

Tema 1 – Introducción a las aplicaciones para Internet

1.1. ¿Qué es una aplicación para Internet?

Ideas clave:

- En general, una **aplicación para Internet** (o *web application*) es un programa que se usa a través de un **navegador web** (el navegador actúa como cliente).
- Es **interactiva**:
 - El usuario no solo lee información, sino que **envía datos**, pulsa botones, hace búsquedas, etc.
- El código suele estar en un **servidor remoto**, no instalado en tu ordenador.

Frases que puedes usar en el examen:

Una aplicación para Internet es una aplicación interactiva a la que se accede mediante un navegador. Parte se ejecuta en el cliente (navegador) y parte en el servidor, usando protocolos de Internet como HTTP.

1.2. Sistema web vs aplicación web

- **Sistema web**: la **infraestructura** que permite que funcione la aplicación web (servidores, redes, bases de datos, etc.).
- **Aplicación web**:
 - Es la **aplicación distribuida** que realiza una función de negocio.
 - Está basada en las tecnologías web.
 - Utiliza recursos específicos de la web (HTML, CSS, JS, HTTP...).

Idea: el sistema web es “el conjunto de máquinas y servicios”; la aplicación web es “el programa que hace cosas para el usuario”.

1.3. Ventajas y desventajas de las aplicaciones web

El profe dijo que aquí le interesa que sepas **ventajas y desventajas** (pregunta típica tipo “justifica por qué usarías una app web”).

Ventajas principales (lista simplificada):

- No hay que instalar nada en el cliente → ahorras tiempo de **instalación y despliegue**.
- Son **multiplataforma**: funcionan en Windows, macOS, Linux, móvil... mientras haya navegador.
- Actualizaciones **inmediatas**: actualizas el servidor y todos los usuarios ven la nueva versión.
- Consumen **pocos recursos** en el cliente (no necesitas un PC potente).
- Son **portables**: puedes usarlas desde diferentes dispositivos.
- Suelen tener **alta disponibilidad** (accesibles desde cualquier sitio con Internet).
- Son **ideales para colaboración** (todos trabajan sobre los mismos datos en el servidor).

Desventajas importantes:

- Menos funcionalidad que una aplicación de escritorio “pura” (limitaciones del navegador).
- Dependes de:
 - La **red** (si no hay conexión, mala suerte).
 - El **servidor** (si cae, deja de funcionar para todos).

- Limitaciones del propio protocolo **HTTP** (stateless, etc.).
- Cuello de botella si hay mucho tráfico (ancho de banda).
- Necesitas **criptar datos** (**HTTPS**) y cuidar la **seguridad**.

En el examen, si te preguntan “ventajas / desventajas”, con 3–4 bien explicadas por cada lado vas sobrada.

1.4. Sitio web vs aplicación web

Sitio web (website):

- Conjunto de **páginas web** bajo un mismo dominio.
- Normalmente **informativo** (poca o nula interacción).
- Ejemplos:
 - Web de la universidad.
 - Web corporativa de una empresa.
- Muchas veces se hace con un **CMS** (WordPress, etc.).

Aplicación web (web application):

- Suele ser **el propio negocio** (ej.: Netflix, banca online).
- Requiere interacción constante del usuario; el usuario suele estar **registrado y logueado**.
- Ejemplos:
 - Netflix, Facebook, Instagram, banca online, intranet del CEU.

Frase resumen para el examen:

Un sitio web es principalmente informativo y tiene poca interacción; una aplicación web es interactiva y suele ser el núcleo del negocio, donde el usuario realiza acciones (compras, pagos, ver contenido personalizado, etc.).

1.5. Aplicación web vs servicio web

Esta distinción también le gusta mucho al profe.

Aplicación web (para personas):

- Diseñada para **humanos**.
- Tiene **interfaz gráfica** (HTML/CSS).
- Comunicación **Human to Machine**.
- Habla en “páginas” web.

Servicio web (web service) (para otras aplicaciones):

- Diseñado para que lo use **otra máquina**.
- Expone **APIs** (endpoints de HTTP).
- Comunicación **Machine to Machine**.
- Envía **datos sin presentación**:
 - Normalmente en **JSON o XML**.
- La presentación la hace la app cliente (otra web, un móvil, etc.).

Frase tipo examen:

Una aplicación web tiene interfaz HTML para usuarios humanos; un servicio web expone solo datos (JSON/XML) para que los consuman otras aplicaciones, sin interfaz gráfica.

1.6. Arquitectura básica de una aplicación para Internet

Debes quedarte con la idea de las **tres capas**:

1. **Capa de acceso / presentación**
 - a. Lo que ve el usuario: navegador, GUI.
 - b. Tecnologías típicas: HTML, CSS, JavaScript.
2. **Capa del servidor / lógica de negocio**
 - a. Donde se ejecuta la lógica: servidores web, APIs, reglas de negocio.
 - b. Tecnologías: Node.js, PHP, Java, etc.
3. **Capa de persistencia**
 - a. Bases de datos y almacenamiento de información.

También aparecen modelos más complejos:

- Separar servidores para:
 - Contenido estático.
 - Contenido dinámico.
 - Bases de datos.
 - Gestión/seguridad.
- Uso de **balanceadores de carga** para repartir peticiones entre varios servidores (alto rendimiento).
- **Alta disponibilidad**: duplicar elementos para que si uno cae, el sistema siga funcionando.

En el examen normalmente no te va a pedir “dibuja todos los modelos”, pero sí:

- Saber que hay **capas**.
- Entender el concepto de **balanceador de carga y separación de funciones**.

1.7. Pequeña historia de tecnologías web (solo idea general)

De aquí no quiere fechas, pero sí idea global:

- **JavaScript** aparece en el navegador para dar interactividad a páginas estáticas.
- Hubo otros intentos de “apps ricas” en el navegador:
 - **Java applets**.
 - **Flash**.
 - **Silverlight**.
- Todos acabaron desapareciendo (seguridad, rendimiento, necesidad de plugins, etc.).
- Hoy en día el estándar es **HTML5 + CSS + JavaScript**.

Lo importante: **JS y HTML5 han sustituido a todas esas tecnologías de plugin**.

1.8. Front-end vs Back-end (muy importante)

Esta diferencia es **muy preguntable**.

1.8.1. ¿Qué hace un desarrollador Front-end?

Trabaja en la **parte visible** para el usuario, en el navegador.

Tareas que debes saber nombrar (no hace falta soltar todas, pero sí 3–4 buenas):

- **Web performance**: que la web cargue rápido y responda bien.

- **Responsive web design:** que la web se adapte a móvil, tablet, escritorio.
- **Cross-browser compatibility:** que funcione igual en Chrome, Firefox, Edge, etc.
- **Accesibilidad:** que personas con distintas capacidades puedan usar la web (alt en imágenes, etc.).
- **Usabilidad / UI / UX:** organización y diseño de la interfaz para que sea clara y fácil de usar.
- **Automatizaciones de build:** minificar, empaquetar, optimizar imágenes.

Tecnologías de front-end:

- **HTML, CSS, JavaScript.**
- Librerías: SASS, jQuery (aunque en el examen no entra mucho).
- Frameworks: **Bootstrap, React, Angular, Vue.js** (sobre todo saber que existen).

1.8.2. ¿Qué hace un desarrollador Back-end?

Trabaja en el **lado del servidor**: recibe peticiones, procesa datos, habla con la base de datos...

Tareas típicas:

- **Lógica de negocio:**
 - Decidir qué se puede hacer, qué reglas hay para crear / modificar datos.
- **Acceso a datos:**
 - Control de roles y permisos.
 - Validar que no se confía ciegamente en lo que envía el cliente.
- **Administración de bases de datos:**
 - Consultas eficientes, índices, backups.
- **Escalabilidad y alta disponibilidad:**
 - Cómo aumentar capacidad, reducir tiempos de caída.
- **Seguridad:**
 - Autenticación, autorización, cifrado, etc.

Tecnologías de back-end:

- Lenguajes de scripting: **PHP, Python, Ruby, Node.js.**
- Lenguajes compilados: **C#, Java, Go.**
- Y, por supuesto, **bases de datos.**

1.8.3. Nota terminológica

En la última diapositiva comenta que:

- “Back end” es un término muy genérico; a veces es mejor decir “server”, “database”, etc.
- “Front end” y “back end” se pueden escribir:
 - Sin guion cuando son **sustantivos**.
 - Con guion (“front-end”, “back-end”) cuando son **adjetivos** (“desarrollador front-end”).

Para el examen, con que sepas la diferencia conceptual entre **front-end** y **back-end** y qué hace cada uno, vas perfecta.

Tema 2 – HTTP

1. HTTP: qué es y cómo funciona

HTTP = *Hypertext Transfer Protocol*. Es el protocolo que usamos para enviar documentos hipermedia (HTML, imágenes, etc.) entre un **cliente** (navegador) y un **servidor**. Funciona en la **capa de aplicación** de la pila de protocolos.

Características clave:

- Es un **protocolo de petición–respuesta**:
 - El cliente abre una conexión (normalmente TCP).
 - Envía una **petición HTTP** (request).
 - El servidor responde con una **respuesta HTTP** (response).
 - Se cierra (o se reaprovecha) la conexión.
- Es **independiente de transporte**, pero en la práctica se usa casi siempre sobre TCP/IP.

Esto lo puedes resumir así en el examen:

HTTP es un protocolo de aplicación, sin estado, basado en el modelo petición–respuesta, que se usa para comunicar clientes (navegadores) con servidores web.

