



1 FEBRUARY 2017 / JAVASCRIPT

# Cómo crear un WebComponent de forma nativa

Gracias a los `custom elements` cualquier desarrollador web puede crear sus propios componentes HTML. Es decir, un nuevo `tag HTML` que "esconde" dentro su propia funcionalidad, diseño y layout.

En este artículo te voy a contar como puedes crear tu propio **Web Component desde cero, sin necesidad de utilizar librerías adicionales**, únicamente utilizando JavaScript, HTML y CSS.

- [Demo en JSBin](#)
- [Código en Github](#)



## Definiendo el Custom Element

Con la [Custom Elements v1 Spec](#) el navegador ahora tiene el objeto o clase **HTMLElement** del cual podemos "extender" y crear nuestros *web components*.

No está aún soportado por todos, pero poco a poco los fabricantes lo están incorporando. [En Can I Use puedes ver las versiones que lo soportan en este momento](#)

Con el objeto global **customElement** le indicamos al navegador que nueva *tag* estamos creando y a que clase heredada de `HTMLElement` hacemos referencia.

Dejamos la teoría por ahora y nos ponemos manos a la obra. Vamos a crear un *Web Component* reutilizable que represente un botón de compra. Empezamos por el código JavaScript:

```
// Aquí iría el código del elemento
// Eventos, funciones, etc...
}

window.customElement.define('sell-button', SellButton);
```

De esta manera podríamos usar en nuestro HTML el siguiente tag:

```
<sell-button></sell-button>
```

Pero de momento no veríamos nada. Así que vamos a empezar a añadir cosas.

`HTMLElement` posee métodos de ciclo de vida, como también tienen librerías como React. Son los siguientes:

- **connectedCallback:** Se llama cada vez que el elemento se inserta en el DOM. Aquí podemos hacer llamadas AJAX para pedir datos, configurar cosas, etc.. Funcionaría similar al *componentWillMount* de React.js
- **disconnectedCallback:** Este método se llamaría cuando el componente es eliminado del DOM. Su comparación con React sería el método *componentWillUnmount*
- **attributeChangedCallback:** Este otro método se llamaría cuando se añadiera un nuevo atributo, se actualizase o se eliminara. Algo similar a *componentWillReceiveProps*, *shouldComponentUpdate*, *componentDidUpdate* en React.

Entonces en `connectedCallback` vamos a añadir un poco de HTML con la función `innerHTML` :

```
class SellButton extends HTMLElement {
  constructor () {
    super();
  }

  connectedCallback () {
    this.innerHTML = `
      <div>
        <button>Comprar Ahora</button>
      </div>
    `;
  }
}

window.customElements.define('sell-button', SellButton)
```

Y esto es lo que veríamos ahora en el navegador:



Comprar Ahora

## Encapsulando el Markup en el Shadow DOM

Pero esto no es todo lo elegante que nos gustaría. Entre las propiedades que nos ofrecen los *Web Components* está el llamado **Shadow DOM** que nos es más que una forma de encapsular el DOM del componente (con su funcionalidad y estilos) y que por ejemplo, estos estilos no se "pisen" con otros estilos del documento.

```
class SellButton extends HTMLElement {
  constructor () {
    super();
  }

  connectedCallback () {
    let shadowRoot = this.attachShadow({mode: 'open'});
    shadowRoot.innerHTML = `
      <style>
        :host {
          --orange: #e67e22;
          --space: 1.5em;
        }
        .btn-container {
          border: 2px dashed var(--orange);
          padding: var(--space);
          text-align: center;
        }
        .btn {
          background-color: var(--orange);
          border: 0;
          border-radius: 5px;
          color: white;
          padding: var(--space);
          text-transform: uppercase;
        }
      </style>
      <div class="btn-container">
        <button class="btn">Comprar Ahora</button>
      </div>
    `;
  }
}

window.customElements.define('sell-button', SellButton);
```

Como puedes ver, usamos la función `attachShadow` para poder adjuntar el DOM que vamos a crear como Shadow DOM al WebComponent, y a continuación insertamos el HTML, dónde

funcionará en este Shadow DOM y no estará en contacto con otros estilos. Además podemos usar la propiedad `:host` y definir variables para utilizar dentro del estilo del componente, haciendo uso de nuevas propiedades de CSS.

