# WExpDev

En la actualidad, el desarrollo de software no parte de cero; por el contrario, ya se cuenta con un conjunto de herramientas suministradas por frameworks, lo que permite el desarrollo más rápido de aplicaciones, factor relevante e indispensable para apoyar procesos de mejoramiento continuo en busca de mayores niveles de competitividad en esta sociedad globalizada.

En todos los aspectos, la evolución de las aplicaciones web, ya sea en código libre o propietario, está desarrollándose velozmente, porque proporciona niveles de comunicación de servicios, interoperabilidad y acceso a clientes internos y externos que permiten soportar la administración de diferentes procesos de negocio.

WExpDev es un proyecto en JavaScript que va dirigido a implementar un marco de trabajo trasparente, flexible y que puede ser implementado en el front-end de cualquier tipo de proyecto web. la filosofía de este es brindar una estructura básica de cómo hacer desarrollo fron-end sin este estar atado a un esquema complejo de dependencias y a su vez siendo compatible con las metodologías básicas de desarrollo web, así mismo establecer un esquema de desarrollo orientado a componentes desde el api de WebComponents.

WExpDev muestra una metodología del cómo hacer las cosas y brinda una serie de herramientas para simplificar la estructura del código, sin ser estas una camisa de fuerza de implementación.

# Desarrollo web basado en Componentes

Esta se deriva de la ingeniería de software basada en componentes (CBSE), también conocida como desarrollo basado en componentes (CBD), es una rama de la ingeniería de software que enfatiza la separación de asuntos, separation of concerns (SoC), por lo que se refiere a la funcionalidad de amplio rango disponible a través de un sistema de software dado. Es un acercamiento basado en la reutilización para definir, implementar, y componer componente de software débilmente acoplados en sistemas. Esta práctica persigue un amplio grado de beneficios tanto en el corto como el largo plazo, para el software en sí mismo y para las organizaciones que patrocinan tal software.

El desarrollo web basado en componentes busca crear componentes reutilizables que puedan comportarse con ciertas caracteristicas especificas utiles para tareas comunes, podemos ver el caso de una etiqueta <div> o <span> es bastante genérica y es utilizada para contener texto o una mezcla de elementos.

Un elemento <button> o <input> es más específico en funcionalidad o estilo. Cuando colocamos un botón dentro de nuestro HTML, luce como un botón estándar y cuando hacemos click en este, actúa como un botón. Similar a los diferentes estilos de <input>, ya sea que quieras crear un selector de fechas, un deslizador, o un campo de entrada de texto.

Selector de fechas (date picker)

Tomemos el selector de fechas como ejemplo. Para crear una selector de fechas agregamos esta etiqueta en en código HTML: <input type=”date ”> Parece fácil ¿no es así? ¡Si lo es! Pero lo que obtenemos de esta simple etiqueta es bastante complicado, pero todo es manejado para ti por tu buen amigo — el navegador. Esta etiqueta, cuando utilizamos el tipo “fecha” ofrecen un campo de entrada de texto y podemos dar click en el mes, día o año y recorrer hacia adelante o atrás a través de cualquiera de estos. También, si damos click en la flecha que está a un costado, este mostrará una ventana emergente de un calendario que los usuarios podrán usar para interactuar y escoger una fecha. Adicionalmente, en dispositivos móviles, este actúa de manera un poco diferente, no manda ventana pop-up en su lugar mostrará una ventana modal para seleccionar la fecha.

## Componentes

Y podemos hacer más, el selector de fechas tiene propiedades que puedes consultar incluyendo “value”, lo podemos ver rápidamente si mandamos a la salida la propiedad en la consola de JavaScript.

*console.log( document.querySelector(´input´).value);*

Cuando vemos la consola, podremos ver el valor actual del selector de fechas en esta. Este también despacha eventos que podemos escuchar cuando el valor cambia o es enviado. Podemos también llamar métodos en el selector para pasar a través de las fechas. El selector de fechas es un gran ejemplo de componente o modulo reutilizable con un bastante complejo estilo visual y patrones de interactividad que necesitaron ser programados por los fabricantes de los navegadores y los cuales han trabajado y siguen trabajando en una gran variedad de componentes.

El estándar de ECMA Script, entiende la importancia de este estilo de desarrollo y promueve su utilización bajo la creación de los custom elements, también llamados WEBComponents.

Estos son estructuras personalizadas que poseen las mismas características generales que conocemos de nuestras etiquetas HTML y adicional a esto pueden poseer características y lógica definida por el desarrollador, las cuales pueden ir, desde un simple comportamiento estético, hasta un comportamiento complejo, que incluso puede albergar lógica de aplicación.

# Estructura de proyecto

Esta propuesta va dirigida a crear proyectos con una arquitectura modularizada, utilizando el index.html (página de inicio), para desencadenar la llamada a los diferentes partes de la aplicación, haciendo uso de un renderizado dinámico, donde cada una de las partes de la aplicación se construyen a medida que estas son requeridas.

Por ejemplo, un sitio web normal con una simple etiqueta h1 con el texto “Hola Mundo” se vería de la siguiente manera:

<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>HOLA MUNDO</h1>

</body>

</html>

Sin embargo, haciendo uso de renderizado dinámico esta se vería de la siguiente manera.

<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <script>

        window.onload = ()=>{

            const h1 = document.createElement("h1");

            h1.innerText = "HOLA MUNDO";

            App.append(h1);

        }

    </script>

</head>

<body id="App">

</body>

</html>

Partiendo de este ejemplo podemos destacar algunas características interesantes de JavaScript, y del desarrollo orientado a componentes, lo primero es que podemos hacer uso de los nombres propios de los objetos globales (véase el objeto con identificador App), lo segundo esque al momento de nosotros hacer uso de

const h1 = document.createElement("h1");

estamos haciendo uso de la Api de JS para crear elementos en tiempo de ejecución y la constante h1 a su vez es un componente genérico que posee comportamientos específicos y propiedades particulares que al ser modificados tienen un resultado particular (véase el uso de la propiedad innerText).

h1.innerText = "HOLA MUNDO";

crear una pagina web de esta forma podría parecer en principio una sobre complicación con respecto a la versión original usando solo HTML, sin embargo, lo interesante viene cuando esta forma de trabajar es aplicada para crear lógica para estructuras más compleja, que a su vez son repetitivas y mientras más modularizadas estén el código se vuelve limpio.

En el siguiente código se puede visualizar una estructura HTML que posee tres div con la propiedad class=”card”, aunque no es un ejemplo tan practico podemos ir haciendo un análisis mucho mas completo el cual nos puede dar una luz mucho más clara del porque el desarrollo basado en componentes es importante.

<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <div class="card">

        <label class="title">Card 1</label>

        <section>Contain</section>

        <section>Detail</section>

    </div>

    <div class="card">

        <label class="title">Card 2</label>

        <section> Contain </section>

        <section>Detail</section>

    </div>

    <div class="card">

        <label class="title">Card 3</label>

        <section> Contain </section>

        <section>Detail</section>

    </div>

</body>

</html>

Lo primero es que debemos estar enterados que lo más probable es que queramos no solo dibujar tres cards, en algunas ocasiones querremos muchas más, agregarlas de forma manual seria poco práctico puesto que haría que el HTML creciera desmesuradamente con una estructura muy difícil de controlar, por lo que lo mas obvio es hacer una estructura dinámica para poder construir esta card, por lo que podemos plantear la siguiente estructura partiendo de un arreglo de datos que contiene la información de las diferentes cards (existen diversos estilos de cómo lograrlo):

Usando cadenas de texto concatenadas y luego aplicando los cambios al innerHTML: este método se vale del uso de las cadenas multilínea, y aunque en principio usa menos líneas de código, es bastante limitado puesto que no permite hacer uso de la mayoría de las características de los elementos HTML que conforman la card de manera simple (eventos, atributos varios, etc.):

<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <title>Document</title>

    <script>

        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" }

            ]

            let stringCards = "";

            for (let index = 0; index < cards.length; index++) {

                const element = cards[index];

                stringCards = stringCards +

                    `<div class="card">

                        <label class="title">${element.title}</label>

                        <section>${element.Contain}</section>

                        <section>${element.Detail}</section>

                    </div> `;

            }

            App.innerHTML = stringCards;

        }

    </script>

</head>

<body id="App">

</body>

</html>

Por lo que lo mas recomendable es crear los objetos por medio de la función document.createElement(), esta permite crear lo objetos de forma independiente y permite acceder a todas las características del api de manejo de DOM, también es posible crear funciones que simplifiquen la escritura de estos elementos (esto lo veremos más adelante).

<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <script>

        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 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);

                Frag.append(Card);

            }

            App.append(Frag);