2. Evolución de versiones (1.0, 1.1, 2, 3)

No quieren fechas, ni porcentajes de uso. Quieren que sepas **qué aporta cada versión**.

- **HTTP/1.0**
 - Primera versión “seria”.
 - Cada recurso = **una conexión nueva** → poco eficiente (imagine HTML + 20 imágenes = 21 conexiones).
- **HTTP/1.1**
 - Introduce **conexiones persistentes**: se puede reutilizar la misma conexión para varias peticiones.
 - Soporta **host virtuales** (varios dominios en la misma IP).
 - Mejora caché, compresión, etc. (no hace falta que detallés todo, solo que mejora rendimiento respecto a 1.0).
- **HTTP/2**
 - Basado en SPDY (Google).
 - Permite **multiplexar** varias peticiones y respuestas sobre una sola conexión → muchas cosas a la vez, sin bloqueo.
 - Cabeceras comprimidas, push de recursos, etc. (idea general: mucho más eficiente).
- **HTTP/3**
 - Mantiene la **misma semántica** (mismos métodos, códigos...) pero cambia la parte de transporte.
 - Usa **QUIC**, que va sobre **UDP** en lugar de TCP.
 - Evita ciertos problemas de TCP como el **head-of-line blocking**.
 - Puede ser hasta **4 veces más rápido que HTTP/1.1** en algunos casos.

Para el examen: sé explicar en una frase cómo mejora 1.1 a 1.0, y cómo mejora 2 y 3 a 1.1 (sobre todo: multiplexación y QUIC).

3. Conceptos: User-Agent y “stateless”

3.1. User-Agent

- Es el “**agente de usuario**”: el programa que representa al usuario, normalmente el **navegador**.
- Se envía en una cabecera User-Agent para indicar navegador, sistema operativo, etc.
- En tus diapositivas se ve cómo la cadena es larguísima y mezcla “Mozilla, AppleWebKit, Chrome...” etc.

Idea importante:

Es mala práctica detectar el navegador por la cadena User-Agent. Es mejor detectar **características concretas** (feature detection).

No hace falta memorizar ejemplos de cadenas; solo saber **qué es** y por qué no es buena idea fiarse de ella.

3.2. HTTP es stateless

Stateless significa:

- **HTTP no guarda estado** entre peticiones:
 - Cada petición se trata como independiente.
 - El servidor no “recuerda” quién eres de una petición a otra.

Ventajas (te las pueden preguntar):

- Escalabilidad → más fácil repartir peticiones entre servidores.
- Menos complejidad en el servidor.
- Mejor rendimiento.
- Es más fácil **cachear** recursos (respuestas se pueden reutilizar).

Desventajas:

- Mantener la sesión de usuario es más complicado:
 - Necesitamos **cookies, tokens, etc.** para saber quién eres.
- Más vulnerable a ataques tipo **DDoS** (cada petición se puede procesar “sin contexto”).

Frase de examen:

HTTP es un protocolo “sin estado”: el servidor no mantiene información entre peticiones, lo que mejora escalabilidad y caché, pero hace que mantener sesiones de usuario requiera mecanismos adicionales (cookies, etc.).

4. URL y URI

4.1. Qué es una URL

URL = Uniform Resource Locator.

- Es la **dirección única** de un recurso en la web: HTML, imagen, vídeo, etc.
- Tiene este formato general:

```
scheme://[userinfo@]host[:port]/path[?query][#fragment]
```:contentReference[oaicite:14]{index=14}
```

Partes importantes:

- `scheme`: protocolo (`http`, `https`, `ftp`...).
  - `userinfo`: opcional (`usuario:contraseña@`). No se usa casi nunca hoy en día.
  - `host`: dominio o IP (`www.ejemplo.com`).
  - `port`: opcional; por defecto:
    - HTTP → 80
    - HTTPS → 443
  - `path`: ruta del recurso dentro del servidor (`/blog/post1.html`).
  - `query`: parámetros extra (`?q=text&orden=asc`).
  - `fragment`: parte interna del recurso (`#seccion1`).
- :contentReference[oaicite:15]{index=15}

Ejemplo de las diapositivas:

```
`http://google.es:8080/ejemplo/index.html?q=text#something`
:contentReference[oaicite:16]{index=16}
```

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

### ### 4.2. URI vs URL vs URN

- \*\*URI\*\* (\*Uniform Resource Identifier\*) es el concepto general.

:contentReference[oaicite:17]{index=17}
- Puede ser:
  - \*\*URL\*\*: indica \*\*dónde\*\* está y cómo acceder (localizador).
  - \*\*URN\*\*: indica un \*\*nombre\*\* único, pero no necesariamente dónde está (ej.: `urn:isbn:0451450523`).

Para el examen con esto basta:

> Toda URL es una URI, pero no toda URI es una URL. La URL dice cómo localizar el recurso; una URN solo lo nombra.

---

## ## 5. Peticiones HTTP (requests)

### ### 5.1. Estructura general

La estructura formal es: :contentReference[oaicite:18]{index=18}

```
```text  
Método SP URL SP Versión CRLF  
Cabeza: valor CRLF
```

Cabecera: valor CRLF
...
CRLF
[cuerpo opcional]

- Primera línea = **start line**:
 - Método (GET, POST, ...).
 - URL (generalmente relativa al host).
 - Versión (HTTP/1.1, HTTP/2, etc.).
- Luego van **cabeceras**: Nombre: valor.
- Línea en blanco.
- Opcionalmente, un **cuerpo** (body) con datos (por ejemplo, datos de un formulario en POST).

5.2. Ejemplo GET

Ejemplo simplificado de las transparencias:

```
GET /en/html/dummy?name=MyName&married=not+single&male=yes HTTP/1.1
Host: www.explainth.at
User-Agent: ...
Accept: ...
...
```

- Los parámetros del formulario (name, married, male) van **en la URL** (`?name=...&married=...&male=...`).
- No hay cuerpo → la **petición GET no lleva body normalmente**.

5.3. Ejemplo POST equivalente

La misma información pero con POST:

```
POST /en/html/dummy HTTP/1.1
Host: www.explainth.at
...
Content-Type: application/x-www-form-urlencoded
Content-Length: 39

name=MyName&married=not+single&male=yes
```

- La URL ya no lleva los parámetros.
- Los datos van en el **cuerpo** (body), pero con el mismo formato campo=valor&campo=valor.
- Aparecen cabeceras que indican el tipo (Content-Type) y tamaño (Content-Length) del cuerpo.

En el examen tienes que entender que:

- **GET** → parámetros en **query string** de la URL.
- **POST** → parámetros en el **cuerpo** de la petición.

6. Métodos HTTP (verbos)

Los métodos indican la **acción** sobre el recurso.

Los más importantes:

- **GET**
 - Pide un recurso al servidor.
 - No debería modificar nada en el servidor (solo lectura).
 - **Safe e idempotente.**
- **HEAD**
 - Igual que GET, pero solo devuelve **cabeceras**, sin cuerpo.
 - Para saber si el recurso existe, tamaño, etc.
 - Safe e idempotente.
- **POST**
 - Envía datos al servidor para que los procese.
 - Puede **crear o modificar** recursos.
 - **No** es ni safe ni idempotente (dos POST pueden crear dos cosas).
- **PUT**
 - Enviar un recurso completo para **crear o reemplazarlo**.
 - Idempotente (si envías lo mismo varias veces, el resultado final es el mismo).
- **DELETE**
 - Solicita borrar el recurso indicado.
 - Idempotente (borrarlo una vez o varias deja el recurso “no existe”).

Otros que debes conocer de nombre:

- **OPTIONS** → pregunta al servidor qué métodos soporta para una URL.
- **CONNECT** → se usa para tunelizar tráfico cifrado (HTTPS a través de un proxy).
- **TRACE** → diagnóstico (hace eco de la petición).
- **PATCH** → modificación parcial de un recurso.

6.1. Métodos **safe** e **idempotent**

La tabla de colores de tu PDF resume esto.

- **Safe:**
 - No deberían cambiar el estado del servidor, solo leer.
 - Ej.: GET, HEAD, OPTIONS, TRACE.
- **Idempotent:**
 - Hacer la misma petición varias veces deja el mismo estado que hacerla una vez.
 - Ej.: GET, PUT, DELETE, HEAD, OPTIONS, TRACE.

Ten en mente:

- Todos los **safe** deberían ser **idempotent** también.
- Pero **POST** no es safe ni idempotent.

6.2. GET vs POST (tabla importante)

La tabla de la diapositiva 40–41 te da todas las diferencias.

Qué tienes que poder decir:

- **GET:**
 - Parámetros van en la URL → **visibles** en la barra de direcciones.
 - Limitados por la longitud máxima de la URL.
 - Solo permite caracteres ASCII (datos binarios complicados).

- Se puede **cachear**, se puede poner en **favoritos**, queda en el **historial**.
- **MENOS seguro** (nunca usar GET para contraseñas o datos sensibles).
- **POST:**
 - Parámetros van en el **body** → no se ven en la URL.
 - No hay limitación típica de tamaño (solo la que ponga el servidor).
 - Permite datos binarios (**multipart/form-data**).
 - No se cachea igual ni queda en historial con los parámetros.
 - Es **más adecuado** para formularios con datos sensibles.

