izertis

GUÍA COMPLETA

FRONT

Guía para directivos y técnicos

V.1

Front

Guía completa

El primer contacto de nuestros usuarios con nuestro negocio, y en muchos casos el único, es a través de una aplicación web o móvil. En esta guía hablamos de los 3 pilares del desarrollo front-end: HTML, CSS, y JavaScript.

En conjunto, deben ofrecer un diseño atractivo, con contenidos organizados, una experiencia de usuario agradable, accesible y que se adapte de manera adecuada para ser usada en diferentes dispositivos (Responsive). De lo contrario, se tendrá una pérdida masiva de potenciales clientes e influirá, negativamente, en el posicionamiento que los motores de búsqueda asignan a las aplicaciones. Proporcionar aplicaciones bien diseñadas inspira seguridad y confianza en nuestros clientes.

Ahora bien, si hiciésemos la siguiente pregunta: ¿qué idioma habla la Web? La respuesta no podría ser otra: JavaScript. Por ello, le dedicamos un capítulo completo. Prácticamente casi todos los dispositivos incluyen un navegador y por lo tanto, también un intérprete de este lenguaje.

Es el lenguaje seleccionado por algunos sistemas operativos o sistemas de escritorio como Gnome o Google's Chrome OS para la capa de presentación de sus aplicaciones nativas, colándose también en el mundo de backend de la mano de Node o Deno e incluso de dispositivos hardware como Arduino o Tessel.

"Conocer un lenguaje NO significa que seas desarrollador de Software"

En el último capítulo veremos **cómo es el ecosistema o entorno JavaScript** y qué herramientas necesitamos para montar un proyecto y un entorno. Hay muchas herramientas útiles en el desarrollo de JavaScript según la tarea que se quiera realizar: gestión de paquetes, transpiladores, herramientas de testing, automatización, etc. Algunas de estas no parecen necesarias en un principio, ya que el navegador sabe interpretar cualquier archivo .js que escribamos, pero estas herramientas nos ayudan a ser más eficientes y tener unas mejores prácticas de desarrollo (por ejemplo, nos permiten usar TypeScript o CoffeeScript frente a JavaScript). Como existen muchas alternativas para cada tipo de tarea, vamos a recomendar las de **uso más general.**

Además, nos adentraremos un poco en los **entresijos** de los **navegadores** y las posibilidades que nos ofrecen, así como

las **restricciones** que nos imponen de seguridad, memoria y comunicaciones. Todos estos conocimientos son necesarios para crear programas sólidos, mantenibles y seguros, facilitarnos la comprensión del ecosistema y la comunicación con compañeros del otro lado de la frontera (el Backend) y ayudarnos a **resolver los problemas** que de manera casi inevitable ocurrirán.

Front

Guía completa

Índice

        - Strong
        - Em
        - Button
        - Input
        - Image
        - Nav
        - Details y Summary
        - Blockquote
        - Article
        - Anchor
    - Etiquetas especiales
        - Script
        - Noscript
- CSS
    - Selectores
        - Selector de tipo
        - Selector de clase
        - Selector de id
        - Selector universal
        - Selector de atributo
        - Combinador de hermanos adyacentes
        - Combinador general de hermanos
        - Combinador de hijo
        - Combinador de descendientes
    - Unidades de longitud
        - Unidades absolutas
        - Unidades relativas
    - Layouts
        - Flexbox
        - Grid
    - Custom properties
        - Declarando una variable
        - Utilizando una variable
    - Posicionamiento

        - Static
        - Relative
        - Absolute
    - Diseño fluido
    - Diseño responsive
- Accesibilidad
    - Soluciones tecnológicas (AT)
        - Visuales
        - Auditivas
        - Motrices
        - Cognitivas
    - Normativas y estándares
        - WCAG 2.1
    - Técnicas básicas de HTML y CSS

Parte 2: Javascript

- Introducción
- Tipos básicos
    - String
    - Number
    - BigInt
    - Boolean
    - Undefined
    - Null
    - Object
    - Array
    - Function
- Variables y operadores

- Constantes
- Variables con let
- Template literals
- Property shorthand
- Property methods
- Parámetros por defecto
- Arrow function
- Desestructuración
  - Arrays
  - Objetos
- Operador Spread
  - Arrays
  - Objetos
- Estructuras de datos
  - Map
  - Set
  - WeakMap
  - WeakSet
- Optional chaining operator
- Nullish coalescing operator

- Asincronía
  - Callbacks
  - Promises
  - Async / await
- Array.prototype
  - Filter
  - Reduce
  - Sort
- Object.prototype
  - Object.keys

    - Object.values
    - Object.entries
    - Object.fromEntries
    - Object.assign
- ES Modules
    - Export default
    - Export nombrado

Parte 3: Javascript y entorno

- Node
- Gestores de paquetes
    - npm
        - package.json
        - Comandos
    - Yarn
        - Comandos
    - npm vs. yarn
- Transpiladores
    - Babel
- Bundlers
    - Webpack
    - Parcel
    - Snowpack
- Herramientas CSS
    - Sass
    - PostCSS
        - Autoprefixer
        - PostCSS Preset Env
        - Stylelint
- Browserslist
- Polyfills
- Herramientas de depuración
    - Consola
        - Cómo acceder a la consola

        - A tener en cuenta
    - Inspector de elementos
    - Inspector de peticiones
    - Modo dispositivo
- Storage
    - Local Storage
    - Session Storage
    - Cookies
- Seguridad
    - Ataques
        - Cross-Site Scripting (XSS)
        - Cross-Site Request Forgery (CSRF)
    - Políticas y aspectos de seguridad
        - Local Storage vs Cookies
        - CORS
        - Content Security Policy
- Web APIs
    - Caché
    - Pagos
    - Files
    - Mutation Observer
    - Request animation frame
- Gestión de memoria
    - Errores frecuentes
        - Variables globales accidentales
        - Timers o callbacks olvidados
        - Closures
        - Event listeners
        - Elementos eliminados del DOM
        - Observables
    - Cómo detectar memory leaks
- Bibliografía
- Lecciones aprendidas

Parte 1

HTML y CSS

Las bases del desarrollo Web

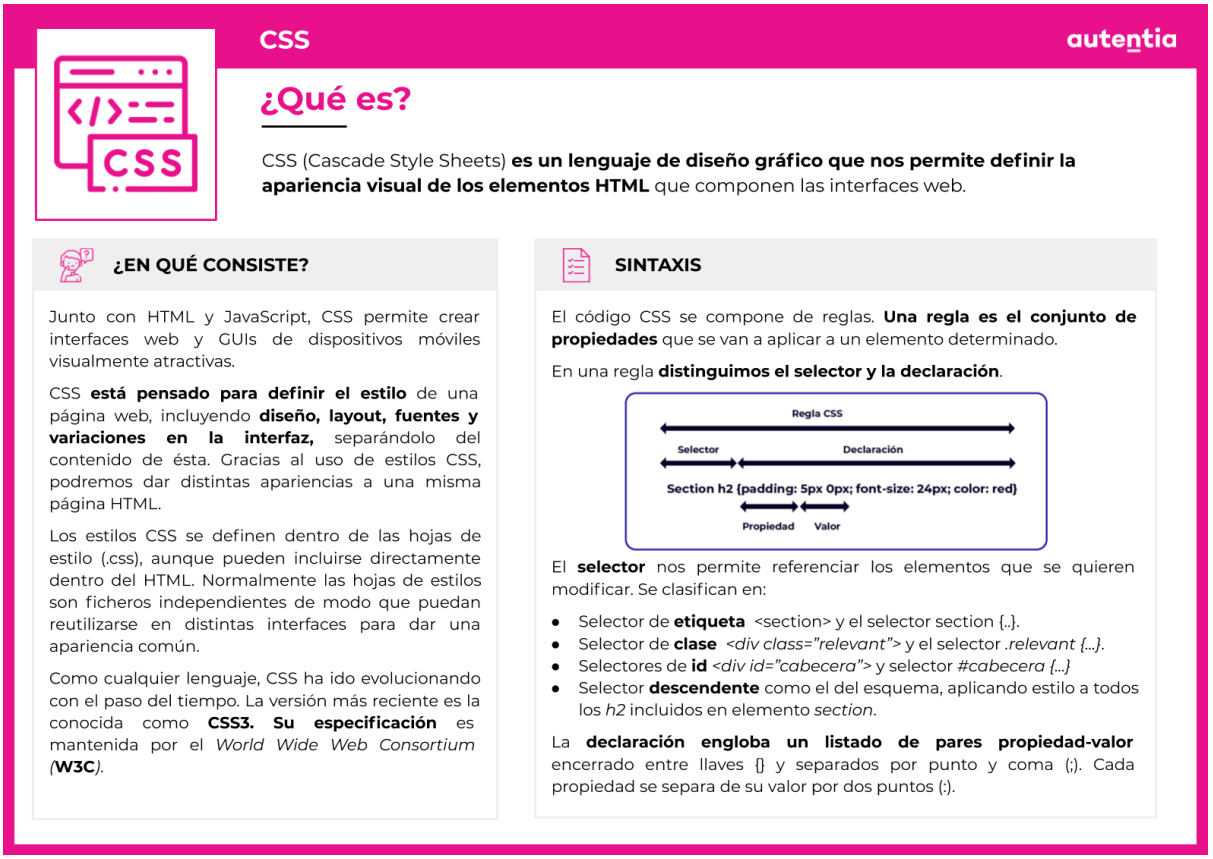
El HTML, o Lenguaje de Marcado de Hipertextos, es el lenguaje utilizado para **crear documentos en la web**. Proporciona significado (semántica) y estructura al contenido de una página web. Por ejemplo, sus contenidos podrían ser párrafos, una lista con viñetas o imágenes y tablas de datos.

"Hipertexto" se refiere a enlaces que conectan páginas web entre sí, ya sea dentro de un único sitio web o entre distintos sitios web. Los enlaces son un aspecto fundamental de la Web. Al cargar contenido en Internet y vincularlo a páginas creadas por otras personas, se convierte en un participante activo en la World Wide Web.

HTML utiliza "marcado" para anotar texto, imágenes y otro contenido para mostrar en un navegador web. El marcado HTML incluye "elementos" especiales. Un elemento HTML se separa de otro texto en un documento mediante "etiquetas", que consisten en el nombre del elemento rodeado por "<" y ">". El nombre de un elemento dentro de una etiqueta no distingue entre mayúsculas y minúsculas. Es decir, se puede escribir en mayúsculas, minúsculas o una mezcla.

“HTML y CSS son dos de los pilares (junto con JavaScript) del desarrollo front-end.”

¿Y qué es el CSS o Cascading Style Sheets? Se trata del lenguaje utilizado **para la presentación** (diseño o aspecto visual) **del contenido de una página web**. Utilizando CSS daremos estilo a cualquier tag de HTML.



CSS es un lenguaje de hojas de estilo utilizado para describir la presentación de un documento escrito en HTML o XML. Describe cómo se deben representar los elementos tanto visualmente, como de forma auditiva, por ejemplo, para los que sufren de alguna discapacidad visual y necesitan de alguna tecnología de asistencia de lectura de pantallas. CSS es una tecnología incluida dentro de la Web abierta y está estandarizado en todos los navegadores web de acuerdo con la especificación.

Podemos pensar en HTML, CSS y JavaScript como si fueran el cuerpo humano. El HTML sería el esqueleto, que es la estructura del cuerpo, JavaScript sería los músculos, lo que proporciona movimiento y flexibilidad, y CSS sería la piel, el cabello, los ojos y todo lo que se corresponde con la estética visual.

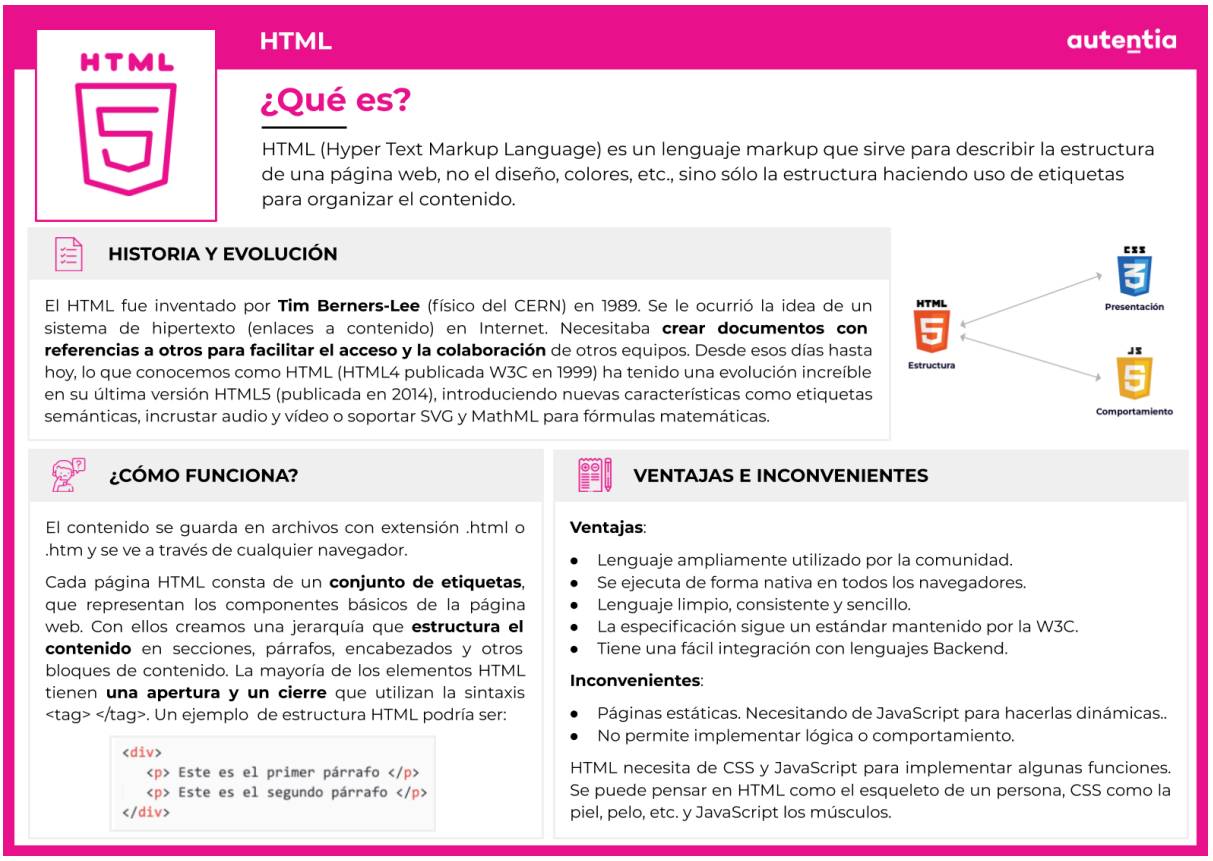
HTML

HTML (Hyper Text Markup Language) es un lenguaje markup que sirve para describir la estructura de una página web, no el diseño, colores, etc., sino sólo la estructura. Se basa en etiquetas. Por ejemplo, si queremos utilizar un espacio donde haya algún tipo de contenido, como puede ser simplemente texto, podríamos hacerlo así:

```
<div> Este texto se mostraría en la página </div>
```

Dentro de las etiquetas puede haber otras etiquetas. Podríamos, por ejemplo, tener dos párrafos dentro de un :

```
<div>
   <p> Este es el primer párrafo </p>
   <p> Este es el segundo párrafo </p>
</div>
```



HTML es un lenguaje de marcado “hermano” de XML y XHTML. Todos extienden de **SGML** (Standard Generalized Markup Language o lenguaje de marcado generalizado estándar), pero cada uno está pensado para un propósito diferente:

- **XML**: se utiliza para definir un conjunto de reglas para codificar documentos en un formato que sea comprensible tanto para humanos como para máquinas. Es una restricción de SGML que no permite etiquetas sin cerrar (deben estar cerradas con su correspondiente etiqueta de cierre o, en el caso de ser de autocierre, con la terminación “/>”), junto con otras restricciones más. Una de sus tantas aplicaciones es, por ejemplo, el intercambio de información entre sistemas.

- **HTML**: restringe SGML definiendo una lista de etiquetas que se permite utilizar y sólo es adecuado para la representación de páginas web.
- **XHTML**: XHTML restringe SGML con las etiquetas de HTML (con algunas exclusiones, como frameset y otras), agregando además, las restricciones de etiqueta y entidad de XML. Es, por lo tanto, un documento HTML con especificaciones adicionales para cumplir con el estándar XML.

Anatomía de un documento HTML

El siguiente código es sencillo, pero nos permitirá mostrar la estructura básica de un documento HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="iso-8859-1">
    <title>Página de prueba</title>
  </head>
  <body>
    <img src="images/autentia-icon.png" alt="Mi página de prueba">
  </body>
</html>
```

Aquí podemos encontrar los siguientes elementos:

- `<!DOCTYPE html>`: todo documento HTML debe comenzar con una declaración Doctype. Su función es informar a los navegadores de qué tipo de documento están procesando. En HTML 5 la declaración es tan sencilla como la del ejemplo, pero en versiones anteriores esta

era más complicada porque debía referir a un fichero DTD (Document Type Definition).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
```

- `<html>`: es el elemento raíz del documento HTML y envuelve todo el contenido de la página.
- `<head>`: este elemento sirve como contenedor para todo aquello que necesitemos incluir en nuestra página pero que no deseemos mostrar a los usuarios. Toda esta información es conocida también como metadatos (información sobre la información) y no se muestra a los usuarios. Aquí es donde solemos definir el título de la página, los íconos, el conjunto de caracteres, los estilos, scripts y más. Sólo puede haber un elemento head por documento HTML.
- `<body>`: este elemento envuelve a todo el contenido que deseamos mostrar a los usuarios cuando visitan nuestra página como texto, imágenes, audio, vídeo, etc. Sólo puede haber un elemento body por documento HTML.

HTML semántico

El HTML semántico es aquél que introduce significado o la semántica a la página, no sólo la presentación. Por ejemplo, la etiqueta `<b>` (negrita) o `<i>` (cursiva) no son semánticas ya que definen cómo se debería mostrar el texto, pero no aportan ningún significado.

También, si utilizamos un `<div>`, este no aporta nada de información ya que podría contener cualquier cosa y mostrarse de cualquier forma,

dependiendo del CSS. Sin embargo, si por ejemplo utilizamos `<p>`, estamos indicando que lo que hay entre estas etiquetas será un párrafo. Las personas saben lo que es un párrafo y los navegadores saben cómo representarlo, por tanto, `<p>` sí es una etiqueta semántica pero `<div>` no lo es.

También, por ejemplo, si utilizamos una etiqueta `<h1>` para envolver un texto, estamos dándole significado, estamos indicando que ese es un título principal en la página por tanto, también es una etiqueta semántica.

Class / id

Class e id son formas de identificar un elemento HTML. Class se utiliza para referenciar el elemento desde un archivo CSS y poder darle un estilo. El atributo global id define un identificador único, el cual no debe repetirse en todo el documento y cuyo propósito es identificar el elemento para poder hacer referencia al mismo, tanto en scripts como en hojas de estilo.

En el documento de CSS se pueden referenciar elementos de cualquiera de estas dos formas, entre otras. La diferencia entre una clase y un id, es que con id se puede identificar un sólo elemento, mientras que la clase se puede usar para identificar más de uno. Así es como se utilizarían en el HTML:

```
<p class="class-example"> Este es el ejemplo con clase </p>
<p id="id-example"> Este es el ejemplo con id </p>
```

Elementos Void

En HTML existen los **Void Elements** o elementos “vacíos”. Estos elementos solo tienen etiqueta de apertura ya que no tienen contenido y **no deben**

tener etiqueta de cierre. Ejemplos de este tipo de elementos son las etiquetas `<input>`.

Ejemplo:

```
<input type="number" placeholder="Introduzca un número">
```

o

```
<input type="number" placeholder="Introduzca un número"/>
```

son correctos, mientras que

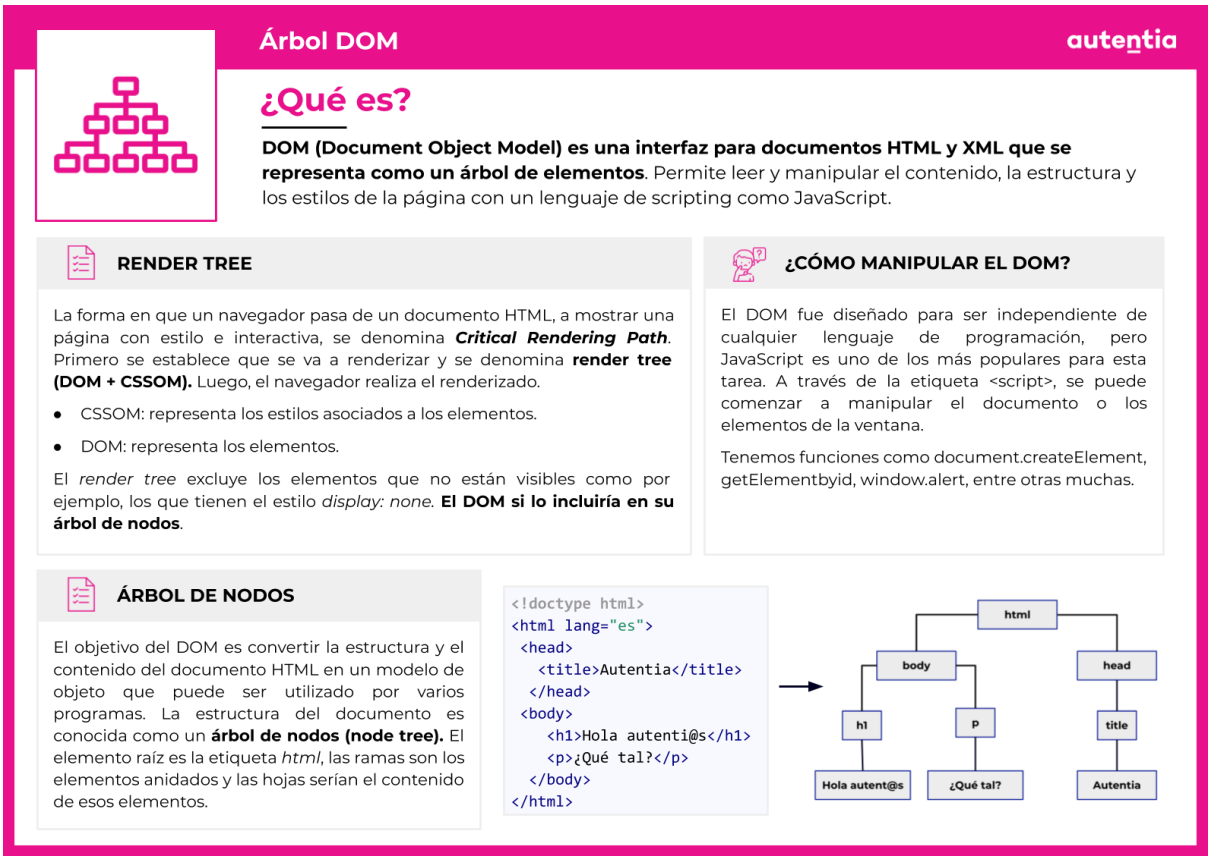
```
<input type="number" placeholder="Introduzca un número"></input>
```

es incorrecto.

DOM

DOM (Document Object Model) es la interfaz que permite que lenguajes de programación como JavaScript, modifiquen la estructura, el estilo o el contenido de un sitio web.

El modelo nos ofrece una abstracción que nos **permite tratar el documento y sus elementos como si fuesen objetos**. Para ello, se construye lo que se conoce como un "**Árbol de DOM**" en el que cada elemento del documento es un nodo.

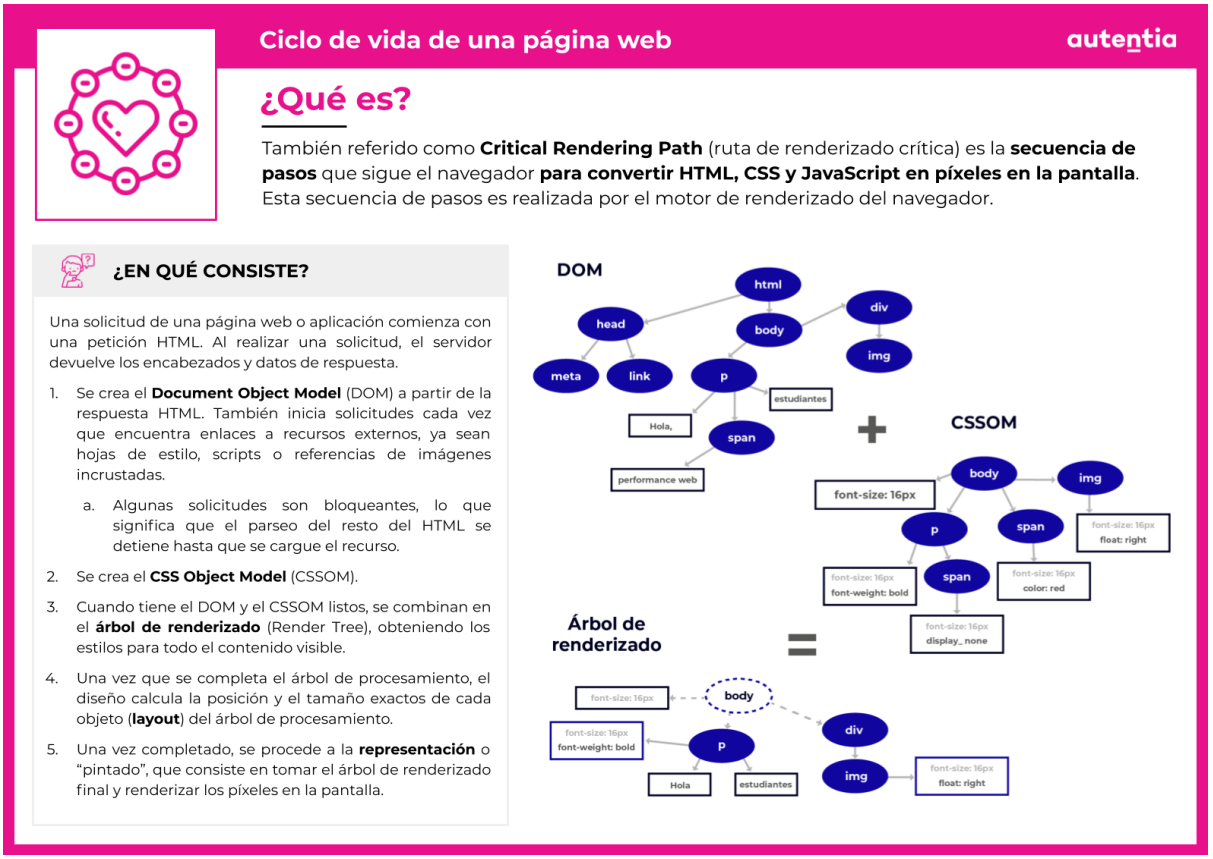


El DOM nos ofrece una multitud de propiedades y métodos para acceder a los diferentes elementos y poder modificarlos. Algunos de los métodos más utilizados son `getElementById()` o `getElementsByClassName()`. Ellos nos permiten acceder a los elementos del DOM indicando su atributo id y class respectivamente, similar al uso de selectores en HTML vistos en apartados anteriores.

Esto nos permite consultar y modificar los atributos de cada elemento. Algunos ejemplos de esto serían: extraer el texto introducido en una caja de texto o cambiar el estado de un elemento, deshabilitando un botón. Esta capacidad de manipular el DOM, nos ofrece muchas posibilidades como crear aplicaciones personalizadas que cambien su diseño sin necesidad de actualizar la página.

El DOM forma parte del árbol de renderizado, parte fundamental dentro del

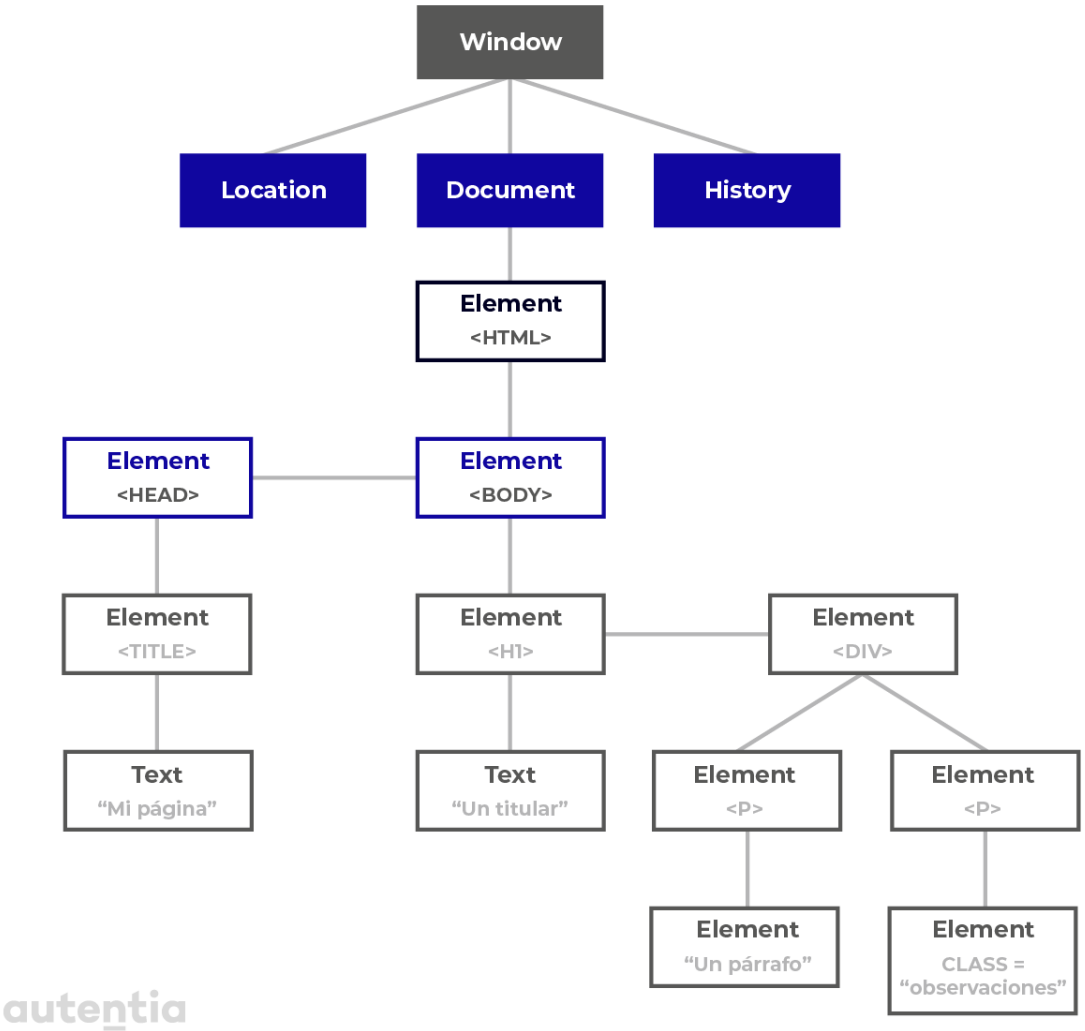
ciclo de vida de una p ágina web, representando todos sus componentes visibles en píxeles.



El DOM no sólo contiene representaciones visibles, también contiene eventos que se pueden producir en el documento o sus elementos, como arrastrar, clicar o teclear, entre otros.

También dispone de objetos que nos ofrecen metainformación sobre el navegador (objeto navigation), la pantalla (objeto screen), la url (objeto location) y más. Al árbol que contiene alguno de los objetos mencionados anteriormente, como los que representan componentes del navegador, se le llama BOM ("Browser Object Model").

Una representación del árbol BOM creado después de leer el documento con todos sus objetos, podría ser el siguiente:



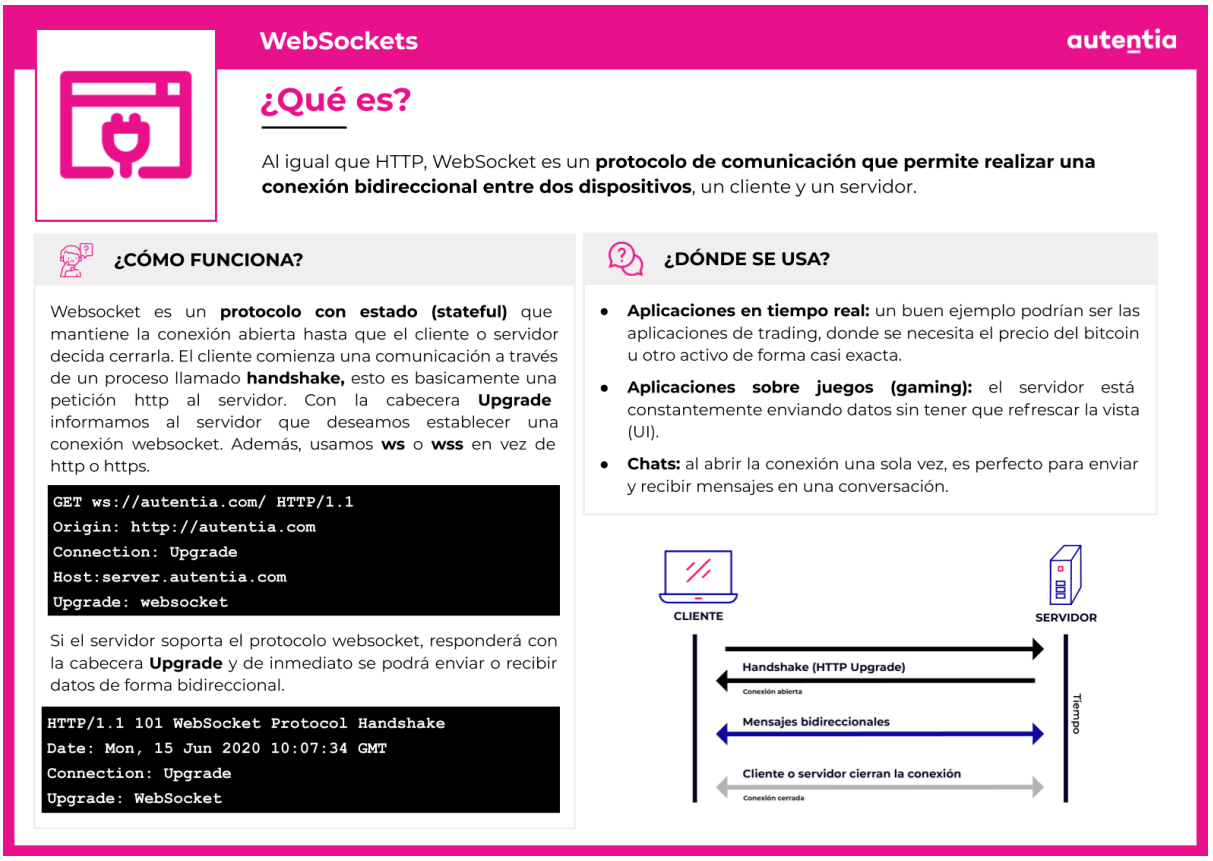
HTML5

HTML5 es la última evolución del estándar que define HTML. El término representa dos conceptos diferentes. Es una nueva versión del lenguaje HTML, con nuevos elementos, atributos y comportamientos y un conjunto más amplio de tecnologías que permiten la creación de sitios web y aplicaciones más diversas y potentes.

