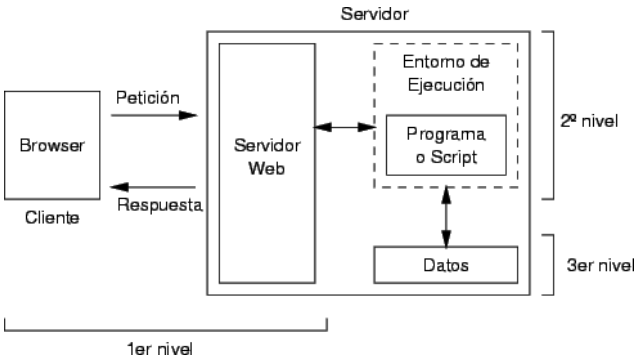
Web Components

https://www.webcomponents.org/

INTRODUCCIÓN

Arquitectura Web

Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



Aplicaciones web vs Sitios web

- Aplicaciones web
  - El objetivo principal de una aplicación es que el usuario realice una tarea. También pueden entenderse como un programa que se utiliza desde el navegador. Para crearlos, se usan los lenguajes CSS, HTML, JavaScript y se puede utilizar software gratuito de fuente abierta, como Drupal, Symfony o Meteor.
- Sitios web
  - El objetivo principal de un sitio web es entregar información. Por lo tanto, consumir contenidos es la tarea más importante hacen los usuarios en este tipo de plataformas.
  - Esta idea puede sonar confusa, ya que todos los sitios incluyen algún llamado a la acción adicional, como por ejemplo realizar un contacto o suscribirse a un newsletter. La diferencia está en que estas interacciones representan una parte pequeña y usualmente se pueden lograr solo después de guiar al usuario a través del contenido.

Enfoques

- Hay múltiples enfoques generales para crear una interfaz de usuario web moderna:
  - Sitios web:
    - Static Side Generation (SSG): Generación estática del sitio (compilados)
    - Incremental Static Regeneration (ISR): Regeneración estática incremental basada en tiempo o en caches.
  - Aplicaciones que representan la interfaz de usuario desde el servidor.
    - Server-Side Rendering (SSR)
    - Multi Page Application (MPA)
  - Aplicaciones que representan la interfaz de usuario en el cliente.
    - Client-Side Rendering (CSR)
    - Single Page Applications (SPA)
  - Aplicaciones híbridas que aprovechan los enfoques de representación de la interfaz de usuario en servidor y en cliente.

Interfaz de usuario representada en el servidor

- Una aplicación de interfaz de usuario web que se representa en el servidor genera dinámicamente el lenguaje HTML y CSS de la página en el servidor en respuesta a una solicitud del explorador. La página llega al cliente lista para mostrarse.
- Ventajas:
  - Los requisitos de cliente son mínimos porque el servidor realiza el trabajo de la lógica y la generación de páginas: Excelente para dispositivos de gama baja y conexiones de ancho de banda bajo, Permite una amplia gama de versiones de explorador en el cliente, Tiempos de carga de página iniciales rápidos, JavaScript mínimo o inexistente para ejecutar en el cliente.
  - Flexibilidad de acceso a recursos de servidor protegidos: Acceso de bases de datos, Acceso a secretos, como valores para las llamadas API a Azure Storage.
  - Ventajas del análisis de sitios estáticos, como la optimización del motor de búsqueda.
- Inconvenientes:
  - El costo del uso de proceso y memoria se centra en el servidor, en lugar de en cada cliente.
  - Lentitud, las interacciones del usuario requieren un recorrido de ida y vuelta al servidor para generar actualizaciones de la interfaz de usuario.

Interfaz de usuario representada por el cliente

- Una aplicación representada en el cliente representa dinámicamente la interfaz de usuario web en el cliente, actualizando directamente el DOM del explorador según sea necesario.
- Ventajas:
  - Permite una interactividad enriquecida que es casi instantánea, sin necesidad de un recorrido de ida y vuelta al servidor. El control de eventos de la interfaz de usuario y la lógica se ejecutan localmente en el dispositivo del usuario con una latencia mínima. Aprovecha las funcionalidades del dispositivo del usuario.
  - Admite actualizaciones incrementales, guardando formularios o documentos completados parcialmente sin que el usuario tenga que seleccionar un botón para enviar un formulario.
  - Se puede diseñar para ejecutarse en modo desconectado. Las actualizaciones del modelo del lado del cliente se sincronizan finalmente con el servidor una vez que se restablece la conexión.
  - Carga y costo del servidor reducidos, el trabajo se descarga en el cliente. Muchas aplicaciones representadas por el cliente también se pueden hospedar como sitios web estáticos.
- Inconvenientes:
  - El código de la lógica debe descargarse y ejecutarse en el cliente, lo que se agrega al tiempo de carga inicial.
  - Los requisitos de cliente pueden excluir a los usuarios que tienen dispositivos de gama baja, versiones anteriores del explorador o conexiones de ancho de banda bajo.

Single-page application (SPA)

- Un single-page application (SPA), o aplicación de página única es una aplicación web o es un sitio web que utiliza una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio.
- En un SPA todo el códigos de HTML, JavaScript y CSS se carga de una sola vez o los recursos necesarios se cargan dinámicamente cuando lo requiera la página y se van agregando, normalmente como respuesta de los acciones del usuario.
- La página no se tiene que cargar otra vez en ningún punto del proceso, tampoco se transfiere a otra página, aunque las tecnologías modernas (como el pushState() API del HTML5) pueden permitir la navegabilidad en páginas lógicas dentro de la aplicación.
- La interacción con las aplicaciones de página única pueden involucrar comunicaciones dinámicas con el servidor web que está por detrás, habitualmente utilizando AJAX o WebSocket (HTML5).

Aplicaciones Isomorficas

- También conocidas como Universal JavaScript, son aquellas que nos permiten ejecutar el código JavaScript, que normalmente ejecutamos en el cliente, en el servidor.
- Una aplicación SPA (Single Page Application) aunque inicialmente tarda mas en cargarse, luego tiene una velocidad de página y experiencia de usuario muy buena. Como desventaja presentan la perdida de SEO: los motores de búsqueda no interaccionan con las páginas cuando las indexan y no generan los contenidos dinámicos por lo que gran parte del contenido puede quedar sin indexar, cosa que no pasa cuando se genera la pagina en el lado del servidor. Otro problema son los navegadores que no soportan o tienen desactivado el JavaScript.
- Las aplicaciones isomorfas permiten:
  - Ejecutar el mismo código en el lado cliente y en el lado servido.
  - Decidir si modifica directamente el DOM o manda una cadena con el HTML modificado.
  - Obtener las ventajas de ambos modos.

Aplicaciones web progresivas

- Las aplicaciones web progresivas (mejor conocidas como PWAs por «Progressive Web Apps») no es un nombre oficial ni formal, es solo una abreviatura utilizada inicialmente por Google para el concepto de crear una aplicación flexible y adaptable utilizando solo tecnologías web.
- Es mucho más fácil y rápido visitar un sitio web que instalar una aplicación, y se pueden compartir enviando un enlace. Por otro lado, las aplicaciones nativas están mejor integradas con el sistema operativo y, por lo tanto, ofrecen una experiencia más fluida para los usuarios, se puede instalar una aplicación nativa para que funcione sin conexión, y los usuarios pueden tocar sus íconos para acceder fácilmente a sus aplicaciones favoritas sin tener que navegar. Las PWA nos brindan la capacidad de crear aplicaciones web que se comportan como nativas.
- Son aplicaciones web que utilizan APIs y funciones emergentes del navegador web junto a una estrategia tradicional de mejora progresiva para ofrecer una aplicación nativa, con la experiencia de usuario de aplicaciones web multiplataforma. Se puede pensar que PWA es similar a AJAX u otros patrones similares, son aplicaciones web desarrolladas con una serie de tecnologías específicas y patrones estándar que les permiten aprovechar las funciones de las aplicaciones nativas y web, aunque no son un estándar formalizado.
- Las PWA deben ser detectables, instalables, enlazables, independientes de la red, progresivas, reconectables, responsivas y seguras.

JAMStack

- JAMStack es un término genérico para referirse a una forma de desarrollar aplicaciones web: Pila JavaScript, API y Markup.
- Jamstack es fundamentalmente una filosofía, una metodología, es un enfoque, una forma de pensar, un conjunto de principios y mejores prácticas. Es una arquitectura, no se trata realmente de tecnologías específicas, no existe un instalador de Jamstack, ni es impulsado por una gran empresa, ni existe algún organismo de normalización que lo controle o defina.
- JAMStack es estática en el sentido de que el contenido es servido como archivos estáticos (Markup) pero los datos utilizados pueden ser dinámicos (JavaScript, API). El HTML se renderiza previamente en archivos estáticos, este código se distribuye a una CDN que es una red global de servidores, y las tareas que alguna vez se procesaron y administraron en el lado del servidor ahora se realizan a través de API y microservicios.
- Componentes de Jamstack
  - JavaScript: las funcionalidades dinámicas son manejadas por JavaScript. No se exige que se use algún framework, ni hay alguna restricción sobre diseñar tu aplicación de alguna manera en particular. JavaScript es lo que hace que un sitio de Jamstack sea dinámico, permite que los activos estáticos cambien dinámicamente a través de la manipulación DOM.
  - APIs: Jamstack no requiere una necesidad real de construir un backend o servidor y se basa principalmente en APIS de terceros, en cambio complementa la tecnología sin servidores y se puede usar con funciones sin servidor, como funciones de Netlify o similares. Los desarrolladores usan API para obtener los datos que necesitan para crear e implementar aplicaciones y sitios Jamstack estáticos. Las API se utilizan a menudo en el cliente cuando desea acceder a contenido dinámico desde el navegador.
  - Markup: abarca lenguajes de plantillas como Markdown por ejemplo y lenguajes de serialización de datos como YAML y JSON. Específicamente, el Markup o marcado en Jamstack se refiere al marcado o contenido "prerenderizado" servido en su forma final, HTML.

WebAssembly (wasm)

- WebAssembly es un nuevo tipo de código que puede ser ejecutado en navegadores modernos. Es un lenguaje de bajo nivel, similar al lenguaje ensamblador, con un formato binario compacto que se ejecuta con rendimiento casi nativo y provee un objetivo de compilación para lenguajes como C/C++ y Rust que les permite correr en la web. También está diseñado para correr a la par de JavaScript, permitiendo que ambos trabajen juntos.
- WebAssembly tiene grandes implicaciones para la plataforma web, provee una forma de ejecutar código escrito en múltiples lenguajes en la web a una velocidad casi nativa, con aplicaciones cliente corriendo en la web que anteriormente no podrían haberlo hecho.
- WebAssembly esta diseñado para complementar y ejecutarse a la par de JavaScript, usando las APIs WebAssembly de JavaScript, se pueden cargar módulos de WebAssembly en una aplicación JavaScript y compartir funcionalidad entre ambos. Esto permite aprovechar el rendimiento y poder de WebAssembly con la expresividad y flexibilidad de JavaScript en las mismas aplicaciones.
- En diciembre de 2019, se publicaron la especificaciones WebAssembly Core Specification, WebAssembly Web API y WebAssembly JavaScript Interface como recomendaciones del W3C.

Desarrollo de Software basado en Componentes

- Los sistemas de hoy en día son cada vez más complejos, deben ser construidos en tiempo récord y deben cumplir con los estándares más altos de calidad.
- Para hacer frente a esto, se concibió y perfeccionó lo que hoy conocemos como Ingeniería de Software Basada en Componentes (ISBC), la cual se centra en el diseño y construcción de sistemas computacionales que utilizan componentes de software reutilizables.
- Esta ciencia trabaja bajo la filosofía de "comprar, no construir", una idea que ya es común en casi todas las industrias existentes, pero relativamente nueva en lo que a la construcción de software se refiere.
- En esencia, un componente es una pieza de código preelaborado que encapsula alguna funcionalidad expuesta a través de interfaces estándar.
- Los componentes son los "ingredientes de las aplicaciones", que se juntan y combinan para llevar a cabo una tarea.
- El paradigma de ensamblar componentes y escribir código para hacer que estos componentes funcionen se conoce como Desarrollo de Software Basado en Componentes.

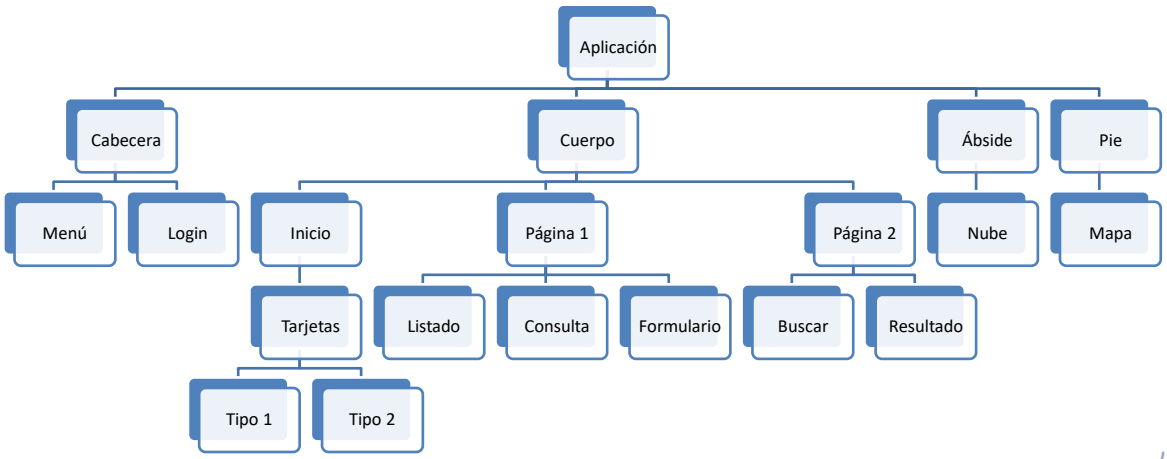
Ventajas del DSBC

- El uso de este paradigma posee algunas ventajas:
  - **Reutilización del software**. Nos lleva a alcanzar un mayor nivel de reutilización de software.
  - **Simplifica las pruebas**. Permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados.
  - **Simplifica el mantenimiento del sistema**. Cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.
  - **Mayor calidad**. Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.
- El optar por utilizar componentes de terceros en lugar de desarrollarlos, posee algunas ventajas:
  - **Ciclos de desarrollo más cortos**. La adición de una pieza dada de funcionalidad tomará días en lugar de meses o años.
  - **Mejor ROI**. Usando correctamente esta estrategia, el retorno sobre la inversión puede ser más favorable que desarrollando los componentes uno mismo.
  - **Funcionalidad mejorada**. Para usar un componente que contenga una pieza de funcionalidad, solo se necesita entender su naturaleza, más no sus detalles internos. Así, una funcionalidad que no sería práctica de implementar en la empresa, se vuelve ahora completamente asequible.