        }

    </script>

</head>

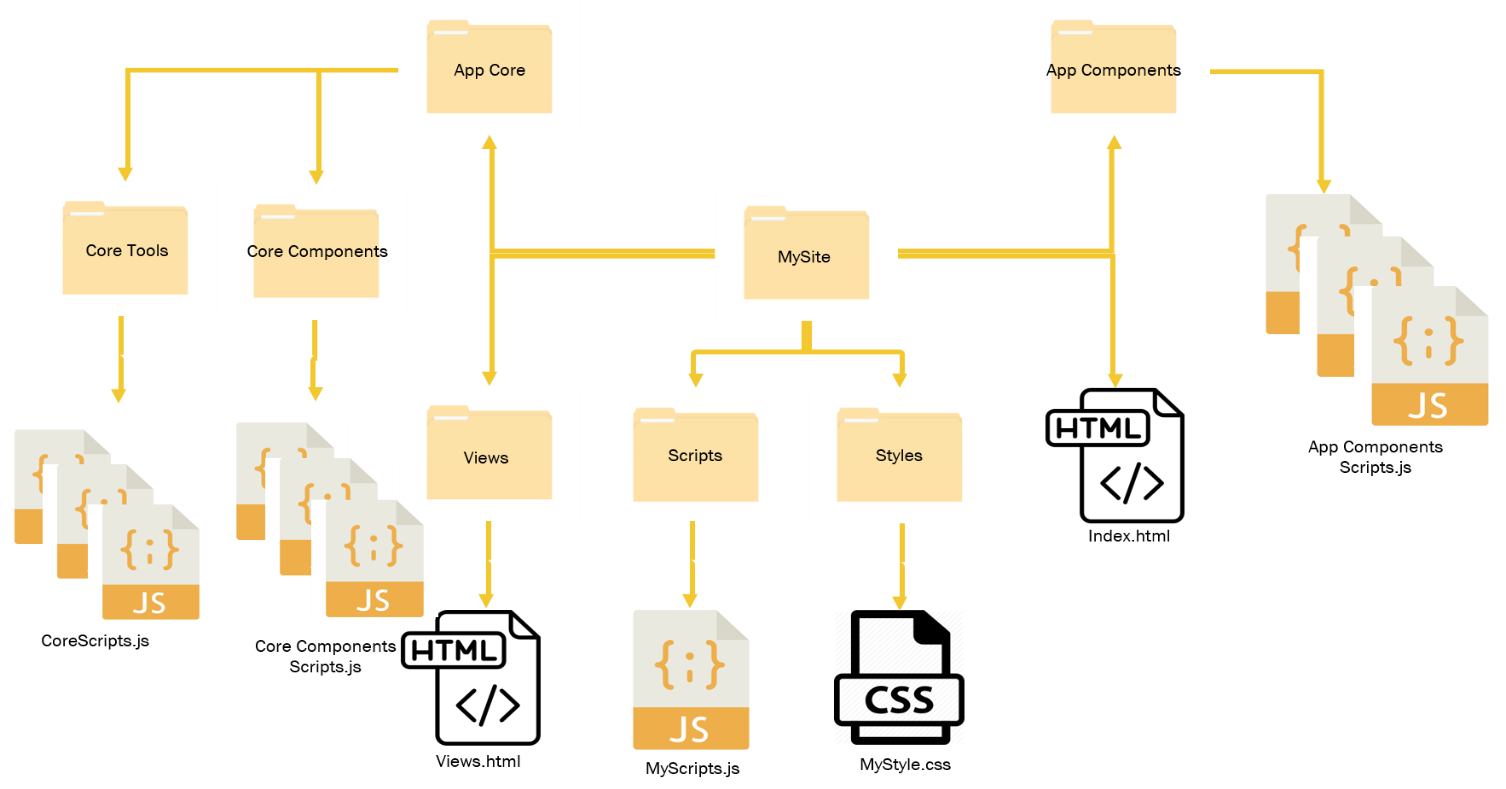
<body id="App">

</body>

</html>

De esta manera podemos tener una construcción de elementos dinámicos dentro de nuestra aplicación y con solo tener una lista de objetos ya sea procesada dentro de nuestra aplicación front-end o traída desde el back-end, garantizamos que nuestras cards se crearan de forma dinámica. Sin embargo, aun hay situaciones que debemos resolver, entre estas la reutilización de este tipo de elementos en otras partes de la aplicación.

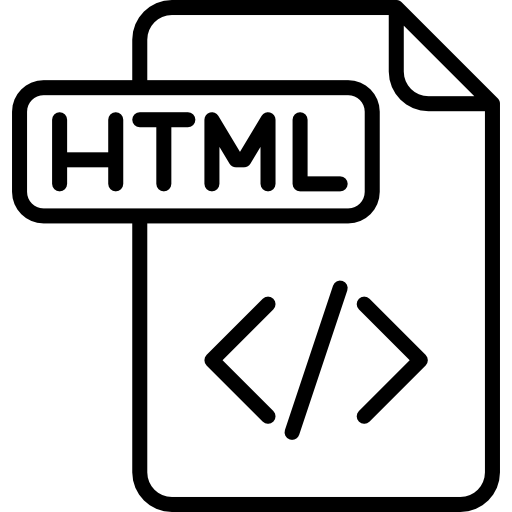
Por ello la correcta estructuración del proyecto y separación de este en módulos se hace sumamente importante. Los proyectos comunes tienen una estructura bastante simple, que a partir de la raíz tienen al menos la separación de scripts, estilos y vistas HTML (cualquier lenguaje que se utilice), para una correcta estructuración de aplicaciones orientado a componentes se hace requerido tener más segmentación de cada una de las partes de la APP, esto con la intención de tener el proyecto organizado de forma eficiente (cabe destacar que esta es solo una propuesta básica de cómo crear la estructura).



En esta propuesta se define que debe existir una separación entre los scripts referentes a partes generales de la App (carpeta Scripts), los scripts que contienen la lógica del marco de trabajo (Carpeta AppCore) y los scripts de los componentes propios de la app, ya sea componentes meramente de estilos o de lógica de UI (Carpeta AppComponets).

Por otro lado, el AppCore tendrá separación entre las herramientas del marco y los componentes reutilizable (CoreComponents).

La diferencia entre AppComponents y CoreComponents radica es que este último contendrá todos aquellos componentes generales reutilizables que trascienden a la lógica del proyecto, por ejemplo, componentes de tablas, cards, calendarios, gráficos, etc. Mientras que los AppComponents contendrá solo componentes propios de la App y que es poco probables sean reutilizables en otros proyectos, ejemplo: diseño de la interfaz, diseño de las vistas de contenido etc.



Scripts

Styles

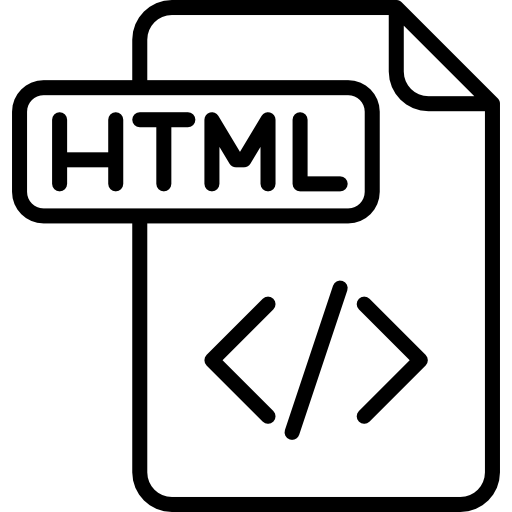
Index.html

MyStyle.css

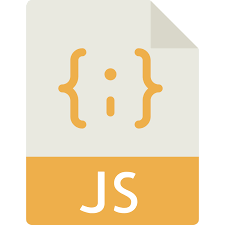
MySite



HTML views



Views.html



MyScripts.js

Ilustración: Estructura de sitio orientado a componentes

Ilustración: estructura básica de sitio web

El CoreTool debería de contener todas aquellas funciones y estructuras lógicas diseñadas para simplificar la escritura de nuestro proyecto, por ejemplo: funciones de renderizado, de peticiones, manejo de modelos, etc.

Volviendo al pequeño ejemplo que teníamos anteriormente lo lógico es que separemos la app en las partes necesarias para una correcta orientación a componentes. La página index.html debería de poseer solo referencias a los scripts y componentes que debe utilizar, y en el caso de poseer código embebido (no recomendado) solo debería ser el necesario para resolver su propia lógica interna.

*Nota: a partir de este punto renombraremos algunas carpetas del esquema dado que esta estructura estará disponible para su uso público, recordemos que, si se quiere hacer una implementación propia de este marco de trabajo, los nombres pueden ser definidos a discreción del desarrollador.*

* *AppCore -> WDevCore*
* *CoreComponents -> WComponents*
* *CoreTools -> WModules*

<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <!-- referencias a estilos, scripts, CoreTools, Componentes etc.  -->

    <script src="./WDevCore/WComponents/CardComponent.js"></script>

    <script src="./index.js"></script>

</head>

<body id="App">

</body>

</html>

El archivo CardComponent.js (ubicado dentro del WDevCore/CoreComponents) el cual se referencia en el index.html es el que tendrá la lógica correspondiente al componente card, en este ejemplo se hace uso de una función, la cual retorna toda la estructura interna de la card y recibe como parámetro el objeto element.

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;

}

El archivo index.js (ubicado en la raíz de proyecto) es el que poseerá la lógica de la vista y realizará todo el procesamiento requerido, para mostrar la estructura de la aplicación.

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];

        Frag.append(CardComponent(element));

    }

    App.append(Frag);

}

De esta forma Podemos visualizar una sencilla aplicación web escrita en Vanilla JS con lógica orientada a componentes. Lo interesante de esto es que ahora el CardComponent está preparado para ser utilizado en cualquier parte de la aplicación y al estar dentro de la estructura del WDevCore es la primera piedra que tenemos para construir lo que sería nuestro propio marco de trabajo.

# 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.

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);

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

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);

}

# Función de renderizado estructurado

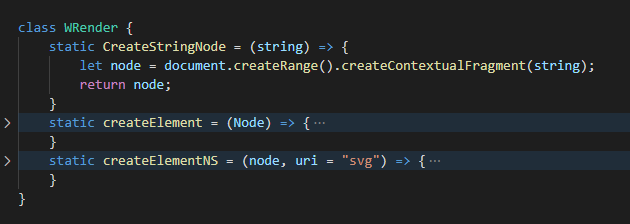
En este punto debemos encontrar la manera de simplificar la escritura de nuestros componentes dado que el uso de atributos y asignación de eventos a los diferentes elementos del DOM puede ser una tarea tediosa, por lo que sería interesante crear una estructura lógica que nos permita escribir el código 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.

## Definición del modulo

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.



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

## Función createElement

Existen muchas formas con las cuales se puede implementar esta función, en este caso se creará una función estática llamada createElement que recibe un objeto llamado Node y se realizaron validaciones desde diversas perspectivas, lo primero a tomar en cuenta es que una buena práctica de desarrollo siempre invita a usar manejo y control de errores usando bloques try catch.

class WRender {

    static createElement = (Node) => {

        try {

            if (typeof Node === "undefined" || Node == null) {

                return document.createTextNode("Nodo nulo o indefinido.");

            } else if (typeof Node === "string" || typeof Node === "number") {

                return document.createTextNode(Node);

            } else if (Node.\_\_proto\_\_ === HTMLElement.prototype

|| Node.\_\_proto\_\_.\_\_proto\_\_ === HTMLElement.prototype) {

                return Node;

            } else {

                const element = document.createElement(Node.type);

                if (Node.props != undefined && Node.props.\_\_proto\_\_ == Object.prototype) {

                    for (const prop in Node.props) {

                        if (prop == "class") element.className = Node.props[prop];

                        else element[prop] = Node.props[prop];

                    }

                }

                if (Node.children != undefined && Node.children.\_\_proto\_\_ == Array.prototype) {

                    Node.children.forEach(Child => {

                        element.appendChild(this.createElement(Child));

                    });

                }

                return element;

            }

        } catch (error) {

            console.log(error, Node);

            return document.createTextNode("Problemas en la construcción del nodo.");

        }

    }

}

Dentro del bloque try se harán diversas verificaciones, entre estas la primera es determinar si el nodo es nulo o indefinido y en caso de serlo retornar un mensaje, esto es sumamente importante debido a que es probable que el nodo se defina por medio de funciones o llamados a variables dinámica.

if (typeof Node === "undefined" || Node == null) {

                return document.createTextNode("Nodo nulo o indefinido.");

            }

La segunda verificación va dirigida a determinar si el nodo es simplemente una cadena de texto o un número y retornar un textNode con su valor.

else if (typeof Node === "string" || typeof Node === "number") {

                return document.createTextNode(Node);

            }

La tercera verificación tiene como objetivo identificar si el nodo enviado ya es un nodo HTML previamente construido, esto podría parecer innecesario dado que se supone la función está diseñada para construir este tipo de nodos, sin embargo, es probable que a un nodo al que se le han definido diversos hijos, alguno de estos sea un HTMLElement.

else if (Node.\_\_proto\_\_ === HTMLElement.prototype) {

                return Node;

       }

Por último, el bloque else es el que se encargara de crear el árbol de nodos con la estructura tal cual se haya definido.