Entre las nuevas características que introduce la nueva versión,

encontramos:

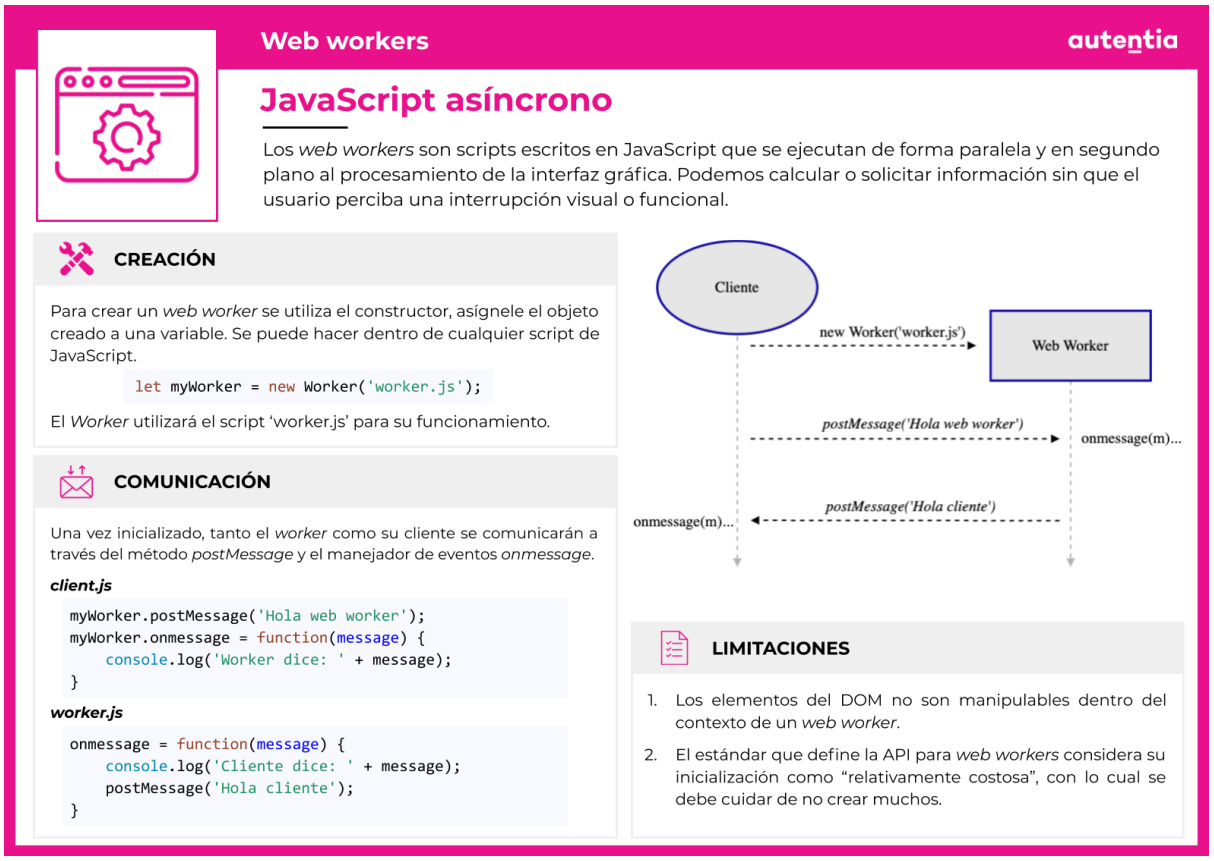
- **Semántica**: con HTML5 se incluye un conjunto más rico de etiquetas, junto con RDFa, microdatos y microformatos, que permiten una web más útil basada en datos para los programas y sus usuarios.
- **Fuera de línea y almacenamiento**: las aplicaciones inician más rápido gracias a la caché de aplicaciones HTML5, así como al almacenamiento local, la base de datos indexada y las especificaciones de la API de archivos.
- **Acceso a dispositivos**: se han implementado innovaciones como la API de geolocalización, el acceso a dispositivos, el acceso de entrada de audio / vídeo a micrófonos y cámaras, datos locales como contactos y eventos e incluso, orientación de inclinación.
- **Conectividad**: más eficiente y rápida, y una mejor comunicación. Web Sockets y Server-Sent Events, sirven para enviar datos entre el cliente y el servidor de manera más eficiente.



Multimedia: se añade un mejor soporte sobre contenido multimedia, permitiendo la reproducción de audio y vídeo sin necesidad de componentes o plugins adicionales y permitiendo que la página reproduzca de forma nativa dicho contenido. De esta forma, las páginas tardan menos en cargar y su navegación es más rápida.

3D, gráficos y efectos: funciones 3D de SVG, Canvas, WebGL y CSS3 para imágenes representadas de forma nativa en el navegador.

Performance e integración: implementación de tecnologías como Web Workers y XMLHttpRequest 2 para aplicaciones y contenido web dinámico más rápidos.



CSS3: CSS3 ofrece una amplia gama de estilización y efectos, mejorando las aplicaciones web sin sacrificar su estructura semántica o rendimiento. Además, Web Open Font Format (WOFF) proporciona flexibilidad tipográfica y control, mucho más allá de lo que la web ha ofrecido antes.

Etiquetas HTML5

Estas son algunas de las etiquetas HTML5 más utilizadas:

Etiquetas para el <head>

Title

Esta etiqueta sirve para definir el título del documento. El título debe ser solo de texto y se muestra en la barra de título del navegador o en la pestaña de la página. Esta etiqueta es requerida en todos los documentos HTML.

El contenido del título de una página es muy importante para la optimización de motores de búsqueda (SEO). Los algoritmos de los motores de búsqueda utilizan el título de la página para decidir el orden cuando se enumeran las páginas en los resultados de búsqueda.

```
<head>
  <title>Este es el título de la página</title>
</head>
```

Meta

Define información (metadata) sobre un documento HTML como el autor, la descripción de la página, la codificación de los caracteres, la configuración del viewport, etc.

Algunos valores y/o propiedades más usadas y conocidas dentro de esta etiqueta son las siguientes:

- **Description**: define una breve descripción de nuestra web y es la que usan los motores de búsqueda.

```
<meta name="description" content="HTML (HyperText Markup Language) is the most basic building block of the Web. It defines the meaning and structure of web content. Other technologies besides HTML are generally used to describe a web page.">
```

HTML: Hypertext Markup Language | MDN

HTML (HyperText Markup Language) is the most basic building block of the Web. It defines the meaning and structure of web content. Other technologies ...

- **Keywords**: define palabras clave sobre la página web.

```
<meta name="keywords" content="HTML, CSS">
```

- **Viewport**: define el área visible de un usuario en una web. `device-width` adapta el ancho de la página al dispositivo, aunque también se podría asignar un tamaño fijo en píxeles (no recomendado). También tiene propiedades para limitar el zoom que puede hacer el usuario.

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

- **Charset**: define la codificación de los caracteres del documento.

```
<meta charset="utf-8">
```

Link

Define la relación entre el documento actual y un recurso externo. La relación que tienen ambos se define con la propiedad rel (relationship), pero se enlazan a través de la propiedad href. La propiedad rel es obligatoria y sus valores más usados son stylesheet e icon, aunque tiene muchos más.

El uso más común de esta etiqueta es para enlazar hojas de estilos.

```
<link href="main.css" rel="stylesheet">
```

Style

La etiqueta style se utiliza para aplicar una hoja de estilos simple a un documento HTML. Últimamente no se suele utilizar ya que los estilos se suelen definir en un fichero aparte del HTML, mientras que esta etiqueta sirve para definir el CSS dentro del documento HTML. Por ejemplo:

```
<html>
  <head>
    <style>
      h1 {color:red;}
      p {color:blue;}
    </style>
  </head>
  <body>
    <h1>A heading</h1>
    <p>A paragraph.</p>
  </body>
</html>
```

Etiquetas para el <body>

Div

Estas etiquetas se utilizan a modo de contenedores ya sea para texto, para otras etiquetas o cualquier otro tipo de contenido. Se utilizan de la siguiente manera:

```
<div>Este es un contenido dentro del div</div>
```

Esta etiqueta no es semántica y no es recomendable abusar de ella. Cuando sea posible, se debería sustituir por otras etiquetas que sí sean

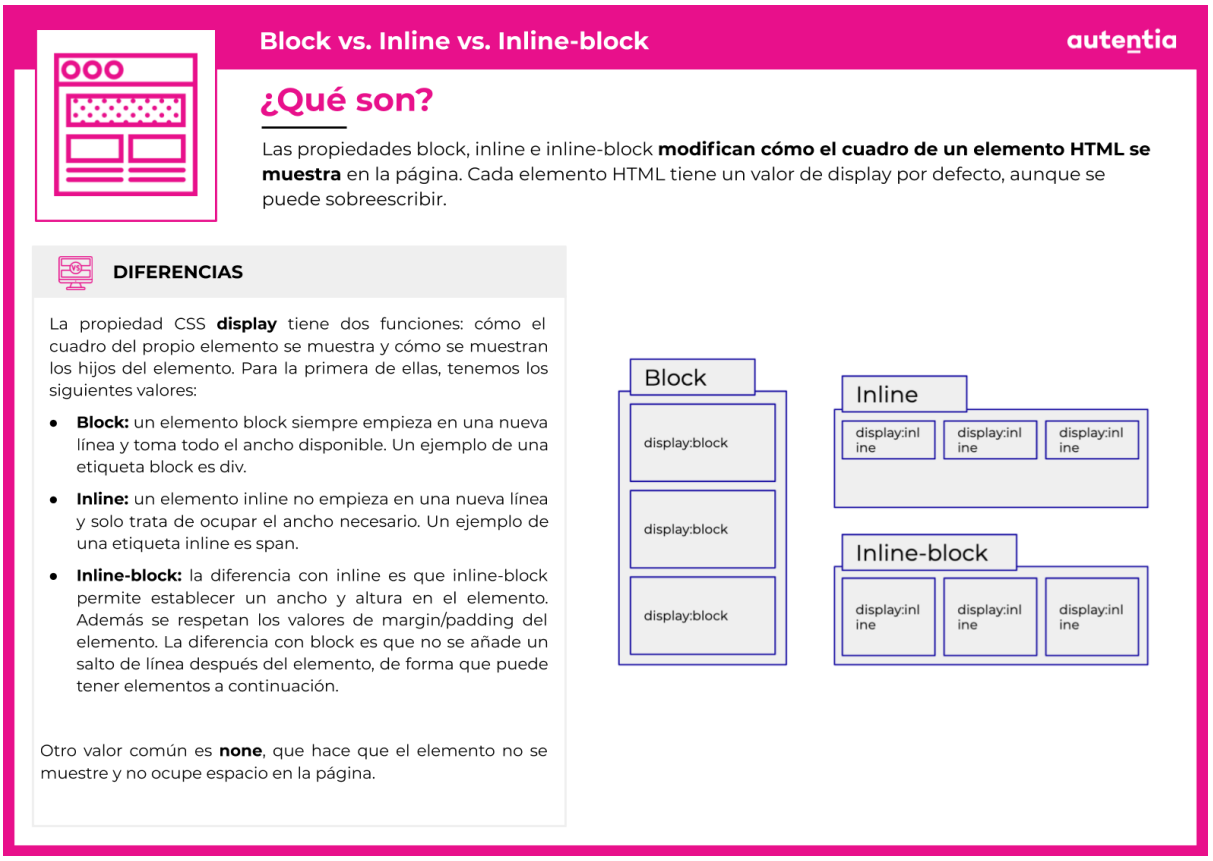
semánticas.

Span

El span es un contenedor inline, es decir, se utiliza para envolver parte de un texto o un documento. Esto puede ser para darles a esos elementos un estilo distinto (aplicando una clase o id) o porque comparten un atributo, por ejemplo, lang (lenguaje). Es básicamente como un div, con la diferencia de que div es un elemento a nivel de bloque, mientras que span es inline. Este es un ejemplo de cómo usarlo:

```
<p>
  En esa empresa hay <span class="workers-number"> 25 </span>
trabajadores
</p>
```

Y en la clase de CSS “workers-number” tendríamos los estilos que le queremos dar únicamente al número que representa la cantidad de trabajadores.



Paragraph

Esta etiqueta se utiliza a la hora de mostrar un párrafo y este sería un ejemplo de uso:

```
<p> Este es un párrafo </p>
```

Section

Su uso es bastante directo: definir una sección. Un ejemplo de una sección podría ser este:

```
<section>
   <h2> Este sería el título de la sección </h2>
   <p> Este es un párrafo </p>
</section>
```

Aside

Esta etiqueta define cierto contenido aparte del contenido en el que se coloca. Este contenido aparte debería estar indirectamente relacionado con el contenido que le rodea:

```
<section>
   <p>El verano pasado fuimos a una playa increíble en Cádiz.</p>
   <aside>
      <h4>Playa de Cortadura</h2>
      <p>Un texto explicativo sobre la playa</p>
   </aside>
   <p>Aparte de la playa, fuimos a cenar a unos cuantos sitios, a
      las ferias....</p>
</section>
```

Main

Con esta etiqueta se define el contenido principal del documento o página. Lo que contenga esta etiqueta debe ser único en el documento y no tener elementos que se repitan en otros, como puede ser una barra de navegación, un menú lateral, etc.

```
<main>
   Aquí iría el contenido
</main>
```

Header y Footer

Estas etiquetas, como su propio nombre indica, se usan para definir la cabecera y pie de de página respectivamente:

```
<header>Esta es una cabecera</header>
<footer>Esto es un pie de página</footer>
```

Headings

Los encabezamientos, comúnmente utilizados como títulos, se representan con las etiquetas de la `<h1>` a la `<h6>`, siendo la primera la correspondiente al formato de encabezado con mayor importancia y, por lo general, con mayor tamaño de fuente, entre otras cosas. Los encabezados <h6> son los más pequeños. Para utilizar estas etiquetas solo hay que envolver un texto entre ellas:

```
<h1>Esto es un título</h1>
```

Strong

La etiqueta `<strong>` se usa para definir un texto (o una parte de él) que sea importante. Tradicionalmente, el navegador aplicará negrita a ese texto, aunque el efecto de esta etiqueta se puede cambiar por medio de CSS. Su uso es muy sencillo:

```
<p>Así es como se define la parte <strong>importante<strong/> de
una frase<p/>
```

Em

Esta etiqueta se utiliza para enfatizar texto. El texto dentro de esta etiqueta `<em>` normalmente se muestra en itálicas aunque, al igual que con la etiqueta strong, se puede cambiar este comportamiento con CSS. Su uso es exactamente igual que strong.

Button

Esta etiqueta se utiliza para crear un botón en el que podemos escuchar al evento del click y decirle a qué función debe llamar en ese caso:

```
<button onclick="handleOnClick()">Click me</button>
```

Input

El input se usa para crear un campo en el que un usuario pueda ingresar información, como por ejemplo un texto. Entre las muchas propiedades que tiene el input, se puede especificar el tipo de input que es con type (texto, número, fecha...) o definir un texto que se mostrará si el input está en blanco con placeholder. Input es un **elemento de tipo Void**, por lo que este elemento no debe cerrarse con otra etiqueta, como en otros casos.

Este sería un ejemplo de un input en el que sólo se pueden introducir números y con un placeholder:

```
<input type="number" placeholder="Introduzca un número">
```

Image

Este sería un ejemplo de cómo representar una imagen:

```
<img src="route-to-image.jpg" alt="Texto alternativo">
```

En src también se puede poner la url de una imagen de Internet. Alt es un texto alternativo que se muestra cuando la imagen no se ha podido mostrar por alguna razón.

Figure

Sirve para identificar contenido como diagramas, ilustraciones, etc. Hace uso de `<figcaption>` para determinar el texto identificativo. Así es como se utilizaría

```
<figure>
   <img src="graph.jpg" alt="Hubo un error al cargar la imagen">
   <figcaption>Fig. 1 - Gráfica progreso mensual<figcaption/>
<figure/>
```

Nav

Esta etiqueta se usa para definir una serie de links navegables. No todos los links de una página web deben estar en un nav pero si hay una serie de links juntos, deberían contenerse dentro de un nav:

```
<nav>
   <a href="link.com">Link 1</a> |
   <a href="some-link.com">Link 2</a> |
   <a href="another-link.com">Otro link</a> |
   <a href="yet-another-link.com">Random Link</a>
</nav>
```

Details y Summary

La etiqueta details crea un desplegable interactivo que el usuario puede abrir y cerrar. Por defecto, comienza cerrado. El texto o elemento que se mostrará, incluso cuando el desplegable esté cerrado, deberá estar entre las etiquetas `<summary>` y éste debe ser el primer elemento dentro de `<details>`:

```
<details>
   <summary>Playa de Cortadura</summary>
   <p>Un texto explicativo sobre la playa</p>
</details>
```

En este caso, “Playa de Cortadura” se mostrará siempre y el párrafo de debajo se mostrará sólo cuando el elemento esté desplegado.

Blockquote

Esta etiqueta se usa para citar un texto o sección de otra fuente. Hay que especificar la fuente en la propiedad cite:

```
<blockquote cite="https://www.typescriptlang.org/">
   TypeScript is an open-source language which builds on
   JavaScript, one of the world's most used tools, by adding
   static type definitions.
</blockquote>
```

Article

Esta etiqueta específica contenido independiente y auto-contenido. Este contenido debería tener sentido por su cuenta y poder ser distribuido de forma independiente al resto de la página. Las etiquetas article no hacen que su contenido se renderice de forma distinta, sino que aporta meramente un significado contextual. Un pequeño ejemplo puede ser el siguiente:

```
<article>
   <h2>Algún título</h2>
   <p>Un texto cualquiera</p>
</article>
```

Anchor

La etiqueta `<a>` define un hipervínculo que se utiliza para enlazar a otra sección dentro del mismo documento, de una página a otra, dentro del mismo sitio o a un sitio diferente. El atributo más importante del elemento `<a>` es el atributo `href` que indica el destino del enlace.

```
<a href="https://www.autentia.com">Visita nuestra página</a>
```

El atributo href se utiliza para señalar la URL a la que apunta el link o enlace. Estos links pueden contener URLs relativas o absolutas. Una URL absoluta es aquella que contiene toda la información necesaria para localizar un recurso, mediante el formato *scheme://server/path/resource*, mientras que una URL relativa generalmente consiste del *path* y, opcionalmente, el *resource*, pero sin *scheme* o *server*, ya que estos los toma de la URL base desde la que está partiendo (en este caso, la del documento HTML). Pero los links no están restringidos a URLs basadas en el protocolo HTTP, si no que se puede utilizar cualquier esquema URL soportado por los navegadores.

Por ejemplo:

- Secciones de una página mediante fragmentos de URLs. Un fragmento de URL es una cadena de caracteres que refiere a un recurso que está subordinado o incluido en otro. Estos fragmentos son opcionales y se suelen marcar precediéndolos mediante el caracter #.

```
<a href="https://www.autentia.com#ejemplos">Visita nuestros
ejemplos</a>
```

Estos fragmentos también se pueden utilizar para enlazar a secciones dentro del mismo documento mediante el uso de otro tag `<a>` con el atributo id definido:

```
<a id="top">Bienvenidos</a>
<a href="#top">Volver arriba</a>
```

- Índices temporales específicos o segmentos dentro de ficheros de

medios mediante fragmentos de medios. Estos fragmentos de URL se utilizan para enlazar a una posición específica o a un fragmento de tiempo, a un fichero de audio o vídeo o a una porción de una imagen, etc.

Por ejemplo, el siguiente enlace nos llevará al minuto 30 del vídeo enlazado:

```
<a href="https://youtu.be/OQtS400I8m8?t=1800">Desarrollo de Juegos
en JAVA</a>
```

- Números telefónicos mediante el uso del esquema `tel:` en la URL.

```
<a href="tel:+34916753306">Llámenos!</a>
```

- Direcciones de email con el esquema `mailto:` en la URL.

```
<a href="mailto:info@autentia.com">Contáctenos!</a>
```

Etiquetas especiales

Estas etiquetas tienen la particularidad de que pueden incluirse tanto dentro de la etiqueta `<body>,` como del `<head>` y algunas, incluso también dentro de la etiqueta `<html>`.

Script

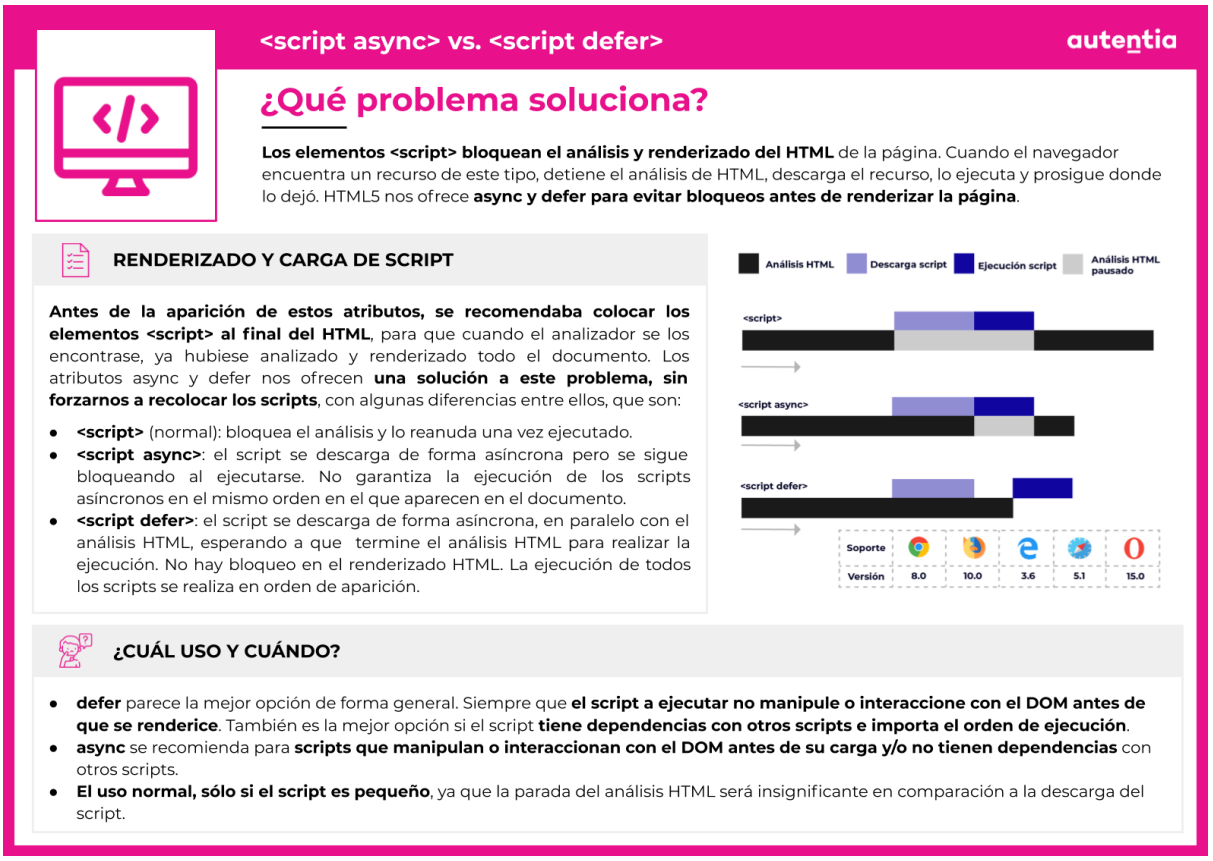
La etiqueta `<script>` permite integrar código ejecutable en el lado del cliente (normalmente JavaScript).

```
<!DOCTYPE html>
<html>
```

```
  <body>
    <p id="demo"></p>
    <script>
      document.getElementById("demo").innerHTML = "Hola Mundo";
    </script>
  </body>
</html>
```

Es importante destacar que, si bien es posible añadir todo el código JavaScript dentro de esta etiqueta (embebido), es preferible hacerlo a través de un fichero externo con extensión .js. Normalmente, esta etiqueta suele situarse al final del body del documento HTML. ¿Por qué? Porque de este modo, se impiden bloqueos al parsear el HTML. En caso de no hacerlo así, el usuario tendría que esperar a que se descarguen y ejecuten todos los scripts y esto no ofrece una buena experiencia de usuario.

Existen alternativas a lo nombrado anteriormente, que permiten la descarga y ejecución del script de una forma diferente.



- `<script async>`: permite descargar el script de forma asíncrona pero bloquea en análisis del HTML al ejecutarse. Además, no garantiza que los scripts se vayan a ejecutar en el mismo orden en el que están enlazados en el documento HTML.

- `<script defer>`: el script se descarga de forma asíncrona, en paralelo con el análisis del HTML. Cuando el análisis del HTML finaliza, los scripts se ejecutan en el orden en el que están.

Debemos tener cuidado con los problemas de compatibilidad entre navegadores. Por ejemplo, hay versiones de Gecko (motor de Mozilla) que difiere en el uso del atributo defer, ya que sólo funcionará cuando la etiqueta script tenga el atributo src y no funciona con código embebido en el html. En cualquier caso, podemos comprobar siempre la compatibilidad entre navegadores con herramientas como Can I Use.

Una vez vistas las diferentes alternativas, se podría concluir que usar el atributo defer y emplazar el elemento script dentro de la etiqueta *head* del HTML es la mejor opción para enlazar ficheros externos, ya que nos permite obtener el mismo comportamiento que al ubicarla al final del *body* pero sin mezclar contenido y scripts. En caso de no querer usar esta propiedad, se podría situar el script al final del documento HTML, obteniendo esta misma experiencia de usuario.

Noscript

La etiqueta `<noscript>` permite mostrar un contenido distinto en caso de que el usuario deshabilite los scripts en el navegador o sea el propio navegador el que no los soporte.

```
<!DOCTYPE html>
```

```
<html lang="en">
  <head>
    <title>demo autentia</title>
  </head>
  <body>
    <noscript>
      <strong>Lo sentimos, pero `demo autentia` no funciona
      correctamente si JavaScript está deshabilitado. Por favor
      habilítelo para continuar.</strong>
    </noscript>
  </body>
</html>
```

Podemos usar esta etiqueta tanto en la cabecera como en el body del documento HTML. Si la usamos en la cabecera, `<noscript>` solo puede contener las etiquetas `<link>`, `<style>`, y `<meta>`.

CSS

El CSS (Cascading Style Sheets) describe cómo se tienen que mostrar **visualmente** los elementos de HTML. Se utiliza para definir el estilo de una página web, incluyendo el **diseño**, **layout** y **variaciones** en la **interfaz** para distintos dispositivos y tamaños de pantalla. En resumen, el **HTML** aporta la **estructura** y los elementos de una web (esqueleto) y el **CSS** aporta la **capa visual** a los elementos del HTML.

El CSS hace referencia a elementos de HTML mediante varias cosas como puede ser el nombre de la etiqueta, la clase o el id. Aquí tenemos un ejemplo de un div con un fondo rojo:

HTML

```
<div class="class-example">
   <p> Este es el primer párrafo </p>
   <p> Este es el segundo párrafo </p>
</div>
```

CSS

```
.class-example {
   background-color: red;
}
```

Selectores

Los selectores sirven para definir los elementos sobre los que se van a aplicar reglas CSS. Hay distintos tipos de selectores que además se pueden combinar utilizando unos operadores (llamados combinadores) para hacer selecciones más complejas.

Selector de tipo

Selecciona todos los elementos del tipo que se ha definido en el selector.

```
div {
  // El estilo se aplicará a los elementos div.
}
```

Selector de clase

Selecciona todos los elementos que tienen la clase del selector.

```
.example {
  // El estilo se aplicará a todos los elementos que tengan la
  // clase 'example'.
}
```

Selector de id

Selecciona el elemento que tenga el id definido en el selector. Solo se selecciona un elemento porque el id es único en un documento HTML.

```
#example {
  // El estilo se aplica al elemento con el id 'example'.
}
```

Selector universal

Selecciona todos los elementos. Se puede utilizar junto con combinadores para hacer selecciones más complejas como seleccionar todos los hijos de un tipo de elemento.

```
* {
  // El estilo se aplica a todos los elementos.
}
```

Selector de atributo

Selecciona los elementos que tengan el atributo definido en el selector. También se puede poner el atributo con un valor para seleccionar solo los elementos que tengan el atributo y el valor.

```
[attr] {
  // El estilo se aplica a todos los elementos con ese atributo.
}
[attr=value] {
  // El estilo se aplica a todos los elementos con ese atributo y
  // valor.
}
```

Con los selectores anteriores se pueden hacer selecciones más complejas usando **combinadores**. Los combinadores son operadores que se utilizan entre selectores.

Combinador de hermanos adyacentes

Selecciona hermanos adyacentes, dos elementos que comparten padre y uno se encuentra al lado del otro. El combinador es el símbolo **+**.

```
h1 + p {
  // El estilo se aplica a todos los p que estén inmediatamente a
  // continuación de un h1.
}
```

Combinador general de hermanos

Selecciona hermanos, sin necesidad de que uno siga a otro. El combinador es el símbolo **~**.

```
h1 ~ p {
```

```
  // El estilo se aplica a los p que son hermanos del h1 que les
  // precede.
}
```

Combinador de hijo

Selecciona los hijos directos del primer elemento. El combinador es el símbolo **>**.

```
div > p {
  // El estilo se aplica los elementos p que son hijos de un div.
}
```

Combinador de descendientes

Selecciona todos los elementos descendientes del primer elemento. El combinador es un **espacio**.

```
div li {
  // El estilo se aplica a los li que estén dentro de un div, da
  // igual si son hijos directos o no.
}
```

Unidades de longitud

En CSS existen **diversas unidades** para expresar una longitud.

Unidades absolutas

En CSS **no se recomienda** utilizar unidades de **longitudes absolutas** porque los tamaños de pantalla varían mucho. Las distintas unidades absolutas son: **cm**, **mm**, **in** (pulgada), **px** (píxeles), **pt** y **pc**. La unidad más utilizada de estas son los píxeles.

Unidades relativas

Las unidades relativas expresan una longitud relativa a otra propiedad de longitud. Este tipo de unidades **escalan mejor** a la hora de renderizar en distintos tipos de pantalla. Las unidades más utilizadas son:

- **em**: esta unidad es relativa a la propiedad font-size del elemento. 2em significa dos veces el tamaño de la fuente.

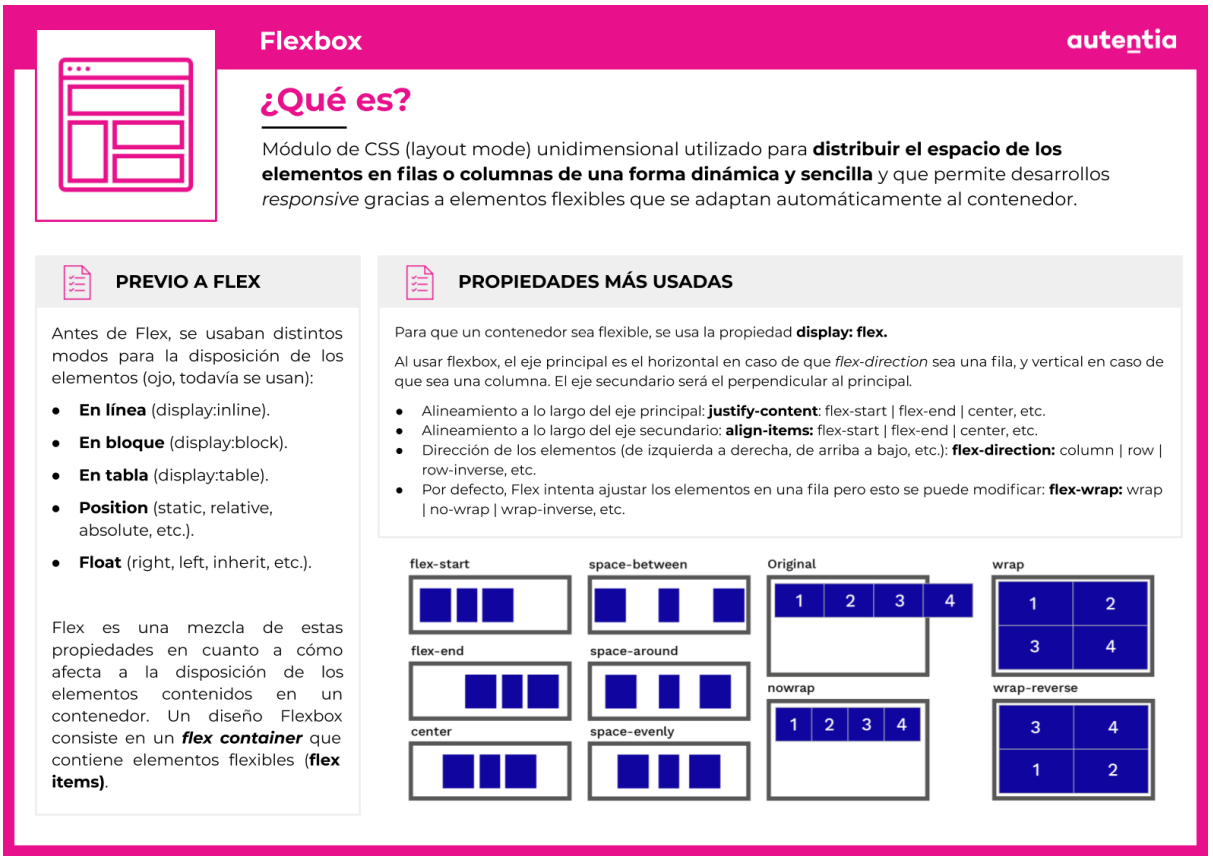
- **rem**: igual que la anterior pero esta es relativa al font-size del elemento root.
- **ch**: relativa al ancho del cero (0).
- **vw**: unidad relativa al 1% del ancho del viewport. El viewport es el tamaño de la ventana del navegador.
- **vh**: igual que la anterior pero relativa al 1% del alto del viewport.
- **%**: unidad relativa al elemento padre.