Componentes

- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente. Sigue un modelo de composición jerárquico con forma de árbol de componentes. La división sucesiva en componentes permite disminuir la complejidad funcional favoreciendo la reutilización y las pruebas.
- Los componentes establecen un cauce bien definido de entrada/salida para su comunicación con otros componentes.
- Los componentes se pueden clasificar en:
  - Componentes de aplicación: Son componentes íntimamente ligados a la aplicación cuya existencia viene determinada por la existencia de la aplicación.
    - Estos componentes deben favorecer el desacoplamiento máximo de entre la presentación, la estética y la lógica del componente.
  - Componentes de presentación (compartidos, widged, controles, …): Son componentes de bajo nivel, que existen para dar soporte a los componentes de aplicación pero son independientes de aplicaciones concretas, existen para ampliar la limitada oferta de presentación del HTML.
    - Estos componentes requieren un mayor control sobre la presentación y la estética por los que el acoplamiento será mayor, así como la dependencia entre ellos.
    - En muchos casos serán implementaciones de terceros que se incorporaran como módulos externos.

Árbol de componentes



Los componentes y la web

- Como desarrolladores, sabemos que reutilizar código tanto como sea posible es una buena idea. Esto tradicionalmente no es sencillo para estructuras de marcado personalizadas: piense en el complejo HTML (y sus estilos y scripts asociados) que en ocasiones se han tenido que escribir para renderizar controles de interfaz (UI) personalizados, y ahora usarlos múltiples veces puede crear un caos en la página si no se es cuidadoso.
- Los componentes Web (Web Components) apuntan a resolver dichos problemas.
- Los Web Components nos ofrecen un estándar que va enfocado a la creación de todo tipo de componentes utilizables en una página web, para realizar interfaces de usuario y elementos que nos permitan presentar información (o sea, son tecnologías que se desarrollan en el lado del cliente). Los propios desarrolladores serán los que puedan, en base a las herramientas que incluye Web Components crear esos nuevos elementos y publicarlos para que otras personas también los puedan usar.

Web Components

- Los Web Components son una reciente incorporación al HTML5 que, si siguen evolucionando al ritmo al que lo están haciendo, pueden suponer el mayor cambio en el mundo web en años y solucionar de golpe varios problemas históricos de HTML.
- El estándar, en proceso de definición, en realidad, se compone de 4 subelementos complementarios repartidos en los estándares principales (HTML, DOM y CSS), pero independientes entre si:
  - Custom Elements http://w3c.github.io/webcomponents/spec/custom/
  - Shadow DOM https://www.w3.org/TR/shadow-dom/
  - Templates Elements https://www.w3.org/TR/html5/scripting-1.html#the-template-element
  - ~~HTML Imports http://w3c.github.io/webcomponents/spec/imports/~~
- Cualquiera de ellos puede ser usado por separado, lo que hace que la tecnología de los Web Components sea, además de muy útil, muy flexible.

Web Components

- **Custom Elements**: Un conjunto de APIs de JavaScript que permiten definir elementos (etiquetas) personalizados y su comportamiento, que después pueden ser utilizado como se deseé en la interfaz del usuario (HTML).
- **Shadow DOM**: Un conjunto de APIs de JavaScript para fijar un árbol DOM "sombra" encapsulado a un elemento, que es renderizado por separado del documento DOM principal, y controlando funcionalidad asociada. De esta forma, se pueden mantener características de un elemento en privado, así puede tener el estilo y los scripts sin miedo de colisiones con otras partes del documento.
- **HTML Templates** (plantillas HTML): Los elementos <template> y <slot> permiten escribir plantillas de marcado que no son desplegadas en la página renderizada. Éstas pueden ser reutilizadas en múltiples ocasiones como base de la estructura de un elemento personalizado.
- **HTML Import** (descartada→módulos de ES2015+): ~~es la posibilidad de incluir un HTML dentro de otro mediante un tag <link rel="import" href="include.html">.~~

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

Módulos ES2015+

- La especificación de módulos ECMAScript define la inclusión y reutilización de documentos JS en otros documentos JS.
- Los módulos ECMAScript permiten que los componentes web se desarrollen de forma modular, alineados con otras implementaciones aceptadas en la industria para el desarrollo de aplicaciones JavaScript.
- Se puede definir la interfaz del elemento personalizado en un archivo .js que luego se incluye con un atributo type="module".
- Los archivos del módulo ECMAScript se fusionan en un solo archivo del lado del cliente o se pueden agrupar en paquetes individuales con anticipación.

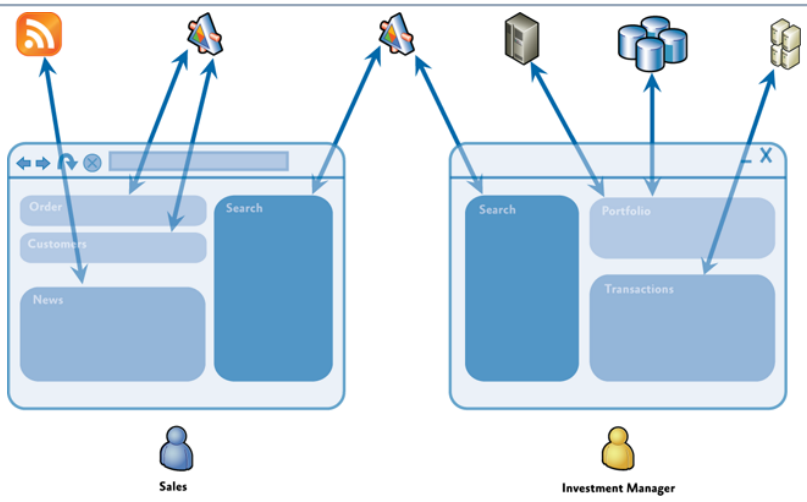
Objetivos

- **Ser fáciles de reutilizar**: A menudo, creamos elementos o partes que se repiten en nuestra web. Los WebComponents nos permiten reutilizar ciertas partes e incluirlas mediante una etiqueta HTML personalizada. Una forma cómoda y sencilla de reutilizar elementos web.
- **Encapsulación**: Necesitamos una forma de crear elementos aislados de otros. Por varias razones: evitar cambiar su contenido por error, evitar colisiones de CSS de elementos que se llaman igual, etc. Se trata de poder utilizar una característica muy popular y deseada en el mundo de la programación, que facilita la labor de los desarrolladores y hace más fácil la reutilización en aplicaciones muy grandes.
- **Interoperabilidad**: Facilitar la posibilidad de intercambiar información entre sistemas diferentes, utilizando tecnologías abiertas y estándares, que aseguran que van a funcionar en diferentes dispositivos y con las que no dependemos de tecnologías de empresas particulares y sus decisiones de negocio privadas.
- **Ser fáciles de mantener**: A medida que escribimos código en una web, la cantidad de líneas de código se hace mayor y cada vez es más complicado con un enfoque global. Necesitamos una forma de poder enfocarnos en una característica concreta, que sólo tenga HTML, CSS y Javascript que influya a dicha característica específica.

Aproximación básica

- Crear una clase (ES2015) o función en la cual especificar la funcionalidad del componente web.
- Utilizando el método CustomElementRegistry.define(), registrar el nuevo elemento personalizado (Custom Element), pasándole el nombre del elemento a ser definido, la clase o función en la cuál su funcionalidad esta especificada y, opcionalmente, de que elemento hereda.
- Si se requiere, adjuntar un Shadow DOM al elemento personalizado usando el método Element.attachShadow(). Añadir elementos hijos, oyentes de eventos, etc., al Shadow DOM utilizando métodos normales del DOM.
- Si se requiere, definir una plantilla HTML utilizando <template> y <slot>. Nuevamente usar métodos regulares del DOM para clonar la plantilla y acoplarla al Shadow DOM.
- Utilizar los componentes personalizados en la página web cuando se desee, como se utilizaría cualquier otro elemento HTML.

Patrón Composite View



Micro Frontends

- El término Micro Frontends apareció por primera vez en ThoughtWorks Technology Radar a finales de 2016. Extiende los conceptos de los micro servicios al mundo del frontend. La tendencia actual es crear una aplicación de navegador potente y rica en características, también conocida como “single page app”, que se asiente sobre una arquitectura de microservicio. Con el tiempo, la capa de frontend, a menudo desarrollada por un equipo independiente, crece y se vuelve más difícil de mantener. Eso es lo que llamamos una Interfaz Monolítica.
- La idea detrás de Micro Frontends es pensar en un sitio web o aplicación web como una composición de características que son propiedad de equipos independientes. Cada equipo tiene un área de negocio definida o misión de la que se preocupa y se especializa. Un equipo es cross functional y desarrolla sus características end-to-end, desde la base de datos hasta la interfaz de usuario.
- Los Custom Elements, el aspecto de interoperabilidad de las especificaciones de Web Components, son una buena primitiva para la integración en el navegador. Cada equipo construye sus componentes usando la tecnología web de su elección y lo envuelve dentro de un Custom Element. La especificación DOM de dichos componentes (nombre de etiqueta, atributos y eventos) actúa como el contrato o API pública para otros equipos. La ventaja es que pueden usar el componente y su funcionalidad sin tener que conocer la implementación. Solo tienen que ser capaces de interactuar con el DOM. El equipo de producto decide la composición de la aplicación, cómo ensamblar una página con componentes que pertenecen a diferentes equipos.

Vainilla (Vanilla JS) vs biblioteca

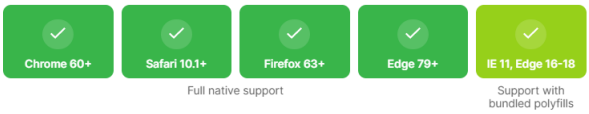
- Los componentes web pueden resultar difíciles de escribir desde cero. Hay mucho en qué pensar y escribir, un componente puede requerir una gran cantidad de código repetitivo. Afortunadamente, existen algunas bibliotecas excelentes que pueden hacer que la creación de elementos personalizados sea más sencilla y ahorrarle mucho tiempo y esfuerzo.
- Es importante tener en cuenta que no es necesario utilizar una biblioteca para crear y compartir un elemento personalizado. Los elementos personalizados sin procesar son excelentes si su tarea se limita a uno o unos pocos elementos, ya que permiten una implementación más optimizada y evitan el bloqueo de la biblioteca. Vanilla.JS es una broma creada por Eric Wastl que promueve la vuelta a las esencias.
- Sin embargo, si se está escribiendo muchísimos elementos personalizados, el uso de una biblioteca puede hacer que su código sea más simple y limpio, y que su flujo de trabajo sea más eficiente.
- Una buena biblioteca de componentes web debería producir un componente web que "simplemente funcionen" como cualquier otro elemento HTML. Las buenas bibliotecas también tienen una alta relación valor-carga útil, es decir, proporcionan mucho valor con el mínimo tamaño de su descarga.

Bibliotecas para construir componentes web

- Hybrids es una biblioteca de interfaz de usuario para crear componentes web con API simple y funcional.
- Polymer proporciona un conjunto de funciones para crear elementos personalizados.
- LitElement usa lit-html para representar en el Shadow DOM del elemento y agrega API para ayudar a administrar las propiedades y atributos del elemento.
- Slim.js es una biblioteca de componentes web liviana de código abierto que proporciona enlace de datos y capacidades extendidas para componentes, utilizando la herencia de clases nativa de es6.
- Stencil es un compilador de código abierto que genera componentes web que cumplen con los estándares.
- Los Angular Elements son componentes Angular empaquetados como Web Components.
- Implementaciones con diferentes frameworks
  - https://github.com/shprink/web-components-todo
  - https://github.com/thinktecture-labs/web-components-chat

Soporte en navegadores

- Los Custom Elements v1 se admiten de forma nativa en Chrome, Edge, Firefox y Safari (incluido iOS).
- Para navegadores sin soporte nativo, un pequeño polyfill permite usar elementos personalizados sin problemas y con poca sobrecarga de rendimiento.
- Stencil usa un cargador dinámico para cargar los polyfill solo en los navegadores que lo necesitan.



