesa las operaciones de E/S, el Event Loop espera en esta fase, hasta que hayan callbacks o un temporizador expire.
4. Check: Aquí se ejecutan los callbacks asociados con setImmediate(). setImmediate() siempre se ejecuta después de completar la fase Poll, independientemente del orden de definición.
5. CloseCallbacks: finalmente se ejecutan todos los callbacks asociados a cierres (como \$ socket.on('close');).

Es un evento que **JAMAS podemos bloquear**, con Ticks (iteraciones) cortos, donde el trabajo asociado será corto (dividiendo las tareas intensivas).

Un ejemplo (no de clase, pero sobre el Event Loop):

```
javascript

console.log('Inicio'); // (1)

setTimeout(() => {
  console.log('Timeout'); // (4)
}, 0);

Promise.resolve().then(() => {
  console.log('Promise'); // (3)
});

console.log('Fin'); // (2)
```

Resultado final en la consola:

```
javascript Copiar código

Inicio
Fin
Promise
Timeout
```

¿Qué enseña este ejemplo?

1. Prioridad del Event Loop:

- Las **Microtare**as (como promesas) tienen mayor prioridad que las **Tare**as normales (como `setTimeout`).

2. Orden de ejecución:

- Lo síncronico se ejecuta primero (directamente en el Call Stack).
- Las promesas se ejecutan después (Microtask Queue).
- Finalmente, las callbacks de `setTimeout` u otros se ejecutan en la Task Queue.



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, autenticación, 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)

En Express un **template Engine** (motor de plantillas): es la herramienta que **permite generar contenido dinámico** en las **vistas**. La idea consiste en tener **plantillas estáticas** (las `/views/<file>.ejs`) que luego se **“rellenan”/completan** con **datos dinámicos** (con la sintaxis del tipo `<%= datoDinámico %>`), en tiempo de ejecución, generando el HTML que se envía finalmente al navegador del cliente.

Adicionalmente Express soporta varios motores de plantilla como Pug, Handlebars, y EJS (Embedded JS) (que es con lo que trabajamos).

Para trabajar con Express y su motor de plantillas base:

```
$ npm install ejs
```

Para configurarlo como motor de plantillas a usar:

```
//app.js --> main file of the application (launching point)
const express = require("express");
const app = express(); // app is an instance of express
app.set('view engine', 'ejs'); // This sets the view engine to ejs
app.set('views', './views');
```

```
**{
```

Si se genera automáticamente con el motor de Express, se verá algo como:

```
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');
```

```
}**
```

Y las referencias correspondientes a las rutas del resto de páginas:


```
app.locals.title = "Mi Título de Aplicación GLOBAL"; //Establishing a global var to be used in the views
let indexRouter = require('./src/routes/index');
app.use('/', indexRouter);
```

Y en el resto de rutas JS (/routes/<file>.js), agregaremos lo siguiente, que reutiliza la configuración principal:

```
1  const express = require("express");
2  const router = express.Router();
3
4  router.get("/", (req, res) => {
5    |   res.render('index')
6  | });
7
8  module.exports = router;
```

Steps taken:

npm install -g express-generator

npx express-generator -v ejs <src>

```
express_test/
├─ app.js
├─ package.json
├─ /bin
│   └─ www
├─ /public
├─ /routes
│   ├── index.js
│   └─ users.js
└─ /views
    ├── error.ejs
    └─ index.ejs
```

```
//This is the way to serve static files in express
app.use(express.static(path.join(__dirname, 'public')));
```

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

- ```
app.METHOD(path, callback [, callback ...])
```
- METHOD: get, post, delete... ([lista completa](#))
  - path: String, patrón, expresión regular...
  - callback: Una o más funciones

