DESARROLLO WEB EN ENTORNO CLIENTE

UNIDAD 4 (parte 2/3):

Creación de componentes

1. Web Componentes

Los Web Components están formados por un **conjunto de diferentes tecnologías que encapsulan la estructura interna de elementos HTML** y su correspondiente funcionalidad con el fin de que puedan ser reutilizados tanto en webs como aplicaciones. Entre estas tecnologías normalmente se encuentran CSS y JavaScript.

Los Web Components están basados en cuatro especificaciones principales:

- **Custom Elements:** Esta especificación está formada por varios APIs de JavaScript que **permiten crear y definir elementos personalizados y su funcionalidad** y que pueden ser utilizados como cualquier otro elemento nativo de HTML.
- **Shadow DOM:** Está formado por un conjunto de APIs y sirve para encapsular elementos HTML de manera que acaba formando un *DOM* que solo es visible desde la etiqueta del propio elemento. Se renderiza por separado del *DOM* principal evitando así las posibles colisiones y conflictos de estilos y funcionalidades con el resto de elementos.

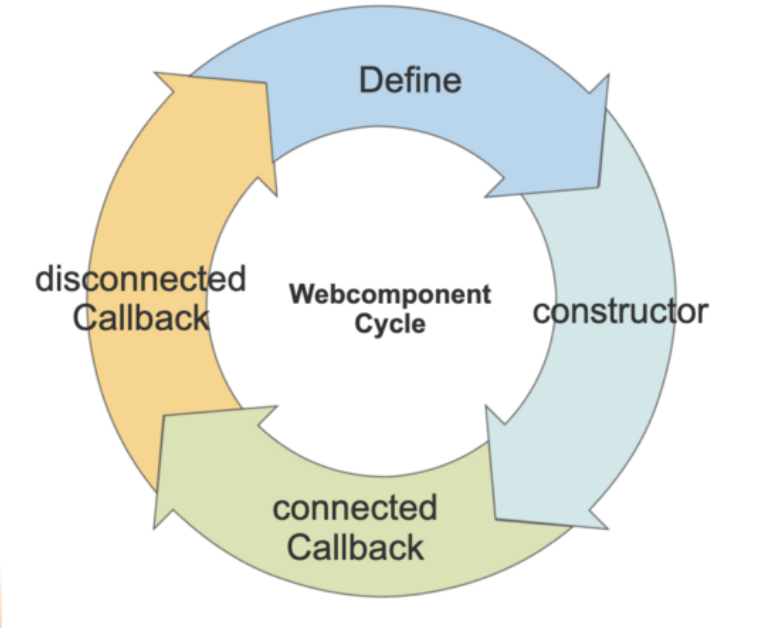
1. Web Componentes

- **ES Modules:** Sirve para definir la inclusión y la reutilización de documentos JavaScript de manera modular, **basada en estándares y de una manera muy eficiente.**
- **HTML Templates:** Los elementos <template> y <slot> **posibilitan definir plantillas que no son mostradas en la página renderizada.** Además pueden ser reutilizadas de manera recurrente para ciertos elementos previamente definidos.

1.1 Ciclo de vida

Los cuatro estados principales del ciclo de vida de un Web Component son:

- **Defined:** Es el primer paso que iniciará todo el flujo del Web Component. Hay que realizar una llamada a *customElements.define()* con el nombre de la etiqueta que se desea crear y una clase JavaScript que extienda el HTMLElement básico. Con esto lo que se consigue es registrar nuestro Web Component para luego poder utilizarlo como un elemento HTML más.




```
window.customElements.define (my-first-component,
MyFirstComponent);
```

1.1 Ciclo de vida

- **constructor():** Siempre empieza llamando a super() de forma que se establezca correctamente el encadenado del prototipo. Dentro del constructor, debemos definir toda la funcionalidad que tendrá el elemento cuando se instancie.
- **connectedCallback():** Se invoca cada vez que se añade un Web Component al *DOM.*
- **disconnectedCallback():** Se invoca cada vez que el Web Component se desconecta del *DOM.*

1.1 Ciclo de vida

A su vez hay otras funciones que también están dentro del ciclo de vida y nos pueden resultar útiles:

- **adoptedCallback():** Se invoca cada vez que el elemento se mueve a un nuevo DOM. Este caso es muy particular y generalmente solo tiene lugar cuando se trata de elementos <iframe>, ya que cada iframe tiene su propio *DOM*. Esta función se lanza automáticamente cuando se ejecuta la función de adoptNode().
- **attributeChangedCallback():** Se invoca cada vez que los atributos del elemento son modificados (añadidos, editados o eliminados). Los atributos que quieran ser monitorizados deberán estar previamente especificados a través del método estático *observedAttributes().*

2. Usando elementos personalizados

El controlador de los elementos personalizados en un documento web es el objeto **CustomElementRegistry** — este objeto te permite registrar un elemento personalizado en la página, devolver información de qué elementos personalizados se han registrado, etc.

Para registrar un elemento personalizado en la página, debes usar el método **CustomElementRegistry.define()** . Éste toma los siguientes argumentos:

- Un **DOMString** que representa el nombre que estás dando al elemento. Nótese que los nombres de los elementos personalizados deben contener un guión (kebab-case). Los nombres no pueden ser palabras simples.
- Un **objeto class** que define el comportamiento del ejemplo.
- Opcionalmente, **un objeto de opciones** que contiene la propiedad extends , que especifica el elemento preconstruido del que hereda (solo es relevante para elementos personalizados preconstruidos).

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

2.1. Elementos personalizados autónomos

Los **elementos personalizados** autónomos no heredan de elementos estándar HTML. Se usan estos elementos en una página escribiéndolos literalmente como un elemento HTML nuevo.

```
class PopUpInfo extends HTMLElement {
    constructor() {
      // Siempre llamar primero a super en el constructor
      super();}
```

constructor() para la clase, que siempre empieza llamando a super() de forma que se establezca correctamente el encadenado del prototipo.

2.1. Elementos personalizados autónomos

Dentro del constructor, definimos toda la funcionalidad que tendrá el elemento cuando se instancie. En este caso, adjuntamos una shadow root al elemento personalizado, mediante manipulación de DOM creamos la estructura interna del shadow DOM del elemento — que se adjunta a su vez a la shadow root — y finalmente añadimos algo de estilos CSS al shadow root.

```
class PopUpInfo extends HTMLElement {
    constructor() {
      // Siempre llamar primero a super en el constructor
      super();}

      // La funcionalidad del elemento se escribe aquí
      connectedCallback() {
        // Create a shadow root
        const shadow = this.attachShadow({ mode: "open" });

        // Create spans
        const wrapper = document.createElement("span");
        wrapper.setAttribute("class", "wrapper");

        const icon = document.createElement("span");
        icon.setAttribute("class", "icon");
        icon.setAttribute("tabindex", 0);

        const info = document.createElement("span");
        info.setAttribute("class", "info");

        // Take attribute content and put it inside the info span
        const text = this.getAttribute("data-text");
        info.textContent = text;
```

2.1. Elementos personalizados autónomos

Finalmente, registraremos nuestro elemento en el CustomElementRegistry usando el método define() que mencionamos anteriormente — en los parámetros especificamos el nombre del elemento, y el nombre de la clase que define su funcionalidad.

```
customElements.define("popup-info", PopUpInfo);
```

Llamada en html:

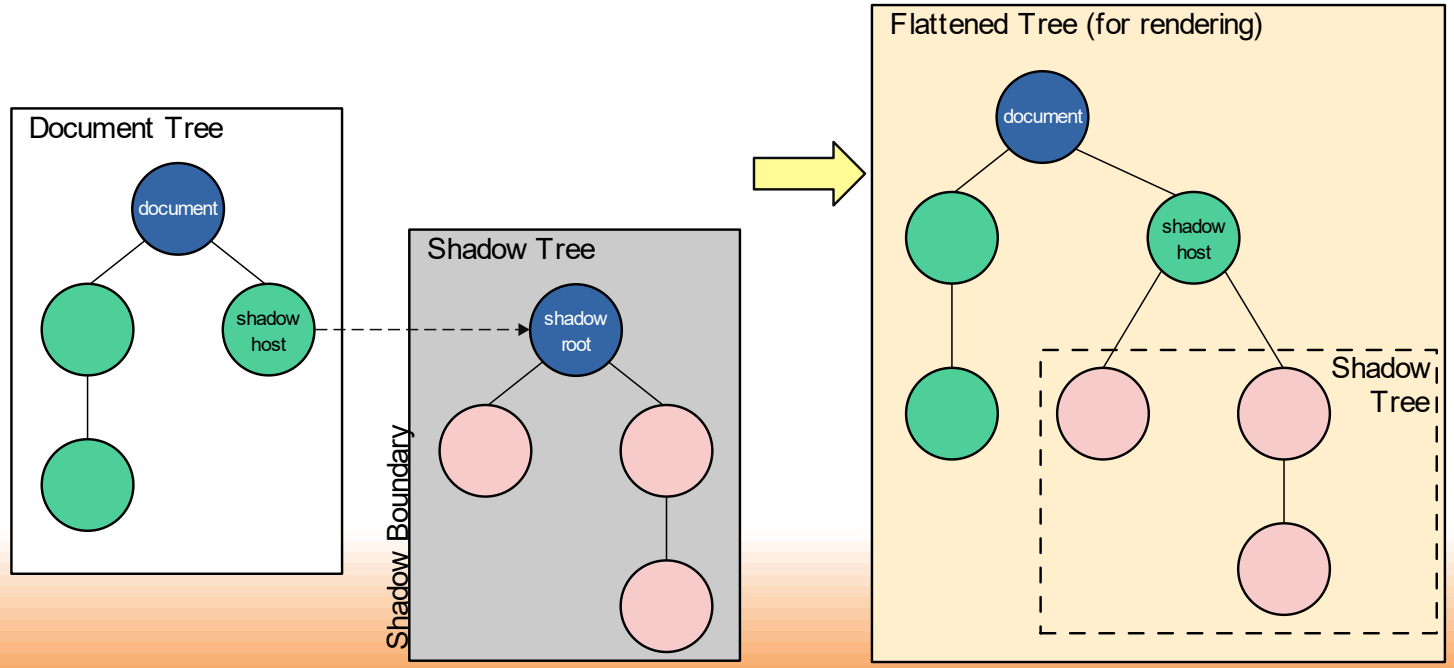
```
<popup-info
  img="img/alt.png"
  data-text="Your card validation code (CVC)
  is an extra security feature — it is the last 3 or 4 numbers on the
  back of your card."></popup-info>
```

2.1. Estilos internos o estilos externos CSS

Si aplicamos estilos al **Shadow DOM** usando el elemento <style>, pero podríamos perfectamente hacerlo referenciando una hoja de estilos externa mediante un elemento <link>.

2.1.1 Shadow DOM

Shadow DOM permite adjuntar arboles DOM ocultos a elementos en el árbol DOM regular — este árbol shadow DOM comienza con un elemento **shadow root,** debajo del cual se puede adjuntar cualquier elemento que desee, de la misma manera que el DOM normal.



2.1.1 Shadow DOM

Hay algunos conceptos de Shadow DOM que deben ser tomados en cuenta:

- **Shadow host:** El nodo regular del DOM al que es atado el shadow DOM.
- **Shadow tree:** El arbol DOM dentro del shadow DOM.
- **Shadow boundary:** El punto en el que el shadow DOM termina y el DOM regular comienza.
- **Shadow root:** El nodo raiz del arbol Shadow.

Se puede manipular los nodos del 'shadow DOM' de la misma manera que los nodos del arbol DOM regular. Por ejemplo, agregando hijos o estableciendo atributos, dando estilo a nodos individuales utilizando element.style.foo, o agregando estilo a todo el árbol de 'shadow DOM' dentro del elemento <style>. La diferencia es que nada del código dentro de un 'shadow DOM' puede afectar a nada fuera de él, lo que permite una encapsulación práctica.

