ÍNDICE

1. WebComponents: Introducción y especificaciones

En el apartado anterior se definía la **interacción** como el proceso que establece un usuario con un dispositivo, sistema u objeto determinado. Es una acción recíproca entre el elemento con el que se interacciona y el usuario.

Algunas tendencias actuales son:

- Reactividad: El concepto de reactividad ha estado siempre presente en JavaScript en el uso de eventos. Si un usuario hace click en un botón o cambia su valor, la página web responde con una serie de acciones como se puede observar en el ejemplo.

```
const $increment = document.querySelector('#increment');
let counter = 0;

$increment.addEventListener('click', onIncrement);

function onIncrement(event) {
   counter += 1;
   document.querySelector('#counter');
}
```

Cada vez que se hace un click, se aumenta un contador y se actualiza automáticamente

Sin embargo, actualmente la reactividad también reside en lo que se conoce como el **patrón del observador,** en donde un objeto, llamado sujeto, mantiene una lista de sus dependientes, llamados observadores, y les notifica automáticamente de cualquier cambio de estado, usualmente llamando alguno de sus métodos.

```
<h1 id="nombre">
   Mi nombre es Eliseo <!-- Esto será reactivo -->
</h1>

<input id="texto" type="text" placeholder="Cambia el nombre del h1">

<script>
   var $h1 = document.getElementById('nombre')
   var yo = { nombre: 'Eliseo' } // El input se asocia al atributo nombre del objeto
yo

   // Cambiamos el atributo nombre cuando cambie el texto del input
   document.getElementById('texto').addEventListener("change",
       (e) => yo.nombre = e.target.value);

   // Definimos qué va a ocurrir cuando cambie el atributo nombre
   yo.cuandoCambie('nombre', function (valorNuevo, valorAnterior) {
      $h1.textContent = 'Mi nombre era ' + valorAnterior + ' pero cambio a ' +
valorNuevo
   })
</script>
```

La desventaja del primer fragmento de código es que cada vez que hacemos click en el botón se actualizaba el contador y teníamos que asignar manualmente este valor al HTML. En este caso, se implementa el código para que esta asignación sea automática y se asocie un atributo de JavaScript con HTML. Es decir, cada vez que cambiemos el atributo "nombre" se actualizará el elemento H1. Este es solo un ejemplo ilustrador y hay librerías y otras utilidades de código que permiten realizar este proceso ahorrando líneas de código.

- Reutilización: La interactividad es dar a un botón, escribir algo y que la web reaccione, incluir animaciones que el usuario puede activar o desactivar. Existen utilidades que nos permiten extender el HTML o crear componentes reutilizables en JavaScript para no tener que escribir siempre el mismo código cada vez que creamos una nueva página web.

En relación a estos conceptos ha surgido un estándar denominado **WebComponents** (componentes web). Se trata de una herramienta que permite etiquetas personalizables que permiten extender el código HTML e implementar la parte interactiva con JavaScript. Como resultado se obtiene una reducción del código, la modularización de este y una mayor capacidad de reutilización en nuestras aplicaciones.

Los menús, campos de formularios o botones se pueden encapsular en nuevas etiquetas facilitando la tarea de diseño. La idea es que una página web tenga esta forma y cada etiqueta incluya los estilos y funcionalidades internamente, aunque igualmente se puede modificar si el usuario lo desea:

```
<div id="app">
   <app-nav></app-nav>
   <app-view>
      <app-sidebar></app-sidebar>
      <app-content></app-content>
   </app-view>
</div>
```

La creación de componentes web se basa en cuatro especificaciones que veremos más adelante:

- Custom Elements: Se trata de una especificación que describe el método que nos permitirá crear nuevas etiquetas personalizadas, propietarias. Estas etiquetas las podremos diseñar para dar respuesta a cualquier necesidad que podamos tener.

- HTML Templates: 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. Normalmente se puede definir un template de manera implícita en JavaScript sin necesidad de usar esta etiqueta.
- Shadow DOM: Este sistema permite tener una parte del DOM oculta a otros bloques de la página. Básicamente sirve para solucionar un caso común que ocurre al incluir un archivo JavaScript de terceros. A veces usan clases o identificadores para aplicar estilos que afectan a otros elementos de la página, alterando elementos que no debería o alterando su aspecto. De esta manera, con el Shadow DOM podemos hacer que los componentes encapsulen partes que no estarían visibles desde fuera, pudiendo aplicar estilos que solo afectan al Shadow DOM de un WebComponent y evitando que estilos de la página lo sobrescriban.
- Módulos de ES6: Originalmente se utilizaba la etiqueta link con el atributo rel="import", aunque esta alternativa ya está obsoleta y no debe utilizarse. Los módulos de JavaScript surgieron a partir de la especificación denominada ECMAScript6 (ES6). Se trata de archivos que permiten exportar solamente una serie de variables, objetos, clases o funciones. La principal ventaja es que no hace falta incluir todo el código JavaScript en una página como ocurre tradicionalmente. De esta manera, se puede crear con componente web e importar solamente el código que nos interesa.

A continuación, veremos cada una de estas especificaciones.