Lo primero es crear el elemento del tipo especificado por la propiedad type del objeto Nodo.

const element = document.createElement(Node.type);

dado que los HTMLElement pueden tener diversos atributos estos los englobaremos dentro de una propiedad llamada props, en las cuales podremos hacer uso de atributos como son el: id, className, atributos propios en el caso de que estos sean WebComponents, funciones ligadas a eventos tales como el onclick, onchange etc, y funciones propias del componentes customizadas y controladas por el desarrollador (esto lo veremos más adelante).

Por temas de validación deberemos verificar que la propiedad props exista y que a su vez si prototipo sea realmente el de un objeto de js. Una vez verificado solo se procederá a recorrer cada una de las props definidas y asignárselas según su nombre al HTMLElement. La única condición a realizar en esta tarea es verificar que en el caso de usar la propiedad “class” esta sea remplazada por “className” debido a que esta es una palabra reservada del lenguaje.

if (Node.props != undefined && Node.props.\_\_proto\_\_ == Object.prototype) {

        for (const prop in Node.props) {

             if (prop == "class") element.className = Node.props[prop];

             else element[prop] = Node.props[prop];

        }

}

El siguiente paso está ligado a los hijos del nodo, dado que este puede tener diversos nodos hijos, haremos uso de un arreglo almacenado dentro de la propiedad llamada “children”. Siguiendo la misma línea de validación y verificación debemos de determinar si este objeto existe y a su vez si su prototipo corresponde a un Array. Por otro lado, una vez que estas condiciones se cumplan recorreremos el arreglo con un forEach y por objeto haremos una llamada recursiva de esta misma función y ejecutaremos un appendChild del elemento retornado.

if (Node.children != undefined && Node.children.\_\_proto\_\_ == Array.prototype) {

      Node.children.forEach(Child => {

          element.appendChild(this.createElement(Child));

      });

}

Por último, una vez que el nodo esté preparado con todas sus propiedades e hijos ya renderizados, simplemente retornaremos el HTMLElement creado.

return element;

ahora que ya tenemos nuestra función creada y correctamente validada, procederemos a ponerla en práctica, lo primero será referenciarla nuestro head, del index.html.

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

Lo siguiente será reestructurar todo el contenido de nuestro WebComponent de muestra, haciendo uso de nuestra función de renderizado en lugar de la función básica de js.

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);

class WCardComponent extends HTMLElement {

    constructor() {

        super();

        this.attachShadow({ mode: "open" });

    }

    connectedCallback() {

        this.shadowRoot.append(this.DrawCard());

    }

    DrawCard() {

        return WRender.createElement({

            type: "label", props: { className: "card" },

            children: [

                { type: "label", props: { className: "title", innerText: this.element.title }},

                { type: "section", props: { innerText: this.element.Contain } },

                { type: "section", props: { innerText: this.element.Detail } },

            ]

        });

    }

}

customElements.define("w-card", WCardComponent);

En este ejemplo podemos notar como logramos reducir unas cuantas líneas de código al utilizar una estructura de objeto js en lugar de las clásicas declaraciones de document.createElement(), también se debe tomar en cuenta que a medida que los elementos son más complejos esta metodología compactara y estructurara el comportamiento de estos de mejor manera, así mismo podemos encapsular muchas propiedades en estructuras de datos más complejas y estas podrían ser procesadas por otras funciones del frontend, traídas desde el backend o incluso podrían ser retornadas por algún API y luego leída y renderizada por la App sin problemas.

*Nota: se debe entender que la razón por la que podemos hacer uso de WRender y el WCardComponent en cualquier parte del código es porque, estamos haciendo referencia a los scripts directamente y según el orden de carga de los scripts, ninguna tarea realmente se esta ejecutando hasta que el documento este completamente cargado, gracias al evento onload. Esto tiene algunos inconvenientes dado que si la aplicación es demasiado grande y utiliza demasiados componentes estos se descargarán en nuestro cliente antes de comenzar cualquier tarea de la App y por obvias razones tendremos menos eficiencia en el rendimiento, por ello lo mejor es siempre usar ES6 modules los cuales permiten hacer carga de cada componente de forma asíncrona solo al momento que vaya a ser utilizado.*

En el caso del index.js haríamos también una pequeña modificación que también nos ahorrara un par de líneas de código.

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);

}

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];

        Frag.append(WRender.createElement({type: "w-card", props: {element: element}}));

    }

    App.append(Frag);

}

## Función createElementNS

Existe un tipo de elementos muy particular que contienen espacios de nombres URI, para crearlos normalmente se utiliza document.createElementNS, y dentro de sus parámetros se define el URI correspondiente.

Los nameSpace validos son los siguientes:

* HTML - Usa http://www.w3.org/1999/xhtml
* SVG - Usa http://www.w3.org/2000/svg
* XBL - Usa http://www.mozilla.org/xbl
* XUL - Usa http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul

Por lo que al momento crear un elemento con JS se debe declarar de la siguiente forma:

***Let newdiv = document.createElementNS("http://www.w3.org/1999/xhtml","div")***

Implementando una función dentro de nuestro WRender quedaría de la siguiente forma.

static createElementNS = (node, uri = "svg") => {

        try {

            let URI = null;

            switch (uri) {

                case "svg":

                    URI = "http:\/\/www.w3.org/2000/svg";

                    break;

                case "html":

                    URI = "http://www.w3.org/1999/xhtml";

                    break;

                case "xbl":

                    URI = "http://www.mozilla.org/xbl";

                    break;

                case "xul":

                    URI = "http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul";

                    break;

                default:

                    URI = null;

                    break;

            }

            const element = document.createElementNS(URI, node.type)

            if (node.props) {

                for (const prop in node.props) {

                    if (typeof node.props[prop] === "function") {

                        element[prop] = node.props[prop];

                    } else if (typeof node.props[prop] === 'object') {

                        element[prop] = node.props[prop];

                    } else {

                        try {

                            element.setAttributeNS(null, prop, node.props[prop])

                        } catch (error) {

                            element.setAttributeNS(URI, prop, node.props[prop]);

                        }

                    }

                }

            }

            if (node.children) {

                node.children

                    .map(this.createElementNS)

                    .forEach(child => element.appendChild(child, uri))

            }

            return element;

        } catch (error) {

        }

    }

Por lo que la utilización de esta función seria de la siguiente forma:

var SvgElement = WRender.createElementNS({

            type: "svg",

            props: {

                viewBox: "0 0 120 120",

            }

        });

Dado que por defecto la función entiende que el espacio de nombre sera “svg”, no es requerido especificarlo. Sin embargo, si se requiriera especificar el uri simplemente se haría de la siguiente forma.

var Chart = WRender.createElementNS({

            type: "svg",

            props: {

                viewBox: "0 0 120 120",

            }

        }, "svg");

## Funcion CreateStringNode

En algunas ocasiones será necesario crear nodos a partir de cadenas de texto, por lo que se preparo una función destinada este propósito con la intención de acortar la sintaxis original de vanilla JS

static CreateStringNode = (string) => {

        let node = document.createRange().createContextualFragment(string);

        return node.childNodes[0];

    }

Su uso es bien simple, solo se debe de llamar la función y pasar la estructura del nodo dentro de una cadena de texto.

WRender.CreateStringNode("<h1>My App</h1>");

# ES6 Modules

Los programas JavaScript comenzaron siendo bastante pequeños, la mayor parte de su uso en los primeros días era para realizar tareas de scripting aisladas, proporcionando un poco de interactividad a tus páginas web donde fuera necesario, por lo que generalmente no se necesitaban grandes scripts. Pasado los años los cambios son bastantes significativos, ahora tenemos aplicaciones completas que se ejecutan en navegadores con mucho JavaScript.

Por lo tanto, en los últimos años se ha comenzado a pensar en proporcionar mecanismos para dividir programas JavaScript en módulos separados que se puedan importar cuando sea necesario. Node.js ha tenido esta capacidad durante mucho tiempo, y hay una serie de bibliotecas y marcos de JavaScript que permiten el uso de módulos (por ejemplo, CommonJS y AMD otros basados en sistemas de módulos como RequireJS, y recientemente Webpack y Babel) y el día de hoy gracias a las mejoras constantes del lenguaje, los módulos son soportados de forma nativa y podemos utilizarlo sin ninguna dificultad.

Al momento de hacer uso de ES6 modules debemos tomar en cuanta algunos detalles:

* Debes prestar atención a las pruebas locales — si intentas cargar el archivo HTML localmente (es decir, con una URL file:///), te encontrarás con errores de CORS debido a los requisitos de seguridad del módulo JavaScript. Necesitas hacer tus pruebas a través de un servidor.
* Además, ten en cuenta que puedes obtener un comportamiento diferente de las secciones del script definidas dentro de los módulos en comparación con los scripts estándar. Esto se debe a que los módulos automáticamente usan [strict mode](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Strict_mode).
* No es necesario utilizar el atributo defer (ve [atributos de <script>](https://developer.mozilla.org/es/docs/Web/HTML/Element/script#Attributes)) al cargar un script de módulo; los módulos se difieren automáticamente.
* Los módulos solo se ejecutan una vez, incluso si se les ha hecho referencia en varias etiquetas <script>.
* Por último, pero no menos importante, dejemos esto en claro las características del módulo se importan al alcance de un solo script no están disponibles en el alcance global. Por lo tanto, solo podrás acceder a las funciones importadas en el script en el que se importan y no podrás acceder a ellas desde la consola de JavaScript, por ejemplo. Seguirás recibiendo errores de sintaxis en DevTools, pero no podrás utilizar algunas de las técnicas de depuración que esperabas utilizar.