	Chrome 60+	Safari 10.1+	Firefox 63+	Edge 79+	Edge 16-18	IE 11
CSS Variables	✓	✓	✓	✓	✓	○
Custom Elements	✓	✓	✓	✓	○	○
Shadow Dom	✓	✓	✓	✓	○	○
es2017	✓	✓	✓	✓	✓	○
ES Modules	✓	✓	✓	✓	✓	○

https://caniuse.com/

Polyfills

- Algunos navegadores todavía están en proceso de actualización para admitir los estándares de componentes web. Mientras tanto, los polyfills simulan lo más fielmente posible las capacidades que faltan en el navegador.
- Puede detectar las funciones necesarias del navegador antes de cargar los polyfills. Esto significa que a medida que más y más navegadores implementen los estándares de componentes web, la carga útil para ejecutar sus aplicaciones y elementos disminuirá.
- Los polyfills están disponibles en GitHub:
  - https://github.com/WebComponents/webcomponentsjs

Repositorios

- WebComponents.org
- The Component Gallery
- generic-components
- web-components-examples
- awesome-standalones
- accessible_components
- Kickstand UI

HERRAMIENTAS DE DESARROLLO

IDEs

- Visual Studio Code - http://code.visualstudio.com/
  - VS Code is a Free, Lightweight Tool for Editing and Debugging Web Apps.
  - Visual Studio Code for the Web provides a free, zero-install Microsoft Visual Studio Code experience running entirely in your browser: https://vscode.dev
- StackBlitz - https://stackblitz.com
  - The online IDE for web applications. Powered by VS Code and GitHub.
- CodeSandbox - https://codesandbox.io/
  - Create, share, and get feedback with collaborative sandboxes.
- IntelliJ IDEA - https://www.jetbrains.com/idea/
  - Capable and Ergonomic Java * IDE
- Webstorm - https://www.jetbrains.com/webstorm/
  - Lightweight yet powerful IDE, perfectly equipped for complex client-side development and server-side development with Node.js

Instalación de utilidades

Consideraciones previas

- Las utilidades son de línea de comandos.
- Para ejecutar los comandos es necesario abrir la consola comandos (Símbolo del sistema)
- Siempre que se realice una instalación o creación es conveniente “Ejecutar como Administrador” para evitar otros problemas.
- En algunos casos el firewall de Windows, la configuración del proxy y las aplicaciones antivirus pueden dar problemas.

GIT: Software de control de versiones

- Descargar e instalar: https://git-scm.com/
- Verificar desde consola de comandos:
  - git

Node.js: Entorno en tiempo de ejecución

- Descargar e instalar: https://nodejs.org
- Verificar desde consola de comandos:
  - node --version

npm: Node Package Manager

- Aunque se instala con el Node es conveniente actualizarlo:
  - npm update -g npm
- Verificar desde consola de comandos:
  - npm --version
- Configuración:
  - npm config edit
  - proxy=http://usr:pwd@proxy.dominion.com:8080 ← Símbolos: %HEX ASCII
- Generar fichero de dependencias package.json:
  - npm init
- Instalación de paquetes:
  - npm install -g grunt-cli grunt-init karma karma-cli ← Global (CLI)
  - npm install jasmine-core tslint --save --save-dev
  - npm install ← Dependencias en package.json
- Arranque del servidor:
  - npm start

https://html.spec.whatwg.org/multipage/scripting.html#the-template-element

TEMPLATES ELEMENTS

Introducción

- El estándar de HTML 5 incluyo los elementos <template> y <slot> para crear una plantilla flexible que luego puede ser usada en JavaScript con el API DOM o para rellenar el Shadow DOM de un componente web.
- Cuando tienes que reutilizar las mismas estructuras de lenguaje de marcado repetidas veces en una página web, tiene sentido utilizar algún tipo de plantilla en lugar de repetir la misma estructura una y otra vez.
- La plantilla es un fragmento HTML de contenido que está siendo almacenado para un uso posterior en el documento. El analizador procesa el contenido del elemento <template> durante la carga de la página, pero sólo lo hace para asegurar que esos contenidos son válidos; sin embargo, estos contenidos del elemento no se renderizan ni se procesan por lo que no carga sus recursos, como scripts o imágenes, hasta que se use la primera vez.
- Esto tiene varios beneficios:
  - Se puede modificar el código HTML sin tener que reescribir cadenas dentro de su JavaScript.
  - Los componentes se pueden personalizar sin tener que crear clases de JavaScript independientes para cada tipo de presentación.
  - Es más fácil definir HTML en HTML y se puede modificar en el servidor o cliente antes de que se procese el componente.

<template>

- El elemento HTML <template> es un mecanismo para mantener el contenido HTML del lado del cliente que no se renderiza cuando se carga una página, pero que posteriormente puede ser instanciado durante el tiempo de ejecución empleando JavaScript. El contenido de la plantilla no tiene requisitos de conformidad. Normalmente, si intentamos por ejemplo poner <tr> dentro de un , el navegador detecta la estructura DOM como inválida y lo soluciona añadiendo un <table> alrededor, cosa que no hace en las plantillas.

```
<template id="tmpl-saludo">
  <h1>Hola <span>mundo</span></h1>
</template>
```

- El contenido de la plantilla no aparecerá en la página hasta que se haga una referencia a elle con JavaScript y se agregue al DOM:

```
let template = document.getElementById('tmpl-saludo');
document.body.appendChild(template.content);
```

Clonación

- Cada elemento template tiene un objeto DocumentFragment asociado que es el contenido de la plantilla. Dado que un nodo DOM solo puede tener un nodo padre, el DocumentFragment solo se puede adjuntar una vez la árbol DOM, se vacía al moverse sus nodos al nuevo padre. Si se utiliza varias veces, modificando o no su contenido, o dentro de Web Components se debe realizar una copia (clonar) y trabajar sobre la copia.
- El método .content.cloneNode(deep) sobre un template realiza una copia. En el caso de establecer deep a true, realizara una copia profunda (deep clone), es decir, se clona el elemento y todos sus elementos hijos. En caso de establecerlo a false, se hará una copia superficial (shallow clone), es decir, clonará sólo el elemento indicado y no sus hijos, los cuales serán una referencia a los originales.

```
let template = document.getElementById('tmpl-saludo');
let copia = template.content.cloneNode(true);
copia.querySelector('span').textContent = 'MUNDO';
document.body.appendChild(copia);
```

- La clase del componente web puede acceder a plantillas, obtener su contenido y clonar los elementos para asegurarse de que está creando un fragmento DOM único en todos los lugares donde se usa.

Procesar plantilla

```
<table>
  <thead>
   <tr><th>Name <th>Color <th>Sex <th>Legs
  <tbody>
   <template id="row"><tr><td><td><td><td></template>
 </table>
 <script>
  let data = [
    { name: 'Pillar', color: 'Ticked Tabby', sex: 'Female (neutered)', legs: 3 },
    { name: 'Hedral', color: 'Tuxedo', sex: 'Male (neutered)', legs: 4 },
  ];
  let template = document.querySelector('#row');
  for (let i = 0; i < data.length; i += 1) {
    let cat = data[i];
    let clone = template.content.cloneNode(true);
    let cells = clone.querySelectorAll('td');
    cells[0].textContent = cat.name;
    cells[1].textContent = cat.color;
    cells[2].textContent = cat.sex;
    cells[3].textContent = cat.legs;
    template.parentNode.appendChild(clone);
  }
 </script>
```

Plantillas dinámicas

- Como cualquier otro elemento HTML, los elementos template se pueden crear por código con JavaScript y el API DOM. Esto evita depender de una página HTML para utilizar las plantilla, especialmente interesante en el caso de los Web Component.

```
let nombre = 'mundo'
let template = document.createElement('template');
template.innerHTML = /* html */`
   <h1>Hola ${nombre}</h1>
`;
let copia = template.content.cloneNode(true);
// copia.querySelector('span').textContent = 'MUNDO';
document.body.appendChild(copia);
```

- En los Web Component permite separar la presentación (plantilla) de la lógica del componente (clase) y, en un único fichero, incluir todo el componente sin dependencias externas.

Estilos

- Podemos incluir cierta información de estilo dentro de la plantilla con un elemento <style>, que luego se encapsula dentro del elemento personalizado (Shadow DOM) permitiendo separar la estética de la presentación. Esto no funciona si se agrega directamente el contenido de la plantilla al DOM estándar, el estilo se filtraría al resto de la página y afectaría a todos los elementos que cumplan los selectores.

```
<template id="tmpl-saludo">
   <style>
      span {
         font-style: italic;
      }
   </style>
   <h1>Hola <span>mundo</span></h1>
</template>
<p>Hola <span>mundo</span></p>
```

Slots

- Los componentes son etiquetas que pueden tener contenido, por lo que a menudo se necesitan representar elementos secundarios dinámicos en ubicaciones específicas en su árbol de componentes, lo que permite que un desarrollador proporcione contenido secundario cuando usa nuestro componente, con nuestro componente colocando ese componente secundario en la ubicación adecuada. Para hacer esto, se usa la etiqueta Slot dentro de la plantilla.
- El elemento HTML <slot> es un placeholder (punto de inserción) en un componente donde se proyecta el contenido de la etiqueta (innerHTML) e indica donde mostrar el contenido recibido, permite crear árboles DOM por separado y presentarlos juntos sin tener que analizar el contenido.

```
<template id="tmpl-my-card">
  <div>
    <h2>(sin título)</h2>
    <slot></slot>
  </div>
</template>
<my-card title="Saludo"><p>Hola mundo</p></my-card>
```

- Se puede definir la etiqueta <slot> con el contenido alternativo a usar si no recibe contenido:

```
<slot>(sin contenido)</slot>
```


Múltiples Slots

- En algunos casos el contenido se debe repartir en varias ubicaciones dentro del componente. El componente puede usar mas de una etiqueta Slot si cuentan con un identificador único en el atributo name. Tener varios slot se denomina composición y el resultado flattened DOM (aplanado).

```
<template id="tmpl-my-card">
  <div>
    <slot name="title" />
    <slot>(sin contenido)</slot>
    <slot name="pie" />
  </div>
</template>
```

- En el contenido, a través de un atributo slot, indica el name del <slot> de destino:
  - solo pueden tener el atributo slot los nodos hijo del contenedor, en otros descendientes se ignora.
  - si hay varios nodos con el mismo nombre de slot, se añaden al slot, uno tras otro.

```
<my-card><h1 slot="title">Saludo</h1><p slot="pie">Adiós</p><p>Hola mundo</p></my-card>
```

- Opcionalmente, se puede contar con un único slot sin name, slot predeterminado, para todos los nodos que no están ubicados en otro slot, sin el slot predeterminado dicho contenido se pierde.

https://dom.spec.whatwg.org/#shadow-trees

SHADOW DOM

Introducción

- Shadow DOM es una API integrada en el navegador que permite la encapsulación de DOM y la encapsulación de estilo. La encapsulación es un aspecto central de los estándares de componentes web. El Shadow DOM protege los estilos, el marcado y el comportamiento de un componente del entorno que lo rodea. Esto significa que no tenemos que preocuparnos por aplicar nuestro CSS a nuestro componente, ni preocuparnos por que el DOM interno de un componente sea interferido por algo fuera del componente.
- Shadow DOM permite adjuntar arboles DOM en la sombra (shadow tree), ocultos a elementos en el árbol DOM regular (light tree). Este árbol Shadow DOM, el shadow hosts, comienza con un elemento shadowRoot (uno por componente), debajo del cual se puede adjuntar cualquier elemento que desee, de la misma manera que el DOM normal.
- El Shadow DOM oculta y separa el DOM de un componente para evitar conflictos de estilos o efectos secundarios no deseados.

Objetivos

- Shadow DOM está diseñado como una herramienta para crear aplicaciones basadas en componentes, aunque se puede usar la API de Shadow DOM y sus beneficios fuera de los componentes web.
- Su objetivo es brindar soluciones a problemas comunes del desarrollo web:
  - DOM aislado: El DOM de un componente es independiente (p.ej., document.querySelector() no mostrará nodos en el Shadow DOM del componente).
  - Composición: Diseña una API declarativa basada en lenguaje de marcado para tu componente.
  - CSS con alcance: el CSS definido dentro del Shadow DOM está dentro del alcance. Las reglas de diseño no se filtran y los diseños de página no se filtran.
  - CSS simplificado: DOM con alcance significa que puedes usar selectores CSS simples, nombres de ID o clases más genéricos, y no preocuparte por conflictos de nombres.
  - Productividad: Piensa en las aplicaciones como fragmentos del DOM en lugar de una página grande (global).

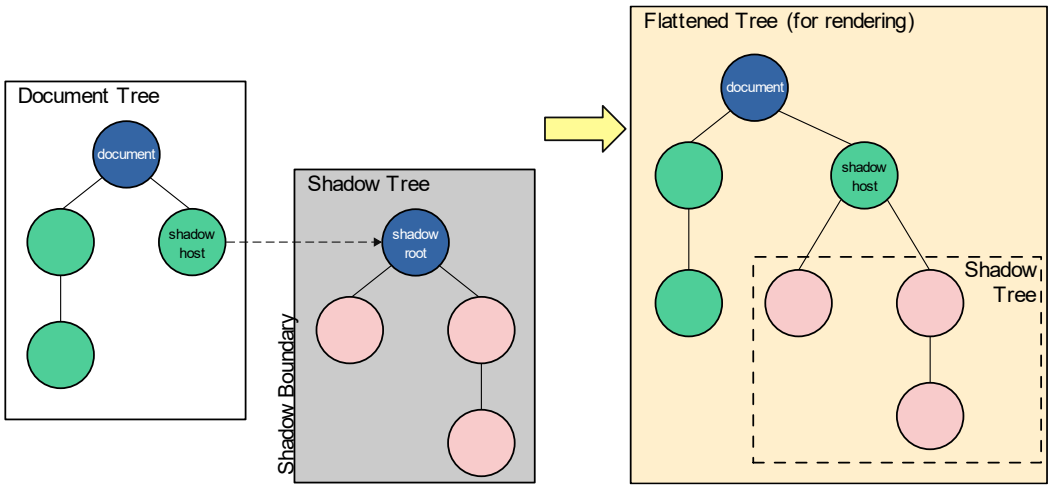
Árbol de nodos

- Antes de poder utilizar el DOM, el navegador transforma internamente el archivo XML o HTML original en una estructura más fácil de manejar formada por una jerarquía de nodos.
- Los nodos son objetos que implementan el interfaz Node o uno de sus herederos (tipos). Los nodos participan en un árbol, que se conoce como árbol de nodos .
- El árbol generado no sólo representa los contenidos del archivo original (mediante los nodos del árbol) sino que también representa sus relaciones (mediante las ramas del árbol que conectan los nodos): antecesores-padre-hijos-hermanos-descendientes.
- El Shadow DOM amplia el estándar del DOM con nuevos conceptos para permitir la encapsulación.
- Los conceptos principales de Shadow DOM:
  - Light tree: El árbol DOM al que pertenece el Shadow DOM.
  - Shadow host: El nodo regular del DOM al que esta vinculado el Shadow DOM.
  - Shadow root: El nodo raíz de un árbol Shadow.
  - Shadow tree: El árbol DOM dentro del Shadow DOM.
  - Shadow boundary: El punto en el que termina el Shadow DOM y comienza el DOM a la luz.

