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).

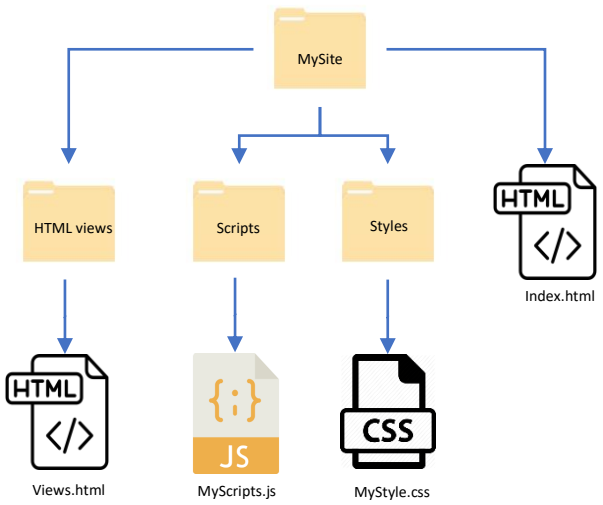

Ilustración: estructura básica de sitio web

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

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*

```html
<!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.

```js
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.