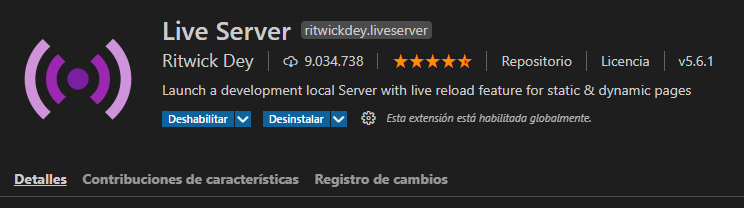
El último punto es algo de suma importancia dado que cualquier información que procesemos desde un módulo, solo es accesible desde ese modulo y no puede ser modificada por otros módulos o submódulos, amenos que estos estén siendo pasados como parámetros y/o gestionados por algún manejador de datos global. Ejemplo: el manejador de estados de REACT.

Con respecto al punto uno solo tendremos en cuenta que debemos ejecutar nuestra app desde un servidor ya sea local, externo o un servidor virtual.

Ejemplos de servidores que se pueden utilizar no requieren mucha configuración:

XAMPSERVER: este web server que incluye apache dentro de sus características, es bien fácil de utilizar, solo se debe instalar y colocar dentro de la carpeta htdocs la carpeta de nuestra app, luego acceder desde el “localhost/nombre\_de\_carpeta”, asumiendo que tienes problemas de puerto, solo se debe configurar el xamp para usar un puerto determinado distinto al que está definido por defecto.

Live Server: esta es una extensión de VSCode permite correr aplicaciones en un server virtual fácil de utilizar.



Tome en cuenta que estas son solo un par de recomendaciones para hacer pruebas de su app, dado que hay muchos más server y extensiones que podrían fácilmente dar el mismo resultado, adicional a esto la mayoría de los IDE de desarrollo poseen su propio server virtual de pruebas. Y una vez que la app será puesta en producción los servers por defecto soportan ES6 modules.

## IMPORT/EXPORT

Los ES6 modules traen consigo la potencia de hacer uso de la importación y exportación de librerías/bloques de código JS, permitiendo que las aplicaciones no tengan que cargar todos los elementos a la vez, así mismo la exportación puede ser tanto dinámica como estática, lo que implica que puedo declarar la traída de cada script desde el momento de la definición de la aplicación, o cargarlo en tiempo de ejecución por medio de acciones y eventos.

La declaración **export** se utiliza al crear módulos de JavaScript para exportar funciones, objetos o tipos de dato primitivos del módulo para que puedan ser utilizados por otros programas con la sentencia [import](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Statements/import). Los módulos exportados están en [strict mode](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Strict_mode) tanto si se declaran, así como si no. La sentencia export no puede ser utilizada en scripts embebidos.

**Exportación por defecto:** normalmente es utilizada cuando un módulo está diseñado solo para exportar un único elemento ya sea objeto, función o datos.

export default function myFun() {

     //code

}

Dado que myFun es una exportación por defecto, este puede ser invocada con cualquier nombre de instancia desde cualquier parte del código.

import mF from './ubicacion/myFuncModule.js'

Exportación de elementos específicos: esta se utiliza cuando un modulo posee múltiples objetos exportables.

function myFun() {}

class myClass {}

let myVar = "value";

export {myFun, myClass, myVar}

al hacer uso de este tipo de exportación, al momento de importar estos elementos se debe hacer uso del nombre de este para especificar cual es el que será utilizado.

import {myClass} from './myModule.js'

import {myFun} from './myModule.js'

También es posible hacer múltiples importaciones en la misma línea.

import {myClass, myVar} from './myModule.js'

o en el caso de querer exportar todos los elementos usar (este tipo de exportación no se puede usar para importar elemento con exportación por defecto):

import \* from './myModule.js'

Para realizar exportaciones dinámicas, se debe hacer uso de promesas, por medio de la sintaxis async/await. Esta es una de las incorporaciones más recientes al lenguaje JavaScript, son las funciones asincrónicas, que permiten hacer uso de la palabra clave await, la cual permite indicarle a la función que debe esperar a que se resuelva una promesa, antes de continuar ejecutando el código.

La sintaxis sería bastante simple, se declararía la función que se encargaría de realizar la importación, esta sería una función del tipo asíncrona y la importación se realizaría por medio de la declaración de una variable, que este definida por la importación del módulo utilizando la palabra clave await:

const myFun = async() => {

    const { myClass } = await import ('./myModule.js');

    //continuar con el código

}

Es posible que nuestro modulo tenga solamente la definición de nuestro componente y no tenga en ningún momento definido ninguna exportación, por lo que no es posible declarar el nombre de ningún objeto a importar, por lo que simplemente queremos que en cuanto el modulo se ponga en marcha defina algún customElement o simplemente ejecute ciertas tareas, para lograr esto simplemente debemos definir la importación.

De forma estática de la siguiente forma:

import "./myModule.js";

y de forma dinámica dentro de una función de la siguiente forma:

const myFun = async() => {

    await import('./myModule.js');

    //continuar con el código

}

Habiendo explicado todo esto, es momento de poner en práctica esta lógica dentro de nuestro entorno de trabajo. Para ello partiremos de modificar nuestro index.html, dado que necesitamos quitar las referencias a los módulos y de paso convertir el index.js en uno.

<!DOCTYPE html>

<html lang="es">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <script src="./index.js" type="module"></script>

    <!-- referencias a estilos, scripts, coreTools, Components etc.  -->

</head>

<body id="App">

</body>

</html>

Como podemos notar hemos retirado las referencias a CardComponent y a WComponentsTools, por otro lado dentro del index js, para hacer uso de estos elementos haremos importación de estos objetos, por lo que el WComponentsTools debería poder exportar el WRender, y en el caso de CardComponent no será necesario hacer ningún tipo de exportación dado que solamente haremos uso de la definición de customElement.

Por lo que el WComponentsTools quedaría de la siguiente manera:

class WRender {

    static createElement = (Node) => {

        try {

            if (typeof Node === "undefined" || Node == null) {

                return document.createTextNode("Nodo nulo o indefinido.");

            } else if (typeof Node === "string" || typeof Node === "number") {

                return document.createTextNode(Node);

            } else if (Node.\_\_proto\_\_.\_\_proto\_\_ === HTMLElement.prototype) {

                return Node;

            } else {

                const element = document.createElement(Node.type);

                if (Node.props != undefined && Node.props.\_\_proto\_\_ == Object.prototype) {

                    for (const prop in Node.props) {

                        if (prop == "class") element.className = Node.props[prop];

                        else element[prop] = Node.props[prop];

                    }

                }

                if (Node.children != undefined && Node.children.\_\_proto\_\_ == Array.prototype) {

                    Node.children.forEach(Child => {

                        element.appendChild(this.createElement(Child));

                    });

                }

                return element;

            }

        } catch (error) {

            console.log(error, Node);

            return document.createTextNode("Problemas en la construcción del nodo.");

        }

    }

}

export { WRender }

El CardComponent necesitaría hacer uso de la importación del WRender para poder ejecutar sus métodos, por lo que incluiríamos en la cabecera de este la importación de esta clase import {WRender} from "../WModules/WComponentsTools.js"

import { WRender } from "../WModules/WComponentsTools.js";

class WCardComponent extends HTMLElement {

    constructor() {

        super();

        this.attachShadow({ mode: "open" });

    }

    connectedCallback() {

        this.shadowRoot.append(this.DrawCard());

    }

    DrawCard() {

        return WRender.createElement({

            type: "label",

            props: { className: "card" },

            children: [

                { type: "label", props: { className: "title", innerText: this.element.title } },

                { type: "section", props: { innerText: this.element.Contain } },

                { type: "section", props: { innerText: this.element.Detail } },

            ]

        });

    }

}

customElements.define("w-card", WCardComponent);

el index.js también necesitaría importar el WRender y la definición de nuestro componente, al utilizar importación estática quedaría de la siguiente forma.

import { WRender } from "./WDevCore/WModules/WComponentsTools.js";

import "./WDevCore/WComponents/CardComponent.js"

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];

        Frag.append(WRender.createElement({ type: "w-card", props: { element: element } }));

    }

    App.append(Frag);

}

# CSS IN JS

CSS-in-JS es una técnica de estilo en la que se usa JavaScript para diseñar componentes. Cuando se analiza el código JavaScript, se genera CSS (generalmente como un <style>elemento) y se adjunta al DOM. Permitiendo abstraer CSS al nivel del componente en sí, utilizando JavaScript para describir estilos de forma declarativa y mantenible.

Hay múltiples implementaciones de este concepto en forma de bibliotecas, la mayoría diseñadas para react, sin embargo, ya hay varios frameworks populares como vue y stvelt también están implementando esta línea de trabajo.

## Beneficios

* Pensando en componentes. Ya no tienes que mantener un montón de hojas de estilo. CSS-in-JS abstrae el modelo CSS al nivel del componente, en lugar del nivel del documento (modularidad).
* CSS-in-JS aprovecha todo el poder del ecosistema JavaScript para mejorar CSS.
* Aislamiento de las reglas verdaderas. Los selectores con alcance no son suficientes. CSS tiene propiedades que se heredan automáticamente del elemento principal, si no se definen explícitamente
* Selectores con alcance. CSS tiene un solo espacio de nombres global. Es imposible evitar colisiones de selectores en aplicaciones no triviales. CSS en JavaScript genera nombres de clase únicos de forma predeterminada, cuando se compila en CSS.
* Prefijo de proveedor. Las reglas de CSS tienen automáticamente el prefijo del proveedor, por lo que no tiene que pensar en ello.
* Compartir código. Comparta fácilmente constantes y funciones entre JavaScript y CSS.
* Solo los estilos que están actualmente en uso en su pantalla están en el DOM.
* Eliminación de código muerto.
* Manejo de estilos totalmente dinámicos, definidos por arreglos globales, atributos propios de las clases o modelos de datos.

## Implementación

Esta propuesta también incluye una implementación propia de esta librería y busca tener un estilo minimalista de este por medio de la implementación de la POO.

Esta propuesta parte de la utilización de la estructura que hasta el momento estamos construyendo, por lo que iniciaremos por crear dentro de WDevCore/Wmodules un archivo llamado WStyledRender.js.

Dentro de esta haremos uso del WRender, por lo que procederemos a importarlo.

import { WRender } from "./WComponentsTools.js";

posteriormente será necesario tener alguna especie de intelliSense, dado que no tendremos la ayuda por defecto que aportan los editores/IDES en el manejo de archivos CSS, se hará complicado gestionar los nombres de propiedades existentes en CSS, por ello crearemos una clase que incluya esta lista de nombres y la usaremos como guía para implementar nuestros propios estilos.