Layouts

El layout es básicamente la disposición de los elementos en la pantalla. En este apartado se explicará justamente eso: cómo colocar los elementos HTML donde queremos. Antiguamente, para maquetar una página web se utilizaban tablas pero hace un tiempo, surgieron herramientas como flexbox o grid como una solución mucho más práctica.

Flexbox

Flexbox es un método de layout **uni-dimensional** utilizado para colocar elementos en **filas** o **columnas**. De esta forma, se consigue que los elementos se **expandan** o se **contraigan** de forma **dinámica** según el ancho o alto de la pantalla. Aunque en este apartado se resumen algunas de las características más importantes de flexbox, es muy recomendable echar un ojo a esta página, ya que explica muy bien la herramienta de principio a fin.



Al utilizar flexbox, habrá ciertas **propiedades** que se podrán aplicar al elemento **padre** y otras que se podrán aplicar al **hijo**. En esta explicación sólo mencionaremos las propiedades aplicables al padre. Veamos un ejemplo:

Lo primero es definir, en el CSS, **cuál** será el componente **padre**. Por ejemplo:

HTML

```
<div class="wrapper">
   <p> Este es el primer párrafo </p>
   <p> Este es el segundo párrafo </p>
</div>
```

CSS

```
.wrapper {
   display: flex;
}
```

Esto, lo que hará, será indicar que el **elemento** con la **clase wrapper** pasa a tener un **display** de tipo **flex** y todos los elementos que sean **hijos directos** pasarán a ser **flex items**, poniendo los dos párrafos en una **fila** (esta es la dirección que aplica flexbox por defecto). Podemos cambiar la dirección para que los elementos formen una **columna** utilizando la propiedad flex-direction y asignándole el valor column.

Si queremos modificar la **alineación** de los elementos en el **eje principal** (por ejemplo, el **eje horizontal** si es una **fila** y el **vertical** si se trata de una **columna**), tendremos que usar la propiedad justify-content, con la que podremos situar los objetos al principio, mitad o final del eje, o determinar la separación entre los objetos de forma simétrica con respecto al centro del eje:

- `justify-content: flex-start`

- `justify-content: center`

- `justify-content: flex-end`

- `justify-content: space-evenly`

- `justify-content: space-around`

- `justify-content: space-between`

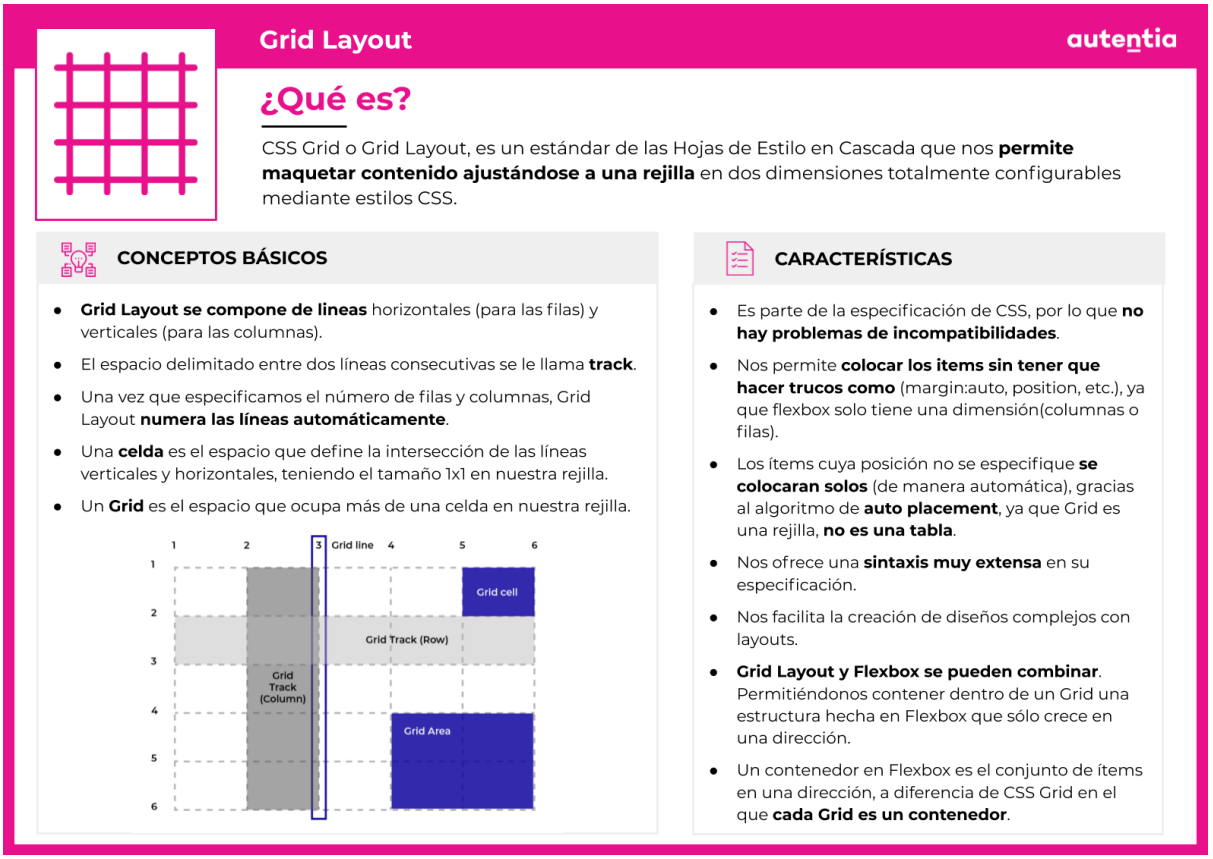
Si queremos modificar la alineación de los elementos en el **eje secundario** (por ejemplo, el eje vertical si es una fila), deberemos usar la propiedad align-items.

Grid

CSS Grid Layout es un sistema de layout **bi-dimensional** que permite

disponer el contenido en **filas y columnas** (grid significa cuadrícula). Para comenzar a definir un grid hacemos algo muy parecido que con flexbox: utilizamos la propiedad de CSS display.

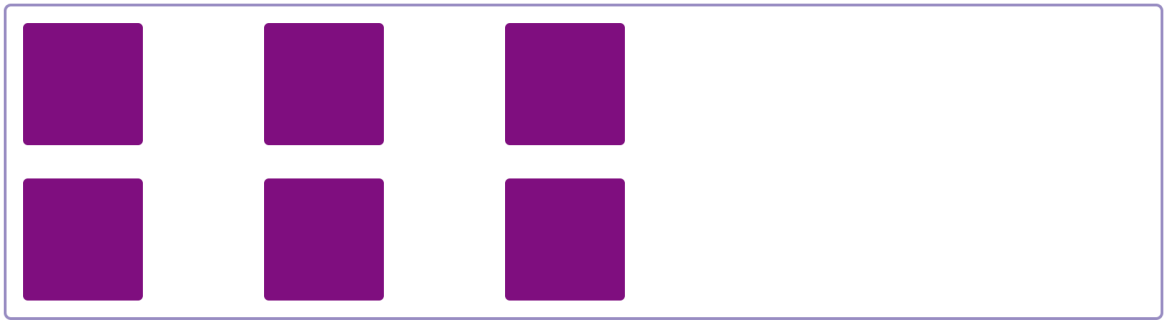
```
.wrapper {
   display: grid;
}
```



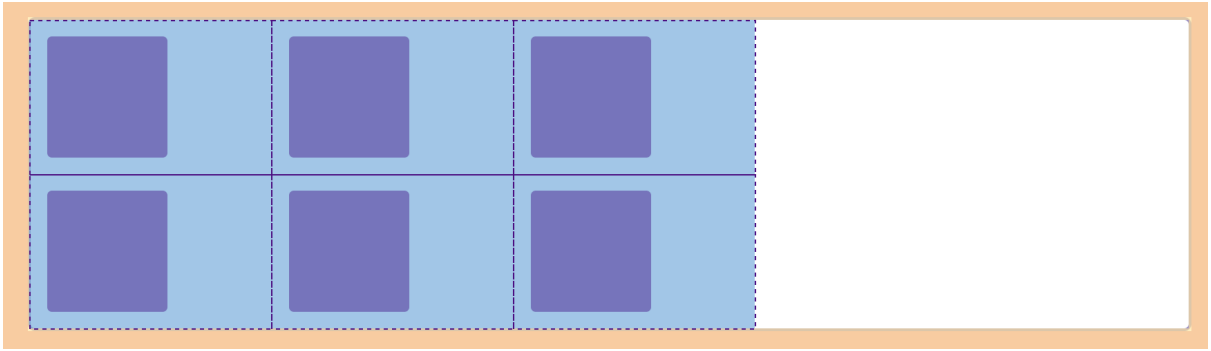
Al contrario que con flexbox, tendremos que seguir algún otro paso más antes de notar la diferencia visual. Esto es porque tras aplicar la propiedad anterior, el elemento de clase wrapper pasará a ser una cuadrícula de **una sola columna** y por tanto, no habrá diferencia visual. Para añadir más columnas, tendremos que añadirle la propiedad grid-template-columns al padre:

```
.wrapper {
   display: grid;
   grid-template-columns: 100px 100px 100px;
}
```

Al introducir esta línea, los hijos directos del elemento con clase wrapper se organizarán en 3 columnas de 100 píxeles cada una.

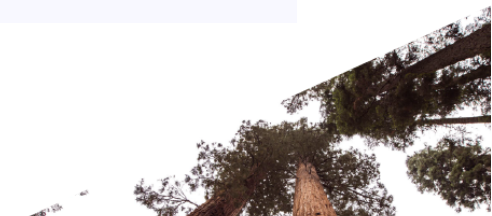


Si analizamos la cuadrícula con el inspector del navegador, se ve de la siguiente forma:

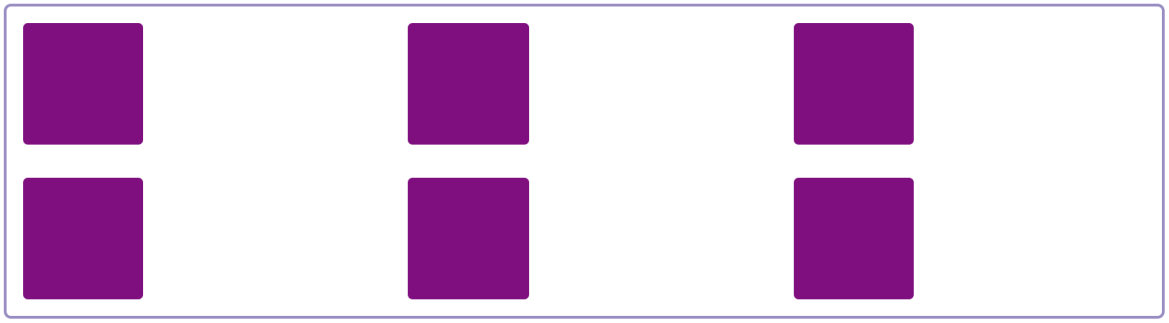


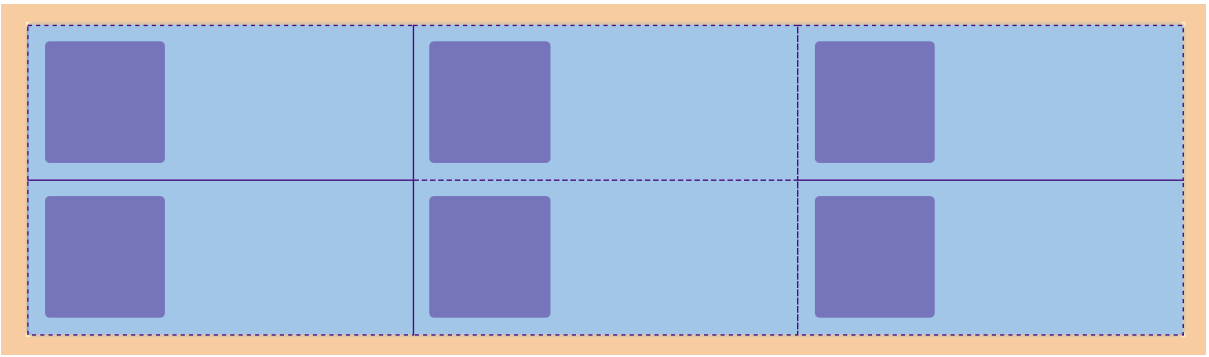
Además de poder crear grids usando longitudes y porcentajes, podemos usar fr para flexibilizar el tamaño de las filas o columnas. La unidad fr representa una **fracción** del **espacio libre** en el contenedor de la cuadrícula. Si tomamos el ejemplo anterior y cambiamos las unidades por estas, quedaría así:

```
.wrapper {
```

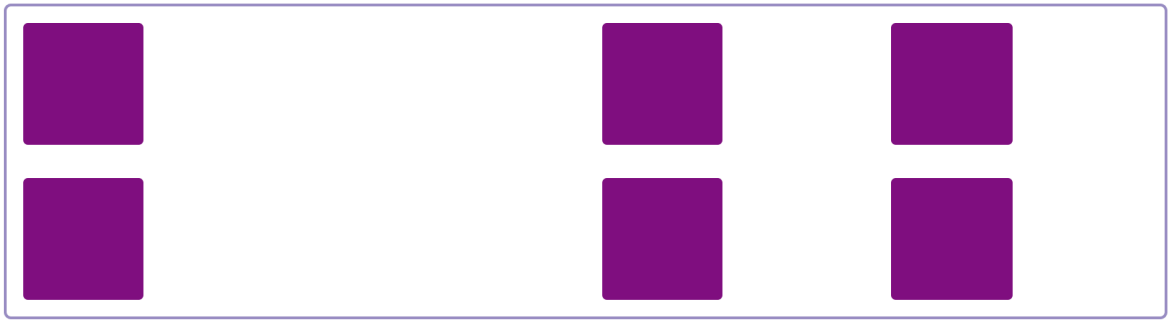


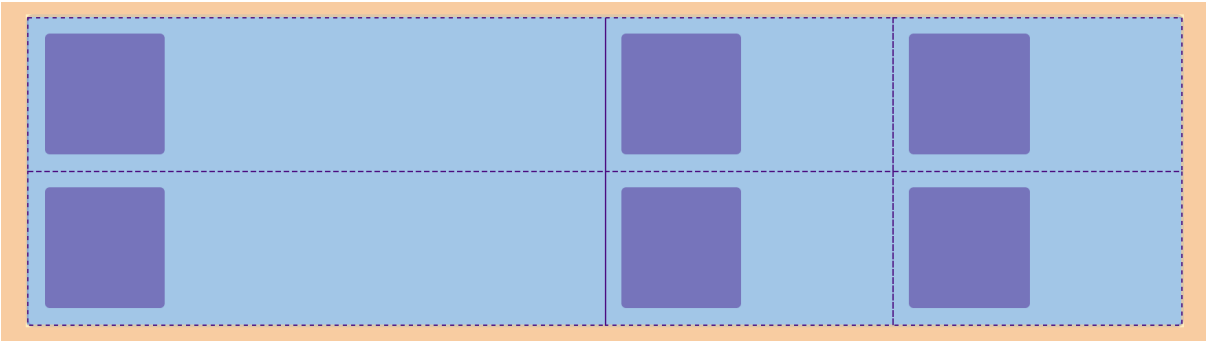
```
    display: grid;
    grid-template-columns: 1fr 1fr 1fr;
}
```





Esto, lo que hará, es coger el espacio (en este caso horizontal) restante y repartirlo en tres partes iguales que serán ocupados por cada una de las columnas. Esto funciona de forma **proporcional**. Es decir, si hubiéramos usado “grid-template-columns: 2fr 1fr 1fr;” se hubiera dividido el espacio en **cuatro fracciones**, **dos** de las cuales hubieran sido ocupadas por la **primera columna** que sería el doble de grande que las otras dos.





También podemos generar columnas, si son del mismo tamaño, usando repeat. De esta forma, estas dos expresiones producirían el mismo efecto:

- `grid-template-columns: 1fr 1fr 1fr`
- `grid-template-columns: repeat(3, 1fr)`

Para cambiar el **espaciado** entre las celdas de la cuadrículas se puede usar gap. Esta propiedad acepta unidades de distancia y porcentajes pero no una unidad fr.

También se puede especificar a qué columna y/o fila pertenece un elemento si a ese elemento hijo se le añade la propiedad grid-column y/o grid-row. Todo esto y otras muchas funcionalidades de grid están explicadas con más en detalle en esta página.

Custom properties

Las custom properties en CSS son, básicamente, **variables** en las que se almacenan **valores específicos** que se pueden **reutilizar** a lo largo de la página web. La adición de esta funcionalidad en CSS es algo muy importante ya que normalmente, en las páginas web hay mucho CSS y con muchos valores repetidos.

Declarando una variable

Declarar una variable es tan sencillo como nombrarla con **dos guiones** antes del nombre y asignarle el valor deseado:

```
element {
  --primary-color: brown;
}
```

El elemento donde se declare la variable indicará el scope dentro del cual se puede utilizar. Una práctica común es declarar variables en el elemento root, para que estén disponibles a lo largo de toda la aplicación. Por ejemplo:

```
:root {
  --primary-color: brown;
  --secondary-color: purple;

  --title-text-size: 22px;
  --body-text-size: 18px;
}
```

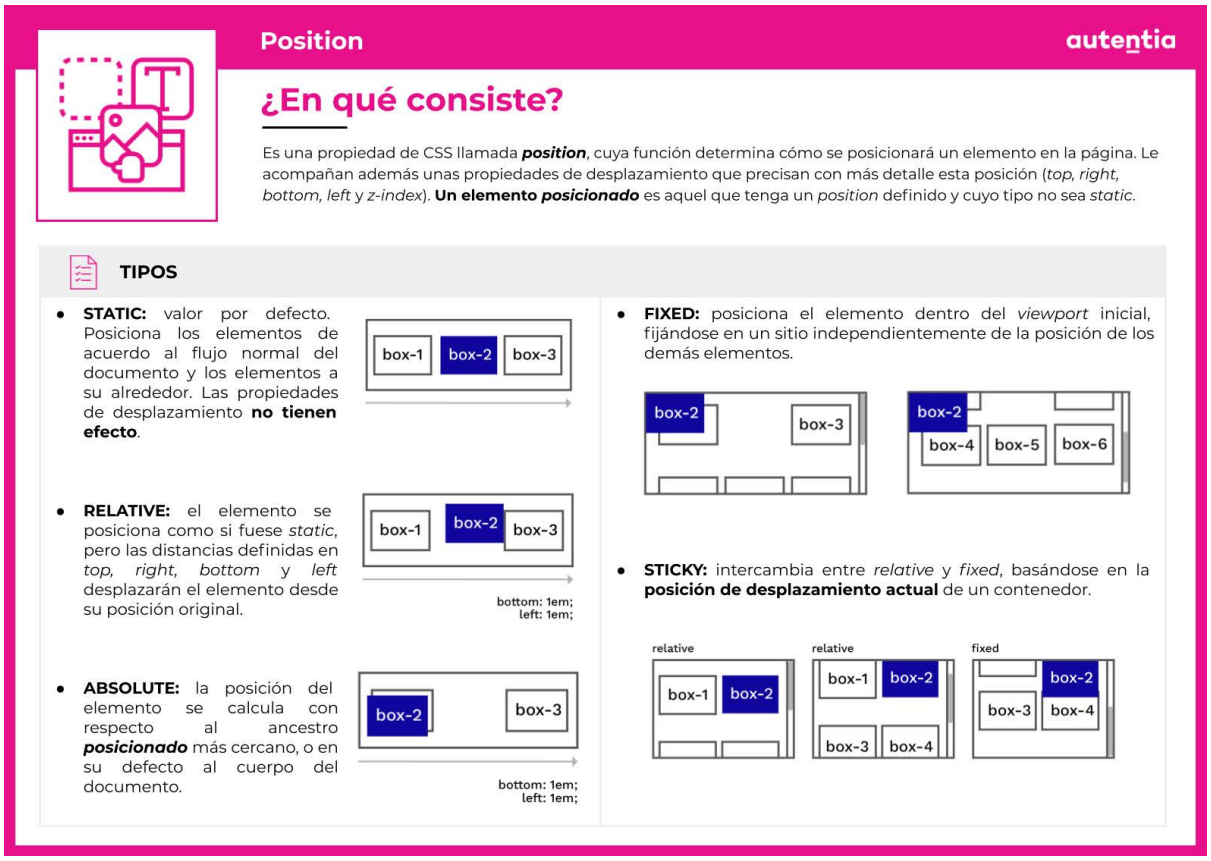
Utilizando una variable

Siguiendo el ejemplo del anterior apartado, para utilizar la variable primary-color, habría que utilizar la palabra “var” de la siguiente forma:

```
element {
  background-color: var(--primary-color);
}
```

Posicionamiento

La propiedad *position* se utiliza para posicionar un elemento en un documento. Hay distintos tipos de posicionamiento: *static*, *relative*, *absolute*, *fixed* y *sticky*. En este documento explicaremos los más utilizados.



Static

Este es el valor **por defecto** de la propiedad position y posiciona el objeto de acuerdo con el flujo normal del documento (es decir, el elemento se sitúa de forma normal y depende de la posición del resto de elementos de su alrededor). En este caso, las propiedades CSS top, right, bottom, left y z-index no tienen efecto.

Relative

Este valor posiciona el objeto de acuerdo con el flujo normal del documento pero modifica su offset con respecto a sí mismo, según los valores de top, right, bottom y left. El offset **no afecta a la posición de los elementos que le rodeen**. La propiedad z-index define la capa o altura en la que se encuentra un elemento.

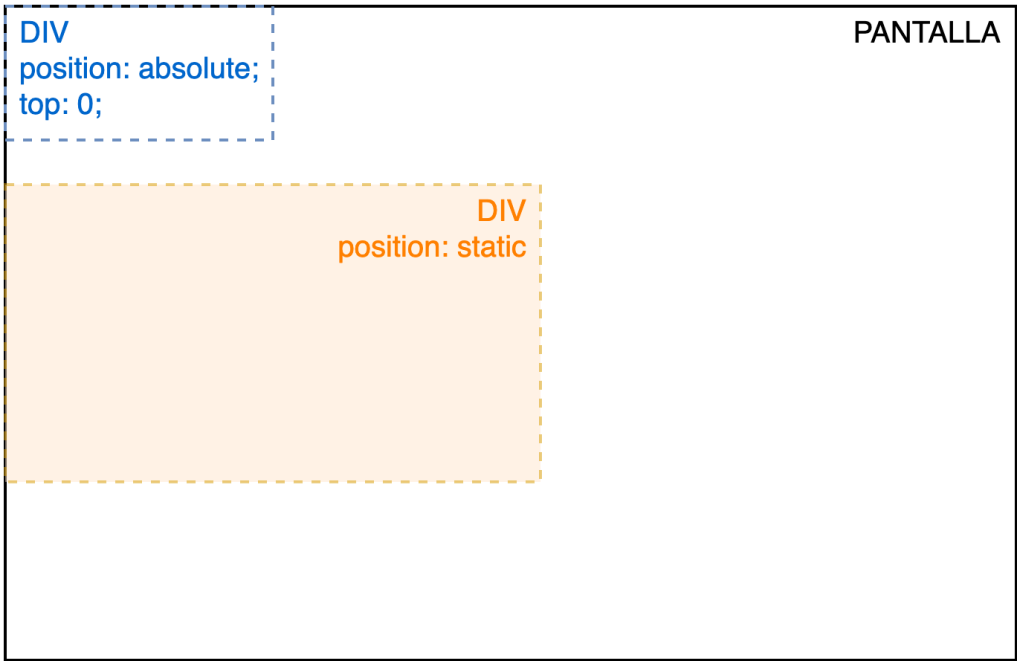
Absolute

Si establecemos la propiedad position de un elemento a absolute, **se extraerá del flujo normal** del documento y no se reservará ningún espacio para éste en la interfaz. El objeto se posicionará con respecto a su ancestro posicionado más cercano (es decir, al padre más cercano al que hayamos asignado un valor a position, siempre y cuando, el valor no sea static). Si no hay ninguno, se utiliza el cuerpo del documento.

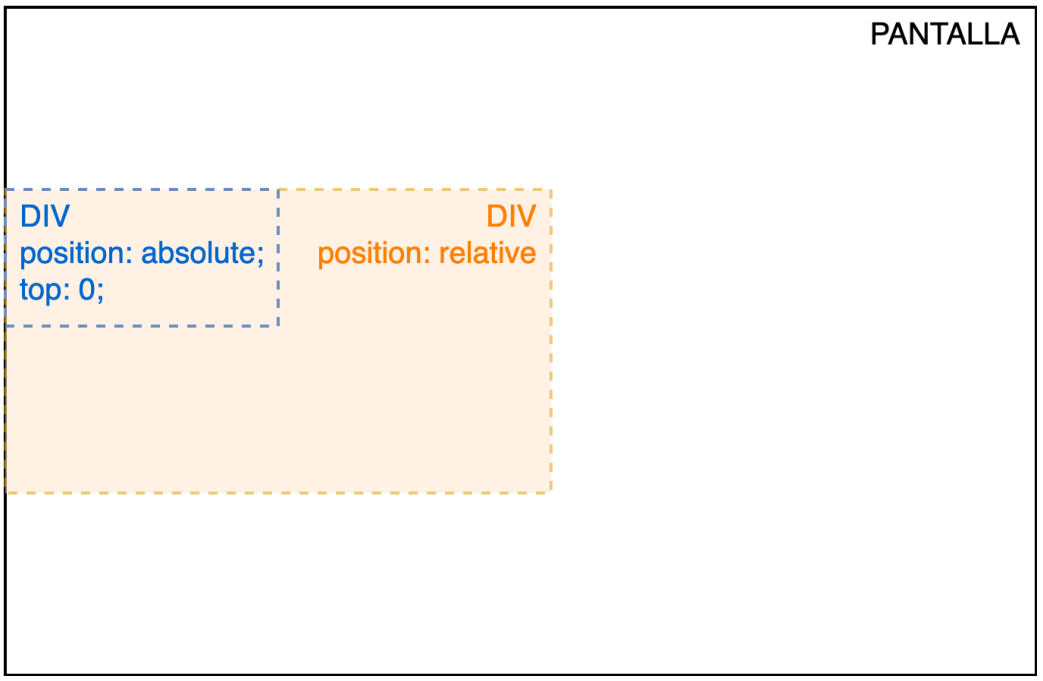
Por ejemplo, pongamos que tenemos un div y hay un elemento dentro al que aplicamos el siguiente CSS:

```
element {
   position: absolute;
   top: 0;
}
```

Lo que ocurrirá, es que este elemento se pegará a la parte superior de la pantalla:

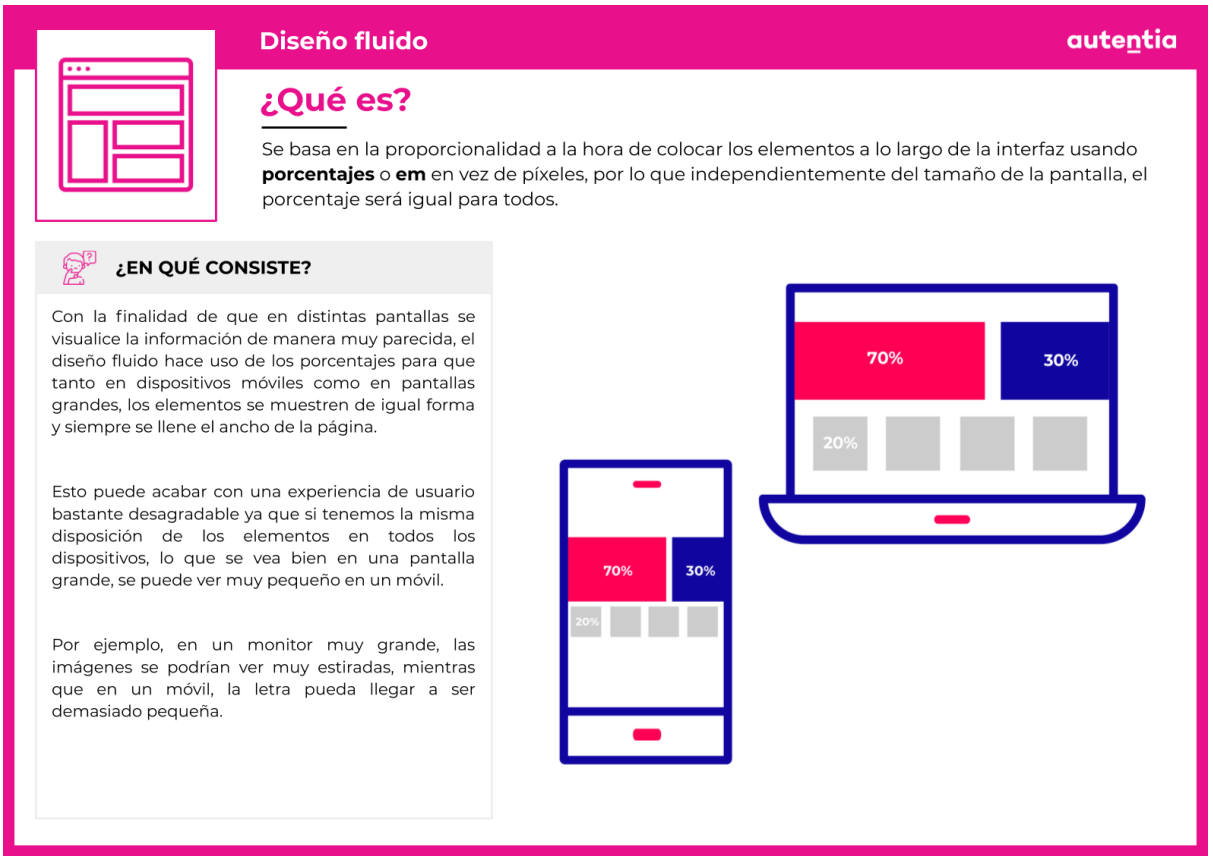


Sin embargo, si al div padre le añadimos la propiedad “position: relative” (o cualquier tipo de position que no sea static), éste se convertirá en un ancestro posicionado y por tanto, el elemento interno que tiene “position: absolute” se quedará contenido dentro de ese div:



Diseño fluido

El diseño fluido se basa en la **proporcionalidad** a la hora de colocar los elementos a lo largo de la interfaz, por lo que estos ocupan siempre el **mismo porcentaje** del espacio en diferentes tamaños de pantalla. Esto quiere decir que cuando se utilizan unidades de medida en CSS, se utilizan porcentajes.



Diseño responsive

A diferencia del diseño fluido, el diseño responsive usa **CSS Media Queries** para presentar **distintos layouts** dependiendo del tamaño o tipo de pantalla.

Responsive Web Design

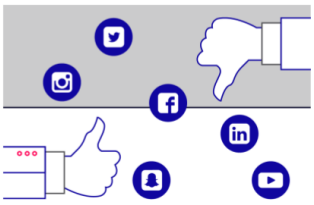
autentia

¿Qué es?

Es un **enfoque que se preocupa de desarrollar y diseñar sitios web** que puedan ajustarse a cualquier resolución, adaptando la fuente y las imágenes a cualquier dispositivo. Se intenta **que el usuario tenga una experiencia satisfactoria independientemente del dispositivo** que utilice para acceder.

¿POR QUÉ LO NECESITAS?

- **Mejorar la experiencia del usuario**. Según Google, los usuarios de móvil tienen una tasa de rebote del 61% frente al 67% de conversión, si tienen una buena experiencia.



- **El acceso a contenidos desde el móvil está en auge**. El uso de Internet desde el móvil es de más del 75% a nivel global y sigue subiendo gracias a la mejora del ancho de banda y los dispositivos.

- **Google favorece el posicionamiento** de los sitios con Responsive Design en sus búsquedas, ya que aumenta de forma natural el tráfico orgánico. Esta **actualización en el algoritmo de Google** recibió el apodo de **Mobilegeddon** (Mobile + Armagedón). Así que, si no lo haces por los usuarios, hazlo por tu posicionamiento SEO.

- **Aumenta la velocidad de carga**. Ya que son más ligeros y optimizados para móviles que una versión desktop.
- **Mejora la difusión por RRSS,** aumentando las ventas y la tasa de conversión.
- Responsive Web Design tiene un enfoque adaptativo, lo que nos **ofrece una ventaja competitiva al estar preparados para cualquier dispositivo**.

Responsive Web Design

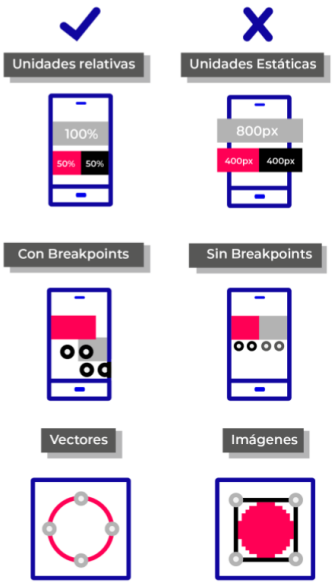
autentia

¿Cómo hacerlo?