2. Custom Elements

Tal y como se definía anteriormente, la especificación de **Custom Elements** es la que nos permite crear nuevos elementos de HTML que realizan funcionalidades personalizadas o presentan información de una nueva manera. Además, estos Custom Elements se pueden usar directamente, sin necesidad de apenas incluir una línea de código. 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.

A lo largo de este apartado vamos a aprender a usar las diferentes especificaciones de Web Components usando JavaScript estándar, lo que se conoce en el argot de los desarrolladores como **VanillaJS**. No obstante, el desarrollo de Web Components será mucho más rápido si nos basamos en alguna librería que nos permita automatizar procesos habituales.

La sintaxis básica de un Custom Element es la siguiente:

```
<dw-holamundo></dw-holamundo> <-- Incluir etiqueta en el HTML -->

<script>
   class DwHolamundo extends HTMLElement {
      // Implementación del componente
   }

   customElements.define('dw-holamundo', DwHolamundo); // Definición de componente
</script>
```

A continuación, se explica más en detalle haciendo referencia a los comentarios del código:

- **Incluir etiqueta en el HTML:** La sintaxis es la misma y se pueden añadir atributos y contenido, como veremos más adelante.
- **Implementación del componente:** Se trata de una clase de JavaScript que hereda HTMLElement. Más adelante veremos una serie de métodos y atributos que se pueden sobrescribir para añadir la funcionalidad del nuevo componente. También se puede heredar de elementos más concretos de HTML, por ejemplo HTMLParagraph (que sería la etiqueta <p>). Las clases tienen soporte en la mayoría de navegadores actuales, aunque para navegadores antiguos hay que realizar adaptaciones que se denominan Polyfills.
- **Definición del componente:** Un componente no está presente en una página hasta que no lo definimos. Se define con un nombre y se le asocia una clase que herede como mínimo de HTMLElement. La sintaxis del nombre debe incluir siempre un guión (-) y caracteres alfanuméricos en minúsculas. Por ejemplo: my-button, new-paragraph, cool-item1, etc.

Ahora vamos a incluir algo de funcionalidad al componente:

```
class DwHolamundo extends HTMLElement {
   constructor() {
      super();
      this.textContent = 'Hola mundo!!!'
   }
}
```

Hemos añadido un constructor sin parámetros, en el cual lo primero que hay que hacer es llamar al método **super** que invoca al constructor del padre (HTMLElement). Luego en el constructor se modifica el atributo textContent. Efectivamente, textContent es el mismo método que normalmente empleamos en el DOM (como innerHTML). Estamos modificando el contenido textual del nuevo elemento.

La clase HTMLElement permite emplear otros elementos que definiremos ahora, aunque veremos su implementación en otros apartados, ya que se usan en conjunto con el resto de especificaciones. Esta sería la estructura de un Custom Element.

```
class DwHolamundo extends HTMLElement {
   constructor() {
      super();
      // Código del constructor
   }
   connectedCallback() {
      // Código de connectedCallback
   }
   attributeChangedCallback (attr, oldVal, newVal) {
      // Código de attributeChangedCallback
   }
   static get observedAttributes(){
      // Devuelve array con el nombre de atributos reactivos
   }
   disconnectedCallback () {
      // Código de disconnectedCallback
   }
   adoptedCallback (oldDocument, newDocument) {
      // Código de adoptedCallback
   }
}
```

A continuación se indica el propósito de cada uno de los métodos y cuándo se ejecutan:

- **constructor:** Se crea o se actualiza una instancia del elemento. Es útil para inicializar el estado, configurar receptores de eventos o crear un Shadow DOM.

- **connectedCallback**: Se llama cada vez que se inserta el elemento en el DOM. Es útil para ejecutar código de configuración, como la obtención de recursos o la representación. En general, se debe demorar este trabajo hasta este momento, ya que en el constructor puede dar problemas.
- **disconnectedCallback**: Se llama cada vez que se elimina el elemento del DOM. Es útil para ejecutar código de limpieza (eliminación de receptores de eventos, etc.).
- **attributeChangedCallback:** Se llama cada vez que se modifica el valor de los atributos del HTML del componente creado que se indiquen en la propiedad observedAttributes. También se llama para obtener valores iniciales cuando el analizador crea un elemento o lo actualiza.
- **observedAttributes:** Devuelve un listado con el nombre de aquellos atributos del HTML del componente creado que provocan que se llame al método attributeChangedCallback. Básicamente, es una optimización del rendimiento. Cuando los usuarios cambien un atributo común como style o class, no conviene recibir cientos de callbacks no deseados.
- **adoptedCallback:** Se llama cuando el elemento personalizado se traslada a un nuevo documento. Por ejemplo, con document.adoptNode(el).

Por lo demás, aparte de estos elementos, como heredamos de HTMLElement se puede acceder a cualquier elemento del DOM como cuando en JavaScript trabajamos con documentGetElementById o querySelector para acceder a alguna etiqueta de la página. Los métodos y atributos más empleados en un Custom Element son:

- **getAttribute(nombre):** Permiten obtener el valor del atributo del HTML que se especifica como parámetro.
- **setAttribute(nombre, valor):** Permiten cambiar el valor del atributo del HTML cuyo nombre coincide con el primer parámetro. El nuevo valor será el segundo parámetro.
- **innerHTML:** Permite modificar el contenido HTML del componente.
- **textContent:** Permite establecer el contenido textual de un elemento sin interpretar el HTML.

