Manual de Web Components

Miguel Angel Alvarez

desarrolloweb.com

desarrolloweb.com/manuales/web-components.html

Introducción: Manual de Web Components

En este manual vamos a conocer el nuevo estándar de los Web Components, una nueva tecnología implementada en navegadores modernos que permitirá llevar la web a un nuevo nivel.

Web Components incluye cuatro especificaciones que nos van a servir para cambiar radicalmente el modo en el que construimos las páginas web, aportando nuevas herramientas capaces de extender el lenguaje HTML. Las veremos por separado y luego las utilizaremos en conjunto.

Los componentes, también llamados Custom Elements, son el corazón y objetivo final de este estándar y tienen como objetivo construir nuevos elementos para el lenguaje HTML. Éstos son como etiquetas HTML nuevas que puedes desarrollar tú mismo con muy poco esfuerzo, de modo que puedas realizar componentes para implementar cualquier tipo de tarea en el ámbito de una web, interfaz de usuario, etc.

Como Web Components es un estándar, podemos desarrollar directamente con Javascript y no estaremos obligados a usar alguna librería o framework adicional. En este manual nos centraremos en esta posibilidad, usar Web Components con Javascript estándar.

Por todo ello es una tecnología que ya podemos usar y beneficiarnos de extraordinarias posibilidades. En este manual vamos a recorrer diversos puntos del estándar para explicarlo, de modo que los desarrolladores puedan aprender y comenzar a usar Web Components.

Encuentras este manual online en:
http://desarrolloweb.com/manuales/web-components.html

Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

Miguel Angel Alvarez

Fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.

Introducción a Web Components

Web Components es un estándar de la W3C que está siendo definido en el momento de escribir este manual. Explicaremos con detalle en qué consiste el estándar y su filosofía.

Las especificaciones están a distintos niveles de finalización pero, como ya se encuentran publicados sus borradores, varios navegadores las vienen implementando. Todos los navegadores en algún momento se adaptarán para soportar Web Components, pero para los navegadores antiguos explicaremos cómo usar el Polyfill, que nos aporta compatibilidad entre los clientes web que no lo implementan todavía.

Qué son Web Components

Conoce el estándar Javascript de Web Components, cuáles son sus 4 especificaciones y cómo nos permiten extender el HTML para crear componentes reutilizables. Por qué es una revolución en el desarrollo frontend.

En este artículo vamos a realizar una introducción teórica a lo que son los Web Components, una revolución en el mundo del desarrollo para la web que ya es toda una realidad, en la medida en la que actualmente todos los navegadores soportan el estándar.

Lo veremos con detalle en este artículo pero Web Components es un estándar Javascript, por lo que no es necesario el uso de librerías para crear nuestros propios componentes. Aún así existen ibrerías como Lit que todavía permiten agregar funcionalidad encima del estándar, con pesos cercanos a los 5KB de código. Son una excelente opción para desarrollar componentes reutilizables y están dando mucho de que hablar últimamente.

> Otras librerías como Angular o React también permiten desarrollar componentes pero no están basados en el estándar Web Components. Si las conoces te puedes hacer una idea de qué son componentes y cómo permiten organizar y reutilizar tu código. Aunque para ser exactos el estándar es bastante más importante que una tecnología en particular, cuya vida depende del equipo de desarrollo y el tiempo que consiga permanecer "de moda".

Vamos a comenzar explicando el objetivo que vienen a cubrir los Web Components y luego trataremos acerca de las 4 especificaciones que podemos encontrar en esta tecnología. Además en este artículo vamos a aclarar algunos puntos interesantes sobre la tecnología y su evolución, a lo largo de sus diversas versiones y el soporte de los navegadores.

Por qué de los Web Components

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.

En resumen, este nuevo estándar viene a facilitar la creación de nuevos elementos que enriquezcan la web. Pero además, está pensado para que se puedan reutilizar de una manera sencilla y también extender, de modo que seamos capaces de crear unos componentes en base a otros.

Como veremos, al diseñarse los estándares para los Web Components también se ha procurado que se pueda trabajar con los componentes de manera aislada, permitiendo que las nuevas piezas puedan usarse en el contexto de una web sin que afecten a otras ya existentes. Paralelamente se ha tratado de que el proceso de cargar un nuevo componente en una página se pueda realizar de manera atómica (un solo bloque) en lugar de como se suele hacer con muchas librerías y plugins actuales que requieren de escribir los estilos por una parte y el javascript por otra.

> Nota: El W3C está encargado de mantener los estándares, pero lo cierto es que sus procedimientos para permitir la evolución de la web son un poco pesados. Nos referimos a que los desarrolladores habitualmente detectamos necesidades mucho antes que la W3C realice un estándar para poder cubrirlas. De hecho, pueden pasar años desde que algo comienza a ser usado en el mundo de la web hasta que se presenta el estándar. En resumen, el mundo de la web va mucho más rápido que la definición de los estándares.

Ejemplos clásicos de Web Components

El ejemplo más típico que veremos por ahí es un mapa de Google. Hoy, si no usamos web components, cuando queremos mostrar un mapa en una página web, tenemos que crear código en tres bloques.

1. Un HTML con el elemento donde se va a renderizar el mapa
2. Un CSS para definir algún estilo sobre el mapa, por ejemplo sus dimensiones
3. Lo más importante, un Javascript para que puedas generar el mapa, indicando las coordenadas que deseas visualizar (para centrar la vista inicial) y muchos otros detalles

de configuración que tu mapa necesite.

Otro ejemplo sería un calendario, que necesitas de nuevo básicamente tres partes:

1. HTML para crear el elemento donde se mostrará el calendario
2. CSS para indicar las dimensiones de ese calendario, colores, etc.
3. Javascript para decir qué mes, día o año debe mostrar.

Son tres lenguajes diferentes, que se especifican en bloques de código separados y usualmente en archivos separados. Sin Web Components, para tener todos los bloques agrupados y tener un código único para embeber un elemento se usaba generalmente la etiqueta IFRAME, que permite cargar un HTML, CSS y Javascript y reducir su ámbito a un pequeño espacio de la página. Esta técnica se sigue utilizando, pero en el futuro se va a sustituir gracias a las bondades de los Web Components.

A partir de ahora podremos expresar un mapa de Google con una etiqueta propietaria, que no pertenece al estándar del HTML, que simplifica la tarea y la acota a un pequeño bloque independiente.

```
<google-map latitude="12.678" longitude="-67.211"></google-map>
```

Para incluir un calendario que indique los días que estamos libres u ocupados podremos usar una etiqueta propietaria en la que indicamos las características de ese calendario.

```
<google-calendar-busy-now
   calendarId="TU_ID_CAL"
   apiKey="TU_LLAVE_API"
   busyLabel="Ocupado"
   freeLabel="Estoy libre">
</google-calendar-busy-now>
```

Son dos ejemplos tomados directamente de Web Components reales, creados por el equipo de Google. Tienen como intención reflejar:

1. Es como si estuviéramos inventando etiquetas nuevas. Esa es una de las capacidades de los Web Components, pero no la única.
2. Las etiquetas propietarias que nos estamos inventando son "google-map" y "google-calendar-busy-now"
3. No tenemos el HTML por un lado, el CSS y el Javascript por otro. Es simplemente la etiqueta nueva y ésta ya es capaz de lanzar el comportamiento.
4. Obviamente, en algún lugar habrá un Javascript que se encargará de procesar esa etiqueta, pero será genérico para cualquier tipo de mapa y reutilizable. Lo que además debe verse es que en el HTML estás colocando información que antes estaría en el Javascript. Por ejemplo en el caso del mapa de google los atributos latitude="12.678" longitude="-67.211" antes eran datos que se escribían en el Javascript. Ahora se declaran en el HTML. El Javascript por tanto es genérico y no tendremos que programarlo nosotros, sino que nos vendrá dado por Google o por el creador del web component de turno.

Versiones del estándar Web Components V0 y V1

Actualizado en septiembre de 2018: Aunque estemos ante una tecnología relativamente nueva, ya han surgido diversos cambios en el estándar que es importante explicar. Se trata de las versiones de Web Components V0 y V1 y sus implicaciones.

Este estándar ha sido impulsado principalmente por Google, empresa donde comenzaron el diseño de Web Components, según como ellos mismos consideraban que debería ser. Sin embargo, para la creación de estándares abiertos, como el HTML, CSS, etc. no solamente opina una empresa, sino un conjunto de empresas y profesionales bien posicionados en el sector. Es por ello que, al tiempo que Web Components pasaba de ser un proyecto particular a un estándar abierto, se fueron introduciendo modificaciones.

La versión inicial de Web Components, creada prácticamente en exclusiva por Google, se denominó "Web Components V0". En palabras de ellos mismos, era un experimento para ver cómo salían las cosas y hacia dónde les llevaban los objetivos de traer el mundo de los componentes a la web.

La versión del estándar que conocemos hoy se llama ", cuando Google "Web Components V1". Esta versión ya podemos considerarla definitivamente un estándar abierto, dado que en el proceso de su creación han participado todo un bloque de empresas y profesionales de diversas áreas de la informática.

Web Components V1 trajo con si una novedad principal con respecto la versión V0, la sustitución de los HTML Imports por los ES6 Module Imports. Enseguida hablaremos de ello con más detalle.

Especificaciones en Web Components

Ahora que ya hemos entendido alguna cosa de lo que son los componentes web, el concepto en sí, vamos a ser un poco más formales y describir las distintas especificaciones que podemos encontrar en los Web Components.

Custom Elements: Esta especificación describe el método que nos permitirá crear nuevas etiquetas personalizadas, propietarias. Estas etiquetas las podremos ingeniar para dar respuesta a cualquier necesidad que podamos tener. Son los casos básicos que hemos visto en los puntos anteriores de este artículo.

HTML Templates: Incorpora un sistema de templating en el navegador. Los templates pueden contener tanto HTML como CSS que inicialmente no se mostrará en la página. El objetivo es que con Javascript se acceda al código que hay dentro del template, se manipule si es necesario y posteriormente se incluya, las veces que haga falta, en otro lugar de la página.

HTML Imports: Permite importar un pedazo de código que podrás usar en un lugar de tu página. Ese código podrá tener HTML, CSS y Javascript. El HTML no se visualizará directamente en la página, pero lo podrías acceder con Javascript e inyectar en algún lugar. Pero aunque se llame específicamente "HTML Imports", realmente sirve para cargar de una manera única tanto HTML como CSS como Javascript. Además podrás tener dentro un "HTML Template", con las ventajas que ellos aportan. Mendiante código Javascript seremos

capaces también de registrar componentes personalizados "Custom Elements" o realizar otro tipo de acciones sobre el contenido de la página que sean necesarias.

Shadow DOM: Este sistema permite tener una parte del DOM oculta a otros bloques de la página. Se dice comúnmente que estamos encapsulando parte del DOM para que no interfiera con otros elementos de la página. Básicamente te sirve para solucionar un caso común que ocurre al incluir un plugin de terceros. A veces usan clases o identificadores para aplicar estilos que afectan a otros elementos de la página, descolocando cosas que no debería o alterando su aspecto. Pues con el Shadow DOM podemos hacer que los componentes tengan partes que no estarían visibles desde fuera, pudiendo colocar estilos que solo afectan al Shadow DOM de un web component y evitando que estilos de la página sean capaces de afectar al Shadow DOM.

> Nota: Estas 4 especificaciones, aunque las tengamos por separado, están encaminadas a trabajar en conjunto para un mismo fin: poder realizar tus propios componentes para la web. Las veremos más adelante con detalle.

Coyuntura de los HTML Imports y su evolución a los Imports de ES6 definitiva

Los HTML Imports fue la parte de Web Components que menos apoyos obtuvo por parte de la comunidad durante el proceso de estandarización. Tanto es así que muchos navegadores nunca los han llegado a soportar.

El problema de los HTML Imports era que cubria un mismo objetivo de otra herramienta usada para requerir código, los ES6 Modules. Mientras que los HTML Imports estaban preparados para requerir archivos HTML, con los Module imports de ES6 estaban preparados para traerse código Javascript, pero esta diferencia no fue suficiente para convencer a la comunidad de la necesidad de un estándar para importar código en la web.

Así que finalmente se decidió usar los ES6 Modules, que actualmente ya están disponibles en los navegadores, en detrimento de los HTML Imports. Esto tuvo una implicación importante en los Web Components, porque antes se programaban dentro de archivos HTML y se comenzó a programar dentro de archivos Javascript. Hoy queda casi como una anécdota, pero el desarrollador que ha seguido de cerca la evolución de este estándar seguro que tiene una opinión sobre si era más conveniente o menos el escribir componentes en el contexto de ficheros HTML o en ficheros Javascript.

Ventajas de ES6 Modules respecto a HTML Imports: Se adapta mejor a las costumbres de la comunidad en cuanto al desarrollo frontend, ya que los desarrolladores están acostumbrados a escribir componentes en frameworks Javascript, dentro de archivos Javascript. Pero sobre todo, gracias a escribir en archivos Javascript es más fácil hacer convivir Web Components en el marco de cualquier proyecto web, ya que las mismas herramientas que se usan para llevar a producción código frontend son las que se usan para llevar a producción elementos personalizados del estándar Web Components.

Ventajas de HTML Imports respecto a ES6 Modules: La curva de aprendizaje para personas

que no tengan muchos conocimientos de programación era mucho más sencilla. Al escribirse todo en el contexto de un archivo HTML el procedimiento era mucho más similar a cómo se desarrolla una web tradicional. Al escribir los templates en archivos HTML, el editor ayuda mejor al desarrollador, con completado de código, resaltado de sintaxis, etc. En resumen, la experiencia de desarrollo es más sencilla y ayuda a que cualquier persona pueda crear sus propios componentes con bastantes menos conocimientos de Javascript.

Sea como sea, lo cierto es que Web Components con ES6 Modules crea muchas menos fricciones con el proceso de desarrollo de aplicaciones modernas y resulta mucho más sencillo integrar las ventajas de este estándar en el estado del desarrollo actual. Sin embargo, no se ha descartado definitivamente la incorporación de alguna herramienta o especificación que permita escribir los templates en archivos HTML, para poder definitivamente aunar las ventajas de HTML imports y ES6 Imports.

Compatibilidad con navegadores

Actualmente Web Components es un estándar Javascript soportado por todos los navegadores del mercado. Es decir, que los puedes usar tal cual, sin preocuparte de si tal navegador o tal otro los muestre bien o mal.

No obstante, hay navegadores como Internet Explorer que no reciben actualizaciones, dado que su propio fabricante ha retirado completamente el soporte a ese software. Esos navegadores no son compatibles con el estándar Web Components, ni lo serán nunca. Aún así, para los navegadores desactualizados es posible usar lo que llamamos un Polyfill.

Estado de compatibilidad en 2015

En el momento de publicación de este artículo, noviembre de 2015, el estándar estana todavía comenzando y no estaba totalmente implementado en los navegadores. En esta actualización (2022) el panorama ha cambiado mucho. El estado de compatibilidad en 2015 se podía resumir así:

Custom Elements Estado de la especificación: Working Draft W3C Soporte total en Chrome, Opera y Android Browser > 4.4.4 y Chrome para Android 46

HTML Templates Estado de la especificación: LS (Living Standard, por la Whatwg) Soporte total para todos los navegadores menos IE y Opera mini (Edge lo aplicará de manera inminente)

HTML Imports Estado de la especificación: Working Draft W3C Soporte total en Chrome, Opera y Android Browser > 44 y Chrome para Android 46

Shadow DOM Estado de la especificación: Working Draft W3C Soporte total en Chrome, Opera y Android Browser > 4.4 y Chrome para Android 46

Actualización del estado del estándar en septiembre de 2018

Hoy todos los navegadores soportan o están desarrollando el soporte a todas las

especificaciones del estándar, menos los HTML Imports, que están en estado de discusión. Además el soporte de los imports con Javascript es prácticamente total:

ES6 Modules (Módulos de ES2015) Estado de la especificación: Estándar consolidado Soporte total en todos los navegadores modernos (Inclusive Edge pero excluido Internet Explorer)

Como has podido ver, el que más soporte le da es Chrome. Otros navegadores como Firefox o Edge apenas están empezando, por lo que tendríamos que usar algún tipo de Polyfill. De todos modos, para saber el soporte en el momento actual una rápida consulta a Caniuse.com te ofrecerá la información actualizada.

