**CURSO BÁSICO DE WEB COMPONENTS**

**¿Qué problemas resuelven los Web Components?**

**Web components** son encapsulados de código, que coexisten entre si

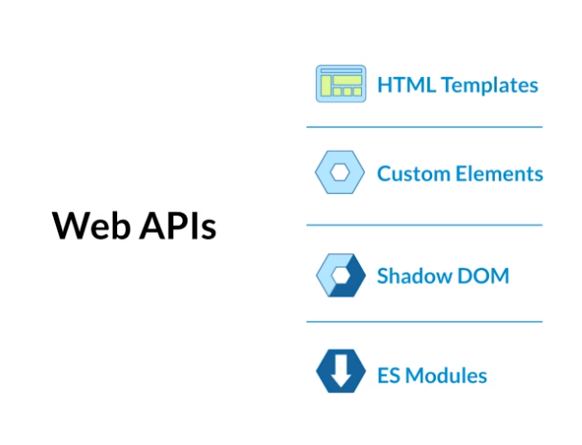
**Problema**

* El desarrollo web se hace más complicado
  + Actualmente se tiene un ecosistema gigante de JavaScript
    - Varias tecnologías no pueden co existir en un mismo proyecto a la vez, como ser angular con react
  + Entonces debemos elegir bien un stack de tecnologías
* Para solucionar este problema existen Web Components
  + Usan estándares web, para conseguir el mismo resultado e incluso mejor
  + No necesitamos cargar ninguna librería para obtener una arquitectura más escalable

**RESUMEN:** El desarrollo web actualmente se está volviendo complicado debido a que existen muchas tecnologías que si bien facilitan el desarrollo no están dejando usar otras para un mismo proyecto, es ahi donde entran los webs components para obtener el mismo resultado con web api’s

**¿Qué son los Web Components?**





Los web components están hechos para ser reutilizables, encapsulamos cierto código que podemos reutilizar en diferentes partes de nuestra aplicación o diferentes aplicaciones, sin tener que repetir el código.

El hecho de que son como piezas de lego no es algo tan literal ya que porque no se puede reutilizar los componentes, pero los web components si pueden ser reutilizados son hechos para eso y para no solo reutilizarlo en tu aplicación si no que en otras tecnologías también seria compatible.

Los web components son primitivos de bajo nivel que te permiten definir tus propios elementos HTML

Una vez tienes un componente listo es como si tuvieras una etiqueta de html ya lista para ser usada

Los web components están construidos con web apis

* HTML Templates : etiquetas
* Custom Elements: algo que nos ayuda a definir una etiqueta que se convierte en una etiqueta estándar después
* Shadow DOM: es lo que nos ayuda a encapsular
* Es Modules: Los módulos que nos ayudan a importar cierto código de js a otro código de js y reutilizar ciertas cosas

**APIs de Web Components**

**HTML Standard customelements**

Permite definir la etiqueta y hacer que el web component se pueda convertir en una etiqueta que el navegador pueda entender.

Estas etiquetas personalizadas o custom element, se deben crear con cautela puesto que quienes estandarizan HTML están constantemente creando nuevas etiquetas y para que no exista una conflicto las etiquetas personalizadas deben tener mínimo 2 palabras separadas por un -, ejemplo, mapa-geopolitico, la-etiqueta-mas-personalizada-del-mundo.

**Shadow DOM**

Normalmente se confunde con el virtual DOM que algunas framework y librerías tienen.

El Shadow DOM es generar un encapsulado perfecto permitiendo etiqueta contener código que no coexista con el código de a fuera, imposibilitando la sobre escritura de los estilos.

**Custom Elements**

Son APIS de JS que permite definir elementos personalizados y su comportamiento, que entonces puede ser utilizado como deseé en la interfaz del usuario.

**HTML Templates**

Es una sola etiqueta <template></template> usada junto con JavaScript. Si utilizamos esta etiqueta en nuestra arquitectura HTML no se va a renderizar y devolverá un document Fragment, ósea es un fragmento del documento y contenido será clonado y renderizado en el HTML.

Por esto debe ser combinado con JS para generar esta clonación y renderización del componente.

Este proceso agrega mucho performance en el momento del renderizar los nodos en el DOM.

**ES Modules**

Anteriormente existía una API web llamada HTML Imports que permitía introducir código HTML en otro código HTML, pero no logro ser un estándar en todos los navegadores por lo cual desistieron de el.

Con la llegada de los módulos de ECMAScript 6 lograron exporta e importar código con JavaScript.

Las 4 web APIS son un conjuntos de estándares que facilitan el desarrollo de componentes sin la utilización de frameworks o librerías, otorgando un mayor performance al DOM.

**RESUMEN:** Para crear web components se necesitan cuatro tecnologías, Custom Elements para crear etiquetas de HTML, Shadow DOM para no tener conflictos con el código externo y pueda vivir nuestro código en diferentes lugares y finalmente ES Modules para poder reutilizar el código.

**Beneficios de Web Components**

* **Reutilización**
* **Legibilidad**
* **Mantenibilidad**
* **Interoperabilidad**
* **Consistencia**

**REUTILIZACIÓN**

* Dont Repeat Yourself, no repitas el código que ya hiciste, en el tema de web components se utiliza mucho el término donde usamos el código solo 1 vez.
* Cuando logres construir un componente lo puedes usar en cualquier tipo de aplicación sin importar el framework o librería que uses.

**LEGIBILIDAD**

* Si tenemos programas muy extensos la legibilidad se empieza a ver afectada por la cantidad de líneas de código.
* Trataríamos de hacer una estructura con nombres entendibles y de esta forma contribuimos ala semántica.

**MANTENIBILIDAD**

* Cada uno de los componentes pueden ser escritos y probados de forma individual.
* No debemos preocuparnos por si rompemos algo y que toda nuestra aplicación se dañe, esto también suele pasar con algunos framework y libreras de JavaScript.
* Resulta en menos líneas de código y aumentar su legibilidad y su mantenibilidad.

**INTEROPERABILIDAD**

* Los frameworks y librerías no están hechos para coexistir entre ellos, por lo que si quieres mezclar Angular con React es imposible.
* Pero los componentes web si, están hechos para coexistir con todo el ecosistema de desarrollo web.

**CONSISTENCIA**

Gracias a la naturaleza reutilizable e interoperable de los web components ya no tendrás que crear los mismos componentes en diferentes frameworks o librerías.

**Ciclo de vida de un componente**

El ciclo de vida de los Web Components está 100% ligado al DOM. Son parte fundamental del Critical Rendering Path.

**Constructor**

Las clases tienen un constructor(), este es el paso #1 del ciclo del vida. El CRP con el Engine de JS guarda en memoria lo que tiene el constructor. Todos los componentes tienen que tener un constructor.

Directamente desde el JavaScript Engine, el constructor nos servirá para definir y cargar todas las variables en memoria que necesitemos, es mala práctica pintar el componente aquí

* Son parte del critical Render Path
* Las clases cuando generamos un constructor generamos están guardando en memoria que es lo que tiene el constructor
  + Todos los componentes deben tener un constructor
  + Una mala práctica es pintar el template directamente el template en el constructor

En el constructor solo se debe asegurar todo lo que está en memoria exista para pintar el componente

***MALA PRÁCTICA:*** hacer que en el constructor se pinte el template. En el constructor sólo tenemos que asegurarnos de que todo lo que esté en memoria exista en ese momento para usarlo en el connectedCallback().