Un uso bastante típico de getAttribute y setAttribute consiste en asociarlo con métodos getter y setter. Por defecto, la única forma de acceder a los atributos del HTML es a través de los propios **getAttribute** y **setAttribute.** Sin embargo, veamos el siguiente fragmento de código.

```
get status() {
   return this.getAttribute('status');
}

set status(newVal) {
   this.setAttribute('status', newVal);
}
```

De esta manera, al igual que se puede acceder a atributos como **id** o **innerHTML,** tenemos nuestra referencia **this.status** como se muestra debajo en lugar de tener que llamar todo el rato a los métodos getAttribute o setAtribute.

```
console.log(this.status);
this.status = 3;
```

Otros métodos y atributos de HTMLElement: https://developer.mozilla.org/es/docs/Web/API/Element

También es posible añadir eventos al igual que en cualquier elemento de una página. Se suele hacer en el constructor sobre la referencia **this** del propio componente. Por ejemplo:

```
constructor() {
   super();
   this.x = 0;
   this.addEventListener('click', () => {
     this.x++;
     this.render();
   });
)
```

En el ejemplo anterior se define un evento que se lanza al hacer click en el nuevo componente, actualiza un contador y llama a un método definido debajo que se encarga de actualizar el contenido HTML.

De la misma forma, un componente web es una clase al fin y al cabo y sus propiedades y métodos públicos son accesibles desde fuera.

Por ejemplo, sea el método siguiente definido en un componente web.

```
// Cualquier método definido en un componente web se puede llamar desde fuera
metodoPublico(lugar){
   console.log(`Llamando a método público desde ${lugar}`);
}
```

Nuestra etiqueta es accesible como cualquier otra. Podemos llamar a las propiedades y métodos que se heredan de **HTMLElement** y también a las definidas en la clase que crea el componente web.

```
<test-element id="test1" status="1"></test-element>
<script src="test-element.js"></script>
<script>
   var testE = document.querySelector("test-element");
   console.log("Pruebas de acceso a la propiedad x de la clase:", testE.x);
   testE.metodoPublico("HTML");
</script>
```

Por último, se comentaba anteriormente que otra opción es extender la implementación de alguna etiqueta de HTML en lugar de la clase genérica HTMLElement. Esto se haría creando la clase de la misma manera como se indica debajo.

```
class FancyButton extends HTMLButtonElement {
   constructor() {
      super(); // Primero se llama al constructor del padre
      this.textContent = "Hola Mundo botón!";
   }
}

customElements.define('fancy-button', FancyButton, { extends: 'button' });
```

Cabe destacar que existen **propiedades y métodos privados directamente añadiendo #** delante del identificador, pero aun no funcionan en muchos navegadores.

La implementación simplemente ha añadido un texto al botón. Lo que ha cambiado realmente es la forma de definir el elemento, en el que indicamos en el tercer parámetro el objeto con la propiedad extends y el elemento que estamos sobrescribiendo.

A la hora de incluir este botón el en el HTML, en lugar de añadir la nueva etiqueta, en este caso se emplea el **atributo is** de la propia etiqueta button para indicar que se refiere al nuevo componente que hemos

definido que añade nuevas funcionalidades. Como se puede observar, el atributo disabled seguirá funcionando y en este caso simplemente hemos modificado el texto de la etiqueta.

```
<button is="fancy-button"></button>
<button is="fancy-button" disabled></button>
```

3. HTML Templates

El estándar template nos permite crear plantillas que podemos completar con datos y presentar luego en el contenido de la página mediante JavaScript. Esto nos facilita el mantenimiento de código, ya que podemos crear la estructura de un componente y luego insertarla donde sea necesario empleando el DOM de JavaScript.

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

Se trata de una etiqueta bastante especial, puesto que es la única etiqueta de contenido que no tiene una representación directa en el renderizado de la página. El navegador al leer una etiqueta tempate 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 un navegador encuentra un template no hace nada con él, aparte de leerlo, esperando que se use más adelante de alguna manera.

La estructura 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>
```

Para mostrar el contenido de un template en una página tenemos que seguir tres pasos:

- Acceder al template a partir de algún selector o mediante su id
- Clonar el template en la memoria de JavaScript
- Inyectar el clon del template en el lugar donde se desee de la página

Un ejemplo podría ser el siguiente:

```
<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>
```

Este ejemplo no está asociado al uso de WebComponents, pero sirve para hacernos una idea de cómo emplearlo. Como se puede observar, los templates tienen una propiedad denominada content de la clase **HTMLFragment** que permite referenciar a su contenido y acceder en definitiva a muchas propiedades y métodos de HTMLElement. Una vez que tenemos el contenido lo clonamos en una variable y a

continuación se inserta en el documento. Véase que el método cloneNode tiene un parámetro booleano que es true cuando queremos incluir los hijos del elemento que vamos a clonar (es decir, si el párrafo contiene otras etiquetas internas).

De cara a incluir un template en un CustomElement, el problema es que un template es una etiqueta HTML y un WebComponent ya creado con un CustomElement se añade vía JavaScript. Por tanto, tenemos que definir una variable con dicho template como se indica a continuación.

