NOMBRE: Diego Cadena

FECHA: 21/11/2023

NRC: 14385

Actividad de Consulta – Desarrollo Moderno con Web Components

Web Components y Conceptos Clave

1. Definición de Web Components:

- Describe en tus propias palabras qué son los Web Components y por qué

son esenciales en el desarrollo web moderno.

Los Web Components son fragmentos de código frontend diseñados para ser reutilizables en el desarrollo web moderno. Funcionan como piezas de lego, encapsulando código para su uso en diversas partes de una aplicación o incluso en proyectos distintos, sin tener que repetir el código. Son esenciales porque permiten una mayor modularidad y reutilización de código, lo que facilita el mantenimiento y la escalabilidad de las aplicaciones web. Al utilizar Web Components, los desarrolladores pueden crear elementos personalizados con sus propios comportamientos y estilos, contribuyendo a un desarrollo más eficiente y organizado.

2. HTML Templates:

- ¿Cuál es el propósito de la etiqueta <template> en Web Components?

La etiqueta `<template>` se usa con el propósito de definir plantillas de marcado que no se despliegan directamente en la página. Esto contribuye al rendimiento de la página web al evitar el renderizado inmediato del contenido. En lugar de mostrar el contenido de la plantilla, la etiqueta `<template>` facilita la creación de fragmentos del documento que pueden ser clonados y renderizados eficientemente cuando sea necesario.

- ¿Cómo contribuye al rendimiento de la aplicación?

Evita el renderizado inicial: El contenido dentro de la etiqueta <template> no se renderiza de inmediato al cargar la página. Esto significa que el navegador no tiene que procesar ni mostrar ese contenido inicialmente, esto permite acelerar la carga al iniciar la aplicación.

3. Custom Elements:

- Explica la función de la API de Custom Elements.

Su función principal es permitir la definición y el uso de elementos HTML personalizados. Esto significa que los desarrolladores pueden crear sus propias etiquetas HTML con comportamientos específicos, encapsulando la funcionalidad y el estilo de estos elementos.

- ¿Qué reglas de nomenclatura debemos seguir al crear elementos personalizados?

Prefijo único: Es recomendable utilizar un prefijo único para el nombre de la etiqueta personalizada, asegurando así que no entre en conflicto con futuras etiquetas HTML estándar.

Usar guiones en el nombre: El nombre de la etiqueta personalizada debe contener al menos un guion bajo (_) o un guion (-). Esto es una práctica común y ayuda a distinguir los elementos personalizados de las etiquetas HTML convencionales.

4. Shadow DOM:

- ¿Por qué es importante el Shadow DOM en Web Components? Menciona al

menos un beneficio de encapsular estilos y scripts.

El Shadow DOM proporciona encapsulamiento al crear un árbol DOM "sombra" asociado a un elemento. Este encapsulamiento es esencial para evitar conflictos de estilos y scripts en aplicaciones web complejas donde múltiples componentes pueden coexistir.

Uno de los beneficios clave es la capacidad de mantener características del elemento en privado. Al encapsular el árbol DOM de un componente, los estilos y scripts asociados a ese componente no afectarán ni serán afectados por los estilos y scripts de otros elementos en la página.

5. ES Modules:

- ¿Cuál es el papel de ES Modules en Web Components?

facilita el modularidad y la reutilización de código. ES Modules permiten la importación y exportación de funciones, clases u objetos entre archivos JavaScript, cada componente o módulo puede ser desarrollado y mantenido de manera independiente

- ¿Cómo facilita la modularidad y reutilización de código?

Algunas formas en que ES Modules facilita esto son:

Importación y exportación: Los módulos pueden exportar funciones, clases y objetos específicos para ser utilizados en otros módulos. Esto permite una clara definición de la interfaz de un módulo y qué funcionalidades están disponibles para su uso externo.

Independencia de módulos: Cada módulo puede ser desarrollado y mantenido de manera independiente, lo que facilita la gestión del código y permite a los desarrolladores concentrarse en unidades lógicas de funcionalidad.

Ejemplo Práctico: Lista de Tareas

6. Constructor y Connected Callback:

● En el ejemplo de la lista de tareas, ¿cuál es el propósito del constructor y el connectedCallback en el ciclo de vida del componente?

-Constructor: El constructor se utiliza para inicializar el objeto y establecer el Shadow DOM. En este caso, se crea un Shadow DOM con el modo "open" para que sea accesible desde fuera del componente. Además, se clona la plantilla definida en el `<template>`.

- connectedCallback: El método `connectedCallback` se llama automáticamente cuando el elemento personalizado (`task-list`) se conecta al DOM. Aquí, se utiliza para definir la lógica de actualización de la lista (`updateList`). Este método se ejecuta después de que el constructor haya creado el Shadow DOM y establecido la estructura inicial del componente.

7. Utilización de HTML Templates:

● ¿Cómo se utiliza la etiqueta <template> en el ejemplo para mejorar la eficiencia del renderizado?

La etiqueta <template> se utiliza en el ejemplo para mejorar la eficiencia del renderizado al permitir la definición de una plantilla de marcado que no se despliega directamente en la página. En lugar de añadir elementos directamente al DOM durante la creación del componente, se clona la plantilla del <template> y se agrega al Shadow DOM, evitando así el renderizado inmediato de su contenido.