Árbol de nodos

- Un árbol de documento es un árbol de nodos cuya raíz es un documento. El árbol del documento es único y está a la luz (se ve y se puede acceder) por lo que se denomina light tree o DOM normal.
- Un árbol a la sombra es un árbol de nodos cuya raíz es un elemento shadowRoot (una raíz en la sombra). Una raíz oculta siempre está adjunta a otro árbol de nodos a través de su host, el shadow host. Por lo tanto, un árbol a la sombra nunca está solo. El árbol de nodos del anfitrión del shadow hosts a veces se denomina árbol a la luz. El árbol a la luz correspondiente a un árbol de sombras puede ser el propio árbol de documento o, a su ver, otro árbol a la sombra.
- El árbol del documento es único y puede contener múltiples árboles a la sombra. Un árbol a la sombra puede contener a su vez múltiples árboles a la sombra, en cuyo caso asume el rol de árbol a la luz de dichos árboles a la sombra.
- El shadow boundary establece los limites hasta donde se puede llegar desde la luz accediendo a nodos y aplicando estilos, delimitando las búsquedas desde la luz hacia las sombras, evitando los choques de identificadores o impidiendo que el estilos definido dentro de un 'Shadow DOM' puede afectar a nada fuera de él.

Árbol de nodos



Habilitar el Shadow DOM

- La especificación define una lista de elementos que pueden alojar un shadow tree, pueden ser shadow host: custom element, article, aside, blockquote, body, div, footer, h1, h2, h3, h4, h5, h6, header, main, nav, p, section y span.
- Existen varios motivos por los que un elemento puede no estar en la lista: el navegador ya aloja su propio Shadow DOM interno para el elemento (<textarea>, <input>) o no tiene sentido que el elemento aloje un Shadow DOM (<img>).
- Para crear y habilitar el Shadow DOM de un elemento se utiliza el método attachShadow.

```
const shadow = div.attachShadow({ mode: 'open' });
```

- El Shadow DOM tiene dos modos de encapsulación:
  - "open": En modo abierto (open), el elemento con Shadow DOM tendrá una propiedad .shadowRoot por la cuál se puede acceder al shadowRoot del elemento, y a partir de ahí, al árbol DOM que contiene. Todo código puede acceder el árbol shadow del elemento (indirectamente).
  - "closed": En modo cerrado (closed), la propiedad .shadowRoot devolverá null, siendo imposible acceder al árbol DOM desde fuera, solo internamente si se guarda la referencia devuelta por attachShadow. (debe evitarse)

Encapsular JavaScript

- La raíz shadow root, devuelta por attachShadow, es un elemento: se trata como un árbol DOM normal, usando .innerHTML o métodos DOM tales como .appendChild para llenarlo.
- Los elementos Shadow DOM no son visibles para los .getElementBy o .querySelector desde el DOM visible (light tree). En particular, los elementos Shadow DOM pueden tener ids en conflicto con ids en el DOM visible, deben ser únicos solamente dentro del árbol shadow. En caso de conflicto la búsqueda se puede realizar a través del shadowRoot o del document.

```
<template id="tmpl-saludo">
  <h1>Hola <span id="saludo-name">mundo</span></h1>
</template>
<div id="estilo"></div>
let div = document.getElementById('estilo')
const shadow = div.attachShadow({ mode: "open" });
div.shadowRoot.appendChild(document.getElementById('tmpl-saludo').content.cloneNode(true))
console.log(document.getElementById('saludo-name') ? 'Encontrado': 'NO Encontrado')
console.log(div.shadowRoot.getElementById('saludo-name') ? 'Encontrado': 'NO Encontrado')
```

Encapsular CSS

- El Shadow DOM tiene stylesheets propios. Las reglas de estilo del DOM exterior no se le aplican. Si un estilo en el DOM normal usa un selector que coincide con un elemento en el shadowRoot, los estilos del DOM normal no se aplicarán al shadowRoot dado que los selectores CSS externos no pueden localizar elementos internos del shadowRoot para aplicar los estilos.
- El Shadow DOM puede incluir las etiquetas <style> y <link rel="stylesheet" href="…"> (las hojas se almacenan en la caché y no se vuelven a descargar) para definir el estilo local.
- Como resultado, los selectores de CSS dentro del shadowRoot se pueden simplificar, ya que solo se aplicarán a los elementos dentro del shadowRoot y no tienen que tener en cuenta nada mas. El shadow boundary establece un limite para que CSS definido dentro del shadowRoot no salga del shadow host (se filtre al DOM normal). El estilo se define con los selectores normales, aunque hay selectores específicos para el interior del Shadow DOM.
- Los elementos en los slot provienen del light tree, por lo que usan los estilos del documento o de su ámbito. Los estilos locales no afectan a los elementos contenidos en los slot.
- Las propiedades personalizadas CSS atraviesan el Shadow DOM, son visibles en todas partes.

Selectores especiales

- :host o :host(), pseudoclases que permiten seleccionar el shadow host (el elemento que contiene el shadow tree). No se puede llegar al elemento host desde el interior del Shadow DOM con otros selectores porque no forma parte del shadow tree dado que está en el light tree anfitrión.

```
:host { border: 1px solid black; }
:host(.active), :host(:hover), :host([disabled]) { border-width: 3px; }
```

- :host-context(), selector que busca una condición CSS en cualquier antecesor del elemento host del componente, hasta la raíz del documento, solo es útil cuando se combina con otro selector para aplicar estilos basados en alguna condición externa al componente (si está contenido dentro de un elemento con determinadas características).

```
:host-context(.theme-light) h2 { background-color: #eef; }
```

- ::slotted() pseudoelemento que permite dar estilo a etiquetas insertadas en un slot, pero desde dentro del shadow host. En el interior de los paréntesis se indica el selector CSS que los elementos dentro del slot deben cumplir. El estilo definido tiene preferencia sobre el estilo global (salvo si se ha priorizado con !important).

```
::slotted(h2) { color: green; }
#tabsSlot::slotted(*) { padding: 16px 8px; }
```

Propiedades personalizadas de CSS

- En CSS, las propiedades personalizadas (conocidas como Variables CSS) son entidades definidas por autores de CSS que contienen valores específicos que se pueden volver a utilizar en un documento.
- Se declaran en las reglas CSS con un nombre personalizado, sensible a la tipografía y prefijados por --.

```
:root {
    --error-color: red;
}
```

- Pueden tener un alcance global o local. El alcance es global si se declara dentro del pseudo selector :root. El alcance local permite su uso en el selector donde se declara o en cualquier selector anidado, primando las definiciones mas locales.
- La función var() permite recuperar el valor de la variable para definir el valor de otra propiedad. Opcionalmente se puede definir un valor por defecto para cuando no exista la variable.

```
.error-msg {
    color: var(--error-color, red);
    margin-top: calc(var(--gap) * 1px);
}
```

- Las variables CSS están en el DOM, por lo que se pueden cambiar con JavaScript.

Personalización de componentes con propiedades personalizadas

- Las propiedades personalizadas de CSS pueden permitir a los consumidores de un componente personalizar los estilos de un componente desde el DOM normal.
- Solo las propiedades personalizadas de alcance global, las declaradas dentro del pseudo selector :root, están expuestas a la aplicación consumidora.
- Si el componente define en su CSS:

```
:host {
  --negativo-back-color:  pink;
}
.negativo {
  background-color: var(--negativo-back-color);
}
```

- El consumidor puede definir en su hoja de estilos:

```
my-contador {
  --negativo-back-color: rgb(255, 69, 100);
}
document.querySelector('my-contador').style.setProperty("--negativo-back-color", "red");
```

CSS Shadow Parts

- La especificación CSS Shadow Parts define el pseudo-elemento ::part() y los atributos de etiqueta part y exportparts para los shadow hosts, lo que permite que los shadow hosts expongan selectivamente elementos elegidos de su árbol oculto a la página exterior con el fin de poder aplicarles estilo.
- Cualquier elemento en un árbol oculto puede tener un atributo part, que se usa para exponer el elemento fuera del árbol de oculto. El atributo part contiene una lista separada por espacios con los nombres de la parte del elemento. Es mejor dar varios nombres a una parte, debe considerarse similar a una clase CSS, no una identificación o nombre de etiqueta.
- Cualquier elemento en un shadow tree puede tener un atributo exportparts que, si el elemento es un shadow hosts, permite exponer partes de elementos de su árbol oculto anidado a las que se aplicaran las reglas CSS definidas fuera del shadow hosts (como si fueran elementos del mismo árbol).
- El pseudo-elemento ::part() permite seleccionar elementos a través del los nombres de partes, definidos con el atributo part, siempre que estén en el mismo árbol o hayan sido exportado con un atributo exportparts.

Personalización de componentes con Partes CSS

- Para definir partes del componente:

```
<output part="pantalla">{this.counter}</output>
```

- Para exponer sus partes:

```
<my-contador exportparts="pantalla"></my-contador>
```

- Para definir el estilo de una de las partes:

```
my-contador::part(pantalla) {
  font-size: 2em;
}
```

- Las propiedades personalizadas de CSS permiten personalizar valores individuales, por lo que tienen un alcance limitado. Para situaciones en las que el consumidor requiere una personalización mas compleja, las partes CSS ofrecen un mayor grado de flexibilidad en el estilo.

Shadow root closed

- Las shadow root cerradas no son muy útiles y deben evitarse. Algunos desarrolladores verán el modo cerrado como una función de seguridad artificial pero seamos NO es una función de seguridad. El modo cerrado simplemente evita que el JavaScript externo explore el DOM interno de un elemento.
- El modo cerrado solo oculta:
  - Element.shadowRoot devuelve null
  - Element.assignedSlot / TextNode.assignedSlot devuelve null
  - Event.composedPath() devuelve el array de nodos recorrido por el evento incompleto, no incluye los nodos de los shadow DOM cerrados
- Las razones por las que deben evitarse son:
  - El propio código del elemento personalizado no puede acceder a su propio shadow DOM salvo que haya cacheado su referencia.
  - Hace que el componente sea menos flexible para los usuarios finales al no tener una vía alternativa de personalización.
  - Dan una sensación artificial de seguridad, hay otros mecanismos para obtener la referencia.

https://html.spec.whatwg.org/multipage/custom-elements.html

CUSTOM ELEMENTS

Introducción

- Los Custom Elements proporcionan una manera para que los autores creen sus propios elementos DOM con todas las funciones. Aunque los autores siempre pueden usar elementos no estándar en sus documentos, con un comportamiento específico de la aplicación agregado después la funcionalidad mediante secuencias de comandos o similares, dichos elementos históricamente no han sido conformes y no han sido muy funcionales. Al definir un Custom Elements, los autores pueden informar al analizador cómo construir correctamente un elemento y cómo deben reaccionar los elementos de esa clase a los cambios.
- Los Custom Elements son parte de un esfuerzo mayor para "racionalizar la plataforma", al explicar las características existentes de la plataforma (como los elementos de HTML) en términos de puntos de extensibilidad expuestos por el autor de nivel inferior (como la definición de elementos personalizados).
- Aunque hoy en día existen muchas limitaciones en las capacidades de los Custom Elements (tanto funcional como semánticamente) que les impiden explicar completamente el comportamiento de los elementos existentes de HTML, se espera reducir esta brecha con el tiempo.

Tipos de elementos personalizados

- **Elementos personalizados autónomos**: heredan de la clase abstracta HTMLElement (base de elementos concretos de HTML) o extienden otros elementos personalizados autónomos. Estos elementos se usan en una página escribiéndolos literalmente como un elemento HTML nuevo. Por ejemplo <app-element>, o document.createElement('app-element').
- **Elementos personalizados preconstruidos**: heredan de elementos HTML concretos para extenderlos. Para crear un elemento de este tipo, hay que especificar qué elemento extiende y se usan escribiendo el nombre de la etiqueta de elemento que extiende, pero añadiendo un atributo (o propiedad) **is** cuyo valor es el nombre del elemento personalizado que se ha desarrollado. Por ejemplo <p is="word-count">, o document.createElement('p', { is: 'word-count' }).

CustomElementRegistry

- El controlador de los elementos personalizados de un documento web es el objeto CustomElementRegistry, este objeto permite registrar un elemento personalizado en la página, devolver información de qué elementos personalizados se han registrado, etc.
- Para registrar un elemento personalizado en la página, debes usar el método CustomElementRegistry.define(). Éste toma los siguientes argumentos:
  - Un DOMString o tagName que representa el nombre que estás dando al elemento. Los nombres de los elementos personalizados deben contener un guión (kebab-case) y no pueden ser palabras simples.
  - Un objeto class de JavaScript que extiende un tipo de elemento HTML y define el comportamiento del nuevo elemento.
  - En los elementos personalizados preconstruidos, un objeto de opciones con la propiedad extends que especifica el elemento preconstruido que extiende.

```
customElements.define('word-count', WordCount, { extends: 'p' });
```


Nomenclatura

- El estándar de HTML5 define que las etiquetas HTML oficiales deben estar formadas por una sola palabra, mientras que los custom elements (nuestras propias etiquetas HTML) deben estar formadas por al menos 2 palabras, separadas por un guion. De esta forma, protegemos nuestras páginas o aplicaciones web para que, si en el futuro se añade una nueva etiqueta HTML estándar, no coincida con el mismo nombre que alguna nuestra.
- El prefijo en la etiqueta tiene un papel importante cuando se crea una colección de componentes destinados a ser utilizados en diferentes proyectos. Los componentes web no tienen alcance porque se declaran globalmente dentro de la página web, lo que significa que se necesita un prefijo "único" para evitar colisiones, que también ayuda a identificar rápidamente la colección de la que forma parte. Además, la especificación requiere que contengan un guion "-" dentro del nombre de la etiqueta (kebab-case), usar la primera sección como espacio de nombres para distinguir sus componentes es una buena idea (habitualmente app- para los componentes de aplicación).
- Los componentes son conceptualmente cosas, no son acciones, deben usar sustantivos en lugar de verbos en el nombre de etiqueta. Cuando varios componentes están relacionados y/o acoplados, es una buena idea compartir el nombre y luego agregar diferentes modificadores.
- Los nombres de la clase del componente utilizan PascalCase y no necesitan tener un prefijo ya que las clases tienen alcance y no hay riesgo de colisión, aunque suelen coincidir con el de la etiqueta.