Esta clase tendrá dentro de sus atributos la lista completa de propiedades CSS, con un valor por defecto en NULL(es posible que haya propiedades faltantes no incluidas en la lista).

class CSSProps {

    "align-content" = null;

    "align-items" = null;

    "align-self" = null;

    "all" = null;

    "animation" = null;

    "animation-delay" = null;

    "animation-direction" = null;

    "animation-duration" = null;

    "animation-fill-mode" = null;

    "animation-iteration-count" = null;

    "animation-name" = null;

    "animation-play-state" = null;

    "animation-timing-function" = null;

    "caption-side" = null;

    "caret-color" = null;

    //"@charset" = null;

    "clear" = null;

    "clip" = null;

    "color" = null;

    "column-count" = null;

    "column-fill" = null;

    "column-gap" = null;

    "column-rule" = null;

    "column-rule-color" = null;

    "column-rule-style" = null;

    "column-rule-width" = null;

    "column-span" = null;

    "column-width" = null;

    "columns" = null;

    "content" = null;

    "counter-increment" = null;

    "counter-reset" = null;

    "cursor" = null;

    "direction" = null;

    "display" = null;

    "empty-cells" = null;

    "filter" = null;

    "flex" = null;

    "flex-basis" = null;

    "flex-direction" = null;

    "flex-flow" = null;

    "flex-grow" = null;

    "flex-shrink" = null;

    "flex-wrap" = null;

    "float" = null;

    "font" = null;

    "font-family" = null;

    "font-feature-settings" = null;

    "font-kerning" = null;

    "font-language-override" = null;

    "font-size" = null;

    "font-size-adjust" = null;

    "font-stretch" = null;

    "font-style" = null;

    "font-synthesis" = null;

    "font-variant" = null;

    "font-variant-alternates" = null;

    "font-variant-caps" = null;

    "font-variant-east-asian" = null;

    "font-variant-ligatures" = null;

    "font-variant-numeric" = null;

    "font-variant-position" = null;

    "font-weight" = null;

    "grid" = null;

    "grid-area" = null;

    "grid-auto-columns" = null;

    "grid-auto-flow" = null;

    "grid-auto-rows" = null;

    "grid-column" = null;

    "grid-column-end" = null;

    "grid-column-gap" = null;

    "grid-column-start" = null;

    "grid-gap" = null;

    "grid-row" = null;

    "grid-row-end" = null;

    "grid-row-gap" = null;

    "grid-row-start" = null;

    "grid-template" = null;

    "grid-template-areas" = null;

    "grid-template-columns" = null;

    "grid-template-rows" = null;

    "hanging-punctuation" = null;

    "height" = null;

    "hyphens" = null;

    "image-rendering" = null;

    "isolation" = null;

    "justify-content" = null;

    "left" = null;

    "letter-spacing" = null;

    "line-break" = null;

    "line-height" = null;

    "list-style" = null;

    "list-style-image" = null;

    "list-style-position" = null;

    "list-style-type" = null;

    "margin" = null;

    "margin-bottom" = null;

    "margin-left" = null;

    "margin-right" = null;

    "margin-top" = null;

    "max-height" = null;

    "max-width" = null;

    "min-height" = null;

    "min-width" = null;

    "mix-blend-mode" = null;

    "object-fit" = null;

    "object-position" = null;

    "opacity" = null;

    "order" = null;

    "orphans" = null;

    "outline" = null;

    "outline-color" = null;

    "outline-offset" = null;

    "outline-style" = null;

    "outline-width" = null;

    "overflow" = null;

    "overflow-wrap" = null;

    "overflow-x" = null;

    "overflow-y" = null;

    "padding" = null;

    "padding-bottom" = null;

    "padding-left" = null;

    "padding-right" = null;

    "padding-top" = null;

    "page-break-after" = null;

    "page-break-before" = null;

    "page-break-inside" = null;

    "perspective" = null;

    "perspective-origin" = null;

    "pointer-events" = null;

    "position" = null;

    "quotes" = null;

    "resize" = null;

    "right" = null;

    "scroll-behavior" = null;

    "tab-size" = null;

    "table-layout" = null;

    "text-align" = null;

    "text-align-last" = null;

    "text-combine-upright" = null;

    "text-decoration" = null;

    "text-decoration-color" = null;

    "text-decoration-line" = null;

    "text-decoration-style" = null;

    "text-indent" = null;

    "text-justify" = null;

    "text-orientation" = null;

    "text-overflow" = null;

    "text-shadow" = null;

    "text-transform" = null;

    "text-underline-position" = null;

    "top" = null;

    "transform" = null;

    "transform-origin" = null;

    "transform-style" = null;

    "transition" = null;

    "transition-delay" = null;

    "transition-duration" = null;

    "transition-property" = null;

    "transition-timing-function" = null;

    "unicode-bidi" = null;

    "user-select" = null;

    "vertical-align" = null;

    "visibility" = null;

    "white-space" = null;

    "widows" = null;

    "width" = null;

    "word-break" = null;

    "word-spacing" = null;

    "word-wrap" = null;

    "writing-mode" = null;

    "z-index" = null;

    "background" = null;

    "background-color" = null;

    "background-image" = null;

    "border" = null;

    "border-radius" = null;

    "border-top" = null;

    "border-right" = null;

    "border-left" = null;

    "border-bottom" = null;

    "box-shadow" = "";

}

Posteriormente procederemos a crear una clase bastante sencilla que servirá como modelo de estructura de selector CSS. El constructor de esta clase solamente recibirá como parámetro el nombre del clase o selector, esto deberá ser una cadena de texto la cual puede incluir cualquier forma de escritura de selector CSS (ejemp: “#componetId” / “.ClassName” / “tagName” ), así mismo recibirá un segundo parámetro que incluirá la lista de propiedades de ese selector y este tendrá un valor por defecto definido por un objeto del tipo CSSProps.

class WCssClass {

    constructor(ClassName, PropsList = (new CSSProps())) {

        this.Name = ClassName;

        this.CSSProps = PropsList;

    }

}

Una vez construidas las estructuras necesarias, crearemos un WebComponent que defina todo el comportamiento del bloque de estilos, que pueda distinguir entre estilos normales, mediaquerys y keyframes. Para este caso no utilizaremos shadowRoot dado que la intención de este componente es que afecte a elementos externos a este ya sea elementos globales del DOM o componentes que lo incluyan.

class WStyledRender extends HTMLElement {

    constructor() {

        super();

    }

    attributeChangedCallBack() {

        this.DrawStyle();

    }

    connectedCallback() {

        if (this.innerHTML != "") {

            return;

        }

        this.DrawStyle();

    }

    DrawStyle() {

        let styleFrag = {

            type: "style",

            props: {},

            children: []

        }

        if (this.ClassList != undefined && this.ClassList.\_\_proto\_\_ == Array.prototype) {

            styleFrag.children.push(this.DrawClassList(this.ClassList));

        }

        if (this.MediaQuery != undefined && this.MediaQuery.\_\_proto\_\_ == Array.prototype) {

            this.MediaQuery.forEach(MediaQ => {

                let MediaQuery = `@media ${MediaQ.condicion}{

                    ${this.DrawClassList(MediaQ.ClassList)}

                }`;

                styleFrag.children.push(MediaQuery);

            });

        }

        if (this.KeyFrame != undefined && this.KeyFrame.\_\_proto\_\_ == Array.prototype) {

            this.KeyFrame.forEach(KeyF => {

                let KeyFrame = `@keyframes ${KeyF.animate} {

                    ${this.DrawClassList(KeyF.ClassList)}

                }`;

                styleFrag.children.push(KeyFrame);

            });

        }

        this.append(WRender.createElement(styleFrag));

    }

    DrawClassList(ClassList) {

        let bodyStyle = "";

        ClassList.forEach(Class => {

            let bodyClass = "";

            if (Class.\_\_proto\_\_ == Object.prototype) {

                for (const prop in Class.CSSProps) {

                    bodyClass = bodyClass + `${prop}: ${Class.CSSProps[prop]};`;

                }

                bodyClass = `${Class.Name} {${bodyClass}}`;

                bodyStyle = bodyStyle + bodyClass;

            }

        });

        return bodyStyle;

    }

}

Esta clase tendrá algunas características interesantes que podremos revisar parte por parte.

Lo primero es que la creación de toda la lógica del componente style dependerá de la funcionalidad de WRender.createElement, esto permitirá compactar de manera muy significativa el código requerido para poder construir un componente tan complejo. Por otro lado, como es costumbre, no basta con solo verificar que las propiedades existen para proceder la construcción, sino que se debe verificar que sean del tipo correcto para evitar errores lógicos dentro de la aplicación, por ello siempre usamos la validación por medio de la propiedad \_\_proto\_\_ la cual nos permite determinar si el objeto enviado es del prototipo correcto.

if (this.ClassList != undefined && this.ClassList.\_\_proto\_\_ == Array.prototype)

if (this.MediaQuery != undefined && this.MediaQuery.\_\_proto\_\_ == Array.prototype)

 if (this.KeyFrame != undefined && this.KeyFrame.\_\_proto\_\_ == Array.prototype)

este tipo de validaciones son indispensables para evitar comportamientos inesperados dentro de cualquier app. Así mismo se debe verificar que cada objeto de la lista también posee un prototipo adecuado por ello dentro de la función DrawClassList se realiza una verificación similar, esta vez utilizando la definición del prototipo de WCssClass.

  if (Class.\_\_proto\_\_ == WCssClass.prototype)

Por último, solo habrá que definir el componente como customElement y exportar las clases requeridas para facilitar la escritura de estas estructuras.

customElements.define("w-style", WStyledRender);

export { WCssClass };

## Uso de Estilos Normales.

para hacer uso del componente, basta con solo referéncialo y renderizar un w-style dentro de los componentes que necesitemos.