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

\$ *res.locals* → es un objeto específico para cada petición, que “vive” durante el ciclo de la misma; almacena variables que están disponibles en las vistas renderizadas, durante esa única petición. Se usan regularmente para datos temporales o dinámicos que dependen del contexto de la petición (como mensajes de error o datos de usuario autenticado).

```

src > JS app.js > ...
25 var backdoorRouter = require('./routes/backdoor');
26 app.use((req, res, next) => {
27 // res.locals.currentUser = req.user || null;
28 res.locals.currentUser = "orianna";
29 next();
30 });
31 app.use('/backdoor', backdoorRouter);
32 |

src > views > <> backdoor.ejs > ...
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <meta name="viewport" content="width=device-width, initial-scale=1.0">
6 <title>Backdoor</title>
7 </head>
8 <body>
9 <h1>Bienvenido al Backdoor</h1>
10 <p>Usuario actual: <%= currentUser %></p>
11 Volver al inicio
12 </body>
13 </html>
14 |

```

\$ app.locals → es un objeto global para toda la aplicación, y las variables definidas en este objeto se mantienen disponibles en todas las vistas durante la toda la vida/ejecución de la aplicación.

```

JS app.js > ...
11 app.locals.title = "Mi Título de Aplicación GLOBAL";
12 |

src > routes > JS index.js > ...
1 const express = require("express");
2 const router = express.Router();
3
4 router.get("/", (req, res) => {
5 res.render('index')
6 });
7
8 module.exports = router;

JS app.js M <> index.ejs M ● JS index.js U

src > views > <> index.ejs > ...
1 <h1><%= title %></h1>

```

# Login

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:

- A. Rotación de los Logs: Rotar los logs después de cierto tamaño o edad.
- B. 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.
- C. Compresión de Logs
- D. 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?

| <code>console.log()</code>                                      | Logging                                                               |
|-----------------------------------------------------------------|-----------------------------------------------------------------------|
| No se puede desactivar. Tendríamos que borrar todas las líneas. | Se puede activar/desactivar                                           |
| No es granular. Se imprime todo.                                | Se pueden usar distintos niveles y solo imprimir los que nos convenga |
| Se mezcla con otra información en la consola                    | Se distingue en la consola                                            |
| No es persistente                                               | Podemos usar archivos u otros soportes                                |
| No hay más información que la que añadimos nosotros             | Se pueden añadir metadatos                                            |

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
```

```
const logger = require('morgan'); //import the logging library

// This checks the environment of the application and sets the logging format accordingly
// if (app.get('env') == 'production') {
// app.use(logger('common', {
// skip: function (req, res) { return res.statusCode < 400 },
// stream: __dirname + '/../morgan.log'
// }));
// } else {
// app.use(logger('dev'));
// }
//app.use(logger('common')); // Is the format when the project is in production stage
app.use(logger('dev'));
```

*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 install winston
```

```
//app.js
// const logger = require('./utils/logger'); // Asegúrate de poner la ruta correcta
// const app = express();

// // Ejemplo de uso del logger
// app.get('/', (req, res) => {
// logger.info('Solicitud GET a la ruta principal');
// res.send('Hello World!');
// });

// app.listen(3000, () => {
// logger.info('Servidor corriendo en http://localhost:3000');
// });
```

```
JS logger.js > ...
1 const winston = require('winston');
2
3 const logger = winston.createLogger({
4 level: 'info', // Nivel mínimo de logs que se registrarán.
5 format: winston.format.json(), // Los logs se registrarán en formato JSON.
6 defaultMeta: { service: 'user-service' }, // Metadatos por defecto en cada log.
7 transports: [
8 new winston.transports.File({ filename: 'error.log', level: 'error' }), // Logs de error en `error.log`.
9 new winston.transports.File({ filename: 'combined.log' }) // Todos los logs en `combined.log`.
10],
11 });
12
13 // Si no es producción, añade la consola como transporte adicional.
14 if (process.env.NODE_ENV !== 'production') {
15 logger.add(new winston.transports.Console({
16 format: winston.format.simple(), // Formato simple para la consola.
17 }));
18 }
19
20 module.exports = logger;
```

*Pino ...*

# SOCKET.IO

Es una **biblioteca** de JS que permite la **comunicación en tiempo real**, entre **clientes** y **servidores**, por lo que se dice que es **bidireccional**. SOCKET.IO utiliza **WebSockets** que es un **protocolo** que crea una **conexión persistente** (con respuesta automática a intentos de reconexión en caso de fallos) entre el cliente y el servidor; cuenta con un sistema robusto que incorpora un sistema de **fallback**, donde si el protocolo de **WebSockets** no es soportado por el cliente o la red, puede utilizar otros métodos de comunicación alternativos (como XHR polling (envío de solicitudes HTTP, como sondeo), o encuestas de AJAX).