Crear un elementos personalizado autónomo

- El componente básico mínimo necesario para hacerlo funcionar como elemento personalizado autónomo esta formado por una clase, que debe extender de HTMLElement, y definir el nombre de etiqueta HTML por medio de customElements.define(), que asocia el nombre de la etiqueta HTML a la clase y lo deja registrado en el CustomElementRegistry del documento.

```
class MyElement extends HTMLElement {
  constructor() {
    super();
  }
}
customElements.define('my-element', MyElement);
```

- Como buena práctica, se crean en un fichero denominado igual que la etiqueta, my-element.js, en una carpeta especifica como /components, aunque esto depende de la arquitectura utilizada. Los ficheros son la unidad de carga, ámbito y reutilización en los módulos ES2015+, pueden contener varios componentes pero con matices.

Cargar componentes

- Una vez tengamos un WebComponent creado, hay que cargar el archivo Javascript del componente en la página HTML, usualmente desde el <head> del documento o al final del <body> (en cuyo caso el navegador realiza una "Actualización de componentes"):

```
<script src="./components/my-element.js"></script>
```

- Si tenemos un gran número de componentes podemos centralizar varos import en un fichero Javascript, que pueda ser reutilizado, en lugar de tener varias etiquetas <script> en varios archivos HTML diferentes. Es obligatorio utilizar el atributo type establecido a module.

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

- En el fichero index.js tendremos que hacer las importaciones necesarias para cargar el componente y poder utilizarlo posteriormente:

```
import './components/my-element.js';
```

- Hay una forma híbrida entre las dos anteriores, podemos hacer las importaciones en un script de tipo module:

```
<script type="module">
  import './components/my-element.js';
</script>
```

Actualización de componentes

- Los elementos personalizados pueden usarse antes de registrar su definición: la mejora progresiva es una característica de los elementos personalizados que permite utilizar las etiqueta de los Custom Elements en la página e invocar a customElements.define() mucho más tarde.
- Si el navegador encuentra alguna etiqueta de un Custom Element antes de customElements.define, no es un error, el elemento todavía es desconocido como cualquier etiqueta no estándar.
- Dicho elemento queda como “undefined” y puede recibir estilo con el selector CSS :not(:defined). Cuando customElement.define es invocado, estos elementos “undefined” son “actualizados”: para cada elemento, se crea una nueva instancia del Web Component, se invoca a connectedCallback y, cuando se completan, pasan a :defined.
- Un Custom Element (con - en el nombre) siempre tiene una instancia HTMLElement aunque quede sin definir (no cuenta con funcionalidad o presentación adicional), frente a los elementos que la especificación no define (sin - en el nombre) que se analizan como HTMLUnknownElement.

Errores de Registro y Carga

- El elemento customElements es una referencia al registro de Custom Elements del navegador, el registro (CustomElementRegistry) donde se almacenan los tipos de etiquetas HTML personalizadas para que el navegador las reconozca.
- El método .get(tagName) obtiene síncronamente la clase de un custom element completamente definido o el valor undefined (no esta definido o no se ha completado la definición).

```
customElements.get('my-element') === MyElement
```

- Un Custom Elements solo puede estar registrado una vez, si se define varias veces emitirá una DOMException de *Nombre de Custom Element ya usado* o de *Clase ya usada*, para evitarlos:

```
if(!customElements.get('my-element')) customElements.define('my-element', MyElement);
```

- Se puede importar su módulo (fichero) varias veces sin problema porque solo se carga la primera vez.
- El método .whenDefined(tagName) obtiene una promesa que se resuelve cuando el custom element este definido que evita precipitarse al intentar realizar una acción si aun no esta completamente definido, dado que inicialmente es una instancia de HTMLElement que no dispone de las propiedades y métodos definido en el.

```
customElements.whenDefined('my-element').then((data) => {
  console.log('MyElement ha sido definido');
});
```

Utilizar un componente

- Una vez tengamos nuestro componente cargado y enlazado, existen varias formas de utilizarlo en las páginas.
- Los componentes son etiquetas HTML (personalizadas) y pueden ser utilizadas como cualquier otra en el HTML. Los elementos personalizados no pueden cerrarse automáticamente porque HTML solo permite que algunos elementos se cierren automáticamente, debe escribirse siempre la etiqueta de cierre:

```
<my-element></my-element>
```

- Los componentes son elementos DOM (personalizados) y pueden ser creados desde Javascript:

```
const myElement = document.createElement('my-element');
document.body.appendChild(myElement);
```

- Existe una variación de la anterior, instanciar directamente la clase, en cuyo caso no requiere define:

```
import { MyElement } from './components/my-element.js';
document.body.appendChild(new MyElement());
```

- Para hacerlo de esta forma tendríamos que exportar la clase, para así poder importarla desde otro módulo:

```
export class MyElement extends HTMLElement {
```


Definición del componente

- Una definición de elemento personalizado describe un elemento personalizado mediante CustomElementRegistry.define() y una clase.
- CustomElementRegistry.define() asocia el nombre a la clase y, opcionalmente, las opciones para los elementos personalizados preconstruidos.
- La clase define:
  - un constructor.
  - una lista de atributos observados.
  - una colección de devoluciones de llamadas del ciclo de vida.
  - una colección de métodos (permiten la interacción), propiedades y eventos (notifican cambios).
- Los atributos, métodos, propiedades y eventos públicos deberían contar con su documentación JSDocs o similar dado que forman parte del interfaz externo del Web Component y facilitan su reutilización.

Ciclo de vida

- Durante la carga de una página y durante su uso, los WebComponent pasan por una serie de fases que conocida como el ciclo de vida del WebComponent:
  - Fase 1: Componente no creado, ni insertado en el DOM (Documento HTML)
  - Fase 2: Componente creado, pero no insertado en el DOM (Etiqueta HTML)
  - Fase 3: Componente creado e insertado en el DOM (Documento HTML)
- Una serie de métodos especiales (devoluciones de llamadas del ciclo de vida), con nombre predefinido (hooks) e implementados en la clase, permiten personalizar el comportamiento de las diferentes fases:
  - constructor(): Cuando se crea el custom element, directamente con new o en el registro.
  - connectedCallback(): Cuando el custom element se ha insertado en el documento HTML (DOM).
  - disconnectedCallback(): Cuando el custom element se ha eliminado del documento HTML (DOM).
  - adoptedCallback(): Cuando el custom element se ha movido a un nuevo documento (común en iframes).
  - attributeChangedCallback(): Cuando se ha modificado un atributo observado del componente.

Ciclo de vida

```
class MyElement extends HTMLElement {
  constructor() {
    super();
    // elemento creado
  }
  connectedCallback() {
    // es llamado cuando el elemento es agregado al documento (puede ser llamado varias veces si un elemento es agregado y quitado repetidamente)
  }
  disconnectedCallback() {
    // es llamado cuando el elemento es quitado del documento (puede ser llamado varias veces si un elemento es agregado y quitado repetidamente)
  }
  adoptedCallback() {
    // es llamado cuando el elemento es movido a un nuevo documento (ocurre en document.adoptNode, muy raramente usado)
  }
  static get observedAttributes() {
    return [/* array de nombres de atributos a los que queremos monitorear por cambios */];
  }
  attributeChangedCallback(name, oldValue, newValue) {
    // es llamado cuando uno de los atributos listados arriba es modificado fuera del componente
  }

  // puede haber otros métodos y propiedades de elemento
}
```

Constructor

- El método constructor() en un WebComponent tiene la misma función que en una clase de programación. Se ejecutará cada vez que se cree un custom element particular, y que previamente haya sido definido en el registro global del navegador con customElements.define(). Si creamos varias etiquetas de un componente (instancias), se ejecutará una vez por cada una de ellas.
- El objetivo del constructor es hacer las tareas de inicialización rápidas o tareas iniciales, como la definición e inicialización de atributos, es esencial que en el constructor del componente se hagan sólo las tareas más prioritarias y ligeras.
- En general, el trabajo debe posponerse a connectedCallback tanto como sea posible, especialmente el trabajo que implica buscar recursos o renderizar.
  - Cuando el constructor es llamado, es aún demasiado pronto, cuando el elemento es creado el navegador aún no procesó ni asignó atributos en este estado y las llamadas a getAttribute devolverían null. Mejora el rendimiento demorar el trabajo hasta que realmente se lo necesite.
- Sin embargo, hay que tener en cuenta que se puede llamar a connectedCallback más de una vez, por lo que cualquier trabajo de inicialización que sea realmente único necesitará una protección para evitar que se ejecute dos veces.

Constructor

- Los constructores de elementos personalizados tiene los siguientes requisitos de conformidad:
  - Una llamada a super() sin parámetros debe ser la primera declaración en el cuerpo del constructor, para establecer la cadena de prototipos correcta y este valor antes de ejecutar más código.
  - Una declaración return no debe aparecer en ningún lugar dentro del cuerpo del constructor, a menos que sea un simple retorno anticipado (return o return this).
  - El constructor no debe utilizar los métodos document.write() o document.open().
  - Los atributos y elementos secundarios del elemento no deben inspeccionarse, ya que en el caso de que no se actualice ninguno estará presente y depender de las actualizaciones hace que el elemento sea menos utilizable.
  - El elemento no debe adquirir ningún atributo o elemento secundario, ya que esto viola las expectativas de los consumidores que utilizan los métodos createElement o createElementNS.
  - En general, el constructor debe usarse para configurar el estado inicial y los valores predeterminados, y para configurar detectores de eventos y posiblemente una raíz oculta .
  - Al crear reacciones de elementos personalizados, se debe evitar manipular el árbol de nodos, ya que esto puede generar resultados inesperados.
- Varios de estos requisitos se verifican durante la creación del elemento, ya sea directa o indirectamente, y si no se siguen, se generará un elemento personalizado que el analizador o las API DOM no podrán crear instancias.

Atributos observados

- Los atributos de los Custom Elements son la entrada del componente que se utilizarán para pasar información desde el exterior al propio componente, permitiendo la personalización que amplia los escenarios de reutilización. Esta información puede ser de tipo textual (string) y existir o no existir para indicar alguna característica booleana de verdadero o falso.
- Para trabajar con los atributos de un Custom Element, tenemos los mismos métodos que los elementos HTML estándar:
  - El método .**hasAttributes**() indica si tiene atributos, .**hasAttribute**(name) indica si tiene un determinado atributo y .**getAttributeNames**() devuelve un array de cadenas con todos los atributos del elemento (los nombre siempre están en minúsculas).
  - El método .**getAttribute**(name) devuelve el valor o null cuando no existe el atributo indicado (si existe pero no tiene valor, devuelve una cadena de texto vacía), .**setAttribute**(name, value) establecerá el atributo con el valor como cadena (si el valor establecido es cadena vacía, se añade como boolean) y .**removeAttribute**(name) elimina el atributo.
  - El método .**toggleAttribute**(name, *force*) añade un atributo si no existía previamente o lo elimina si ya existía. Si añadimos el segundo parámetro force, simplemente forzaremos a añadir o eliminar el atributo, sin tener en cuenta su estado previo. Este método devuelve si el atributo, tras las operaciones realizadas, existe o no.

Atributo HTML vs Propiedad DOM

- Los atributos están definidos por el HTML y las propiedades se definen mediante el DOM (Document Object Model):
  - Hay atributos que tienen una correspondencia 1: 1 con propiedades (Ej.: id).
  - Algunos atributos no tienen una propiedad correspondiente (Ej.: colspan).
  - Algunas propiedades no tienen un atributo correspondiente (Ej.: textContent).
- El atributo HTML (tiempo de diseño) y la propiedad DOM (tiempo de ejecución) nunca son lo mismo, incluso cuando tienen el mismo nombre. Los valores de una propiedad pueden cambiar, mientras que el valor del atributo no puede cambiar una vez expresado (forma en la que está escrito el documento HTML). Los atributos pueden inicializar propiedades, siendo las propiedades las que luego pueden cambiar.
- Es común que las propiedades reflejen su valor en el DOM como atributos HTML y viceversa. Los atributos también son útiles para configurar un elemento de forma declarativa y para ciertas APIs, como los selectores de CSS y de accesibilidad, que dependen de los atributos y no de las propiedades. Reflejar una propiedad es útil en cualquier lugar en el que desees mantener la representación del elemento DOM sincronizada con su estado de JavaScript.

```
get disabled() { return this.hasAttribute('disabled'); }
set disabled(value) { value ? this.setAttribute('disabled', '') : this.removeAttribute('disabled'); }
```

Detección de cambios (observación)

- Los WebComponents incorporan una interesante forma de detectar cambios en los atributos del custom element de forma automática, para que así podamos crear lógica que reaccione a dichos cambios. Por defecto, si establecemos un valor inicial al atributo de un componente y luego lo modificamos, no sabremos cuando ha cambiado el valor, ya que tendríamos que consultarlo de forma manual para obtener la información actualizada.
- Los WebComponents exponen, con la propiedad estática observedAttributes(), un array con los nombres de los atributos que queremos observar.

```
static observedAttributes = ['name', 'value'];
static get observedAttributes() { return ['name', 'value']; }
```

- Cada vez que cambien los valores fuera del componente de los atributos que están siendo observados se disparará el método del ciclo de vida denominado attributeChangedCallback(), donde se puede implementar la lógica de detección de cambios. El método attributeChangedCallback(name,old,now) recibe por parámetros el nombre del atributo que cambia en name, el valor que tenía antes en old y el valor que tiene actualmente en now.

```
attributeChangedCallback(property, oldValue, newValue) {
    if (oldValue !== newValue) this[property] = newValue;
}
```


Renderizar el componente

- Los elementos personalizados pueden administrar su propio contenido.
- Para pintar o presentar el componente es necesario generar el árbol DOM con el contenido del componente y asociarlo a través del this del componente, reemplazando o agregando contenido en el constructor o en connectedCallback(). Por lo general, no se recomienda reemplazar los elementos secundarios de un elemento por contenido nuevo, ya que es una acción inesperada y los que utilizan la etiqueta se sorprenderían si pierden su contenido.
- Los métodos interactuaran con los nodos (propiedades/métodos) o modificaran el árbol para que se repinte el componente.
- La forma más cómoda de crear el árbol DOM del componente es utilizar etiquetas HTML tradicionales dentro de una cadenas, los elementos disponen de propiedades que interpretan las cadena con HTML para producir el árbol DOM.
- El API del DOM de Javascript ofrece mecanismos para gestionar el árbol DOM, ofrece un mejor rendimiento pero requiere mucho código y puede resultar tedioso.
- Como alternativa, con HTML Templates, la etiqueta <template> permite crear un fragmento de código HTML de forma aislada del documento.