Librerías para Web Components

En cuanto a librerías Javascript para producir Web Components hay que aclarar primero que realmente no hacen falta. Como has visto, los Web Components forman parte de un estándar, que está siendo discutido todavía en mayor media, pero es un estándar. Eso quiere decir que, más tarde o temprano, todos los navegadores lo tendrán en su "core" y podrás usarlo con tan solo usar Javascript estándar, sin necesidad de ninguna librería adicional.

No obstante, lo cierto es que diversos actores se han apresurado a presentar algunas librerías que nos permiten desarrollar hoy mismo con la tecnología de los Web Components. Te las resumimos a continuación:

Lit: Lit, antes conocido con el nombre de LitElement, es una micro-librería para el desarrollo de custom elements (pesa más o menos 5KB, por lo que su huella es mínima y sus ventajas muy elevadas). Lit está creada por un equipo de desarrollo dependiente de Google que trata de cubrir las necesidades que el estándar no llega a resolver, para conseguir una experienca de desarrollo de alto nivel. A nuestro modo de ver, Lit es la mejor alternativa para desarrollar componentes.

Polymer: Es una librería impulsada por Google que actualmente se encuentra solo en fase de mantenimiento. Fue la mejor alternativa para desarrollar Web Components, aunque el propio equipo de desarrollo de Polymer publicó más adelante LitElement / Lit, que mejora todavía las prestaciones de Polymer, el rendimiento y reduce el peso.

Stencil Es la librería desarrollada por el equipo de Ionic, que pretende ser lo más abierta posible, para que se pueda usar junto con cualquier stack de tecnologías moderno. Se integra muy bien con el novedoso Ionic 4, aunque lo podemos usar donde queramos.

X-Tag: Es la apuesta de Mozilla para la creación de Web Components, específicamente los custom elements, al alcance de todos los navegadores modernos. Actualizado: Hemos quitado el enlace porque no la recomendamos ya por no recibir actualizaciones ya desde hace varios años.

> Si te interesa este tema, te recomendamos un artículo completo dedicado a analizar las librerías basadas en el estándar de Web Components.

Conclusión

Hemos conocido únicamente la punta del iceberg en lo que respecta a web components, pero en breve analizaremos cada una de las partes de este estándar que va a revolucionar el desarrollo front end.

Muchos sitios están ya usando partes de las especificaciones de Web Components para producir sus interfaces e implementar funcionalidad del lado del cliente, como por ejemplo Youtube o Github. No dejes de usarlo porque no estén totalmente disponibles en los navegadores, puesto que puedes usar los mencionados polyfills para obtener esa compatibilidad. Te estarás preparando para el futuro.

En los próximos artículos vamos a recorrer cada una de las partes de Web Components para ver ejemplos sobre cómo se implementan, ya en la práctica. Comenzaremos viendo cómo se hace un Custom Element con Javascript nativo.

En el Manual de Web Components usamos generalmente código de Web Components V0, por lo que algunas cosas pueden hacerse de manera distinta, sobre todo en lo que se refiere a los HTML Imports. Sin embargo, la filosofía sigue siendo la misma, por lo que sigue siendo una buena lectura, aunque los ejemplos puedan no funcionar en navegadores actuales. De todos modos, si lo que quieres es desarrollar con Web Components lo ideal sería usar además alguna librería como Polymer o Stencil, ya que ye simplifican bastante el proceso de trabajo y te llevan mucho más lejos con menos esfuerzo.

Presentación de Web Components en vídeo

Si quieres saber mucho más y tienes disponible un rato para seguir aprendiendo, tenemos esta presentación de Web Components que seguro te gustará y te aportará mucha otra información de utilidad para poder comenzar a usar este estándar.

Para ver este vídeo es necesario visitar el artículo original en:
https://desarrolloweb.com/articulos/que-son-web-components.html

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *21/03/2022*
Disponible online en *https://desarrolloweb.com/articulos/que-son-web-components.html*

Especificaciones de Web Components

Vamos a explicar las cuatro especificaciones del estándar de los Web Components, que básicamente nos ofrecen nuevas etiquetas HTML y APIs Javascript para programación. Se pueden usar por separado, pero la verdadera potencia se da cuando se juntan para el desarrollo de nuevos elementos. Analizaremos de manera independiente cada una de las especificaciones que contiene, aportando ejemplos de uso con el código Javascript necesario para operar con ellas.

Especificación Custom Elements, explicaciones y práctica

Cómo realizar Custom Elements con Javascript básico, basando los estándares de los Web Components y sin usar ninguna librería.

En el anterior artículo conocimos qué son los Web Components y por qué estas especificaciones representan una novedad muy significativa en el mundo del desarrollo de interfaces de usuario y aplicaciones web con Javascript del lado del cliente en general.

Conocimos que una de las especificaciones es la de "Custom Elements", que quizás sea la más representativa porque es la que nos permite hacer nuevos elementos del HTML, que realizan funcionalidades personalizadas o presentan información de una nueva manera. Además, estos Custom Elements los puedes usar directamente en tu página, sin necesidad de programación. Para ser correctos el desarrollador que crea el custom element generalmente sí necesitará realizar tareas de programación, aunque todos aquellos que lo usen, lo harán mediante la expresión de una etiqueta, de manera declarativa.

Actualización: Hemos actualizado el código de este artículo para mostrar los ejemplos usando el estándar definitivo de Web Components V1, que es el que finalmente se ha instaurado, así que el texto está al día. El vídeo final no obstante pertenece a una versión antigua de Web Components.

Los Custom Elements no son algo totalmente ajeno al HTML tradicional. Si lo ves bien, un ejemplo de Custom Element que conocemos de toda la vida es una etiqueta SELECT, que

permite definir por medio de un código HTML sencillo un componente que tiene un comportamiento propio: lo pulsamos y nos permite ver varias opciones sobre las que podemos escoger una o varias. Esos SELECT los podemos agrupar con otros campos para hacer componentes mayores, como serían formularios. Obviamente, esos elementos existen desde toda la vida y no los habíamos entendido desde la perspectiva de los web components, pero nos hacen entender bien en qué se basa esta novedad de los custom elements.

Desarrollo con VanillaJS

En este y en los próximos artículos del manual de Web Components vamos a aprender a usar las diferentes especificaciones de web componets usando Javascript estándar, o sea, lo que se conoce en el argot de los desarrolladores como VanillaJS. Es importante porque veremos que para crear nuevos custom elements no necesito ninguna librería externa al Javascript que ya te soportan de manera nativa los navegadores.

No obstante, debemos insistir en que el desarrollo de web components será mucho más rápido si nos basamos en alguna librería que nos facilite ciertos procesos habituales, principalmente por hacer nuestras horas de desarrollo más productivas.

Las librerías como Polymer, LitElement o Stencil también nos aportan una capa adicional en relación a la compatibilidad, además de optimizar algunos procesos interesantes de cara a crear aplicaciones web más rápidas. Es por ello que los ejemplos de este manual tienen un valor más didáctico que otra cosa, ya que conocer los procesos de Javascript para la creación de Web Components ayudará mucho a la hora de entender cómo se realizan usando una librería por encima.

> Nota: Si te interesa saber más sobre estas bibliotecas de utilidad para desarrollo encima del estándar, te recomendamos la lectura del artículo Librerías Javascript basadas en el estándar Web Components.

Creamos nuestro primer Custom Element

Iremos poco a poco introduciéndonos en el mundo de los Custom Elements, creando elementos sencillos que no nos compliquen demasiado la vida inicialmente. Obviamente, de momento no serán los más atractivos funcional o estéticamente, pero facilitará el aprendizaje.

> Nota: Aunque en este artículo usaremos solamente la especificación de los Custom Elements, lo habitual es que se usen en conjunto varias, o todas, las especificaciones de Web Components.

Verás que sencillo es esto. Vamos a crear un componente llamado "dw-holamundo". Una vez definido lo usaremos de esta manera:

```
<dw-holamundo></dw-holamundo>
```

Ciertamente, la etiqueta "dw-holamundo" no existe en el HTML, pero gracias a los Web Components la podemos crear para usarla en nuestra página. Es tan sencillo como registrarla, por su nombre, mediante Javascript. Para ello necesitarenis haber definido una clase (de programación orientada a objetos) que implemente el componente. El código resumido podría parecerse algo a esto.

```
<script>
class DwHolamundo extends HTMLElement {
  // Implementación del componente
}
customElements.define('dw-holamundo', DwHolamundo);
</script>
```

Ya está! tenemos registrado nuestro primer custom element y sabemos usarlo. Si estabas esperando alguna complejidad adicional, sentimos decepcionarte.

Nuestro problema es que, tal cual hemos hecho nuestro elemento personalizado, no hace absolutamente nada. Tendremos que asignarle algún comportamiento, alguna forma, lo que sea, para que realmente tenga sentido nuestro primer ejemplo. Ahora veremos cómo mejorarlo pero antes quiero que veas el código fuente de este ejercicio:

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Primer Custom Element</title>
</head>
<body>
  <h1>Hola Mundo Web Components - Custom Element</h1>
    <p>No aparece nada porque no le hemos asignado ninguna función al Web Component</p>

  <!-- Uso mi Web Component -->
    <dw-holamundo></dw-holamundo>

  <script>
  class DwHolamundo extends HTMLElement {
    // Implementación del componente
  }
  customElements.define('dw-holamundo', DwHolamundo);
  </script>
</body>
</html>
```

¿No te está faltando algo? ¿Dónde está el script de la librería que da soporte a los web components? (es una pregunta con trampa)

La respuesta ya la debes saber, porque lo hemos comentado. Web Components es algo que funciona tal cual en los navegadores, Javascript puro y estándar. No obstante, quizás hayas pensado en la necesidad de usar un Polyfill, para que los navegadores antiguos puedan entender estas sentencias nuevas de Javascript.

Nota: El tema de los Polyfill lo veremos con detalle más adelante, pero de momento tenemos que comentar que este código lo vas a poder ejecutar sin problemas en cualquier navegador excepto Internet Explorer (Chrome, Firefox, Safari, Opera y Edge, que esta migrando a Chromium también posiblemente ya puedas en el momento de leer este texto). Dado que es un estándar del W3C los navegadores ya lo han implementado en bloque. Insistimos, es Javascript nativo. Sin embargo, si quieres extender soporte a todos los demás navegadores, aunque sean viejos, necesitas implementar un polyfill. Tienes más información sobre esto en el artículo qué son los Web Components.

Mejoramos el custom element para aplicar un comportamiento

Para aplicar algún comportamiento específico a nuestro primer web component se nos complica un poco el código, porque requerimos de varios pasos. Realmente son pocas sentencias con las que esperamos te familiarizarás rápidamente.

Básicamente, realizaremos estas acciones:

1. Decidir la clase molde que vamos a usar para partir como base para la especialización de nuestro componente. El la clase es algo propio de Javascript y es algo así como el molde con el que se va a crear el elemento personalizado. Podemos usar como molde otros elementos de HTML, o simplemente el prototipo de un elemento HTML genérico, que nos lo ofrece la clase que hemos usado para extender "HTMLElement"
2. Por medio del constructor de la clase, aplicar cualquier código Javascript que permita construir el componente con sus particularidades. Existen diversos eventos propios del estándar de los Custom Elements donde también puedo asignar comportamientos cuando se están creando los elementos, además del constructor. Por ejemplo el "connectedCallback", que se ejecuta cada vez que el custom element inserta en la página. Existen en el estándar varios eventos de estos para realizar acciones en distintos momentos del ciclo de vida de los web components. Los veremos más adelante con detalle

Ahora te muestro el código completo, solo la parte de Javascript que es la que ha cambiado, con estas tres acciones que acabamos de comentar.

```
class DwHolamundo extends HTMLElement {
  constructor() {
    super();
    this.textContent = 'Hola mundo!!!'
  }
}
customElements.define('dw-holamundo', DwHolamundo);
```

Tómate un tiempo para revisar el código y identifica estos tres bloques necesarios para poder definir nuestro elemento personalizado.

Verás que se define la clase del componente, que hemos llamado "DwHolamundo". Esta clase es la que implementará este custom element.

La clase se crea en base a otra clase, y por medio de la herencia (extends) conseguimos que nuestro componente especialice a otra etiqueta HTML existente. Ese prototipo en este caso, sobre el que partimos para definir nuestro custom element se llama "HTMLElement".

Ya en el código de implementación de la clase verás que se utiliza un método constructor. Los constructores sirven en programación orientada a objetos para resumir las tareas de inicialización de los objetos cuando se están creando. Es importante que el constructor llame a super(), que es una invocación al constructor de la clase padre, así se inicializa el componente genérico, y luego se realizan las acciones propias para inicializar nuestro nuevo custom element.

La única cosa propia de este componente que estamos agregando encima del elemento HTML genérico está en la línea `this.textContent = "Hola Mundo!";`. Por medio de este código solamente se está accediendo al elemento concreto que se está creando y asignando un texto a su propiedad "textContent", con lo que conseguiremos escribir algo dentro del contenido del Custom Element que se está definiendo.

El código que estoy usando lo tienes en GitHub en esta dirección: Código hola-mundo-web-components.html del repositorio de ejemplos de web components.

Ten en cuenta que este primer custom element lo hemos hecho muy sencillo, evitando tocar algunos temas importantes, que reservamos para próximos artículos del Manual de Web Components. Más que nada por dejar las cosas fáciles al principio y evitar más complejidades de las estrictamente necesarias para este "hola mundo".

Videotutorial de creación de Web Components con Javascript estándar

Para completar y ampliar estas explicaciones te recomendamos ver el siguiente vídeo en el que mostramos cómo se crean Web Components con Javascript nativo, es decir, sin usar ninguna librería más allá de las que nos ofrece el estándar de Javascript.

Por favor, ten en cuenta que este vídeo fue grabado con un código ligéramente diferente, ya que utilizaba la versión anterior de Web components que finalmente no vió la luz como un estándar definitivo. Por tanto, sigue las indicaciones en el texto del artículo que acabas de leer.

> Para ver este vídeo es necesario visitar el artículo original en:
> https://desarrolloweb.com/articulos/desarrollo-custom-elements-javascript-estandar.html

> Este artículo es obra de *Miguel Angel Alvarez*
> Fue publicado / actualizado en *04/12/2019*
> Disponible online en *https://desarrolloweb.com/articulos/desarrollo-custom-elements-javascript-estandar.html*

Extendiendo elementos en Web Components

Explicamos la modalidad de desarrollo de Web Components (Custom Elements) mediante extensión de elementos nativos del HTML.

En este artículo vamos a ver un sencillo ejemplo que nos permita explorar otra de las prestaciones de los Web Components. Se basa simplemente en la posibilidad de crear unos componentes en base a otros. Para ello usaremos las técnicas de creación de Custom Elements ya relatadas en el artículo anterior, agregando nuevo conocimiento práctico.

Es algo parecido a lo que conocemos como herencia en la Programación Orientada a Objetos, que nos permite extender las clases apoyándonos en el código de otras clases padre. En el mundo de los elementos de una web, nos permitirá construir nuevos elementos que son especializaciones de elementos que ya existen anteriormente.

Soporte actual en los navegadores a la extensión mediante "is"

ACTUALIZACIÓN: Este artículo explica una modalidad de desarrollo de componentes que no ha llegado a establecerse definitivamente en el estándar de Web Components, debido principalmente a la oposición de Apple.

Ofrece el desarrollo de un estilo de "Custom Elements" llamada "Customized Built-In Elements" que no te recomendamos, porque no está disponible en todas las plataformas y probablemente ya no funcione en el momento de realizar esta actualización. Por tanto, es mucho mejor desarrollarlos mediante la creación de clases, como se explicó en el artículo anterior o en los siguientes artículos de este manual.

Por tanto, te recomendamos saltar este artículo y aprender a realizar los llamados "Autonomous Custom Element" (los basados en clases, usando etiquetas nuevas con nombres inventados por ti mismo), que son los componentes que se pueden usar de manera autónoma, sin extender una etiqueta nativa como un botón. Quizás más adelante retomen esta propuesta y permitan exender etiquetas nativas, pero va a depender de si finalmente se apoya o no esta modalidad de creación de componentes.

Extender Custom Elements

Creemos que el concepto de extensión se debe entender, pero queremos insistir en ello porque es una de las principales filosofías de trabajo que nos traen los Web Components. Básicamente, este estándar se ha creado de manera que permita que los desarrolladores extiendan el HTML, creando aquellos nuevos elementos esenciales para realizar su tarea.

Esa capacidad de extensión no solo se da con elementos que existan actualmente en el HTML, sino también con otros custom elements. Es algo que vemos continuamente en todos los ámbitos.

En un coche tenemos varios elementos, ruedas, motor, suspensión y éstos a su vez están formados de otros elementos: por ejemplo el motor a base de un cilindros, pistones, bielas, cigueñal, etc. En el mundo del lenguaje de las personas tenemos las letras y éstas forman palabras y las palabras forman frases, etc.

En el mundo de la web podemos tener sistemas compuestos de varios elementos, como un cuadro de búsqueda, que está hecho de un botón y un campo de texto. Para construir una web podré usar botones y campos de texto, pero si lo que quiero hacer es un campo de búsqueda, usaré directamente el componente de búsqueda. Este componente funciona exactamente igual que si fuera un elemento suelto, es decir, tiene un nuevo tag (etiqueta) que usaré para insertarlo en una página. Por tanto, su complejidad y los componentes internos que necesite para representarse, quedarán encapsulados y protegidos del exterior.

Adicionalmente a crear unos componentes en base a la reunión de otros componentes, también podemos extender componentes ya creados, para dotarles de un comportamiento diferente de los componentes padre. Sobre este punto vamos a crear un ejemplo.

Ejemplo de elemento que extiende otro elemento

> Ten en cuenta que el siguiente código está desactualizado por no haberse concretado finalmente en el estándar esta modalidad de desarrollo de componentes. como se explicó ya.

En el momento en el que nos encontramos todavía es un poco pronto para hacer un ejemplo complejo, con el que podamos representar la capacidad de los Web Components, de asociarse unos con otros para crear elementos sofisticados. Para hacer todo esto necesitamos hablar antes de otras especificaciones de las 4 disponibles en este estándar. Así que nos vamos a conformar por ahora de hacer un elemento que extienda a otro.

Crearemos un tipo de botón nuevo, que especializa los botones que existen en el lenguaje HTML común. Nuestro botón se llama "botonholamundo". No hemos sido demasiado originales. Su comportamiento es tan básico como representarse con un texto ya definido (escrito dentro del botón) "Hola Mundo!".

Usando el botón: Para empezar vamos a ver cómo usaríamos este custom element, porque difiere un poco del ejemplo del artículo anterior.