La estructura simple seria la siguiente:

 { type: "w-style", props: {

                ClassList: [

                    new WCssClass(".card", {

                        display: "inline-block",

                        border: "solid 1px #444",

                        margin: "5px",

                    }), new WCssClass(".title", {

                        "font-weight": "bold",

                        padding: "15px 25px",

                        "background-color": "#444",

                        color: "#fff",

                        display: "block",

                    }), new WCssClass("section", {

                        padding: "10px 25px",

                        width: "180px"

                    }),

                ]

            }

 }

En este ejemplo se incluyen dos clases **card** y **title,** con propiedades comunes de estilo y la modificación de un selector por medio del tagName **section** para incluirlo directamente en nuestro componente, haríamos la modificación en este de la siguiente forma (archivo CardComponnet.js, ubicado dentro WDevCore/Wcomponents):

import { WRender } from "../WModules/WComponentsTools.js";

import { WCssClass } from "../WModules/WStyledRender.js";

class WCardComponent extends HTMLElement {

    constructor() {

        super();

        this.attachShadow({ mode: "open" });

    }

    connectedCallback() {

        this.shadowRoot.append(this.DrawCard());

        const style = {

            type: "w-style",

            props: {

                ClassList: [

                    new WCssClass(".card", {

                        display: "inline-block",

                        border: "solid 1px #444",

                        margin: "5px",

                    }), new WCssClass(".title", {

                        "font-weight": "bold",

                        padding: "15px 25px",

                        "background-color": "#444",

                        color: "#fff",

                        display: "block",

                    }), new WCssClass("section", {

                        padding: "10px 25px",

                        width: "180px"

                    }),

                ]

            }

        }

        this.shadowRoot.append(WRender.createElement(style));

    }

    DrawCard() {

        return WRender.createElement({

            type: "label",

            props: { className: "card" },

            children: [

                { type: "label", props: { className: "title", innerText: this.element.title } },

                { type: "section", props: { innerText: this.element.Contain } },

                { type: "section", props: { innerText: this.element.Detail } },

            ]

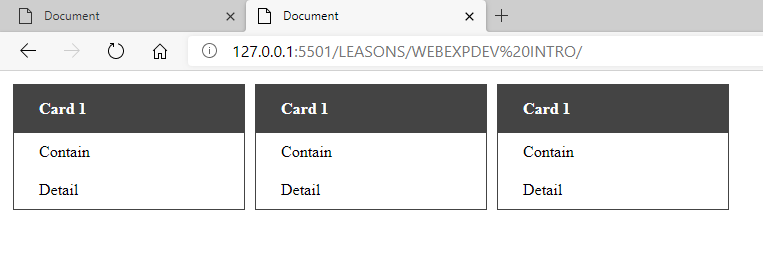
        });

    }

}

customElements.define("w-card", WCardComponent);

al incluir el w-style dentro del shadowRoot del componente, evitamos directamente que este tenga afectación sobre elementos externos al propio CardComponent. Asi mismo evitamos que otros estilos independientes afecten a sus elementos.



## Uso de MediaQuerys

Para hacer uso de MediaQuerys debemos incluir una propiedad con su mismo nombre, esta deberá ser un arreglo de objetos, donde cada elemento del arreglo deberá tener la propiedad condición la cual determinará qué tipo de MediaQuerys implementaremos.

y una ClassList con sus estilos WCssClass, que determinaran cual será el comportamiento de los elementos una vez que se cumpla la condición.

MediaQuery: [{

                    condicion: "(max-width: 1200px)",

                    ClassList: [

                        new WCssClass(`.myClass`, {

                            display: "grid",

                        }),

                    ]

             },{

                    condicion: "(max-width: 600px)",

                    ClassList: [

                        new WCssClass(`.myClass`, {

                            display: "flex",

                        }),

                    ]

             }],

La estructura completa de cómo se vería un w-style con MediaQuerys incluidos, seria la siguiente (Modificando la variable style del CardComponent):

const style = {

            type: "w-style",

            props: {

                ClassList: [

                    new WCssClass(".card", {

                        display: "inline-block",

                        border: "solid 1px #444",

                        margin: "5px",

                    }), new WCssClass(".title", {

                        "font-weight": "bold",

                        padding: "15px 25px",

                        "background-color": "#444",

                        color: "#fff",

                        display: "block",

                    }), new WCssClass("section", {

                        padding: "10px 25px",

                        width: "180px"

                    }),

                ], MediaQuery: [{

                    condicion: "(max-width: 700px)",

                    ClassList: [

                        new WCssClass(`.card`, {

                            display: "block",

                        })

                    ]

                }]

            }

        }

Al aplicarlo a nuestro CardComponent tendremos el siguiente resultado:

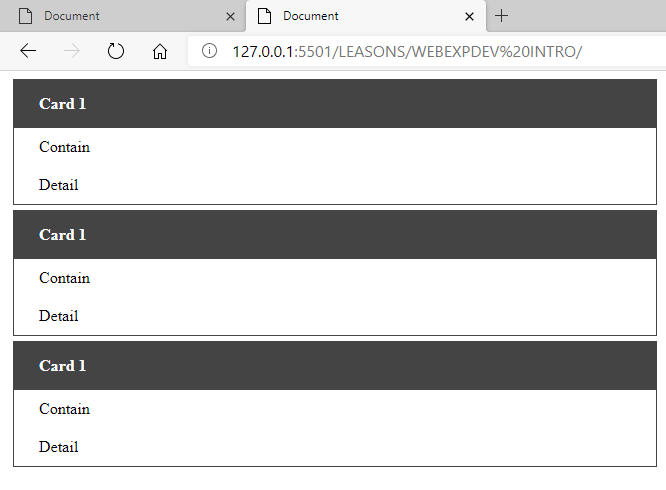
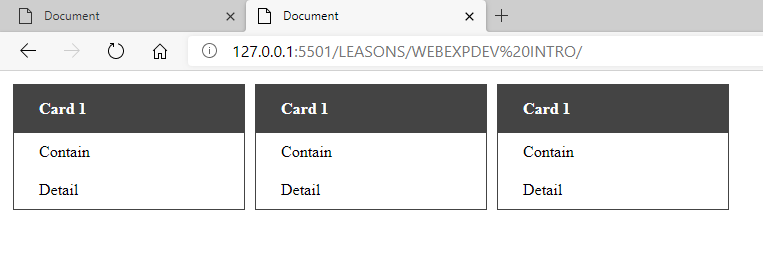
 

Ilustración: resolución mayor a 700 px de ancho.

Ilustración: Resolución menos a 700px

## Uso de KeyFrames

Al igual que los mediaquery los keyframe se rigen bajo la misma lógica que implica un arreglo de objetos que poseen una propiedad anímate y una ClassList que determinan que propiedades CSS se animaran. Ejemplo:

KeyFrame: [{

                    animate: "slide",

                    ClassList: [

                        new WCssClass("0%", {

                            transform: "translateX(-25%)"

                        }), new WCssClass("100%", {

                            transform: "translateX(25%)"

                        })

                    ]

              }]

# Manejo y uso de peticiones asíncronas con Fetch.

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest. Fetch proporciona una alternativa mejor que puede ser empleada fácilmente por otras tecnologías como Service Workers. Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.

El objeto Promise devuelto desde fetch() no será rechazado con un estado de error HTTP incluso si la respuesta es un error HTTP 404 o 500. En cambio, este se resolverá normalmente (con un estado ok configurado a false), y este solo será rechazado ante un fallo de red o si algo impidió completar la solicitud.

La forma de realizar una petición es bastante simple, a partir de su uso dentro de una función asíncrona simplemente se utiliza la siguiente sintaxis:

let response = await fetch(“url”, {

                method: 'GET',

                headers: {

                    'Content-Type': 'application/json',

                    'Accept': 'application/json'

                }

            });

  const dataJSON =  await response.json();

El método puede ser del tipo GET, PUT, DELETE, POST por lo que realizar este tipo de peticiones es bastante similar, excepto que en el caso de la petición POST y PUT se adjunta lo que serian los datos por medio de la propiedad body.

let response = await fetch((“url”, {

                method: 'POST',

                headers: {

                    'Content-Type': 'application/json',

                    'Accept': 'application/json'

                },

                body: JSON.stringify(Datos)

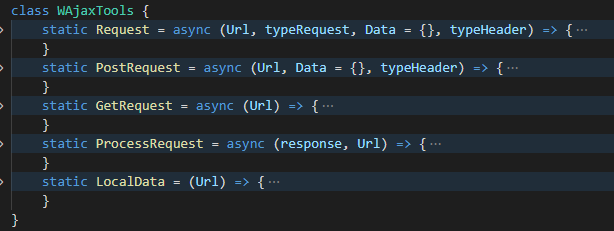
       });

const dataJSON =  await response.json();

## Implementación

En muchas ocasiones querremos que nuestras peticiones sean manejadas con control de errores y es probable que sea necesario que la data esperada por esta sea conservada en el cliente de la aplicación para evitar problemas con APIs deprecadas, servers que por razones desconocidas o fuera de nuestro control están fuera de línea o servicios que tienen sus propios horarios de funcionamiento, o incluso errores inesperados bajo ciertas circunstancias. Un ejemplo claro podría ser que nuestra app esta siendo desarrollada como una PWA (Progresive Web App) y esta requerirá tener funcionamiento fuera de línea. Por ende, esta propuesta incluye una función que intenta manejar estos escenarios valiéndose del API LocalStorage de JS.

La implementación se hará por medio de una clase llamada WAjaxTools dentro de WComponentsTools, esta clase constará de 3 métodos principales y 2 métodos de ayuda.



El primer método estático llamado Request, este recibirá el tipo de petición que ejecutara además de un parámetro Data y el tipo de encabezado que enviara, estos parámetros tendrán valores por defecto para evitar dificultades en su implementación.

static Request = async (Url, typeRequest = "GET", Data = {}, typeHeader) => {

        try {

            let ContentType = "application/json; charset=utf-8";

            let Accept = "\*/\*";

            if (typeHeader == "form") {

                ContentType = "application/x-www-form-urlencoded; charset=UTF-8";

                Accept = "\*/\*";

            }

            let dataRequest =  {

                method: typeRequest,

                headers: {

                    'Content-Type': ContentType,

                    'Accept': Accept

                }

            }

            if (Data != {}) {

                dataRequest.body = JSON.stringify(Data);

            }

            let response = await fetch(Url,);

            const ProcessRequest = await this.ProcessRequest(response, Url);

            return ProcessRequest;

        } catch (error) {

            if (error == "TypeError: Failed to fetch") {

                return this.LocalData(Url);

            }

        }

    }

Sin embargo, para simplificar los llamados y el número de parámetros que se deben declarar crearemos funciones específicas para peticiones GET y POST.

 static PostRequest = async (Url, Data = {}, typeHeader) => {

        try {

            let ContentType = "application/json; charset=utf-8";

            let Accept = "\*/\*";

            if (typeHeader == "form") {

                ContentType = "application/x-www-form-urlencoded; charset=UTF-8";

                Accept = "\*/\*";

            }

            let response = await fetch(Url, {

                method: 'POST',

                headers: {

                    'Content-Type': ContentType,

                    'Accept': Accept

                },

                body: JSON.stringify(Data)

            });

            const ProcessRequest = await this.ProcessRequest(response, Url);

            return ProcessRequest;

        } catch (error) {

            if (error == "TypeError: Failed to fetch") {

                return this.LocalData(Url);

            }

        }

    }

    static GetRequest = async (Url) => {

        try {

            let response = await fetch(Url, {

                method: 'GET',

                headers: {

                    'Content-Type': 'application/json',

                    'Accept': 'application/json'

                }

            });

            const ProcessRequest = await this.ProcessRequest(response, Url);

            return ProcessRequest;

