# Instalación y configuración para el uso del framework para el desarrollo Front End: React

## Node.js y NPM

Como paso inicial, al tener que ocupar el manejador de paquetes npm en nuestros proyectos debemos de instalarlo, para eso instalamos Node.js en nuestro sistema. Se instala la versión más actual (v17.7.2), y en el instalador se ubica en donde se requiera, con las opciones predeterminadas.

Para más información de Node.js consultar Documentation | Node.js (nodejs.org)

Para determinar si se realizó la instalación de npm de forma correcta, se ejecuta el comando

*npm -v*

dándonos como resultado que la versión actual de npm es la 8.5.2. Podemos modificar esta versión con el comando

*npm install -g npm@[version]*

Para más información de npm consultar npm Docs (npmjs.com)

## React

Debemos de ubicar dónde queremos crear nuestro proyecto para la parte de front-end, si estamos trabajando un proyecto donde esta tanto back-end y front-end entonces debemos de ponerlo al mismo nivel que la carpeta (recomendable 'server') para el back-end y llamar a nuestro proyecto de React (recomendable 'client').
Si no, solo nos ubicamos en un directorio donde queramos crear el proyecto.

Ejecutamos el comando

*npx create-react-app [nombre_del_proyecto]*

Esto va a crear la configuración para nuestro proyecto de React, además de instalar paquetes como react, react-dom, react-scripts, y create-react-app si no habíamos creado un proyecto de react en nuestro sistema anteriormente.

Si estamos corriendo un servidor para la parte back-end en un puerto específico (recomendable 3000), necesitamos acceder al archivo package.json y cambiar en la sección de scripts el puerto para el front-end (recomendable 8080), el script a ejecutar cuando se hace el npm o yarn start, tal que tenemos un json como

```
{…
“scripts”:{
        …
        “start”: “PORT=[puerto] react-scripts start”,
        …
}
…}
```

Y ahora somos libres para crear nuestra aplicación.

## Esqueleto de Proyecto o Boilerplate React

Dentro de los boilerplates, se tiene una variedad para cada tipo de proyecto, y dependen muchos de las bibliotecas a utilizar.

- El comando create-react-app ya nos genera un boilerplate, el cual incluye Babel para que el código se pueda leer por navegadores en cualquier versión, y Webpack que recopila todo nuestro código en módulos para juntarlo para el navegador. Getting Started | Create React App (create-react-app.dev)
- Hay una página oficial donde se tiene un boilerplate para React con componentes y contenedores separados, rutas, selectores, y sagas para tener una estructura robusta. React.js Boilerplate (reactboilerplate.com)
- Existen otros boilerplates como Gatsby Static, React Starter Kit, entre otros que se pueden encontrar fácilmente.

Nosotros ocupamos create-react-app, sin embargo vamos a tener que instalar otras bibliotecas, además de crear carpetas para poder crear una aplicación que pueda utilizar React Query, API calls en Axios, y que sea con componentes de forma modular teniendo que el estilo o CSS del componente estará en el mismo archivo del componente (JSS), se tendrán también hooks por componente para tener de una parte la parte visual y en el hook el manejo de información, las acciones y toda la lógica de negocios. Cada componente con su archivo y hook tendrá su carpeta, y estas carpetas van a estar en forma de árbol por cada sección contenedora de la aplicación, por lo que el directorio inicial será src/components y de ahí se crearán carpetas para cada contenedor y subsecuentemente para cada contenido o componente dentro del contenedor, esto es una excepción para aquellos componentes que se puedan reusar dentro de cada sección o que son generales. Las solicitudes de datos o envíos de datos por endpoints se manejan en una carpeta de forma separada. El manejo del estado global y las acciones se realizan con Redux, por lo que se necesita su directorio separado. Resultando en el siguiente árbol de directorios:

- *src*
    - *api*

        - *…*
    - *components*
        - *…*
    - *redux*
        - *actions*
        - *modules*
        - *selectors*