```
<!-- Uso mi Web Component -->
<button is="dw-botonholamundo"></button>
```

Como puedes ver, ahora no estoy creando una nueva etiqueta personalizada, sino una

especialización de una etiqueta ya existente.

La etiqueta sobre la que he partido como base es BUTTON y le hemos colocado el atributo is="dw-botonholamundo" para indicar que no es un botón normal, sino uno que lo extiende y especializa.

Creando el custom element: Ahora vamos a ver cómo creamos el Javascript para generar ese elemento especializado. Realmente cambian pocas cosas a lo que ya conoces de los custom elements.

Para comenzar, al crear el prototipo no vamos a partir del prototipo genérico de elemento HTML, sino del prototipo de un elemento HTML botón: HTMLButtonElement.prototype.

```
var prototipo = Object.create(HTMLButtonElement.prototype);
```

Ahora asignamos un pequeño comportamiento a este botón, que especializa el botón genérico del HTML. Realmente solo le estamos cambiando el texto.

```
prototipo.createdCallback = function() {
    this.textContent = "Hola Mundo botón!";
};
```

A la hora de registrar el componente hay otro detalle fundamental para crear estos elementos que extienden otros y es el uso del atributo "extends" al que le hemos colocado el valor del elemento que está extendiendo: "button".

```
document.registerElement('dw-botonholamundo', {
    prototype: prototipo,
    extends: 'button'
});
```

Con eso es todo! Ya tenemos nuestro "dw-botonholamundo" listo, un botón que especializa y extiende los botones básicos que existen en el HTML tradicional.

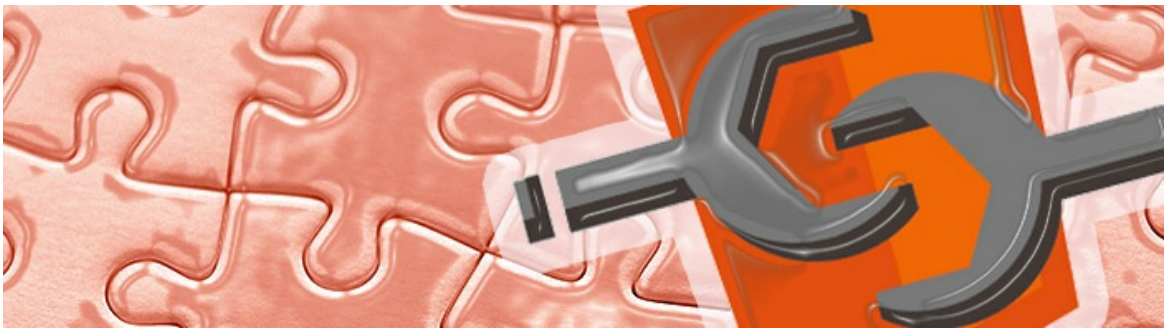
El código del ejemplo completo lo puedes encontrar en Github
https://github.com/midesweb/web-components-samples/blob/master/01_custom_elements/holamundo3.html

Esperamos que te haya gustado, prueba a extender otros elementos y hacer tus propios experimentos. Nosotros para seguir avanzando vamos a aprender en el siguiente artículo otra de las especificaciones de los Web Components como es el sistema de templates.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *13/04/2021*
Disponible online en *https://desarrolloweb.com/articulos/extendiendo-elementos-web-components.html*

Template con Web Components

Te explicamos a usar las etiquetas template, que pertenecen al estándar de Web Components. Veremos un ejemplo práctico de uso de un template como sistema de templating nativo en Javascript.



Estamos revisando poco a poco los distintos elementos de los Web Components en Javascript, el estándar de la W3C para el desarrollo del lado del cliente. En pasados artículos ya presentamos los Web Components y además vimos cómo se desarrollan Custom Elements.

Ahora le toca el turno al estándar Template, que nos permite crear plantillas que podemos completar con datos y presentar luego en el contenido de la página mediante Javascript. Es una novedad muy importante, ya disponible en casi todos los navegadores, por lo que deberíamos tenerlo presente para desarrollar con Javascript. En este artículo explicaremos en qué consiste y veremos ejemplos para entender su funcionamiento, siempre con Vanilla JS (Javascript nativo).

Por qué un sistema de templates estándar

Los sistemas de templates son uno de los componentes de aplicaciones web que nos facilitan el mantenimiento del código. Es una herramienta general que encontramos en diferentes lenguajes y es básica para separar la capa de presentación de la programación de procesos o la obtención de datos.

En Javascript hasta el momento no contábamos con ningún sistema para hacer templating, por lo que teníamos que usar alguna librería de terceros, como podría ser Handlebars JS. Afortunadamente para los desarrolladores la W3C ha sido consciente de esta necesidad en los estándares abiertos y ha creado un sistema de templates que los navegadores son capaces de interpretar de manera nativa.

Obviamente, al tratarse de un estándar disponible en los navegadores, el sistema de templates nativo siempre será más rápido que cualquier librería que podamos encontrar, facilitando un desarrollo homogéneo en todos los proyectos y más optimizado.

Como veremos a continuación, para utilizar sistema de templates requerimos usar dos componentes principales. Por un lado tendremos un HTML con el conjunto de elementos que contiene nuestra plantilla y por otro lado necesitaremos de un poco de Javascript para volcar datos dentro y presentarlos junto con el contenido de la página.

Etiqueta template

La parte de HTML para implementar el sistema de plantillas de los Web Components se escribe mediante la etiqueta TEMPLATE.

La etiqueta TEMPLATE es bastante especial, puesto que es la única etiqueta de contenido que no tiene una representación directa en el renderizado de la página. Dicho de otra manera, el navegador al leer una etiqueta TEMPLATE no la inserta en el contenido visible de la página, sino que la interpreta y la deja simplemente en memoria para que luego mediante Javascript se pueda utilizar. Por tanto, cuando en navegador encuentra un template no hace nada con él, aparte de leerlo, esperando que se use más adelante de alguna manera.

La forma de un template en HTML es como cualquier otro código HTML, sin nada en particular, aparte de estar englobada entre las etiquetas de apertura y cierre de la plantilla.

```
<template>
   <p>Esto es un template!!</p>
</template>
```

Javascript para usar un template

Como hemos mencionado, para usar un template aparecido entre el HTML de la página, necesitamos un poco de Javascript.

Básicamente tendremos que hacer tres pasos:

1. Acceder al template a partir de algún selector o mediante su id
2. Clonar el template en la memoria de Javascript
3. Inyectar el clon del template en el lugar donde se desee de la página

Esos tres pasos los vamos a ver representados a continuación en el siguiente código.

```
var template = document.querySelector('template').content;
var clone = template.cloneNode(true);
document.body.appendChild(clone);
```

Como ves, un template lo podemos acceder a través de un selector. Puedes usar document.querySelector() o incluso algo como document.getElementById(), si es que le pusiste un identificador al template.

```
var template = document.getElementById('id_template').content;
```

Como verás también, los templates tienen un atributo "content" que contiene el HTML de dentro de la etiqueta TEMPLATE.

El paso de realizar un clon es básicamente porque la idea de un template es que lo puedas insertar repetidas veces. Por ejemplo, podrías tener una lista de contactos y cada uno de esos

contactos podría tener un template para representarse (con el mismo template puedes representar todos los contactos, simplemente cargando en la plantilla datos distintos). Si tu lista tiene 20 contactos, ese template de un contacto lo repetirás 20 veces, cargando datos diferentes dentro de él, o sea, se realizarán 20 clones del template que se inyectarán en la página posteriormente. Luego veremos un ejemplo de template que contiene una repetición para mostrar diferentes elementos en una lista.

Código de una página que usa un template básico

Ahora podemos ver cómo sería una página elemental que está usando el template system nativo de Javascript.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Template simple</title>
</head>
<body>
    <h1>Template simple</h1>

    <template id="mitemplate">
        <p>Esto es un template!!</p>
    </template>
    <script>
        var template = document.getElementById('template').content;
        var clone = template.cloneNode(true);
        document.body.appendChild(clone);
    </script>
</body>
</html>
```

Este código es realmente poco útil por dos motivos. Primero porque generalmente vas a usar el sistema de templating junto con otros estándares de los Web Components. Pero segundo porque si querías presentar un texto directamente en la página podrías haberlo colocado tal cual en el cuerpo, en vez de accionar el sistema de plantillas y volcar ese contenido con Javascript. Paralelamente, como hemos dicho, es muy habitual contar con algún tipo de repetición que nos permita iterar y repetir varias veces el template en el cuerpo de la página.

Otra cosa que veremos en el ejemplo a continuación es que habitualmente dentro de un template podrás encontrar no solo código HTML, sino también código CSS que afectará a los elementos de este template.

Realizar una iteración para repetir el contenido de un template con un bucle

Ahora veremos un ejemplo más completo de uso de templates, en el que ya tenemos un bucle que recorre un array para repetir un template determinadas veces

En nuestro ejemplo vamos a hacer un listado de ciudades del mundo, con un encabezamiento que rotule el título de este template. Antes de comenzar este código vamos a aclarar dos puntos.

Estilos CSS son válidos en un template: Aparte de código HTML podrás incluir también código

CSS en un template. Este código lo colocas como siempre, con la etiqueta STYLE. La novedad es que estos estilos solo afectan al HTML del template, es decir, no salen para afuera y por tanto no afectan a otros elementos del cuerpo de la página. Este punto es muy interesante y a la vez muy útil porque permite que coloquemos estilos a etiquetas sin preocuparnos que éstos puedan trastocar el aspecto del resto de la página.

Unos templates contienen a otros: Cuando tienes un template más elaborado, puede que te encuentres en la necesidad de anidar templates. Por ejemplo en nuestro caso, identificamos dos bloques fundamentales:

1. Titular del template, el encabezado, que aparece una única vez
2. Cada una de las ciudades, que estará en un template que se repetirá una serie de veces, una vez por cada ciudad

Entendidos los puntos anteriores, serás capaz de interpretar bien este código.

```
<template id="templatesimple">
      <style>
      h1{
         color: red;
      }
      p{
         background-color: #ddd;
      }
      </style>
      <h1>Ciudades del mundo</h1>
      <template id="templateciudades">
         <p></p>
      </template>
   </template>
```

Ahora veamos el código de Javascript para usar ese template.

```
var ciudades = ["Madrid", "Barcelona", "Valencia"];

var template = document.querySelector('#templatesimple').content;
var p = template.querySelector("#templateciudades").content.querySelector("p");
ciudades.forEach(function(ciudad){
   var newP = p.cloneNode(true);
   newP.textContent = ciudad;
   template.appendChild(newP);
});
var clone = document.importNode(template, true);
document.body.appendChild(clone);
```

Como estás observando, en el código se accede a ambos templates de manera independiente. Además estamos recorriendo el array y realizando diferentes clones para cada ciudad. Tendremos un clon para todo el template general y otro clon para cada párrafo donde se va a representar cada ciudad.

> Nota: Casi sin lugar a dudas te parecerá algo complejo para la relativamente sencilla tarea que se está realizando. Sin embargo, librerías como Polymer te ayudan a simplificar bastante este código.

Conclusión

Hemos conocido el sistema de templates nativo de Javascript y hemos hecho un par de ejemplos para aclarar cómo se usa, en un template básico y en otro que incluye una repetición.

Ya hemos advertido que el verdadero uso de los templates se da cuando los usas en conjunto con otras herramientas del estándar de los web components, por ejemplo cuando el template forma parte de un Custom Element.

Aunque te pueda haber parecido complejo el código Javascript para usar un template tenemos que insistir en dos puntos:

1. El template que usas dentro de un Custom Element queda encapsulado en el custom element, por lo que lo puedes programar una vez y usar infinitas veces en uno o varios proyectos. O sea, al final toda la complejidad se queda en el código que vas a reutilizar sin preocuparte de nada
2. Existen librerías que nos permiten volcar de una manera más sencilla datos en los templates, que facilitarán crear templates con variables que se rellenan con propiedades de un objeto. Esa parte no la incluye el estándar de Javascript así que para hacer un código verdaderamente fácil de mantener se recomendaría usar alguna librería adicional como LitElement. LitElement es la evolución de la librería Polymer, creada por el mismo equipo pero mucho más ligera y rápida. LitElement permite sacarle partido a los template string literals de Javascript, creando un sistema de templating de mucho más alto nivel, que mejorará mucho la experiencia de desarrollo y la mantenibilidad de los proyectos.

En futuros artículos seguiremos usando el sistema de template de Web Components, por lo que podrás ver nuevos ejemplos en breve.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *02/12/2015*
Disponible online en *https://desarrolloweb.com/articulos/template-web-components-javascript.html*

Shadow DOM de Web Components

Explicaciones y ejemplos de uso de Shadow DOM el estándar de Web Components que nos sirve para crear elementos del DOM encapsulados en otros elementos de la página.

De todas las especificaciones de los Web Components, el estándar de la W3C para el desarrollo de componentes modulares y completamente reutilizables, Shadow DOM nos ofrece los mecanismos más importantes para que los módulos sean realmente autónomos e independientes de otros elementos de la página. En síntesis, Shadow DOM permite insertar elementos dentro del DOM de la página, pero sin exponerlos hacia afuera, de modo que no se

puedan tocar accidentalmente.

Cuando creamos un custom element a menudo éste necesita generar nuevos elementos, como botones, campos de texto, iconos, párrafos, que colocará debajo de su jerarquía. Todos esos elementos que cree el custom element podremos decir que le pertenecen directamente. El custom element dueño de sus elementos podrá, o no, ocultarlos de modo que no se puedan acceder desde fuera. Si se decide ocultar o encapsular esos elementos se usará el Shadow DOM. En ese caso, los elementos estarán físicamente en el DOM de la página, dependiendo únicamente del custom element que los ha generado y solo se podrán manipular por su dueño.