Una aplicación o **sitio web *Responsive* debe tener un diseño flexible y capaz de adaptarse** a diferentes resoluciones de pantalla y dispositivos. Estas son **algunas técnicas que nos permiten adaptarnos mejor** a cada dispositivo para así ofrecer una experiencia satisfactoria a los usuarios finales.

TÉCNICAS BÁSICAS

- **Cambiar el tamaño de la caja,** de *box-sizing* a *border-box* para evitar que cada elemento añada sus propiedades de tamaño.
- **Usar la etiqueta meta** name="viewport" content="width=device-width initial-scale=1", **indica a la página el ancho de la pantalla en píxeles independientemente del dispositivo**. También se pueden establecer atributos como minimum-scale, maximum-scale, user-scalable.
- **Hacer uso de CSS Layouts que nos permiten la creación de diseños fáciles y flexibles** como Grid Layout, Flexbox o Multicol.
- **Definir puntos de ruptura**. Son expresiones condicionales que aplican diferentes estilos dependiendo del dispositivo. Esto se hace utilizando una herramienta llamada **Media Queries**.
- **Usar imágenes vectoriales SVG.** La imagen no pierde calidad al redimensionarse.
- **Envolver objetos en un contenedor**. Esto hace un diseño comprensible, limpio y ordenado.
- En el ciclo de desarrollo hay que tener presente que se va a acceder al sitio o la aplicación desde **diferentes dispositivos con distintos tipos de pantalla y resoluciones.**



Por ejemplo, pongamos que tenemos el siguiente CSS:

```
@media only screen and (max-width: 600px) {
  body {
    font-size: 40px;
  }
}
```

Lo que pasará, es que cuando el ancho de la pantalla sea **menor** de 600px, se modifica el tamaño de la fuente del elemento body a **40px**. Este es un ejemplo muy simple pero esta herramienta es muy útil y se puede usar para, por ejemplo, reorganizar la interfaz en función del tamaño de pantalla.

Accesibilidad

La accesibilidad hace que los sitios web puedan ser utilizados por el mayor número de personas posible.

El principal objetivo de la accesibilidad web es el de permitir a cualquier usuario, independientemente del tipo de discapacidad que tenga, el acceso a los contenidos del sitio y permitirle la navegación necesaria para realizar las acciones deseadas. Si nuestra página no es accesible sucederán dos cosas: **perderemos muchos usuarios** que no entenderán nuestra página y por otro lado, los que no perdamos **sufrirán** para poderse manejar con el contenido de la página. Además, aunque no lo parezca, una **gran cantidad de usuarios** tienen algún tipo de discapacidad.

Soluciones tecnológicas (AT)

Las discapacidades que puede tener un usuario que acceda a nuestra web son **variadas**. Entre ellas podemos encontrar discapacidades visuales, auditivas, motrices, cognitivas... Estos usuarios utilizan las llamadas **tecnologías de apoyo** o assistive technologies (**AT**). Por lo general, una regla que se debería cumplir para funcionar correctamente junto con las AT es que los **controles** de la página sean **accesibles por el teclado** u otro tipo de dispositivo de asistencia, sin necesidad de usar el ratón.

¿Qué tipos de discapacidades hay y qué **herramientas** se utilizan según el tipo de discapacidad?

Visuales

Las personas con discapacidades visuales utilizan un **lector de pantalla**. Lo que éste hace es leer el contenido de la pantalla a una **gran velocidad** y permitir al usuario navegar e interactuar con la página web.

Sin embargo, esto **depende** también de **cuán accesible sea la página web**, ya que si nuestra página no es accesible, el lector de pantalla tendrá más dificultades para leer el contenido y los elementos de la pantalla según el usuario vaya navegando y por tanto, le costará mucho más usarla.

Auditivas

Realmente no hay AT específicos para las personas con este tipo de discapacidad que estén orientados al uso del ordenador/web. Aún así, hay **técnicas específicas** para ofrecer alternativas textuales a contenidos de audio que van desde **simples transcripciones,** hasta **subtítulos** que se pueden mostrar junto con el vídeo.

Motrices

Las discapacidades motrices pueden implicar **problemas puramente físicos** (como la pérdida de una extremidad o la parálisis) o **trastornos neurológicos/genéticos** que conllevan la debilidad o pérdida de control en las extremidades.

Algunas personas, simplemente pueden tener dificultades a la hora de mover el ratón, mientras que otras podrían verse más gravemente afectadas, tal vez estén paralizadas y necesiten utilizar un puntero de cabeza para interactuar con el ordenador. Esto quiere decir que es probable que estos usuarios tengan **limitaciones en cuanto al hardware** (algunos usuarios podrían no tener un ratón).

Cognitivas

Este tipo de discapacidad engloba una **amplia gama** de discapacidades, desde las personas que presentan capacidades intelectuales más limitadas, hasta toda la población que tiene problemas a la hora de recordar derivados de la edad, u otros.

Aunque dentro de este conjunto hay una amplia gama de discapacidades, todas ellas tienen un **conjunto común de problemas funcionales** en cuanto a páginas web se refiere que incluye dificultades a la hora de **entender** los contenidos, **recordar** cómo completar las tareas y **confusión** ante páginas web diseñadas de forma incoherente.

Una **buena base** de accesibilidad para personas con **deficiencias cognitivas** incluye:

- Proporcionar el contenido en más de un formato, como puede ser texto-a-voz o vídeo.
- Proporcionar contenidos fáciles de entender, como texto escrito con estándares de lenguaje sencillo.
- Centrar la atención en el contenido importante.
- Minimizar las distracciones, tales como contenidos innecesarios o anuncios.
- Proporcionar un diseño coherente de la página web y del sistema de navegación.
- Usar elementos ya conocidos, como los enlaces subrayados en azul cuando aún no se han visitado y en morado cuando sí.
- Dividir los procesos en pasos lógicos y esenciales con indicadores de progreso.
- Ofrecer un sistema de autenticación del sitio web de la forma más fácil posible sin comprometer la seguridad.

- Diseñar formularios fáciles de completar, con mensajes de error claros y de fácil solución.

Normativas y estándares

En Europa y EE.UU. hay estándares que definen los requerimientos de accesibilidad para los productos/servicios de tecnologías de la información y comunicación. Estos son el estándar **EN 301 549** para Europa y la **Section 508** para EE.UU. Ambas se alinean con las pautas o guías que propone el WCAG (2.0 en el caso de EEUU y 2.1 en el caso de Europa).

WCAG 2.1

Aunque tiene unas pautas muy extensas, se pueden resumir en ciertos objetivos generales. Tu página web ha de ser:

- **Perceptible**: la información y los componentes de la interfaz de usuario deben ser mostrados a los usuarios en formas que ellos puedan entender.
- **Operable**: los componentes de la interfaz de usuario y la navegación deben ser manejables, aunque sólo sea con el teclado u otros tipos de métodos de entrada.
- **Comprensible**: la información y las operaciones de usuarios deben ser comprensibles.
- **Robusta**: el contenido debe ser suficientemente robusto para que pueda ser bien interpretado por una gran variedad de agentes de usuario, incluyendo tecnologías de asistencia.

Técnicas básicas de HTML y CSS

Hay **numerosas técnicas** para facilitar la accesibilidad, tanto para HTML como para CSS. Alguna de estas puede ser:

- Esconder mediante CSS texto que indique el objetivo de, por ejemplo, un link o similar de forma que se pueda disponer de esa información.
- Posicionar el contenido con CSS basándose en la estructura del HTML utilizado. Esto quiere decir que si el CSS no está disponible o no es legible por el dispositivo AT, el usuario debe poder seguir determinando el significado del contenido. Por ejemplo, si se utiliza el CSS para colocar cierto contenido en ciertas partes de la pantalla pero el HTML usado no concuerda estructuralmente con esta disposición, si el CSS no estuviera disponible sería extremadamente difícil comprender la página web y la organización o significado de su estructura y contenido, porque este estaría “desperdigado”. Por esto también es muy importante dar preferencia al HTML semántico frente al no semántico.
- Utilizar la propiedad alt en las imágenes.
- Usar notación de tablas en HTML para representar datos tabularmente (por ejemplo, no utilizar sólo divs y mostrar los datos como una tabla mediante CSS.

Hay **muchas otras técnicas** aparte de estas, y se pueden encontrar en estas páginas:

- Técnicas de CSS
- Técnicas de HTML

Parte 2

JavaScript

Introducción

JavaScript es un lenguaje de scripting basado en ECMAScript que puede ejecutarse tanto en el lado del cliente como en el del servidor y permite crear contenido dinámico en una web. Es un lenguaje de programación orientado a objetos basado en prototipos, dinámicamente tipado.

ECMAScript es una especificación de lenguaje de programación que publica la organización ECMA, la implementación más famosa es JavaScript, pero existen otras como ActionScript o JScript. ECMAScript llegó al punto de madurez en 2015 con la publicación de ECMAScript 6, que trajo consigo un gran número de novedades y pulido general al lenguaje. Todos los navegadores modernos incluyen una implementación del estándar ECMAScript.

JavaScript, desde su concepción, siempre ha estado ligado al navegador, siendo uno de los pilares del desarrollo Front-End. Mediante este lenguaje podemos mostrar un pop up al usuario, enviar un mensaje, actualizar los datos de la página en tiempo real, etc. Con la aparición de entornos de ejecución, como Node.js, ha sido posible extender el uso de JavaScript al desarrollo Backend y crear herramientas auxiliares al desarrollo.

"JavaScript desde su concepción siempre ha estado ligado al navegador, siendo uno de los pilares del desarrollo front-end."

Tipos básicos

JavaScript es dinámicamente tipado, es decir, cuando declaramos una variable no podemos especificar de qué tipo es, sino que lo hará JavaScript por detrás. Además, una variable puede ser de un tipo en un instante y más tarde cambiar de tipo, según el valor que se le asigne.

En JavaScript existen varios tipos básicos de datos, pudiendo separarse en tres categorías: tipos **primitivos** (String, Number y Boolean), **compuestos o referencias** (Object, Array, Function) y **especiales** (Undefined y Null).

Para saber el tipo de una variable en un momento determinado, podemos usar el operador *typeof*:

```
var a = "Hello World!";
console.log(typeof a); // Output = "string"
```

String

Este tipo, común con otros muchos lenguajes, se usa para representar datos textuales. Se pueden crear usando tanto **simples comillas** como **dobles**:

```
var a = "Hello World!";
var a = 'Hello World!';
```

Se pueden también usar comillas dentro de los strings, siempre y cuando no coincidan con el tipo de las externas:

```
var a = "Hey, what's up?";
var a = 'Dijo que era "muy poco estable" para usarlo en
producción';
```

O escapar el símbolo:

```
var a = 'Hey, what\'s up?';
```

Number

En JavaScript no existen distintos tipos de variable para cada tipo de número como en otros lenguajes (int, double, float...) sino que se engloban todos en uno sólo: Number. Estos serían varios ejemplos de cómo se puede declarar:

```
var a = 15;
var a = -15;
var a = 20.5;
var a = 20.5e+4; // a = 205000
var a = 20.5e-4; // a = 0,00205
```

El tipo Number también incluye algunos **valores especiales** para casos concretos: Infinity, -Infinity y NaN (Not a Number).

```
var a = 16 / 0; // a = Infinity
var a = -16 / 0; // a = -Infinity
var a = "someStr" / 16; // a = NaN
var a = Math.sqrt(-1); // a = NaN
```

BigInt

BigInt es un objeto que ofrece una forma de representar números enteros mayores que 2^{53} - 1, que es la cifra más grande que JavaScript puede representar con un Number. Esta cifra se representa con la constante `Number.MAX_SAFE_INTEGER`.

BigInt es parecido a Number en algunas cosas, pero difiere en ciertas características clave:

- No se puede usar en métodos de Math.
- No se puede mezclar con variables de tipo Number en operaciones (hay que pasarlos al mismo tipo aunque hay que tener cuidado porque se puede perder precisión al hacer las conversiones).

Para crear una variable de tipo BigInt sólo hace falta añadir una “n” al final de un entero (`const myBigInt = 900719925474099n`) o llamando a la función BigInt():

```
const alsoHuge = BigInt(9007199254740991);
const hugeString = BigInt("9007199254740991");
const hugeHex = BigInt("0x1fffffffffffff");
const hugeBin =
BigInt("0b11111111111111111111111111111111111111111111111111111");
```

Boolean

Este tipo sólo puede albergar los valores true o false:

```
var a = true;
var a = false;
```

También podemos asignar un valor a una variable de tipo Boolean asignándole el resultado de una comparación o de cualquier operación booleana:

```
var a = 1 > 16; // a = false
var a = 1 < 16; // a = true
```

Undefined

Este tipo sólo puede albergar el valor *undefined*. Una variable que haya sido declarada pero a la que no se le haya asignado ningún valor, tendrá el valor *undefined*.

```
var a;
console.log(a); // a = undefined
```

Null

Este, al igual que undefined, es otro tipo que sólo puede albergar el valor *null*. Un valor que sea *null*, quiere decir que no tiene valor. Se puede vaciar una variable de su contenido explícitamente si se le asigna el valor *null*:

```
var a = null; // a = null
var b = "Hello World!";
b = null; // b = null
```

Normalmente, en JavaScript no se utiliza el valor *null*, al contrario que *undefined*, cuya utilización es muy común.

Object

Este es un tipo de dato complejo que permite almacenar colecciones de datos:

```
var emptyObject = {};
var car = {
   model: "BMW X3",
   color: "red",
   doors: 5,
};
```

Los nombres de las propiedades (model, color, doors) se pueden escribir sin comillas, como en este ejemplo, siempre y cuando el nombre sea un nombre válido de JavaScript. Esto quiere decir que si alguna propiedad tiene un nombre con un guión, como por ejemplo *owner-name*, o coincide con una palabra reservada, habrá que añadirle comillas:

```
var car = {
   "owner-name": "foo",
   model: "BMW X3",
   color: "red",
   doors: 5,
};
```

También, si queremos usar el valor de una variable como nombre de una propiedad, usaremos corchetes:

```
var propertyName = "model";

var car = {
   [propertyName]: "BMW X3",
   color: "red",
   doors: 5,
};
```

Array

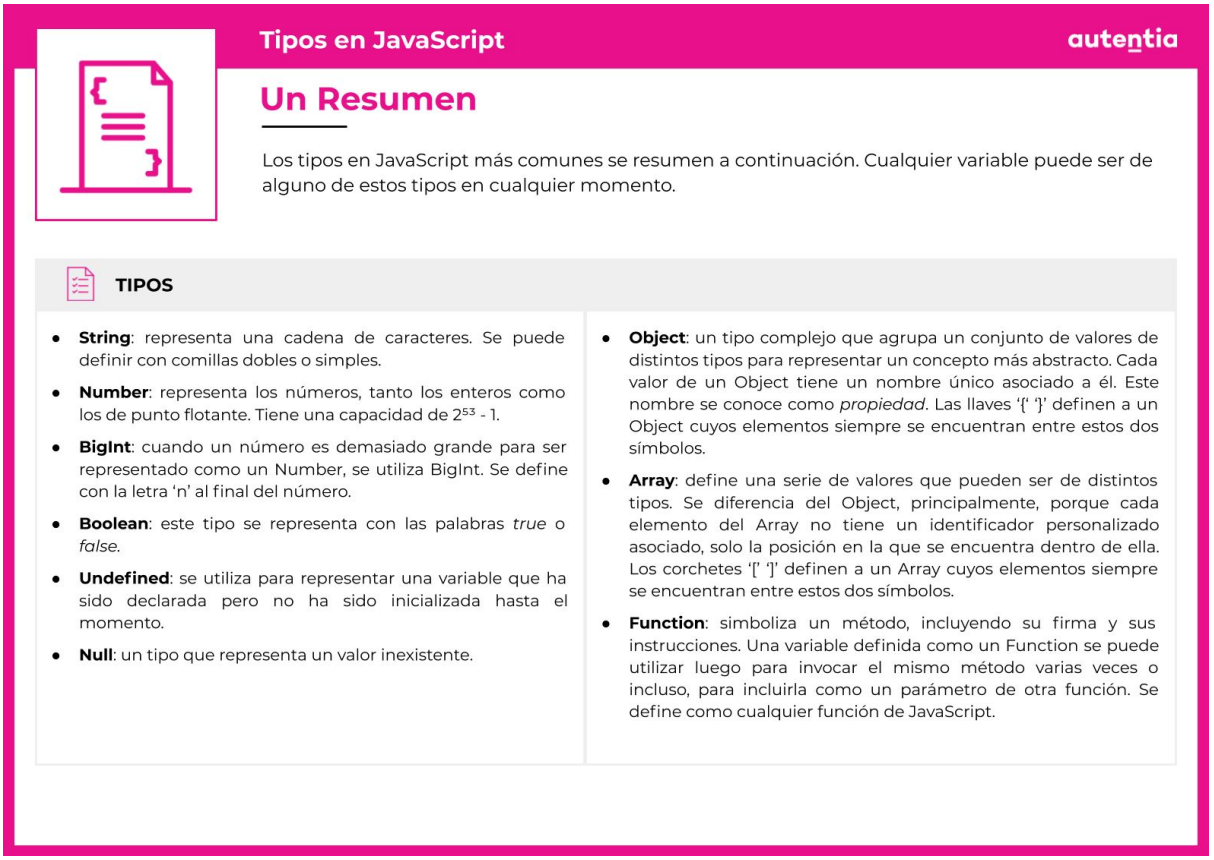
Este tipo de variable se usa para almacenar varios valores en una sola variable:

```
var colors = ["red", "yellow", "purple", "white"];
```

Function

Las funciones son objetos invocables que ejecutan un bloque de código y se pueden asignar a una variable o declararlas como normalmente se hace en otros lenguajes:

```
var greeting = function() { return "Hello World!"; }
function greetAgain() { return "Hello again, World!"; }
console.log(greeting()); // Output: Hello World!
console.log(greetAgain()); // Output: Hello again, World!
```



Variables y operadores

En JavaScript, como en otros lenguajes de programación, usamos las variables para almacenar valores. En los siguientes capítulos se van a mostrar las distintas formas de definir variables y de trabajar con los valores que tienen.

Aparte de los operadores básicos (aritméticos y booleanos), JavaScript tiene otros operadores, como el *optional chaining* o el *nullish coalescing*, que ofrecen azúcar sintáctico para hacer más fácil la manipulación de los objetos. También hay operadores que ayudan en el trabajo con estructuras de datos, como la desestructuración y el operador *spread*.

Constantes

En JavaScript, cuando queremos declarar una constante, lo hacemos de la siguiente manera:

```
const a = "Hello World!";
```

Podemos declarar como constante cualquier tipo de dato y **siempre** es aconsejable declarar una variable como constante, excepto en el caso de que la vayamos a modificar posteriormente.

Variables con let

Como hemos visto en el anterior apartado, **siempre** se debería declarar una variable como *const*, excepto en los pocos casos en los que vayamos a

modificarla. Para esos casos, existen otras dos maneras de declararla: *var* (como se ha visto en los ejemplos) y *let*:

```
let a = 5;
```

A efectos prácticos, *let* y *var* son muy similares pero **su scope varía**: el scope de **let** se limita al **bloque** de código, mientras que el de **var** se limita a la **función**. Veamos esto con dos ejemplos.

```
console.log(a);
var a = 5;
console.log(a);
```

En este caso, el primer console.log imprimirá *undefined* y el segundo el valor 5. Si en vez de **var** utilizáramos **let**, el primer console.log lanzará un error: *"Cannot access 'a' before initialization"*.

Ahora, vamos a ver otro ejemplo:

```
if(true) {
   var a = 5;
} else {
   a = 10;
}

if(true) {
   let b = 5;
} else {
   b = 10;
}

console.log(a); // Output: 5
console.log(b); // Error: b is not defined
```

De hecho, en el segundo caso nos dirá el error mientras programamos y nos avisará de que desde fuera del bloque *if* no puede encontrar la variable *b* que estamos declarando dentro del bloque. Esto se debe a que, como

hemos explicado antes, el scope de *let* se limita al bloque, no a la función como *var*.

Una vez explicada esta diferencia, y aunque por comprensión en los ejemplos de las secciones previas se utilizó *var*, es importante decir que **siempre debemos usar let o const** y **nunca var**. Las variables *let* fueron incluidas en 2015 y desde entonces es recomendable utilizar estas en lugar de *var*. Esto hará más sencillo identificar posibles errores, como por ejemplo, si se produce el llamado hoisting (con *var*, podríamos utilizar una variable antes de haberla declarado, mientras que con *let* no).

Template literals

Los template literals son una forma de declarar el **string** que permite añadir **valores embebidos** en él. El string se declara con acentos graves (`) y la variable se envuelve entre llaves precedidas por el símbolo del dólar:

```
const name = "Pedro";
const yearsOld = 34;
console.log(`Su nombre es ${name} y tiene ${yearsOld} años`);
// Output: Su nombre es Pedro y tiene 34 años
```

Property shorthand

Cuando en JavaScript declaramos un objeto utilizando variables como valores, normalmente lo haríamos de la siguiente manera:

```
const name = "Pedro";
const yearsOld = 34;
const job = "Developer";

const person = {
   name: name,
   yearsOld: yearsOld,
```

```
    job: job
}
```

Aquí es donde entra en juego esta funcionalidad. Si el nombre de la propiedad del objeto coincide con el nombre de la variable que se está usando (como es el caso), se puede simplemente declarar de la siguiente manera:

```
const person = {
    name,
    yearsOld,
    job
}
```

Y funcionará exactamente igual que si pusiéramos la palabra dos veces.

Property methods

En un objeto de JavaScript no sólo se pueden introducir propiedades, sino también métodos:

```
const person = {
    firstName: "Pedro",
    lastName: "Martínez",
    fullName() {
        return `${this.firstName} ${this.lastName}`
    }
}
```

O también se puede declarar de la siguiente manera:

```
const person = {
    firstName: "Pedro",
    lastName: "Martínez",
    fullName: function() {
        return `${this.firstName} ${this.lastName}`
```

```
    }
}
console.log(person.fullName()); // Output: Pedro Martínez
```

Parámetros por defecto

En las funciones o métodos de JavaScript se pueden usar parámetros por defecto así:

```
function  multiply(a, b = 1) {
    return a * b;
}

console.log(multiply(5, 2)); // Output: 10
console.log(multiply(5)); // Output: 5
```

Arrow function

Se podría decir que las *arrow functions* de JavaScript son algo similares a las funciones lambda de otros lenguajes como Java o Python.

Estas funciones permiten que en vez de declarar una función así:

```
function() {
    return "Hello world";
}
```

Se pueda declarar así:

```
() => {
    return "Hello world";
}
```

Y si además la función sólo tiene un *statement*, se puede obviar las llaves,

además del *return* (el resultado se devuelve automáticamente):

```
() =>  "Hello world";
```

Desestructuración

La desestructuración (o *destructuring*) en JavaScript, se puede aplicar tanto a arrays como a objetos y permite declarar varias variables a la vez y asignar valores de un array o un objeto a estas, todo en una misma línea.

Arrays

En un array, para asignar sus valores a unas variables de forma tradicional, haríamos lo siguiente:

```
const names = ["Pedro", "Martínez"];
const firstName = names[0];
const lastName = names[1];
```

Pero gracias a la desestructuración, esto se puede hacer en una línea, envolviendo las variables entre corchetes. En el caso de los arrays, la asignación a las variables depende del orden:

```
const [firstName, lastName] = ["Pedro", "Martínez"];
console.log(firstName); // Output: Pedro
console.log(lastName); // Output: Martínez
```

Como se puede ver, el primer valor del array se asigna a la variable que hayamos declarado en primer lugar, el segundo valor a la segunda variable, y así sucesivamente.

Objetos

En el caso de los objetos sin embargo, esto no depende de la posición, sino del nombre de la variable. La forma tradicional de asignar variables de un objeto a otras variables sería de la siguiente forma:

```
const person = {
   firstName: "Pedro",
   lastName: "Martínez"
}

const firstName = person.firstName;
const lastName = person.lastName;
```

Y aplicando la desestructuración, se podría resumir en la siguiente línea, envolviendo las variables entre llaves (en lugar de corchetes como en los arrays):

```
const { firstName, lastName } = person;
console.log(firstName); // Output: Pedro
console.log(lastName); // Output: Martínez
```

Operador Spread

Este operador, que se utiliza con 3 puntos (...), sirve para hacer una copia del contenido de un objeto o un array. Esto también nos permite componer un objeto o un array juntando varios.

Arrays

```
const numbers1 = [1, 2, 3];
const numbers2 = [4, 5, 6];
const allNumbers = [...numbers1, ...numbers2];
console.log(allNumbers); // Output: [1, 2, 3, 4, 5, 6]
```

Aparte de esto, se pueden combinar el operador *Spread* y la desestructuración vista en el anterior apartado. De esta forma, si hacemos lo siguiente:

```
const [num1, num2, ...rest] = [1, 2, 3, 4, 5];
```

Lo que estamos diciendo es que queremos asignar el primer valor del array a num1, el segundo a num2 y el resto de valores que haya en el array, a la variable rest.

```
console.log(num1); // Output: 1
console.log(num2); // Output: 2
console.log(rest); // Output: [3, 4, 5]
```

Podemos hacer algo similar con los objetos, como veremos a continuación.

Objetos

En los objetos sucede algo parecido a los arrays. Podemos hacer lo siguiente:

```
const personNames = {
   firstName: "Pedro",
   lastName: "Martínez"
}

const personalData = {
   job: "developer",
   bornIn: "10-3-93",
   country: "Spain"
}

const person = {...personNames, ...personalData};
console.log(person);
/* Output:
{
```

```
   firstName: "Pedro",
   lastName: "Martínez",
   job: "developer",
   bornIn: "10-3-93",
   country: "Spain"
} */
```

¡Cuidado! Al hacer Spread, si utilizamos una propiedad que ya existe, se asignará el último valor dado. En el ejemplo vemos que la propiedad country es sobreescrita por “Andrómeda” en lugar de “Spain”.

```
const person =
{...personNames, ...personalData, country: "Andrómeda"}
console.log(person);
/* Output:
{
   firstName: "Pedro",
   lastName: "Martínez",
   job: "developer",
   bornIn: "10-3-93",
   country: "Andrómeda"
} */
```

Y, al igual que con los arrays, si combinamos el operador *Spread* y la desestructuración, podemos asignar las propiedades que queramos a ciertas variables y el resto de propiedades a la variable restante:

```
const { firstName, job, ...otherProps } = person;
console.log(firstName); // Output: Pedro
console.log(job); // Output: developer
console.log(otherProps);
/* Output:
{
   lastName: "Martínez"
   bornIn: "10-3-93",
   country: "Spain"
} */
```

Estructuras de datos

Map

Un Map es un objeto que guarda parejas de clave-valor, recordando también el orden en que estas parejas se van insertando. Podemos usar cualquier tipo de variable u objeto ya sea como clave o como valor.

Ofrece distintos métodos, como puede ser *set*, con el que insertamos una nueva pareja clave-valor en el Map:

```
const myMap = new Map();
myMap.set(1, "Pedro");
```

O *get*, con el que recogemos un valor buscándolo por su clave:

```
console.log(myMap.get(1)); // Output: Pedro
```

También hay otros métodos, como por ejemplo para recoger un array que contenga las claves, los valores o ambos.

Set

Un Set es una colección de **valores únicos**. Ofrece diversos métodos como *add(value)*, *has(value)* (devuelve un booleano indicando si *value* existe en el Set), *values()*...

WeakMap

Un WeakMap es similar a un Map, con la diferencia de que las claves sólo pueden ser objetos. Además, las referencias a los objetos de las claves son débiles y de esta forma, si no hay ninguna otra referencia a ese objeto, no impide la recolección de basura. Esto conlleva también que las claves o valores no se almacenan en un array o similar y por tanto, el WeakMap no tiene ningún método que nos devuelva un array con las claves, valores o ambos (al contrario que el Map).

WeakSet

Un WeakSet es similar a un Set, pero sólo puede almacenar objetos y sigue la misma estrategia que el WeakMap en cuanto a referencias débiles y recolección de basura.

Optional chaining operator

Este operador permite acceder a las propiedades de un objeto sin tener que validar que el objeto no sea *undefined* o *null* previamente. Si la propiedad a la que se intenta acceder es *undefined* o *null*, no se intentará acceder a la propiedad y el operador devolverá automáticamente *undefined*, con lo que se evitan errores en tiempo de ejecución. También se puede encadenar el operador si se quiere acceder a una *deep property*. Se utiliza de la siguiente manera:

```
let person;
console.log(person?.name); // Output: undefined

person = {
   name: "Pedro"
```

```
}
console.log(person?.name); // Output: Pedro
console.log(person?.address?.country); // Output: undefined

person = {
   name: "Pedro",
   address: {
      country: "Spain",
      province: "Madrid"
   }
}
console.log(person?.address?.country); // Output: Spain
```

Nullish coalescing operator

El nullish coalescing operator es un operador que retorna su operando derecho si el izquierdo es *null* o *undefined*. En el resto de casos, devuelve el operando izquierdo. Se utiliza tal que así:

```
let a;
console.log(a ?? "foo"); // Output = foo
a = "bar"
console.log(a ?? "foo"); // Output = bar
```

Este operador viene muy bien para asignar valores por defecto a ciertas variables en caso de que el valor que normalmente se les asigne resulte ser *null* o *undefined*.

Asincronía

La asincronía se produce cuando una función espera un resultado pero no sabe cuándo le llegará. Por tanto, no se puede ejecutar el código línea a línea, sino que la ejecución prosigue su curso y cuando la respuesta llegue, se realizarán las acciones que sean necesarias. La estrategia para abordar la asincronía en JavaScript ha ido evolucionando a lo largo del tiempo.

Callbacks

Esta idea es simple: pasarle un callback como parámetro a una función. De esta forma, la función internamente ejecutará ese callback cuando haya terminado su propia ejecución (o cuando sea necesario). Por ejemplo, si usamos el método que ofrece la librería 'fs' para leer el contenido de un directorio, tenemos que declarar lo que queremos hacer con el resultado mediante un callback:

```
fs.readdir(source, (err, files) => {
   if(err) {
      console.log(err);
   } else {
      // Do something with the files
   }
});
```

De esta forma, podemos decirle a esta función qué queremos hacer con la información cuando esté disponible.

Esta mecánica no es la mejor, siendo una de las razones el llamado *callback hell*. El *callback hell* es el resultado de ir anidando callbacks, lo que resulta en una pirámide horizontal. Por ejemplo:

```
fs.readdir(source, (err, files) => {
   if(err) {
      console.log(err);
   } else {
      files.forEach((filename, fileIndex) => {
         gm(source + filename).size((err, values) => {
            // Do something with the sizes
         });
      }
   }
});
```

Como podemos ver, los callback se van acumulando uno dentro de otro, volviendo el código difícil de comprender y mantener.

Promises

Las promesas intentan acabar con el problema del *callback hell*. Una promesa (Promise) es un tipo de variable, digamos, que nos da JavaScript. Las promesas, internamente, ejecutarán la función *reject* si se produce algún error, o *resolve* si todo va bien. Cuando nosotros creamos nuevas promesas, somos los responsables de llamar a estas funciones, como veremos a continuación. Esto desde fuera se puede utilizar para decidir qué hacer con el resultado sin tener que pasar un callback por parámetro.

Por ejemplo, podemos coger la función anterior y crear una promesa alrededor:

```
function readDir(folder) {
   return new Promise((resolve, reject) =>
      fs.readdir(folder, (err, filenames) => err ? reject(err) :
resolve(filenames))
   )
}
```

De esta forma, si queremos hacer algo con los resultados de *fs.readdir*, sólo tenemos que hacer lo siguiente:

```
readDir("/folderpath")
.then((files) => // Do something with the files)
.catch((error) => console.log(error))
```

Cuando llamamos a una promesa, el bloque *then* será llamado cuando dentro de la promesa se llame a la función *resolve*, con el mismo parámetro con el que se haya llamado a esta, y el bloque *catch* se ejecutará cuando dentro de la promesa se llame a *reject*, también con el parámetro que se introduzca en esta.
De esta forma, si se ejecutan varias promesas seguidas, quedaría algo tal que así:

```
readDir("/folderpath")
  .then((files) => getFilesSize(files))
  .then((filesSize) => // Return another promise)
  .then((result) => // Return another promise)
```

Y podríamos seguir hasta el infinito, sin necesidad de ninguna anidación. Sin embargo, es importante saber que hay que devolver la promesa dentro de un bloque *then* para poderlos seguir encadenando de esta manera. En este caso concreto, no hay que declarar el *return* porque estamos utilizando arrow functions con una sola declaración y sin llaves, lo que hace que se retorne el valor automáticamente.

Aún así, esta implementación no parece del todo limpia o comprensible. Es por esto que se introdujo en JavaScript el “async/await”.

Async / await

El await es básicamente un azúcar sintáctico para promesas. Lo que se consigue con esto es hacer que el código asíncrono se asemeje a un código procedural o síncrono. De esta forma, al utilizar la palabra *await*, la ejecución espera hasta que se termine de ejecutar la operación asíncrona antes de pasar a la siguiente línea. Además, en vez de enviar el resultado al bloque *then*, cuando dentro de la promesa se ejecuta la función *resolve*

esto hará que retorne el resultado. Si lo utilizamos con el ejemplo anterior, quedaría de la siguiente manera:

```
async function foo() {
   const files = await readDir("/folderpath");
   // Do something with files
}
```

Y si queremos ejecutar varias promesas seguidas, sólo tenemos que poner la palabra *await* delante de cada una de ellas.

En este ejemplo, vemos que en la función hemos utilizado la palabra *async*. Cada vez que dentro de una función vayamos a utilizar un *await*, hemos de avisar a JavaScript de que esta función es asíncrona poniendo un *async* delante.