```
const templateMyParagraph = document.createElement('template');
templateMyParagraph.innerHTML = `<p>Mi texto predeterminado</p>`;

customElements.define('my-paragraph', class extends HTMLElement {
   constructor() {
      super();
      this._shadowRoot = this.attachShadow({ mode: 'open' });
      this._shadowRoot.appendChild(templateMyParagraph.content.cloneNode(true));
   }
});
```

Como se puede observar, lo más sencillo es definir directamente el template con la función createElement y asociar su HTML. Luego a la hora de definir el elemento se clona como hemos visto anteriormente.

Dos aspectos a tener en cuenta del código:

- Se puede crear una clase anónima directamente en el parámetro a la hora de definir el CustomElement.
- El shadowRoot permite emplear métodos del DOM sin afectar a la estructura original donde se incluye el componente. Lo veremos más adelante y en cualquier caso los métodos son iguales y estamos empleando appendChild para incluir el contenido del template clonando el nodo.

Una etiqueta template se puede complementar con el uso de otra etiqueta denominada **slot.** Hasta ahora hemos visto que, si queremos utilizar un template para establecer el contenido de un componente web, apenas admite flexibilidad, ya que no podemos modificar el contenido del HTML interno. Esto se puede solucionar con el uso de slots.

Si volvemos al código anterior, podemos definir nuestro párrafo personalizado de la siguiente manera:

```
const templateMyParagraph = document.createElement('template');
templateMyParagraph.innerHTML = <p><slot name="my-text">Mi texto personal</slot></p>`;
```

Y a la hora de definir la etiqueta:

```
<my-paragraph>
   <span slot="my-text">¡Tengamos un texto diferente!</span>
</my-paragraph>
```

En este caso se está definiendo el slot con el atributo name. De cara a definir el WebComponent en la página, el HTML del slot se reemplazará por el span y esto se logra con el atributo slot con el nombre que hemos definido en la plantilla.

El atributo name no es ni siquiera necesario a la hora de definir el slot. En este caso, el span se podría incluir sin parámetros y el contenido del slot se reemplazaría automáticamente por dicho span. No obstante, queda más clara la organización asociando cada slot a un nombre.

Además, es posible añadir estilos específicos a los elementos que se encuentran dentro de un slot. Para ello se emplea el **pseudo-elemento denominado ::slotted**.

```
p::slotted(*) {
   color: white;
}
p::slotted(span) {
   color: orange;
}
```

El selector slotted puede recibir el parámetro *, que se aplicaría independientemente de la etiqueta que reemplaza el slot. También se puede restringir empleando cualquier selector válido. Por ejemplo, en este caso estamos empleando un span, pero si el span tuviera una clase denominada “verde”, entonces podríamos aplicar el selector ::slotted(.verde).

Otra posibilidad es consultar los nodos o elementos asociados a una etiqueta slot a través de los métodos **assignedNodes** y **assignedElements** de la especificación: https://developer.mozilla.org/en-US/docs/Web/API/HTMLSlotElement

4. Shadow DOM

De todas las especificaciones de los WebComponents, 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 autónomos e independientes de otros elementos de la página.

Normalmente, al insertar herramientas como jQuery o Bootstrap modifican estilos indeseados en la página. Todo esto está solucionado en los WebComponents y mucho depende directamente de la especificación de Shadow DOM.

Cuando creamos un Custom Element, a menudo este necesita generar nuevos elementos, como botones, campos de texto, iconos, párrafos, que colocará debajo de su jerarquía. El Custom Element 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 propietario.

El beneficio básico es que, al usar un Custom Element con Shadow DOM en la página su contenido encapsulado no podrá interaccionar con otros elementos de fuera, evitando daños colaterales. Un ejemplo podría ser el siguiente. En este código todavía no hay shadow DOM.

```
<style> <!-- Estilos generales de la página -->
   p{ color: red; }
</style>
<body>
   <p id="elem">Hola </p>
   <template>
      <style>
        p { color: blue; }
      </style>
      <p>Hello world!</p>
   </template>
   <hello-world></hello-world>
</body>
```

El componente hello-world se crearía de la siguiente manera:

```
<script>
   class HelloWorld extends HTMLElement {
      constructor() {
         super();
         // Clono el contenido del template
         var template = document.querySelector('template').content;
         var elem = template.cloneNode(true);

         // Se inserta el template en el
         this.appendChild(elem);
      }
   }
</script>
```

De esta manera, a la etiqueta asociada al elemento **<hello-world>** se le aplican los estilos de la página y no solo esto, si quisiéramos acceder al contenido interno con `document.querySelector("hello-world").querySelector("p")`,entonces obtendríamos una referencia válida. El objetivo de los WebComponents es precisamente evitar esto y encapsular el contenido del nuevo elemento.

Este problema se soluciona con el shadow DOM. Para ello, basta con modificar el código JavaScript anterior.