Básicamente, este DOM oculto a otros elementos de la página es el que nos permite aislar los componentes, produciendo la deseada encapsulación. El beneficio básico es que, al usar un custom element en la página, su contenido encapsulado no podrá interaccionar con otros elementos de fuera, evitando daños colaterales: Sobre todo, otros elementos de la página no podrán romper el estilo o comportamiento del custom element que usó el Shadow DOM.

> Nota: Quizás hayas experimentado alguna vez la desagradable situación que al insertar un plugin jQuery éste rompe estilos en tu página. O el componente no funciona porque otras partes de tu código interaccionan con él, u otros estilos CSS que tenías declarados de manera global. Todo esto está solucionado en los Web Components y mucho depende directamente de la especificación de Shadow DOM.

Crear Shadow DOM con Javascript

Ahora vamos a crear unos ejemplos básicos en los que usaremos la especificación de Shadow DOM para que, mediante Javascript, podamos crear e inyectar nuevos elementos en el DOM de la página, pero posibilitando que estén ocultos.

> Nota: Como otras especificaciones de los Web Components, podemos usar Shadow DOM sin necesidad de utilizarlo en conjunto con otras especificaciones como la de Custom Elements. Sin embargo, lo cierto es que cobra especial sentido y utilidad cuando usamos varias de las especificaciones en conjunto. Por ello, el siguiente ejemplo tiene sobre todo valor didáctico, pero no ilustra del todo su uso más habitual. Veremos ejemplos que usen el Shadow DOM junto con otras especificaciones más adelante, siendo que en este artículo en el último ejemplo mezclaremos la especificación de Template y la de Shadow DOM.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Shadow DOM</title>
<style>
        p{
            color: red;
        }
    </style>
</head>
<body>
    <div id="elem"></div>

    <script>
    //un elemento de la página
    var elem = document.querySelector("#elem");

    //convertimos ese elemento en un "shadow root"
    var shadow = elem.createShadowRoot();

    //cambiamos el contenido de ese elemento, esto será creado como "shadow DOM"
    shadow.innerHTML = "<p>Esto va al shadow DOM</p>";
    </script>
</body>
</html>
```

Para entender el ejemplo hemos colocado varios comentarios. No obstante, lo explicamos de nuevo. Primero tenemos un elemento de la página al que le hemos colocado un identificador, solamente para luego poder referirnos a él: id="elem".

Luego accedemos a ese elemento mediante Javascript:

```
var elem = document.querySelector("#elem");
```

Más tarde, cuando ya hemos decidido que queremos usar Shadow DOM, tenemos que producir una raíz donde se va a insertar todo elemento que vaya a estar encapsulado. A esa raíz se la conoce como "Shadow Root" y se genera con la siguiente instrucción:

```
var shadow = elem.createShadowRoot();
```

Por último tenemos que añadir nuevos elementos dentro del Shadow Root y eso lo podemos hacer mediante varios mecanismos. Un ejemplo sería editar su propiedad innerHTML.

```
shadow.innerHTML = "<p>Esto va al shadow DOM</p>";
```

Ese párrafo no se podrá tocar desde fuera. Por eso, si te fijas, en la cabecera teníamos un estilo que aplicaría a todos los párrafos de la página y, sin embargo, a la hora de la verdad, no está afectando al párrafo que hemos colocado dentro del Shadow Root.

Nota: Como ves en este ejemplo, insistimos nuevamente, no es necesario incluir ningún tipo de librería adicional para que el navegador entienda el Shadow DOM. Sin embargo, de momento esto solo funcionará en Chrome y Opera que son los que más se han apresurado a cumplir el estándar. Por supuesto, si usas el correspondiente polyfill podrás ver el ejemplo funcionando también en otros navegadores. Sobre el Polyfill hablaremos también en detalle en artículos futuros, aunque también tenemos unas notas interesantes que aportar más tarde.

Pseudo elemento CSS ::shadow para acceso al Shadow DOM

Obviamente, en ocasiones conviene poder saltarse la regla y aplicar estilos a elementos que están dentro de shadow DOM. Para ello tenemos un selector llamado ::shadow. En realidad es un pseudo elemento que se usa anteponiendo al selector que queramos aplicar dentro de un nodo Shadow Root.

Para que el párrafo anterior estuviera afectado por el CSS tendríamos que usar ::shadow de la siguiente manera.

```
<style>
   ::shadow p{
      color: red;
   }
</style>
```

Es una funcionalidad útil, aunque debes tener en cuenta que el pseudo elemento ::shadow se ha marcado como "depretated" (va a estar obsoleto y por tanto no se aplicará soporte en navegadores en adelante). No obstante, aunque este pseudoelemento sirva para saltarse el encapsulamiento entendemos que resulta bastante interesante, por lo que esperaríamos que se permita el uso de alguna alternativa similar para poder cubrir esta previsible necesidad.

Polyfill y Shadow DOM

Quizás la parte que resulta más complicado de simular meditante un polyfill, dentro de lo que respecta a los web components, es la de Shadow DOM. Por ello, el soporte a esta especificación de la W3C en los "polyfilled browsers" no es completo.

Por tanto, aunque el navegador muestre en la página aquellos elementos que se hayan colocado dentro de un Shadow Root, realmente no existirá esa mencionada encapsulación y se podrán tocar desde fuera, o alterar su aspecto con CSS definidos de manera global.

Ese motivo también hace que librerías como Polymer toquen el Shadow DOM, al menos por ahora, de una manera especial, no aportando todas las ventajas que tendría a priori en los navegadores que no lo soportan de manera nativa. Esto se hace para evitar afectar muy negativamente al rendimiento de las aplicaciones con Web Components y para que el comportamiento sea diferente en navegadores que lo implementan de manera nativa y los que no.

Shadow DOM que importamos desde un template

Antes de acabar, vamos a ver cómo alterar un poco nuestro ejemplo para que podamos usar un template, cuyo contenido se va a insertar como Shadow DOM. Recuerda que vimos templates de Web Components en un artículo anterior.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <title>Shadow DOM</title>
</head>
<body>
   <div id="elem"></div>

   <template>
      <p>Esto es un template!!</p>
   </template>

   <script>
   //accedo al template
   var template = document.querySelector('template').content;
   //clono el contenido del template
   var clone = template.cloneNode(true);

   //accedo a un elemento
   var elem = document.querySelector("#elem");
   //creo el shadow root
   var shadow = elem.createShadowRoot();
   //le añado el clon del template
    shadow.appendChild(clone);
   </script>
</body>
</html>
```

La diferencia es bien poca, simplemente tengo que acceder al template y clonar aquella parte que quiero usar dentro del Sadow DOM de otro elemento.

Luego ese clon del template es el que añado al Shadow Root con el método appendChild().

```
//creo el shadow root
var shadow = elem.createShadowRoot();
//le añado el clon del template
 shadow.appendChild(clone);
```

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *03/12/2015*
Disponible online en *https://desarrolloweb.com/articulos/shadow-dom-web-components.html*

Ejemplo de Custom Element Nativo en Web Components V1

En este artículo crearemos un completo elemento personalizado (custom element), que incluye trabajo con Shadow DOM y uso de slots, usando el estándar Javascript Web Components.

Estamos actualizando el Manual de Web Components a la versión más reciente y definitiva del estándar de Javascript, Web Components V1, ya que inicialmente se escribió para la versión previa (V0) que actualmente ya no está soportada. Además lo estamos ampliando para agregar nuevos ejemplos como el que nos ocupa en este artículo.

En este ejercicio vamos a realizar un componente nativo de un botón con animación, que además es capaz de atender a diversos estados. Básicamente es un botón que, cuando se pulsa, crea una pequeña animación y que además tiene un atributo llamado "status" que permite actualizar el aspecto del botón, de modo que de un poco de feedback visual al usuario.

En este ejercicio queremos repasar:

- El método de crear shadow DOM
- Novedades del método del ciclo de vida para el atributo "attributeChangedCallback"
- Cómo usar templates nativos de Javascript, gracias a ES6 Template Strings.
- Cómo usar slot para reutilizar el contenido del tag host

Realizaremos estos ejemplos de componentes usando únicamente Javascript nativo.

Creación de la clase para implementar el componente

Una de las novedades del estándar Web Components V1 es que usa clases (de programación orientada a objetos) para implementar los componentes. La clase puede extender cualquier elemento nativo del HTML, pero lo común será extender HTMLElement.

Nuestra clase tendrá este aspecto:

```
class BotonStatus extends HTMLElement {
    // implementar el componente
}
```

Luego registramos el componente con el nombre de la etiqueta, que tiene que contener un guión, y el nombre de la clase usada para implementarlo.

```
window.customElements.define('boton-status', BotonStatus);
```

Creación de un template con ES6 template strings

Podemos aprovechar una de las herramientas más útiles de ES6 como son los template strings para la creación del template del componente. Esto nos permite interpolar variables o propiedades del componente de una manera muy sencilla, que además produce un código muy legible.

Para facilitar la utilización del template dentro del componente me voy a apoyar en un método getter de Javascript, lo que me permitirá usar este template dentro de la clase del componente, tal como se usaría una propiedad común.

```
get template() {
  return `
    <style>
      div {
        display: inline-block;
        color: #fff;
        border-radius: 3px;
        padding: 10px;
        cursor:pointer;
        outline:none;
        animation-duration: 0.3s;
        animation-timing-function: ease-in;
        background-color: #000;
      }
      div:active{
        animation-name: anim;
      }
      @keyframes anim {
        0% {transform: scale(1);}
        10%, 40% {transform: scale(0.7) rotate(-1.5deg);}
        100% {transform: scale(1) rotate(0);}
      }
      .neutral {
        background-color: #888;
      }
      .danger {
        background-color: #d66;
      }
      .success {
        background-color: #3a6;
      }
    </style>
    <div class="${this.status}"><slot></slot></div>
  `;
}
```

La parte más interesante del template lo tenemos en la línea siguiente:

```
<div class="${this.status}"><slot></slot></div>
```

Aquí está interesante apreciar como se ha embutido el valor de la propiedad status del componente. Esta propiedad pertenece al objeto botón, una vez instanciado. Además en breve veremos cómo poblar esa propiedad con el valor introducido en el atributo "status", indicado al usar el componente.

Trabajo con slots

Además, muy interesante también es la etiqueta SLOT, que sirve para colocar en este punto el contenido que tenga la etiqueta del componente.

Por ejemplo, al usar el componente podemos tener algo como esto:

```
<boton-status>Haz clic aquí</boton-status>
```

El texto "Haz clic aquí" será lo que se introduzca dentro del template gracias a la etiqueta SLOT.

Cómo crear Shadow DOM

El constructor de la clase es el lugar más apropiado para construir el Shadow DOM del componente. Además es el lugar donde se deben inicializar las propiedades y donde podemos hacer otras cosas, como el acceso a los atributos seteados en la etiqueta del componente, para inicializar las propiedades con ellos.

En este caso vamos a tener el siguiente constructor:

```
constructor() {
  super();

  let currentStatus = this.getAttribute('status');
  if(currentStatus) {
    this.status = currentStatus;
  } else {
    this.status = 'neutral';
  }

  let shadowRoot = this.attachShadow({mode: 'open'});
  shadowRoot.innerHTML = this.template;
}
```

- El constructor debe llamar a super() como primer paso, para invocar a cualquier constructor de la clase padre. Esto es super importante para que todo funcione correctamente.
- Inicializamos la propiedad "this.status" con el valor del atributo status, que hemos obtenido con this.getAttribute('status'). Sin embargo, si el atributo no estaba definido en la etiqueta, simplemente lo inicializamos con un valor adecuado, en nuestro caso la cadena "neutral".
- Luego creamos el shadow DOM con el método this.attachShadow y agregamos el template dentro del shadowDOM gracias su propiedad inner.HTML.

Cómo reaccionar a los cambios en los atributos de la etiqueta del componente

Ahora nos queda hacer que nuestro componente sea reactivo y que pueda actualizar su estado cada vez que el atributo "status" de la etiqueta del componente cambie.

Esto lo tenemos que hacer con el método del ciclo de vida "attributeChangedCallback", que recibe el nombre del método, con sus valores anterior y nuevo.

```
attributeChangedCallback(attr, oldVal, newVal) {
  console.log('attributeChangedCallback');
  if(attr == 'status' && oldVal != newVal) {
    this.status = newVal;
    console.log(this.status);
    this.shadowRoot.innerHTML = this.template;
  }
}
```

En este ejemplo estamos reaccionando cuando el atributo actualizado sea "status" y cuando el contenido antiguo sea distinto que el nuevo seteado (aunque esta comprobación quizás sea un poco innecesaria, porque el método del ciclo de vida sólo debería invocarse cuando realmente haya cambios).

En caso que se detecten cambios lo que hacemos es actualizar el valor de la propiedad this.status y a continuación hacer que se renderice de nuevo el template, asignando la propiedad this.template a el innerHTML del shadowRoot. Esto provocará que se procese de nuevo el contenido de todo el template y se asigne como Shadow DOM.

Solo que nos falta un detalle muy importante, por motivos de optimización, el estándar de Web Components V1 nos obliga a crear un método getter llamado "observedAttributes", en el que tenemos que devolver un array de los atributos que en verdad se desean observar.

```
static get observedAttributes() {
  return ['status'];
}
```

De este modo, nuestro Javascript solamente estará pendiente del atributo "status", para invocar al attributeChangedCallback() solamente cuando éste cambie. De este modo conseguiremos que attributeChangedCallback() se ejecute solamente cuando es estrictamente necesario.

Usando el componente personalizado

Con esto hemos terminado nuestro web component, que será capaz de mostrarse con un estilo determinado, dependiendo de su atributo status (estilos válidos serán "neutral" en color gris, "success" en color verde y "danger" en color rojo, aparte de que se usará el color negro para cualquier status desconocido).

El componente lo ideal es que lo guardemos en un archivo con extensión .js, con el mismo nombre que el nombre del componente. En este caso sería "boton-status.js". El código completo sería el siguiente:

```
class BotonStatus extends HTMLElement {

  constructor() {
    super();

    let currentStatus = this.getAttribute('status');
    if(currentStatus) {
      this.status = currentStatus;
    } else {
```

```
      this.status = 'neutral';
    }

    let shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML = this.template;
  }

  static get observedAttributes() {
    return ['status'];
  }

  attributeChangedCallback(attr, oldVal, newVal) {
    console.log('attributeChangedCallback');
    if(attr == 'status' && oldVal != newVal) {
      this.status = newVal;
      console.log(this.status);
      this.shadowRoot.innerHTML = this.template;
    }
  }

  get template() {
    return `
      <style>
        div {
          display: inline-block;
          color: #fff;
          border-radius: 3px;
          padding: 10px;
          cursor:pointer;
          outline:none;
          animation-duration: 0.3s;
          animation-timing-function: ease-in;
          background-color: #000;
        }
        div:active{
          animation-name: anim;
        }
        @keyframes anim {
          0% {transform: scale(1);}
          10%, 40% {transform: scale(0.7) rotate(-1.5deg);}
          100% {transform: scale(1) rotate(0);}
        }
        .neutral {
          background-color: #888;
        }
        .danger {
          background-color: #d66;
        }
        .success {
          background-color: #3a6;
        }
      </style>
      <div class="${this.status}"><slot></slot></div>
    `;
  }

}

window.customElements.define('boton-status', BotonStatus);
```

Ahora, en cualquier página donde se pretenda usar lo tenemos que incluir como script:

```
<script src="boton-status.js"></script>
```

Y luego utilizar la etiqueta del componente, con los status que deseemos:

```
<boton-status>Haz clic aquí</boton-status>
<boton-status status="danger">No hagas clic aquí</boton-status>
<boton-status status="success">Haz clic aquí para tener éxito!</boton-status>
```

Eso es todo, hemos podido crear un componente nativo, completamente basado en las características de Web Components V1 y sin necesidad de usar ninguna librería Javascript.

Recuerda que puedes aprender mucho más en el Manual de Web Components.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *11/12/2019*
Disponible online en *https://desarrolloweb.com/articulos/custom-element-nativo-web-components*

Especificación Import en Web Components

Aquí explicamos la especificación HTML Import que se propuso, pero que finalmente no se apoyó y se descartó finalmente para el desarrollo de componentes con el estándar Web Components V1.

En artículos anteriores hemos relatado qué hay de nuevo con los Web Components y cómo van a revolucionar el desarrollo para la web. Es un nuevo estándar que nos trae cuatro especificaciones para poder construir, en resumen, nuevos elementos o componentes que nos ayudarán a extender el HTML a todos los niveles.