8. Custom Element Definition:

● En el script del ejemplo, ¿Cómo se define el nuevo elemento personalizado "task-list"?

El nuevo elemento personalizado "task-list" se define utilizando la función customElements.define en el script.

9. Shadow DOM en la Lista de Tareas:

● ¿Por qué se utiliza el Shadow DOM en la lista de tareas?

El Shadow DOM se utiliza para encapsular los estilos y el DOM de la lista de tareas. Al crear un Shadow DOM con el modo "open", se establece un ámbito aislado para los estilos y la estructura del componente, evitando que afecten o sean afectados por estilos y scripts externos.

● ¿Qué beneficio proporciona en este contexto?

El beneficio principal es evitar conflictos y mantener la encapsulación. Al encapsular la lista de tareas en el Shadow DOM, se asegura de que los estilos definidos dentro del componente no afecten a otros estilos en la página y viceversa. Esto mejora el modularidad y la reutilización del

componente al garantizar que sus estilos y estructura no sean influenciados por el entorno global de la aplicación.

Actividad Práctica: Galería de Imágenes

10. HTML Template en la Galería de Imágenes:

● En la actividad práctica, ¿cómo se utiliza la etiqueta <template> para definir la estructura de la galería de imágenes?

La etiqueta <template> se utiliza para definir la estructura de la galería de imágenes al proporcionar una plantilla de marcado en el código. En este caso, el contenido de la galería, que incluye estilos y la estructura del contenedor de la galería ( ), se encuentra dentro de la etiqueta <template>.

11. Uso de Shadow DOM en la Galería de Imágenes:

● Explica por qué se aplicó el Shadow DOM en la galería de imágenes.

Se lo aplicó para encapsular los estilos y la estructura del componente. Al utilizar el Shadow DOM con el modo "open", se crea un ámbito aislado para la galería de imágenes, evitando que sus estilos y elementos afecten o sean afectados por estilos y scripts externos. Esto mejora la modularidad y la reutilización del componente al proporcionar una encapsulación clara de su implementación interna.

● ¿Qué problemas podría resolver?

Se podría resolver posibles problemas de colisión de estilos y conflictos en nombres de clases o etiquetas. Al encapsular la galería en el Shadow DOM, se evita que los estilos dentro del componente afecten otros estilos en la página y viceversa

12. ES Modules en la Actividad Práctica:

● ¿En qué situaciones podríamos utilizar ES Modules en la actividad práctica de la galería de imágenes?

Organización del código: Puedes modularizar el código de la galería en diferentes archivos utilizando ES Modules. Por ejemplo, podrías tener un archivo para la lógica del componente, otro para la definición de estilos, y otro para funciones de utilidad.

Reutilización de código: Si tienes funcionalidades comunes que podrían ser compartidas entre diferentes componentes o módulos, puedes exportar esas funcionalidades como módulos para ser importadas donde sea necesario.

13. Importancia del Constructor:

● ¿Cuál es la importancia del constructor en el ciclo de vida del componente?

El constructor en el ciclo de vida del Web Component es esencial para la configuración y preparación inicial del componente antes de que se conecte al DOM. Proporciona un punto de entrada crucial para establecer el estado inicial y llevar a cabo cualquier configuración necesaria antes de que comience su interacción con el entorno del DOM.

- ¿Qué tipo de tareas deberíamos realizar en este paso?

En el constructor, se suelen realizar tareas como la configuración de propiedades iniciales, la creación de eventos, la inicialización de variables, y la configuración del Shadow DOM. Es el lugar adecuado para preparar el componente antes de que sea utilizado, estableciendo un estado inicial y configurando elementos esenciales para su funcionamiento.

14. Connected Callback vs. Disconnected Callback:

- Explica las diferencias entre el connectedCallback y el disconnectedCallback.

connectedCallback: Se llama cuando el componente se conecta al DOM. Aquí es donde se realiza la renderización y la lógica asociada al inicio.

disconnectedCallback: Se llama cuando el componente se desconecta del DOM. En este punto, el componente ya no es parte del documento y puede ser útil para realizar limpieza, liberar recursos y detener cualquier lógica que ya no sea necesaria.

- ¿En qué situaciones podríamos necesitar el disconnectedCallback?

Puede ser útil liberar recursos, como desuscribirse de eventos o anular peticiones de red, cuando el componente ya no está en el DOM. También si el componente tiene animaciones en curso o temporizadores en ejecución, el disconnectedCallback puede ser el lugar para detenerlos para evitar problemas cuando el componente ya no está visible.