```
class HelloWorld extends HTMLElement {
   constructor() {
      super();

      // Clono el contenido del template
      var template = document.querySelector('template').content;
      var elem = template.cloneNode(true);

      // Accedo al shadow root e incluyo el template
      this.attachShadow({ mode: 'open' });
      this.shadowRoot.appendChild(elem);
   }
}
```

Para crear un shadow DOM de cualquier elemento de la página (no necesariamente un WebComponent), basta con ejecutar el método **attachShadow** en el objeto que referencia a dicho elemento (en este caso this, ya que se está haciendo desde la propia clase que lo define).

Luego para referenciar al contenido con **shadowRoot** hay dos opciones en función del parámetro con el **atributo mode**:

- Si mode es open, entonces podemos acceder al atributo shadowRoot en el elemento al que se le ha asociado. Como se puede observar en el código anterior, ejecutamos un appendChild propio del DOM, pero no se adjuntará al DOM de la página si no al shadow DOM del propio componente.
- Si mode es closed, la referencia a shadowRoot será nula. Esto impide que se acceda al contenido del elemento desde la página en la que se inserta el elemento. Incluso desde la propia definición del componente tendremos que volcar la salida de attachShadow a otra variable para poder acceder al shadow DOM.

```
// Accedo al shadow root e incluyo el template
this._DOMOculto = this.attachShadow({ mode: 'closed' });
this._DOMOculto.appendChild(elem);
```

De esta manera, creamos un atributo en la clase para acceder al shadow DOM sin hacerlo directamente a través de shadowRoot, lo que aporta mayor seguridad.

5. Módulos de ES6

Uno de los principales problemas que ha ido arrastrando JavaScript desde sus inicios es la dificultad de organizar de una forma adecuada una aplicación grande con muchas líneas de código.

En un principio, y de forma nativa, la forma más extendida era incluir varias etiquetas <script> desde nuestra página HTML. De esta forma, podíamos tener varios ficheros JavaScript separados, cada uno para una finalidad concreta. Sin embargo, este sistema es poco modular, ofrecía un control pobre y resultaba lento, ya que sobrecargaba al cliente con múltiples peticiones extra. Además, si hay dos métodos o variables con el mismo nombre ya nos da un error.

En ECMAScript 6 se introduce una característica nativa denominada **Módulos ES6**, que permite la importación y exportación de código entre diferentes ficheros JavaScript, eliminando las desventajas que teníamos hasta ahora y permitiendo trabajar de forma más flexible desde el código JavaScript.

Por defecto, un fichero JavaScript no tiene módulo de exportación si no se usa un **export** al menos una vez. Existen varias formas de exportar código mediante la palabra clave export:

```
let number = 4;
const saludar = () => "¡Hola!";
const goodbye = () => "¡Adiós!";
class Clase {}

export { number }; // Se crea un módulo y se añade number
export { saludar, goodbye as despedir }; // Se añade saludar y despedir al módulo
export { Clase as default }; // Se añade Clase al módulo (default)
export { saludar as otroNombre }; // Se añade otroNombre al módulo
```

Es posible exportar cualquier tipo de variable, objeto, clase o función que se pueda declarar en JavaScript. Un archivo js pasa a ser un módulo en cuanto tiene un solo export. En muchas ocasiones se cambia la extensión a **mjs** para diferenciarlos.

Si queremos utilizar aquellos elementos que se han exportado, se puede hacer de la siguiente manera:

```
import nombre from './file.js'; //Importa el elemento por defecto de file.js en nombre.
import { nombre } from './file.js'; //Importa sólo el elemento nombre de file.js.
import { n1, n2 } from './file.js'; //Importa los elementos indicados desde file.js.
import './file.js'; //No importa elementos, pero ejecuta el código de file.js.

import * as obj from './file.js'; /* Importa todos los elementos de file.js en el objeto obj. Para acceder a ellos sería con obj.n1, obj.n2, etc */
```

Véase que una de las exportaciones (solo una) puede tener la palabra reservada default. En ese caso, cuando importamos no hace falta incluir corchetes, ya que por defecto el nombre asignado se asocia directamente al elemento por defecto. El resto de elementos exportados se añaden individualmente con import y cada nombre entre corchetes, como por ejemplo { n1, n2}.

Es posible importar componentes de módulos en un archivo .js o desde un **script del tipo modulo** en el HTML**.**

```
<script type="module">
   import f2 from "./testmodules.js"
   import { saludar, despedir } from "./testmodules.js"
</script>
```

Solo se puede importar un JavaScript de tipo módulo si se indica como tal en la etiqueta script con **type="module"**.

Aplicando los módulos a los WebComponents, por un lado, es perfectamente posible añadir un CustomElement de la forma tradicional con la etiqueta script.

Sin embargo, los módulos nos aportan la ventaja de poder organizar mejor el código y compartir clases u otros elementos entre componentes.

Un WebComponent definido como módulo se puede importar de esta manera:

```
<script type="module" src="./my-component.js">
```

De la misma manera, se puede incluir con la cláusula import.