Shadow DOM con Custom Elements

- El Shadow DOM proporciona una alternativa para que un elemento posea una parte del DOM independiente del resto de la página, la renderice y le aplique estilos. El Shadow DOM nace de la necesidad de tener una forma de encapsular elementos y mantenerlos aislados del resto de la página.
- Para usar Shadow DOM en un elemento personalizado se llama a this.attachShadow dentro del constructor que obtiene el shadow root y convierte al componente en un shadow host:

```
this.attachShadow({mode: 'open'});
this._shadow = div.attachShadow({ mode: 'closed' });
```

- El renderizado del componente se realiza sobre el shadow root:

```
this.shadowRoot.innerHTML = /* html */`<p>En la sombra</p>`;  // open
this._shadow.innerHTML = /* html */`<p>En la sombra</p>`; // closed
```


Renderizar con HTML

- Las propiedades .innerHTML y .outerHTML se utilizan para reemplazar el marcado HTML de un componente, reciben una cadena con HTML que se interpreta, aunque .outerHTML no es demasiado práctico en los componentes, puesto que también sustituye la propia etiqueta del componente.

```
this.innerHTML = /* html */ `<div>Hola ${this.name}</div>`
```

- El uso de cadenas con contenido HTML que se interpretan está muy sujeto a controversia porque son poco eficientes, dado que se analizará el código HTML cada vez que se modifica la propiedad, y, si se compone la cadena dinámicamente incluso con entradas de usuario o terceros, pueden ser un riesgo de seguridad que requieren un proceso de saneamiento (futuros Sanitizer y .setHTML()).
- Las propiedades .textContent y .innerText se utilizan para obtener o cambiar solo el contenido textual, reciben una cadena que genera un nodo de texto y no se interpreta el HTML, se mostraría literalmente si lo contiene. Se recomienda usar .textContent ya que .innerText no obtiene el texto de determinado tipos de nodos.

```
this.textContent = `Hola ${this.name}`
```

Renderizar con API DOM

- Los principales métodos DOM disponibles para el mantenimiento del árbol de nodos son los siguientes:
  - createElement(eti, opt): Crea un elemento del tipo indicado en el parámetro.
  - createAttribute(atrib): Crea un nodo de tipo atributo con el nombre indicado.
  - createTextNode(texto): Crea un nodo de tipo texto con el valor indicado como parámetro.
  - cloneNode(deep): Clona el nodo HTML y devuelve una copia.
  - appendChild(nodo): Añade un nodo al final de la lista childNodes de otro nodo. Se debe invocar sobre el nodo que va a ser nodo padre del nodo añadido.
  - replaceChild(new, old): intercambia un nodo por otro. Se debe invocar sobre el nodo padre que contiene el nodo que se va a cambiar.
  - removeChild(nodo): elimina un nodo. Se debe invocar sobre el nodo padre del nodo que se va a eliminar.

Renderizar con API DOM

```
connectedCallback() {
    const container = document.createElement('div');
    container.className = 'container';
    this.appendChild(container);

    this.pantalla = document.createElement('output');
    this.pantalla.setAttribute('id', 'pantalla')
    this.pantalla.classList.add('pantalla');
    this.pantalla.textContent = this.count;
    container.appendChild(this.pantalla);
```

Pasos para crear elementos

1. Crear un nodo de tipo elemento
2. Cualificar el nuevo elemento con atributos. Por cada atributo:
   1. Crear un nodo de tipo atributo
   2. Asociar el nodo de atributo al elemento
3. Dar contenido al nuevo elemento.
   1. Crear un nodo de tipo texto
   2. Asociar el nodo de texto al elemento
4. Modificar al elemento padre que lo va a contener en la página original.
   - Añadir el nodo del nuevo elemento.
   - Buscar el nodo original y sustituirlo por el nodo del nuevo elemento.

Fragmentos

- En algunas ocasiones, nos puede resultar muy interesante utilizar fragmentos. Los fragmentos son una especie de documento paralelo, aislado de la página con la que estamos trabajando, que tiene varias características:
  - No tiene elemento padre. Está aislado de la página o documento.
  - Es mucho más simple y ligero (mejor rendimiento).
  - Si necesitamos hacer cambios consecutivos, no afecta al repintado de un documento (reflow).
- Es una estrategia muy útil para usarlo de documento temporal y no realizar cambios consecutivos, con su impacto de rendimiento:

```
const fragment = document.createDocumentFragment();
for (let i = 0; i < 5000; i++) {
  const div = document.createElement('div');
  div.textContent = `Item número ${i}`;
  fragment.appendChild(div);
}
document.body.appendChild(fragment);
```

Renderizar con HTML Templates

- La plantilla es un fragmento HTML de contenido que está siendo almacenado para un uso posterior en el documento. Las plantillas son marcadores de posición ideales para declarar la estructura de un elemento personalizado.
- Definir la representación del Web Component en plantillas presenta múltiples beneficios:
  - Permiten separar la presentación (plantilla) y la estética (style) de la lógica del componente (clase).
  - Permiten la composición, la proyección del contenido a través de los slot.
  - Se pueden externalizar para definirlas en páginas, el componente la recupera por identificador único o recibe el identificador como atributo. Esto tiene varios beneficios:
    - es más fácil definir HTML en HTML
    - se puede modificar el HTML sin tener que reescribir el JavaScript
    - se pueden modificar en el servidor o cliente antes de que se procese el componente
    - permiten personalizar el tipo de presentación si tener que extender el componente (crear herederos)
    - se puede incrustar los estilos
  - Se pueden crear en JavaScript con .createElement('template') en el mismo fichero que la clase del componente para evitar dependencias externas.

Renderizar con HTML Templates

- La clase del componente web debe acceder a la plantilla, obtener su contenido y clonar los elementos para asegurarse de que está creando un fragmento DOM único en todos los lugares donde se usa.
- Plantilla con identificador único, habitualmente coincide con el del componente:

```
<template id="my-element"><p>Con ID único</p></template>
this.appendChild(document.getElementById(this.localName).content.cloneNode(true))
```

- Plantilla con un identificador que se pasa al componente en un atributo:

```
<template id="my-template"><p>Con ID en atributo</p></template>
<my-element template="my-template"></my-element>
this.appendChild(document.getElementById(this.getAttribute('template')).content.cloneNode(true))
```

- Plantilla dinámica creada en JavaScript:

```
// cache fuera de la clase del componente
const myElementTemplate = document.createElement('template');
myElementTemplate.innerHTML = /* html */`<p>Todo en JavaScript</p>`;
// En el constructor o en connectedCallback
this.appendChild(myElementTemplate.content.cloneNode(true))
```

Renderizar con HTML Templates

- Las plantillas son dependencias externas al componente y su ausencia debe estar contemplada para evitar o tratar errores.
- El método .hasAttribute indica si se le ha pasado un determinado atributo al componente, en cuyo caso con .setAttribute se puede establecer el identificador único como valor predeterminado:

```
if(!this.hasAttribute('template')) // sin identificador de plantilla
   this.setAttribute('template', this.localName)
```

- Los métodos .getElementBy y .querySelector devuelven null si no encuentran el elemento.

```
if(!document.getElementById(this.localName)) // plantilla no encontrada
```

- En caso de error, dentro del constructor se puede eliminar el componente y lanzar una excepción (el objeto siempre se construye independientemente de las excepciones):

```
if(!document.getElementById(this.getAttribute('template')) {
   this.remove()
   throw new Error(`Falta la plantilla en ${this.id || this.localName}`)
}
```


Composición con Slot

- Los componentes son etiquetas que pueden tener contenido, por lo que a menudo se necesitan representar elementos secundarios dinámicos en ubicaciones específicas en su árbol de componentes, lo que permite que un desarrollador proporcione contenido secundario cuando usa nuestro componente, con nuestro componente colocando ese componente secundario en la ubicación adecuada.
- El Shadow DOM compone distintos árboles DOM con el elemento <slot>. Los slot son marcadores de posición dentro del componente que los usuarios pueden completar con su propio lenguaje de marcado. Los slot son una forma de crear un "API declarativa" para un componente.
- Los elementos pueden "cruzar" el límite del shadow DOM cuando un <slot> los invita. Estos elementos se denominan nodos distribuidos. Los ranuras no mueven físicamente el DOM, sino que lo renderizan en otra ubicación dentro del shadow DOM.
- Por lo tanto, el contenido del slot no pertenece al componente, pertenece a uno de sus contenedores, por lo que las acciones sobre nodos distribuidos están sumamente limitadas desde el componente para evitar efectos secundarios indeseados.

Trabajar con los Slots

- El navegador monitorea los slots y actualiza la representación si se agregan/eliminan elementos en el contenedor, no hay que hacer nada para actualizar el renderizado.
- Si el código del componente quiere saber acerca de los cambios del slot dispone del evento slotchange (el slot cambiado se indica en el event.target).
  - se activa la primera vez que se llena un slot
  - se activa en cualquier operación de agregar/quitar/reemplazar en el nodo con el atributo slot, pero no en sus hijos.
- A veces, es útil saber qué elementos están asociados con un slot (las consultas que no modifican evitan los efectos secundarios indeseados):
  - slot.assignedNodes(), slot.assignedElements(): devuelve la colección de nodos/elementos dentro del slot.
  - node.assignedSlot: propiedad inversa, devuelve el slot que contiene al nodo.

Anidamiento de componentes

- Cuando el constructor es llamado, el elemento es creado, pero el navegador aún no procesó ni asignó atributos en este estado, entonces las llamadas a getAttribute devolverían null, así que no podemos renderizar ahora si dependemos de ellos. El connectedCallback se dispara cuando el elemento es agregado al documento, cuando realmente se vuelve parte de la página. Así podemos construir un DOM separado, crear elementos y prepararlos para uso futuro y realmente renderizados una vez que estén dentro de la página. En términos de rendimiento es mejor demorar el renderizado hasta que realmente sea necesario.
- Los autores de componentes pueden decidir si el renderizado se realiza en el constructor o en connectedCallback. Cuando se anidan componentes, esto influye en el ciclo vida y la secuencia se puede alterar, lo que a su vez puede generar errores de precipitación al acceder árbol DOM:
  1. El constructor del componente contenedor
  2. Los constructores de los componentes contenidos
  3. El connectedCallback del componente contenedor
  4. Los connectedCallback de los componentes contenidos
  5. El post-connectedCallback del componente contenedor
  6. Los post-connectedCallback de los componentes contenidos
- Para los métodos post se utiliza un setTimeout a 0 que los ejecuta al terminar el ciclo actual.

Anidamiento de componentes

```
class MyElement extends HTMLElement {
  constructor() {
    super();
    console.log(`${this.id} created (content: ${this.children.length ? 'lleno' : 'vacío'})`)
  }
  connectedCallback() {
    console.log(`${this.id} connected (content: ${this.children.length ? 'lleno' : 'vacío'})`)
    setTimeout(() => console.log(`${this.id} post-connected (content: ${this.children.length ? 'lleno' : 'vacío'})`))
    this.appendChild(myElementTemplate.content.cloneNode(true))
  }
  disconnectedCallback() {
    console.log(`${this.id} disconnected (content: ${this.children.length ? 'lleno' : 'vacío'})`)
    setTimeout(() => console.log(`${this.id} post-disconnected (content: ${this.children.length ? 'lleno' : 'vacío'})`))
  }
}

<my-element id="contenedor">
  <my-element id="contenido"></my-element>
</my-element>
```


Eventos

- Frente a la programación tradicional, donde las aplicaciones se ejecutaban secuencialmente de principio a fin, en la actualidad el modelo predominante es el de la programación asíncrona basada en eventos.
- Los scripts y programas inicializan el entorno o página y quedan a la esperan, sin realizar ninguna tarea, hasta que se produzca un evento, situación ante la cual les interesa reaccionar. Una vez producido, para realizar el tratamiento del evento, ejecutan alguna tarea asociada a la aparición de ese evento. Al concluir el tratamiento, el script o programa vuelve al estado de espera de siguiente evento.
- El tratamiento del evento suele encapsularse en una función, conocidas como “controladores de eventos” o “manejadores de eventos”. El DOM permite asignar una o varias funciones a cada uno de los eventos con JavaScript. De esta forma, cuando se produce cualquier evento, JavaScript ejecuta su función asociada.
- Como cualquier elemento DOM, los WebComponents disponen de la capacidad para emitir y capturar eventos, ya cuentan con los eventos del tipo que extienden y pueden añadir sus propios eventos personalizado que actúan de salida desacoplada del componente.