De entre todas las especificaciones ahora vamos a explicar la relacionada con los import, que es la última que nos quedaba por ver en el manual de los Web Components. Se trata de una especificación bastante sencilla que realmente usaremos poco a nivel de Javascript y más a nivel de HTML. Pero mejor vamos directamente con las explicaciones.

Especificación HTML Imports no apoyada

Comenzamos diciendo que esta especificación no fue apoyada de manera mayoritaria por las organizaciones que hay detrás del estándar de Web Components, de modo que no salió a la luz finalmente.

Web Components V0 la introdujo y durante mucho tiempo fue la manera que teníamos de

importar componentes, pero finalmente se pensó que era más adecuado usar los imports de Javascript. Se pensó que "*Si ya tenemos una manera de importar código con Javascript ES6, ¿Para qué necesitamos otra manera de hacer lo mismo desde HTML?*"

Finalmente, en Web Components V1, la especificación oficial que salió definitivamente a la luz y que hoy nos beneficiamos ya en todos los navegadores, se decidió importar siempre usando los imports de Javascript.

> Puedes encontrar más información sobre la evolución de las especificaciones Web Components en el artículo Qué son Web Components

Por qué necesitamos un "import" en HTML

Cuando desarrollamos tradicionalmente del lado del cliente nos valemos del lenguaje HTML para definir el contenido, del CSS para definir el aspecto y del Javascript para la funcionalidad o programación en general. Estos tres lenguajes se pueden escribir en el mismo documento HTML, pero generalmente su código se coloca en archivos independientes por diversos motivos.

Cuando quieres extender el HTML, por medio de lo que tradicionalmente se conoce como plugins (recuerda los plugin de jQuery), generalmente tienes que incluir diversos códigos por separado. Por una parte necesitaremos colocar un poco de HTML que es donde se va a embutir la funcionalidad del plugin, tendrás un script Javascript que colocarás en tu archivo .js del sitio o en el archivo plugins.js junto con otros plugins que quieras usar y por último un poco de CSS que generalmente colocarás en el archivo de estilos globales de tu sitio.

No existe una regla que sea totalmente obligatoria sobre cómo situar esos pedazos de código en tu página y a veces genera un poco de confusión, pero sobre todo dificulta la distribución de plugins y su mantenimiento. Los creadores de los plugins seguramente les gustaría poder decirte "mira, coloca este archivo aquí y no te preocupes por nada más". Pero no pueden porque esas tres porciones de código (HTML + CSS + JS) que tendrás que situar en tu proyecto en lugares diferentes dependiendo de la arquitectura de tu página.

Para ese caso concreto es el que se crea la nueva especificación de los "import". Se trata básicamente de incluir todo lo necesario para distribuir un componente en un único archivo .html. Aunque, a pesar de la extensión no tendrá solo el código HTML para funcionar, sino también sus estilos y su Javascript para darle vida.

En resumen, cuando quieras usar un Web Component en una página web no vas a tener que estar incluyendo los distintos códigos de los distintos lenguajes por separado, simplemente colocarás un único import a tu componente y ya lo tendrás listo para usar. ¿interesante, no?

> Nota: Hasta el momento no existen en HTML ninguna etiqueta que nos permita traer un código HTML que tengamos en otro documento. Esta tarea es algo normal en el día a día del desarrollo y seguro que la has realizado en alguna ocasión si programas en lenguajes como PHP por medio de las sentencias include o require. Todos los lenguajes tienen herramientas para traerse y usar código que hay en otros ficheros, la pregunta mejor sería ¿Cómo es que HTML no la tenía?
>
> En el pasado se trató de hacer uso de algún tipo de técnica que nos permitiera acceso a pedazos de HTML para mantener en un único lugar partes de la página que se repetían innúmeras veces a lo largo de todo un sitio web, como por ejemplo la cabecera o el pie. Ninguna de las alternativas se llegó a establecer por diversos motivos y nos veíamos

obligados a implementar esa funcionalidad del lado del servidor.

Los import de Web Components podrían suplir esta necesidad, pero la verdad es que no están pensados solo para ello. Realmente, como hemos dicho, están pensados para distribuir componentes en un único archivo que contiene todo el código necesario para que funcionen.

Como ya sabes lo que es un Custom Element, te aclarará saber que con un Import incluyes todo el código fuente necesario para que el navegador conozca uno de estos elementos personalizados. El import lo haces en la cabecera de la página y luego a lo largo de todo el cuerpo podrás usar la etiqueta que implementa el Custom Element todas las veces que necesites.

Cómo se usan los import

Para realizar un import en un documento HTML se usa la etiqueta LINK que ya existe desde hace tiempo en el lenguaje. Anteriormente LINK te servía únicamente para acceder a una hoja de estilos externa, en un archivo CSS que generalmente enlazas con todas las páginas de tu sitio. Ahora los LINK tendrán la posibilidad de definir su atributo rel="import" y con ello indicas que estás usando esta especificación de Web Components.

El código será como este:

```
<link rel="import" href="archivo_a_importar.html">
```

Esta etiqueta ahora permite enlazar con un archivo HTML que, como decimos, puede tener código no solo HTML, sino también CSS o incluso Javascript.

Ahora bien, hacer el import no implica que vayas a mostrar un HTML en un lugar concreto de la página!! No veas el import como si fuera un "include" de un HTML, sino como un enlace a un código que realmente no se va a mostrar donde está situado tu import, sino donde tú lo necesites. Como consecuencia, los import se suelen situar en el HEAD de la página. Allí podremos colocar todos los import que necesitemos en nuestro documento. Luego usaremos los import donde se necesiten, atendiendo a estas reglas fundamentales:

- El HTML de un import no se va a volcar en el sitio donde has definido tu etiqueta LINK. Osea, si colocas un import a un archivo que solo contiene código HTML será como si no colocases nada, ese HTML no aparecerá por ningún sitio, tendrás que volcarlo más tarde con Javascript para que aparezca donde quieras.
- El CSS que haya en un archivo que importes se incluirá como CSS de la página, sumándose al CSS global con la regla de la cascada. Ahora bien, si colocas CSS lo tendrás que incluir con las correspondientes etiquetas STYLE y barra STYLE.
- El Javascript que haya en un import se ejecutará directamente. Pero para colocar un script lo tendrás que hacer dentro de las etiquetas SCRIPT.

Ojo, esto es HTTP

Los import que te traes con la etiqueta LINK rel="import" se acceden mediante HTTP, que es el protocolo de transferencia de las páginas web. Esto es importante porque un import solo funcionará si tu página se accede a través de http://. En definitiva, para que funcionen tendrás que acceder al documento que realiza el import a través de un servidor web.

Para los que ya conocen Javascript de antes, es algo parecido a lo que pasa con las solicitudes Ajax. Realmente es que un import es como si fuera una solicitud Ajax para acceder a otro documento, que el navegador solicita a un servidor web a través del protocolo HTTP.

Contar con un servidor es sencillo y te vale cualquier servidor que puedas conseguir. En último término, si no sabes cómo proveerte de un servidor web, te vale subir los archivos a un espacio de hosting web. Pero si trabajas desde hace tiempo en el mundo web sabrás cómo conseguir un servidor en local, que es lo más adecuado para la etapa de desarrollo. En DesarrolloWeb.com tienes mucha información para conseguir esto.

Ejemplo Javascript para acceder a un import

Terminaremos este artículo con una pequeña práctica sobre Javascript para acceder a un import. El objetivo es no quedarnos solo en el conocimiento teórico, pero hay que remarcar que esta práctica no es realmente habitual en el día a día del desarrollo Web Components. Lo que vamos a hacer es acceder a un código HTML que está dentro de un import para presentarlo en un lugar de la página, pero recuerda que lo normal es que el import te sirva para incluir código en diferentes lenguajes que implementa un Custom Element.

Archivo que vamos a importar: archivo-importar.html Tenemos primero el archivo que vamos a importar, que contiene código HTML simplemente (insistimos que ésto no es lo más normal).

```
<p>Esto lo voy a importar de un archivo externo</p>
```

Archivo que realiza la importación: Ahora, desde cualquier archivo donde vayamos a importar un elemento, usamos en el HEAD la etiqueta LINK con rel="import" y luego en el href la ruta del archivo a importar.

Recuerda que por hacer el import, ese contenido HTML a importar no se mostrará dentro de la página. Luego en el código veremos el script que nos permitiría acceder al contenido del import para presentarlo en un lugar de la página.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>HTML Imports</title>
    <link rel="import" href="archivo-importar.html" id="miimport">
</head>
<body>

    <script>
        //accedo al elemento import con id=miimport
        var importado = document.querySelector("#miimport");
```

```
        //accedo al contenido del import
        var contenidoImport = importado.import;

        //accedo a un elemento en concreto del contenido del import
        var elementoDentroImport = contenidoImport.querySelector("p");

        //inyecto ese contenido del import en la página
        document.body.appendChild(elementoDentroImport);
    </script>
</body>
</html>
```

Como decimos, este código Javascript no deja de ser anecdótico. Se puede hacer esto que ves y el ejemplo funciona, pero no es lo más habitual. El código está comentado para que lo puedas analizar y entender como facilidad. No lo vamos a comentar más porque no es lo que realmente se espera de los import como hemos remarcado hasta la saciedad.

En el siguiente artículo realizaremos un ejemplo de import a un Custom Element que habrá en un archivo .html que contendrá tanto HTML como CSS como Javascript. En el próximo ejemplo, por tanto, sí podrás ver cuál es el uso que se ha pensado para dar a esta especificación de Web Components.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *10/08/2022*
Disponible online en *https://desarrolloweb.com/articulos/especificacion-import-web-components.html*

Aplicación de estilos CSS en Web Components

Cómo se aplica estilos CSS a los Custom Elements para definir el aspecto de los componentes. Explicando diversas técnicas disponibles para estilizar los componentes y permitir que se puedan personalizar desde fuera para adaptarlos al look & feel de cada sitio.

CSS Part

Qué son los CSS Shadow Parts, cómo usar el selector ::part() para conseguir aplicar estilos CSS globales a los custom elements con Shadow DOM.

CSS parts es una nueva tecnología que nos aporta el lenguaje CSS y que viene a solucionar una de las demandas más importantes de los desarrolladores de Web Components de los últimos años.

En este artículo te vamos a explicar cómo trabajar con CSS Part en nuestros custom elements y cómo escribir estilos globales CSS para que penetren en el shadow DOM.

El problema del CSS y el Shadow DOM

Una de las ventajas que nos aporta el desarrollo de Web Components es la encapsulación de los componentes en lo que se conoce como Shadow DOM. Este estándar Javascript permite que los componentes que desarrollamos estén aislados de otras partes de la página y que no sufran interferencias indeseadas.

> Realmente puede ser un poco impreciso decir que el shadow DOM deje completamente asislado al componente. Existen algunos estilos que, naturalmente, se aplican a todos los componentes, como por ejemplo el font-family o el color. Si estos estilos no están

redefinidos dentro del propio componente, se usarán los definidos en el CSS global. Pero este es el comportamiento habitual del HTML y el CSS, por lo que no debería de extrañarnos.

Esto soluciona diversos problemas habituales que existían anteriormente con otros sistemas como los plugins de jQuery, que cuando tenías varios en la página podrían romperse por interacciones entre ellos, o por el CSS que habíamos definido en la página a nivel global.

Con Shadow DOM los estilos CSS globales, definidos para toda la página, no pueden causar problemas en los componentes, lo que facilita mucho su reutilización. Sin embargo esto también es un arma de doble filo que ha causado conflictos de opiniones entre la comunidad de desarrolladores, especialmente aquellos que están acostumbrados a librerías sin encapsulación, como React, donde sí que se pueden usar los estilos globales dentro de los componentes.

Muchos desarrolladores acostumbrados también a trabajar con frameworks CSS como Bootstrap o Tailwind veían que al usar Web Components no podían reutilizar todas las clases del framework, lo que les resultaba en una experiencia inesperada y frustrante.

Obviamente, hay maneras de conseguir que los CSS globales consigan penetrar en los componentes, como las CSS Custom Properties (variables CSS), sin embargo hasta ahora se quedaban cortas o eran ineficaces.

- Por ejemplo, algunos desarrolladores decidían no usar Shadow DOM, lo que es perfectamente posible, pero con ello perdemos muchas de las ventajas del componente y se hace mucho más difícil trabajar con slots.
- Otros desarrolladores deciden usar @import en las declaraciones de los componentes, trayendo consigo una declaración de CSS global a cada custom element, pero eso es totalmente desaconsejado porque crea cantidades inmensas de código CSS que en realidad se está duplicando y mulplicando por la página, para cada vez que se usa el componente. Por supuesto, esta técnica influiría negativamente en el rendimiento de las aplicaciones.

CSS Parts mejora la aplicación de estilos en componentes con Shadow DOM

El Shadow DOM ha venido para quedarse y sus ventajas superan mucho a sus inconvenientes. Los avances en el estándar CSS y Web Components están dirigidos a permitir que la encapsulación de los Custom Elements sea un problema menor para los desarrolladores.

Aquí es donde CSS Part aparece, con el objetivo de mejorar sensiblemente las posibilidades del CSS global en componentes encapsulados con Shadow DOM. La idea es simplemente proporcionar un mecanismo para que los desarrolladores puedan crear CSS en los estilos globales y que este CSS pueda aplicarse en los componentes.

De este modo, podemos aplicar CSS desde fuera de los componentes con estilos tan complejos

como sea necesario, consiguiendo que aumente el grado de personalización de los componentes, sin tener que modificar el código de los propios componentes.

Conociendo el selector CSS ::part()

::part es un pseudo-elemento de CSS que permite aplicar estilos a otros componentes usados en la página, cuyos elementos pertenecen a un Shadow DOM.

Para que funcione deben entrar en juego dos requisitos:

- En los elementos del Shadow DOM, o sea, en el marcado HTML del template, debemos usar el atributo "part" indicando el nombre de esta parte del componente. Este atributo "part" se puede colocar en cualquier etiqueta y contendrá el valor del nombre de esta CSS part.
- En el CSS global, usaremos el pseudo-elemento ::part, indicando la parte del componente a la que nos estamos refiriendo para aplicar los estilos.

Ahora vamos a ver algunos ejemplos para que quede todo perfectamente claro.

Usando el atributo part en el template de nuestro custom element

Primero vamos a ver cómo se usa el atributo part, que debemos colocar en las etiquetas HTML sobre las que queremos que desde fuera se les puedan aplicar estilos CSS.

```
<button part="un_boton">Esto es un botón estilizable desde fuera</button>
```

Como puedes ver, definir un CSS part es tan sencillo como aplicar un simple atributo HTML. Cada sección del componente que quieras que se pueda estilizar desde fuera necesita que se le asigne este atributo, dando un nombre a esa part, en este caso sería "un_botton".

Aplicando estilos con el pseudo-elemento ::part

Ahora vamos a ver cómo se le puede aplicar estilos a ese botón. Vamos a suponer que ese botón se haya usado en un custom element llamado "mi-elemento". Entonces podrías usar este CSS para aplicar estilos al botón que habría dentro del componente <mi-elemento>.

```
mi-elemento::part(un_boton) {
  border: 1px solid orange;
  font-size: 2rem;
}
```

También podríamos definir el "part" sin indicar el nombre del componente, para que afecte a todos los CSS parts que haya con este mismo nombre.

```
::part(un_boton) {
```

```
  border: 1px solid orange;
  font-size: 2rem;
}
```

De este modo, si hay varios componentes en la página que tienen botones con part="un_boton", todos se verían afectados por las reglas de estilo anteriores. La ventaja en este caso es que unas mismas reglas de CSS se pueden además especificar en un único lugar, lo que reduce la cantidad de código que debe ser escrito e interpretado por el navegador.

> Es interesante mencionar que ::part() tiene más especificidad que los propios estilos definidos en el componente. Por ello, si los estilos globales definidos con ::part() colisionan con estilos definidos en el propio componente, ganarán los ::part() globales. Dicho de otro modo, si colocas un part en un componente lo estás exponiendo a cualquier posible manipulación de su estilo, incluso aunque tú hayas definido el CSS de una manera particular en el custom element. Como alternativa siempre podríamos usar !important, para conseguir vencer en los estilos definidos globalmente en un CSS Part, si es que fuera necesario para un componente en particular.

Estos selectores pueden complicarse un poco más usando otras pseudo-clases como :hover o :focus.