**connectedCallback**

* Cuando el componente ya está pintado dentro del DOM y podemos hacer uso de él.
  + Es cuando ya es un nodo del DOM
  + Aqui es donde se pueden renderizar el HTML y CSS

**disconnectedCallback**

* + Es cuando quitamos del DOM (desconectar) el componente
  + Cuando queramos liberar memoria los cuales está consumiendo nuestro componente

**attibuteChangedCallback**

* + Cuando exista un cambio de los atributos del componente desde la etiqueta del HTML, este método nos avisará que cambiaron

**adoptedCallback**

* + Cuando un componente va a ser adoptado en otra cosa como ser un iframe
  + No es muy usado y es mala práctica usar iframes.
  + Cuando el componente es movido a un nuevo DOM, básicamente cuando es pintado desde un iframe por ejemplo
  + Un ciclo de vida que casi nunca vamos a utilizar es **adoptedCallback()** porque solamente se utiliza cuando usamos un componente dentro de iframe (y usarlo es una MUY MALA PRÁCTICA).

**WEB COMPONENTS**

**1.-Custom Elements**

**Vamos a crear nuestro primer Web Component 😄**

Creando el componente

En vs-code creamos un archivo HTML y JS :

Después de crear nuestra estructura básica de HTML vamos a nuestro archivo de JavaScript y crearemos una clase que será nuestro componente:

**class** **MyElement** **extends** **HTMLElement** {

**constructor** ( ) {

// Heredamos las propiedades de HTML element con "super()"

super();

}

}

customElements.define('my-element', MyElement);

Para crear un componente primero debemos crear una clase que extienda a lago llamado el HTMLElement, el cual nos va a permitir definir nuestro componente web.

Para convertir nuestro componente en una etiqueta HTML debemos usar el método estatico de customElements llamado define( ), el cual recibe dos parámetros, el primero el nombre de como se llamará la etiqueta de nuestro componente, y el segundo la clase que utilizaremos para definir los comportamientos de nuestro componente .

En este punto ya generamos nuestra etiqueta, y podemos usarla en el HTML:

<!DOCTYPE html>

<html lang="es">

<head>

...

</head>

<body>

\*\*<my-element></my-element>\*\*

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

</body>

</html>

**Lógica de nuestro componente**

Ahora vamos a empezar a definir los comportamientos de nuestro componente:

**class** **MyElement** **extends** **HTMLElement** {

**constructor** ( ) {

super();

// Creamos una etiqueta p

this.p = document.createElement('p');

}

// Creamos el método de conenxión

**connectedCallback**() {

// Introducimos texto a nuestra etiqueta

this.p.textContent = 'Hola mundo';

// Agregamos esta etiqueta como hijo de <my-element>

this.appendChild(this.p)

}

}

customElements.define('my-element', MyElement);

Si vemos nuestro archivo en el navegador podremos ver ya nuestro “Hola mundo”.

**Complejidad de componente**

Podemos añadir muchas más cosas a nuestro componente, tales como estilos y demás, veamos un ejemplo.

// Creamos un div

**const** template = document.createElement('div');

/\* Dentro de este div podemos escribir estilos y demás cosas,

recordemos qu estos estilos no harán colición con otros estilos

debido a que los estilos que escribamos aquí solo aplican a nuestro

componente y NO a otros elementos externos a este

\*/

template.innerHTML = `

<style>

p {

color: #9C27B0

}

.text {

font-size: 1rem;

color: #212121;

}

</style>

<p>Hola mundo 2</p>

<p class="text">Texto de ejemplo</p>

`

**class** **MyElement** **extends** **HTMLElement** {

**constructor** ( ) {

super();

this.p = document.createElement('p');

}

**connectedCallback**() {

this.p.textContent = 'Hola mundo';

this.appendChild(this.p);

// Añadimos nuestro template a nuestro componente como hijo

this.appendChild(template)

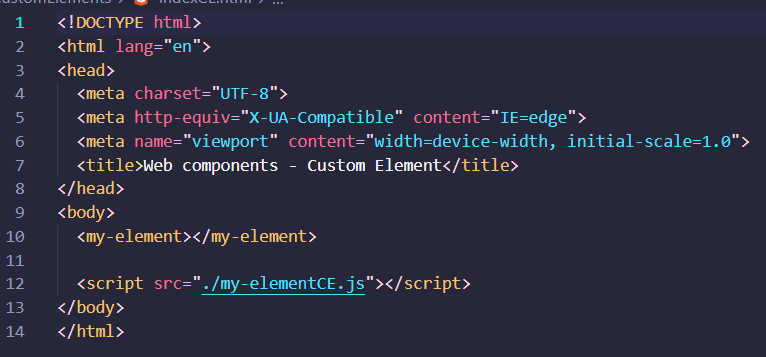
}

}

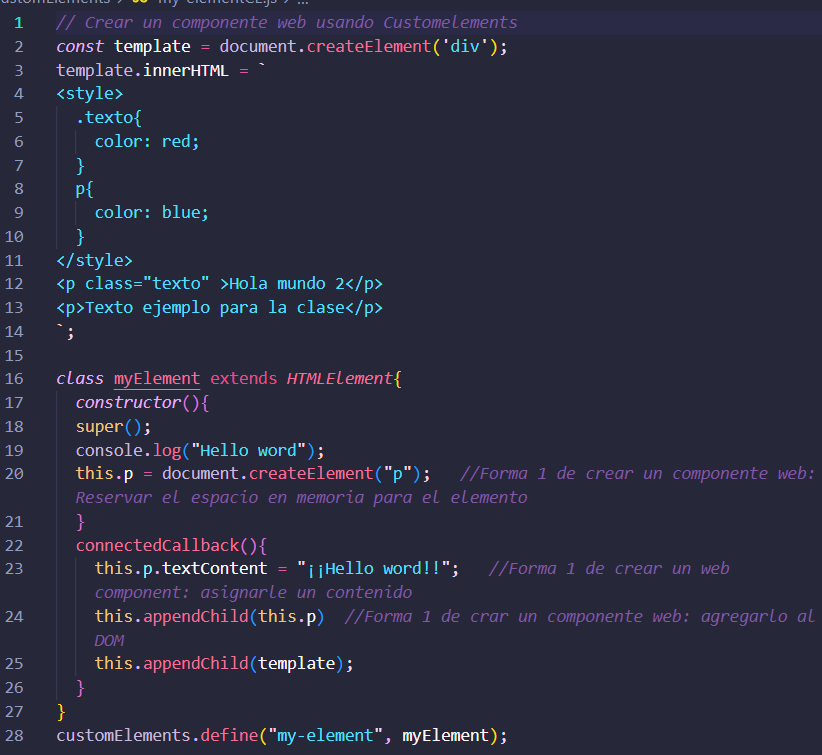
customElements.define('my-element', MyElement);

Y si todo está correcto, veremos nuestro componente en el navegador con todo lo que definimos

HTML



Javascript



**2.-Templates**

Es opcional puedfes ocuparlo o no. Pero si dentro del componente se van a generar varios nodos lo indicado sería usar template ya que se generaría el renderizado una sola vez y no varias veces con cada nodo sin el template

Basicamente la API Template nos permite conectar un web component de forma más profesional y organizada. También nos ayuda a clonar los elementos facilmente *(Ya que como lo hicimos en la clase anterior el elemento no se clonaba, sino que se pasaba de etiqueta en etiqueta hasta la ultima en ser renderizada)*

**La etiqueta <template>**

Es una etiqueta que nos sirve como contenedor de código. Todo lo que escribamos adentro de esta etiqueta no se va a renderizar , sino que hay que activarlo mediante Javascript. El profesor sin embargo no muestra cual es dicho proceso y procede a utilizar una forma distinta. En el siguiente enlace vas a ver cómo se activa desde JS:

**Escribir y activar el código dentro de la clase**

De esta forma estamos armando toda la estructura HTML dentro de Javascript, pero insertandola en la clase y fraccionando el HTML y CSS para más placer.