Al utilizar *async*/*await*, si tenemos la necesidad de capturar un posible error que se pueda producir dentro de la promesa, tendremos que usar un *try*/*catch*. Básicamente, pondremos el código que puede fallar dentro del bloque *try* y el código que maneje el posible error, dentro del bloque *catch*. Utilizando el anterior ejemplo, quedaría así:

```
async function foo() {
   try {
       const files = await readDir("/folderpath");
      // Do something with files
   } catch(err) {
       console.log(err)
   }
}
```

Promesas - then/catch vs. async/await

autentia

¿Qué son?

Then/catch y async/await son dos formas distintas gestionar promesas, siendo la segunda la más moderna. Es recomendable usar siempre async/await frente a la otra forma, ya que facilita la lectura y evita problemas como el *callback hell*.

THEN / CATCH

Then/catch fue la primera forma que había de gestionar las promesas. Gracias a la introducción de las promesas, se solucionó el problema del *callback hell* (anidación). De esta forma, es posible encadenar promesas:

```
readDir("/folderpath")
  .then((files) => getFilesSize(files))
  .then((filesSize) => // Return another promise)
  .then((result) => // Return another promise)
```

Aún así, **todavía es posible que aparezca un *callback hell*** (si, por ejemplo, necesitamos el resultado que se obtuvo hace más de una promesa), como aquí:

```
connectToDatabase().then(db => {
  return getUser(db).then(user => {
    return getUserSettings(db).then(settings => {
      return enableAccess(user, settings);
    });
  });
});
```

ASYNC / AWAIT

El async/await **es la siguiente iteración que se introdujo en las promesas**. Básicamente, es un azúcar sintáctico que consigue hacer que el **código asíncrono se asemeje a un código procedural o síncrono**. Gracias a esto, también se soluciona del todo el problema del *callback hell*. Si volvemos a escribir el ejemplo previo con async/await, quedaría de la siguiente manera:

```
const db = await connectToDatabase();
const user = await getUser(db);
const settings = await getUserSettings(db);
await enableAccess(user, settings);
```

Hay que tener en cuenta que al utilizar algún *await*, habrá que utilizar la palabra *async* en la función que contenga ese código:

```
async function example() {
   const db = await connectToDatabase();
   return await getUser(db);
}
```

Array.prototype

La propiedad *Array.prototype* representa el prototipo del constructor *Array* y permite agregar nuevas propiedades y métodos a todos los objetos *Array*.

Por ejemplo, podríamos agregar un método que nos devolviera el primer valor del *Array* de la siguiente manera:

```
Array.prototype.first = function() {
   return this[0];
}
```

Y podríamos utilizarlo así:

```
const myArray = [4, 1, 7, 2];
console.log(myArray.first()); // Output = 4
```

Aún así, hacer esto no es recomendable, ya que si en algún momento se implementa un nuevo método que tenga el mismo nombre que uno que hemos hecho nosotros, puede romper nuestra aplicación.

Aparte de esto, contiene todos los métodos de los array, de los cuales los más importantes los vamos a ver a continuación. Estos métodos pueden ser de dos tipos:

- **Mutables**: modifican el array original, no retornan uno nuevo.
- **Inmutables**: no modifican el array original, retornan el resultado de la operación.

Map

El método *map* sirve para construir un nuevo array a partir de otro. Lo que hace es iterar cada uno de los elementos del array original y devolver el nuevo, es decir, es un método inmutable. Un ejemplo de esto podría ser el siguiente:

```
const people = [
  { name: "Pedro", yearsOld: 27 },
  { name: "Antonio", yearsOld: 42 },
  { name: "Luis", yearsOld: 24 },
];
const names = people.map((person) => person.name + '!');
// names = ["Pedro!", "Antonio!", "Luis!"]
```

Como podemos ver, lo que hemos hecho es iterar cada persona y devolver sólamente la propiedad *name* con un texto añadido. También podríamos haber devuelto el mismo objeto con una propiedad añadida o incluso, ejecutar un *map* sobre un array de valores primitivos como *strings* y devolver objetos.

Si lo que se quiere es simplemente iterar sobre un array, realizando operaciones mientras se recorre (no construir uno nuevo), existe también el método *forEach*, que se utiliza de la misma forma. Aunque en cualquier situación que se use un *forEach* se puede sustituir por un *map*, no se debería hacer, ya que se debería indicar de forma explícita qué es lo que está haciendo nuestro código (si está ejecutando operaciones mientras recorre un array o si está construyendo un nuevo array a partir de otro).

Filter

Este método nos permite filtrar de un array los elementos que cumplan una condición que nosotros mismos especificamos. Este es un método inmutable, igual que *map*, por tanto, devolverá un array filtrado en vez de modificar el original. Por ejemplo, podemos saber qué personas del array people tienen menos de 30 años:

```
const people = [
  { name: 'Pedro', yearsOld: 27 },
  { name: 'Antonio', yearsOld: 42 },
  { name: 'Luis', yearsOld: 24 },
];
const peopleYoungerThan30 = people.filter(person => person.age < 30);
/* peopleYoungerThan30 =
[
 { name: 'Pedro', yearsOld: 27 },
 { name: 'Luis', yearsOld: 24 },
]; */
```

Reduce

El método *reduce()* es inmutable y ejecuta una función reductora sobre cada elemento de un array, devolviendo como resultado un único valor (no un array). Vamos a verlo en un ejemplo:

```
const myArray = [4, 1, 7, 2];
const sum = myArray.reduce((accumulator, currentValue) => accumulator + currentValue); // sum = 14 (suma de los elementos del array)
```

En el método *reduce* tenemos 2 variables principales y aunque se les puede dar cualquier nombre a la hora de programar, se les ha dado un nombre completo para facilitar la lectura:

- **accumulator**: el resultado de cada operación se asigna al acumulador, cuyo valor se recuerda en cada iteración del array, siendo el valor final que se devuelve cuando se han completado todas las iteraciones.
- **currentValue**: valor del array sobre el que se está iterando en ese momento específico.

Sort

Este método se utiliza para ordenar un array y es mutable, es decir, modifica el array original. Si lo aplicamos a un array de números o a uno de strings, los ordenará de menor a mayor y en orden alfabético, respectivamente.

```
const myArray = [4, 1, 7, 2];
myArray.sort(); // myArray = [1, 2, 4, 7]
```

También podemos especificar el método de ordenación que queramos. Si por ejemplo, quisiéramos ordenar las personas del siguiente array según su edad de menor a mayor, deberíamos pasarle por parámetro al método sort, cómo queremos que realice la ordenación. En cada iteración comparará dos valores. Si retornamos un -1, pondrá en primer lugar al primero de los dos valores (person1 en este caso) y si retornamos un 1, pondrá por delante al segundo valor (person2). Si se retorna un 0, quiere decir que ninguno de los dos valores tiene preferencia frente al otro.

```
 const people = [
   { name: 'Pedro', yearsOld: 27 },
   { name: 'Antonio', yearsOld: 42 },
   { name: 'Luis', yearsOld: 24 },
 ];
 people.sort((person1, person2) => {
   if (person1.yearsOld < person2.yearsOld) {
     return -1;
   }
   if (person1.yearsOld > person2.yearsOld) {
     return 1;
   }
   return 0;
 });
}
```

Object.prototype

Object.prototype representa el prototipo del constructor *Object*. En esta sección, veremos los métodos que ofrece *Object* y cómo los podemos usar.

Object.keys

Este método nos devuelve las claves de un objeto (los nombres de sus propiedades) en un array. Por ejemplo:

```
const person =
{
   firstName: "Pedro",
   lastName: "Martínez",
   job: "developer",
}
console.log(Object.keys(person));
// Output = ["firstName", "lastName", "job"]
```

Object.values

Este método nos devuelve los valores de un objeto en un array, es decir, sólo el valor de cada propiedad, no el nombre de la propiedad en sí. Utilizando el ejemplo anterior:

```
const person =
{
   firstName: "Pedro",
   lastName: "Martínez",
   job: "developer",
}
console.log(Object.values(person));
//Output = ["Pedro", "Martínez", "developer"]
```

Object.entries

Este método nos devolverá un array y dentro de ese array, tendremos un array por cada propiedad que tenga el objeto. En cada uno de esos arrays, en primera posición tendremos el nombre de la propiedad (la clave) y en la segunda posición tendremos el valor de la propiedad.

```
const person =
{
   firstName: "Pedro",
   lastName: "Martínez",
   job: "developer",
}
console.log(Object.entries(person));
/* Output = [
   ["firstName", "Pedro"],
   ["lastName", "Martínez"],
   ["job", "developer"]
] */
```

Object.fromEntries

El método *Object.fromEntries()* nos permite construir un objeto a partir de arrays que contengan un string (que será el nombre de cada propiedad) y otro valor (que será el valor de dicha propiedad. Este es el método inverso de *Object.fromEntries()*.

```
const personEntries = [
   ["firstName", "Pedro"],
   ["lastName", "Martínez"],
   ["job", "developer"],
]

console.log(Object.fromEntries(personEntries));
/* Output = {
   firstName: "Pedro",
   lastName: "Martínez",
   job: "developer"
} */
```

Object.assign

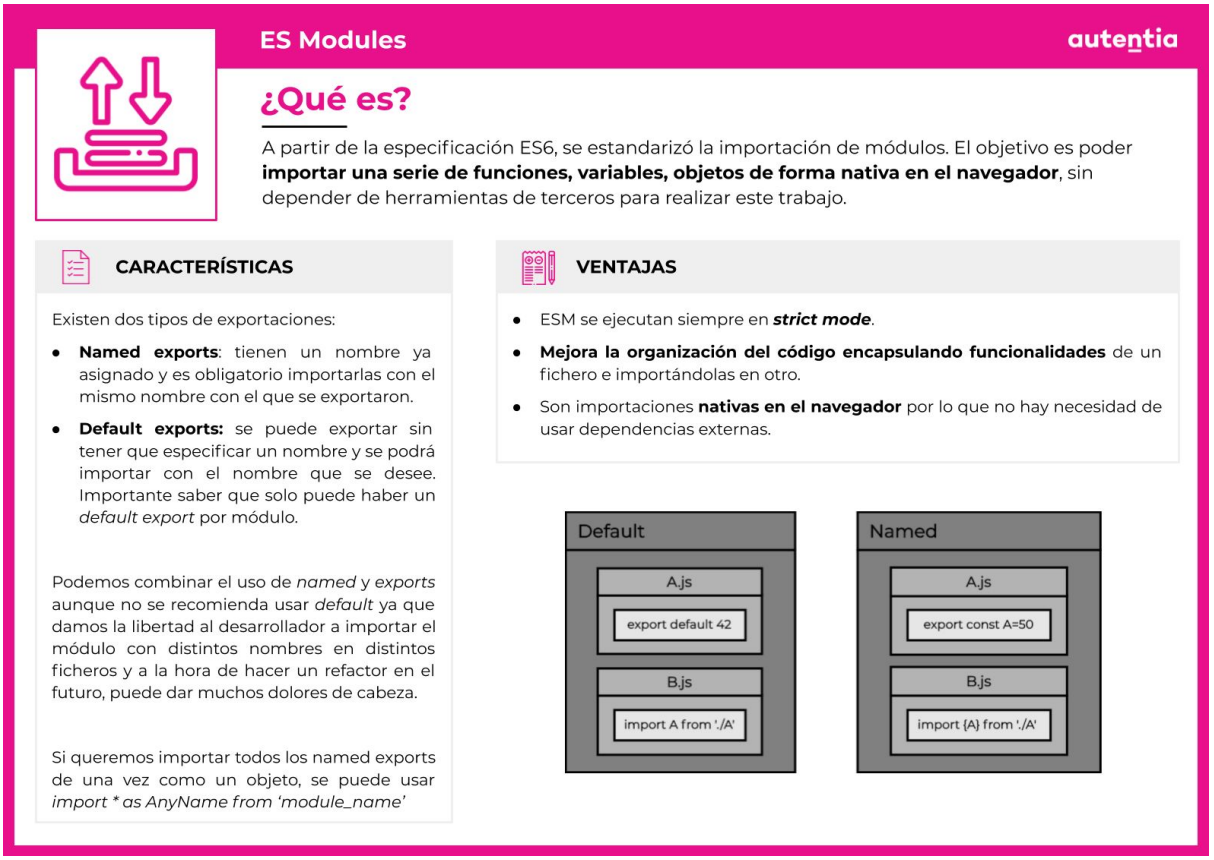
Este método permite asignar las propiedades y valores de un objeto a otro. Si una de las propiedades del objeto ya existe, se sobreescribirá el valor de esa propiedad. Cuando utilicemos este método, como primer parámetro pasaremos el objeto al que queremos asignar las propiedades y como segundo parámetro, el objeto del que queremos coger las propiedades a copiar.

```
const person =
{
   firstName: "Pedro",
   lastName: "Martínez",
}
const personalData =
{
   job: "developer",
   bornIn: "10-3-93",
   country: "Spain",
}
Object.assign(person, personalData);
console.log(person);
/* Output = {
   firstName: "Pedro"
   lastName: "Martínez",
   job: "developer",
   bornIn: "10-3-93",
   country: "Spain",
} */
```

ES Modules

Un módulo es un fichero que exporta uno o más valores (ya sean objetos, funciones o variables). ES Modules (ESM) es el estándar de ECMAScript para trabajar con módulos. Aunque Node.js ha estado usando el estándar de CommonJS desde hace algunos años, los navegadores no tenían un sistema de módulos, cuya estandarización se completó con ES6.

La ventaja de los módulos es que nos permiten encapsular funcionalidad y exponerla a otros ficheros JavaScript como si fueran librerías.



Export default

Si exportamos algo desde un módulo utilizando la palabra *default*, quiere decir que cuando lo importemos desde fuera le podremos dar cualquier nombre. Por ejemplo, si desde un fichero exportamos una variable de la siguiente forma:

- *some-file.js*

```
const myVar = 5;
export default myVar;
```

Luego lo podemos importar dándole el nombre que queramos:

```
import someRandomName from "route-to-dir/some-file"

console.log(someRandomName); // Output: 5
```

Sólo puede haber un *export default* por fichero. Realmente, esta forma de exportar no es muy recomendable ya que dejamos en manos del desarrollador que importe nuestra funcionalidad, el ponerle un nombre según su criterio y esto puede dar lugar a confusiones. Por tanto, es mucho más recomendable usar exports nombrados, como vamos a ver a continuación.

Export nombrado

Si realizamos un export nombrado, obligaremos a importar la funcionalidad utilizando el nombre que nosotros le hayamos dado y por tanto, evitaremos confusiones. Este sería un ejemplo de cómo exportar una función:

```
export function greet() { console.log("Hello World!"); }
```

Al usar exports nombrados, podremos exportar varias funcionalidades desde un mismo archivo. Entonces, ¿cómo importamos la correcta desde otro fichero y la utilizamos? Muy simple:

```
import { greet } from "route-to-file"
```

De esta forma, podemos coger sólo las funcionalidades que nosotros queramos. Por ejemplo, si tenemos el siguiente fichero:

- *some-file.js*

```
export const myVar = 5;
export function greet() { console.log("Hello World!"); }
```

Hay una forma equivalente a la anterior (inline exports) que es la siguiente:

```
const myVar = 5;
function greet() { console.log("Hello World!"); }

export {myVar, greet}
```

Desde otro fichero, podremos importar estas funcionalidades de la siguiente manera:

```
import { myVar, greet } from "route-to-dir/some-file"

console.log(myVar); // Output: 5
greet(); // Output: Hello World
```

Y si sólo queremos utilizar alguna de las funcionalidades, podemos sólo importar esa:

```
import { myVar } from "route-to-dir/some-file"
console.log(myVar); // Output: 5
```

Los *imports* se colocan siempre en la parte superior del archivo y los *exports* no tienen preferencia de colocación. Tiene más preferencia dónde queramos colocar dentro del fichero la función, clase o similar que queramos exportar. Es decir, los exports estarán repartidos por el fichero.

Parte 3

JavaScript (Entorno)

Node

Node.js es un **entorno de ejecución** de JavaScript que permite ejecutar código JavaScript **fuera de un navegador**. Hasta hace no mucho, los navegadores eran el único sitio donde se podía ejecutar este lenguaje. Node está basado, en parte, en el motor de JavaScript V8 de Google Chrome.

De esta forma, **Node permite la creación de servidores web** y similares, usando JavaScript y una colección de módulos que se encargan de distintas funcionalidades core (lectura/escritura de ficheros, herramientas de red como DNS, HTTP, TCP, TLS, UDP, etc., buffers, funciones criptográficas, data stream y otros). Además de JavaScript, se puede utilizar también **cualquier otro lenguaje que compile a JS** como por ejemplo TypeScript, Dart, CoffeeScript y otros.

"Node trae la programación guiada por eventos asíncronos a los servidores web, permitiendo el desarrollo de servidores web rápidos programados en JavaScript."

Una propiedad característica de Node.js es que sus funciones son

no-bloqueantes: los comandos son ejecutados de forma concurrente o incluso en paralelo y usan callbacks para señalizar éxito o error.

Actualmente, Node.js es compatible con los sistemas operativos Linux, macOS, Microsoft Windows 8.1 y Server 2012 (y posteriores).

Gestores de paquetes

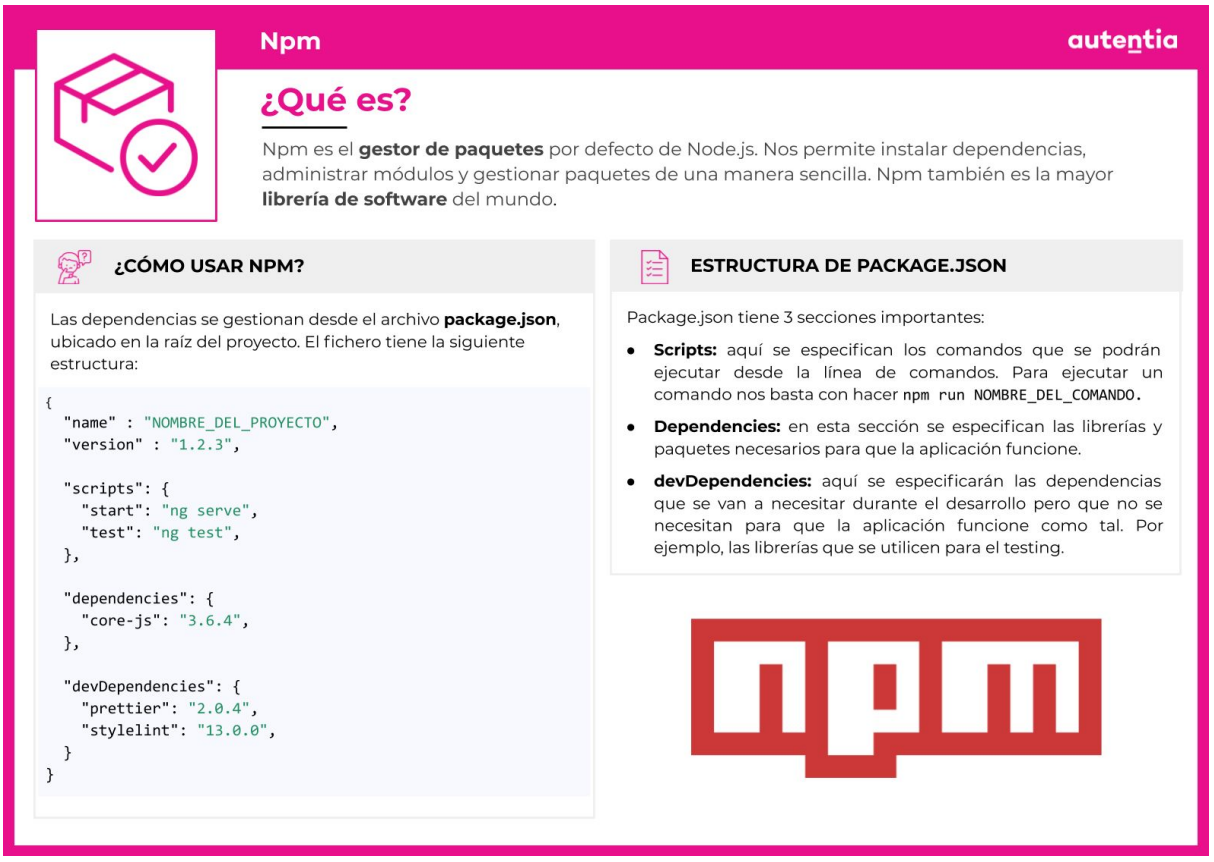
Cuando trabajamos en un proyecto siempre haremos uso de alguna librería externa y por tanto, el proyecto tendrá dependencias. Y con las dependencias viene la necesidad de tener éstas y las dependencias transitivas, es decir, las dependencias de las dependencias, organizadas y versionadas. Por suerte, hoy en día hay herramientas que se encargan de esto por nosotros en cualquier tecnología/lenguaje.

En el mundo JavaScript hay dos gestores de paquetes principales: npm y yarn.

npm

Este es el **gestor de paquetes por defecto de Node.js** y es mucho más utilizado que yarn. Consiste en un cliente de línea de comandos (también llamado npm) y una base de datos en línea de paquetes públicos y otros privados de pago, llamada “npm registry”.

npm se lanzó en 2010 como resultado de ver lo mal que se gestionaba en aquél momento la paquetería de los módulos, y está escrito enteramente en JavaScript.



package.json

Las dependencias se gestionan y quedan registradas en un archivo llamado *package.json* que estará presente en cualquier proyecto en el que se utilice npm. Este sería un ejemplo:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "browserslist": [
    "last 8 Chrome versions"
  ],
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "parcel -p 1234 watch src/index.html",
    "test": "jest ./src",
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@types/node": "13.13.1",
    "prettier": "2.0.4",
    "reflect-metadata": "0.1.13",
    "tslint": "6.1.1"
  },
  "devDependencies": {
    "@types/jest": "25.2.1",
    "jest": "25.4.0",
    "parcel-bundler": "1.12.4",
    "ts-jest": "25.4.0",
    "ts-mockito": "2.5.0",
    "typescript": "3.8.3"
  }
}
```

Podemos ver cómo este archivo tiene distintas secciones:

- **scripts**: aquí se especifican los comandos que se podrán ejecutar desde línea de comandos. Por ejemplo, si necesitamos ejecutar siempre los mismos pasos para hacer el *build* de la aplicación, podemos crear un nuevo script aparte de los que hay en el ejemplo

(*start* y *test*), y suponiendo que le llamáramos *build* (podemos ponerle el nombre que queramos), para ejecutarlo, sólo tendríamos que escribir en línea de comandos lo siguiente: `npm run build`. De esta forma, **podemos ejecutar uno o varios comandos complejos a partir de uno muy simple**. Además, si utilizamos un entorno de integración continua con procesos automatizados, podríamos sacar partido de esto y guardar los scripts a ejecutar en este archivo. Luego, el entorno de CI sólo tendrá que llamar a estos scripts. También, algo muy importante es que estos comandos **nos permiten utilizar comandos de dependencias que no podríamos ejecutar directamente por consola**. Por ejemplo, si instalamos una dependencia e intentamos ejecutar un comando de esa herramienta por consola, nos saltará un error avisandonos de que no existe tal comando (por ejemplo, si instalamos Parcel en el proyecto e intentamos ejecutar `parcel archivo-de-entrada`). Esto es porque esta dependencia no está instalada de forma global en nuestro ordenador, sino que está instalada en el local del proyecto. Por tanto, para ejecutar un comando de esta dependencia, habrá que hacerlo desde los scripts para que lo ejecute npm por nosotros.

- **dependencies**: en esta sección se especifican las dependencias de la aplicación como tal. Aquí irían todas las librerías y paquetes necesarios para que la aplicación funcione.
- **devDependencies**: aquí se especificarán las dependencias que se van a necesitar durante el desarrollo pero que no se necesitan para que la aplicación funcione como tal. Por ejemplo, las librerías que se utilicen para el testing, los bundlers, etc.

El versionado sigue una notación que indica si las dependencias se actualizarán automáticamente:

- **^**1.4.2: Si hay un sombrero antes del número de versión, esto significa que **npm irá actualizando automáticamente** a las versiones

compatibles de las dependencias, es decir, hasta que haya un cambio mayor de versión. En este caso de ejemplo, actualizaría a versiones mayores que 1.4.2, pero menores que 2.0.0. Esta es la notación que se aplica por defecto cuando instalas un nuevo paquete, pero **no es recomendable ya que no tienes control sobre qué versión se está usando** y si alguna actualización de una dependencia introduce un bug que repercute en tu aplicación, será extremadamente difícil determinar que este es el origen, ya que no se sabe cuándo se van actualizando las dependencias.

- **~**1.4.2: Esta notación actualizará a todas las versiones de parches pero no actualizará a la siguiente versión menor. Es decir, en este caso actualizará a versiones mayores de 1.4.2 pero menores que 1.5.0.

"No es recomendable usar estas notaciones en el versionado, ya que se pueden introducir bugs inesperados debido a las actualizaciones automáticas."

Comandos

El cliente de línea de comandos interacciona de forma remota con el *npm registry* y los comandos más utilizados son:

- `npm init`: genera un fichero *package.json* con la estructura básica del JSON ya creada. Se puede especificar que genere un *package.json* dirigido a alguna tecnología en concreto (React, esm, etc.).
- `npm install` / `npm i`: cuando descargas a tu ordenador un proyecto por primera vez, necesitarás instalar en tu proyecto local todas las dependencias que se necesitan para ese proyecto. Este comando lo

que hará es leer todas las dependencias del fichero *package.json* e instalarlas.

- `npm prune`: elimina los paquetes no utilizados, es decir, aquellos que tengamos instalados pero no estén reflejados en el fichero *package.json*.
- `npm install nombre-paquete` / `npm i nombre-paquete`: instala un nuevo paquete e incluye automáticamente la dependencia en el fichero *package.json* y las dependencias del paquete instalado en el fichero *package-lock.json*. Es importante recordar que, por defecto, incluirá la versión con la notación con sombrero (^) que habrá que eliminar.
- `npm i --save-dev nombre-paquete` / `npm i -D nombre-paquete`: igual que el comando de instalación normal pero registra la dependencia junto con las demás dependencias de desarrollos (devDependencies).
- `npm uninstall nombre-paquete`: desinstala el paquete especificado de nuestro local y borra automáticamente la dependencia del fichero *package.json*.

Yarn

Yarn es un **gestor de paquetes** desarrollado por **Facebook**. Se puede utilizar en el mismo proyecto en el que se use npm, ya que es compatible, e interactúa con el fichero *package.json* de la misma forma. En el momento en el que surgió, ofrecía un par de funcionalidades que *npm* no tenía por entonces y que fueron bastante importantes, dando lugar a que la herramienta tuviese bastante éxito. Por ejemplo, cuando se instalaba o se desinstalaba un paquete con la versión de *npm* de aquel entonces, había que hacerlo añadiendo una opción al comando para que actualizase el fichero *package.json* automáticamente, mientras que Yarn lo hacía por defecto. Yarn también añadió un fichero *yarn.lock* que listaba toda las

subdependencias de las dependencias directas que el proyecto utilizase.

Con el correr del tiempo *npm* ha ido implementando también estas funcionalidades, añadiendo a los proyectos el archivo *package-lock.json* como el equivalente a *yarn.lock*. Debido a esto, yarn **ha ido perdiendo popularidad** a lo largo del tiempo, cobrando *npm* más importancia.

Comandos

Los comandos más utilizados con yarn son:

- `yarn init`: equivalente a `npm init`. Genera un fichero *package.json*.
- `yarn add nombre-paquete`: es el equivalente a `npm install nombre-paquete`. Instala un nuevo paquete e incluye automáticamente la dependencia en el fichero *package.json* y las dependencias de ese paquete en *yarn.lock*. Si no existe alguno de los dos ficheros, los crea automáticamente.
- `yarn add --dev nombre-paquete`: igual que el comando de instalación pero para instalar una dependencia de desarrollo.
- `yarn remove nombre-paquete`: equivalente a `npm uninstall nombre-paquete`. Desinstala el paquete especificado de nuestro local y borra automáticamente las dependencias del fichero *package.json* y *yarn.lock*.

npm vs. yarn

Aunque en el momento que *yarn* apareció trajo consigo funcionalidades muy apreciadas, *npm* supo evolucionar y adoptó esas mismas funcionalidades. Por esa razón, hoy en día no es muy común utilizar yarn, siendo la gestión de monorepos uno de sus principales usos en la actualidad.

Transpiladores

Los transpiladores son herramientas que leen código fuente escrito en un lenguaje de programación y producen el código equivalente en otro lenguaje. La diferencia entre un compilador y un transpilador es que el primero **compila lenguajes de alto nivel a lenguajes de más bajo nivel** (por ejemplo, de C a binario), mientras que el objetivo de un transpilador es **convertir de un lenguaje de alto nivel a otro de alto nivel**. Dicho esto, la línea entre los dos suele ser un poco difusa y es importante saber que los transpiladores son en sí un tipo de compilador.

Babel

A los lenguajes que transpilan a JavaScript normalmente se les llama lenguajes ***compile-to-JS***, teniendo ese lenguaje como *target*. Ejemplos de estos lenguajes son TypeScript o CoffeeScript. El problema surge, cuando queremos ejecutar alguno de esos lenguajes en un entorno JavaScript ya que estos entornos sólo entienden ese lenguaje. Aquí es donde los transpiladores entran en juego: leen TypeScript, CoffeeScript o algún otro lenguaje compatible y convierten ese código a código JavaScript.

Aparte de esto, también se suelen usar para utilizar funcionalidades de JavaScript que aún no hayan salido (compilando código ECMAScript 2015+ a una versión de JavaScript, compatible con versiones anteriores que puedan ejecutar los motores JavaScript más antiguos). En este caso, se podría decir que hacen las funciones de compilador. Un ejemplo muy simple: el transpilador convertiría *let* o las *arrow functions* en *var* y *functions* para que

el navegador pueda entenderlo.

Babel es el transpilador/compilador JavaScript **más utilizado** en la actualidad. Para instalarlo en el proyecto, hay que ejecutar los siguientes comandos:

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

La siguiente dependencia permite instalar Polyfill. El objetivo es usar todo el conjunto de funcionalidades de ES6 (aparte de lo que ya transpila Babel). Es como una ayuda extra a Babel que permite usar características como promesas, Map, Set Array.from y Object.assign, entre otros. Sin Babel/polyfill solo podríamos usar funcionalidades como arrow function, destructuring, argumentos por defecto, etc.