Frase para el examen:

GET se usa para obtener recursos y parámetros en la URL; es cacheable pero poco seguro para datos sensibles. POST se usa para enviar datos en el cuerpo, no es seguro por sí mismo, pero es más adecuado para formularios (especialmente con datos sensibles o grandes).

7. Respuestas HTTP (responses)

7.1. Estructura

Una respuesta tiene esta forma:

```
Versión SP código-estado SP frase-explicación CRLF
Cabecera: valor CRLF
Cabecera: valor CRLF
...
CRLF
[cuerpo]
```

Ejemplo:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Content-Length: 1234

<html>...</html>
```

- Start line:
 - Versión de HTTP.
 - **Código de estado** (3 cifras).
 - Frase textual (OK, Not Found...).
- Cabeceras.
- Cuerpo con el HTML, JSON, imagen, etc.

7.2. Categorías de códigos de estado

Las **5 categorías** son obligatorias de saber:

- **1xx** → Informativos.
- **2xx** → Todo OK.
- **3xx** → Redirecciones.
- **4xx** → Error del **cliente**.
- **5xx** → Error del **servidor**.

7.3. Ejemplos importantes por categoría

1xx – Información

- 100 Continue → el servidor dice “sigue, vale”.
- 101 Switching Protocols → cambiando de protocolo (ej. a WebSocket).

2xx – Éxito

- 200 OK → todo bien, respuesta normal.
- 201 Created → se ha creado un nuevo recurso (típico de POST/PUT).
- 204 No Content → petición correcta pero sin cuerpo en la respuesta.

3xx – Redirección

- 301 Moved Permanently → el recurso se ha movido de forma permanente (importante para SEO).
- 302 Found → redirección temporal.
- 304 Not Modified → el cliente puede usar la **copia en caché**.

4xx – Error del cliente

- 400 Bad Request → petición mal formada.
- 401 Unauthorized → requiere autenticación (login).
- 403 Forbidden → tienes credenciales, pero no permisos.
- 404 Not Found → recurso no encontrado.
- 405 Method Not Allowed → método HTTP no permitido para ese recurso.

5xx – Error del servidor

- 500 Internal Server Error → el servidor no sabe qué hacer.
- 503 Service Unavailable → servicio no disponible (sobrecarga, mantenimiento...).

Si te aprendes al menos 2–3 de cada grupo vas sobrada.

7.4. El caso especial del 301 (muy de examen)

Cuando cambias la ruta de un recurso:

- Si no haces nada → los usuarios con la URL antigua verán un **404**.
- Mejor devolver un **301 Moved Permanently** con la nueva URL.
 - Para el usuario → redirección transparente.
 - Para SEO → el buscador entiende que es el **mismo recurso** en una nueva dirección, no uno nuevo.

8. HTTPS

HTTPS = *HTTP Secure*.

- Es HTTP + **capa de cifrado** basada en **SSL/TLS**.
- Usa normalmente el **puerto 443**.
- Proporciona:
 - **Confidencialidad** (nadie puede leer el tráfico).
 - **Integridad** (nadie lo puede modificar sin que se note).
 - **Autenticación** del servidor (certificados).

El esquema de la diapositiva 59 te cuenta el flujo general:

1. El navegador pide una página <https://...>
2. El servidor envía su **certificado** con su **clave pública**.
3. El navegador verifica el certificado.
4. El navegador genera una **clave simétrica**, la cifra con la clave pública y la manda.
5. El servidor la descifra con su **clave privada**.
6. A partir de ahí, todo el tráfico se cifra con la clave simétrica.

Frase de examen:

HTTPS es HTTP sobre TLS/SSL. Usa cifrado para asegurar la conexión (puerto 443), de forma que los datos no viajen en claro y se pueda autenticar el servidor.

Qué te pueden preguntar (mini checklist mental)

Si, sin mirar apuntes, puedes hacer estas cosas, tienes HTTP muy bien:

- Definir en una frase **qué es HTTP** y decir que es **stateless**.
- Explicar **en qué mejora** HTTP/1.1 a 1.0, y qué aportan HTTP/2 y HTTP/3.
- Escribir la estructura de una **petición HTTP** y de una **respuesta HTTP**.
- Diferenciar claramente **GET** y **POST** y saber cuándo usar cada uno (seguridad, parámetros, caché...).
- Explicar qué es un método **safe** y uno **idempotent**, y clasificar GET, POST, PUT, DELETE.
- Explicar qué es una **URL**, descomponer un ejemplo en **scheme, host, path, query, fragment**.
- Diferenciar **URI / URL / URN**.
- Decir las **5 categorías** de códigos de estado y poner ejemplos de cada una.
- Explicar qué hace el **301** y por qué es importante.
- Definir qué es **HTTPS** y qué añade a HTTP.

Si quieres, luego te puedo hacer una tanda de preguntas tipo examen de solo elegir método y código + pequeños ejercicios de “dime qué hace esta petición y si está bien”.

Tema 3 – HTML

1. Qué es HTML y qué es un lenguaje de marcado

- **HTML** = *HyperText Markup Language*.
- Es un **lenguaje de marcado**, no de programación:
 - Sirve para **describir la estructura y el significado del contenido** (qué es un título, un párrafo, una imagen...).
 - El navegador usa esas marcas para saber **cómo mostrar** la página.
- **Hipertexto**: texto que contiene enlaces a otros recursos (links).

Frase para el examen:

HTML es un lenguaje de marcado que define la estructura de una página web mediante elementos (etiquetas) y atributos; permite crear hipertexto con enlaces entre documentos.

2. Elementos HTML: la unidad básica

2.1. Estructura de un elemento

Un elemento HTML típico tiene:

```
<nombre_etiqueta atributo="valor">
    contenido
</nombre_etiqueta>
```

- **Etiqueta de apertura**: <p>, <h1>, <div class="...">.
- **Atributos**:
 - Añaden información extra.
 - Si tienen valor → **siempre entre comillas**.
- **Contenido**:
 - Texto, otros elementos, o puede estar vacío.
- **Etiqueta de cierre**: </p>, </h1>, etc.

2.2. Void elements (muy importante)

- Son elementos que **no pueden tener hijos y no tienen etiqueta de cierre**:
 - Ejemplos: ,
, <input>, <meta>, <link>.
- En HTML **no se usan self-closing tipo** como en XHTML (aunque algunos navegadores lo toleran).

2.3. Anidamiento correcto

- Los elementos se pueden **anidar**, formando un **árbol**.
- El **elemento raíz** es <html> ... </html>.
- Muy importante: **cerrar en el orden correcto**:

✓ Correcto: <tag1><tag2>Contenido</tag2></tag1>

✗ Incorrecto: <tag1><tag2>Contenido</tag1></tag2>

Un HTML “bien” anidado pasará mejor los validadores y evitará errores raros.

3. Estructura mínima de una página HTML

Tiene siempre esta forma:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8">
    <title>Título de la página</title>
  </head>
  <body>
    ¡Hola mundo!
    <!-- Comentario -->
  </body>
</html>
```

Puntos clave:

- <!DOCTYPE html>
 - Declara el tipo de documento (HTML5).
 - No es una etiqueta HTML. Siempre va al principio.
- <html lang="es">
 - Elemento raíz.
 - El atributo lang es **recomendable** (accesibilidad y SEO).
- <head>
 - Metainformación de la página (no se muestra).
 - Veremos su contenido en el siguiente punto.
- <body>
 - Todo lo que el usuario ve: textos, imágenes, formularios, etc.

4. El <head>: metadatos esenciales

Dentro de <head> van cosas que **no se muestran directamente**, pero son fundamentales:

4.1. <title>

```
<title>Mi primera página</title>
```

- Es **obligatorio**.
- Sale en la pestaña del navegador y en resultados de búsqueda.
- Solo puede haber **uno** por documento.

4.2. Metadatos <meta>

Ejemplo típico:

```
<meta charset="UTF-8">
<meta name="description" content="Descripción de la página">
<meta name="author" content="Tu nombre">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

- charset="UTF-8" → codificación de caracteres (importantísimo para tildes y ñ).
- description, author, keywords, viewport... ayudan a buscadores y a móviles.

4.3. Favicon (le gusta al profe que lo sepáis)

- Iconito que aparece en la pestaña del navegador.

Se añade con <link>: <link rel="icon" type="image/png" href="/images/favicon.png">

Detalles:

- Formatos: .ico, .png, .svg.
- Tamaño típico: 16×16 hasta 256×256.

5. El <body>: contenido visible y comentarios

5.1. <body>

- Solo puede haber **un** <body> por documento.
- Contiene todos los elementos **visibles**.

5.2. Comentarios

<!-- Esto es un comentario -->

- No se ven en la página.
- Útiles para explicar secciones de código.
- El navegador los ignora.

6. Elementos básicos de contenido

6.1. Encabezados y párrafos

```
<h1>Título principal</h1>
<h2>Subtítulo</h2>
...
<h6>Encabezado menos importante</h6>
```

```
<p>Esto es un párrafo de texto.</p>
<br> <!-- salto de línea -->
```

- Usar h1–h6 para dar **estructura lógica** al documento.

6.2. Listas

- **Ordenadas:**

```
<ol>
  <li>Primero</li>
```

```
<li>Segundo</li>
</ol>
```

- **No ordenadas:**

```
<ul>
  <li>Elemento</li>
  <li>Otro elemento</li>
</ul>
```

- **De descripción:**

```
<dl>
  <dt>Café</dt>
  <dd>Bebida negra caliente</dd>
</dl>
``` :contentReference[oaicite:11]{index=11}
```

### ### 6.3. Texto con estilo (con cuidado)

- Etiquetas como `<b>`, `<i>`, `<u>` existen, pero hoy en día se prefieren:
  - `<strong>` → énfasis fuerte.
  - `<em>` → énfasis normal. :contentReference[oaicite:12]{index=12}
- El profe prefiere que el \*\*aspecto visual\*\* lo lleve CSS, no HTML.