        } catch (error) {

            console.log(error)

            if (error == "TypeError: Failed to fetch") {

                return this.LocalData(Url);

            }

        }

    }

Los métodos ProcessRequest se usará para verificar y procesar el resultado del response y determinaran si ocurrió un error 404 o 500 y en dependencia del resultado retornara datos locales usando la función LocalData o un valor por defecto.

static ProcessRequest = async (response, Url) => {

        if (response.status == 404 || response.status == 500) {

            console.log("ocurrio un error: " + response.status);

            if (typeof response !== "undefined" && typeof response !== "null" && response != "") {

                return this.LocalData(Url);

            } else {

                return [];

            }

        } else {

            response = await response.json();

            localStorage.setItem(Url, JSON.stringify(response));

            return response;

        }

    }

    static LocalData = (Url) => {

        let responseLocal = localStorage.getItem(Url);

        return JSON.parse(responseLocal);

    }

## Peticiones GET

Para hacer uso de estas funciones una petición podemos hacer una importación de la clase dentro del WComponetsTools, por lo que la declaración quedaría de esta forma.

export { WAjaxTools, WRender }

para realizar una petición GET sería tan fácil como ejecutar esta línea de código

const myFun = ()=>{

    WAjaxTools.GetRequest("url");

}

Si deseáramos almacenar algún valor resultante de la petición se ejecutaría dentro de una función asíncrona utilizando la palabra clave await.

const myFun = async ()=>{

    const response = await WAjaxTools.GetRequest("url");

    //hacer algo con response

}

En el caso que no se pudiese utilizar una función asíncrona o no se deseara hacer uso de estas, se puede utilizar la función reservada then, que permite ejecutar acciones después que la promesa es resuelta

const myFun =  ()=>{

    WAjaxTools.GetRequest("url").then((response)=>{

        console.log(response)

//hacer algo con response

    });

}

## Peticiones POST

El caso de las peticiones post es, su uso es bastante similar, en este caso podremos adjuntar un arreglo de datos si fuese requerido.

const myFun = ()=>{

WAjaxTools.PostRequest("url", {datos:"url"});

}

Almacenando el retorno en una variable.

const myFun = async ()=>{

    const response = await WAjaxTools.PostRequest("url", {datos:"url"});

    //hacer algo con response

}

Uso en funciones no asíncronas

const myFun =  ()=>{

    WAjaxTools.PostRequest("url", {datos:"url"}).then((response)=>{

        console.log(response)

//hacer algo con response

    });

}

En este caso debemos de tener en cuenta que es probable que la aplicación envié los datos en formato de formulario y no en formato JSON, por lo que deberemos especificar como parámetro que se enviara una petición con datos en form-part.

En el caso de recuperar los datos a partir de un formulario en html:

const myFun = async ()=>{

    const formData = new FormData(document.getElementById('formulario'));

    const res = await WAjaxTools.PostRequest("url", formData, "form");

    console.log(res);

}

Para crear datos puramente en JS.

const myFun = async ()=>{

    const formData = new FormData();

    formData.append('dato1', 'valor');

    formData.append('dato2', 'valor');

    const res = await WAjaxTools.PostRequest("url", formData, "form");

    console.log(res);

}

# Interfaces dinámicas de una sola página

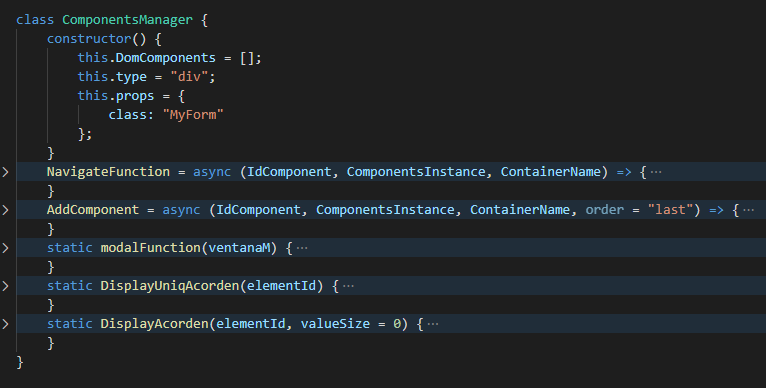
Hoy en día escuchamos términos como páginas estáticas, dinámicas, SPA, Muti-páginas y, recientemente, el término JAMStack. ¿Qué significa todo esto? ¿Por qué es importante entender las diferencias? A medida que se va adquiriendo experiencia, nos damos cuenta de que no existe una "Navaja Suiza" para la web. Por el contrario, cada herramienta suele estar enfocada en solucionar un problema muy particular. Es nuestra responsabilidad entender las ventajas y desventajas para no tomar una mala decisión a la hora de crear un proyecto.

Hay un error muy común de querer desarrollar todo como una SPA (Single Page App), aunque las SPA aumentan la velocidad de nuestro flujo de desarrollo, la realidad es que son pésimas en cuestiones de rendimiento y mantenimiento, por lo que lo más probable es que queramos hacer uso de aplicaciones modularizadas.

Para lograrlo debemos tener un gestor de interfaces y algoritmos capaces de determinar cuales son los módulos que debe mostrar de forma dinámica, además de conservar el estado actual de cada objeto en memoria.

## Implementación

El primer paso a realizar es crear una clase llama ComponentsManager dentro de nuestro WComponentsTools.js, esta clase será la encargada de almacenar y gestionar cada uno de los elementos del DOM que estemos procesando y tendrá funciones y tareas relacionadas a este propósito.



Así mismo una vez implementada deberemos exportar esta clase junto a las otras partes del WComponentsTools.

export { WRender, WAjaxTools, ComponentsManager }

La primera característica de esta clase es que posee un constructor que define algunas características importantes de la clase, entre estas la propiedad this.DomComponents, este es un arreglo de datos que se encargara de almacenar todos los elementos del DOM que deseemos, entre estos bloques completos de componentes complejos. La segunda característica es que hemos definido por defecto la propiedad type y props, el propósito de estas es que se puedan construir componentes que tengan su propia gestión interna de elementos del DOM y haciendo uso de herencia podremos hacer que cualquier clase con características propias pueda tener acceso a estos métodos y funciones sin afectar el comportamiento global de la aplicación.

La clase tendrá 5 funciones principales, la primera llamada NavigateFunction tendrá que ver con la navegación de los sitios, su propósito es que la App muestre solo la información que requiera según las acciones del usuario, recibirá como parámetro el identificador del componente, así mismo recibirá el parámetro ComponentInstance, el cual será la definición del Componente a mostrarse pudiendo ser un objeto JSON, instancia de clase, un HTMLElement o simplemente una cadena de texto, y por ultimo recibirá un ContainerName, el cual hará referencia al nombre del contenedor HTML donde se mostraran los elementos.

NavigateFunction = async (IdComponent, ComponentsInstance, ContainerName) => {

        const ContainerNavigate = document.querySelector("#" + ContainerName);

        let Nodes = ContainerNavigate.querySelectorAll(".DivContainer");

        Nodes.forEach((node) => {

            if (node.id != IdComponent) {

                this.DomComponents[node.id] = node;

                if (ContainerNavigate.querySelector("#" + node.id)) {

                    ContainerNavigate.removeChild(node);

                }

            }

        });

        if (!ContainerNavigate.querySelector("#" + IdComponent)) {

            if (typeof this.DomComponents[IdComponent] != "undefined") {

                ContainerNavigate.append(this.DomComponents[IdComponent]);

                return;

            } else

                const NewChild = WRender.createElement(ComponentsInstance);

                NewChild.id = IdComponent;

                NewChild.className = NewChild.className + " DivContainer";

                this.DomComponents[IdComponent] = NewChild;

                ContainerNavigate.append(NewChild);

                return;

            }

        }

    }

Como podemos observar en la estructura de la función, esta posee un algoritmo que realiza ciertos chequeos al momento de ponerse en marcha. La primera parte tiene como objetivo remover cualquier nodo del componente distinto del que se está seleccionando, sin embargo, este no es destruido, sino que es almacenado dentro de DomComponents con su identificador correspondiente y el estado de los datos tal cual están en ese momento.

const ContainerNavigate = document.querySelector("#" + ContainerName);

        let Nodes = ContainerNavigate.querySelectorAll(".DivContainer");

        Nodes.forEach((node) => {

            if (node.id != IdComponent) {

                this.DomComponents[node.id] = node;

                if (ContainerNavigate.querySelector("#" + node.id)) {

                    ContainerNavigate.removeChild(node);

                }

            }

     });

La segunda parte está relacionada con la creación del objeto y posee tres chequeos fundamentales, el primero es evitar que estos nodos se sobrescriban, al momento de disparar la función.

Si el objeto con ese identificador ya existe dentro del arreglo, no es recreado, sino que es tomado del arreglo tal cual esta en ese momento y lo trae al DOM, sino existe dentro del arreglo es creado y almacenado en el arreglo y a su vez mostrado en el DOM.

if (!ContainerNavigate.querySelector("#" + IdComponent)) {

            if (typeof this.DomComponents[IdComponent] != "undefined") {

                ContainerNavigate.append(this.DomComponents[IdComponent]);

                return;

            } else {

                const NewChild = WRender.createElement(ComponentsInstance);

                NewChild.id = IdComponent;

                NewChild.className = NewChild.className + " DivContainer";

                this.DomComponents[IdComponent] = NewChild;

                ContainerNavigate.append(NewChild);

                return;

            }

        }

La segunda función es AddComponent, esta tiene como propósito agregar nuevos elementos al DOM de forma dinámica en un orden específico y al igual que la función anterior gestiona una copia del componente en la memoria manteniendo sus cambios y estados, evitando así que se agreguen duplicados de estos elementos. Adicional a esto recibirá un parámetro order, el cual determinara si el objeto se insertara al final del contenedor (valor por defecto), o al inicio.