## Conceptos a utilizar de React

Dentro de React se manejan los componentes, estos para mayor facilidad de lectura pueden utilizar la extensión del lenguaje para javascript llamada jsx, esto nos permite declarar componentes los cuales son funciones, que regresan algo que aparenta ser HTML, al que puedes incluir variables o procesos como si fuera Javascript, en el mismo sitio.

```
const Componente = (props) => {
        …
        return (
                <section>
                        <h1>{props.titulo}</h1>
                        <p>Este es un componente.</p>
                </section>
        );
};
```

Son componentes que pueden ser en realidad clases o funciones, depende del gusto del desarrollador, pero se debe mantener una consistencia en todo el proyecto, por lo que hablaremos de los componentes funcionales. Estos componentes se pueden utilizar como variables, o en su sintaxis jsx como

```
<Componente/>
```

Componentes que aceptan ‘props’, simplemente es una variable que puede contener otras variables, esto sirve para mandar información de un componente superior a este, para poder ser mostrada. También se puede ver como el estado global de aplicación, asignado a esta variable. Si se utiliza el componente como contenedor, se le puede asignar como prop los componentes hijos, pero la manera simple y limpia utilizando jsx es simplemente usarlo como elemento de HTML:

```
…
<ComponenteContenedor>
        …
        <ComponenteHijo/>
        …
</ComponenteContenedor>
…
```

Y dentro de los argumentos del componente, después de props, se puede obtener todo lo que esta como hijo del componente con la variable children.

```
const Componente = (props, children) => {
        …
        return (
                <section>
                        …
                        {children}
                </section>
        );
};
```

El estado global de la aplicación o Redux Store, es literalmente un objeto que contiene para cada sección de nuestro proyecto, un objeto y estos tienen las variables necesarias para los componentes a manejar. Para manejar este estado, se utilizan acciones, estas acciones tienen un tipo y un 'payload' o valor de retorno, lo cual nos sirve para que en un 'reducer' que esta constantemente checando por acciones, podamos modificar las variables del estado global.

```
…
const ACCIÓN= 'acción';
…
export const acción = (payload) => ({
 type: ACCIÓN,
 payload
});
…
const estadoInicial = {};
…
const observadorModificador = (estado = estadoInicial , acción) => {
   switch (acción.tipo) {
      case "ACCIÓN":
           return {...state, acciónTomada: true};
      }
      default: return estado;
   }
};
export default observadorModificador;
```

Estos componentes tienen un ciclo de vida, se 'renderizan', cambia un valor dentro del componente, y se destruyen, este ciclo de vida se puede manejar con el hook llamado 'useEffect', entre otros hooks para casos particulares. Se puede modificar este hook o función personalizada para que chequee todos los cambios, que solo se ejecute algo cuando se renderiza el componente, o se puede usar una función cleaner la cual realizara acciones cuando se destruya el componente.

```
useEffect(()=>{
        …
```

*}, []);*

Renderizar, se habló de este concepto y sencillamente React no utiliza el DOM directamente, crea una copia virtual de este y solo esta revisando por los cambios realizados, los actualiza y manda esta copia con los cambios, es decir, no genera todo el DOM entre actualizaciones.

Los componentes también pueden tener estado, este estado se maneja con el hook 'useState' el cual nos proporciona con la variable del estado que queremos manejar, y su setter, esto pues las variables del estado no pueden ser mutadas o cambiadas de forma directa para evitar errores. Se le puede asignar un valor inicial a este estado local.

```
const [estadoLocal, setEstadoLocal] = useState(estadoInicial);
…
const modificadorDelEstado = (modificacion) => setEstadoLocal(estadoPrevio => ({...estadoPrevio, modificacion}));
```