En este caso, creamos la clase, con su extension y constructor, luego creamos un método que contendrá la estructura HTML (getElement) . Adentro insertamos la variable template que contiene la estructura.

getTemplate() {

**const** **template** = document.createElement('template');

**template**.innerHTML = `

...(codigo HTML)

`

}

En otro metodo (getStyles) todo lo que hacemos es retornar un literal string que contiene el código CSS (si queremos podemos contenerla en una variable, eso es a comodidad del programador)

getStyles() {

**return** `...(código CSS)`

}

y luego al final del código de getElement la llamamos de esta forma

**${this.getStyles()}**

**Clonar Elementos**

Para clonar el código debemos indicar mediante el método cloneNode que se puede clonar. Para eso invocamos el contenido de getTemplate, y lo anidamos a la clase (que luego al ser invocada en el HTML se convierte en la etiqueta misma)

**render**() { this.appendChild(this.getTemplate().content.cloneNode(true)) }

**Y FINALMENTE…**

Invocamos el render

**connectedCallback() { this.render(); }**

**para los que no entendieron lo de “template”.**

**lo que pasa es que solo la etiqueta “template” tiene la propiedad “content” con la que puedes hacer el “cloneNode(true)”.**

**this.getTemplate().content.cloneNode(true)**

**HTML**



**Javascript**



**3.-Shadow DOM**

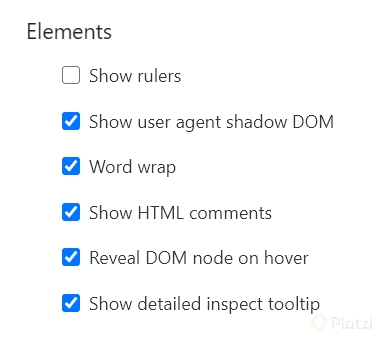
Con esta implementación podremos solucionar los problemas con los estilos css que se rescriben por temas de especificidad.

Generando como un encapsulado, un dom independiente dentro de nuestro dom global, esto qué quiere decir?... que todo lo que coexista en nuestro dom independiente no va existir dentro de nuestro dom global.

Esto nos ayudará evitar problemas de estilos reescribiendose. Esto se logrará por el encapsulado.

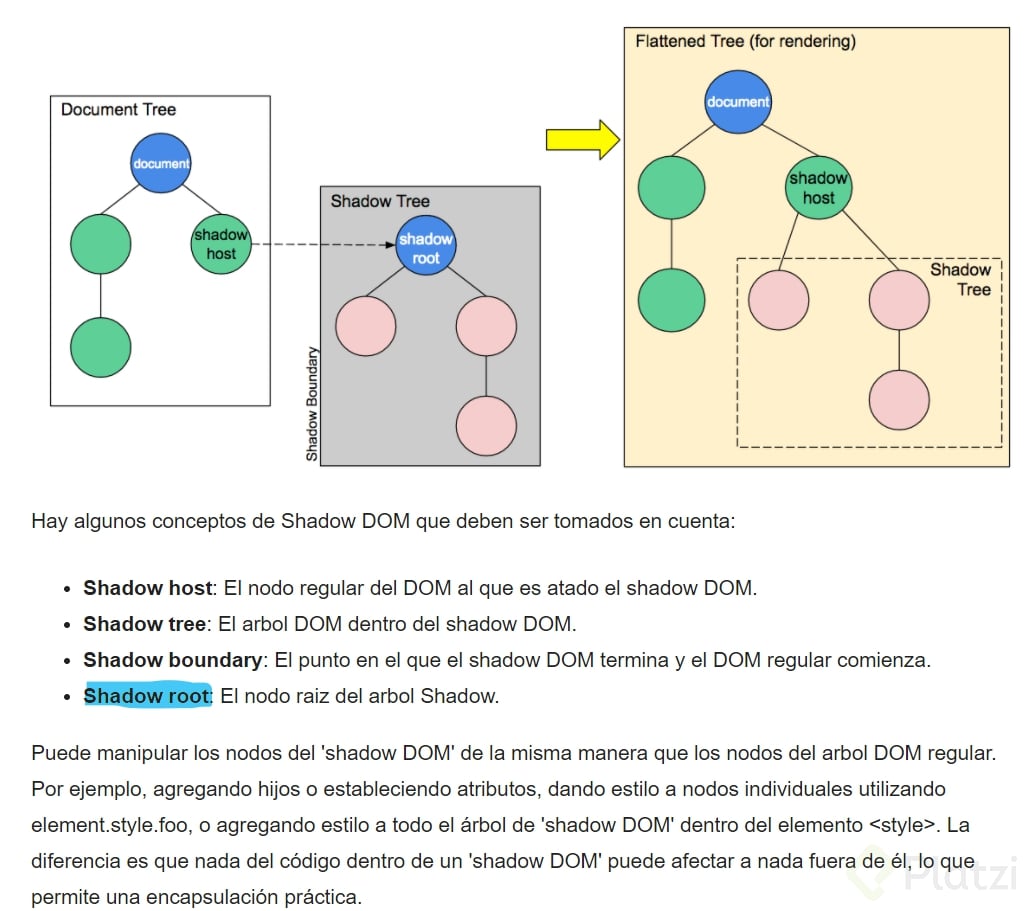
Pensemos en ShadowDOM como **un DOM independiente** adentro del DOM global, es por esto que los estilos estilos serán independientes en cada uno.