```
<script type="module">
   import MyComponent from './modules/my-component.js';
   customElements.define('my-component', MyComponent);
</script>
```

La gran ventaja es que los elementos definidos como globales dentro del módulo no afectan al contexto global del navegador, es decir, una variable o una clase definida como global en el módulo no queda definida en el objeto window del navegador, queda retenida en el ámbito del módulo. En definitiva, solo se puede hacer referencia a aquello definido con **export**.

Otro aspecto a tener en cuenta es quién debe definir el componente: el módulo (primer fragmento de código de esta página que incluye el componente en el atributo src) o la página donde se incluye (segundo

fragmento de código que realiza el import y el define en el HTML). En realidad, no hay una respuesta exacta y depende del desarrollador y el ámbito.

Lo importante, es que estamos facilitando la reutilización de código. Por ejemplo, si un módulo depende de otro podemos emplear sus variables o funciones fácilmente, entre otras ventajas.

Una desventaja que afecta a la hora de hacer pruebas, es que un módulo solamente funciona bajo el protocolo del **Intercambio de Recursos de Origen Cruzado o CORS.** Esto implica que únicamente es posible emplear un módulo desde un dominio y si queremos probar una página web estática en local tendremos que arrancar un servidor. Esto se hace fácilmente mediante la tecnología Node.js de JavaScript, Docker, GitHub pages u otras alternativas comerciales.

6. WebComponents: Implementación y buenas prácticas

Una vez estudiadas las especificaciones, vamos a ver cómo combinar los 4 estándares definidos para crear WebComponents.

Para ello, vamos a partir de un ejemplo que crea un botón desde cero con un heredando directamente desde HTMLElement (no desde HTMLButtonElement).

La estructura será muy parecida a lo que se estudiaba en el apartado de CustomElements añadiendo algún método adicional. En este caso no utilizaremos disconnectedCallback o adoptedCallback, dada la sencillez del componente, que está orientado a pruebas en una sola página.

```
import { elemento } from './modulo.mjs'; // Podemos traernos elementos de otros
módulos

//Exportamos el componente como elemento por defecto
export default class BotonStatus extends HTMLElement { /
   constructor() {
      super();
      // Código del constructor
   }
   connectedCallback() {
      // Código de connectedCallback
   }
   attributeChangedCallback (attr, oldVal, newVal) {
      // Código de attributeChangedCallback
   }
   static get observedAttributes(){
      // Devuelve array con el nombre de atributos reactivos
   }
   get status() {
      // Getter del atributo status
   }
   set status(newVal) {
      // Setter del atributo status
   }
   render(currentStatus) {
      // Llamamos a este método cada vez que queremos importar la página
   }
}
```

El elemento que vamos a implementar tiene una estructura como la que se indica a continuación:

```
<boton-status>Haz clic aquí</boton-status>
<boton-status status="danger">Haz click para tener más suerte</boton-status>
<boton-status status="success">Haz clic aquí para seguir con éxito!</boton-status>
```

Como se puede observar, tenemos un atributo denominado **status** y un HTML que debería mostrarse en el componente.

En primer lugar, codificamos el constructor:

```
constructor() {
   super();

   // Creamos propiedades de la clase que no coinciden con atributos del HTML
   this.attribute1 = true;
   this.attribute2 = "Prueba";

   // Creamos shadow DOM
   this.attachShadow({ mode: 'open' });

   this.shadowRoot.addEventListener('click', () => {
      // Código de evento onclick
   }
}
```

Recuerda que para crear una propiedad en una clase en JavaScript basta con añadir this.nombre_propiedad y ya se puede acceder desde cualquier otro método de la clase y también desde fuera, ya que hasta la versión ES2019 (que apenas está soportada en los navegadores actuales) no se pueden crear miembros privados de una clase. De cara a organizar el código se recomienda crear las propiedades en el **constructor**. No obstante, se pueden crear en cualquier sitio.

Otra opción para crear propiedades es definirlas arriba del todo de la clase.

```
export default class BotonStatus extends HTMLElement {
   attribute1 = true;
   attribute2 = "Prueba";

   // Código del constructor
   constructor() {
      // Creamos shadow DOM
      this.attachShadow({ mode: 'open' });

      this.shadowRoot.addEventListener('click', () => {
      // Código de evento onclick
      }
   }
}
```

Como se puede observar en el código anterior, para crear propiedades al comienzo de la clase no hay que incluir la referencia **this**. Luego en el resto de métodos de la clase se puede acceder con el propio this. Fíjate que arriba del todo tampoco se pueden crear variables con var, let o const.

Por otro lado, en el código constructor nos hemos limitado a crear el shadow DOM asociado al nuevo componente y un evento que se asociará a dicho shadow DOM. Recuerda que una vez que se ejecuta el método attachShadow, si se crea con el atributo open se puede acceder directamente a través de shadowRoot.

En resumen, el constructor debe seguir las recomendaciones que se indican a continuación:

- Debe llamar siempre al método **super** de la clase que hereda.
- No debe tener un **return**.
- No debe modificar el valor de ninguno de los atributos.
- No debe añadir nuevos atributos.
- No debe llamar a los métodos **document.open** o **document.write**.
- Se debe evitar la representación del componente en el constructor y delegar este trabajo a connectedCallback, ya que así garantizamos que se representa cuando se va a insertar en el DOM de la página.
- Aun así, hay muchos componentes web que se representan en el constructor. En realidad, si el JavaScript se incluye en el lugar adecuado de la página cuando se ha cargado el DOM no tendría por qué suponer ningún problema.

En definitiva, el trabajo del constructor se debe limitar crear el shadow DOM, crear eventos o crear atributos de clase (propios de la clase y no de la página).

Un error típico es modificar el valor de un atributo en el constructor. Es decir, en nuestro HTML, el elemento **<boton-status>** tiene el atributo **status.** Este atributo no se debe modificar en el constructor y si queremos cambiar algún valor manualmente, esto debe hacerse en connectedCallback.

Por ejemplo:

```
constructor() {
   super();

   // Modificamos el atributo status del HTML
   let currentStatus = this.getAttribute('status');
   if (currentStatus) {
      this.status = currentStatus;
   } else {
      this.status = 'neutral';
   }

   // Creamos propiedades de la clase que no coinciden con atributos del HTML
   this.attribute1 = true;
   this.attribute2 = "Prueba";

   // Creamos shadow DOM
   this.attachShadow({ mode: 'open' });

   this.shadowRoot.addEventListener('click', () => {
      // Código de evento onclick
   }
}
```

Es muy típico obtener la excepción *The result must not have attributes* al modificar atributos del HTML como status. Esto debería hacerse en connectedCallback.

La diferencia de estos atributos respecto a attribute1 y attribute2 es que están presentes en la etiqueta y no son únicos de la clase.

De hecho, se puede acceder directamente al valor de los atributos del HTML con el método `this.getAttribute('status')`. Esto implica que crear una propiedad en la clase con un mismo nombre que un atributo puede ser una fuente de conflictos.

Si queremos hacerlo, desde luego que no debería ser en el constructor. Si además queremos acceder a la propiedad status, como se indicaba anteriormente lo más recomendable es crear los getter y setter correspondientes, pero de la siguiente manera.

```
/* Getter y setter de status. Se accede cuando se cambia el valor de estos atributos o
para obtenerlo */
get status() {
   return this.getAttribute('status');
}

set status(newVal) {
   this.setAttribute('status', newVal);
}
```

Como status está asociado a un atributo de HTML, si accedemos con `this.status` en los getter y setter obtenemos la misma excepción. Por tanto, tenemos que utilizar `getAttribute` y `setAttribute`.

Con el resto de propiedades de la clase sí que se puede emplear el método normal y, por ejemplo, devolver en el get `this.attribute1` o en el set asignar `this.attribute1 = newVal`. Aunque hay que tener cuidado, ya que según donde definamos la propiedad en el código JavaScript puede haber problemas e interpretar los getter y setter como un atributo del HTML.

Por ejemplo, si se definen directamente las propiedades en el constructor como en este código, se interpretan internamente como atributos del HTML y el único código válido en los getter y setter son las referencias a `getAttribute` y `setAttribute`. Sin embargo, las propiedades definidas arriba del todo como aquí sí se interpretan como específicas de la clase y se pueden acceder directamente en un getter y setter sin necesidad de que sean atributos de HTML.

Este es uno de los aspectos que suelen presentar más problemas a la hora de implementar un WebComponent. El resto dependerá de la complejidad del mismo.

Siguiendo con la implementación del componente, ahora vamos a ver que la reactividad mencionada al comienzo de este documento está presente en esta tecnología.

```
constructor() {
   // Resto de código anterior del constructor…

    this.shadowRoot.addEventListener('click', () => {
       // Creamos un objeto con todos los valores posibles y descripciones
       let opciones = {neutral:"Click para probar",danger:"Error",success:"Correcto"};
       let valores = Object.entries(opciones);

       // Se cambia el status, que desencadena attributeChangedCallback
       this.status = valores[0];
       this.shadowRoot.querySelector("#buttonStatus").textContent = valores[1];
   }
}

// Se muestra el HTML por primera vez cuando se ha cargado el DOM
connectedCallback() {
   this.render(this.status);
}

// Se llama cuando se modifica el valor de los atributos de observedAttributes
attributeChangedCallback(attr, oldVal, newVal) {
   if (attr == 'status' && oldVal != newVal) {
      this.status = newVal;
      this.render(newVal);
   }
}
```