```
npm install @babel/polyfill
```

Hay muchas formas de configurar Babel dependiendo de la necesidad que tengamos (si utilizamos un monorepo o queremos compilar *node_modules*, si queremos aplicar una configuración a una parte determinada del proyecto, etc). Para una configuración a nivel proyecto, se utiliza el fichero *babel.config.js* o, en versiones más recientes, *babel.config.json*.

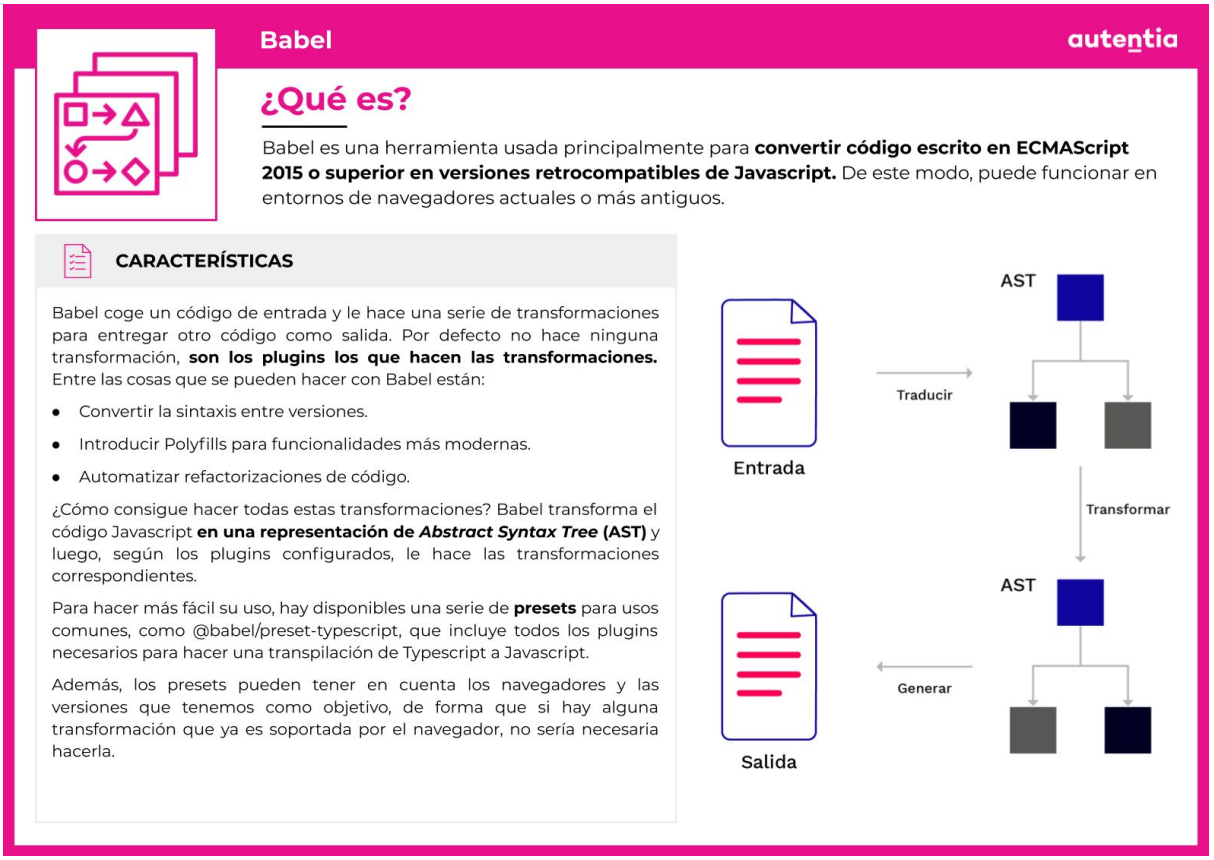
Este fichero nos permitirá configurar, entre otras opciones, las versiones de navegadores que soportará nuestra aplicación, los complementos, presets y más.

Un ejemplo de este archivo:

```
{
  "presets": [
    [
      "@babel/env",
      {
```

```
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1",
        },
        "useBuiltIns": "usage",
        "corejs": "3.6.4",
      }
    ]
  ]
}
```

Para ver todas las opciones de configuración disponibles, podemos visitar la documentación oficial.



Bundlers

La función de un bundler es combinar y optimizar varios módulos en uno o más paquetes para el navegador y dejarlos listos para producción. Los bundlers más utilizados son los siguientes:

Webpack

El proyecto de webpack como tal, comenzó en 2011, pero no fue hasta 2014 cuando realmente arrancó. Actualmente, webpack es el bundler más utilizado y establecido, siendo la elección de empresas como Auth0, Netflix, Instagram, Airbnb, KhanAcademy, Trivago y otras.

Para instalar webpack en un proyecto, sólo habrá que ejecutar el comando `npm install webpack -D`. Después, hay que crear el archivo de configuración *webpack.config.js*. Este archivo de configuración es un lugar para colocar toda la configuración, los *loaders* (explicados más adelante) y otra información específica relacionada con su compilación. Un ejemplo de este fichero sería:

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

module.exports = {
  entry: './index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    rules: [
      { test: /\.css$/, use: 'css-loader' },
      { test: /\.ts$/, use: 'ts-loader' }
    ]
  }
};
```

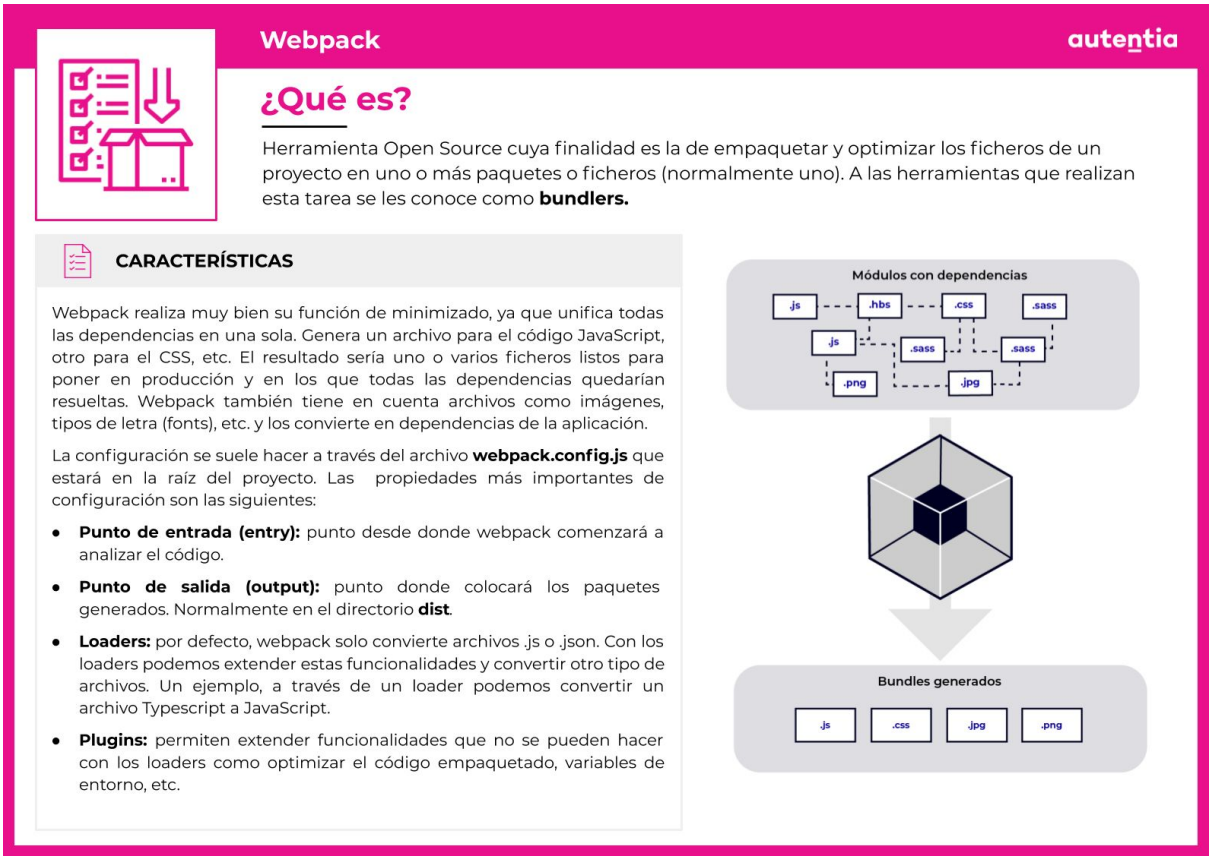
En la propiedad *entry* habrá que poner la ruta al archivo de entrada a nuestra aplicación. El punto por defecto de entrada es "*./src/index.js*" pero se puede especificar otro, incluso se pueden especificar múltiples puntos de entrada. En el *output* especificamos dónde queremos que webpack guarde los ficheros resultado de hacer el empaquetado y cómo queremos llamarlos. Por defecto, suelen generarse en el directorio *dist*. Webpack sólo convierte archivos con extensión .js o .json, pero a través de los *loaders* podemos convertir otro tipo de ficheros o incluir el loader de Babel para que nos haga la transpilación de archivos. Los loaders tienen dos propiedades de configuración:

- **test**: indica el tipo de archivo que vamos a convertir (ts, txt, css, etc.).
- **use**: el tipo de loader que vamos a usar para la conversión.

Por último, nombrar la propiedad *plugin* que permite extender

funcionalidades que no se pueden hacer con los loaders, como optimizar el código empaquetado, inyectar variables de entorno, etc.

Es recomendable añadir el script `"build": "webpack"` al archivo *package.json*, para poder hacer la build simplemente ejecutando `npm run build`.



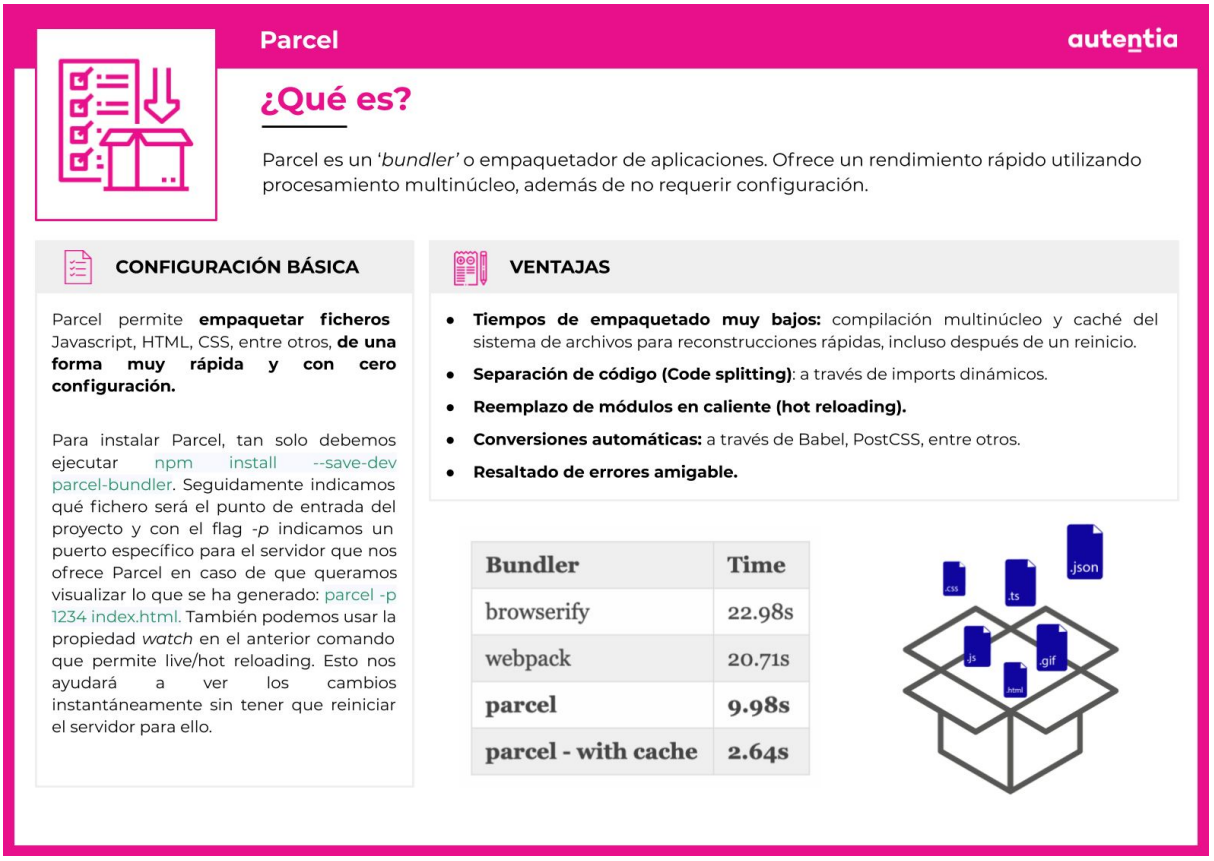
Parcel

Parcel es un bundler que si bien no lleva mucho tiempo disponible, ha llamado mucho la atención, ya que además de ser muy rápido, **no requiere de ninguna configuración**. Es tan sencillo de usar que lo único que hace falta es instalarlo (`npm install parcel-bundler -D`) y añadir el siguiente script al archivo *package.json* por comodidad:

```
"start": "parcel -p 1234 watch src/index.html"
```

De esta forma, haremos uso del servidor web para desarrollo que nos ofrece el propio Parcel. Lo que estamos diciendo con este comando es que cuando ejecutemos en consola `npm run start`, se servirá la aplicación en el puerto 1234. Además, como hemos añadido la palabra "watch", estará atento a los cambios que hagamos mientras desarrollamos y la aplicación se actualizará automáticamente sin tener que volver a ejecutar el comando. Tanto la indicación del puerto, como la adición de "watch" son opcionales. Además, "src/index.html" ha de ser sustituido con la ruta al archivo de entrada de la aplicación.

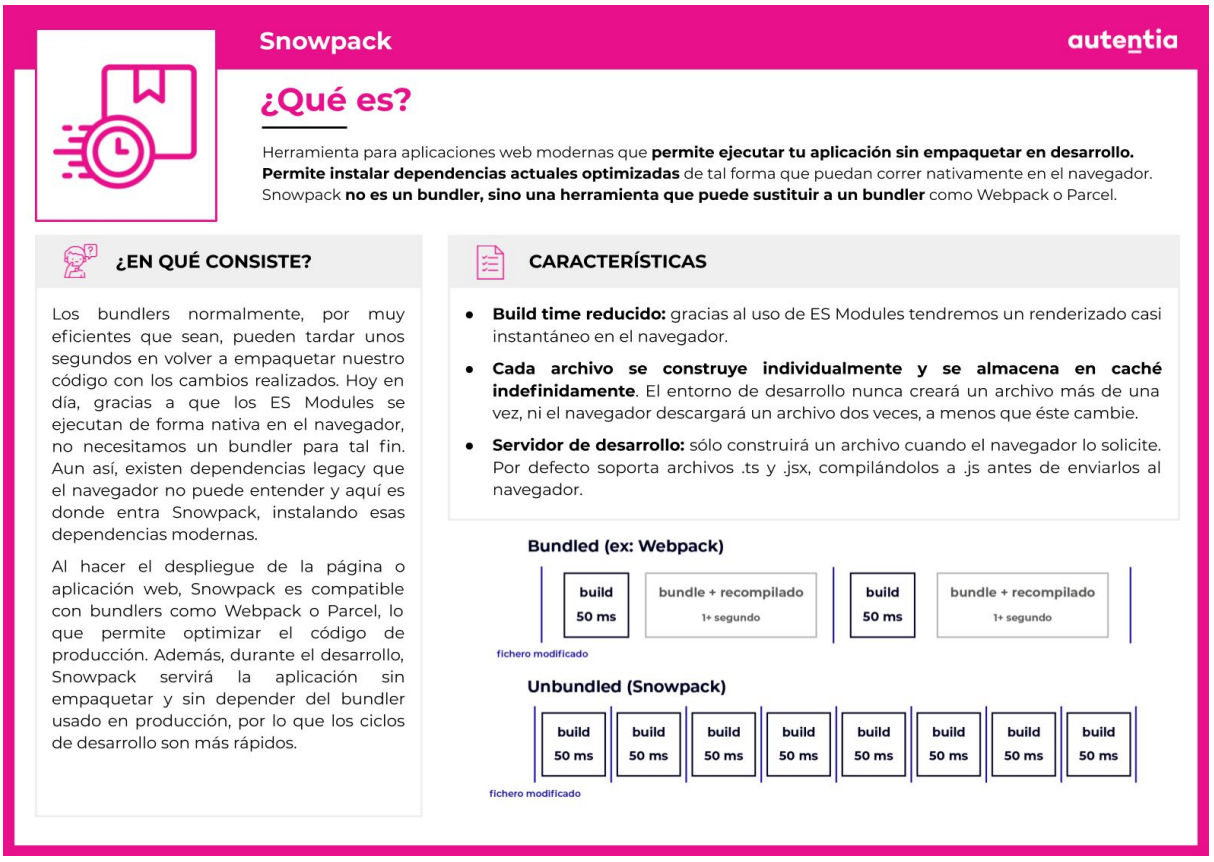
El anterior comando sería el utilizado a la hora de desarrollar. Si lo que queremos es hacer la build para producción, añadimos el siguiente script al fichero *package.json*: `"build": "parcel build <fichero de entrada>"`.



Snowpack

Antes que nada, es importante decir que Snowpack **no es un bundler**. ¿Por qué entonces, está en esta sección? Pues porque, de forma resumida, Snowpack elimina la necesidad de usar un bundler.

Cuando surgió npm, el sistema de módulos que se utilizaba (Common.js) no se podía usar en la web sin empaquetar y fue cuando bundlers como Browserify, Webpack, etc., aparecieron. Sin embargo, el nuevo sistema de módulos ES Modules sí se ejecuta de forma nativa en la web, por tanto, no hay necesidad de usar ningún bundler. Aún así, el problema persiste por razones como por ejemplo, que las dependencias que funcionan con ESM tienen a su vez algunas dependencias legacy o por la forma especial en que los paquetes de npm importan dependencias por nombre; por tanto, al final no consiguen funcionar de forma nativa. Aquí es cuando surge Snowpack.



Snowpack instala dependencias modernas de npm de tal forma que les permite correr nativamente en el navegador. Por tanto, **Snowpack no es un bundler, sino una herramienta que puede sustituir a un bundler** como Webpack o Parcel. De esta forma, si utilizas un bundler es porque realmente quieres usarlo (no porque lo necesites) por ejemplo, para optimizar la build.

Para instalar Snowpack sólo hace falta ejecutar `npm install snowpack -D`. Para ver qué hay que hacer dependiendo de si estamos creando una aplicación nueva o si queremos migrar desde una existente, es mejor referirnos a la documentación. Una vez instalado, para servir la aplicación sólo habría que ejecutar `snowpack dev` (o tener este comando entre los scripts de *package.json* y ejecutarlo con `npm run nombre-comando`).

Herramientas CSS

Sass

Sass (Syntactically Awesome Stylesheet) es un preprocesador que aporta funcionalidades extra al CSS, que traduce a CSS puro para que el navegador lo pueda entender. Algunas de las funcionalidades que ofrece son:

- Variables (Sass ofrecía variables antes de que se estandarizaran en CSS).
- Anidación de elementos.
- Notaciones más cortas para selectores y propiedades.
- Herencia (extiende los estilos de otro selector).
- Mixins (una especie de funciones que generan CSS en función de una variable que le introduzcas).

Sass tiene dos variantes de sintaxis, Sass y Scss. La única diferencia entre las dos es que la segunda es más parecida a CSS, mientra que Sass se parece más a un archivo yaml ya que no usa ni llaves ni puntos y comas.

En la guía oficial está todo lo necesario para entender cómo utilizar las diferentes funcionalidades que aporta Sass.

PostCSS

PostCSS es una herramienta para transformar los estilos de CSS con plugins de JavaScript. Estos plugins pueden lintear el CSS, soportar variables y mixins, transpilar sintaxis CSS futura, etc. La principal diferencia entre PostCSS y preprocesadores como Sass, Less y Stylus es que PostCSS es modular e incluso más rápido. Además, con PostCSS puedes escoger qué funcionalidades utilizar, mientras que los demás preprocesadores incluyen un montón de funcionalidades que puedes necesitar o no.

PostCSS

autentia

¿Qué es?

PostCSS es una herramienta para transformar los estilos de CSS con plugins de JavaScript. Estos plugins pueden lintear el CSS, soportar variables y mixins, transpilar sintaxis CSS futura, etc.

PLUGINS

PostCSS tiene más de 200 plugins, cada uno para una función específica. Estos son algunos de los ejemplos que muestran las posibilidades que ofrece PostCSS:

- **Autoprefixer**: este plugin añadirá prefijos de proveedor (vendor-prefixed properties) a las propiedades de CSS para que sean compatibles con distintos navegadores. De esta forma, reducimos la confusión en nuestro código.
- **PostCSS Preset Env**: con este plugin se puede utilizar sintaxis de CSS futuro (que todavía no haya salido) y el mismo plugin lo traducirá a CSS que los navegadores puedan entender. Tiene el mismo objetivo que Babel con JavaScript pero aplicado a CSS.
- **Stylelint**: este plugin nos indica errores que haya en nuestro código y nos fuerza a seguir un estándar de buenas prácticas a la hora de escribir código CSS. Además, soporta la última versión de sintaxis de CSS. A veces, no necesitamos que Stylelint nos indique todos los errores en el código, es por esto que nos permite habilitar o deshabilitar las reglas que más nos convengan para que se adapte a nuestro desarrollo.

¿EN QUÉ SE DIFERENCIA CON SASS?

La principal diferencia entre PostCSS y preprocesadores como Sass, Less y Stylus es que PostCSS es modular y llega a ser incluso más rápido que el resto.

Con PostCSS puedes escoger qué funcionalidades utilizar, mientras que los demás preprocesadores incluyen un montón de funcionalidades que puedes necesitar o no.

AUTOPREFIXER

```
:fullscreen {
}
```

→

```
:-webkit-full-screen {
}
:-ms-fullscreen {
}
:fullscreen {
}
```

Lint de código

autentia

¿Qué es?

Se encarga de analizar el código en busca de errores programáticos y estilísticos, como por ejemplo, errores de sintaxis o nombres de variables mal escritos.

LINT VS. FORMAT

Format

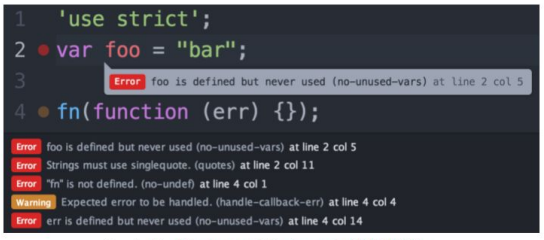
- Cubre la necesidad de que todo el mundo en un mismo proyecto utilice el mismo formato de código.
- Revisa y modifica los espacios, tabulaciones, etc.

Lint

- Cubre la necesidad de que todo el mundo en un mismo proyecto utilice las mejores prácticas para un código de buena calidad (por ejemplo, usar *let* y *const* en JavaScript en vez de *var*).
- Ayuda a utilizar las mejores sintaxis o nuevas funcionalidades de un lenguaje y atrapar posibles errores.

LINTERS MÁS UTILIZADOS PARA JS

- ESLint
- JSLint
- JSHint



Fuente: https://developer.com/linting-is-parenting-878b2470836a

Para instalarlo sólo hay que ejecutar: `npm i postcss`. En su mismo github explican muy bien cómo implementar los archivos de configuración necesarios, dependiendo del bundler que se esté usando.

PostCSS tiene más de 200 plugins, cada uno para una función específica. Estos son algunos de los ejemplos que muestran las posibilidades que ofrece PostCSS:

Autoprefixer

Este plugin añadirá prefijos de proveedor (vendor-prefixed properties) a las propiedades de CSS, para que sean compatibles con distintos navegadores. De esta forma, reducimos la confusión en nuestro código. Por ejemplo, si nuestro código es el siguiente:

```
:fullscreen {
}
```

El plugin generaría automáticamente el siguiente output:

```
:-webkit-:full-screen {}
:-ms-:full-screen{}
:full-screen {}
```

El prefijo *-webkit* es usado por el motor de renderizado de Chrome y Safari y *-ms* para Internet Explorer. También existen otros prefijos como *-moz* para firefox y *-o* para Opera. En la siguiente página se puede testear y visualizar el resultado de distintas propiedades CSS.

Está considerada una buena práctica situar los prefijos de proveedor antes del que no tiene prefijo, de este modo se evita que se sobreescriban propiedades concretas.

PostCSS Preset Env

Con este plugin se puede utilizar sintaxis de CSS futuro (que todavía no haya salido), y el mismo plugin lo traducirá a CSS que los navegadores puedan entender. Tiene el mismo objetivo que Babel con JavaScript pero aplicado a CSS.

Stylelint

Este plugin nos indica errores que haya en nuestro código y nos fuerza a seguir un estándar de buenas prácticas a la hora de escribir código CSS. Además, soporta la última versión de sintaxis de CSS.

A veces, no necesitamos que Stylelint nos indique todos los errores en el código, es por esto que nos permite habilitar o deshabilitar las reglas que más nos convengan para que se adapte a nuestro desarrollo.

Browserslist

Nos permite configurar las versiones de navegadores que soportará nuestra aplicación. Esta configuración la tienen en cuenta las herramientas que tienen que trabajar con el navegador. Por ejemplo: PostCSS, Babel, Autoprefixer, etc.

Normalmente, se configura dentro del fichero package.json, por ejemplo:

```
"browserslist": [
   ">0.2%",
   "last 1 chrome version",
   "last 1 firefox version",
   "last 1 safari version"
]
```

En el ejemplo de arriba estamos soportando aquellos navegadores que tengan una cuota de mercado mayor a 0.2% y la última versión de Chrome, Firefox y Safari.

Browserslist acepta varias combinaciones. Cuantos más navegadores y versiones soporte nuestra aplicación, más usuarios podrán usarla. Restringirnos a los navegadores más populares en sus últimas versiones puede ser cómodo para el desarrollador pero deja fuera a una parte grande de los usuarios. La decisión depende fundamentalmente de negocio.

Compatibilidad entre navegadores

autentia

¿En qué consiste?

Cada navegador implementa los estándares de manera distinta. Esto puede dar lugar a que los usuarios tengan una experiencia diferente en función del navegador que están usando.

CONCEPTO

La compatibilidad entre navegadores es un concepto a tener en cuenta a la hora de desarrollar aplicaciones web, debido a que para una misma web, **usar distintos navegadores puede resultar en una experiencia distinta**. Puede darse el caso extremo en el que exista incompatibilidad con un navegador, quedando limitada la audiencia de esa web.

Cuando el diseño se desajusta entre navegadores, puede ocurrir que el texto no quepa en la pantalla, que no sea visible la barra de scroll o que cierto código en JavaScript no se ejecute, etc. Como es inviable comprobar la compatibilidad entre todos los navegadores del mercado, merece la pena asegurarlo entre los que tienen más cuota de mercado, como Chrome, Firefox y Safari.

Hay distintas acciones que nos ayudan a asegurar esta compatibilidad, entre las que se encuentran:

- **Validar tanto el HTML como el CSS** de la web para que cumplan el estándar.
- **Resetear los estilos CSS**. Cada navegador tiene unos valores por defecto para ciertas propiedades, haciendo que algunos elementos se vean distintos.
- Usar **técnicas soportadas**. La web de Can I Use muestra la compatibilidad de funciones de la API de JavaScript para distintos navegadores.

DIFERENCIAS ENTRE NAVEGADORES

Hay dos piezas fundamentales en un navegador: por un lado el motor de renderizado (que analiza el código HTML y CSS) y por otro el motor de JavaScript.

Cada **navegador utiliza un motor distinto** que implementa los estándares con pequeñas diferencias. Además, esta implementación puede cambiar con las versiones y con el sistema operativo.

Es por esto que no todos los navegadores interpretan el HTML, CSS y JavaScript igual. Aunque a día de hoy las diferencias sean pequeñas, pueden hacer que un usuario no pueda ver correctamente la página.

Polyfills

Cuando un lenguaje que se usa en el navegador introduce nuevas características que nos facilitan el desarrollo, los navegadores tienen que actualizarse para tener soporte. Los más populares lo hacen antes, pero si queremos que los que aún no se hayan actualizado, funcionen correctamente, tenemos que hacer uso de uno o más polyfills, esto es, un código que simula esa implementación que falta en el navegador. Esto tiene un coste: el rendimiento y el tamaño del bundle final que puede crecer en exceso debido a la cantidad de polyfills que incluyamos. A día de hoy, la mayoría de las tecnologías que usamos para construir apps en el front incluyen automáticamente los polyfills necesarios.

En polyfill.io podemos crear un fichero con todos los polyfill que queremos incluir en nuestro proyecto.

Polyfill

autentia

¿Qué es?

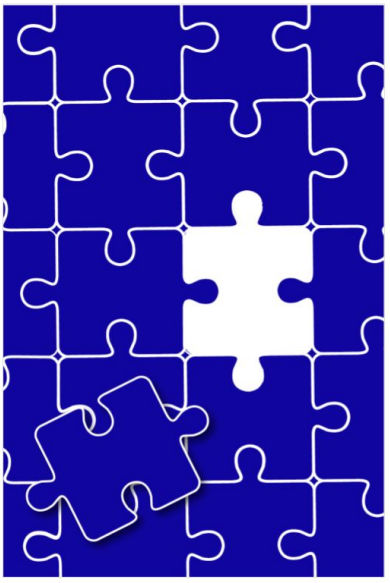
Un *polyfill* es un trozo de código que **proporciona una funcionalidad** en navegadores que no la soportan, dando a las aplicaciones web la posibilidad de tener cierta retrocompatibilidad. También son utilizados para implementar una funcionalidad propuesta o futura en navegadores actuales.

CARACTERÍSTICAS

Un polyfill **replica una funcionalidad en un navegador que no la soporta de manera nativa,** teniendo así soporte en navegadores antiguos. En general implementan múltiples formas de replicar una misma funcionalidad y así tener siempre una alternativa por si la primera no funciona.

Un ejemplo es la propiedad *sessionStorage* de la API Window que no es soportada en Internet Explorer 7. Para darle soporte existen varias técnicas: un almacenamiento local basado en cookies, usar la propiedad *localStorage*, un almacenamiento con Flash 8, etc. El polyfill se encarga, de forma **transparente para el desarrollador,** de comprobar si está soportada por el navegador y elegir la técnica óptima para dársela en caso de que no esté soportada.

La palabra Polyfill busca describir este concepto juntando *poly*, ya que siempre hay varias formas de replicar una funcionalidad, y *fill*, en cuanto a que llena los vacíos que existen en las tecnologías del navegador.



Herramientas de depuración

Antes de comenzar, todos los ejemplos siguientes se han realizado sobre el navegador Chrome. En otros navegadores como Firefox o Safari las herramientas son muy parecidas y se accede de la misma forma, así que no debería variar mucho la experiencia.

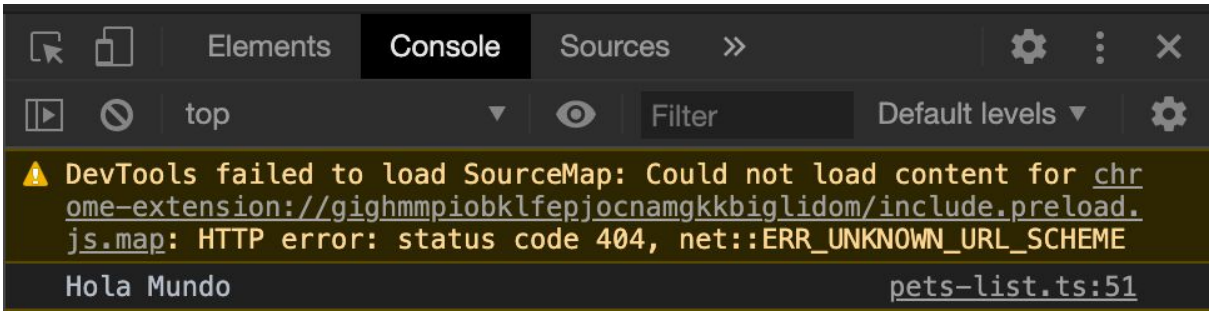
Consola

Cuando hablamos de depurar en front, aunque existe la posibilidad de hacerlo utilizando puntos de interrupción, esto requiere configuración dependiendo del IDE que utilicemos y suele ser más sencillo y rápido utilizar la consola para trazar información. En esta sección vamos a explicar por una parte cómo acceder a la consola y cómo utilizarla y por otro lado, mostraremos distintos ejemplos para entender cómo usar esta técnica de depuración dependiendo de la situación.

Cómo acceder a la consola

Con el navegador abierto y en la pestaña donde hayamos accedido a nuestra web o aplicación, hacemos click con el botón derecho y después, sobre “Inspeccionar”. Esto hará que se abra el inspector que es donde encontraremos todas las herramientas necesarias para depurar. Ahora, hacemos click en Console. Aparte de ser donde se mostrarán los errores y excepciones que se produzcan en la aplicación, es también donde veremos el output del comando *console.log*. Cuando en nuestro proyecto

escribamos `console.log('Hola Mundo');` veremos lo siguiente:



El warning que se puede ver es simplemente un aviso que ha comenzado a surgir con las últimas actualizaciones de Google Chrome pero no afecta de ningún modo al funcionamiento de la aplicación ni tiene efecto alguno.

A continuación, veremos distintas técnicas o dinámicas a seguir a la hora de depurar con *console.log*.

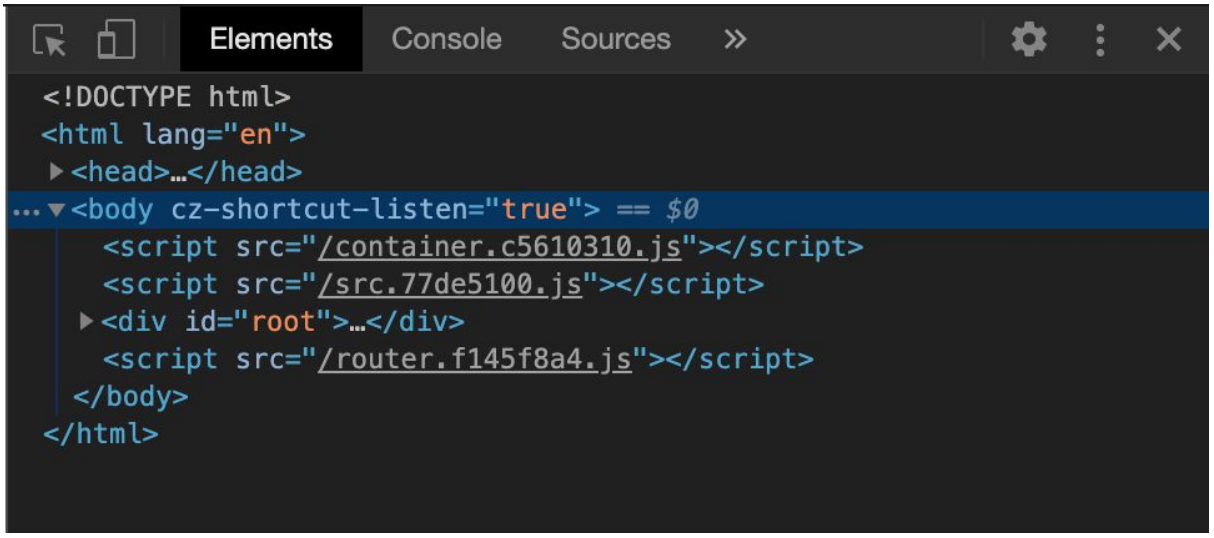
A tener en cuenta

Es preferible **depurar el código** usando el debugger del navegador antes que tener unos cuantos console.log repartidos por el proyecto para resolver un bug. Es **muy recomendable configurar el linter** para que nos marque como error todas las sentencias console.log del proyecto para animar al desarrollador a eliminarlos antes de subir los cambios al repositorio por despiste.