Ahora el componente tiene esta pinta en el HTML:



## Usando Template

Si no te gusta escribir el HTML como string, puedes utilizar **template** en tu HTML, llamarlo y "popularlo" dentro del objeto *HTMLElement* que estamos definiendo.

Sería así:

```
<!-- Documento HTML con la plantilla -->
<template id="sellBtn">
  <style>
    :host {
      --orange: #e67e22;
      --space: 1.5em;
    }
    .btn-container {
      border: 2px dashed var(--orange);
      padding: var(--space);
```

```
    }  
    .btn {  
      background-color: var(--orange);  
      border: 0;  
      border-radius: 5px;  
      color: white;  
      padding: var(--space);  
      text-transform: uppercase;  
    }  
  </style>  
  <div class="btn-container">  
    <button class="btn">Comprar Ahora</button>  
  </div>  
</template>
```

Y el código JavaScript:

```
class SellButton extends HTMLElement {  
  constructor () {  
    super();  
  }  
  
  connectedCallback () {  
    let shadowRoot = this.attachShadow({mode: 'open'});  
    const t = document.querySelector('#sellBtn');  
    const instance = t.content.cloneNode(true);  
    shadowRoot.appendChild(instance);  
  }  
}  
  
window.customElements.define('sell-button', SellButton)
```

## Modularizar el WebComponent e importarlo desde otro archivo

Incluso podrías incluir todo el código de tu *WebComponent* (Template y Script) en un mismo fichero HTML.

En este caso, tienes que encapsular el `<template>` y el `<script>`

También tendremos que modificar un poco el código JavaScript si queremos importar este documento como un **HTMLImport** ya que si usamos `document.querySelector('#sellBtn')` `document` va a hacer referencia al documento HTML desde el que se importa y no va encontrar nuestro *template*.

Como queremos hacer referencia al documento importado, no al que lo importa, necesitamos añadir esta línea:

```
let importDocument = document.currentScript.ownerDocument;
```

Con esto ya podemos tener un fichero `components/sell-button.html` por ejemplo, con la siguiente estructura:

```
<html>
  <template id="sellBtn">

    <style>
      :host {
        --orange: #e67e22;
        --space: 1.5em;
      }
      .btn-container {
        border: 2px dashed var(--orange);
        padding: var(--space);
        text-align: center;
      }
      .btn {
        background-color: var(--orange);
        border: 0;
        border-radius: 5px;
        color: white;
        padding: var(--space);
        text-transform: uppercase;
      }
    </style>
```



```
</div>
</template>

<script>
  class SellButton extends HTMLElement {
    constructor () {
      super();
      this.importDocument = document.currentScript.ownerDocument;
    }

    connectedCallback () {
      let shadowRoot = this.attachShadow({mode: 'open'});
      const t = this.importDocument.querySelector('#sellBtn');
      const instance = t.content.cloneNode(true);
      shadowRoot.appendChild(instance);
    }
  }

  window.customElements.define('sell-button', SellButton);
</script>
</html>
```