```
// Atributos reactivos cuando se invoca attributeChangedCallback
static get observedAttributes() {
   return ['status'];
}
```

A continuación se explican una serie de aspectos importantes del código anterior:

- **connectedCallback:** Llama al método render que se encarga de mostrar el HTML del componente por primera vez según el valor de status (que se obtiene con los getter y setter anteriores).
- **get observedAttributes:** Sirve para indicar que el atributo status "reaccionará" cuando se cambie su valor.
- **attributeChangedCallback:** Se llama cuando se modifica cualquier atributo definido en el array observedAttributes. En este caso nos limitamos a actualizar la propiedad status asociada al atributo de HTML con el mismo nombre y mostrar de nuevo la página. Lo importante del código es que el componente está preparado para desencadenar un evento cuando se modifica cualquier atributo de observedAttributes
- **Evento onClick:** Realiza una serie de acciones cuando se hace click en el componente. A tener en cuenta que se modifica el atributo status. Esto hace que se llame a attributeChangedCallback y automáticamente se renderiza de nuevo el componente.
- En el evento onClick tenemos el código `this.shadowRoot.querySelector("#buttonStatus")`. Es bastante típico tener una propiedad de clase que hace referencia a algún elemento del HTML. De esta manera, solamente tenemos que cambiar el elemento afectado y no cargar todo el HTML de nuevo. En este ejemplo se hace de las dos maneras a modo ilustrativo.

Por último, ¿dónde se está representando el HTML de la página? ¿Y la propiedad template? Como estamos empleando un JavaScript la única forma de crear esta etiqueta es de manera dinámica con `document.createElement('template')` como se estudiaba anteriormente.

Incluso podría no ser necesario y en la práctica ocurre que los templates se crean de manera implícita y directamente tenemos una variable con el código HTML del componente. Veamos nuestro método **render:**

```
render(currentStatus) {
   this.shadowRoot.innerHTML = `
      <style>
         div {
            display: inline-block;
            color: #fff;
            border-radius: 3px;
            padding: 10px;
            cursor: pointer;
            background-color: #000;
         }
         .neutral {
            background-color: #888;
         }
         .danger {
            background-color: #d66;
         }
         .success {
            background-color: #3a6;
         }
      </style>
      <div id="buttonStatus" ${currentStatus ? `class="${currentStatus}"` :
          'class="neutral"'}><slot></slot></div> `;
```

Básicamente estamos añadiendo un Template String de la versión ES6 de JavaScript y se asocia directamente con el innerHTML del shadow DOM. El Template String añade una condición que cambia el atributo class en función del parámetro con el atributo status que se reciben en la llamada.

Además, se añade un slot sin nombre que automáticamente reemplaza su contenido por el texto de las etiquetas <boton-status> definidas anteriormente.

Como se decía anteriormente, puede resultar bastante ineficiente renderizar todo el HTML cada vez que se hace un pequeño cambio. Una vez mostrado por primera vez podría ser interesante guardar alguna referencia a los elementos del shadow DOM que queremos modificar posteriormente. En este caso, por ejemplo el div con id buttonStatus.

```
// Se muestra el HTML por primera vez cuando se ha cargado el DOM
connectedCallback() {
   this.render(this.status);

   // Nos quedamos con una referencia a 'buttonStatus' para solo modificar esta parte
   this.boton = this.shadowRoot.querySelector("#buttonStatus");
}
```

Luego en el listener del evento nos bastaría con modificar solamente esta parte del HTML

```
this.boton.className = opcionElegida[0];
this.boton.textContent = opcionElegida[1];
```

En conclusión, la forma de renderizar el HTML es bastante flexible. Se pueden emplear templates como se explicaba en el apartado de HTML Templates, incluso una opción que permite organizar mejor el código es definirlo en otro módulo de JavaScript. Lo importante a fin de cuentas es tener el código estructurado de la forma más flexible posible para manipular el HTML, ya sea con templates explícitos o no.

Al principio de este apartado, exportábamos el módulo como default. Si lo definimos en el JS bastaría con incluirlo con alguna de las dos siguientes alternativas.

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

<script type="module">
   import SellButton from "./boton-status.js"
</script>
```

Si no, bastaría con la segunda etiqueta script definiendo el componente con `window.customElements.define('boton-status', BotonStatus);`

7. WebComponents: Otras herramientas

Esta unidad tiene como objeto de estudio la tecnología nativa de WebComponents, sin embargo, en la práctica se emplean una serie de librerías basadas en este estándar. A continuación, se citan algunas de ellas:

- Angular: Conocida librería que permite el enfoque de componentes web. Tutorial oficial: https://angular.io/guide/architecture-components
- Vue.js: Otra librería que también permite arquitecturas basadas en componentes. Tutorial oficial: https://es.vuejs.org/v2/guide/components.html
- React: Muy demandada en empresas, aunque hoy en día Vue.js está empezando a ganar en popularidad. Tutorial oficial: https://es.reactjs.org/docs/components-and-props.html
- Polymer: Es una librería original para el desarrollo de WebComponents, aunque ya obsoleta en detrimento de LitElement. Tutorial oficial: https://polymer-library.polymer-project.org/
- LitElement: Es la librería para desarrollo de Custom Elements creada por el equipo de Polymer (Google). Es muy cercana al estándar de WebComponents y al Javascript nativo, de hecho es necesario emplear lo nativo para la mayoría de funionalidades, de ahí su reducido peso. Es extremadamente rápida, más que cualquier otro framework y librería popular, como React, Vue y por supuesto Angular. Tutorial oficial: https://lit-element.polymer-project.org/
- Stencil.js: Inspirado en Angular, React, Vue y Polymer. Se podría definir como un compilador de WebComponents que permite emplear estos componentes tanto en aplicaciones implementadas mediante cualquier "framework", como en aplicaciones hechas con vanilla JS (JavaScript nativo). Tutorial oficial: https://stenciljs.com/