2.1.1 Shadow DOM

Puede adjuntar un 'shadow root' a cualquier elemento utilizando el método Element.attachShadow (). Éste toma como parámetro un objeto que contiene una propiedad — modo — con dos posibles valores: 'open' o 'closed'.

```
let shadow = elementRef.attachShadow({ mode: "open" });
let shadow = elementRef.attachShadow({ mode: "closed" });
```

open significa que puede acceder al shadow DOM usando JavaScript en el contexto principal de la página . Ejemplo: **let myShadowDom = myCustomElem.shadowRoot;**

close no podrás acceder al shadow DOM desde fuera. Ejemplo: **myCustomElem.shadowRoot returns null**

2.1.1 Shadow DOM

Si adherimos un shadow DOM a un componente personalizado en su contructor (una aplicación muy útil y uno de sus propósitos esenciales), se emplea de la siguiente manera:

```
let shadow = this.attachShadow({ mode: "open" });
```

Cuando se añade un shadow DOM a un element, manipularlo es análogo a la manupulación del DOM con su correspondiente API. Ejemplo de la inclusión de un element tipo párrado en un shadow DOM.

```
var para = document.createElement("p");
shadow.appendChild(para);
// etc.
```

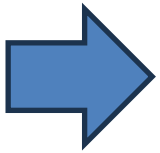
2.3. Elementos preconstruidos personalizados

Los **elementos precontruidos personalizados** heredan de elementos HTML básicos. Para crear un elemento de este tipo, tienes que especificar de qué elemento extiende (como se verá en los ejemplos de abajo), y se usan escribiendo el nombre del elemento básico, pero añadiendo un atributo (o propiedad) **is** y dándole como valor el nombre del elemento personalizado que se ha desarrollado.

```
class ExpandingList extends HTMLUListElement {
  constructor() {
    // Siempre llamar primero a super en el constructor
    super();

    // La funcionalidad del elemento se escribe aquí

    ...
  }
}
```



```
<ul is="expanding-list">
  ...
</ul>
```

```
customElements.define("expanding-list", ExpandingList, { extends: "ul"
});
```

3. Templates

El elemento HTML **<template>** es un mecanismo para mantener el contenido HTML del lado del cliente que no se renderiza cuando se carga una página, pero que posteriormente puede ser instanciado durante el tiempo de ejecución empleando JavaScript.

```
<!-- Lo definido en template, no aparecerá en la página -->
<template id="my-paragraph">
    <p>Mi párrafo</p>
</template>
```

Deberá ser incluído por JavaScript, por ejemplo:

```
let template = document.getElementById("my-paragraph");
let templateContent = template.content;

//3. Un línea así, tiene que aparecer.La inclusión en el DOM
document.body.appendChild(templateContent);
```

3.1. Templates en componentes

Definición del template:

```
<!--2. Definición del template -->
    <template id="my-paragraph">
    <style>
        p {
        color: white;
        background-color: #666;
        padding: 5px;
        }
    </style>
    <p>Mi párrafo</p>
    </template>
```

Creación del componente a partir del template en JavaScript.

```
customElements.define(
    "my-paragraph",
    class extends HTMLElement {
      constructor() {
        super();
        let template = document.getElementById("my-paragraph");
        let templateContent = template.content;

        const shadowRoot = this.attachShadow({ mode: "open" })
            .appendChild(
                templateContent.cloneNode(true),
        );
      }
    },
  );
```

El punto clave a tener en cuenta aquí es que agregamos **un clon** del contenido de la plantilla al shadow root creado usando el método Node.cloneNode(). Y debido a que estamos agregando su contenido a un shadow DOM, podemos incluir cierta información de estilo dentro de la plantilla en un elemento <style>, que luego se encapsula dentro del elemento personalizado. Esto no funcionaría si nosotros solo lo agregásemos al DOM estándar.

4. Slot

El elemento HTML **<slot>** —parte de la suite tecnologica Web Components — es un placeholder en un componente que tu puedes llenar con tu propio marcado, que te permite crear árboles DOM por separado y presentarlos juntos.

```
<body>

    <h1>Template básico</h1>

  <template id="my-paragraph">
    <style>
      p {
        color: white;
        background-color: #666;
        padding: 5px;
      }
    </style>
    <p><slot name="my-text">Texto por defecto</slot></p>
  </template>

  <my-paragraph>
    <span slot="my-text">Un texto un poco diferente!</span>
  </my-paragraph>

  <my-paragraph>
    <ul slot="my-text">
      <li>Un texto un poco diferente!</li>
      <li>En una lista!</li>
    </ul>
  </my-paragraph>
</body>
```

Ejercicio

- Crea un componente con alguno de los métodos mostrados.