Es importante activar el shadow dom para poderlo visualmente en nuestro inspector de elementos



El API de [DOM Shadow](https://developer.mozilla.org/es/docs/Web/Web_Components/Using_shadow_DOM) es un parte clave para esto, proporcionando una forma de enlazar un DOM oculto y separado a un elemento. Este artículo cubre los aspectos básicos para utilizar Shadow DOM.  
ㅤ  
El *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.  
ㅤ  
Hay algunos conceptos de Shadow DOM que deben ser tomados en cuenta:

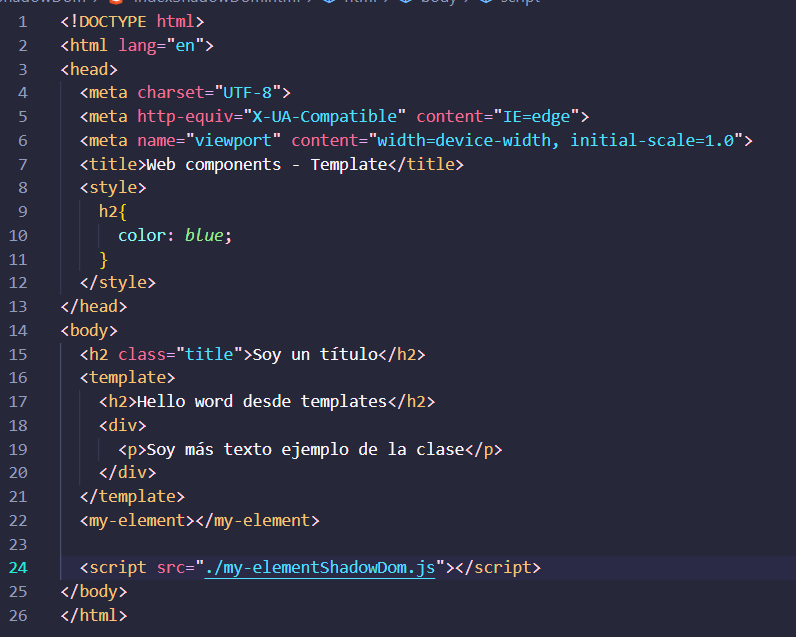
* **Shadow host**: El nodo regular del DOM al que es atado el shadow DOM.
* **Shadow tree**: El árbol DOM dentro del shadow DOM.
* **Shadow boundary**: El punto en el que el shadow DOM termina y el DOM regular comienza.
* **Shadow root**: El nodo raíz del árbol Shadow.  
  ㅤ



Puede adjuntar un **shadow root** a cualquier elemento utilizamos el método Element.attachShadow (). Éste toma como parámetro un objeto que contiene una propiedad mode con dos posibles valores: open o closed.  
ㅤ  
open: Significa que puede acceder al shadow DOM usando JavaScript en el contexto principal de la página. Por ejemplo, usando la propiedad [Element.shadowRoot](https://developer.mozilla.org/en-US/docs/Web/API/Element/shadowRoot)

closed: Significa que no se puede acceder al shadow Dom desde afuera, como con la etiqueta video. Pero esto cierra oportunidad de poder reutilizar el componente.

HTML



Javascript



**MANEJO DE DATOS**

**Content Slot**

Para aprender como hacer que nuestros componentes sean totalmente reutilizables debemos aprender a manejar los datos dinámicos desde afuera de los componentes, esto lo podemos lograr con las etiquetas especificas y traerlas.

En este punto vamos a aprender a usar la etiqueta content-slot.

De buenas a primeras debemos lograr que podamos escribir algo por fuera de las etiquetas y que sea parte del #shadow-root, ya que si ahora lo hacemos pues no va a pasar:

<body>

<my-element>

<!-- Esto no hara parte del #shadow-root ❌ (de momento) -->

Hola que hace

</my-element>

</body>

Para lograr que se vea representado dentro de una etiqueta junto con la lógica del #shadow-root debemos usar la etiqueta <slot>.

Veamos un ejemplo:

**class** **MyElement** **extends** **HTMLElement** {

**constructor** ( ) {

...

}

**getTemplate**() {

**const** template = document.createElement('template');

template.innerHTML = `

<section>

<h2>

\*\*<slot></slot>\*\*

</h2>

</section>

${this.getStyles()}

`

**return** template;

}

...

}

Lo que hace la etiqueta slot es que actúa como un placeholder en el cual nosotros podemos introducir contenido, este contenido lo puede dar el usuario si así se programara, de esta forma ya se inyectaría lo que introdujéramos en la etiqueta HTML  
Ósea que algo como esto ya tendría sentido:

<body>

<my-element>

<!-- Ahora si veríamos un resultado correcto con los estilos

y el comportamiento de nuestro #shadow-root como lo deseamos ✔ -->

Hola que hace

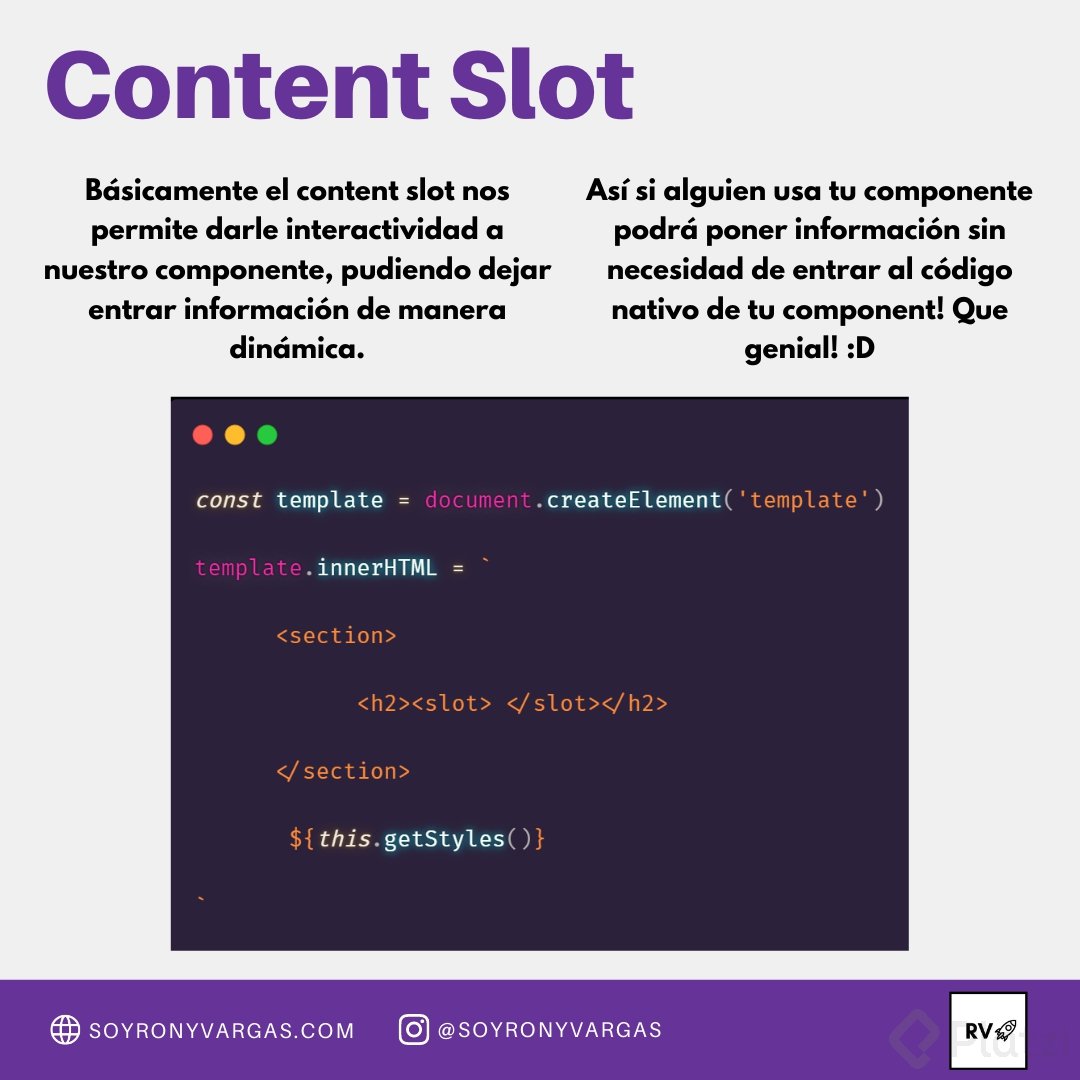
</my-element>

</body>

**<slot>**

Es una etiqueta de HTML 5 que nos va a ayudar a poder generar el placeholder en donde irá el texto o cierto contenido que necesito para que, afuera de le etiqueta, nosotros podamos pasarle contenido que el componente pueda renderizar. Slot se introduce en el template del componente en JS y el texto que deseamos renderizar en el HTML global dentro de nuestra etiqueta custom element donde agregamos nuestro compoente.

