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 cuáles 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.

```
class ComponentsManager {
    constructor() {
        this.DomComponents = [];
        this.type = "div";
        this.props = {
            class: "MyForm"
        };
    }
    NavigateFunction = async (IdComponent, ComponentsInstance, ContainerName) => {···
    }
    AddComponent = async (IdComponent, ComponentsInstance, ContainerName, order = "last") => {···
    }
    static modalFunction(ventanaM) {···
    }
    static DisplayUniqAcorden(elementId) {···
    }
    static DisplayAcorden(elementId, valueSize = 0) {···
    }
}
```

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.

```
import { ComponentsManager, WRender } from "../WDevCore/WModules/WComponentsTools.js";
import { WCssClass } from "../WDevCore/WModules/WStyledRender.js";

class Navigator extends ComponentsManager {···
}
class MainTemplate {
    constructor() {
        this.type = "div";
        this.props = { class: "AppTemplate"}
        this.Main = { type: "Main", props: { id: "AppMain" } };
        this.children = [
            this.styleTemplate,
            WRender.CreateStringNode(`<h1 style="text-align: center; font-family:arial; color:#888">
                My App
            </h1>`),
            new Navigator(this.Main.props.id),
            this.Main
        ]
    }
    styleTemplate = {···
    }
}
export { MainTemplate }
```

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.

```
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);
}
```

```
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;
```

El resultado obtenido de este ejemplo será el siguiente:

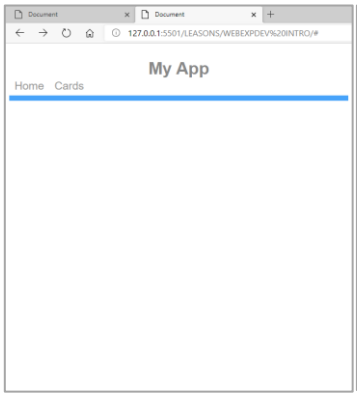

Ilustración: interfaz inicial

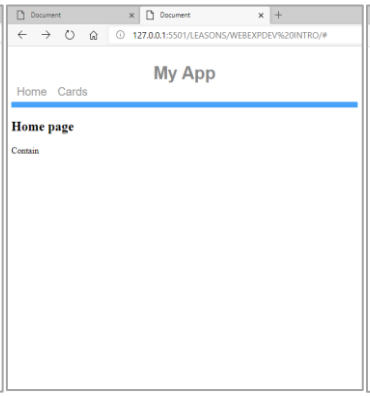

Ilustración: Al dar clic en Home

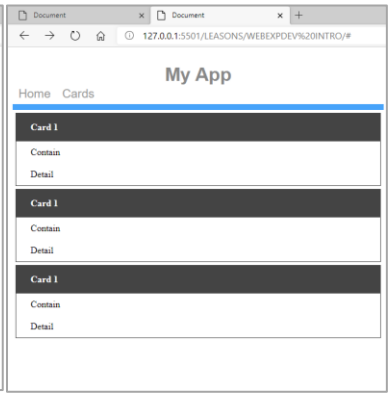

Ilustración: Al dar clic en Cards

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 página y procurando la optimización de los procesos