Para simplificar los componentes, se debe evitar crear varios componentes en el mismo archivo, y se usan hooks personalizados, estos consisten en utilizar datos iniciales que tu proporcionas, se maneja el estado del componente, sus ciclos de vida, 'callbacks' o funciones que son ejecutadas por un evento, llamadas de API o acciones despachadas al Redux, en estos hooks se puede hacer lo que se necesite dada la lógica de negocio.

```
const Componente = ({estadoInicial}, children) => {
        const {variables, funciones, …estado} = useHookPersonalizado(estadoInicial);
        …
        return (...);
};

const useHookPersonalizado = (estadoInicial) => {
        …
        return {
                variables,
                funciones,
                …
        };
};
```

Se debe de tener en cuenta que los componentes pueden recibir datos dentro de los 'props', pero no pueden mandar datos directamente hacia el componente padre o contenedor, para esto se utiliza Redux, para poder conectar los componentes al estado global y entre sí sin tener que bajar los datos por el árbol de aplicación.

```
const store = configureStore({
  reducer: {
     datos,
  },
});
```

```
const seleccionaDatos = (estado) => estado.datos;

const seleccionaDatoPorId = createSelector(
  [seleccionaDatos, (estado, id) => id],
  (datos, id) => datos.find(({ identificador }) => identificador === id)
);

const dato = useSelector((estado) => seleccionaDatoPorId(estado, id));
```

Para conectar la aplicación en sus diferentes contenedores, generando un 'SPA' o aplicación de una sola página, se utiliza React Router, en la base de nuestro proyecto el componente tendrá las diferentes secciones de la página en componentes contenedores, donde se les asigna una sección o subdirectorio del url diferente.

```
<Router>
        <>
                …
                <Link to="/">Inicio</Link>
                …
                <Routes>
                        …
                        <Route exact path='/' element={<Inicio/>}></Route>
                        …
                </Routes>
        </>
</Router>
```

Los componentes nos van a regresar el contenido que queramos con su HTML, CSS, y funcionalidad, sin embargo la parte del API y las llamadas a sus distintos endpoints puede realizarse de distintas formas, directamente en el componente (no recomendable), utilizando un middleware como React Redux Sagas y teniendo acciones que puedan ser 'trigger' o iniciar esta llamada, claro teniendo que almacenar estos valores en el estado global, o se puede realizar de forma modular utilizando React Query, solo basta con tener una configuración en la base del proyecto, y utilizando sus diferentes funciones con valores asignados para realizar las llamadas y obtener los datos de forma modular para el componente en los hooks personalizados mencionados anteriormente. Si se va a utilizar un hook por cada componente y manejando hooks como useState o useEffect se recomienda mantenerlo simple con Axios, Promise, y resolviendo el resultado.

También debemos considerar que al desarrollar con React, se debe de seguir utilizando una buena práctica para la utilización de elementos de HTML, asignar clases con el atributo de elemento className y no utilizar estilos en linea, ademas de tener mucho conocimiento de las buenas y nuevas prácticas al utilizar Javascript como utilizar funciones callbacks, utilizar ternary operators o utilizando las funciones de arreglos map/filter/find/sort, entre otras para simplificar el código.

Para más información sobre React consultar [Empezando – React (reactjs.org)]

Para más información sobre React Redux consultar [Getting Started with React Redux | React Redux (react-redux.js.org)]

## Instalación e implementación de bibliotecas para React

Para instalar cualquier biblioteca que se quiera usar, se utiliza el comando

*npm install [biblioteca]*

Todas las bibliotecas o la mayoría de ellas sigue el patrón de tener una configuración en el componente principal a incluir en el HTML como contenedor de la base de nuestra aplicación, y para el uso de las funcionalidades se puede hacer con la instrucción

*import defaultExport,{namedExports…} from '[biblioteca]';*

dentro de cualquier componente o archivo del proyecto.

Para más información, buscar la biblioteca a utilizar para su instalación, implementación, y documentación en general.

Repositorio con código de un ejemplo de aplicación: https://github.com/jpfragoso30/ReduxUsers