Básicamente modificamos la etiqueta en HTML y JS hará el trabajo sucio por nosotros.



**HTML**



**Javascript**



**Multi Content Slot**

Podemos tener multiples <slots> en un <template>. A esto le llamamos Multi Content Slot  
Para nombrar los <slots> usamos el atributo name:

<template>

<slot name="title"></slot>

<slot name="description"></slot>

</template>

Cuando necesites agregar añadir contenido en un <slot> en específico se usa el atributo slot (NAME Y SLOT ESTARÁN VINCULADAS CON EL MISMO NOMBRE):

<my-element>

<h2 slot="title">¡Nuevo Curso!</h2>

<p slot="description">Crea componentes reutilizables con Web Components.</p>

</my-element>



**Atributos**

Hay más maneras de manejar los datos dinámicos, una de estas es el uso de los atributos, podemos utilizar ciertos atributos de las etiquetas para obtener su valor y se pueda renderizar nuestro componente

Crearemos nuestros propios atributos en JS para colocarlos en la etiqueta HTML. Así mandamos información de manera sencilla.

**class** **MyElement** **extends** **HTMLElement** {

**constructor** ( ) {

...

// con getAttribute() podemos obtener un atributo de una etiqueta

\*\*this.title = this.getAttribute('title');

this.pharagraft = this.getAttribute('pharagraft');

this.img = this.getAttribute('img')\*\*

}

**getTemplate**() {

**const** template = document.createElement('template');

template.innerHTML = `

<section>

<h2>

\*\*${this.title}\*\*

</h2>

<div>

<p>\*\*${this.pharagraft}\*\*</p>

<img src="\*\*${this.img}\*\*" />

</div>

</section>

${this.getStyles()}

`

**return** template;

}

...

}

Luego en HTML simplemente podemos colocar ese atributo en nuestra etiqueta personalizada:

<my-element

\*\*title="Soy un título"\*\*

\*\*pharagraft="soy un párrrafo"

img="https://avatars3.githubusercontent.com/u/1905708?s=280&v=4/"\*\*

>

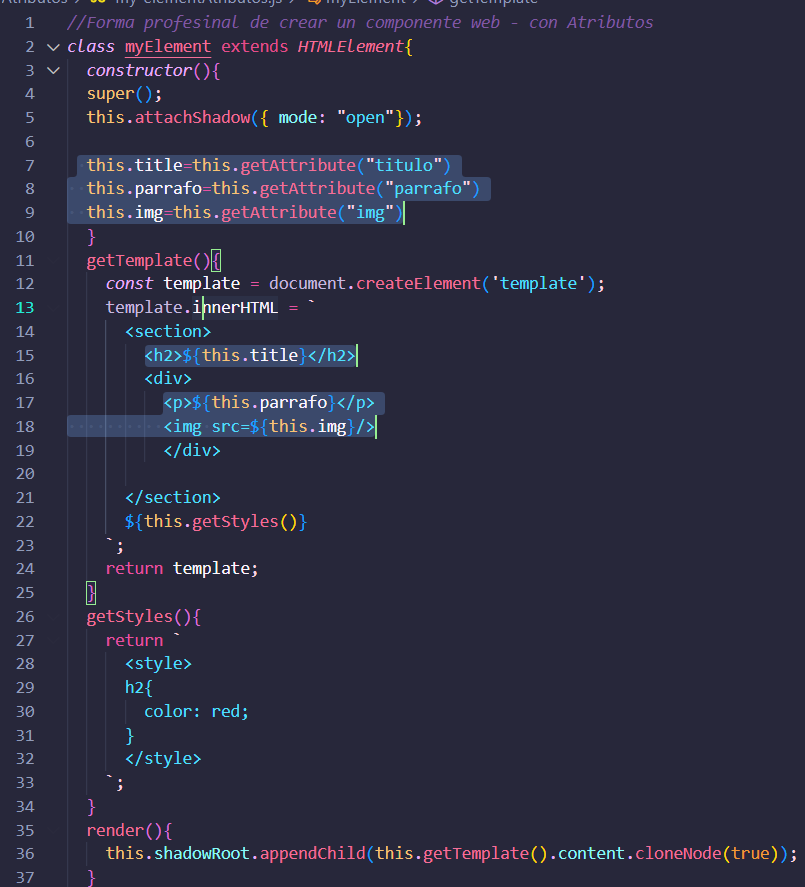
</my-element>

Esto ya nos renderizará un resultado en nuestro navegador. Así de fácil es de trabajar con los atributos en nuestros web components.

**HTML**



**Javascript**



**AttributeChangedCallback**

Veamos ahora como trabajar con este ciclo de vida.  
Un pequeño recordatorio es que debemos escribir las funciones exactamente como se nos esta pidiendo, ya que en caso contrario cuando el componente busque esos métodos no los encontrará y nuestro componente tendrá fallos de lógica.

Utilizaremos un observer, esto estará observando nuestros atributos.Es como indicarle al componente cuales son los atributos tendrá. Si hay algo en otro que no está en esta lista no es del componente

static get observedAttributes(){

return [‘title’, ‘parrafo’, ‘’img]

}

Aquí le decimos al observador que atributos tendrá nuestro componente. Si hay otro atributo que no este aquí este no se le dará importancia y no se le toma´ra en cuenta para escuchar sus cambios.

Ya teniendo los atributos que serán observados podremos utilizar el attributeChangedCallback

Esta función recibe tres parámetros:

1. valor actual

2. valor viejo

3. nuevo valor

attributeChangedCallback(attr, oldVal, newVal){

if(attr === ‘title’){

this.title =newVal;

}

}

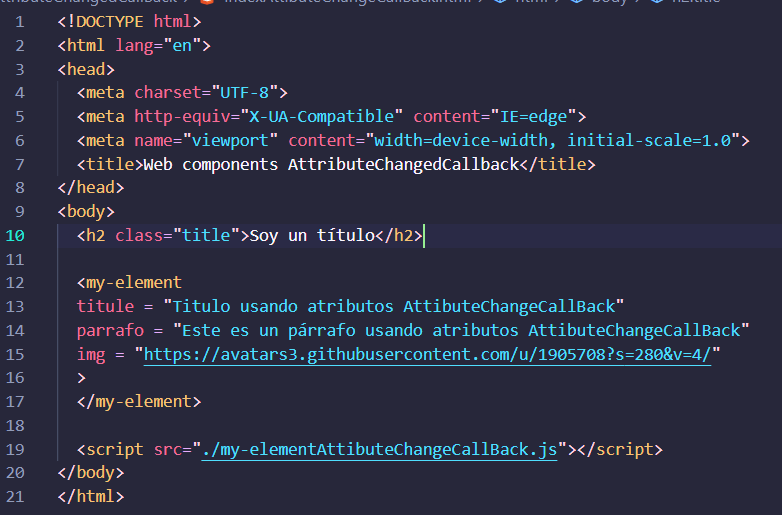
Aquí vamos a generar los cambios de acuerdo a lo que existe en los atributos Aquí hacemos una validación para verificar si existe el atributo en nuestro componente

