## 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.

```
class 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__.__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__.__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.

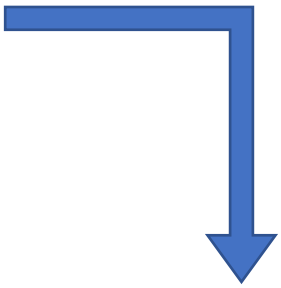
```
return element;
```

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.de
```



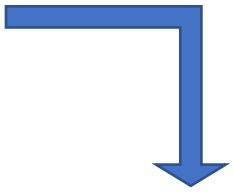
```
class WCardComponent extends HTMLElement {
    constructor() {
        super();
        this.attachShadow({ mode: "open" });
    }
    connectedCallback() {
        this.shadowRoot.append(this.DrawCard());
    }
    DrawCard() {
        return WRender.createElement({
            type: "label", props: { 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");
```