Es además una tecnología altamente **escalable** gracias a que soporta las **conexiones concurrentes**, y es compatible con otras herramientas, tales como Redis (que permite compartir información entre servidores).

Entre su catálogo, SOCKET.IO permite trabajar (enviar y recibir) **eventos personalizados** (entendiendo a un evento como un mensaje, que lleva un nombre y datos asociados).

La instalamos con:

```
$ npm install socket.io
```

//app.js

```
const { createServer } = require("http");
const { Server } = require("socket.io");

var chatRouter = require('./routes/chat'); // Chat client logic
const chatSocket = require("./socket/chat"); // Chat server logic

const httpServer = createServer(app); // Crear servidor HTTP
const io = new Server(httpServer); // Inicializar Socket.IO

chatSocket(io); // Inicializar el servidor de chat
```

//socket/chat.js (Lógica básica del Servidor)

```
//Detectar la conexión de un nuevo cliente
module.exports = (io) => {
 io.on("connection", (socket) => {
 console.log("Un usuario se ha conectado");

 // Obtener chat_id desde el cliente
 const chat_id = socket.handshake.query.chat_id;
 if (!chat_id) {
 console.error("Chat ID no proporcionado");
 socket.emit("error", { message: "Chat ID requerido" });
 return;
 }
 //...
 //Desconexión de un cliente
 socket.on("disconnect", () => {
 console.log("Un usuario se ha desconectado");
 });
 });
};
```

## Steps to Debug (using debug module)

---

To install debug module:

```
npm install debug
```

Also add the following line in the app.js:

```
const debug = require('debug')('appLogs');
debug('This is a verbose log');
```

To excute the code with debug logs:

```
$env:DEBUG="*"; node app.js
```



# Passport

Es un middleware de autenticación, integrable con aplicaciones basadas en Node.JS y que incorporan Express como su framework.

Ejemplo de integración básica:

```
javascript

const express = require('express');
const passport = require('passport');

const app = express();

// Middleware de inicialización de Passport
app.use(passport.initialize());
app.use(passport.session());
```

Es una herramienta versátil, ya que da soporte a diferentes [estrategias de autenticación](#), como el clásico método de [usuario y contraseña](#); como también [OAuth](#) que es el mecanismo de autenticación a través de proveedores externos, como el inicio de sesiones, con cuentas de *Google*, *Facebook*, *Twitter*, etc; finalmente otra estrategia que permite incorporar es [OpenID](#), un protocolo abierto que permite a los usuarios autenticarse, en diferentes plataformas con una única identidad.

# Tema 7: Accesibilidad

Una buena experiencia de usuario (UX) se genera con **usabilidad** (facilidad con la cual los **usuarios** pueden **interactuar** con un **sistema**, producto o interfaz, para cumplir sus objetivos de manera eficiente) + buen **diseño gráfico** (**centrado en el humano**) + **accesibilidad** (que **cualquiera**, bajo **cualquier circunstancia** pueda acceder/**interactuar** con los **recursos**).

**Diseño centrado en el humano:** involucra, que haya feedback, consistencia, intuitivo y predecible, confort, sencillez, estructura, tolerancia a fallos, límites, visibilidad, mapeo apropiado, descubrimiento...

Se considera que existe **6 niveles de madurez de UX:**

1. Ausente → 2. Limitado → 3. Emergente → 4. Estructurado → 5. Integrado → 6. User-driven

**Accesibilidad Web WCAG** (web content accessibility guidelines): (por W3C)

Es el **estándar** para la **accesibilidad** web que cuenta con **4 principios**: que se **perceptible**, **operable**, **robusto** y **entendible**. Con más de 13 guías para hacer el contenido accesible, y criterios de éxito (A-AA-AAA).

**Consejos de diseño accesible:**


- No transmitir información únicamente con color.
- Texto claro y legible.
- Variedad de formatos para el contenido.
- Diseño lineal, lógico, consistente (y de acuerdo con el idioma (de izquierda a derecha)).
- Imágenes y diagramas para dar soporte al texto.
- Tener en cuenta la navegación por teclado.
- Etc...