y en tu `index.html` solo tendrías que importar este document con `HTMLImports` y usar el nuevo tag:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>Mi primer Vanilla Webcomponent</title>
  <link rel="import" href="components/sell-button.html">
</head>
<body>
  <sell-button></sell-button>
</body>
</html>
```

*NOTA: LOS [HTMLImports](#) solo funcionan en Chrome y Opera. En Firefox, Safari e Internet Explorer/Edge no. Te recomiendo que pruebes esto en un navegador Chrome actualizado*

*Si quieres usar `customElements` sin problema puedes hacer uso de un [polyfill](#)*

## Resumen

¿Qué hemos hecho? Hemos usado la plataforma web (HTML, CSS y JavaScript) para crear un *WebComponent* reutilizable. No confundir con un *Component* de React o Angular. Eso es una forma de construir aplicaciones web modularizadas. El propósito de un *WebComponent* es que sea reutilizable en cualquier proyecto web. De igual manera que usamos elementos HTML como `<select>`, `<input>`, `<video>`, `<a>`, etc... que tienen un determinado estilo y comportamiento, con los *WebComponents* se trata de extender ese ecosistema.

Unos buenos ejemplos son componentes web como `<google-map>` o `<youtube-video>` que podríamos usar en cualquier proyecto que hagamos.

## Enlaces útiles

Puedes encontrar más *WebComponents* creados por la comunidad, ya sea con VanillaJS o utilizando librerías como *Polymer* en la web [WebComponents.org](https://www.webcomponents.org)



Carlos Azaustre | Formación en JavaScript

Compártelo! 

En este [enlace te dejo un JSBin](#) para que puedas "jugar" con ello, y en [éste repositorio de GitHub](#) te dejo el código de éste ejemplo.

Y ampliar tu información sobre [customElements](#) y [Shadow DOM](#) en la web de **Google Developers**.



**Carlos Azaustre**

CTO y Cofundador de Chefly. Formador en tecnologías web: JavaScript, Node, Firebase, React y Vue.

[Leer más](#)



## ¿Quieres aprender y dominar JavaScript?

Únete a mi newsletter dónde comparto semanalmente los mejores recursos, enlaces y noticias relacionadas con el ecosistema JavaScript.

Además, te llevas de regalo **una guía en PDF para iniciarte en React** y como bonus un **curso gratuito de 4 vídeos** para aprender a desarrollar una aplicación web con **React y Firebase**

Tu nombre

Tu email

✓ Quiero unirme

 Libre de spam. Sólo contenido que te interesa.



Recomendar

Ordenar por los mejores ▾



Sé el primero en comentar...

INICIAR SESIÓN CON

O REGISTRARSE CON DISQUS

Nombre

Sé el primero en comentar.

TAMBIÉN EN CARLOS AZAUSTRE | FORMACIÓN EN JAVASCRIPT

## Cloud Functions for Firebase: Utilizando los activadores de Firebase Storage

3 comentarios • hace 5 meses



**webserveis** — Muy interesante el artículo

## Automatizando tu flujo de trabajo en el Frontend con GulpJS

1 comentario • hace 6 meses



**Daniel Lucumi** — Hola carlos actualmente trabajo con Browserify, estoy haciendo un sitio web con laravel donde todas mis views

## El futuro de los WebComponents gracias a Polymer 3.0

3 comentarios • hace 6 meses



**Oscar Ocegüera** — Master, sabe para cuando estará disponible la versión sin ser beta?

## Vue.js - Primeros pasos con el framework

10 comentarios • hace 6 meses



**Carlos Azaustre** — Muchas gracias Paco Pastor ! Siempre intento explicar los conceptos tal y como los he aprendido para



Suscríbete



Añade Disqus a tu sitio web

Añade Disqus Añadir



Privacidad

— Carlos Azaustre | Formación en JavaScript —

# JavaScript

WebComponents Nativos: Cómo pasar



ES6 Tagged Template Literals

Cómo conectar Firebase a una aplicación React

[See all 45 posts →](#)

## #NavidadConFirebase en Youtube

Las pasadas navidades, un grupo de desarrolladores nos juntamos para crear una iniciativa que consistía en enseñar a través de Youtube cómo integrar Firebase en diferentes tecnologías. Durante 1 mes, semana a semana,



CARLOS AZAUSTRE

REACT.JS

## Formas de crear un Componente en React

Un componente en React puede crearse de diferentes formas dependiendo de la versión de JavaScript que estemos utilizando y del propósito del componente. En este artículo recopiló las diferentes formas que existen y



CARLOS AZAUSTRE