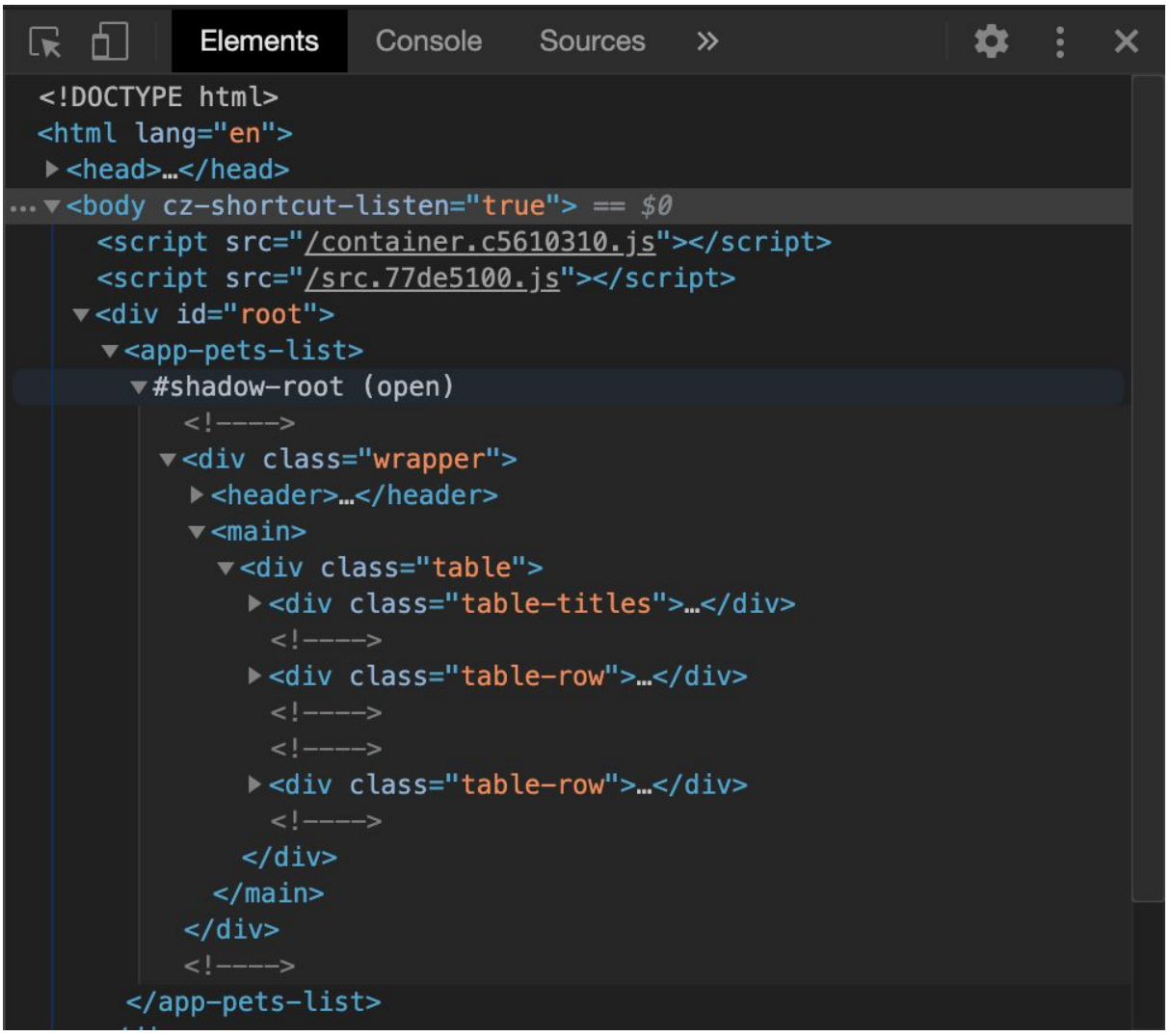
Inspector de elementos

Esta herramienta del inspector del navegador nos ayudará a analizar el html y el css en caso de que veamos alguna discrepancia en la interfaz. Para acceder a él, sólo tenemos que hacer click derecho en la página y seleccionar “Inspeccionar”. Cuando se abra el inspector, nos encontraremos por defecto en la sección “Elements” donde podremos ver el HTML de

nuestra aplicación web.



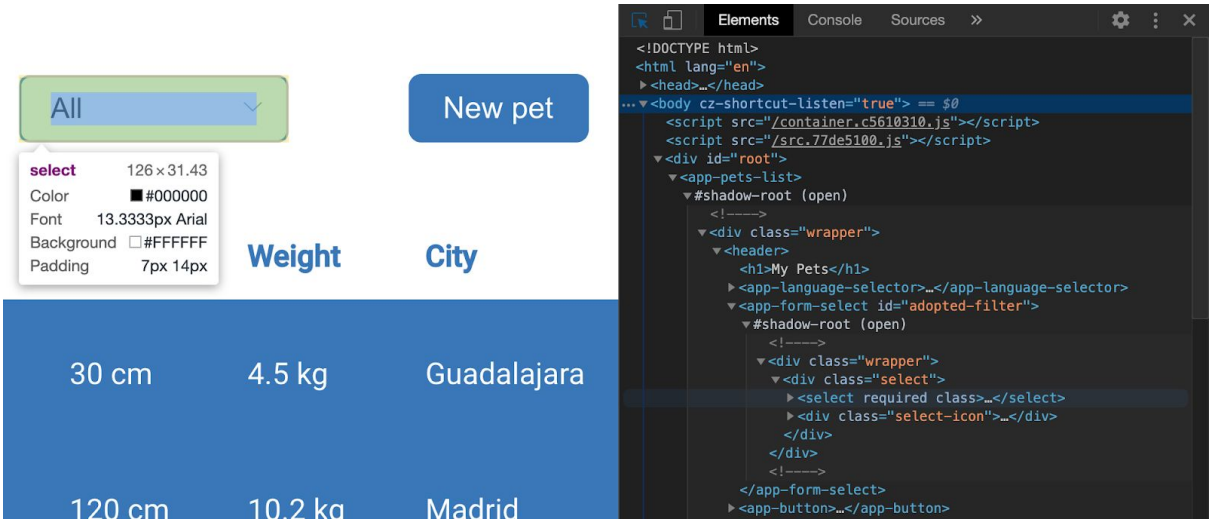
Una vez aquí, podemos ir desplegando elementos para ir profundizando más y más en la página:



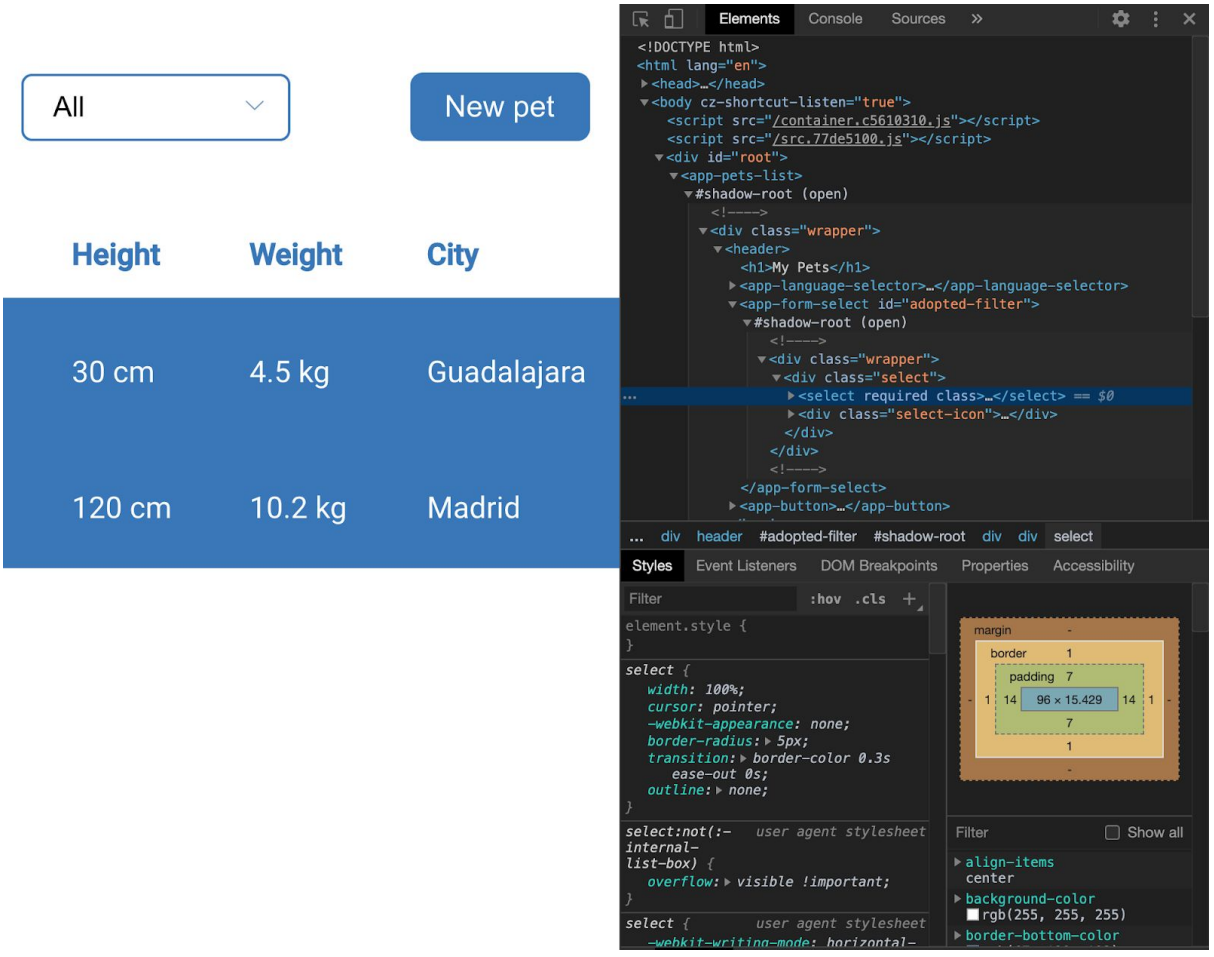
Esto, obviamente, no es el mejor método para llegar a un elemento en concreto cuando estamos depurando, ya que nos podemos pasar buscando un largo tiempo. Para eso tenemos el selector de elementos, arriba a la izquierda:

Cuando hagamos click en él, entraremos en el modo de selección de elemento. Cuando estemos en este modo y pasemos el ratón por nuestra aplicación web, podremos ver cómo se van resaltando los elementos, indicándonos el nombre además de su situación en el HTML con un leve

sombreado:



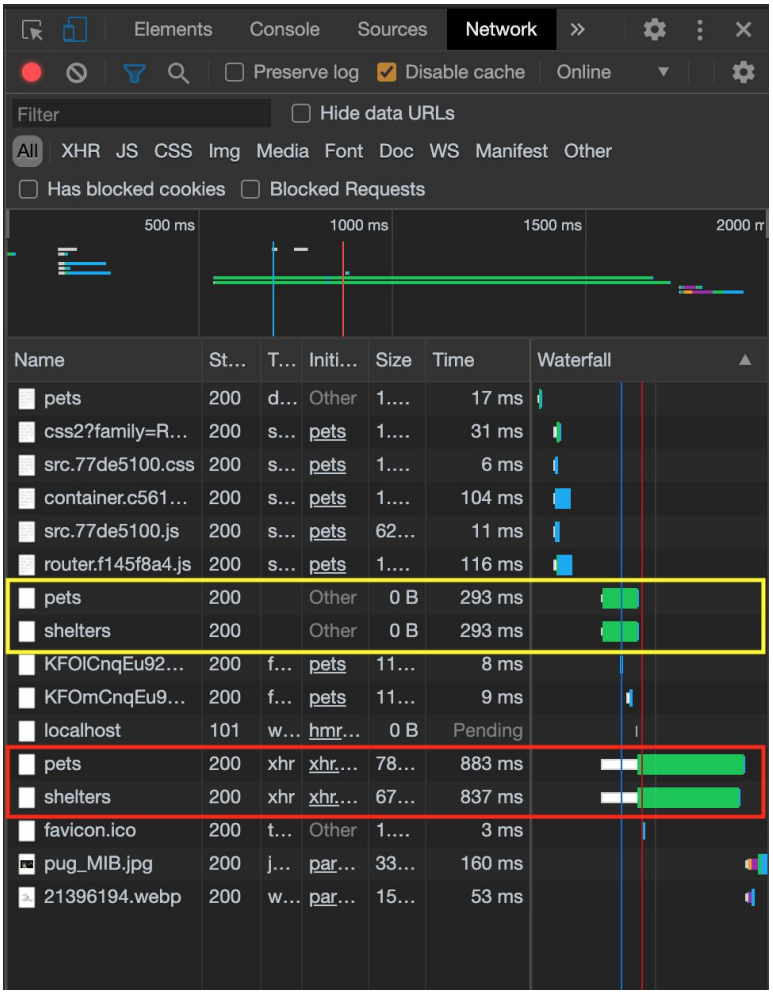
Cuando hagamos click sobre uno de los elementos, se seleccionará el elemento correspondiente en el HTML y podremos también ver su CSS:



Gracias a esta función es muy sencillo navegar por la interfaz e ir analizando cada elemento. De esta forma, si algún elemento de nuestra interfaz no se muestra dónde o cómo se debería mostrar, podemos analizarlo detenidamente y comprobar si hay algún problema.

Inspector de peticiones

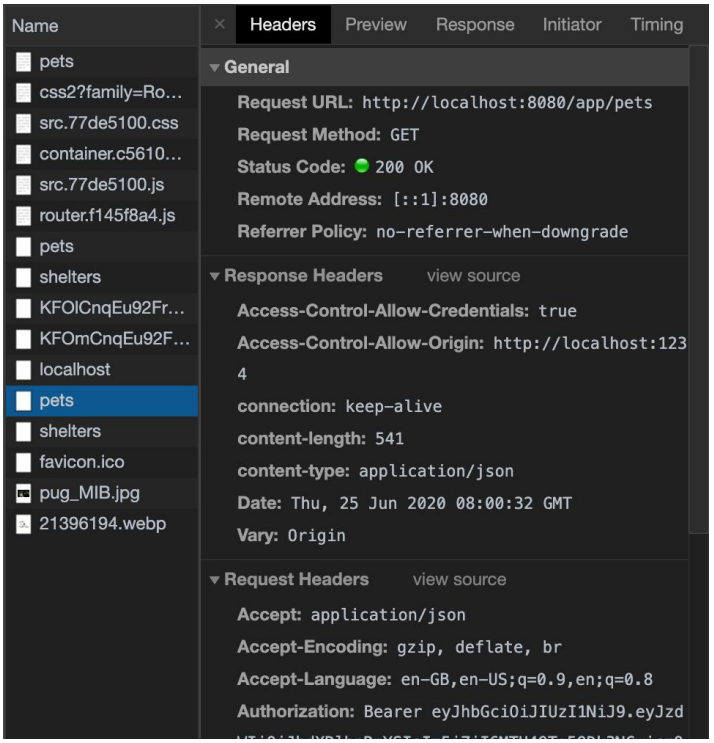
Gracias a esta herramienta podemos ver todos los detalles de las peticiones que hace nuestra aplicación además, de las respuestas a estas. Para acceder a la herramienta, accederemos al inspector como en los apartados previos y después seleccionaremos “Network”. Podremos ver una lista de las peticiones que realiza la app:



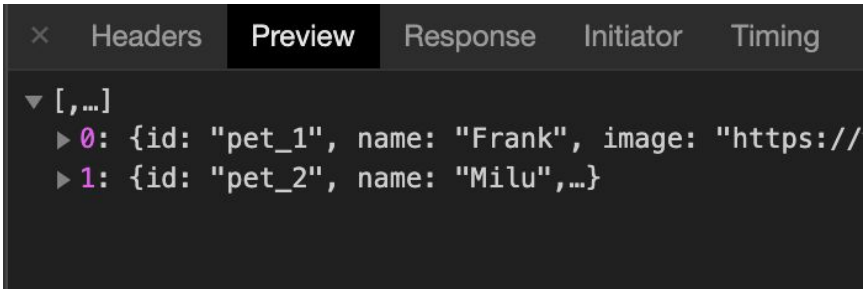
Normalmente, cuando hacemos una petición, el navegador antes hará lo que se llama una *preflight request*. Esto es básicamente una request automática que hace con el verbo OPTIONS que sirve para comprobar si el protocolo CORS del servidor es el esperado e informarse de los verbos y cabeceras que soporta el servidor. No nos tenemos que preocupar por estas peticiones ya que no influyen en el funcionamiento de la aplicación.

En este ejemplo, que es una aplicación para adoptar animales, podemos ver cómo se hacen dos peticiones (aparte de otras muchas que no nos interesan): una para recoger los datos de las mascotas y otra para recoger las protectoras de animales. Rodeadas de amarillo, podemos ver las *preflight requests* y en rojo las peticiones como tal. El primer "pets" de la lista, aunque parezca una petición igual que la otra, es mera coincidencia. Esta petición es del navegador al servidor que está sirviendo la aplicación porque la url de esa pantalla es "/pets".

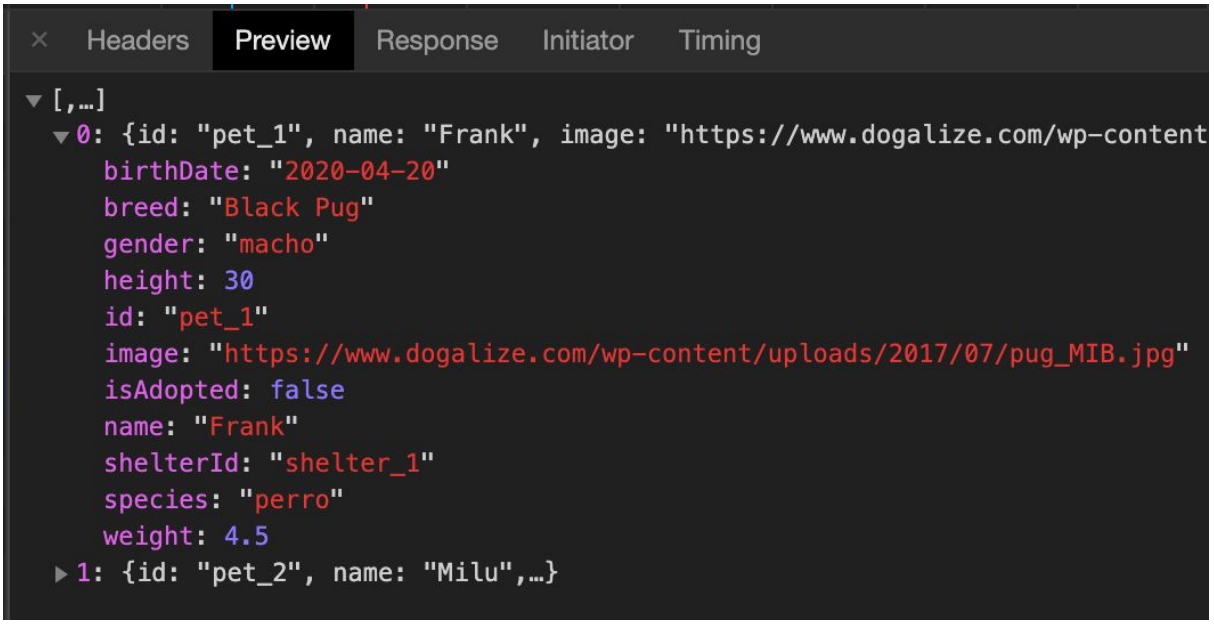
Si queremos saber más detalles acerca de las peticiones que la aplicación ha hecho, no tenemos más que hacer click en la petición.



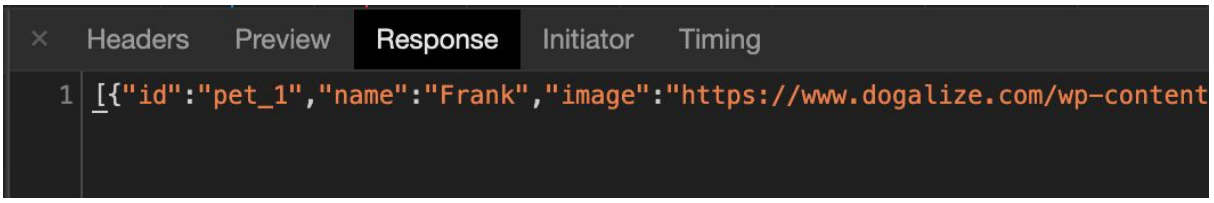
Por defecto, se abrirá siempre la pestaña “Headers” primero, donde podremos ver todos los datos de la petición como el endpoint (la uri), el método HTTP, las cabeceras... Si hacemos click en la pestaña de “Preview”, podremos ver la respuesta a la petición de forma estructurada y formateada (en vez de ser un string puro):



Para ver los objetos en detalle, sólo tenemos que desplegarlos haciendo click en las flechas de la izquierda:



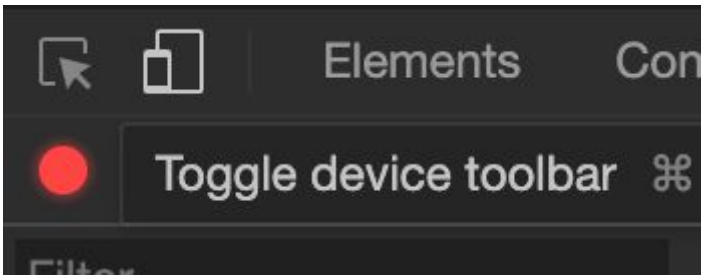
Podemos ver cómo aquí el objeto se ve de una forma muy clara. Sin embargo, si vemos la respuesta desde la pestaña “Response”, lo que veremos es la respuesta tal cual llega del servidor, sin parsear ni estructurar. Veremos un string muy largo de una sola línea:



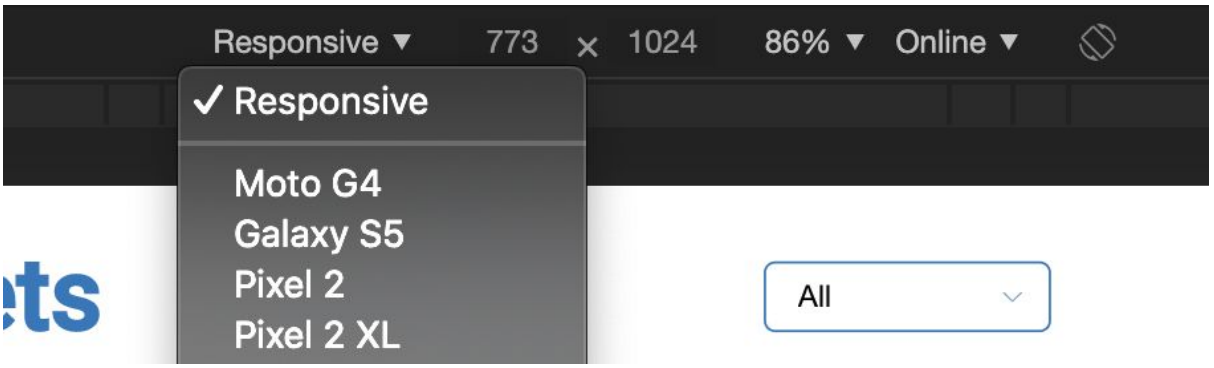
Por tanto, aunque pareciera lógico acceder a la pestaña “Response” para ver la respuesta, lo mejor es hacer esto en la pestaña “Preview”.

Modo dispositivo

Si queremos ver cómo quedará nuestra página o aplicación web en la pantalla de una tablet o un dispositivo móvil, no tenemos por qué utilizar uno de estos. El propio navegador nos ofrece esta función. Sólo tenemos que abrir el inspector y hacer click sobre el icono derecho de los dos iconos en la esquina superior izquierda. Cuando pasemos el ratón sobre él, nos mostrará el texto “Toggle device toolbar”.



Esto hará que se abra el modo dispositivo. Una vez en esta pantalla, podemos escoger la pantalla del dispositivo que queremos usar en el menú que encontraremos en la parte superior de la pantalla:

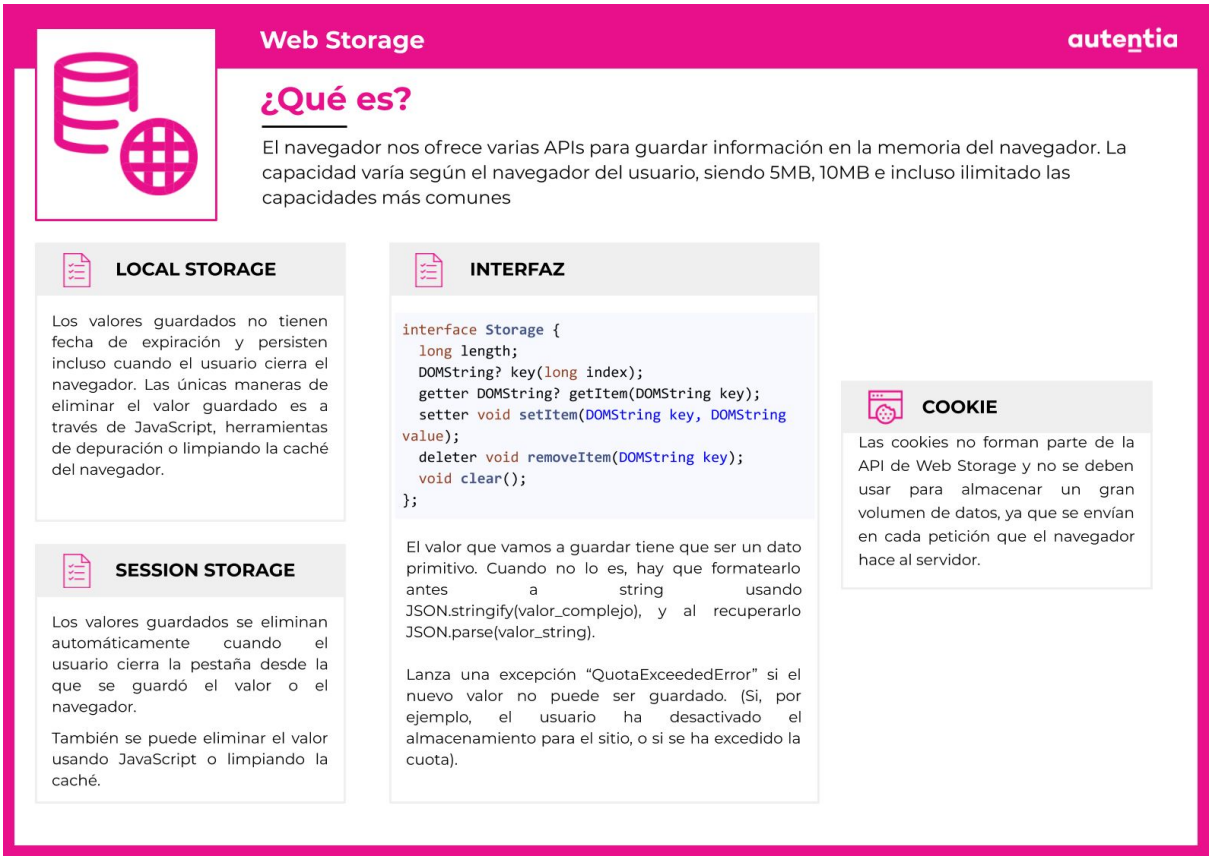


Para simular las distintas orientaciones del dispositivo (horizontal/vertical) sólo tenemos que hacer click sobre el icono que se encuentra arriba a la derecha en esta misma captura. Podemos ver un icono que representa un móvil girando.

Storage

La API de Web Storage nos permite guardar información en la memoria del navegador. El formato es tipo clave/valor. Cuando el valor que se quiere guardar no es un dato primitivo, hay que formatearlo antes a string usando *JSON.stringify(valor_complejo)*, y al recuperarlo *JSON.parse(valor_string)*.

Es importante recalcar que la capacidad varía según el navegador del usuario, siendo 5MB, 10MB e incluso ilimitado, las capacidades más comunes.



Local Storage

Los valores guardados en el *local storage* no tienen fecha de expiración y persisten incluso cuando el usuario cierra el navegador. Las únicas maneras de eliminar el valor guardado son a través de JavaScript, herramientas de depuración o limpiando la caché del navegador.

La WHATWG aconseja usar *local storage* para los datos a los que es necesario acceder a través de múltiples sesiones (ventanas o pestañas) y en los que puede ser necesario almacenar un gran volumen de datos.

Session Storage

Los valores guardados se eliminan automáticamente cuando el usuario cierra la pestaña desde la que se guardó el valor o el navegador.

LA WHATWG aconseja usar *session storage* para los datos que son relevantes para una pestaña, como los detalles de una reserva de un vuelo, mientras que en otras pestañas, el usuario puede hacer otro flujo y reservar otro vuelo.

Cookies

Las *cookies* no forman parte de la API de Web Storage y no se deben usar para almacenar un gran volumen de datos ya que se envían en cada petición que el navegador hace al servidor.

El formato de las *cookies* también es clave/valor y se suelen usar para identificar qué usuario está accediendo a tu página para poder ofrecerle una experiencia personalizada.

Seguridad

La seguridad es una característica fundamental en cualquier aplicación. Implica aspectos de desarrollo, tanto de front como de back, de infraestructura, de protocolos... En esta sección vamos a ver algunas consideraciones a la hora de desarrollar, tipos de ataques que podemos prevenir desde el front y varias tecnologías implicadas.

Vulnerabilidades en el código

Debemos intentar mantener lo más actualizadas posibles las dependencias que estamos utilizando en el proyecto. Al instalar las dependencias, el gestor de paquetes que utilizamos nos avisa si una dependencia es vulnerable. No debemos hacer caso omiso a esa advertencia, pues podríamos tener graves problemas de seguridad.

También es recomendable configurar el entorno de CI para que nos haga una auditoría de las dependencias y el código en busca de vulnerabilidades.

Ataques

Conocer los ataques a las aplicaciones web permite contrarrestar o aplicar soluciones de seguridad. Entre los ataques más comunes están:

Cross-Site Scripting (XSS)

Consiste en conseguir **ejecutar código JavaScript en una página web** para tratar de acceder a datos confidenciales como las credenciales o las cookies, acceder a la cámara, etc.

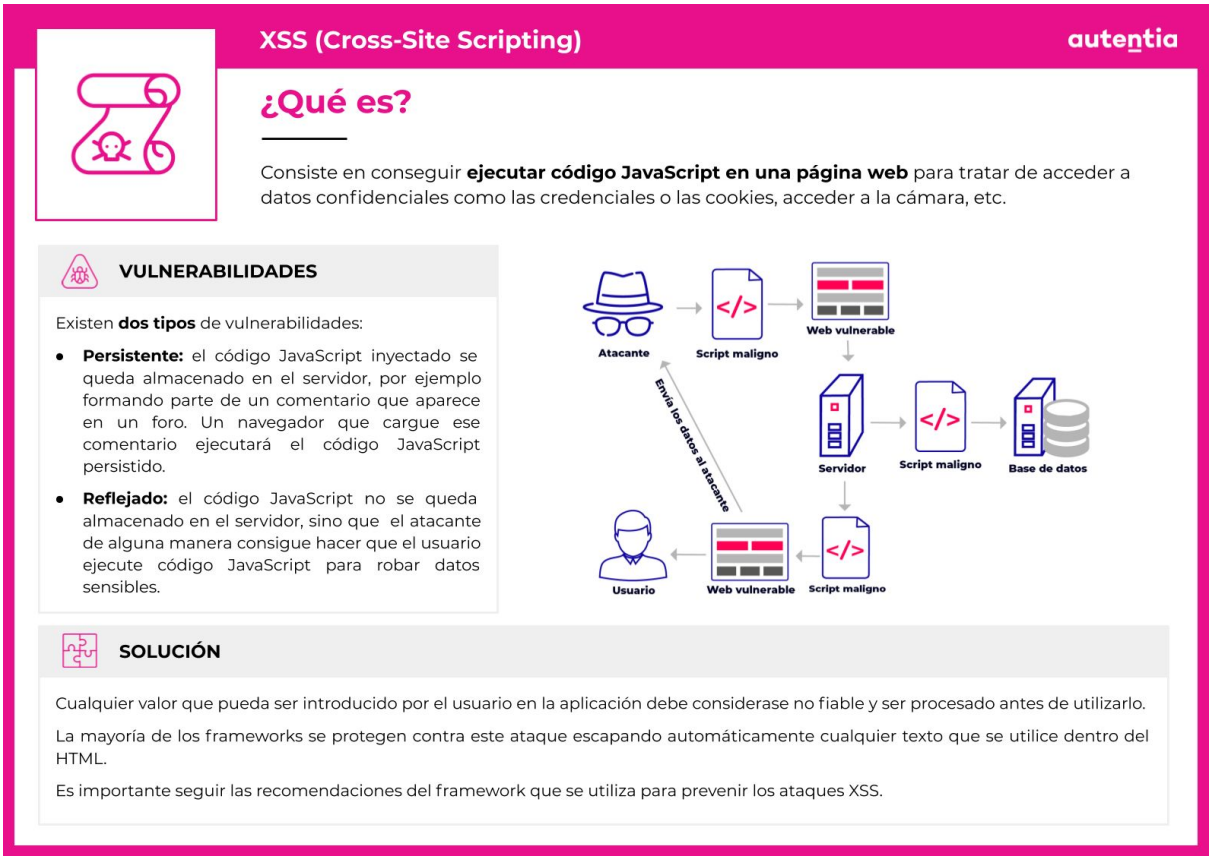
La mayoría de los frameworks JavaScript para construir interfaces de usuario se protegen contra este ataque escapando automáticamente

cualquier texto que se utilice dentro del HTML.

Básicamente, existen dos tipos de vulnerabilidades XSS:

- Persistente: el código JavaScript inyectado se queda almacenado en el servidor, por ejemplo, formando parte de un comentario que aparece en un foro. Un navegador que cargue ese comentario ejecutará el código JavaScript persistido.
- Reflejado: el código JavaScript no se queda almacenado en el servidor, sino que el atacante, de alguna manera, consigue hacer que el usuario ejecute código JavaScript para robar datos sensibles.

La manera de evitar estos ataques es siguiendo las recomendaciones del framework que se utilice para prevenir los ataques XSS.



Cross-Site Request Forgery (CSRF)

Consiste en hacer que **el usuario, sin saberlo, envíe una petición a algún servidor** de algún sistema en el que esté autenticado para hacer una transacción, modificar una contraseña, etc.

Cuando el navegador envía una petición al servidor, envía también todas las cookies asociadas a ese dominio en la petición. Por este motivo, no es necesario que el atacante tenga acceso a las cookies del usuario, solo necesita ser capaz de enviar la petición al servidor con la acción que desea y las cookies serán enviadas automáticamente por el navegador.

Incluso si consumes una API REST, es necesario que te protejas de un ataque CSRF si envías el token de acceso en una cookie. En cambio, si el token se añade dinámicamente a cada petición en la cabecera "Authorization", se previene el ataque CSRF.

La mayoría de los frameworks del lado servidor, para protegerse contra este ataque, generan un token CSRF que pasan al cliente y validan que reciben el mismo token de vuelta cuando el cliente envía una petición HTTP.

CSRF (Cross-Site Request Forgery)

autentia

¿Qué es?