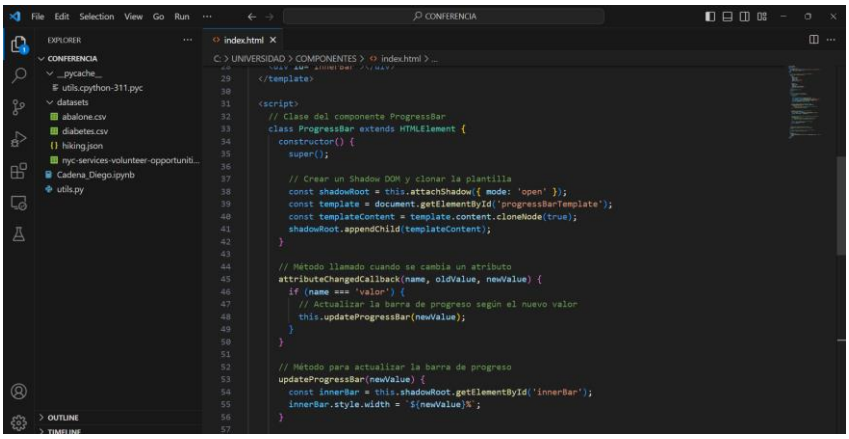
15. Attribute Changed Callback:

- ¿Por qué es útil el attributeChangedCallback?

El attributeChangedCallback es útil porque se llama cuando un atributo del componente cambia, permitiendo que el componente reaccione dinámicamente a esos cambios. Esto es esencial para gestionar dinámicamente el estado y la apariencia del componente en respuesta a cambios en sus atributos.

- Proporciona un ejemplo de cómo podríamos usarlo en un componente.

Se va a desarrollar un ejemplo simple de un componente de barra de progreso (<progress-bar>) que utiliza el attributeChangedCallback para actualizar dinámicamente su apariencia en función del atributo "valor".



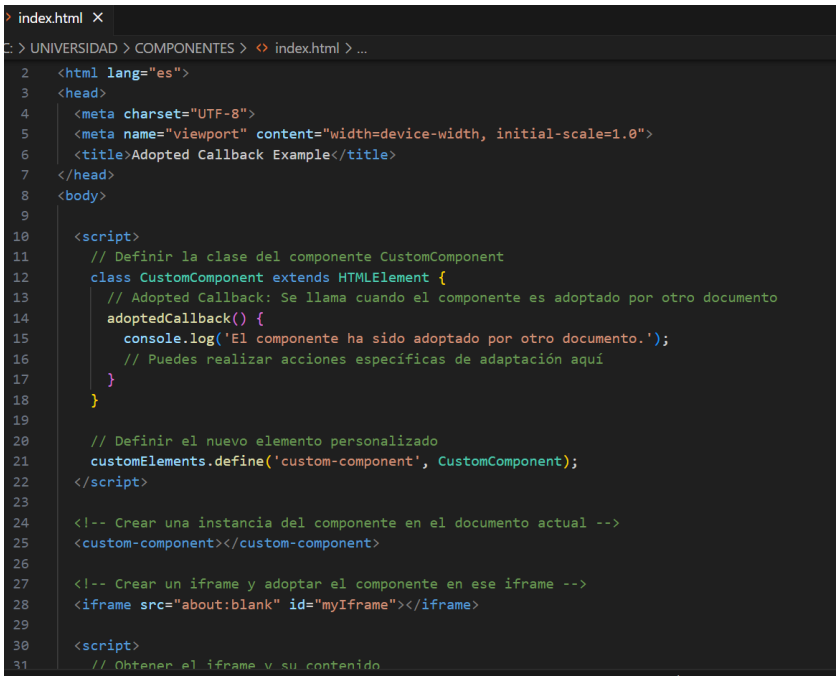
16. Adopted Callback:

● Aunque es menos común,¿Puedes pensar en un escenario en el que sería útil utilizar el adoptedCallback?

Aunque el uso de adoptedCallback es menos común, puede ser útil en escenarios específicos donde los componentes web pueden ser adoptados por otro documento. Un caso de uso potencial podría ser cuando se trabaja con iframes o sombras del DOM.

Se puede tener un conjunto de componentes web que se diseñan para funcionar dentro de un iframe. Estos componentes tienen su propio estado, estilos y lógica interna. Cuando un componente web se "adopta" por otro documento (por ejemplo, se inserta en un iframe diferente), podrías querer realizar ciertas acciones específicas, como restablecer su estado interno o ajustar su apariencia para adaptarse al nuevo entorno.

este ejemplo demuestra cómo el método adoptedCallback se activa cuando un componente es adoptado por otro documento, y puedes realizar acciones específicas de adaptación dentro de ese método.



index.html

C: > UNIVERSIDAD > COMPONENTES > index.html > ...

```
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Adopted Callback Example</title>
</head>
<body>

  <script>
    // Definir la clase del componente CustomComponent
    class CustomComponent extends HTMLElement {
      // Adopted Callback: Se llama cuando el componente es adoptado por otro documento
      adoptedCallback() {
        console.log('El componente ha sido adoptado por otro documento.');
        // Puedes realizar acciones específicas de adaptación aquí
      }
    }

    // Definir el nuevo elemento personalizado
    customElements.define('custom-component', CustomComponent);
  </script>

  <!-- Crear una instancia del componente en el documento actual -->
  <custom-component></custom-component>

  <!-- Crear un iframe y adoptar el componente en ese iframe -->
  <iframe src="about:blank" id="myIframe"></iframe>

  <script>
    // Obtener el iframe y su contenido
```