//le asignamos a la variable title que obtenemos desde el observador de //nuestro componente un nuevo valor //Esto quiere decir que si hay un nuevo valor se tiene que hacer el cambio de Forma dinámica

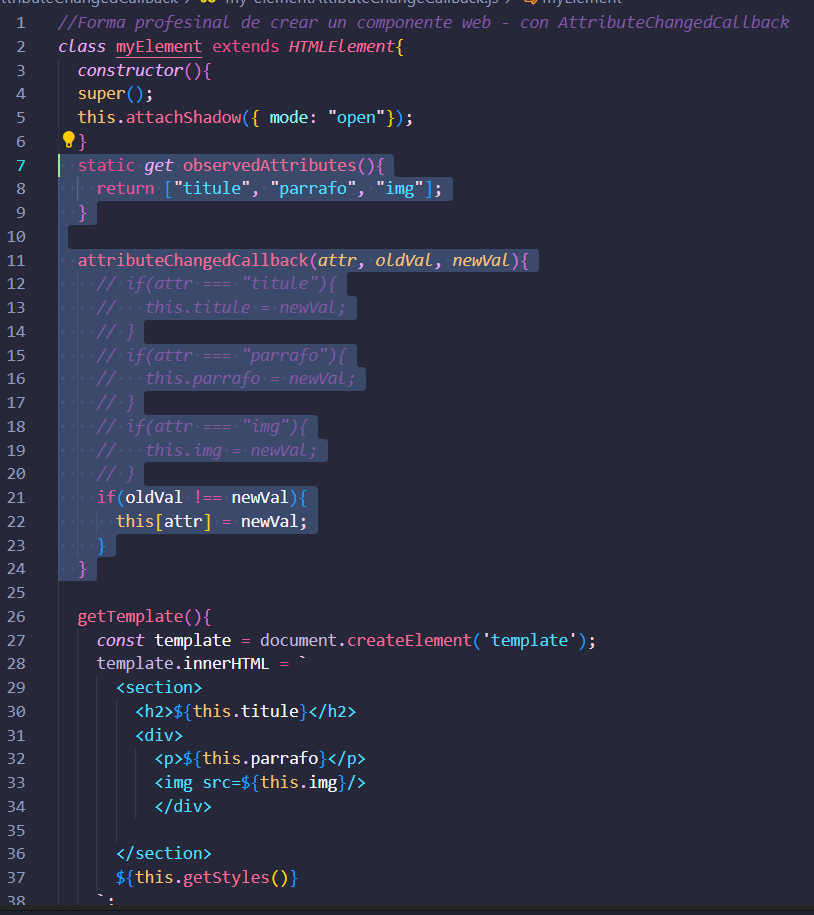
Como podemos ver, al cambiar atributos no se renderizan los cambios, es por eso que he decidido hacer un pequeño cambio en el metodo render() para utilizarlo también en el attributeChangedCallback()

Ahora en el momento que hagamos cambios en los valores de nuestro componente se cambiarán también los valores en tiempo real de este, y no tendremos valores desactualizados ya que los estaremos observando 👀.

HTML



Javascript



**DisconnectedCallback**

Disconnected callback nos servirá para deslindar de eventos a nuestros componente, dentro de esa función podremos retirarlos para liberar espacio en memoria.

La ventaja de usar este método, es que si por ejemplo usamos remove() tendríamos problemas con los nodos hijos conectados al elemento a remover. Pero disconnectedCallback nos ayuda a deslindar esta conexión.

[Documentación de los callbacks de ciclo de vida](https://developer.mozilla.org/es/docs/Web/Web_Components/Using_custom_elements#usando_callbacks_de_ciclo_de_vida)

En el momento en que tengamos que remover un componente antes de quitarlo debemos de eliminar todas las funciones o variables propias del elemento para eso nos sirve el **disconnectedCallback(){}**una vez hecho esto podemos eliminar nuestro componente del DOM con **remove()**

document.querySelector("etiqueta-del-componente").remove()

**ESTRUCTURA COMPLETA DEL CICLO DE VIDA DE UN COMPONENTE**

**-----------------------------------------------------------------------------------------------------**

**class** **MyCustomElement** **extends** **HTMLElement**{

constructor(){

**super**()

//Aqui estamos establecieno en memoria las cosas que vamos a utilizar

//Para nuestro componente

console.log('hola desde el constructor - **Memoria**')

//Pero esto no quiere decir que ya exista en el dom

}

//Para que exista en el dom llamaremos a nuestro siguiente ciclo de via

connectedCallback(){

//Nuestro elemento ya existe como nodo en el dom entonces ya podemos generar

//la interaccion

console.log("hola desde el dom");

}

disconnectedCallback(){

//Tercer ciclo de vida esto quiere decir que estamos retirando el elemento del dom

console.log("adios del DOM");

}

}

//Definimos nuestro componente para utilizarlo como etiqueta

customElements.define("my-custome-element", **MyCustomElement**);

//Retiramos el elemento del dom seleccionandolo con un query selector

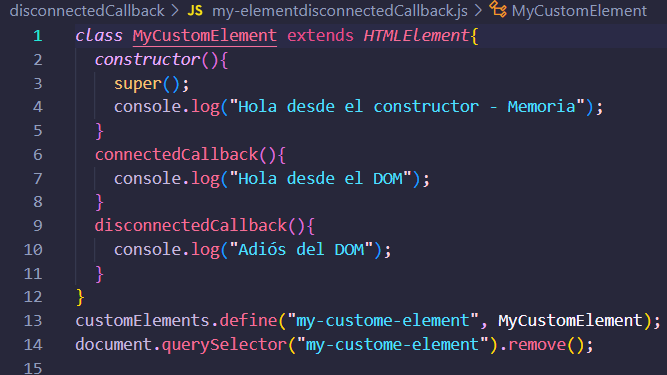
//Y removiendolo con la funcion remove que sirve para eliminar nodos.

document.querySelector("my-custome-element").remove();