**Accesibilidad Web ARIA** (accessible rich internet applications):

Propone mirar por capas (de dentro hacia afuera): Contenido (semántico HTML) → Presentación CSS → Comportamiento JS → Paquetes para accesibilidad ARIA.

ARIA es una **especificación** del W3C que **proporciona** un conjunto de **roles** y **atributos** para **mejorar la accesibilidad** en webs interactivas, como aquellas que involucran tecnologías avanzadas como JS o AJAX.

```
<div
 id="percent-loaded"
 role="progressbar"
 aria-valuenow="80"
 aria-valuemin="0"
 aria-valuemax="100"></div>
```



Un ejemplo de diseño adaptado/pensado para copear con discapacidades, es:

Atkinson Hyprlegible: que es una fuente tipográfica diseñada específicamente para mejorar la legibilidad, a personas con discapacidades visuales.

“Se sostiene que el poder de la web está en su universalidad, por lo mismo, debería ser accesible para todos también”

## Cosas de talleres:

### OWASP (Open Web Application Security Project):

Es una **Fundación** sin fines de lucro dedicada a **mejorar la seguridad del software**. Es conocida por proporcionar recursos, herramientas y documentación relacionadas con ciberseguridad, especialmente en el desarrollo de aplicaciones web.

Ofrece recursos como, un Top Ten de vulnerabilidades más críticas en páginas web; Proyectos como ZAP para pruebas de seguridad; Guías, manuales y buenas prácticas.

### Bastionado de aplicaciones:

Es la **práctica** de **reforzar la seguridad** de una aplicación mediante la **implementación** de **controles** y medidas de protección (tales como, controles de acceso, cifrado, auditoría y monitorización...) que minimizan riesgos y vulnerabilidades.

### CI (Integración continua) (en contexto de Dev-Ops):

Requiere: Controles de versiones, infraestructura de ejecución de pipelines, repositorios de artefactos... Directamente relacionado con la *entrega continua*.

Algunas herramientas que ayudan a este proceso:

#### GitFlow:

Es una **estrategia de ramificación** para gestionar el desarrollo de software en proyectos que utilizan git; define un conjunto de reglas y flujos, para organizar las ramas de un repositorio.

Algunos de sus componentes/ramas clave: **master-develop-feature-release-master-hotfix-master/develop**.

## Hacer para el EXAMEN:

Usar el res.locals para las cookies o para el header del user logeado?

```
app.use((req, res, next) => {
 res.locals.currentUser = req.user || null; // Pasar el usuario actual a las vistas
 next();
});

app.get('/dashboard', (req, res) => {
 res.render('dashboard'); // 'currentUser' estará disponible en la vista
});

<% if (currentUser) { %>
 <p>Bienvenido, <%= currentUser.name %>!</p>
<% } else { %>
 <p>Inicia sesión para continuar.</p>
<% } %>
```

Más ejercicios:

Ejercicio 1 • Crea un servidor en node.js • Que al iniciar muestre por consola información del sistema y versión de node.js • De forma periódica muestre por consola la siguiente información: – Uso de CPU – Uso de memoria – Tiempo que el sistema lleva activo – Tiempo que lleva ejecutándose node.js • Que la información que se muestra periódicamente sea configurable en un fichero, incluyendo cada cuantos segundos se muestra

Ejercicio 3 • Crea un servidor en node.js • Que al cargar la página principal muestre una contraseña aleatoria • Generada a partir de X palabras aleatorias seleccionadas de un diccionario • El número de palabras (X) está definido como parámetro en la query – Accesible en req.url

(con las diapos de HTTP de 80-98)

Ejercicio 4 Web scraping • Extraer datos de páginas web • De forma automática haciendo uso de bots • Una vez extraída la información se procesa • Se repite el proceso a lo largo del tiempo para comparar la evolución de los datos • Ejemplos: – Google extrayendo información de las webs para el buscador – Internet Archive para conservar webs antiguas – Páginas de comparación de precios

## Anexos:

```
npm install ejs
npm install express
npm install express body-parser
```

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.