```
::part(un_boton):hover {
  border: 2px solid red;
  background-color: #666;
  color: #ffc;
}
```

Compatibilidad de CSS Parts con los navegadores actuales

Esta característica de las CSS y Web Components la podemos usar con total tranquilidad, dado que está actualmente disponible en todos los navegadores actuales.

> Por supuesto, Internet Explorer no la admite, pero ya sabemos que este navegador tiene una cuota de mercado residual, y ahora más desde que la propia Microsoft ha dejado de darle soporte.

Así pues gracias a ::part es posible ahora que desarrolladores reacios a usar Web Components se sientan más atraídos por este estándar Javascript.

Bonus: Cómo podrías usar clases Tailwind en componentes con Shadow DOM

Como hemos dicho, muchas de las reticencias al uso de Web Components provenían de la incapacidad de usar frameworks de CSS global, como Tailwind o Bootstrap. Gracias al selector ::part() ahora esto tendría una manera de solucionarse más fácilmente.

En el caso de Tailwind la solución más sencilla sería aplicar @apply para definir los estilos globales de un CSS part, utilizando en ella todas las clases de Tailwind que quieras que penetren en el shadow DOM.

```
::part('navegador') {
  @apply p-4 m-2 bg-slate-700 text-white;
}
```

De esta manera, todos los componentes donde le pongamos el atributo part="navegador" a uno de sus elementos, tendrán los estilos Tailwind que se están definiendo con el @apply. Es verdad que seguiríamos sin poder usar directamente las clases de Tailwind en el template del componente, pero sería perfectamente viable usarlas a través del estilo global en las parts donde las queramos aplicar.

Desarrollo basado en componentes

En los siguientes artículos nos dedicaremos a mostrar ejemplos que nos permitan entender cómo usar Web Components, con sus diversas especificaciones en conjunto, para el desarrollo web. De este modo pretendemos ofrecer ejemplos ilustrativos sobre las posibilidades de este nuevo estándar.

Ciclo de vida en los Web Components

Explicamos qué es el ciclo de vida en los componentes basados en el estándar Web Components. Cómo podemos definir callbacks para ejecutar código en los distintos momentos del ciclo de vida.

Los Web Components son una tecnología que ha cambiado el modo con el que se desarrollan las aplicaciones para navegadores. Ya hemos empezado a hablar sobre ellos en otras ocasiones en el Manual de Web Components, así que nos centraremos ahora en un tema específico y clave para los desarrolladores, como es su ciclo de vida.

El ciclo de vida es parte de la especificación de los Web Components y básicamente incluye diversos estados o situaciones por las que pueden pasar los Custom Elements a lo largo de su existencia en una página. Desde la instanciación de un componente, su inserción dentro del documento web, hasta su retirada del DOM, por poner varios ejemplos. Por medio de estos momentos del ciclo de vida por los que pasa cualquier componente, podemos personalizar su comportamiento en circunstancias muy concretas.

Básicamente el ciclo de vida está ahí para servir de utilidad a los desarrolladores de componentes, ya que permiten escribir código Javascript que será ejecutado cuando el componente va pasando por sus diferentes estados. Esta operación se realiza por medio de lo que se conoce como funciones "callback": funciones que son declaradas pero que no se ejecutan hasta que pasan ciertas cosas.

Estados del ciclo de vida de un componente

Para comenzar, vamos a describir los estados de un componente que definen su ciclo de vida:

- Created: Ocurre cuando el elemento se crea, pero ojo, no tiene que ver con que el elemento se muestre. Es como su instanciación en memoria de Javascript. Cada elemento de un tipo de custom element generado, lanza el método created.
- Attached o connected: Ocurre cuando un elemento que había en la memoria de Javascript se inyecta dentro de un documento, o sea, pasa a formar parte del DOM de la página.
- Detached o disconnected: Ocurre cuando un elemento se quita del DOM, se retira del documento y por tanto desaparece de la página.
- Atribute Changed: Ocurre cuando uno de sus atributos cambia de valor, siempre que el atributo esté siendo observado y por tanto se haya declarado como uno de sus "observedAttributes"

> Nota: Estos son los estados del ciclo de vida de los custom elements en Javascript estándar, aunque algunas librerías basadas en Web Components incorporan otros adicionales.

Ciclo de vida Custom Elements V0 vs V1

Existen dos especificaciones de Web Components, una experimental que estuvo vigente durante cierto tiempo, hasta que el estándar se estabilizó y quedó la especificación definitiva, que es la que apotaron todos los navegadores. V0 es la especificación inicial, que ahora mismo no aplica y V1 es la especificación que debemos usar.

Mencionamos esto porque el ciclo de vida de los componentes cambió de la especificación experimental (V0) a la especificación definitiva (V1), por lo que dependiendo de la antiguedad de un artículo puedes encontrar que se explica una u otra. De hecho, este artículo explicaba la especificación experimental y lo hemos actualizado ahora para cubrir la especificación definitiva.

Ciclo de vida de los componentes V1

Vamos a comenzar explicando cómo es la especificación definitiva, la que tenemos que usar en el desarrollo con Web Components. Esta espeficación contiene los métodos siguientes:

- constructor: Ejecutado al instanciar el componente
- conectedCallback: Ejecutado al insertar en el DOM
- disconnectedCallback Ejecutado al retirarlo del DOM
- attributeChangedCallback: Ejecutado al cambiar uno de sus atributos
- adoptedCallback: Ejecutado cuando el componente se mueve a otro documento.

Ahora vamos a ver cómo realizar un componente que realiza acciones cuando ocurren cosas en su ciclo de vida. No importa tanto la funcionalidad del componente como apreciar cómo se va produciendo la ejecución de los métodos correspondientes a medida que hacemos cosas con el componente.

Método constructor del componente

Empezamos con algo sencillo para aclarar cuándo se ejecuta el constructor. Primero aclarar que el constructor es un método tipico de las clases de programación orientada a objetos, pero también en cierto modo forma parte del ciclo de vida del componente.

Lo relevante del constructor es que se ejecuta cuando el elemento se instancia. Esto quiere decir que, aunque el elemento solamente se haya declarado como una variable de Javascript, el constructor se habrá puesto en funcionamiento. Incluso aunque el componente no aparezca en la página por ningún lugar.

Esto lo podemos ver en el siguiente pedazo de código, donde definimos un componente con su constructor.

```
class TestLifecycle extends HTMLElement {
    constructor() {
        super();
        console.log('Soy el constructor');
    }
}
customElements.define('test-lifecycle', TestLifecycle);

// Creamos un elemento basado en este custom element
document.createElement('test-lifecycle');
```

Al ejecutarse la línea del createElement, aunque el componente no se muestra todavía en la página, el constructor se pondrá en marcha.

> Es muy importante la llamada a super() en el construtor, antes de hacer cualquier otra cosa, para que se ejecute el constructor de la clase padre.

Inserciones y eliminaciones del componente en el DOM

Ahora vamos a ver cómo podemos detectar las inserciones en el DOM de este elemento, así como el momento en el que se elimina del DOM.

```
class TestLifecycle extends HTMLElement {
    constructor() {
        super();
        console.log('Soy el constructor');
    }
    connectedCallback() {
        console.log('El elemento se ha insertado en el DOM');
    }
    disconnectedCallback() {
        console.log('El elemento se ha retirado del DOM');
    }
}
customElements.define('test-lifecycle', TestLifecycle);

// Creamos un elemento
```

```
var element = document.createElement('test-lifecycle');
// Insertamos en el DOM
document.body.appendChild(element);
// Eliminamos del DOM
element.remove();
```

Por último vamos a aprender a realizar acciones cuando un atributo del elemento cambia. Esto incluye dos pasos importantes:

Declarar el atributo como observedAttributes

Para poder detecar cambios en los atributos, éstos tienen que ser observados. Para ello encontramos una propiedad del estándar que se llama observedAttributes.

Es una propiedad estática, que podemos crear con un getter y que debe contener un array con todos los nombres de los atributos que deben ser observados.

```
static get observedAttributes() {
    return ['dia', 'mes'];
}
```

Definición del método del ciclo de vida attributeChangedCallback

A continuación podemos definir ya el método attributeChangedCallback(), que se ejecutará cuando los atributos "dia" y "mes" cambien.

El método attributeChangedCallback tiene una particularidad importante y es que recibe tres parámetros:

- El nombre del atributo que ha cambiado
- El valor anterior del atributo antes del cambio
- El valor al que ha cambiado.

```
attributeChangedCallback(name, oldValue, newValue) {
    console.log(`Ha cambiado el atributo ${name}, que tenía el valor ${oldValue} y pasa a tener el valor ${newValue}`);
}
```

Para acabar vamos a ver un código de un componente que implementa todos los métodos del ciclo de vida que hemos visto y su uso con Javascript para crearlo, insertarlo en el DOM, cambiar un atributo un par de veces y por último retirarlo del DOM.

```
class TestLifecycle extends HTMLElement {
    constructor() {
        super();
        console.log('Soy el constructor');
    }

    connectedCallback() {
```

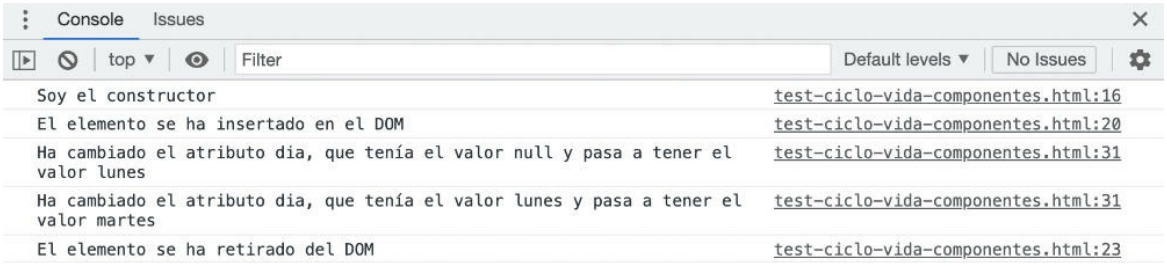
```
        console.log('El elemento se ha insertado en el DOM');
    }
    disconnectedCallback() {
        console.log('El elemento se ha retirado del DOM');
    }

    static get observedAttributes() {
        return ['dia'];
    }

    attributeChangedCallback(name, oldValue, newValue) {
        console.log(`Ha cambiado el atributo ${name}, que tenía el valor ${oldValue} y pasa a tener el valor ${newValue}`);
    }
}
customElements.define('test-lifecycle', TestLifecycle);

// Creamos un elemento
var element = document.createElement('test-lifecycle');
// Insertamos en el DOM
document.body.appendChild(element);
// Cambiamos un atributo que no existía antes
element.setAttribute('dia', 'lunes');
// Volvemos a cambiar el atributo
element.setAttribute('dia', 'martes');
// Eliminamos del DOM
element.remove();
```

Dado el código anterior, al ejecutarlo aparecerían los siguientes mensajes en la consola de Javascript del navegador.



Ciclo de vida en web components V0

Ahora vamos a ver toda otra serie de ejemplos que son muy similares a los que hemos visto en el artículo hasta ahora, pero que pertenecen a Web Components en su especificación experimental. Básicamente es lo mismo, pero algún elemento del ciclo de vida ha cambiado de nombre.

> Observarás otro cambio y es que se le están aplicando los métodos del ciclo de vida desde fuera del componente. Es perfectamente posible, ya que Javascript es muy laxo con respecto a la visibilidad y acceso a los métodos de las clases.

Asociar comportamientos a instantes del ciclo de vida de custom elements

Puedes ver todos esos estados son como si fueran eventos que ocurren durante la vida de un componente. Como a los eventos, seremos capaces de asociar funciones manejadoras, que se

encargan de producir comportamientos personalizados para cada suceso. En este caso llamamos a esas funciones con el término "calback", usado en el estándar como ahora podrás ver.

Ahora veremos el código del registro de un componente en el que usaremos los diversos métodos callback del ciclo de vida. Pero para entenderlo te sugerimos la lectura del artículo del estándar de los Custom Elements, en el que se explicaron ya muchas cosas que aquí vamos a dar por sabidas.

```
// Creo un nuevo objeto para registrar un componente, generando un nuevo prototipo
 var elemento = Object.create(HTMLElement.prototype);

 // Defino una función callback para el instante del ciclo de vida "created"
 elemento.createdCallback = function() {
  console.log('se ha creado un elemento');
 };

 // Defino una función callback para el instante del ciclo de vida "attached"
 elemento.attachedCallback = function() {
  console.log('se ha añadido un elemento al DOM');
 };

 // Defino una función callback para el instante del ciclo de vida "detached
 elemento.detachedCallback = function() {
  console.log('se ha retirado un elemento del DOM');
 };

 // Defino una función callback para el instante del ciclo de vida "attributeCanged"
 elemento.attributeChangedCallback = function(attr, oldVal, newVal) {
  console.log('Cambiado ', attr, ' al valor: ', newVal);
 };

 // Este es el código para registrar el componente, en el que indicamos su prototipo que acabamos de definir
 document.registerElement('ciclo-de-vida', {
    prototype: elemento
 });
```

Apreciando los estados del ciclo de vida

Solo con que uses un elemento, colocando la etiqueta HTML 'ciclo-de-vida' éste se generaría y se adjuntaría al DOM, con lo que ya se pondrán en marcha los métodos del ciclo de vida, con sus correspondientes console.log().

```
<ciclo-de-vida></ciclo-de-vida>
```

> Nota: Obviamente, para que ese elemento funcione debe conocerse previamente el elemento, por lo que el script para registrarlo visto en el punto anterior debería aparecer antes en el código HTML. (Mira al final el código completo del ejercicio).

Quizás con nuestro ejemplo te sorprenda que no observarás cambios en la página, porque el elemento del ejemplo no tiene template, pero sí deberías ver los mensajes si abres la consola Javascript.

- se ha creado un elemento
- se ha añadido un elemento al DOM

Para ver otros métodos del ciclo de vida necesitas el código de algunas funciones Javascript de manipulación del DOM. Para ello hemos colocado tres botones que invocan tres manipulaciones diferentes sobre el DOM, que provocarán nuevos mensajes a la consola de Javascript.

```
<button id="cambiaAtr">Cambia Atributo</button>
<button id="quitarDOM">quitar el elemento del DOM</button>
<button id="crearElemento">crear un elemento,  solo en memoria</button>
```

Esos eran los tres botones, a los que les colocamos tres manejadores de eventos para realizar cosas con elementos:

```
document.getElementById('cambiaAtr').addEventListener('click', function() {
  document.querySelector('ciclo-de-vida').setAttribute('data-test', 'test-value');
});
document.getElementById('quitarDOM').addEventListener('click', function() {
  document.body.removeChild(document.querySelector('ciclo-de-vida'));
});
document.getElementById('crearElemento').addEventListener('click', function() {
  document.createElement('ciclo-de-vida');
});
```

Código completo con el estándar Web Components V0

Eso es todo lo que necesitas para practicar con los mecanismos del ciclo de vida de los custom elements. Ahora para aclarar posibles dudas dejamos el código completo del ejercicio.

> Nota: Recuerda que, a pesar que esto sea todo Javascript nativo, solo funcionará para los navegadores que ya implementan el estándar de los Web Components. Para navegadores que aún no lo tienen disponible simplemente habría que usar el correspondiente polyfill, del que ya hemos hablado anteriormente en este manual.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Test del ciclo de vida de custom elements</title>
  <script>
  (function() {
  // Creo un nuevo objeto para registrar un componente, generando un nuevo prototipo
  var elemento = Object.create(HTMLElement.prototype);

  // Defino una función callback para el instante del ciclo de vida "created"
  elemento.createdCallback = function() {
    console.log('se ha creado un elemento');
  };

  // Defino una función callback para el instante del ciclo de vida "attached"
  elemento.attachedCallback = function() {
```

```
    console.log('se ha añadido un elemento al DOM');
  };

  // Defino una función callback para el instante del ciclo de vida "detached
  elemento.detachedCallback = function() {
    console.log('se ha retirado un elemento del DOM');
  };

  // Defino una función callback para el instante del ciclo de vida "attributeCanged"
  elemento.attributeChangedCallback = function(attr, oldVal, newVal) {
    console.log('Cambiado ', attr, ' al valor: ', newVal);
  };

  // Este es el código para registrar el componente, en el que indicamos su prototipo que acabamos de definir
  document.registerElement('ciclo-de-vida', {
      prototype: elemento
  });
  }());
  </script>
</head>
<body>
  <ciclo-de-vida></ciclo-de-vida>
<button id="cambiaAtr">Cambia Atributo</button>
<button id="quitarDOM">quitar el elemento del DOM</button>
<button id="crearElemento">crear un elemento,  solo en memoria</button>

  <script>
  window.onload = function() {
  document.getElementById('cambiaAtr').addEventListener('click', function() {
    document.querySelector('ciclo-de-vida').setAttribute('data-test', 'test-value');
  });
  document.getElementById('quitarDOM').addEventListener('click', function() {
    document.body.removeChild(document.querySelector('ciclo-de-vida'));
  });
  document.getElementById('crearElemento').addEventListener('click', function() {
    document.createElement('ciclo-de-vida');
  });
  }
  </script>
</body>
</html>
```

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *28/03/2022*
Disponible online en *https://desarrolloweb.com/articulos/ciclo-vida-web-components.html*

Librerías Javascript basadas en el estándar Web Components

Principales librerías Javascript basadas en Web Components, el estándar para creación de Custom Elements. Librerías que aprovechan las características avanzadas y estándar de Javascript y las posibilidades de los navegadores modernos.