**HTML**



**Javascript**



**ESTILOS EN COMPONENTES**

**:host**

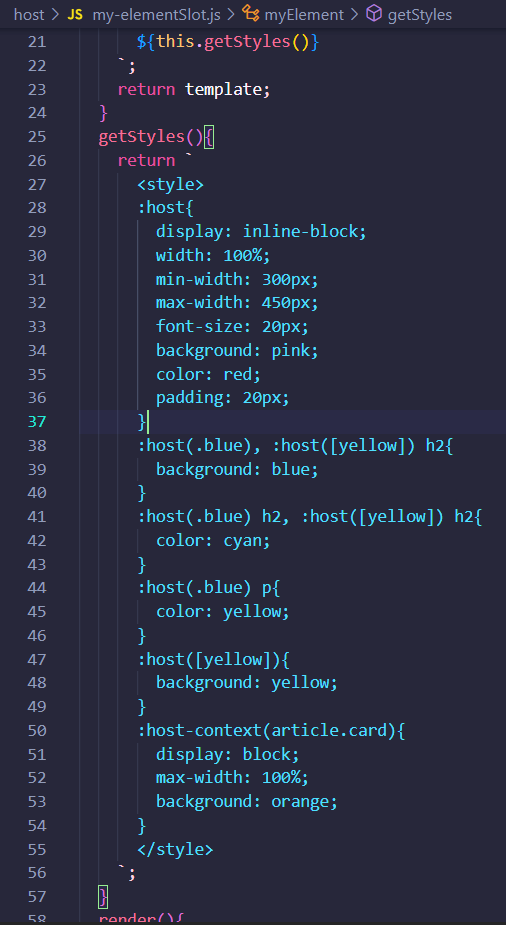
La [pseudo-clase](https://developer.mozilla.org/es/docs/Web/CSS/Pseudo-classes) [CSS](https://developer.mozilla.org/es/docs/Web/CSS) :host selecciona la sombra host de [sombra DOM](https://developer.mozilla.org/es/docs/Web/Web_Components/Using_shadow_DOM) que contiene el CSS que se usa en el interior, esto le permite seleccionar un elemento personalizado desde su sombra DOM.

Pseudoclase que utilizaremos para darle estilos a nuestro componente web (no se trata necesariamente de los estilos visuales).  
  
Se trata de los estilos que vienen definidos por default con una etiqueta, como pueden ser display, padding y margin.  
  
*:host da estilos al componente*  
  
La pseudoclase :host se utiliza dentro del método donde escribíamos nuestro css del componente getStyles(){}  
  
:host{estilos para el componente}  
  
Teniendo varias instancias de un componente, si a una le agregamos una clase por ejemplo ‘blue’  
:host(.blue) {estilos para el componente con la clase blue}  
Va a buscar el elemento que tenga de atributo una clase con el valor blue y le va a agregar los estilos que definimos.  
  
También podemos darle estilos por atributo. Por ejemplo si a una instancia le agregamos el atributo ‘yellow’  
:host([yellow]) {estilos para el elemento que tenga el atributo yellow}  
  
También podemos agregar cierto contexto.  
Por ejemplo, si tenemos una instancia del componente dentro de un article con una clase ‘card’  
:host-context(article.card) {estilos}  
  
Hacer cambios al contenido del componente  
:host([yellow]) h1 {estilos}

**HTML**



**Javascript**



**::slotted**

Pseudoelemento que sirve para poder agregar estilos específicos a todo el contenido dinámico que venga desde fuera del componente (slots) y se vaya a colocar en las etiquetas slot.  
  
::slotted(que tipo de etiqueta viene por fuera) {estilos}  
  
Ejemplo ::slotted(span) {}  
  
Si queremos ser más específicos, podemos usar clases:  
::slotted(.texto) {}  
  
Beneficio  
Que los devs que usen el componente puedan modificar las cosas desde fuera sin tener que entrar al componente para cambiar los estilos.  
  
Este pseudoelemento solo va a funcionar cuando tengamos un shadow DOM

\*\*Tanto la pseudo Clase :host como el pseudo elemneto ::slotted solo se pueden usar dentro del shadow DOM \*\*

:host hace referencia al host del shadow DOM, por lo que en nuestro caso hace referencia a nuestro custom element.

::slotted hace referencia a los elementos que vienen desde afuera del shadow DOM. Podemos especificar el elemento que queremos aplicar los estilos con (etiquetas/clases/ids)

Este pseudoelemento (::slotted) solo va a funcionar cuando tengamos un shadow DOM  
  
:host le dara estilos al componente en el Shadow DOM, ::slotted le dara estilos al contenido dinámico que encontramos dentro del elemento slot.

En esta clase utilizaremos el pseudoelemento sloted que nos ayudara a agregarle estilos especificos a todo el contenido dinámico que venga fuera del componente y se vaya a agregar en todas las etiquetas slot que estan fuera de nuestro componente.

Este elemento solo se va a poder trabajar una vez estemos utilizando el shadow dom

Podremos usar la pseudo clase sloted con el selector \* para seleccionar todas las etiquetas que estan por fuera.

::slotted(\*){

}

::slotted(span){

font-size:30px;

color:red;

}

También podremos seleccionar exactamente a cual etiqueta esta por fuera agregarle estilos

::**sloted**(**etiquetaFuera**){estilos}

::slotted(span){

font-size:30px;

color:red;

}

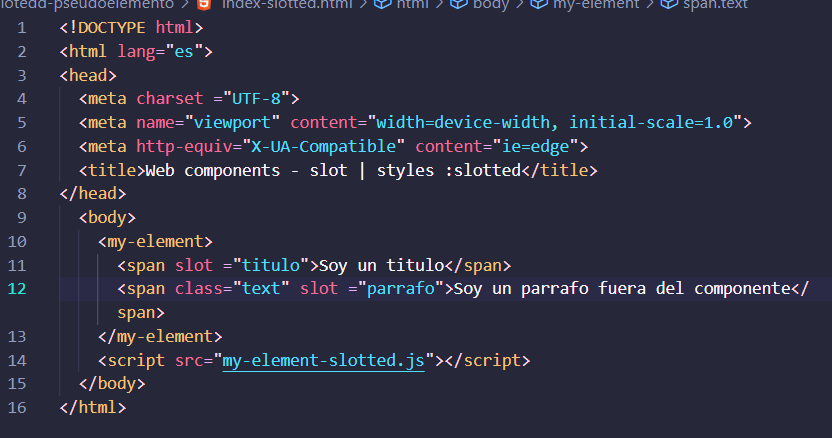
También podremos cambiar los estilos utilizando una clase que tenga la etiqueta

::slotted(.text){

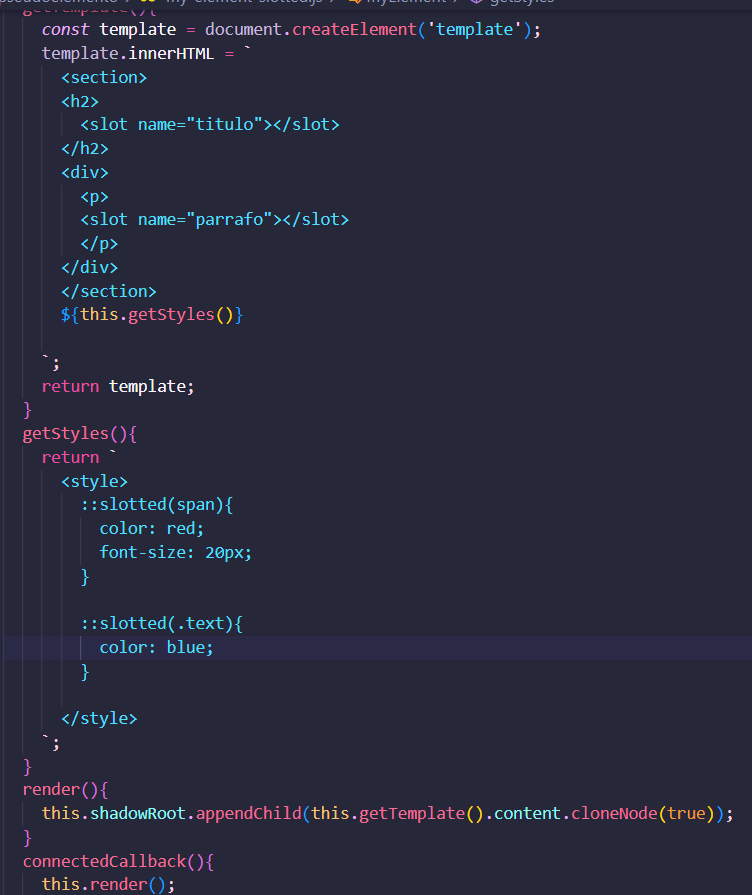
color:blue;

}

HTML



Javascript



**CSS custom properties**

Como se ha comentado en anteriores clases generar un componente reutilizable implica que no tenemos que ir hasta el código de este y cambiarlo, entonces lo que haremos es re asignar nuestros estilos por fuera.