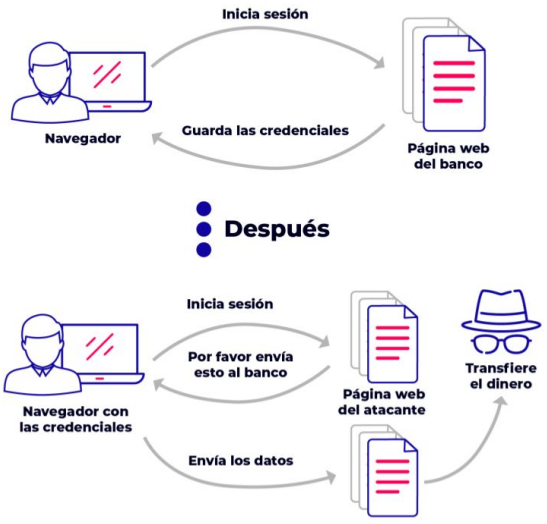
Consiste en hacer que **el usuario, sin saberlo, envíe una petición a algún servidor de algún sistema** en el que esté autenticado para hacer una transacción, modificar una contraseña, etc.

VULNERABILIDAD

Cuando el navegador envía una petición al servidor, envía también todas las cookies asociadas a ese dominio en la petición. Por este motivo no es necesario que el atacante tenga acceso a las cookies del usuario, solo necesita ser capaz de enviar la petición al servidor con la acción que desea y las cookies serán enviadas automáticamente por el navegador.

SOLUCIÓN

La técnica más utilizada para protegerse contra este ataque es generar un token CSRF en el lado del servidor que se envía al cliente. En cada petición HTTP que el cliente envié, el servidor espera recibir el token enviado. Si el token falta o el valor es incorrecto, la petición se rechaza.



Cabeceras HTTP más utilizadas : Seguridad

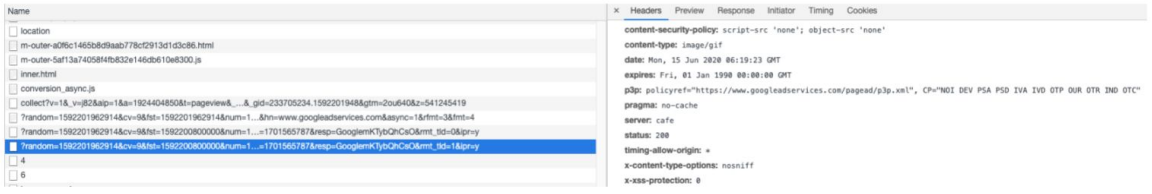
autentia

¿Qué son?

Las **cabeceras de seguridad** HTTP son una **parte fundamental para la seguridad de nuestra web**. Implementarlas dentro de nuestro sitio web **nos protege de ataques** como XSS, inyección de código o clickjacking, entre otros.

CABECERAS

- **HTTP Strict Transport Security (HSTS)**: solo se permite conexiones HTTPS. Esto evita incidentes de seguridad como el secuestro de cookies.
- **Content Security Policy (CSP)**: es una capa de seguridad adicional que ayuda a prevenir ataques de inyección de código.
  - Estableciendo whitelist de contenidos como js, images, font, etc.
  - Permite establecer políticas de navegación (a qué recursos se puede redirigir al usuario).
  - Podemos generar informes o limitar el uso de recursos como plugins.
- **Cross Site Scripting Protection (X-XSS):** habilita un filtro en los navegadores contra ataques XSS.
- **X-Frame-Options:** restringe el renderizado de objetos como <frame>, <iframe>, <embed> o <object> para prevenir los ataques de clickjacking.
- **X-Content-Type-Options:** esta cabecera evita que el usuario pueda determinar el tipo de archivo que espera el servidor e intente camuflar bajo otro tipo de archivos, por ejemplo, hacer pasar script de php como imágenes.



Políticas y aspectos de seguridad

Otros aspectos de seguridad a tener en cuenta, son el uso del Local Storage frente a las Cookies para almacenar credenciales, CORS o CSP.

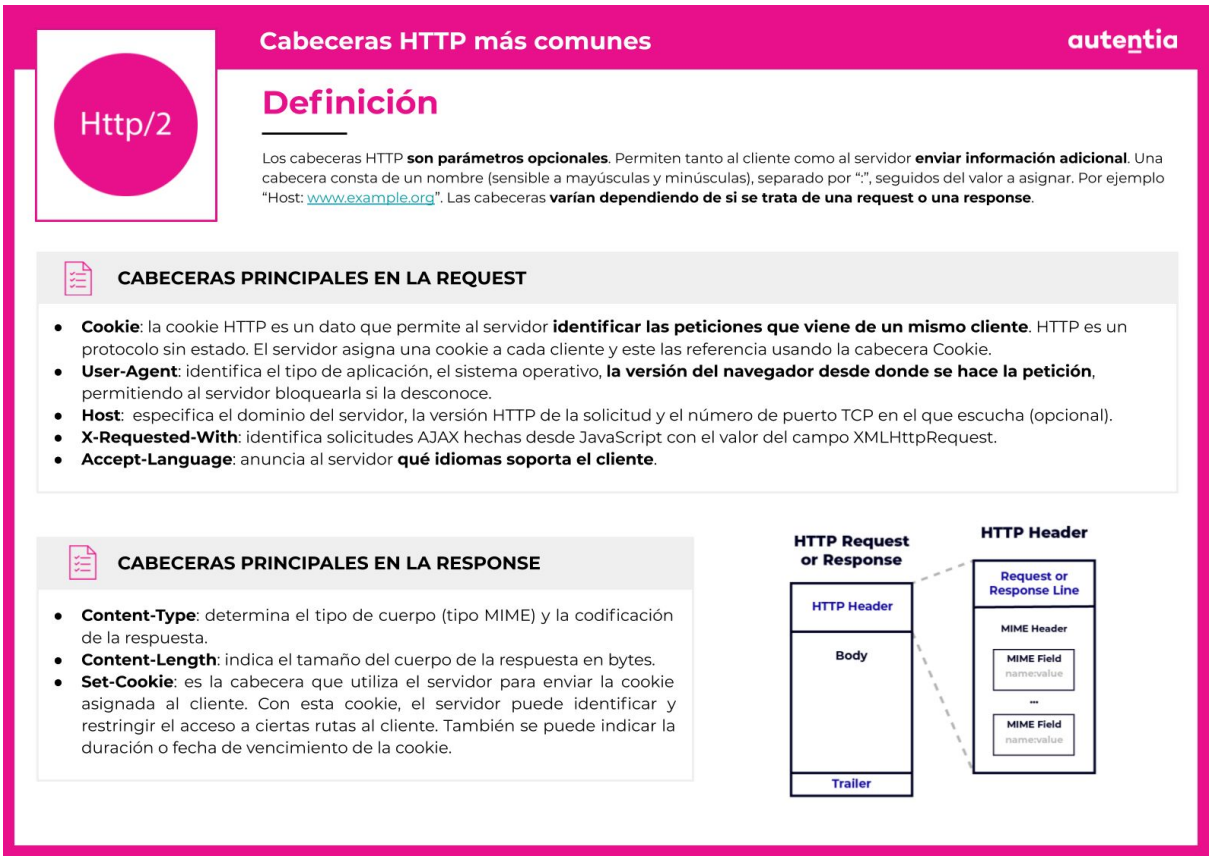
Local Storage vs. Cookies

La mayoría de las veces, en el front vamos a querer guardar el token de acceso del usuario en el navegador para mantener al usuario logado, incluso cuando refresque la página, y evitar que tenga que introducir otra vez las credenciales ya que perjudica la experiencia del usuario.

Tenemos dos opciones:

- Web storage: guardar el token en el local storage o session storage. Pero, si un atacante logra conseguir ejecutar código JavaScript en el navegador, por ejemplo, a través de una extensión del navegador, puede coger el token y robar las credenciales. Se debe evitar guardar datos sensibles en local o session storage.
- Cookies: son una opción más segura a la hora de almacenar el token de acceso del usuario, debido a que proporcionan dos flags: “HttpOnly” que impide que javascript pueda leer o manipular el valor de la cookie y “Secure” que nos garantiza que la cookie solo se va a enviar cuando estamos navegando en una página https. En este caso, es el servidor el que envía la cookie con el token de acceso después de autenticarse el usuario correctamente en el sistema y también es el servidor el que se encarga de refrescar el token cuando expira si estamos usando OAuth.

  Pero, al usar las cookies en el servidor para autenticar al usuario, nos arriesgamos a un ataque CSRF. Para impedir este ataque, el servidor debe enviarnos otra cookie marcada solo como “Secure” con el token CSR para poder leerlo con JavaScript y añadirlo dinámicamente en la cabecera “X-XSRF-TOKEN” de la petición que enviaremos al servidor, donde se validará.



CORS

Cross-Origin Resource Sharing es un mecanismo que permite al navegador acceder a recursos de otros dominios. Mientras que recursos como imágenes, hojas de estilo o scripts no tienen restricciones, otro tipo de peticiones, como las llamadas AJAX, están limitadas por la política de mismo origen de los navegadores.

El mecanismo de CORS es sencillo. El cliente especifica el dominio de origen en el header **Origin** de la petición. El servidor responde correctamente con algunos headers si la petición está autorizada o con error si no lo está.

Cuando las peticiones son complejas, los navegadores realizan una petición *preflighted*. En lugar de realizar directamente la petición, primero consultan mediante el método OPTIONS si es posible realizarla. En caso afirmativo, se

lanza la petición real. Esto previene de cambios no deseados en el servidor.

Se considera una petición simple cuando se cumple:

1. El método es GET, POST o HEAD.
2. Sólo contiene los headers Accept, Accept-Language o Content-Language.
3. content-type es text/plain, application/x-www-form-urlencoded o multipart/form-data.

Es importante establecer una configuración para CORS adecuada en el servidor. De lo contrario, podemos exponernos al robo de datos e incluso suplantación de identidad en los casos en que utilicemos credenciales en este tipo de peticiones.

Un problema típico que podemos encontrar cuando probamos nuestras API, es descubrir que, mientras que todo funciona perfectamente con **Postman** u otras herramientas, las peticiones desde el navegador fallan. Esto es debido a la política de seguridad de mismo origen que incorporan los navegadores. Postman y otras herramientas no utilizan esta política.

Configuración

Cuando se realiza una petición cross-origin, el **navegador** puede introducir varias cabeceras para detallar información sobre la misma:

- Origin: el dominio origen de la petición.
- Access-Control-Request-Method: el método HTTP para el que se consulta el permiso.
- Access-Control-Request-Headers: las cabeceras que se pretenden enviar en la petición.

Por motivos de privacidad, CORS se utiliza para peticiones anónimas. No envía cookies que identifican al usuario. Si queremos alterar este

comportamiento, podemos utilizar las siguientes opciones para nuestra petición:

```
fetch('https://example.com', {
 mode: 'cors',
 credentials: 'include'
})
```

Habilitar las peticiones cross-origin **depende fundamentalmente del servidor**. Es éste el que controla desde qué dominios puede recibir este tipo de peticiones y para qué métodos. Las cabeceras que incorpora en las respuestas pueden ser:

- Access-Control-Allow-Origin**:** indica si el navegador puede acceder al recurso desde el dominio de origen. Un valor de "*" indica que el acceso es público, mientras que si devuelve el dominio origen, significa que nuestro dominio está autorizado específicamente. En caso de no tener permiso, se obtendrá un error de CORS.
- Access-Control-Allow-Methods: los métodos HTTP permitidos, como una lista de valores separados por comas.
- Access-Control-Allow-Headers: los headers permitidos en la petición, como una lista de valores separados por comas.
- Access-Control-Allow-Credentials: permite el uso de credenciales en la petición como un valor booleano. Las credenciales pueden ser cookies, headers de autorización o certificados. Si se devuelve false en una petición no preflighted, el cliente ignorará la respuesta. Sólo se admiten peticiones con credenciales si Access-Control-Allow-Origin **tiene un valor específico, no "*".**
- Access-Control-Expose-Headers: permite controlar qué headers se exponen al cliente. Por defecto, sólo se exponen los siete headers seguros (Cache-Control, Content-Type, Content-Length,

Content-Language, Expires, Last-Modified y Pragma).

- Access-Control-Max-Age: controla cuánto tiempo pueden ser cacheados los resultados de una petición preflighted (métodos y cabeceras permitidas).

Es imprescindible establecer una configuración adecuada para CORS si queremos mantener nuestros datos a salvo. Utilizar orígenes específicos, en lugar de “*” y restringir los métodos que pueden utilizarse desde cada uno de estos orígenes a los estrictamente necesarios, puede prevenir ataques basados en CORS.

Además, podemos exigir que el cliente se identifique con credenciales para aumentar la seguridad. Sin embargo, debemos ser conscientes de que, al enviar cualquier tipo de credenciales, estamos creando un punto más de ataque del que tendremos que preocuparnos.

El valor “*” para los orígenes permitidos sólo debería usarse cuando nuestra API sirva únicamente datos públicos que queremos que sean accedidos sin ninguna restricción.

Content Security Policy

CSP es una capa de seguridad adicional que pretende evitar y mitigar algunos tipos de ataque como XSS o inyección de datos. Permite que el servidor ofrezca una lista de los dominios de donde se pueden cargar distintos tipos de recursos, incluidos scripts, imágenes, etc., que no son afectados por la política de mismo origen. También se pueden definir los protocolos permitidos.

Está diseñada para ser totalmente retrocompatible. Si el navegador no soporta CSP o el servidor no lo habilita, se utiliza la política de mismo origen que ya hemos visto en el apartado de CORS.

Para habilitar CSP, se debe retornar el *header* Content-Security-Policy en la respuesta del servidor. Su valor es una cadena de caracteres que describe la política de seguridad del contenido a través de una o varias directivas,

separadas por punto y coma.

Las directivas más comunes son:

- default-src.
- script-src.
- style-src.
- font-src.
- img-src.
- media-src.
- connect-src.
- child-src.

La directiva *default-src* proporciona una política por defecto para cuando no hay una política concreta para un tipo de recurso. Si *default-src, style-src o script-src* se proporcionan, los estilos y scripts incluidos *inline* en el documento HTML serán ignorados.

Las directivas aceptan uno o varios valores separados por espacios, entre los que destacan:

- Dominios, incluidos puertos o esquemas de protocolos.
- `'self'`, que se refiere al propio dominio de la web con todos sus componentes (pero no a sus subdominios).
- `'None'`, que no permite cargar ese tipo de recurso desde ningún origen.
- Diferentes patrones con comodín “*”.

El siguiente ejemplo muestra una política en la que todos los recursos deben ser cargados desde el dominio de origen, excepto las imágenes, que pueden cargarse mediante https desde otro dominio:

```
Content-Security-Policy: default-src 'self'; img-src
https://*.example.com
```

También existe una versión menos restrictiva de CSP. Esta reporta violaciones de la política pero no impide el acceso a los recursos. Para ello, se emplea el header Content-Security-Policy-Report-Only. Ambas cabeceras son compatibles y una no anula la otra. Para que los informes se envíen, hay que incluir la directiva report-uri.

En el siguiente ejemplo, se permite el acceso a recursos desde un dominio diferente pero creamos informes cuando se accede a ellos:

```
Content-Security-Policy-Report-Only:default-src 'self'; report-uri
http://reportstore.example.com
```

Estas son las bases de CSP, sin embargo, esta tecnología ofrece más posibilidades que puedes explorar en profundidad para gestionar la seguridad de tu aplicación de forma todavía más fina.

Web APIs

Cuando se programa para la web, hay un gran número de APIs disponibles de forma nativa que nos facilitarán enormemente el trabajo con tareas como caché, pagos, sistema de ficheros, etc. En esta sección, explicaremos brevemente unas pocas pero hay otras muchas que podrás encontrar en esta página.

Caché

Esta API nos permite cachear las peticiones que queramos. Cuando hagamos una petición mediante esta API, cacheará automáticamente el resultado. Nos ofrece distintos métodos como por ejemplo:

- `Cache.match(request, options)`: devuelve una promesa que resuelve la respuesta asociada con la primera request que haga match en el objeto Cache.
- `Cache.add(request)`: recoge una URL, realiza la petición y añade el resultado a la caché. Esta operación es equivalente a realizar un `fetch()` y después un `put()` para añadir el resultado a la caché.
- `Cache.put(request, response)`: coge la request y su respuesta y los añade a la caché.
- `Cache.delete(request, options)`: devuelve una promesa con valor *true* si se ha encontrado y eliminado la entrada relacionada con la request dada y *false,* en caso contrario.

Pagos

Implementar un sistema de pago suele ser tedioso, tanto para el desarrollador como para el usuario de nuestra aplicación, en cuanto a formularios se refiere. Aquí es donde entra esta API que permite escoger entre las tarjetas guardadas en el navegador para realizar un pago, convirtiéndolo en un proceso simple para ambas partes. Aquí encontraréis un artículo detallado que explica cómo funciona esta API y cómo implementarla con ejemplos de código.

Files

Esta API nos aporta información sobre ficheros y permite a JavaScript acceder a su contenido en una página web. Normalmente, se utiliza cuando el usuario selecciona un archivo mediante una etiqueta de HTML `<input>`. Esta acción devolverá un objeto `FileList` del cual se obtienen los `File`.

Un objeto `File` es un tipo específico de `Blob (Binary Large Object)` y puede usarse en cualquier contexto en el que se pueda usar este tipo.

Mutation Observer

Hoy en día, las aplicaciones web son cada vez más complejas y los cambios en el DOM pueden ser frecuentes. Como resultado, es probable que haya casos en los que una aplicación necesite escuchar y reaccionar ante un cambio específico en el DOM. Gracias a MutationObserver esto es muy fácil. Un código básico quedaría más o menos así:

```
let observer = new MutationObserver(callback);
function callback (mutations) {
  // hacer algo aquí
```

```
}
observer.observe(targetNode, observerOptions);
```

Cuando terminemos de observar, no podemos olvidarnos de ejecutar `observer.disconnect();`

Request animation frame

Cuando queremos ir actualizando un valor visual de algún elemento para realizar una animación, podemos usar un método que acepta un callback y decirle al navegador que lo ejecute para actualizar la animación antes de que pinte el siguiente frame.

Para utilizar este método, se tiene que hacer una especie de recursión, como en este ejemplo:

```
function step(timestamp) {
 // Actualizar algún valor del elemento que queramos animar, como su posición, color...
 window.requestAnimationFrame(step);
}

window.requestAnimationFrame(step);
```

Para pausar la animación cuando queramos, debemos envolver la llamada en `window.requestAnimationFrame(step);` interna de la función *step* en un bloque condicional *if*. De esta forma, podemos ponerle por ejemplo, un límite de tiempo o de posición.

Gestión de memoria

Todo programa necesita usar variables y por lo tanto necesita reservar una cierta cantidad de memoria para ellas. Por desgracia, la cantidad de memoria es limitada y por ello, es necesario liberar la memoria usada por variables que no se van a usar en el futuro. Este proceso de reserva y liberación de memoria se puede hacer de manera manual o automática.

En la mayoría de los lenguajes de programación modernos, la gestión de memoria se realiza de manera automática ya que, en programas complejos, es fácil cometer errores de programación al reservar y liberar memoria. Además, esto nos permite hacer un código más sencillo de entender y por lo tanto más fácil de mantener:

```
function f() {
  var miLista = [1, 2, 3]; //Se guarda memoria para miLista

  //Trabajar con la variable miLista
}

f(); //La variable miLista solo se usa dentro de f

//Se detecta que la variable miLista ya no se puede utilizar y se
libera automáticamente el espacio que se había reservado para ella
```

La liberación de memoria la realizan los recolectores de basura que son programas que buscan si hay objetos que no son referenciados (y por tanto, ya no se usan) y liberan la memoria que ocupaban. Los recolectores de basura modernos son muy eficientes y pueden liberar casi toda la

memoria que no se vaya a utilizar pero por desgracia no son perfectos. Cuando un bloque de memoria no es liberado, decimos que se ha producido un **memory leak**.

Hoy en día, usando frameworks y linters, es difícil tener memory leaks. Existen algunas excepciones, como por ejemplo, si se usa RxJS sin gestionarlo debidamente. Si usamos JavaScript puro, es fácil tener memory leaks y por lo tanto, es útil conocer cómo localizarlos y cuáles son los errores más frecuentes para poder evitarlos.

Errores frecuentes

Variables globales accidentales

Toda variable global es siempre accesible y por lo tanto, nunca va a ser liberada. Al asignar valores a variables sin declarar o si usamos *this* de manera incorrecta podemos crear sin querer variables globales que nunca serán liberadas. Por ejemplo:

```
function foo() {
  hola = "Hola a tod@s!"; //La variable hola es global
  this.adios = "¡Adios!"; //La variable adios es global
}

for (i = 0; i < array.length; i++){  //La variable i es global
    ...
}
```

Para solucionar este error nos bastaría con declarar las variables con *let*, *const* o usar el modo estricto:

```
function foo() {
  let hola = "Hola a tod@s!";
  const adios = "¡Adios!";
```

```
}

for (let i = 0; i < array.length; i++){
    ...
}
```

Timers o callbacks olvidados

Este error se da al realizar acciones periódicas usando `setInterval()` o `setTimeout()` llamado recursivamente. Si estas acciones no terminan nunca, puede haber variables que crezcan de manera descontrolada. Este error es especialmente peligroso en arquitecturas *Single Page Application* ya que no se refresca nunca la página entera.

```
var globalArray = [];

setInterval(function() {
  var hugeString = new Array(1000000).join('Hola');
  globalArray.push(hugeString);
}, 1000);
```

En este ejemplo, cada segundo hacemos que la variable *globalArray* ocupe cada vez más hasta llenar la memoria del navegador.

Es importante también guardar una referencia al timer que hemos creado para poder eliminarlo cuando ya no lo necesitemos:

```
function refrescarPantalla() {
  ...
}

const miIntervalo = setInterval(refrescarPantalla(), 1000);

//Hacemos cosas hasta que ya no haga falta usar refrescarPantalla
```

```
clearInterval(miIntervalo); //Al haber guardado la variable
miIntervalo podemos parar el timer
```

Closures

Al anidar funciones en JavaScript, las funciones interiores tienen acceso a las variables definidas en las funciones exteriores. Podemos tener una fuga de memoria si se dan las siguientes condiciones:

- Una función A que devuelve una función B.
- B utiliza una variable local de A.
- Referenciamos A en una variable.

Entonces la variable local declarada en A nunca se libera.

```
function getLongList(repeatTimes) {
  const hugeString = new Array(repeatTimes).join('Hola');
  return function foo() {
    return hugeString; //La función foo tiene acceso a hugeString
  }
}

var bar = getLongList(1000000); //Asignamos la función a bar

bar(); //Podemos hacer llamadas a bar. Al hacer esto llamamos a la
función interna foo, que necesita el valor de hugeString, por lo
tanto el valor de hugeString no puede ser eliminado nunca.
```

Para solucionar este memory leak tenemos que entender cómo funcionan los *closures* en JavaScript y tener cuidado con las funciones que se utilizan como *callback*.

Event listeners

Al crear un *event listener,* este sigue activo hasta que llamamos a `removeEventListener()` o el elemento asociado es eliminado del DOM.

```
const hugeString = new Array(repeatTimes).join('Hola');
document.addEventListener('keyup', function() {
    doSomething(hugeString); // hugeString se usa dentro del
listener, por lo que se guarda en memoria para siempre
});
```

Al haber usado una función anónima, no podemos eliminar este listener nunca, a pesar de que ya no se vaya a usar. Podemos corregir esto de la siguiente manera:

```
function listener() {
  const hugeString = new Array(repeatTimes).join('Hola');
  doSomething(hugeString);
}

document.addEventListener('keyup', listener);

// Hacemos cosas hasta que ya no necesitemos el listener

document.removeEventListener('keyup', listener);
```

Elementos eliminados del DOM

Si guardamos un elemento del DOM en una variable, aunque lo eliminemos del DOM, no se borra la referencia:

```
const div = document.createElement('div');
div.id = 'divEliminado';

document.body.appendChild(div);
```

```
document.body.removeChild(document.getElementById('divEliminado'));

//A pesar de haber hecho removeChild el elemento sigue en el heap
ya que la constante div lo referencia
```

Podemos solucionarlo de la siguiente manera:

```
function addDiv() {
  const div = document.createElement('div');
  div.id = 'divEliminado';
  document.body.appendChild(div);
}

addDiv();

document.body.removeChild(document.getElementById('divEliminado'));

//Ahora el removeChild funciona bien ya que no existe ninguna
variable que referencie a divEliminado
```

Observables

Si usamos un *framework* que tenga observables, es muy importante que, al cambiar de pantalla o cuando la acción de un observable haya finalizado, hacer `.unsuscribe()` por cada `.suscribe()` que hayamos hecho.

Cómo detectar *memory leaks*

Vamos a aprender a usar la consola de depuración de Chrome para detectar *memory leaks*. Usaremos como ejemplo la siguiente página, en la que declaramos un botón que al hacer click hace un leak a la variable *hugeArray* y crea mil nodos en el DOM.

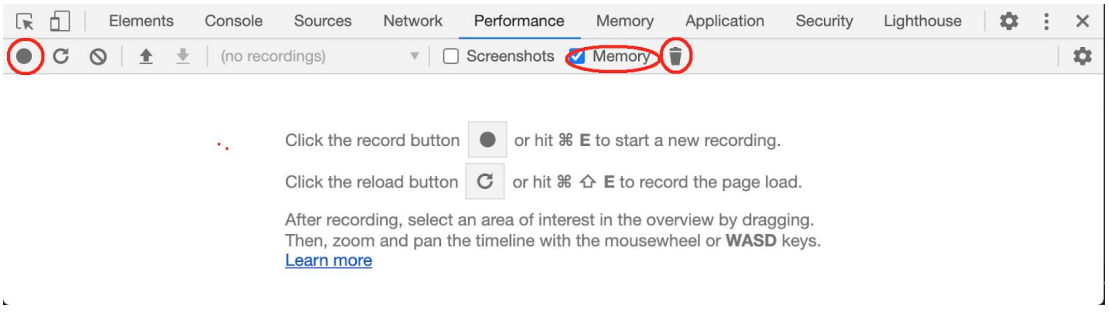
```
<html>
  <body>
    <button onClick="leak()">Leak</button>
  </body>

  <script>
    const hugeArray = new Array(1000000).join('Hola');
    var leakingArray = [];

    function leak() {
      leakingArray.push(hugeArray);

      for (var i = 0; i < 1000; i++) {
        document.body.appendChild(document.createElement('div'));
      }
    }
  </script>
</html>
```

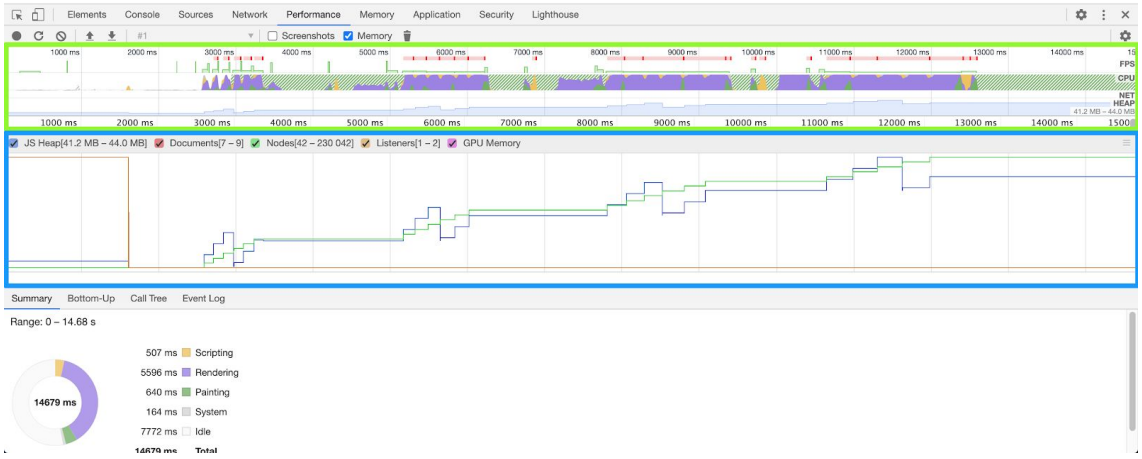
Si abrimos esta página en el navegador y abrimos la pestaña de *Performance*, veremos la siguiente pantalla:



De esta pantalla nos interesan los 2 botones marcados en rojo y tener seleccionada la checkbox de memoria. El botón con el icono de grabar sirve para empezar el análisis de la página. El botón del contenedor de basura fuerza a que el *recolector de basura* se ejecute y limpie la memoria.

Para analizar si hay algún memory leak en la página, le daremos al botón de grabar y a continuación, al botón del recolector de basura. Después, haremos varios clicks en el botón de *Leak* y luego al botón de basura.

Repetimos este paso varias veces para poder apreciar mejor las fugas de memoria y volvemos a pulsar en el botón de grabar para acabar el análisis. Veremos la siguiente pantalla:



En la sección marcada en verde, es importante que seleccionemos todo el cronograma para ver la gráfica entera. Podemos hacerlo fácilmente haciendo scroll en la sección. En la sección marcada en azul es donde identificamos que hay un *memory leak* en nuestra página. Si nos fijamos en la gráfica pintada en azul, que representa la memoria usada por JavaScript, podemos ver primero una fuerte caída seguida de 4 escaleras que suben y luego caen. La primera caída es normal y representa la primera vez que hemos llamado al recolector de basura. Las escaleras que suben y luego caen, también son normales pues son el código que se ejecuta tras hacer click múltiples veces y luego la llamada al recolector de basura. El hecho que nos tiene que preocupar de la gráfica es que tras cada llamada al recolector de basura la gráfica azul no vuelve a la altura a la que estaba antes sino que es cada vez más alta. Esto es una señal clara de que en la función que se ejecuta al hacer click en el botón hay un *memory leak*.

Si nos fijamos en la gráfica verde vemos que es siempre creciente. Esto es un problema y nos indica que el DOM está creciendo de manera descontrolada.

Bibliografía

Estas son las fuentes que hemos consultado y en las que nos hemos basado para la redacción de este material:

- Documentación para desarrolladores de Mozilla.org:

  https://developer.mozilla.org/en-US/docs/Web/HTML

- Documentación para la comunidad de Digital Ocean:

  https://www.digitalocean.com/community/tutorials?primary_filter=tutorial_series

- Consorcio de la World Wide Web (W3C):

  https://www.w3.org

- Especificación del estándar HTML del Grupo de trabajo de tecnología de aplicación de hipertexto web (WHATWG por su nombre en inglés):

  https://html.spec.whatwg.org

- Pautas de accesibilidad al contenido web de la W3C:

  https://www.w3.org/WAI/standards-guidelines/wcag/

- Datos de la OMS sobre discapacidad visual y auditiva:

  https://www.who.int/features/factfiles/blindness/es/

  https://www.who.int/features/factfiles/deafness/es/

- Datos sobre daltonismo:

  https://www.color-blindness.com/

- Distribución por edades de la población española:

  https://www.ine.es/jaxi/Tabla.htm?path=/t20/e245/p08/l0/&file=02002.px

- Especificación del Lenguaje ECMAScript:

  http://www.ecma-international.org/ecma-262/10.0/index.html#sec-promise-objects

- Adictos al Trabajo
  - ES6: el remozado JavaScript. Parte I: variables y constantes:

    https://www.adictosaltrabajo.com/2018/04/30/es6-javascript-variables-y-constantes/

  - ES6: el remozado JavaScript. Parte II: funciones, objetos y arrays:

    https://www.adictosaltrabajo.com/2018/05/08/es6-el-remozado-javascript-parte-ii-funciones-objetos-y-arrays/

  - ES6: el remozado JavaScript. Parte III: clases y otras novedades del lenguaje:

    https://www.adictosaltrabajo.com/2018/05/15/es6-el-remozado-javascript-parte-iii-clases-y-otras-novedades-del-lenguaje/

  - Async/Await en javaScript:

    https://www.adictosaltrabajo.com/2017/02/09/asyncawait-en-javascript/

- Documentación oficial Babel:

  https://babeljs.io/docs/en/

- Documentación oficial PostCSS:

  https://github.com/postcss/postcss/tree/master/docs

- Documentación oficial Webpack:

  https://webpack.js.org/concepts/

- Documentación oficial Snowpack:

  https://www.snowpack.dev

- Documentación oficial Parcel:

  https://es.parceljs.org/getting_started.html

- Documentación oficial npm:

  https://docs.npmjs.com

- Documentación oficial Yarn:

  https://yarnpkg.com/getting-started

- Documentación oficial seguridad OWASP:

  https://owasp.org

Lecciones aprendidas con esta guía

Conocer bien nuestro entorno es tan importante como conocer nuestras propias capacidades. En el desarrollo de front, el entorno tiene multitud de aristas. Entenderlas, amoldarnos a ellas y saber limarlas en nuestro beneficio, es fundamental para conseguir un resultado que cumpla con nuestros objetivos.

Con esto en mente, esta guía ofrece un acercamiento a las principales cuestiones que entran en juego a la hora de construir una aplicación web, desde la configuración del entorno de desarrollo, hasta las particularidades que debemos tener en cuenta durante la ejecución. Podemos destacar las siguientes:

- Configurar el proyecto y sus dependencias mediante gestores de paquetes, y empaquetar nuestro código para entornos de producción.
- Transpilar nuestro código a JavaScript a partir de otros lenguajes de programación como TypeScript o CoffeeScript.
- Utilizar herramientas que nos faciliten el desarrollo de nuestro código y su compatibilidad con el mayor número de navegadores posible.
- Conocer las APIs más relevantes que nos ofrecen los navegadores para las tareas más frecuentes.
- Conocer los problemas que debemos prevenir, tanto a nivel de seguridad como de gestión de memoria.
- Aprender el manejo de las herramientas de depuración para detectar y resolver los errores de nuestras aplicaciones.

Esta guía no pretende cubrir por completo todos los aspectos. El entorno de front está en constante evolución y surgen nuevas tecnologías continuamente. Mantenerse al día es difícil, pero indispensable.

¡Conoce más!