---

## ## 7. Imágenes, tablas, enlaces

### ### 7.1. Imágenes: `<img>`

```
```html
![Descripción de la imagen](imagen.jpg)
```

- **Void element** (sin cierre).
- **src**: ruta de la imagen (mejor relativa).
- **alt**: **obligatorio** a efectos de accesibilidad.
- **width, height**: tamaño en píxeles (o mejor desde CSS).

7.2. Tablas básicas

```
<table>
  <tr>
    <th>Mes</th>
    <th>Ahorros</th>
  </tr>
  <tr>
    <td>Enero</td>
    <td>100 €</td>
  </tr>
</table>
``` :contentReference[oaicite:14]{index=14}
```

```
7.3. Enlaces: `<a>`
```

```
```html
<a href="https://www.google.com">Ir a Google</a>
```

- href: URL de destino.
 - Mejor rutas relativas dentro de tu sitio, no rutas absolutas del sistema (C:\....).
- Puedes enlazar:
 - Teléfono: href="tel:+34123456789".
 - Email: href="mailto:yo@ejemplo.com?subject=Hola".

Enlaces internos y fragmentos

```
<a href="#seccion1">Ir a la sección 1</a>
```

```
<h2 id="seccion1">Sección 1</h2>
```

- #id enlaza a un elemento con ese id en la misma página.

Atributo target

- _self (por defecto) → misma pestaña.
- _blank → abre en nueva pestaña.

8. <div> y y elementos semánticos

8.1. Div y span

```
<div class="bloque">
  <h2>Título</h2>
  <p>Texto dentro del div.</p>
</div>
```

```
<p>Mi madre tiene los ojos <span class="azul">azules</span>. </p>
```

- <div> → contenedor de **bloque** (para agrupar secciones).
- → contenedor **en línea** (para resaltar parte de un texto).

⚠ El profe insistió: **no abusar** de <div> para todo.

Cuando exista una etiqueta más específica, usarla.

8.2. Estructura semántica del contenido

HTML5 añade etiquetas que dan **significado**:

- <header> → cabecera.
- <nav> → navegación (menús, índices).
- <section> → secciones generales del documento.

- <article> → pieza de contenido independiente (post, noticia, comentario).
- <aside> → contenido lateral (barras laterales, publicidad).
- <footer> → pie de página (autor, copyright...).

Usar estas etiquetas ayuda a:

- Accesibilidad.
- SEO.
- Entender mejor la estructura del HTML (y el profe lo valora).

9. Formularios (MUY importante en el examen)

Aquí el profe fue muy pesado 😊 :

tienes que dominar formularios, inputs, labels, **GET vs POST**, y cómo enlazar label con input.

9.1. <form>

```
<form action="/action_page.php" method="get">
  ...
</form>
```

- action: URL a la que se envían los datos.
- method: get o post (minúsculas). Si no pones nada, es get.

9.2. <label> y asociación con inputs

```
<form id="miForm">
  <label for="nombre">Nombre:</label>
  <input id="nombre" name="nombre" type="text">
</form>
```

- for en el label → debe coincidir con el id del input.
- Opcionalmente, form="miForm" si el label está fuera del <form>.
- También puedes meter el <input> dentro del <label>:

```
<label>Nombre: <input type="text" name="nombre"></label>
```

9.3. <input>: tipos importantes

<input> es **void element** y sirve para muchos tipos:

- type="text" → texto normal.
- type="password" → texto oculto.
- type="radio" → opción única de un grupo.
- type="checkbox" → marcar/desmarcar.
- type="number" → números.
- type="hidden" → valor oculto.
- type="submit" → botón de envío.

Atributos útiles:

- required → campo obligatorio.

- `minlength`, `maxlength` → longitud mínima y máxima.

9.4. Listas y selects

- `<datalist> + list` en un `input` → lista de sugerencias.
- `<select> + <option>` → lista desplegable.

```
<label for="cars">Elige un coche:</label>
<select name="cars" id="cars">
  <option value="volvo">Volvo</option>
  ...
</select>
```

9.5. Botones

```
<button type="submit">Enviar</button>
```

- Mejor usar `<button>` que `<input type="button">` porque permite meter HTML dentro (iconos, etc.).

9.6. GET vs POST en formularios (clave de examen)

- **GET:**
 - Parámetros van en la **URL** (`?name=...&lname=...`).
 - No usar con **datos sensibles** (contraseñas, DNI, etc.).
 - Se puede cachear / guardar en marcadores.
- **POST:**
 - Parámetros en el **cuerpo** de la petición.
 - Apto para datos sensibles (siempre mejor con HTTPS).
 - Permite enviar muchos datos.

Regla que el profe repitió mil veces:

Nunca uses GET para enviar contraseñas o datos sensibles; usa POST.

10. Canvas, audio y vídeo

10.1. Canvas

```
<canvas id="myCanvas"></canvas>
<script>
  const canvas = document.getElementById('myCanvas');
  const ctx = canvas.getContext('2d');
  ctx.fillStyle = '#FF0000';
  ctx.fillRect(0, 0, 80, 80);
</script>
```

- `<canvas>` es solo un **contenedor de gráficos**.
- Se dibuja dentro con **JavaScript**.
- El profe dijo claramente:
 - No hace falta aprender de memoria las funciones (rectángulos, círculos, etc.).
 - Sí entender qué es y reconocer código básico.

10.2. Audio y vídeo

- <audio controls src="...">
- <video controls> <source ...> </video>

Solo debes saber que existen y para qué sirven, no detalles profundos.

11. Buenas prácticas de estilo (importante para el validador)

En la parte de “Estilo” te da recomendaciones que el profe quiere que sigas:

- Nombres de **etiquetas y atributos siempre en minúscula**.
- Cerrar siempre las etiquetas que no sean void.
- Atributos **entre comillas**:

```
<p class="texto">...</p>
```

- Imágenes con alt, width, height.
- No poner espacios alrededor de =:

```
class="miClase"    <!-- OK -->
class = "miClase" <!-- MAL estilo -->
```

- Evitar líneas de código interminables.
- Usar sangrado consistente (2 espacios, no tabs).
- Usar siempre <html>, <head>, <body>, <title>.

Validador y caracteres especiales

- Algunos caracteres se escapan:
 - < → <
 - > → >
 - & → &
 - " → "
 - Tildes y ñ suelen funcionar bien con charset="UTF-8", pero existen á, ñ, etc.

El profe insistió:

Si pasas el HTML por el **validador W3C**, no deberían salir **warnings** ni errores.

Enlace del validador: [validator.w3.org] (lo tienes en la última diapositiva).

12. Atributos globales y accesibilidad

Atributos que cualquier etiqueta puede usar:

- **id**:
 - Identificador **único** en el documento.
 - Para CSS, JS y enlaces con #id.
- **class**:
 - Una o varias clases separadas por espacios.
 - Para aplicar estilos y seleccionar elementos con JS.

- `style`:
 - CSS en línea (mejor evitarlo en código “serio”).
- `hidden`:
 - Oculta el elemento en la página.

También menciona:

- Atributos ARIA (`role`, `aria-*`) → accesibilidad.
- Manejadores de eventos (`onclick`, `onfocus`, `onsubmit`, ...).

En el examen basta con que:

- Sepas qué son `id` y `class`.
- Entiendas que ARIA y los `on...` existen, sin profundizar mucho.

13. XHTML (solo idea general)

- XHTML es como HTML pero con reglas de **XML estrictas**:
 - Todo en minúsculas.
 - Todas las etiquetas se cierran.
 - Atributos siempre con comillas.
 - Void elements se cierran tipo `
`.

Solo tienes que saber que **HTML actual es más flexible** y que hoy se usa simplemente “HTML”, no XHTML.

Checklist mental para antes del examen

Si puedes hacer esto sin mirar apuntes, tienes HTML muy bien:

- Escribir la **estructura básica** de una página HTML correcta (`doctype`, `html`, `head`, `body`, `title`, `meta charset`).
- Explicar qué es un **elemento HTML**, qué es un **void element** y cómo se anidan bien.
- Crear un formulario simple que:
 - Use GET o POST según el caso.
 - Tenga `label` bien enlazada con `input` (`for` / `id`).
 - Use algún `required`, `type="password"`, `type="number"`, etc.
- Explicar por qué **no usar GET cuando hay datos sensibles**.
- Añadir imágenes con `alt` y enlaces correctos (internos y externos).
- Usar correctamente `id`, `class` y alguna etiqueta semántica (`header`, `nav`, `section`, `article`, `footer`).
- Pasar mentalmente tu HTML por el validador y pensar si habría errores (cierre, anidamiento, atributos, etc.).

Si quieres, en el próximo mensaje te preparo **un ejercicio de examen** tipo:

“Aquí tienes un HTML con fallos: arréglalo”.
y lo resolvemos paso a paso para que veas exactamente qué espera tu profe.

Tema 4 – CSS

1. Idea clave: qué es CSS y cómo se usa

- **CSS (Cascading Style Sheets)** sirve para definir **cómo se ve** una página HTML: colores, tamaños, márgenes, posiciones...
- Siempre hay que **separar**:
 - HTML → contenido y estructura.
 - CSS → estilos.
 - JS → lógica.

Formas de usar CSS:

1. **Archivo externo** (la buena): `<link rel="stylesheet" href="styles.css">`
2. `<style>` en el `<head>` (para ejemplos pequeños).
3. `style="..."` en elementos → solo para pruebas rápidas, en un proyecto real es **mala práctica**.

En el examen: que tengas claro que lo correcto es **un .css separado**.

2. Sintaxis básica

Todas las reglas siguen este formato:

```
selector {  
    propiedad: valor;  
    otra-propiedad: otro-valor;  
}
```

Ejemplo:

```
p {  
    color: red;  
    text-align: center;  
}
```

Comentarios:

```
/* Esto es un comentario */
```

Tienes que saber **leer y escribir** reglas simples sin liarte.

3. Selectores que debes dominar

3.1. Selectores simples

1. **Por etiqueta:** `p { color: red; }`
2. **Por clase:** `.destacado { color: blue; }`
3. **Por id:** `#cabecera { text-align: center; }`
4. **Elemento + clase:** `p.destacado { font-weight: bold; }`
5. **Universal:** `* { box-sizing: border-box; }`

6. **Agrupación:** h1, h2, p { color: green; }

3.2. Selectores de combinación (combinators)

Usa estos cuatro:

1. **Descendiente** (espacio): div p { color: red; }

Cualquier <p> dentro de un <div>.

2. **Hijo directo (>):** div > p { color: blue; }

Solo <p> que son hijos directos de <div>.

3. **Hermano adyacente (+):** h1 + p { margin-top: 0; }

El primer <p> justo después de un <h1>.

4. **Hermano general (~):** h1 ~ p { font-style: italic; }

Todos los <p> que vienen después de un <h1> como hermanos.

4. Pseudoclases

Son “estados” especiales de los elementos.

Las que tienes que controlar:

```
a:hover {  
    color: red; /* cuando pasas el ratón por encima */  
}  
  
input:focus {  
    border-color: blue; /* cuando el input está seleccionado */  
}  
  
li:first-child {  
    font-weight: bold; /* primer hijo de una lista */  
}  
  
li:last-child {  
    font-style: italic; /* último hijo de una lista */  
}
```

Si sabes explicar estas cuatro estás muy bien.

5. Pseudoelementos (también entra)

Son partes “virtuales” de un elemento.

Los básicos:

```

p::first-line {
    font-weight: bold;
}

p::first-letter {
    font-size: 200%;
}

p::before {
    content: "→ ";
}

p::after {
    content: " ✓";
}

```

Qué tienes que saber:

- ::first-line → primera línea del párrafo.
- ::first-letter → primera letra.
- ::before / ::after → permiten añadir contenido decorativo antes o después (sin tocar el HTML).

No hace falta usarlos de forma creativa; solo saber **qué son** y reconocerlos.

6. Modelo de caja (box model) – súper importante

Todo elemento es una caja con:

1. **Contenido** (content)
2. **Padding** (relleno interior)
3. **Border**
4. **Margin** (espacio exterior)

Ejemplo:

```
.caja {
    margin: 10px 20px;          /* fuera */
    padding: 5px;               /* dentro */
    border: 1px solid black;   /* borde */
}
```

Y muy importante:

```
* {
    box-sizing: border-box;
}
```

→ Hace que width incluya contenido + padding + border, y simplifica mucho la vida.

7. Position: colocar elementos

Tienes que saber explicar qué hace cada valor:

```
.caja {  
  position: relative; /* o absolute, fixed, sticky... */  
  top: 10px;  
  left: 20px;  
}
```

- **static**: por defecto, sigue el flujo normal.
- **relative**: se mueve respecto a donde estaría.
- **absolute**: respecto al ancestro posicionado más cercano.
- **fixed**: pegado a la ventana (no se mueve al hacer scroll).
- **sticky**: se comporta normal hasta un punto, y luego se queda pegado.

No hace falta maquetar una página entera con esto, pero sí **entenderlo**.

8. Specificity y conflictos entre reglas

Orden de “fuerza” (de más fuerte a más flojo):

1. #id
2. .clase
3. etiqueta (p, h1, etc.)

Y además, entre reglas con la misma especificidad, gana la que esté **más abajo** en el CSS.

Ejemplo típico:

```
p { color: blue; }  
.destacado { color: red; }  
#aviso { color: green; }
```

Para:

```
<p id="aviso" class="destacado">Hola</p>
```

→ color final: **verde** (por el **id**).

!important

```
p {  
  color: purple !important;  
}
```

- Hace que esta propiedad gane incluso sobre otras reglas más específicas.
- Solo para casos puntuales; abusar de **!important** es mala idea.

9. Responsive Web Design y media queries (de lo más importante)

9.1. Viewport

En el HTML:

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

→ Necesario para que en móviles no se vea minúsculo.

9.2. Media queries

Cambiar estilos según el ancho de pantalla:

```
/* Estilos por defecto (pantallas grandes) */
.column {
    width: 25%;
    float: left;
}

/* Pantallas medianas */
@media screen and (max-width: 992px) {
    .column {
        width: 50%;
    }
}

/* Móviles */
@media screen and (max-width: 600px) {
    .column {
        width: 100%;
    }
}
```

Idea que debes poder explicar:

Uso media queries para que la página se adapte al tamaño de pantalla → en grande varias columnas, en móvil todo en una columna, sin scroll horizontal.

10. Qué NO le interesa tanto (para que no pierdas tiempo)

- Cosas de diseño muy finas (tipografías raras, sombreados avanzados, animaciones, gradientes complicados...).
- Layouts supercomplejos con flexbox/grid (que están guays pero no los ha machacado en clase).
- Frameworks (Bootstrap, Tailwind...) → solo saber que existen.
- Detalles muy finos de SASS/Less → solo “son preprocesadores de CSS”.

Tema 5 –JS

0. Antes de nada: ¿para qué sirve JS?

- HTML = el texto y la estructura de la página.
- CSS = cómo se ve (colores, tamaños...).
- **JavaScript = lo que hace cosas:**
 - Mostrar/ocultar cosas.
 - Validar formularios.
 - Hacer peticiones al servidor sin recargar la página.
 - Animaciones, juegos, canvas...

Se ejecuta en el **navegador** usando un “motor” (V8 en Chrome, por ejemplo).

1. Cómo se mete JS en una página

1.1. Dos formas básicas

a) Código dentro del HTML

```
<script>
  alert('Hello, world!');
</script>
```

b) Archivo externo (lo bueno en proyectos)

```
<script src="/js/app.js"></script>
```

Importante: <script> **no** es una etiqueta vacía: siempre lleva </script>.

1.2. defer y async (al profe esto le importa)

En vez de poner los scripts al final del <body>, ahora los ponemos en el <head> con defer o async:

```
<head>
  <script src="/js/app.js" defer></script>
</head>
```

- **defer**
 - Baja el script en paralelo.
 - **Se ejecuta cuando el HTML ya está parseado.**
 - Mantiene el orden de los scripts.
- **async**
 - También baja en paralelo.
 - **Se ejecuta en cuanto está listo**, aunque el HTML no esté totalmente cargado.
 - El orden entre varios async puede cambiar.

Para examen:

“Usamos defer para que el JS no bloquee la carga del HTML y se ejecute cuando el DOM está listo.”

2. Cosas básicas para probar: alert, prompt, confirm, console.log

- alert("Hola"); → muestra un mensajito.

- `prompt("¿Cuántos años tienes?", "");` → pide texto; devuelve SIEMPRE un **string**.
- `confirm("¿Estás de acuerdo?");` → devuelve **true** o **false**.
- `console.log("texto");` → imprime en la consola (lo que más usarás).

El profe dijo: en producción **no usar** alert/prompt/confirm, solo para pruebas. Usar `console.log` para depurar.

3. Sentencias y punto y coma

- Cada instrucción es un **statement**:

```
console.log("Hola");
console.log("Adiós");
```

- JS a veces “mete” ; solo, pero se puede liar (Automatic Semicolon Insertion).
- Para el examen: escribe siempre con ; y evita cosas raras tipo:

```
alert("Hola")
[1,2].forEach(alert) // aquí puede pegarse la hostia
```

4. Variables: let, const y por qué NO var

4.1. Declarar variables

```
let message = "Hola";
const PI = 3.14;
```

- `let` → valor que puede cambiar.
- `const` → valor que NO se puede reasignar (tienes que darle valor al declararla).

4.2. ¿Qué pasa con var?

El profe lo dijo claro: **olvídate de var**.

Problemas de var:

- No tiene **scope de bloque**:

```
if (true) {
  var x = 5;
}
console.log(x); // 5 (sigue existiendo fuera del if)
```

- Se puede redeclarar sin que se queje.
- Tiene **hoisting**: puedes usarla antes de declararla, lo que lleva a bugs.

Con `let`:

```
if (true) {
  let x = 5;
}
```

```
console.log(x); // Error
```

Para el examen:

Usa **let** y **const** **siempre**. Si ves código con **var**, entiende qué hace, pero tú no lo uses.

5. Tipos de datos y conversiones

5.1. Tipos básicos

- **number** → enteros y decimales.
 - Valores especiales: `Infinity`, `-Infinity`, `NaN`.
- **string** → texto: `"hola"` o `'hola'` o ``hola``.
 - Template literals:

```
const name = "Ana";
const msg = `Hola, ${name}`;
```

- **boolean** → `true` o `false`.
- **null** → “no hay valor”, vacío a propósito.
- **undefined** → variable declarada pero sin valor.
- Arrays y objetos (los vemos luego).

5.2. NaN, null, undefined

- `NaN` = “Not a Number” -> resultado de una operación numérica rara: `"texto" / 2` // `NaN`
- `null` → valor vacío intencionado.
- `undefined` → todavía no tiene valor.

5.3. Conversiones

A **string**:

```
String(23);      // "23"
String(true);    // "true"
```

A **number**:

```
Number("23")    // 23
Number(" 23 ")  // 23
Number("")       // 0
Number("hola")  // NaN
Number(true)    // 1
Number(false)   // 0
```

Atajo con +: `+"2" + +"3" // 5`

A **boolean**:

```
Boolean(0);      // false
Boolean("");     // false
Boolean("0");    // true
Boolean(" ");    // true
Boolean(null);   // false
```

5.4. Cuidado con + mezclando strings y números

```
"2" + 2      // "22"  
4 + 5 + "px" // "9px"  
"2" / "5"    // 0.4 (se convierten a número)  
false + 34   // 34  
true + "34"  // "true34"
```

6. Comparaciones y condicionales

6.1. Comparar números y strings

Comparadores: >, <, >=, <=, ==, !=, ===, !==.

Strings se comparan **letra a letra** según Unicode:

```
"Z" > "A"      // true  
"Glow" > "Glee" // true  
"Bee" > "Be"     // true
```

6.2. == vs === (esto es clave)

- == → convierte tipos (puede hacer cosas raras).
- === → **no convierte tipos**, compara tipo y valor.

Ejemplos:

```
"01" == 1      // true  
"01" === 1     // false  
true == 1      // true  
true === 1     // false  
null == undefined // true  
null === undefined // false
```

Para examen:

👉 Usa SIEMPRE === y !== salvo que tengas un motivo clarísimo.

6.3. Casos raros con null y undefined

```
null > 0    // false  
null == 0    // false  
null >= 0   // true  
undefined > 0 // false  
undefined == 0 // false
```

Mensaje del profe: *tratar esto con cuidado.*

6.4. if y operador ternario

```
if (edad >= 18) {  
  console.log("Mayor");  
} else {
```

```
        console.log("Menor");
    }
```

Operador ternario (sintaxis que tienes que reconocer pero sin abusar):

```
let mensaje = (edad < 18) ? "Menor" : "Mayor";
```

6.5. Operadores lógicos: ||, &&, !

OR (||): devuelve el primer valor “verdadero” (o el último si todos son falsos).

```
let nick = "" || "Anonimo"; // "Anonimo"
true || alert("no se ve"); // no ejecuta alert
```

AND (&&): devuelve el primer valor “falso” (o el último si todos son verdaderos).

```
1 && 0      // 0
1 && 5      // 5
false && ... // devuelve false sin mirar lo de la derecha
```

NOT (!): invierte.

```
!true // false
!!"hola" // true, truco para convertir a booleano
```

6.6. Nullish coalescing ??

Muy probable que lo ponga en ejercicios: let valor = a ?? 5

Significa:

- Si a **NO** es null ni undefined → usa a.
- Si a es null o undefined → usa 5.

Sirve para dar valores por defecto sin liarla con 0, "", etc.

6.7. switch

Reemplaza varios if usando **igualdad estricta**:

```
switch (a) {
  case 4:
    console.log("es 4");
    break;
  case 5:
    console.log("es 5");
    break;
  default:
    console.log("otra cosa");
}
```

No te olvides del break.

7. Bucles

- `while` → mientras la condición sea verdadera:

```
let i = 3;
while (i) {
  console.log(i);
  i--;
}
```

- `do...while` → se ejecuta al menos una vez.
- `for`:

```
for (let i = 0; i < 3; i++) {
  console.log(i);
}
```

- `break` → salir del bucle.
- `continue` → saltar a la siguiente vuelta.

Las **labels** (`outer:`) existen pero son muy secundarias.

8. Funciones

8.1. Declaración normal

```
function sayHi(name) {
  console.log(`Hola, ${name}`);
}
```

- Las variables dentro de la función solo viven ahí (“scope local”).
- Pueden usar y modificar variables de fuera.

8.2. Parámetros y valores por defecto

```
function showMessage(from, text = "sin texto") {
  console.log(from + ": " + text);
}
```

```
showMessage("Ana", "Hola");    // Ana: Hola
showMessage("Ana");           // Ana: sin texto
```

Si llamas con menos parámetros, los que faltan son `undefined`.

8.3. Número variable de argumentos

Con `arguments`:

```
function foo() {
  console.log(arguments.length);
}
```

O con ...rest:

```
function myLog(x, ...args) {  
  console.log(x, args);  
}  
myLog("Hola", "qué", "tal");
```

8.4. return

- Devuelve un valor.
- Si no pones return, devuelve undefined.

```
function suma(a, b) {  
  return a + b;  
}
```

8.5. Funciones como valores (expresiones)

```
const sayHi = function(name) {  
  console.log(`Hola, ${name}`);  
};
```

Diferencia **importante**:

- Las **declaraciones** de función se pueden usar **antes** en el código (hoisting).
- Las **expresiones** NO.

8.6. Arrow functions (=>)

Sintaxis corta: const sum = (a, b) => a + b;

Si hay más de una línea, usamos {} y return:

```
const sum = (a, b) => {  
  const result = a + b;  
  return result;  
};
```

9. Arrays

Cosas que debes saber hacer seguro: crear, acceder, recorrer.

```
const fruits = ["Apple", "Banana"];  
  
fruits[0];           // "Apple"  
fruits.length;       // 2  
  
fruits.push("Orange"); // añade al final  
const last = fruits.pop(); // quita el último  
const pos = fruits.indexOf("Banana"); // posición  
fruits.splice(pos, 1); // borrar un elemento
```

Recorrer:

```
fruits.forEach((item, index) => {
  console.log(item, index);
});
```

En el examen le encanta liarla con:

- **recorrer arrays como si fueran objetos**
- o **objetos como si fueran arrays** → hay que saber la diferencia.

10. Objetos

Son conjuntos de **clave** → **valor**.

```
const user = {
  name: "John",
  age: 30,
  "likes birds": true
};
```

Acceder:

```
user.name;
user["likes birds"];
```

Añadir / borrar:

```
user.isAdmin = true;
delete user.age;
```

Comprobar si existe una propiedad: "age" in user; // true/false

Recorrer las claves:

```
for (let key in user) {
  console.log(key, user[key]);
}
```

También pueden ser **anidados** (un objeto dentro de otro).

11. DOM y eventos (API básica del navegador)

11.1. window y document

- **window** es el objeto global del navegador.
- **document** representa el HTML.

Seleccionar elementos:

```
document.getElementById("intro");
document.getElementsByClassName("intro");
document.querySelectorAll("p.intro");
```

Insertar elementos: appendChild, etc.

11.2. Eventos y addEventListener (muy importante)

Ejemplo click:

```
const boton = document.getElementById("boton");
boton.addEventListener("click", function() {
  console.log("Has hecho click");
});
```

También para carga de la página: window.addEventListener("load", init);

El profe insiste en que usemos **addEventListener** y no el viejo onclick.

11.3. Eventos de teclado y ratón

Teclado:

- keydown, keyup.

Ratón:

- click, dblclick, mousedown, mouseup, mousemove.

El manejador recibe un objeto event (e) con info:

```
document.addEventListener("click", function(e) {
  console.log(e.pageX, e.pageY);
});
```

12. Canvas (entenderlo, no memorizar todo)

Canvas = etiqueta HTML para dibujar con JS.

```
<canvas id="myCanvas" width="500" height="500"></canvas>
```

JS:

```
const canvas = document.getElementById("myCanvas");
const ctx = canvas.getContext("2d");
```

Con ctx dibujas:

- Líneas: moveTo, lineTo, stroke.
- Rectángulos: strokeRect, fillRect, clearRect.
- Círculos: arc(...).
- Texto: fillText, strokeText.
- Imágenes: drawImage(img, x, y).

LO QUE QUIERE EL PROFE:

--> Que sepas leer un trozo de código de canvas y entender qué hace.
--> No hace falta memorizar todos los métodos.

12.1. Animaciones: `setInterval`, `setTimeout`, `requestAnimationFrame`

- `setTimeout(fn, ms)` → ejecuta una vez después de ms.
- `setInterval(fn, ms)` → ejecuta cada ms.
- `requestAnimationFrame(draw)` → llama a draw antes del siguiente repaintido (lo “bueno” para animaciones).

Patrón típico:

```
function draw() {  
    ctx.clearRect(0,0,canvas.width,canvas.height);  
    // actualizar posición, dibujar...  
    requestAnimationFrame(draw);  
}  
requestAnimationFrame(draw);
```

13. Callbacks (funciones como argumentos)

Idea: una función que se pasa a otra función para que se ejecute más tarde.

Ejemplos:

```
[1,2,3].forEach(function (val) {  
    console.log(val);  
});  
  
setTimeout(function() {  
    console.log("Ha pasado 1 segundo");  
}, 1000);
```

Eso que enseñó de “callback hell” es para que entiendas por qué existen las **promesas**.

14. Strict mode ("use strict")

Activa un modo más estricto que:

- No deja usar variables sin declarar.
- Prohíbe cosas raras.
- Ayuda a encontrar errores.