Con shadow dom esta idea que se plantea de poder cambiar nuestros estilos por fuera no es posible, pero esto en cierto punto es mentira ya que podremos hacerlo atraves de las Custom Properties que son unas variables en donde las generamos y de valor estas tienen un estilo.

Lo que haremos es definir variables en nuestros host y luego agregárselas a nuestros elementos que estan dentro de los componentes para probarlas.

:host{

--primary-color:tomato;

--secundary-color:salmon;

--heading-primary: 30px;

--heading-secundary: 25px;

display: inline-block;

width:100%;

min-width:300px;

max-width: 450px;

}

**section**{

background:var(--primary-color);

}

**section** **div**{

background: var(--secundary-color);

}

**h1**{

font-size: var(--heading-primary)

}

**p**{

font-size: var(--heading-secondary)

}

Luego para poderlas modificar desde fuera del codigo establecido en el componente, lo que hacemos es modificarlas desde nuestro css, seleccionando lo que queremos modificar y reasignando el valor de la variable.

**my-element**{

--primary-color:blue;

--heading-primary:40px;

}

.segundo{

--primary-color:red;

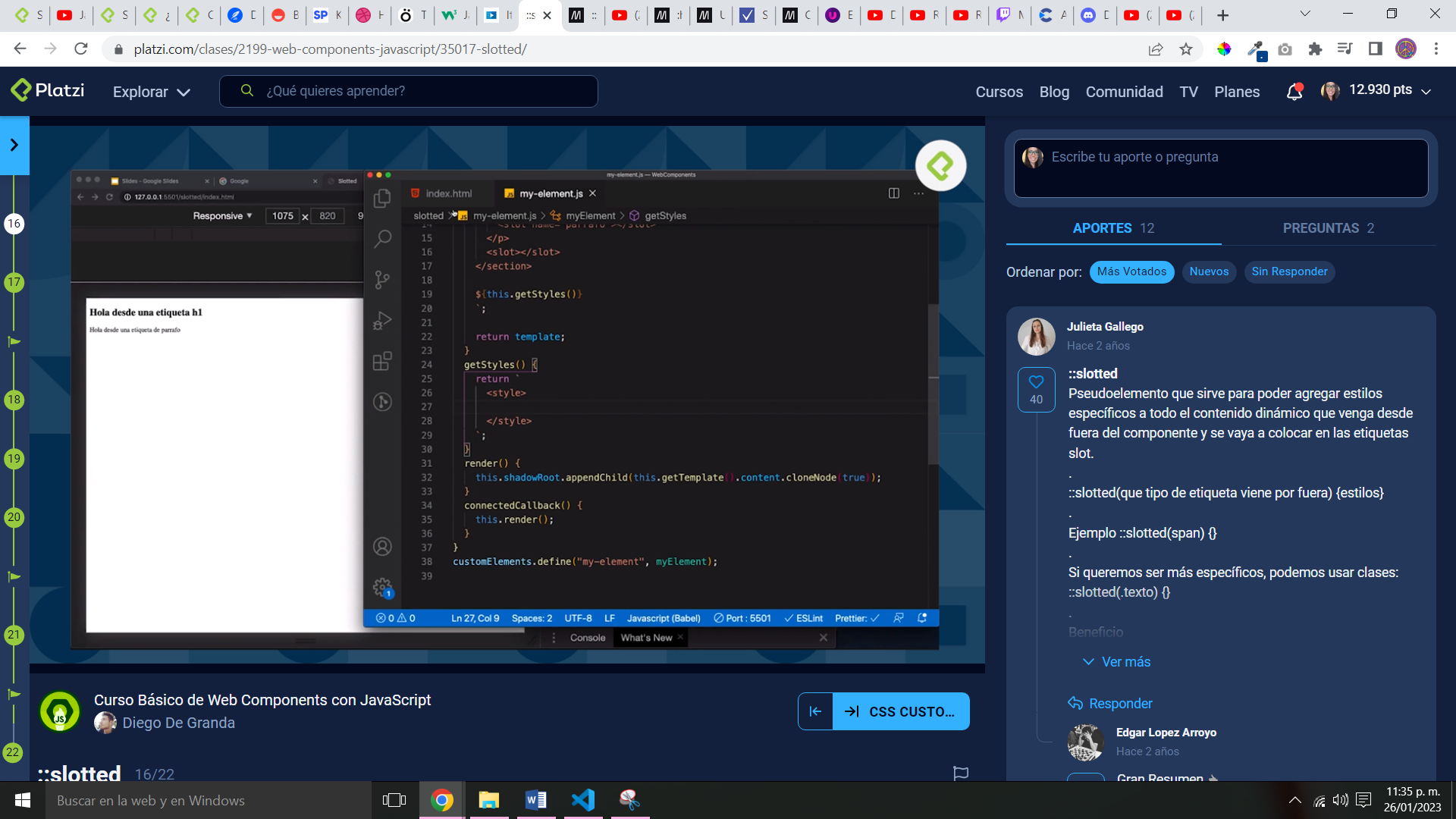
}

.tercero{

--secundary-color: lime;

}

Podremos utilizar como selector el componente y clases que le establezcamos al componente.



Explicación de por qué se nuestros estilos con menor especificidad sobre escriben a los proporcionados en :host: [Explicación detallada](https://github.com/w3c/csswg-drafts/issues/2290#issuecomment-382465643)  
ㅤ  
En resumen los estilos declarados en el :host son considerados como estilos por defecto, como los que ya tiene el navegador que podemos sobre escribir con cualquier tipo de especificidad, pero si el :host del componente tiene algún estilo con !important no podrá ser sobre escrito.  
ㅤ

[CSS custom properties](https://developer.mozilla.org/es/docs/Web/CSS/--*)

Los nombres de las propiedades que tiene el prefijo --, como --ejemplo-nombre, representan las *propiedades personalizadas* que contienen un valor que puede ser usado en otras declaraciones usando la función [var](https://developer.mozilla.org/es/docs/Web/CSS/var).  
ㅤ  
Las propiedades personalizadas tienen como alcance los elementos en los que se declaran y participan en la cascada: el valor de dicha propiedad personalizada es el de la declaración decidida por el algoritmo en cascada.