El objeto event

- El objeto event es el mecanismo definido por los navegadores para proporcionar toda la información sobre el evento producido.
- Se trata de un objeto que se crea automáticamente cuando se produce un evento y que se destruye de forma automática cuando se han ejecutado todas las funciones asignadas al evento.
- El estándar DOM especifica que el objeto event es el único parámetro que se debe pasar a las funciones encargadas de procesar los eventos.
- El objeto event presenta unas propiedades y métodos muy diferentes en función del tipo de navegador en el que se ejecuta la aplicación JavaScript.
- Métodos:
  - preventDefault(): Cancela el evento (si éste es anulable).
  - stopPropagation: Para la propagación de los eventos más allá en el DOM.

Capturar Eventos

- Existen tres formas principales de capturar eventos en nuestro código:
  - Atributos de la etiqueta con el prefijo on de la etiqueta, no recomendable por ser intrusivos (no accesibles).
  - Propiedades del elemento con el prefijo on, limitados, solo aceptan un único controlador de eventos.
  - Listener del elemento: flexibles y no intrusivos, addEventListener() y removeEventListener() permiten asociar dinámicamente uno o varios manejadores de eventos a eventos y desasociar aquellos previamente asociados.
- Dentro de los WebComponents, los controladores de eventos se pueden asociar a los eventos estándar del propio componente o a los eventos de los nodos renderizados:

```
this.addEventListener('click', () => console.log('Hace click'))
this.btn = document.createElement('button');
this.btn.addEventListener('click', this.pulsa.bind(this))
```

Emitir Eventos

- Un WebComponent puede notificar a través de un evento una acción o suceso que ocurre durante su ejecución y que puede desencadenar una respuesta o ejecución de código específica.
- El objeto evento puede ser un evento nativo (build-in) o un CustomEvent (dispone de la propiedad detail para enviar un dato adicional).

```
this.dispatchEvent(new Event('keydown', { cancelable: true, composed: true }))
this.dispatchEvent(new MouseEvent('click', { bubbles: true, cancelable: true, }))
this.dispatchEvent(new CustomEvent('change', { detail: this.count }))
```

- Las principales opciones que definen el comportamiento del evento son:
  - cancelable: Indica si el comportamiento se puede cancelar con .preventDefault().
  - bubles: Indica si el evento debe burbujear en el DOM «hacia la superficie».
  - capture: Indica si el evento debe descender por el DOM «hacia dentro».
  - composed: Indica si la propagación puede atravesar Shadow DOM (WebComponents)
- Cuando enviamos eventos personalizados, necesitamos establecer las propiedades bubbles y composed a true para que se propague hacia arriba y hacia fuera del componente.

Emitir Eventos

- Frente a la instanciación con new de los objetos evento, el enfoque más antiguo para crear eventos utilizaba un API inspiradas en Java. El objeto evento puede ser creado mediante el método document.createEvent e inicializado usando initEvent u otro método, más específicamente, métodos de inicialización como initMouseEvent o initUIEvent.
- El método .dispatchEvent(event) lanza un evento en el sistema de eventos. El evento está sujeto al mismo comportamiento y capacidades que si fuera un evento de lanzamiento directo.

```
element.dispatchEvent(event)
```

- El valor devuelto es true, si ninguno de los controladores de eventos que manejan el evento invoca al .preventDefault() del evento, en caso contrario se considera cancelado (si cancelable: true).

```
const isCancel = !element.dispatchEvent(event)
```

- Los eventos emitidos por los WebComponents se capturan y comportan de la misma forma que el resto de los eventos:

```
document.querySelector('my-element').addEventListener('change', ev => {
```

Eventos en el Shadow DOM

- Los eventos que ocurren en el Shadow DOM se quedan en el Shadow DOM, salvo que tengan establecida la propiedad composed a true. La mayoria de los eventos nativo (build-in) ya la tiene a true, los principales eventos que la tiene a false son mouseenter, mouseleave, load, unload, abort, error, select y slotchange. Estos eventos solo pueden ser capturados dentro del mismo DOM, donde reside el event target.
- Para evitar la propagación de los eventos nativo con composed a true hay que invocar:
  event.stopPropagation()
- Cuando los eventos que ocurren en el Shadow DOM, el navegador los redirige, es decir tienen como target el elemento host (componente) cuando son atrapados fuera del componente, para ocultar los elementos internos.
- No hay redirección si el evento ocurre en un elemento dentro de un slot dado que físicamente se aloja en el light tree, el DOM visible.
- Usando event.composedPath() se puede obtener un array con el conjunto de nodos que recorrió el evento, incluyendo los elementos shadow.
- Los detalles del árbol Shadow solo son provistos con Shadow DOM en {mode:'open'}.

Comunicación entre componentes

- Un componente puede estar compuesto por componentes que a su vez se compongan de otros componentes y así sucesivamente. Sigue un modelo de composición jerárquico con forma de árbol de componentes.
- Los componentes deben establecer un cauce bien definido de entrada/salida para su comunicación con otros componentes (contenedores/contenido, hermanos, …) y con la página. Los componentes flexibles son más reutilizables y mantenibles.
- Los atributos observados son la entrada del componente y se utilizarán para pasar información desde el exterior al propio componente, incorporan la detección cambios para reaccionar ante los mismos.
- Se denomina elevar estado al mecanismo mediante el cual un componente comunica sus cambios de estado, una acción o suceso que ocurre durante su ejecución, a su contenedor o quien pueda interesar. A través de un evento se puede desencadenar una respuesta o ejecución de código específica. Los eventos personalizados serán las salidas y utilizan el patrón delegate que desacopla al emisor (no conoce al receptor) del receptor (no sabe que hace el emisor). Los eventos deben establecer bubbles y composed a true para que se propague hacia arriba y hacia fuera del componente (atravesar el Shadow DOM).

Elementos personalizados preconstruidos

- Los elementos integrados personalizados son un tipo distinto de elemento personalizado, que se definen de manera ligeramente diferente y se usan de manera muy diferente en comparación con los elementos personalizados autónomos.
- Existen para permitir la reutilización de comportamientos de los elementos existentes de HTML, ampliando esos elementos con nueva funcionalidad personalizada.
- Esto es importante ya que, lamentablemente, muchos de los comportamientos existentes de los elementos HTML no se pueden duplicar mediante el uso de elementos personalizados puramente autónomos.
- En cambio, los elementos integrados personalizados permiten la instalación de comportamientos de construcción personalizados, ganchos de ciclo de vida y cadenas de prototipos en elementos existentes, esencialmente "mezclando" estas capacidades además de las ya existente que tiene el elemento en la especificación HTML, no pueden extender elementos heredados.
- Los elementos integrados personalizados requieren una sintaxis distinta de la de los elementos personalizados autónomos porque los navegadores y otros eliminan el nombre local de un elemento para identificar la semántica y el comportamiento del elemento. Es decir, el concepto de elementos integrados personalizados que se construyen sobre el comportamiento existente depende crucialmente de que los elementos extendidos conserven su nombre local original.

Elementos personalizados preconstruidos

- Los elementos personalizados preconstruidos no extienden a HTMLElement si no a uno de sus heredero (elementos HTML básicos), por lo que tiene todas las características del elemento extendido además de la funcionalidad que agreguemos nosotros.

```
class MyButton extends HTMLButtonElement {
```

- Al definir el elemento preconstruido, hay que especificar la opción extends:

```
customElements.define('my-button', MyButton, { extends: 'button' });
```

- Para construir el elemento personalizado preconstruido a partir del texto fuente HTML analizado, se declara una etiqueta del tipo extends definido añadiendo un atributo (o propiedad) is y dándole como valor el nombre del elemento personalizado definido:

```
<button is="my-button">Send</button>
```

- Para crear un elemento personalizado preconstruido por código también es necesario indicar el is o utilizar el constructor:

```
const myButton = document.createElement('button', { is: 'my-button' });
myButton.disabled = true
document.body.appendChild(new MyButton()); // <button is="my-button"></button>
```

Elementos internos

- Ciertas capacidades deben estar disponibles para el creador de un elemento personalizado y no para el consumidor de un elemento personalizado.
- Estas capacidades las proporciona el método element.attachInternals(), que devuelve una instancia de ElementInternals.

  this._internals = this.attachInternals();
- Las propiedades y métodos de ElementInternals permiten el control sobre las características internas que el navegador proporciona a todos los elementos. Estas características internas permiten convertir un elemento personalizado autónomo en un elemento personalizado asociado a un formulario y establecer una semántica de accesibilidad predeterminada.
- Para convertir un elemento personalizado autónomo en un elemento personalizado asociado a un formulario:

  static formAssociated = true;
- Para dar valores predeterminados a los atributos role y :aria-* que posteriormente se pueden personalizar:

  this._internals.role = 'checkbox';

  this._internals.ariaChecked = 'false';

Elementos internos

```
class MyCheckbox extends HTMLElement {
  static formAssociated = true;
  static observedAttributes = ['checked'];
  constructor() {
    super();
    this._internals = this.attachInternals();
    this._internals.role = 'checkbox';
    this._internals.ariaChecked = 'false';
    this.addEventListener('click', this._onClick.bind(this));
    this._setTextContent()
  }
  get form() { return this._internals.form; }
  get name() { return this.getAttribute('name'); }
  get type() { return this.localName; }
  get checked() { return this.hasAttribute('checked'); }
  set checked(flag) { this.toggleAttribute('checked', Boolean(flag)); }

  attributeChangedCallback(name, oldValue, newValue) {
    this._internals.setFormValue(this.checked ? 'on' : null);
  }
  _onClick(event) {
    this.checked = !this.checked;
    this._internals.ariaChecked = this.checked;
    this._setTextContent()
  }
  _setTextContent() {
    this.textContent = this.checked ? 'Acepto' : 'NO Acepto'
  }
}
customElements.define('my-checkbox', MyCheckbox);

<form>
  <my-checkbox role="button" name="agreed"></my-checkbox>
  <input type="submit">
</form>
```

CSS en Custom Elements

- Los Custom Elements son HTMLElement por lo que heredan los atributos class y style, que permiten estilos definidos por el usuario, y tienen una etiqueta (name) que es un selector CSS:

```
my-element {
  display: block;  /* HTMLElement no lo tiene predefinido */
  border: 1px solid black;
}
```

- Los estilos para los componentes se puedes definir globalmente (estilos externos) o localmente (estilos internos). Los estilos externos siempre prevalecen sobre los estilos internos.
- Los estilos globales se definen en el documento o en el contenedor y afectan a todos los elementos visibles (fuera de un shadow host) aunque estén contenidos en un componente. Los elementos de los slot también reciben los estilos globales si provienen de light tree aunque su componente sea un shadow host.
- Los componentes pueden definir localmente el estilo. Si el componente es un shadow host, los estilos se aplican localmente solo al shadow tree, en caso contrario, se agregan (filtran) al estilo global y se aplican a todos los elementos visibles (light tree) independientemente de si pertenecen o no al componente. Los componentes pueden definir su estilo local a nivel de instancia (manipulando sus atributos className y style, no se filtran) o nivel de componente (se filtran sin Shadow DOM).

Definir estilos del componente

- El estilo de una instancia del componente se puede definir manipulando sus atributos className y style que no se filtran:

```
this.style = 'display: block; background-color: lightgreen;'
this.className = 'active valid'
```

- Se pueden definir a través de un bloque de estilos:

```
this.innerHTML = /* html */ `
  <style>div { color: red; }</style>
  <div>Hola mundo</div>`
```

- Se pueden definir referenciando estilos externos con una etiqueta <link rel="stylesheet" href="…"> o una regla @import (en un bloque <style>) que cargue un fichero .css externo. Las hojas de estilo se almacenan en la caché para que no se vuelvan a descargar en cada instancia.

```
this.innerHTML = /* html */ `
  <link rel="stylesheet" href="my-element.css">
  <div>Hola mundo</div>`
```

Hojas de estilo dinámicas

- La especificación Constructable StyleSheets permite crear hojas de estilo de manera imperativa invocando el constructor CSSStyleSheet().
- La interfaz CSSStyleSheet es la raíz de una colección de interfaces de representación de CSS conocida como CSSOM y ofrece una manera programática de manipular las hojas de estilo.
- Los métodos insertRule, replace (devuelve una promesa) y replaceSync reciben una cadena CSS con solo una (insertRule) o varias reglas CSS, salvo deleteRule que recibe el índice de la regla a eliminar. No se admiten las referencias de hojas de estilo externas (reglas @import).

```
// let styles = document.createElement('style') // obsoleto
// style.textContent = `div { color: red; }`
```

- La propiedad adoptedStyleSheets de Document y ShadowRoot se utiliza para añadir o configurar las hojas de estilo que utilizará el documento o el componente.

```
let styles = new CSSStyleSheet();
styles.replaceSync('div { color: red; }');
document.adoptedStyleSheets.push(styles); // Global
this.shadowRoot.adoptedStyleSheets = [styles]; // Local
```


Selectores CSS para componentes

- Los selectores específicos de Shadow DOM son:
  - :host y :host(), permiten acceder al elemento contenedor que contiene el Shadow DOM
  - :host-context(), permite restringir la regla con un selector que consulta todos los antecesores hasta la raíz del documento
  - ::slotted(), permite dar estilo a las etiquetas insertadas en un slot
- Especifico de los Custom Elements, la pseudoclase :defined permite aplicar estilos dependiendo de si el elemento ha sido o no definido en el navegador. En la "Actualización de componentes" el elemento del componete queda como “undefined” hasta que se completa la ejecución de su customElements.define(). Puede ser conveniente aplicar un estilo previo al componente hasta su definición (avisos visuales, Core Web Vitals mejorado, reducción de cambios de diseño, …). Un elemento “undefined” puede seleccionarse con :not(:defined).

```
my-element:not(:defined) {
  display: inline-block;
  height: 100vh;
  opacity: 0;
  transition: opacity 0.3s ease-in-out;
}
```