```
"use strict";  
x = 3; // Error
```

Se activa automáticamente en **módulos y clases**.

15. Módulos (`import / export`)

Sirven para partir el código en archivos reutilizables.

En el HTML:

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

En `square.js`:

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

En `main.js`:

```
import { name, draw } from "./square.js";
```

Puntos clave:

- Los módulos siempre están en **modo estricto**.
- Lo importado no va al global.

16. Clases

Más azúcar sintáctico para trabajar con objetos.

```
class Rectangle {  
    constructor(height, width) {  
        this.height = height;  
        this.width = width;  
    }  
  
    getArea() {  
        return this.height * this.width;  
    }  
}  
  
const r = new Rectangle(10, 20);  
console.log(r.getArea()); // 200
```

También hay:

- Campos estáticos (`static total = 0;`).
- Campos privados (`#id`).
- Herencia: `class Dog extends Animal { ... }`.

17. Ajax, Promesas, fetch, async/await

17.1. Ajax y XMLHttpRequest (saber qué es, no dominarlo)

- Ajax = técnica para pedir datos al servidor sin recargar la página.
- XMLHttpRequest es la API “vieja” para hacerlo.

Ejemplo simplificado:

```
const httpRequest = new XMLHttpRequest();
httpRequest.onreadystatechange = function() {
    if (httpRequest.readyState === XMLHttpRequest.DONE &&
        httpRequest.status === 200) {
        console.log(httpRequest.responseText);
    }
};
httpRequest.open("GET", "diccionario.txt");
httpRequest.send();
```

Para el examen:

- Saber explicar qué es Ajax y XMLHttpRequest.
- Pero lo que de verdad quiere es que controles **fetch + promesas + async/await**.

17.2. Promesas

Una promesa es un objeto que representa algo que se resolverá en el futuro.

Estados:

- pending
- fulfilled
- rejected

Uso:

```
doSomething()
  .then(result => doSomethingElse(result))
  .then(finalResult => console.log(finalResult))
  .catch(error => console.error(error));
```

then encadena, catch captura errores.

17.3. fetch

Es la forma moderna de hacer peticiones HTTP en JS. Está basada en promesas.

GET:

```
fetch("diccionario.txt")
  .then(response => response.text())
  .then(text => {
    const palabras = text.split("\n");
    console.log(palabras.length);
  })
  .catch(err => console.error(err));
```

POST:

```
const data = { name: "John", age: 48 };

fetch("/api/users", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(data)
})
  .then(() => console.log("Enviado con éxito"))
  .catch(e => console.log("Error:", e));
```

17.4. `async / await`

Hace que trabajar con promesas parezca código normal secuencial.

```
async function getWords() {
  const response = await fetch("diccionario.txt");
  const text = await response.text();
  const palabras = text.split("\n");
  console.log(palabras.length);
}

getWords();
```

- `async` delante de la función.
- `await` solo se puede usar dentro de `async` (o en el top de un módulo).
- Puedes usar `try/catch` para errores.

Tema 6 – NODE.JS

1. ¿Qué es Node.js y en qué se diferencia del JS del navegador?

Piensa en JavaScript como un idioma.

- En el **navegador**, JS sirve para:
 - Tocar el HTML (document, window).
 - Hacer cosas en la página (formularios, animaciones...).
- En **Node.js**, usamos JS **en el servidor**:
 - No hay document ni window.
 - Sí tenemos acceso a:
 - Ficheros del sistema.
 - La red.
 - El sistema operativo.

Node es: “*JavaScript runtime environment*” → un programa que **ejecuta JS fuera del navegador**, usando el motor V8 de Chrome.

✓ Para el examen debes poder explicar:

“Node.js es un entorno para ejecutar JavaScript en el servidor. No tiene DOM ni window, pero sí tiene módulos para ficheros, red, etc., y usa el motor V8.”

2. Instalación, versiones, npm, nvm, REPL y npx

2.1. Versiones y LTS

Las versiones siguen el formato **MAJOR.MINOR.PATCH** (por ejemplo 22.3.1):

- MAJOR → cambios incompatibles.
- MINOR → cosas nuevas pero compatibles.
- PATCH → arreglos de bugs.

LTS = *Long Term Support*: versiones que se mantienen más tiempo (las “estables” para proyectos).

2.2. npm: el gestor de paquetes

Cuando instalas Node, viene **npm**:

- Sirve para instalar librerías:
 - npm install express
- Crear package.json:
 - npm init
- Instalar todo lo que está en package.json:
 - npm install o npm i

✓ En el examen te pueden preguntar:

“¿Qué es npm y para qué sirve?”

Respuesta: gestor de paquetes de Node, se usa para instalar dependencias y gestionar versiones en package.json.

2.3. nvm (solo concepto)

nvm = *Node Version Manager*:

- Sirve para tener **varias versiones** de Node y cambiar entre ellas:
 - nvm list, nvm use 22.0.0, nvm install 22.0.0.

Que sepas qué hace y ya.

2.4. REPL

REPL = **R**ead-**E**valuate-**P**rint **L**oop:

- Es una consola interactiva de Node:
 - Escribe node en la terminal.
 - Puedes probar instrucciones JS directamente.

Sirve para hacer pruebas rápidas.

2.5. npx

npx sirve para **ejecutar paquetes** sin tenerlos instalados globalmente:

npx cowsay "Hello"

Ejecuta el paquete “cowsay” al vuelo.

3. Ejemplo 0: tu primer servidor en Node

Código (lo has visto mil veces):

```
const http = require('http');
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  res.end('<h1>Hello, World!</h1>');
});

server.listen(port, () => {
  console.log(`Server running at port ${port}`);
});
```

Punto por punto:

- const http = require('http');
 - require = importar un módulo.
 - http es un módulo **del core** de Node (no hace falta instalarlo).
- http.createServer((req, res) => { ... })
 - Crea un servidor HTTP.

- Cada vez que entre una petición, se ejecuta la función.
- req = información de la petición (método, URL...).
- res = respuesta que enviaremos.
- Dentro del callback:
 - res.statusCode = 200; → código de estado.
 - res.setHeader('Content-Type', 'text/html'); → tipo de contenido.
 - res.end('<h1>Hello, World!</h1>'); → termina la respuesta y manda el HTML.
- server.listen(port, () => { ... })
 - Pone a escuchar el servidor en el puerto 3000.

✓ Para el examen debes saber **leer y explicar** este código.

4. Variables de entorno y process

4.1. Variables de entorno

- En Node se leen con process.env:

```
const PORT = process.env.PORT || 3000;
```

Eso significa:

- Si existe una variable de entorno PORT → úsala.
- Si no, usa 3000.

Se suelen usar para:

- El puerto.
- Modo (NODE_ENV=production o development).
- Claves secretas (APIs, etc.).

También se pueden cargar desde un archivo .env con la librería dotenv, pero basta que sepas que existe.

4.2. Argumentos por línea de comandos

Si haces: node index.js nombre=ana edad=20

En tu programa: console.log(process.argv);

- process.argv[0] → ruta al ejecutable node.
- process.argv[1] → ruta al archivo index.js.
- A partir del 2 → tus argumentos.

Existe minimist para parsearlos bonito (--name=ana...), pero el profe dijo que **no hace falta saberse minimist**.

4.3. process y os

- process:
 - process.argv, process.env, process.pid, process.cwd(), etc.
 - Saber: **es el objeto global con info del proceso Node.**
- os (módulo):
 - os.cpus(), os.totalmem(), os.hostname(), etc.
 - Saber: sirve para obtener info del sistema operativo (CPU, memoria...).

No tienes que memorizar todas las funciones, solo **para qué sirven estos módulos**.

5. JSON (esto es SUPER importante)

5.1. Qué es

JSON = **JavaScript Object Notation**:

- Formato de texto para intercambiar datos.
- Lo usan APIs, ficheros de configuración, etc.

5.2. Los 7 tipos de valor que acepta JSON

En JSON solo puede haber:

1. string (con comillas dobles) → "hola"
2. number → 23
3. object → { "edad": 23 }
4. array → [1, 2, 3]
5. true
6. false
7. null

Así que un JSON que sea **solo**: true

es **válido**, porque true es uno de los 7 valores.

5.3. Un único elemento raíz

Un documento JSON completo puede ser:

- Un objeto: { "nombre": "Juan" }
- Un array: [1, 2, 3]
- O incluso: 4, "hola", true, etc.

Pero **solo uno**, no varias cosas seguidas.

5.4. Objetos

Reglas:

- Van entre { y }.
- Son **pares clave:valor** separados por comas.
- La clave:
 - Siempre string con **comillas dobles**.
 - No puede repetirse en el mismo objeto.
- Cuidado con la **coma colgante** al final.

Ejemplo válido:

```
{  
  "nombreCompleto": "Juan Pérez",  
  "edad": 27  
}
```

5.5. Arrays

Reglas:

- Van entre [y].
- Contienen **valores** (cualquiera de los 7 tipos).

Ejemplo:

```
[  
  { "movil": 612345678 },  
  { "fijo": 912345678 }  
]
```

5.6. Orden en objetos y arrays

- En un **objeto**, el **orden NO importa**:

```
{ "nombre": "Juan", "edad": 27 }  
{ "edad": 27, "nombre": "Juan" }
```

son equivalentes.

Si el orden sí importa, tenemos que usar un **array**.

- En un **array**, el orden **sí importa**.

5.7. Corregir JSON (Ejercicio tipo examen)

Te pueden poner algo como el ejercicio de la [página 49](#) y decirte “¿esto es JSON válido? Arréglalo”.

Típicos errores a cazar:

- Claves sin comillas.
- Comas de más.
- Valores que no son de los 7 permitidos.
- true/false entre comillas (se convierten en strings).
- null mal colocado.

5.8. JSON con JavaScript

Dos funciones clave:

```
let text = JSON.stringify(obj); // objeto JS → texto JSON  
let obj = JSON.parse(text);    // texto JSON → objeto JS
```

Y para recorrerlo:

```
for (let key in obj) {  
  console.log("key:", key, "value:", obj[key]);  
}
```

Esto mezcla **JSON** con **cómo recorrer objetos en JS**, que también lo quiere en el examen.

5.9. Ejercicio 2 (cine)

Te cuentan frases tipo:

- “Spider-Man: No Way Home ha sido dirigida por...”
- “Se puede ver en la sala 17 a las 17:00...”

Y te piden **un JSON bien pensado y sin duplicar datos.**

La idea es:

- Agrupar la info:
 - Películas → dentro director y actores.
 - Cada película → lista de sesiones (sala + hora).
- Evitar repetir “Spider-Man: No Way Home” en cada frase.

6. package.json y package-lock.json

6.1. package.json

Es un fichero **JSON** con metadatos del proyecto:

- name, version, description
- main → archivo de entrada (index.js si no se dice otra cosa).
- scripts → comandos:

```
"scripts": {  
  "start": "node ./bin/www",  
  "test": "mocha"  
}
```

Se ejecutan con npm run start o npm start.

- dependencies → librerías que usa tu app.
- devDependencies → solo para desarrollo (tests, herramientas...).

Se puede crear con: npm init

6.2. Problema de solo usar package.json

Si pones: "vue": "^2.5.2"

- Uno puede acabar con 2.5.2,
- otro con 2.5.3,
- otro con 2.6.0.

Eso puede romper el proyecto.

6.3. package-lock.json

Soluciona eso:

- Guarda la **versión exacta** de cada dependencia que se instaló.
- Se genera solo al hacer npm install / npm update.

- Permite que todos tengan **las mismas versiones**.

✓ Para el examen:

"package-lock.json sirve para fijar las versiones exactas de las dependencias y que el proyecto sea reproducible."

7. Event Loop (diapo roja)

El **event loop** es el corazón de Node:

- Node funciona con **un único hilo**.
- Hay una cola de tareas (callbacks).
- En cada **tick** del loop ejecuta tareas de esa cola.

En el diagrama de la [página 71](#) ves varias fases del loop y cómo se van procesando callbacks.

La idea importante:

- **No bloquear** el event loop con cosas muy pesadas (bucles enormes, cálculos carísimos).
- Tareas largas → dividirlas, usar versiones asíncronas (`fs.readFile`, `fetch`, etc.).

Ejemplo de lo que NO se debe hacer: en la página 73 hacen un bucle enorme generando datos aleatorios con `crypto.randomBytes` dentro del request. Eso hace que una sola petición bloquee a las demás.

✓ Para el examen: explica con tus palabras:

"Node tiene un único hilo y usa un event loop con callbacks. Si bloqueo el loop con código pesado, todas las peticiones se afectan."

8. Trabajar con ficheros: path y fs

8.1. Módulo path

Sirve para manejar **rutas de archivos** de forma segura (independiente de Windows/Linux).

Funciones que tienes que reconocer:

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

path.basename("src/pkg/test.js"); // "test.js"
path.dirname("src/pkg/test.js"); // "src/pkg"
path.extname("src/pkg/test.js"); // ".js"
path.normalize("//a//b//"); // "/a/b/"
path.join("src", "pkg", "t.js"); // "src/pkg/t.js"
path.resolve("t.js"); // ruta absoluta al fichero
```

No hay que memorizar todas, pero sí la idea de que path te ayuda a:

- Unir rutas.
- Quitar .. y ..
- Obtener nombre de fichero, extensión, etc.

8.2. Módulo fs

fs = file system. Sirve para leer, escribir, borrar, etc.

Importante:

- Node tiene versiones **asíncronas y síncronas**:
 - `fs.readFile (async)`
 - `fs.readFileSync (sync)`

Ejemplo de lectura:

```
const fs = require('fs');

fs.readFile('test.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

Escritura:

```
const content = 'Algo de contenido';
fs.writeFile('test.txt', content, err => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('Escrito');
});
```

Versión con promesas:

```
await fs.promises.readFile("data.csv", "utf8");
```

También hay `fs.stat` para ver info de un fichero (tamaño, fechas, etc.).

9. Peticiones HTTP desde Node

Node puede ser:

- **Servidor** (con `http.createServer` o con Express).
- **Cliente** (pidiendo cosas a otros servidores).

Con los módulos `http` y `https` puedes hacer peticiones GET y POST.

Ejemplo GET:

```
const https = require('https');

https.get('https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY', (resp) => {
```

```

let data = '';
resp.on('data', (chunk) => {
  data += chunk;
});
resp.on('end', () => {
  console.log(JSON.parse(data));
});
}).on("error", (err) => {
  console.log("Error: " + err.message);
});

```

Idea clave:

- Te dan pedacitos (data) y al final (end) procesas el resultado.
- Con POST es similar, pero además envías datos y cabeceras (Content-Type, Content-Length).

No hace falta que memorices todo el código; solo entender el **flujo básico**.

10. Express: lo que de verdad usarás

Express es un framework para hacer servidores web más cómodamente.

Ejemplo mínimo:

```

const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:\${port}`);
});

```

10.1. Rutas y métodos

- `app.get('/ruta', handler)`
- `app.post('/ruta', handler)`
- `app.use(middleware)` → para “meter algo en medio” (logs, seguridad, etc.).

10.2. Middleware y next

Un **middleware** es una función que recibe (`req, res, next`):

```

function myLogger(req, res, next) {
  console.log('LOGGED');
  next(); // MUY IMPORTANTE
}

```

```
app.use(myLogger);
```

- Hace algo (por ejemplo, loguear).
- Luego llama a next() para que la petición continúe.

Si no llamas a next(), la petición se queda colgada.

En el ejemplo de login, el profe usa un middleware tipo checkLogin para comprobar si el usuario tiene sesión antes de dejarle pasar a ciertas rutas.

10.3. Plantillas y estructura del proyecto (express-generator)

Con express-generator te genera una estructura estándar:

- bin/www → punto de entrada (arranca el servidor, se hace el app.listen).
- app.js → configuración principal:
 - Motor de vistas:

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

- Carpeta estática:

```
app.use(express.static(path.join(__dirname, 'public')));
```

- Middlewares (logger, parseo JSON, URL encoded...).
- Rutas:

```
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

- Carpeta routes/ → ficheros que definen rutas (por ejemplo index.js, users.js).
- Carpeta views/ → plantillas EJS.
- Carpeta public/ → ficheros estáticos que puede ver el navegador (CSS, JS, imágenes...).

En el examen él puede darte un árbol de carpetas y preguntarte:

- ¿Dónde se configuran las vistas?
- ¿Dónde se define que public es estático?
- ¿Qué fichero se ejecuta primero (bin/www vs app.js)?

10.4. res.locals y app.locals

Muy importante:

- res.locals:
 - Variables **solo para una petición concreta**.
 - Se pasan a la vista que se va a renderizar.
- app.locals:
 - Variables **globales a toda la aplicación**.
 - Disponibles en todas las vistas.

Ejemplo de uso típico:

```
app.locals.title = "Mi Web";  
  
app.use((req, res, next) => {  
  res.locals.user = req.session.user;  
  next();  
});
```

En las plantillas EJS puedes usar <%= title %> o <%= user %>.

11. Socket.IO: comunicación en tiempo real

Socket.IO sirve para tener comunicación **bidireccional** entre cliente y servidor (chat, juegos en tiempo real, etc.).

11.1. En el servidor

```
const express = require("express");  
const { createServer } = require("http");  
const { Server } = require("socket.io");  
  
const app = express();  
const httpServer = createServer(app);  
const io = new Server(httpServer);  
  
io.on('connection', (socket) => {  
  console.log('a user connected');  
  
  socket.on('disconnect', () => {  
    console.log('user disconnected');  
  });  
});  
  
httpServer.listen(3000);
```

- `io.on('connection', callback)` → se ejecuta cuando un cliente se conecta.
- `socket` representa a **ese cliente concreto**.

11.2. En el cliente

```
<script src="/socket.io/socket.io.js"></script>  
<script>  
  const socket = io();  
</script>
```

11.3. Enviar y recibir mensajes

- Desde el servidor:
 - A ese socket: `socket.emit('evento', datos)`
 - A todos: `io.emit('evento', datos)`
 - A todos menos el actual: `socket.broadcast.emit('evento', datos)`

- Desde el cliente:
 - `socket.emit('evento', datos)` hacia el servidor.
 - `socket.on('evento', handler)` para recibir.

También hay **rooms**: un usuario se une a una sala con `socket.join('roomName')` y se puede emitir solo a esa sala con `io.to('roomName').emit(...)`.

En el ejemplo de Kahoot que comentó, la idea era:

- `socket.join("kahoot")` → meter a los jugadores en una room.
- `io.to("kahoot").emit("evento", datos)` → enviar un mensaje a todos los de esa room.