Librerías Javascript hay cientos de ellas. Por suerte o por desgracia sale una nueva por mes!! Esto ha provocado la aparición de un término "framework fatigue", que define un poco la situación en la que nos encontramos en el panorama del desarrollo frontend.

En este artículo no quiero provocar todavía más fatiga a los desarrolladores Javascript, sino poner un poco en relevancia toda una nueva oleada de librerías que, a mi modo de ver, están haciendo cosas positivas por el hecho de basarse en los estándares.

Por qué es importante que una librería esté basada en todo lo nativo

Cuantas más cosas nativas use una librería o framework para el desarrollo de su funcionalidad menos código propietario tendrá que ejecutar el cliente web y con ello obtendremos varios beneficios:

- Se consumirá menos tiempo en la descarga
- Consumirá menos datos el dispositivo, algo que es importante en conexiones con redes móviles
- Consumirá menos tiempo de ejecución, porque no necesitará tanto Javascript para hacer las cosas
- Consumirá menos batería en el dispositivo
- Aumentará el rendimiento de las aplicaciones

Los puntos anteriores, por supuesto, dependen de la propia librería. Puesto que si la librería usa estándares pero es muy pesada porque carga cosas innecesarias en una aplicación, o si sus algoritmos requieren mucho procesamiento, quizás algunas partes como el rendimiento o el consumo de datos no se cumplan. Pero podemos decir que, si usa lo nativo es mucho más susceptible de optimizar todas esas parcelas.

Compatible con todos los frameworks

Pero además, hay otro punto fundamental que es que todas las librerías basadas en Web Components son "CROSS FRAMEWORK". Este es un concepto que quizás no se oye tanto, pero que resulta muy relevante para el momento que nos encontramos.

Cuando elegimos un framework para desarrollar, desde la primera línea de código estamos incorporando una fuerte dependencia en el proyecto. Esto provoca situaciones que en un futuro serán embarazosas:

- Si el framework escogido se actualiza y requiere cambios en la aplicación, nos obligará a

realizar migraciones.
- Si el framework se queda obsoleto (no solo que deje de funcionar, sino porque otro framework mejor aparezca y provoque que el nuestro pase de moda), requerirá construir la aplicación prácticamente desde cero para poder beneficiarse de los avances en la tecnología.

Las situaciones anteriores no son para nada extrañas en el mundo del desarrollo. Al contrario, son bastante habituales y cuando estamos desarrollando un proyecto sabemos que a buen seguro llegará un día que se nos planteen. Es por ello que, cualquier cosa que podamos hacer para mitigar esos problemas, debería ser vista con buenos ojos.

Lo bueno de cualquier pieza desarrollada basada en estándares es que funcionará en cualquier framework que tengas actualmente. Es decir, un custom element creado con Web Components funcionará en una aplicación React, Angular, Vue, etc. Cualquier Web Component podrá funcionar en cualquier librería, sin necesidad de cambio alguno, por lo que podrás seguir usándolos aunque cambies de proyecto, framework, librería, etc.

Librería basada en Web Components

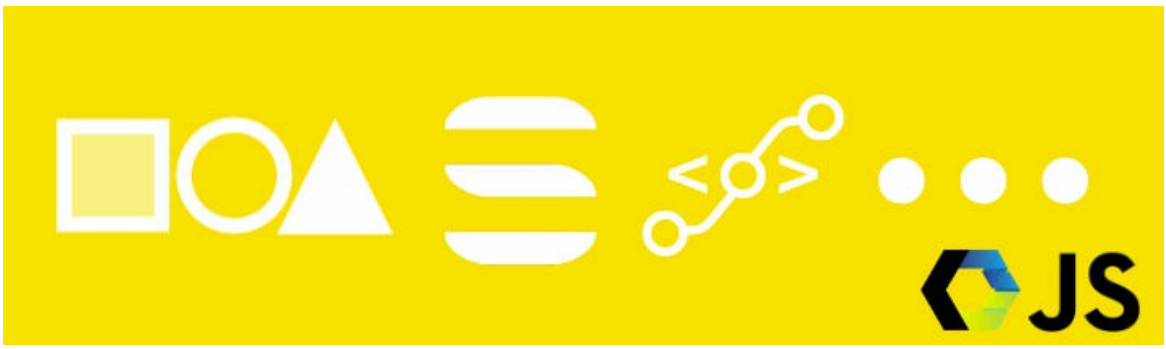
Todos los motivos anteriores hacen estar firmemente convencido de lo nativo en general y de Web Components en particular. Pero alguien se puede preguntar ¿Por qué necesito una librería para desarrollar basado en Web Components?

Lo cierto es que el estándar de Web Components, a día de hoy, no llega tan lejos como llegan muchas de las librerías o frameworks frontend más habituales. Web Components no ofrece algo tan útil como el data-binding o la programación reactiva. Claro que todas esas cosas se pueden incorporar, pero sería a base de tener que escribir mucho código propietario y es justamente algo que deseamos evitar.

Por eso, si estamos acostumbrados a la experiencia de desarrollo que nos ofrecen frameworks como Angular o librerías como React, necesitaremos una librería extra para poder ponernos al mismo nivel.

Sin embargo, las librerías serán generalmente mucho más ligeras y utilizarán muchas más cosas de lo nativo de Javascript. Por ello, las desventajas de usar una librería serán mucho menores.

Así que, si quieres obtener una experiencia de desarrollo ágil y amistosa y a la vez conseguir componentes nativos, que funcionan en cualquier aplicación con cualquier framework, te interesa usar una librería.



¿Qué librerías Web Components son las más destacadas?

Ahora veamos algunas librerías que podremos usar para desarrollar más próximos al estándar y al Javascript nativo, sin perder eficiencia ni productividad. Son librerías que se basan principalmente en los estándares, por lo que todo lo que ofrecen lo consiguen comprimir en muy pocas Kb, en torno a 7Kb solamente, o incluso menos!

LitElement: LitElement es la librería para desarrollo de custom elements creada por el equipo de Polymer (Google). Es muy cercana al estándar de Web Components y al Javascript nativo, de hecho tienes que usar lo nativo para la mayoría de las cosas, de ahí su reducido peso. Es extremadamente rápida, más que cualquier otro framework y librería popular, como React, Vue y por supuesto Angular.

Stencil: Esta librería está creada por el equipo de Ionic. Está enfocada en la creación de interfaces de usuario y de hecho todas las interfaces oficiales que se ofrecen para las aplicaciones Ionic están basadas en Stencil y por tanto en Web Components. Por supuesto, puedes usar Stencil para desarrollo de tus componentes, aunque no estés desarrollando aplicaciones Ionic y podrás usarlas en cualquier framework.

hyperHTML: Me parece otra librería con un enfoque acertado, ya que usa todo lo posible sobre Javascript nativo y las nuevas características de ES6 como los Template String Literals. Tiene un peso muy reducido y un elevado rendimiento.

Quizás los anteriores sean los mayores actores actualmente en el panorama de las librerías basadas en Web Components. Pero hay mucho más:

Polymer: Llevamos años hablando de Polymer y no podemos hacer este análisis sin dejar nombrarlo. Le tenemos que agradecer que haya abierto el camino para la creación del estándar de Web Components y su popularización. Sin embargo hoy Polymer ya no es una alternativa recomendable para comenzar un desarrollo. Sigue en mantenimiento, pero es mucho más pesada que su evolución (LitElement) y ofrece bastante menos rendimiento.

lighterhtml: Basado en hyperHTML recientemente ha aparecido lighterhtml, al que gana en rendimiento. Sin embargo, en este caso no nos encontramos tanto con una librería de componentes, sino con una librería pensada para hacer templates en Javascript, por lo que deberiamos compararla con Lit-HTML, el motor de templates de LitElement. Es una buena muestra de cómo las tecnologías de templates de Javascript están evolucionando hacia lo nativo y cómo lenguajes como JSX ofrecidos por React o su virtual DOM están quedando ya desfasados.

Más librerías capaces de ofrecer la misma utilidad con enfoques diferentes

A partir de aquí nos encontramos con muchas otras alternativas, la mayoría son micro-librerías que pesan en torno de 3KB, cada una con un enfoque particular. Ofrecen una manera simplificada de crear custom elements y permiten beneficiarnos de las herramientas deseables para disponer de una experiencia de desarrollo adecuada, como el mencionado data-binding, manejo del estado, trabajo con templates reactivos y cosas similares.

- Atomico que permite crear componentes de manera sencilla, por medio de funciones y hooks.
- Heresy: Que intenta ser muy parecido a React en su sintaxis.
- SlimJS: que se adelanta todavía más en el uso de tecnologías como los decoradores de ES7.
- Haunted: que ofrece un API como la de los Hooks de React pero para lit-HTML o HyperHTML.
- LWC: el framework de Salesforce basado en WebComponents.
- Omi: el framework Cross-framework.
- Litedom: con un conjunto de funcionalidades abrumador y un peso ultra reducido (retiramos el enlace porque lleva más de 3 años sin actualizarse).
- SkateJS: que consigue implementar diversos motores de templates, de modo que el desarrollador es capaz de escoger su favorito. También falta actualización desde hace años.
- X-Tag: Es la librería de Mozilla para el desarrollo con Web Components, pero también lleva años sin actualizaciones.

Como decía al principio, lejos de querer agobiar con tantas alternativas de librerías y frameworks, lo que quiero resaltar es el hecho de que la comunidad está firmemente concienciada de que el camino a seguir es el estándar. El surgimiento de tantos proyectos interesantes lo demuestra.

Por último, también es importante mencionar que las librerías y frameworks más establecidos en la actualidad, como es el caso de VueJS y Angular, están haciendo esfuerzos para transformarse y, al menos, ofecer al desarrollador la posibilidad de crear los componentes basadados en en estándar, en vez de en sus propias abstracciones. No tengo noticias de que React esté trabajando en el mismo camino. Su justificación es que su librería y el estándar son productos complementarios, mientras que Web Components es bueno en encapsulación, React está pensado para data-binding y templates reactivos. Sin embargo, en mi opinión estas responsabilidades están solapadas y mucho del código de React es innecesario desde que existe un estándar.

Conclusión sobre las librerías basadas en Web Componentes

Seguro que nos dejamos más de uno, pero los que hemos agrupado ya son una considerable lista de proyectos que tratan de sacar lo mejor de Javascript.

Por su peso, muchos de ellos realmente minúsculo, podemos implementarlos con la confianza de saber que nuestro código va a permanecer muy ligero. Eso sí, no veo mucha necesidad de mezclarlos entre ellos, ya que todos sirven para hacer las mismas cosas.

A mi personalmente me gustan los que tienen una sintaxis lo más parecida posible a Javascript y al propio estándar de Web Components. No me gustan tanto los que, por querer simplificar las cosas a los desarrolladores, proponen un código diferente a cómo haríamos las cosas sin usar ninguna librería.

En este sentido, estoy seguro que cualquier persona que aprenda LitElement podrá con muy poco esfuerzo desarrollar web components nativos sin usar librería alguna, pues el código que se tiene que hacer es extremadamente similar. En este sentido hyperHTML también me parece muy acertado. Destaqué además en este artículo a Stencil por venir de parte del equipo de Ionic, aunque el código que se crea es más parecido al de React y menos al estándar, lo que no me atrae tanto.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *21/03/2022*
Disponible online en *https://desarrolloweb.com/articulos/librerias-javascript-web-components*

Custom elements sencillos y sus HTML import

Ejemplos de custom elements sencillos y cómo usar la declaración import para incluirlos en una página web, de modo que se puedan usar.

Este artículo va a presentar un par de ejemplos sencillos de Web Components, muy elementales, que nos permitan asimilar un poco más las nuevas API para trabajo con este nuevo estándar desde Javascript.

En el artículo anterior presentamos la especificación import, pero lo que vimos es algo casi anecdótico, porque realmente un import no sirve para traerse el contenido HTML de un archivo externo, sino más bien para importar el código de nuevos componentes que podrías usar en cualquier documento HTML.

En esta ocasión mostraremos un uso más razonable de import, el de traerse el código de dos custom element creados para la ocasión. En resumen, usaremos las especificaciones:

Custom Elements HTML Import

Componente dw-date

Este componente simplemente muestra la fecha y hora del instante actual en la página. Se usa así:

```
<dw-date></dw-date>
```

Como comportamiento del componente, alí donde aparezca, se sustituirá por la fecha actual. Algo así como:

Mon Mar 28 2016 20:46:13 GMT-0300

El código del componente tiene esta forma: (con los comentarios y las explicaciones anteriores del Manual de Web Components estamos seguros que podrás entenderlo)

```
<script>
  //prototipo de elemento, a partir del elemento base
  var prototipo = Object.create(HTMLElement.prototype);

  // manejador del evento createdCallback,
  //ocurre cuando se instancia el componente
  prototipo.createdCallback = function() {
    //cambio el texto que hay en este elemento
    this.textContent = new Date().toString();
  };

  //Registro el componente "dw-date"
  document.registerElement('dw-date', {
    prototype: prototipo
  });
</script>
```

Componente romper-cadena

Ahora vamos a ver un segundo ejemplo de Web Component sencillo, pero esta vez un poco más útil. Este elemento permite romper un texto en una longitud en caracteres dada, pero sin romper las palabras.

El texto de la cadena original será el propio texto que haya en el elemento y además tendrá un atributo llamado "len" donde se marcará esa longitud máxima de caracteres para el recorte. Si no es posible romper en esa longitud, porque se rompa una palabra, se entrega la cadena hasta el espacio en blanco anterior.

Lo usaremos de esta forma:

```
<romper-cadena len="30">Esta cadena se va a romper en la longitud de 30 caracteres o menos</romper-cadena>
```

Ahora veamos el código de nuestro elemento.

```
<script>
(function() {
```

```
  function recortar(cadena, len) {
    if (cadena.length <= len) {
      return cadena;
    }
    var recortada = cadena.substr(0, len);
    return recortada.substr(0, Math.min(recortada.length, recortada.lastIndexOf(' '))) + '...';
  }

  //prototipo de elemento, a partir del elemento base
  var prototipo = Object.create(HTMLElement.prototype);

  //cuando se cree el elemento
  prototipo.createdCallback = function() {
    // guardo el contenido del elemento
    var cadena = this.textContent;
    //guardo la longitudo deseada
    var len = this.getAttribute('len');
    //lo recorto
    cadena = recortar(cadena, len);
    //cambio el texto que hay en este elemento
    this.textContent = cadena;
  };

  //Registro el componente "dw-date"
  document.registerElement('romper-cadena', {
    prototype: prototipo
  });
})();
</script>
```

Nota: en este elemento hemos realizado la envoltura del código mediante una función, ya que tiene sentido para encapsular la función Javascript usada "recortar()", así evitamos que su nombre colisione con el de otra función creada por aquella persona que use este componente. Ese patron se llama IIFE o closure.