AddComponent = async(IdComponent, ComponentsInstance, ContainerName, order = "last") => {

        const ContainerNavigate = document.querySelector("#" + ContainerName);

        if (ContainerNavigate.querySelector("#" + IdComponent)) {

            window.location = "#" + IdComponent;

            return;

        } else {

            const NewChild = WRender.createElement(ComponentsInstance);

            NewChild.className = NewChild.className + " AddComponent";

NewChild.id = IdComponent;

            this.DomComponents[IdComponent] = NewChild;

            if (order == "last") {

                ContainerNavigate.append(NewChild);

                return;

            } else if (order == "first") {

                ContainerNavigate.insertBefore(NewChild, ContainerNavigate.firstElementChild);

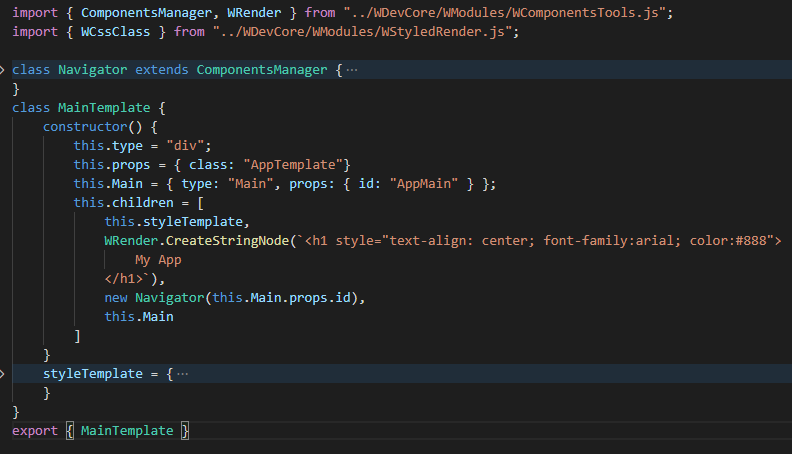
            }

        }

    }

## Ejemplo de NavigateFunction

Dentro de nuestra carpeta AppComponents crearemos un archivo llamado MainTemplate.js y dentro de este crearemos algunas clases que darán pie a la construcción de la interfaz, para luego exportarla para hacer uso de esta dentro de “index.js”. Dentro de este archivo crearemos algunas estructuras de clases que contengan la lógica de la App, una de ellas será la clase MainTemplate, la cual dentro de sus children contendrá los elementos principales del DOM.



Entre los elementos que se incluirá se encuentra el objeto styleTemplate, que incluye los estilos principales de la maqueta.

styleTemplate = {

        type: "w-style", props: {

            ClassList: [

                new WCssClass(".AppTemplate", {

                    display: "grid",

                    "grid-template-rows": "50px 50px calc(100% - 100px)"

                }),

            ]

        }

    }

Así mismo incluye un titulo creado por el método CreatestringNode que crea el nodo a partir de un string y en este caso crea el título “My App”.

WRender.CreateStringNode(`<h1 style="text-align: center; font-family:arial; color:#888">

                My App

        </h1>`);

El siguiente elemento será una instancia de la clase Navigator, la cual extiende de ComponentManager y se encargará de gestionar los componentes, así mismo por medio de su constructor se envía como parámetro el id de la propiedad Main, la cual está definida como un objeto de tipo section y donde se mostrarán los componentes que maneje el Navigator.

new Navigator(this.Main.props.id),

La propiedad Main también s incluida como uno de los hijos de la clase, por lo que también será renderizada al momento que la app se ponga en marcha.

La clase Navigator constara de un constructor que inicializara el tipo del elemento (en este caso será un “nav”), asi mismo se definirá la propiedad class perteneciente a las props con el nombre de clase “nav” y el NavigateContainerId el cual indicara a los elementos del menú donde renderizaran los componentes que creen de forma dinámica, el styleNavigator es el encargado de darle estilos al componente resultante de la renderización de esta clase y entre los children se incluirán primeramente los estilos y dos objetos del tipo “a”, los cuales incluirán un evento onclick cada uno, eventos encargados de preparar todo lo necesario para mostrar el contenido de las vistas.

class Navigator extends ComponentsManager {

    constructor(NavigateContainerId) {

        super();

        this.type = "nav";

        this.props.class = "nav";

        this.NavigateContainerId = NavigateContainerId;

    }

    styleNavigator = {

        type: "w-style", props: {

            ClassList: [

                new WCssClass(".nav", {

                    display: "flex",

                    "border-bottom": "solid 10px #4da6ff"

                }), new WCssClass(".nav a", {

                    margin: "10px",

                    "font-size": "1.3rem",

                    "font-family": "arial",

                    "text-decoration": "none",

                    color: "#999"

                }),

            ]

        }

    }

    children = [

        this.styleNavigator,

        { type: "a", props: { innerText: " Home", href: "#", onclick: () => {

                    const Home = { type: "section", children: [

                        {type: "h2", props: {innerText: "Home page"}},

                        {type: "p", props: {innerText: "Contain"}}

                    ] }

                    this.NavigateFunction("Home", Home, this.NavigateContainerId);

                } }

        }, { type: "a", props: { innerText: "Cards",  href: "#", onclick: async () => {

                    await import("../WDevCore/WComponents/CardComponent.js");

                    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 = WRender.createElement({ type : "section" })

                    for (let index = 0; index < cards.length; index++) {

                        const element = cards[index];

                        Frag.append(WRender.createElement({

 type: "w-card", props: { element: element }

}));

                    }

                    this.NavigateFunction("Cards", Frag, this.NavigateContainerId);

                } }

        }

    ]

}

En este caso el primer elemento “a” al momento de recibir un click creara una estructura JSON para construir un section con un h2 internos y un párrafo para el contenido. El segundo elemento “a” disparar un evento onclick en este caso asíncrono, dado que queremos que haga una importación del CardComponent para luego renderizar el arreglo de cards que teníamos en la definición del evento onload de nuestro index.js.

Por ello haremos una modificación del script del index.js, para que simplemente importe y renderice nuestro MainTemplate.

const OnLoad = async ()=>{

    const {Render} = await import("./Scripts/toolComponets.js")

    const {MyContainer} = await import("./Scripts/MasterTemplate.js")

    const Contenedor = new MyContainer();

    myRoot.append(Render(Contenedor));

}

window.onload = OnLoad;

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];

        Frag.append(WRender.createElement({type: "w-card", props: {element: element}}));

    }

    App.append(Frag);

}

El resultado obtenido de este ejemplo será el siguiente:

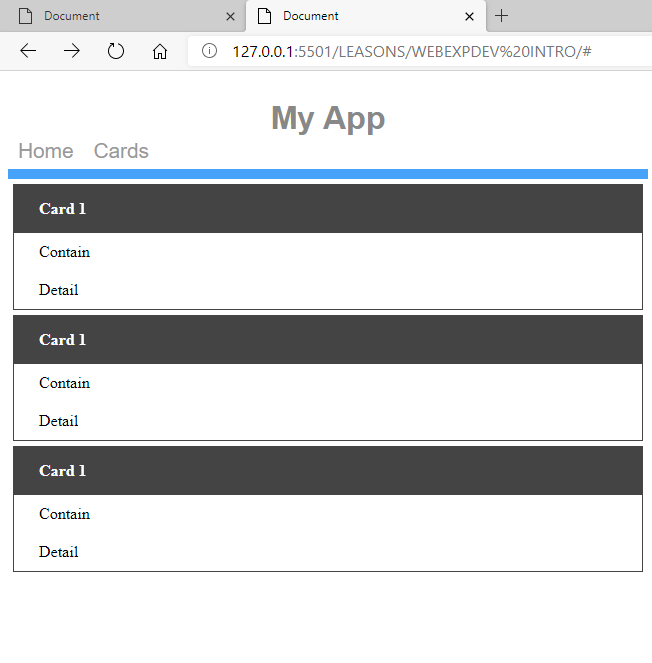
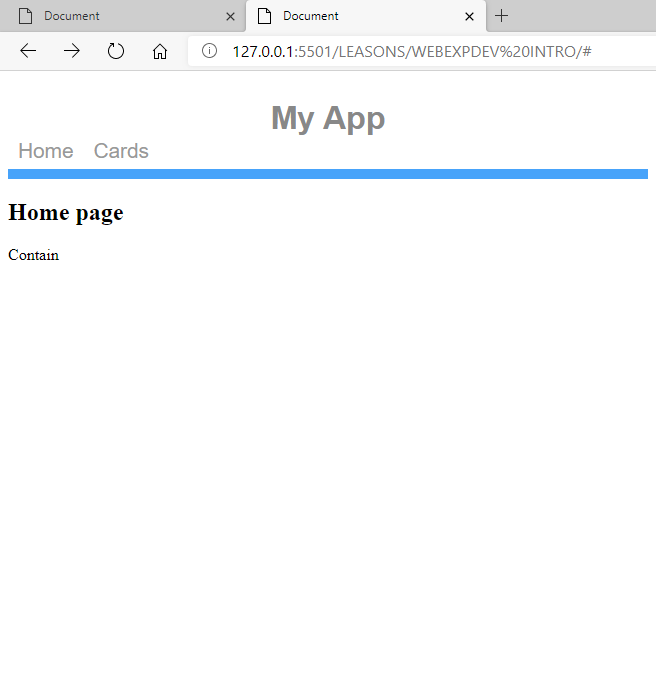
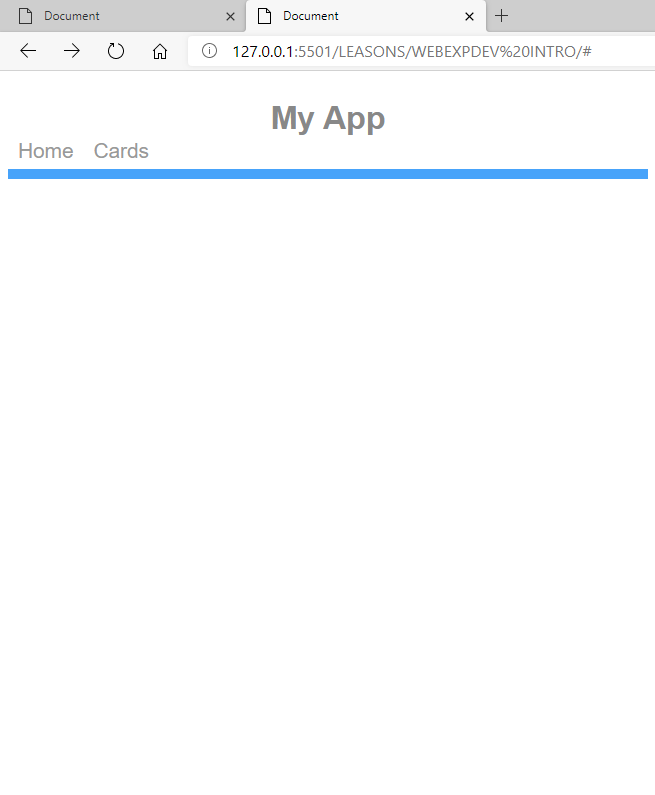


Ilustración: interfaz inicial Ilustración: Al dar clic en Home Ilustración: Al dar clic en Cards

Este método garantizara que la carga de cada uno de los contenidos del menú no sen procesados ni importados (en el caso de ser componentes externos), a menos que sean necesarios, agilizando la carga de la pagina y procurando la optimización de los procesos.