WebComponents

Los Componentes Web son un paquete de diferentes tecnologías que te permiten crear elementos personalizados reutilizables con su funcionalidad encapsulada apartada del resto del código y utilizarlos en las aplicaciones web. Como desarrolladores, sabemos que reutilizar código tanto como sea posible es una buena idea. Esto tradicionalmente no es sencillo para estructuras de marcado personalizadas ya que un complejo HTML (y sus estilos y scripts asociados) que en ocasiones se han tenido que escribir para renderizar controles de interfaz (UI) personalizados, y ahora usarlos múltiples veces puede crear un caos en la página si no se es cuidadoso. Esto se resuelve gracias a la encapsulación de sus propiedades y funciones que permiten mantener la integridad de los elementos sin que estos afecten o se confundan con el resto de los elementos de la interfaz, incluso aunque formen parte de instancias del mismo WebComponent. Adicional a esto nos da la opción de usar el ShadowDOM que evitara que elementos extraños al WebComponent interfieran en sus estilos y funcionalidad.

Adicional a esto los WebComponents utilizan todas las ventajas de la POO y a su vez trae integrado un ciclo de vida el cual permite tener pleno control del componente y su comportamiento en diferentes momentos desde su implementación.

Un WebComponent se estructura de la siguiente manera:

```
class ComponentName extends HTMLElement {
    constructor() {
        super();
    }
    //LIFE CICLE METHODS
    connectedCallback() {}
    //CUSTON METHODS
}
customElements.define("component-name", ComponentName);
```

Entre los métodos del ciclo de vida del WebComponent tenemos los siguientes:

- connectedCallback: Se invoca cada vez que el elemento personalizado se agrega a un elemento conectado a un documento. Esto sucederá cada vez que se mueva el nodo y puede suceder antes de que el contenido del elemento se haya analizado por completo.
- disconnectedCallback: Se invoca cada vez que el elemento personalizado se desconecta del DOM del documento.
- adoptedCallback: Se invoca cada vez que el elemento personalizado se mueve a un nuevo documento.
- attributeChangedCallback: Se invoca cada vez que se agrega, elimina o cambia uno de los atributos del elemento personalizado. Los atributos para los que se debe notar el cambio se especifican en un observedAttributes método de obtención estático.

La forma de incluir contenido dentro de un WebComponent es haciendo uso del método append o appendChild, asi mismo se puede tener acceso a otros métodos como removeChild.

```
connectedCallback() {
   this.append("Hola Mundo!");
}
```

Al agregarle hijos a un WebComponent de esta forma, cada nodo incluido forma parte del DOM global y es accesible y modificable por cualquier otro componente del DOM, incluido script externos y sus estilos son modificables por cualquier clase CSS.

Esto se puede resolver fácilmente usando el ShadowDOM, el cual aísla completamente los elementos internos del WebComponent del resto de los elementos del DOM, permitiéndole total autonomía de su propio ámbito.

```
class ComponentName extends HTMLElement {
    constructor() {
            super();
            this.attachShadow({ mode: "open" });
        }
    connectedCallback() {
            this.shadowRoot.append("Hola Mundo!");
        }
}
```

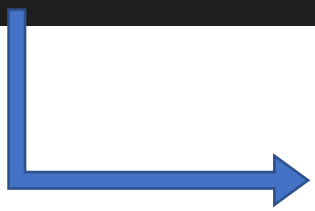
así mismo desde cualquiera de los métodos ya sea del ciclo de vida o incluso desde el constructor se pueden invocar funciones y procesar atributos. Una vez implementada la lógica del componente utilizarlo es tan sencillo como incluir la referencia al script que lo define dentro de nuestra etiqueta head o en el caso de estar usando ES6 modules, simplemente importar el script donde se valla a utilizar. Luego su tratamiento es igual al de cualquier etiqueta HTML normal.

Veamos un ejemplo práctico de la implementación de un WebComponentes a partir de nuestra pequeña aplicación de ejemplo.

Actualmente poseemos una función que devuelve un nodo HTML que posee toda la estructura de nuestra Card, al pasarlo a una estructura de WebComponent ocurriría lo siguiente:

```
const CardComponent = (element) => {
    const Card = document.createElement("div");
    Card.className = "card";
    const labelTitle = document.createElement("label");
    labelTitle.className = "title";
    labelTitle.innerText = element.title;
    const secContain = document.createElement("section");
    secContain.innerText = element.Contain;
    const secDetail = document.createElement("section");
    secDetail.innerText = element.Detail;
    Card.append(labelTitle, secContain, secDetail);
    return Card;
}
```

Si podemos notar básicamente la función CardComponent es convertida en parte de la lógica interna de WCardComponent y este método ahora llamado DrawCard es invocado dentro del connectedCallBack y agregado directamente al shadowRoot.