Import: Usar estos componentes en una página

Ahora llega la parte de los import. Si quieres usar esos componentes en una página los tendrás que importar. El sistema es bien simple, gracias a la etiqueta IMPORT. Podrás ver el demo completo de estas dos etiquetas en este código:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Importar un webcomponent</title>
    <link rel="import" href="date.comp.html">
    <link rel="import" href="romper-cadena.comp.html">
</head>
<body>
    <dw-date></dw-date>
    <br>
    <romper-cadena len="30">Esta cadena se va a romper en la longitud de 30 caracteres o menos</romper-cadena>
    <br>
    <romper-cadena len="15">Este elemento me sirve para muchas cosas</romper-cadena>
</body>
</html>
```

Como puedes comprobar, después de los correspondientes import, somos capaces de usar las nuevas etiquetas creadas al registrar los componentes.

Recuerda que es una tecnología estándar, por lo que no necesitas una librería adicional Javascript para que funcione. Sin embargo, si queremos compatibilidad con navegadores que aún no implementan este estándar, lo ideal es usar el Polyfill, agregando el script en tu HEAD de webcomponents.js. Lo puedes hacer directamente desde el CDN:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/webcomponentsjs/0.7.21/webcomponents-lite.min.js"></script>
```

Esperamos que estos ejemplos te sirvan para seguir avanzando en el aprendizaje de Web Components, ilustrando con nuevos ejemplos la práctica con esta tecnología. Es interesante ver las cosas que se pueden conseguir con los custom elements más sencillos y cómo somos capaces de usarlos en una página web cualquiera.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *30/03/2016*
Disponible online en *https://desarrolloweb.com/articulos/custom-elements-html-import.html*

Practicando web components con VanillaJS

Ejemplo de desarrollo de un custom element con Web Components y Javascript nativo en el que usamos las 4 especificaciones del estándar.

En el Manual de Web Components hemos abordado diversas especificaciones del estándar que nos han mostrado por separado las posibilidades de este nuevo modelo de desarrollo Javascript. Pero la verdad es que estas tecnologías cobran más sentido si se usan en conjunto, así que vamos a juntarlo todo para experimentar el estándar de la mejor manera.

Usaremos únicamente Javascript estándar, sin apoyarnos en ninguna librería adicional, lo que se conoce como VanillaJS. Estos ejemplos funcionarán en cualquier navegador, siempre que incluyas el Polyfill, sin embargo, lo mejor es que los veas en Chrome que es cliente web que más camino andado tiene para implementar la tecnología.

Custom Element 'feedback-message'

Nuestro ejercicio consiste en crear un compoente que muestre un mensaje de feedback con un formato más atractivo estéticamente, que lo que sería un párrafo normal. Algo sencillo para comenzar, pero que nos permite trabajar con las 4 especificaciones:

- Custom Elements
- Templates
- Shadow DOM
- HTML Import

La etiqueta nueva que vamos a crear se llamará feedback-message. Realmente solo presenta un mensaje con un estilo especial, que podríamos haber conseguido con una simple clase de CSS, no obstante como práctica es interesante.

Lo usaremos de esta manera:

```
<feedback-message color="gray">Bienvenidos al Taller de Web Components!!</feedback-message>
```

Comenzamos con el archivo del componente. En un único archivo reunimos todo el HTML y el Javascript necesario para crear un elemento personalizado. Despiezamos sus dos partes principales:

- El template: donde colocas todo el HTML local que va a tener este componente. Ese template a menudo estará vacío de contenido y se cargará con Javascript en tiempo de ejecución. Es nuestro caso. Verás que el template tienen un único párrafo y está vacío. El contenido lo sacaremos del mismo documento HTML, lo que hay dentro de la etiqueta del custom element.
- El script: que registrará el componente y le dará su comportamiento especial. En ese script nos encargamos de hacer diversas tareas laboriosas y de código un tanto largo. Pero en realidad son pequeñas y simples acciones que están comentadas perfectamente.

> Nota: Para entender bien las diferentes acciones deberías leer los artículos sobre las distintas especificaciones de Web Components que hemos enlazado antes. Si te parece demasiado complicado piensa que cuando usas una librería este código se simplifica bastante, llegando a ser mucho más entendible y sobre todo de un mantenimiento sensiblemente más cómodo.

```
<template>
  <style>
    p{
      background-color: azure;
      font-family: trebuchet ms, tahoma, verdana, arial, sans-serif;
      padding: 10px;
    }
  </style>
  <p></p>
</template>
<script>
  // Creamos el prototipo de nuestro nuevo elemento
  // basándonos en el prototipo del elemento html genérico
  var prototipo = Object.create(HTMLElement.prototype);

  //Accedo al template definido antes
  var templateContent = document._currentScript.ownerDocument.querySelector('template').content;

  // definimos un callback a ejecutar al crear el elemento
```

```
    prototipo.createdCallback = function() {

        // guardo el contenido del elemento y luego lo borro
        var elementContent = this.innerHTML;
        this.innerHTML = "";

        // clono el template
        var clone = document.importNode(templateContent, true);

        // accedo al párrago de ese clon del template
        var parrafo = clone.querySelector("p");

        // cambio el contenido de ese párrafo con el contenido del elemento
        parrafo.innerHTML = elementContent;

        // si tiene el atributo color, lo coloco como estilo del párrafo
        if(this.hasAttribute("color")){
            var color = this.getAttribute('color');
            parrafo.style.color = color;
        }

        // Añado el clon del template como shadow dom
        this.createShadowRoot().appendChild(clone);

    };

    //Registro el componente con el nombre "feedback-message"
    document.registerElement('feedback-message', {
        prototype: prototipo
    });
</script>
```

Usar este componente

Ahora para usar este componente necesitamos hacer dos cosas:

1. El import del código del componente, que deberás poner en cada página que necesite usarlo
2. Colocar la etiqueta nueva que has creado

Además opcionalmente podrías usar el Polyfill de Web Components, que nos permitirá usarlo en todos los navegadores modernos.

```
<!DOCTYPE html>
<html lang="ee">
<head>
    <meta charset="UTF-8">
    <script src="../bower_components/webcomponentsjs/webcomponents.js"></script>
    <link rel="import" href="feedback-message-comp.html">
    <title>Mensaje de Feedback con Web Components y VanillaJS</title>
</head>
<body>
    <h1>Un mensaje de Feedbak</h1>
    <p>Debajo de este párrafo aparece el mensaje de Feedback generado con un custom element.</p>

    <!-- Uso mi Web Component -->
    <feedback-message color="gray">Bienvenidos al Taller de Web Components!!</feedback-message>
</body>
</html>
```

El import lo colocas generalmente en la cabecera, con el href hacia la ruta donde se encuentra el código del componente.

Nota: Si tienes muchos import es una práctica habitual crear un import único y que éste sea el que haga toda la lista de los muchos import que puedas estar usando.

Otra cosa que encuentras en el HEAD es el Javascript que carga el polyfill de compatibilidad webcomponents.js.

El uso del componente lo tienes más abajo, en la etiqueta nueva que hemos creado al registrar el componente.

Al ejecutarlo deberías ver el mensaje de feedback con un color de fondo azul y el texto de color gris. Podrías cambiar el color del texto simplemente cambiando el valor del atributo "color".

```
<feedback-message color="purple">Este es otro mensaje de feedback</feedback-message>
```

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *07/04/2016*
Disponible online en *https://desarrolloweb.com/articulos/practica-webcomponents-vanillajs.html*

SEO en Web Components

Qué hay del SEO en las aplicaciones y sitios web realizados con Web Components? Librerías como Polymer, LitElement o Lit ¿Son buenas para tener un correcto posicionamiento en buscadores?

Hemos rebautizado este artículo como SEO en Web Components ya que el contenido que explica lo podemos aplicar transversalmente a todas las librerías basadas en Web Components. Originalmente se escribió con Polymer en la cabeza, porque era la librería existente en ese momento, pero lo mismo lo podemos aplicar a cualquier librería basada en Web Components, ya sea Lit, LitElement, Atomico, etc.

En este artículo voy a hablar del posicionamiento web en buscadores (SEO) en aplicaciones y sitios web realizados con Web Components. Espero que sirvan de ayuda a los interesados, aunque advierto que es un artículo en base a impresiones y experiencias, así que encontrarás no solo datos objetivos y rigurosos, sino también opiniones y consejos diversos. Además, debes tener en cuenta que quizás con el tiempo algunas de las cosas que estoy comentando evolucionen de distintas maneras.

El presente texto viene como respuesta a un estudiante de los cursos de Polymer de EscuelaIT, que me preguntaba:

¿Existe alguna forma de que me indexen el contenido de una web hecha con el sistema de routing de polymer?

Lo primero es avisar que Google sí que indexa páginas que usan el sistema de routing de Polymer. Este punto lo trataré en último lugar en el presente artículo. Pero te recomiendo leer mi respuesta completa, en la que he ampliado algo el ámbito de la pregunta y hablaré sobre SEO en Polymer en general.

Web Components son "SEO Friendly"

Aunque se me preguntó específicamente por aplicaciones que usan el sistema de routing de Polymer, hay que aclarar primero para las personas que puedan leer este texto que el ámbito de los proyecos candidatos a acometer con Polymer es mucho más ámplio que las aplicaciones SPA que usan el sistema de routing.

Con Polymer puedes hacer custom elements de Web Components, como nueva etiquetas que extienden el HTML estándar que entienden los navegadores, que podrías usar en cualquier tipo de sitio. Por tanto, podrías perfectamente usar Polymer en un sitio común, basado en un CMS que posicione tan bien como WordPress, o cualquier tipo de sitio que esté optimizado para buscadores.

Esto quiere decir que la discusión sobre si Polymer es bueno o no para el SEO es un poco ambigua, pues depende de cómo sea el sitio donde estés aplicando Polymer y no de la librería en sí o de Web Components en general.

Lo ideal es el Server Side Rendering

En el tema de SEO lo ideal es que el sitio tenga “server side rendering”. O sea, construir la página del lado del servidor y entregar contenido original en cada URL. Server side rendering es como funciona una página creada con PHP o cualquier lenguaje del lado del servidor, en la que el HTML se construye en el servidor y se manda contenido específico de cada página al cliente. Así es como funciona un CMS por lo general y la mayoría de sitios de la Web, sobre todo los estáticos.

Sin embargo, las aplicaciones de gestión modernas, basadas en web, servicios, paneles de control, etc. cada vez están basándose más en el modelo de las páginas conocidas como SPA. En ellas se usa un sistema de routing basado en Javascript. El navegador se encarga de procesar la dirección a la que se está accediendo y traer los datos con los que construir por él

mismo el HTML. En estos escenarios ya no tenemos el comportamiento ideal, el mencionado server side rendering.

Las páginas que usan el routing de Polymer, o cualquier sistema de routing de los frameworks Javascript con los que se suelen construir las SPA, son siempre en realidad el mismo index.html de la home, junto con un Javascript. Las rutas se tratan desde ese Javascript, para mostrar unos u otros componentes. Como no tienes el contenido original en cada ruta de tu sitio, sino que ese contenido viene generado desde Javascript, no es tan bueno para SEO.

Hay soluciones encaminadas a resolver este problema, implementadas en los frameworks Javascript más avanzados, como Angular, VueJS o los basados en la librería React. Gracias a una capa adicional que se ejecuta del lado del servidor, es posible construir en el lado del servidor una página con contenido original para cada ruta, de modo que Google encuentra contenido original y específico para cada ruta y es capaz de indexar mejor el sitio web.

A día de hoy, no he encontrado una solución oficial para facilitar el Server Side Rendering en una app hecha con el sistema de routing de Polymer. Pero quizás no lo necesites y además hay alternativas.

Usar Web Components en sitios clásicos

La primera sería usar una librería basada en Web Components en un sitio común, con programación del servidor, capaz de producir un HTML único para cada URL. Es justamente lo que comentaba en el punto anterior de este artículo.

Usar otro framework que te facilite el server side rendering

Otra alternativa sería usar con otro framework que sí tenga resuelto el tema del server side rendering (SSR), como los mencionados antes (Angular, React, etc).

> La libería Lit (que es la evolución de LitElement, que a su vez era la evolución de Polymer) ya tiene solucionada la parte del server side rendering, por lo que podríamos usar esta renderización del lado del servidor en aplicaciones SPA que usan solamente Web Components.

Usar Angular u otro framework para poder implementar el SSR podría parecer un poco descabellado, ya que lo que soluciona Web Components se superpone en muchos de los casos al código específico introducido por Angular o React, por lo que puede parecer poco óptimo cargar dos librerías distintas que tienen funcionalidades solapadas. Sin embargo hoy resulta una posibilidad muy viable, ya que en navegadores modernos donde se implementa Web Components 1.0 el peso que tiene la librería no pasa de unos pocos KB (5KB aproximadamente en Lit o LitElement).

Por todo ello, no es un problema usar librerías de Web Components junto con otras no basadas en Web Components. Pero tampoco es algo que necesites, porque todo lo que puedes hacer con

grandes frameworks lo puedes implementar con pequelas librerías o micro-librerías. Incluso en el caso de usar Lit, la propia librería tiene ya disponible la parte del SSR.

> Nota: Ahora el dato no lo recuerdo bien, pero en una conferencia de presentación de Polymer 2.0 decían que estaba en torno a los 5 KB. Claro, que el navegador debe contar con soporte a Web Components 1.0, y puede que no sea siempre así, pero afortunadamente ahora todos los navegadores se han puesto de acuerdo con la especificación y es cuestión de muy poco tiempo que la mayoría de los browsers dispongan de soporte completo a Web Components y así eviten de cargar el correspondiente Polyfill, que es lo que más ocupa realmente en KB y tiempo de procesamiento para el arranque de la aplicación.

Usar herramientas de terceros

Puedes probar con herramientas de terceros. No las he probado, pero en este tema la comunidad está trabajando para hacer algunas soluciones. Por ejemplo tienes Server Components o artículos de blogs que analizan cómo puedes crear tú mismo el sistema de renderizado del lado del servidor para tus aplicaciones.

Me gustaría ver pronto soluciones oficiales del propio Google, ya que el propio Polymer es una apuesta de la empresa del buscador, entiendo que deberían aportar ellos mismos las mejores soluciones para que las aplicaciones Polymer posicionen de manera muy optimizada.

Google sí que indexa páginas con el sistema de routing de Polymer

Como conclusión de este artículo se ha de decir que en realidad Google sí que está indexando páginas de aplicaciones SPA realizadas usando el sistema de routing de Polymer. Y lo que es más maravilloso, está teniendo en cuenta incluso datos que llegan desde la base de datos Firebase.

No sé precisar desde cuándo estamos en esta situación, pero cualquiera de nosotros lo podría comprobar haciendo una búsqueda por "site:app.desarrolloweb.com".

Esa búsqueda te dará una serie de páginas que encuentras en el app.desarrolloweb.com. Esa web está basada en el sistema de routing de Polymer y todos los datos se encuentran en una base de datos Firebase, por lo que no hay apenas ningún contenido escrito "a pelo" en el HTML, sino todo viene desde Javascript y del acceso a servicios externos.

Lo que no sé es el valor que tenga en términos de posicionamiento el contenido detectado en los diferentes componentes de cada ruta, o el texto que venga desde Firebase. Es decir, no sé si Google "trata de manera generosa o rácana" a los contenidos que encuentra en el Javascript de estas aplicaciones, en comparación con el contenido que recibe en el HTML de sitios tradicionales. Me da que no es demasiado generoso en términos de posicionamiento, pero el caso es que sí que lo indexa, lo que ya es una buena noticia. Además, tenemos que considerar que las apps SPA generalmente no tienen la cantidad de texto ideal para que un artículo posicione correctamente. Según los expertos de SEO y el propio Google, un artículo se considera de calidad a partir de 700 a 1000 palabras, lo que dista mucho del contenido original

que se sirve desde nuestra app Polymer para la descarga de manuales.

Con esto es todo. Creo que habré dado un poco de luz a las dudas de las personas que se interesan por el SEO en Polymer. Como decía, seguramente en los próximos meses o años encontremos novedades interesantes y relevantes en cuanto a este importante tema, así que habrá que estar atentos. Pero el hecho de que Google sí que indexe aplicaciones en Polymer es sin duda una realidad.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *25/03/2022*
Disponible online en *https://desarrolloweb.com/articulos/seo-polymer.html*