Restablecer estilos heredables

- Los estilos heredables de los contenedores (background, color, font, line-height, etc.) llegan en cascada al shadow DOM. Es decir, penetran el límite del shadow DOM de forma predeterminada.
- Para restablecer los diseños heredables a su valor inicial cuando crucen el limite (shadow boundary), se usa la propiedad all: initial; en el shadow host del componente, al principio de la regla para que se restablezcan las propiedades posteriores.

```
:host {
  all: initial;
  display: block;
}
```


Propiedades personalizadas de CSS

- En CSS, las propiedades personalizadas (conocidas como variables) son entidades definidas por autores de CSS que contienen valores específicos que se pueden volver a utilizar en un documento.
- Se pueden definir desde el documento global, permiten definir valores y tienen ámbito en el DOM, por lo que son un sistema muy potente y flexible al ahora de utilizarlas.
- Uno de los detalles más importantes y más desconocido es que las propiedades personalizadas CSS son capaces de penetrar y filtrarse a través del Shadow DOM de un elemento, por lo que son una estrategia perfecta, desde un ámbito global, para personalizar los estilos de los componentes con ellas.
- El alcance es global si se declara dentro del pseudo selector :root y local en el resto (accesible en el selector donde se declara o en cualquier selector anidado), primando las definiciones mas locales.

```
:root { --error-color: red; }
```

- La función var() permite recuperar el valor de la variable para definir el valor de otra propiedad. Opcionalmente se puede definir un valor por defecto para cuando no exista la variable. Se pueden utilizar al definir el estilo de un componente (aunque tenga Shadow DOM):

```
:host { color: var(--error-color, red); }
```

- El componente puede alterar dinámicamente su valor:

```
this.style.setProperty('--error-color', 'pink');
```

Personalización con CSS Shadow Parts

- Las propiedades personalizadas de CSS permiten personalizar valores individuales, por lo que tienen un alcance limitado. Para situaciones en las que el consumidor requiere una personalización mas compleja, las CSS Shadow Parts ofrecen un mayor grado de flexibilidad en el estilo.
- Para definir partes del componente:

```
<output part="pantalla">{this.counter}</output>
```

- Para exponer sus partes:

```
<my-contador exportparts="pantalla"></my-contador>
```

- Para definir el estilo de una de las partes:

```
my-contador::part(pantalla) {
  font-size: 2em;
}
```


JAVASCRIPT FRAMEWORK

JavaScript Framework

- En el ámbito del desarrollo del software, el término “JavaScript Framework” significa una biblioteca que proporciona a los desarrolladores plantillas preconstruidas y código JavaScript preescrito para tareas de programación estándar. Millones de desarrolladores usan este tipo de frameworks para acelerar el flujo de trabajo de desarrollo y aplicar las mejores prácticas de una manera fluida y fácil.
- Debido a su gran variedad, su uso depende de los objetivos principales, la funcionalidad general de la plataforma, los requisitos del proyecto y cómo se puede implementar dentro de cada escenario específico.
- Al usar este tipo de marcos desarrollo para JavaScript, se puede ahorrar una gran cantidad de tiempo y esfuerzo en el desarrollo de sitios web y aplicaciones basados en este lenguaje. Simplifica todo el procedimiento y permite a los desarrolladores crear aplicaciones web a gran escala de manera eficiente.
- Los frameworks de JavaScript son una parte esencial del desarrollo web front-end moderno, los cuales proveen a los desarrolladores herramientas probadas y testeadas para la creación de aplicaciones web interactivas y escalables. Muchas empresas modernas utilizan frameworks como parte estándar de sus herramientas, por lo que muchos trabajos de desarrollo front-end en la actualidad requieren experiencia en frameworks.

Clasificación

- Por Tipos:
  - Estructurales
  - Específicos
- Por Aspectos:
  - Presentación
    - Plantillas
    - Enlazado
    - Gráficos, animaciones, …
    - Componentes
  - Comunicaciones
  - Enrutado
  - Arquitectónicas
    - Mantenimiento de Estado
    - Inyección de dependencias
    - Modelos de programación: reactiva, funcional, modularidad, …
  - Testing y QA

https://github.com/sorrycc/awesome-javascript

Generación del esqueleto de aplicación

- Configurar un nuevo proyecto web puede ser un proceso complicado y tedioso, con tareas como:
  - Crear la estructura básica de archivos y bootstrap
  - Configurar SystemJS o WebPack para transpilar el código
  - Crear scripts para ejecutar el servidor de desarrollo, tester, publicación, …
- Disponemos de diferentes opciones de asistencia:
  - Proyectos semilla (seed) disponibles en github
  - Generadores basados en Yeoman
  - Herramientas oficiales de los Framework (CLI):
    - Los *Command Line Interface* (CLI) permiten generar proyectos y elementos de código desde consola, así como ejecutar un servidor de desarrollo o lanzar los tests de la aplicación.

Vainilla (Vanilla JS) vs biblioteca

- Los componentes web pueden resultar difíciles de escribir desde cero. Hay mucho en qué pensar y escribir, un componente puede requerir una gran cantidad de código repetitivo. Afortunadamente, existen algunas bibliotecas excelentes que pueden hacer que la creación de elementos personalizados sea más sencilla y ahorrarle mucho tiempo y esfuerzo.
- Es importante tener en cuenta que no es necesario utilizar una biblioteca para crear y compartir un elemento personalizado. Los elementos personalizados sin procesar son excelentes si su tarea se limita a uno o unos pocos elementos, ya que permiten una implementación más optimizada y evitan el bloqueo de la biblioteca. Vanilla.JS es una broma creada por Eric Wastl que promueve la vuelta a las esencias.
- Sin embargo, si se está escribiendo muchísimos elementos personalizados, el uso de una biblioteca puede hacer que su código sea más simple y limpio, y que su flujo de trabajo sea más eficiente.
- Una buena biblioteca de componentes web debería producir un componente web que "simplemente funcionen" como cualquier otro elemento HTML. Las buenas bibliotecas también tienen una alta relación valor-carga útil, es decir, proporcionan mucho valor con el mínimo tamaño de su descarga.

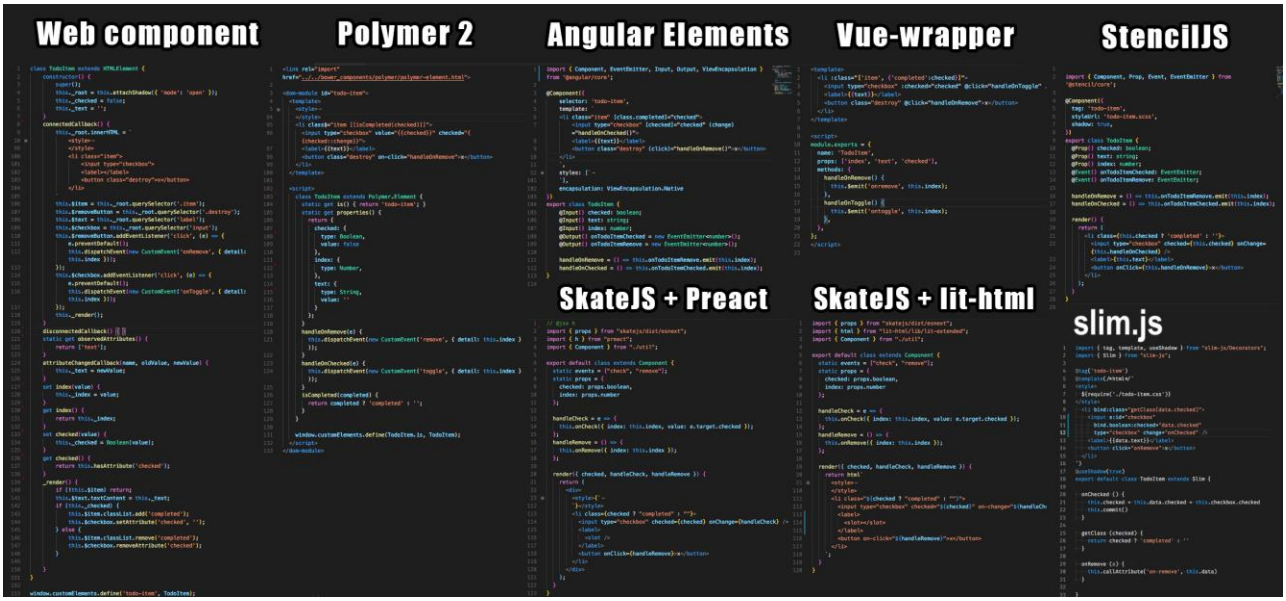
Vanilla JS

- Vanilla JS actualmente es el framework más usado en Internet, siendo usado por Google, Amazon, Twitter y muchas otras webs.
- A la hora de acceder al DOM por IDs es el doble de rápido que Dojo y 100 veces más rápido que Prototype JS cuando accedemos a los elementos por el nombre de la etiqueta, siendo estos unos de los framework más eficiente en cuanto a acceder al DOM, solo superado por Vanilla JS.
- Vanilla JavaScript es como se conoce al lenguaje JavaScript cuando se utiliza sin ninguna librería o framework. La traducción más castellana sería JavaScript a pelo.
- Vanilla JS es una iniciativa, en forma de framework que intenta enseñar las grandes ventajas de no usar frameworks y potenciar nuestras aplicaciones sin necesidad de añadir grandes archivos extra.
- No dicen que los framework sean malos, simplemente que hay que descargarlos, interpretarlos y reducen el rendimiento (mejoran la productividad) aumentando los consumos, en muchas ocasiones se abusa de estos frameworks para cosas que no son necesarias y que se pueden llevar a cabo sin ellos.

Bibliotecas para construir componentes web

- Hybrids es una biblioteca de interfaz de usuario para crear componentes web con API simple y funcional.
- Polymer proporciona un conjunto de funciones para crear elementos personalizados.
- LitElement usa lit-html para representar en el Shadow DOM del elemento y agrega API para ayudar a administrar las propiedades y atributos del elemento.
- Slim.js es una biblioteca de componentes web liviana de código abierto que proporciona enlace de datos y capacidades extendidas para componentes, utilizando la herencia de clases nativa de es6.
- Stencil es un compilador de código abierto que genera componentes web que cumplen con los estándares.
- Los Angular Elements son componentes Angular empaquetados como Web Components.
- Implementaciones con diferentes frameworks
  - https://github.com/shprink/web-components-todo
  - https://github.com/thinktecture-labs/web-components-chat

Bibliotecas para construir componentes web



JavaScript Framework

- **MooTools** (My object oriented tools) es un Framework web orientado a objetos para JavaScript, de código abierto, compacto y modular. Aporta una manera de desarrollar JavaScript sin importar en qué navegador se ejecute de una manera elegante.
- **Prototype** es un framework escrito en JavaScript que se orienta al desarrollo sencillo y dinámico de aplicaciones web. Es una herramienta que implementa las técnicas AJAX y su potencial es aprovechado al máximo cuando se desarrolla con Ruby On Rails.
- **jQuery** es una biblioteca de JavaScript, permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web.
- **jQuery UI** es una biblioteca de componentes para el framework jQuery que le añaden un conjunto de plug-ins, widgets y efectos visuales para la creación de aplicaciones web. Cada componente o módulo se desarrolla de acuerdo a la filosofía de jQuery5 (find something, manipulate it: encuentra algo, manipúlalo).
- **Dojo** es un framework que contiene APIs y widgets (controles) para facilitar el desarrollo de aplicaciones Web que utilicen tecnología AJAX. Contiene un sistema de empaquetado inteligente, los efectos de UI, drag and drop APIs, widget APIs, abstracción de eventos, almacenamiento de APIs en el cliente, e interacción de APIs con AJAX.

JavaScript Framework

- **Twitter Bootstrap** es un framework enfocado en el diseño adaptativo que permite construir sitios web rápidamente con estilos, plantillas y funciones predefinidas puestas a disposición del desarrollador.
- **AngularJS** es un framework de JavaScript de código abierto, que ayuda con la gestión de lo que se conoce como aplicaciones de una sola página. Su objetivo es aumentar las aplicaciones basadas en navegador con capacidad de Modelo Vista Controlador (MVC), en un esfuerzo para hacer que el desarrollo y las pruebas sean más fáciles.
- **Backbone** es una herramienta de desarrollo/API para el lenguaje de programación Javascript con un interfaz RESTful por JSON, basada en el paradigma de diseño de aplicaciones Modelo Vista Controlador. Está diseñada para desarrollar aplicaciones de una única página y para mantener las diferentes partes de las aplicaciones web sincronizadas.
- **Ext JS** es una biblioteca de JavaScript para el desarrollo de aplicaciones web interactivas usando tecnologías como AJAX, DHTML y DOM.
- **Node.js** es un entorno de programación en la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web.

Agrupar y minificar recursos

- La agrupación y la minificación (minimización) son dos técnicas para mejorar el tiempo de carga de la solicitud. La agrupación y minificación mejora el tiempo de carga al reducir el número de solicitudes al servidor y reducir el tamaño de los activos solicitados (como CSS y JavaScript).
- La mayoría de los exploradores principales actuales limitan el número de conexiones simultáneas por cada nombre de host. Esto significa que, mientras se procesan las primeras solicitudes, el explorador pondrá en cola las solicitudes adicionales, a la espera de que vayan recibiendo las respuestas. La agrupación permite combinar varios archivos JavaScript (.js) o varios archivos de hoja de estilos en cascada (.css) para que se puedan descargar como una unidad, en lugar de individualmente.
- Se entiende por "minificación" el proceso mediante el cual se eliminan datos innecesarios o redundantes de un recurso sin que se vea afectada la forma en que los navegadores lo procesan. Por ejemplo, eliminar comentarios y formato innecesario, retirar código que no se usa, acortar los nombres de variables y funciones a un carácter, …
- Es aconsejable minimizar los recursos de HTML, CSS y JavaScript.