Otra cosa interesante que debemos tomar en cuenta es que el objeto “element” en este caso ya no sería un parámetro ahora es un atributo del componente, por lo que se debe hacer uso de la palabra reservada “this” para referenciarlo, y en lugar de utilizar element.title usar this.element.title

```
class WCardComponent extends HTMLElement {
    constructor() {
        super();
        this.attachShadow({ mode: "open" });
    }
    connectedCallback() {
        this.shadowRoot.append(this.DrawCard());
    }
    DrawCard() {
        const Card = document.createElement("div");
        Card.className = "card";
        const labelTitle = document.createElement("label");
        labelTitle.className = "title";
        labelTitle.innerText = this.element.title;
        const secContain = document.createElement("section");
        secContain.innerText = this.element.Contain;
        const secDetail = document.createElement("section");
        secDetail.innerText = this.element.Detail;
        Card.append(labelTitle, secContain, secDetail);
        return Card;
    }
}
customElements.define("w-card", WCardComponent);
```

Si bien es cierto la estructura en un principio se vuelve más compleja, dado que ahora hay muchos más elementos y comportamiento que debemos tener en cuenta, la cantidad de ventajas que esto ofrece hará que el componente tenga acceso a características adicionales que facilitaran en gran manera la programación y control de estos.

Nota: Posiblemente si tu componente es tan simple que solo debe devolver una estructura básica, su lógica no va más allá de un simple nodo procesado y a su vez no necesita estar encapsulado, la mejor opción sea que uses una simple función como es el caso de CardComponent(element). Pero sino es el caso, lo más probable es que requieras de todo el poder del api de customElements para construir tus interfaces. Por otro lado, el uso de WebComponents es totalmente opcional y depende más del estilo de trabajo que tengas, sin embargo, es la mejor forma de gestionar los elementos de DOM de forma anidada y que a su vez sean totalmente reutilizables, con comportamientos flexibles.

Luego para darle un uso dentro de nuestra aplicación solo debemos crear el elemento como si fuese un elemento HTML común y si este componente hiciera uso de alguna propiedad particular deberíamos asignársela, por ejemplo, en este caso la asignación del atributo element. Por lo que dentro del index.js haremos una pequeña modificación para realizar la creación dinámica del WebComponent “w-card” en lugar de la invocación a la función CardComponent(element).

```
window.onload = () => {
    const cards = [
        { title: "Card 1", Contain: "Contain", Detail: "Detail" },
        { title: "Card 1", Contain: "Contain", Detail: "Detail" },
        { title: "Card 1", Contain: "Contain", Detail: "Detail" }
    ]
    const Frag = document.createDocumentFragment();
    for (let index = 0; index < cards.length; index++) {
        const element = cards[index];
        const newCard = document.createElement("w-card");
        newCard.element = element;
        Frag.append(newCard);
    }
    App.append(Frag);
}
```

En este punto debemos encontrar la manera de simplificar la escritura de nuestros componentes dado que el uso de atributos y asignación de eventos a los diferentes elementos del DOM puede ser una tarea tediosa, por lo que sería interesante crear una estructura lógica que nos permita escribir el código de la forma más estructurada y simple posible, realizar esta parte también depende mucho de la creatividad de cada desarrollador, se puede hacer desde una simple función que tome parámetros hasta alguna estructura de clase o prototipos, en el caso de esta propuesta lo plantearemos desde la estructura de una clase con métodos estáticos, esto nos permitirá mantener todos los métodos requeridos agrupados y en el caso de usar ES6 modules facilitara la importación de su funcionalidad.

Para la creación de estas funciones crearemos un archivo JS dentro de la carpeta WModules (carpeta CoreTools del esquema), el archivo llevará por nombre WComponentsTools.js y este deberá ser referenciado o importado desde cualquier punto que se valla utilizar (por el momento usaremos referencias de script simples dentro del head de nuestro html).

```
<script src="./WDevCore/WModules/WComponentsTools.js"></script>
```

Este archive poseerá las clases importantes sobre el manejo de tareas repetitivas de nuestro marco de trabajo, la primera clase que construiremos será la encargada de tareas de renderización, la cual llamaremos WRender.

```
class WRender {
}
```

El primer método que esta clase contendrá será el encargado de crear HTMLElements con sus propiedades e hijos si este tuviera.