Clase 2: Material Complementario

Sitio:	Centro de E-Learning - UTN.BA	Imprimido por:	Nelson Brian Avila Solano
Curso:	Curso de Backend Developer - Turno Noche	Día:	Tuesday, 4 de November de 2025, 20:06
Libro:	Clase 2: Material Complementario		

Tabla de contenidos

1. Introducción React Js y Node Js

Bloques temáticos

1. Introducción.
2. JSX.
3. create-react-app.
4. Componentes.

1.1. Introducción React Js y Node Js

Introducción

ReactJS es una librería JavaScript de código abierto enfocada a la visualización que nos permite el desarrollo de interfaces de usuario de forma sencilla mediante componentes interactivos y reutilizables.

Una librería de JavaScript es un conjunto de funciones y herramientas prescritas que pueden ser utilizadas por desarrolladores para facilitar la creación de aplicaciones web.

Las librerías de JavaScript pueden incluir funciones para manipular elementos HTML, interactuar con servidores, crear animaciones, gestionar eventos de usuario, entre otras cosas.

React está basado en un paradigma llamado programación orientada a componentes en el que cada componente es una pieza con la que el usuario puede interactuar. Estas piezas se crean usando una sintaxis llamada JSX permitiendo escribir HTML (y opcionalmente CSS) dentro de objetos JavaScript.

Estos componentes son reutilizables y se combinan para crear componentes mayores hasta configurar una web completa.

Esta es la forma de tener HTML con toda la funcionalidad de JavaScript y el estilo gráfico de CSS centralizado y listo para ser abstraído y usado en cualquier otro proyecto.

JSX

JSX es una extensión de sintaxis de Javascript, que como podemos observar se parece a HTML.

Uno de sus beneficios es mejorar la legibilidad del código, mejorar la experiencia del desarrollador, y reducir la cantidad de errores de sintaxis, ya que no es necesario repetir tantas veces el mismo código. Otro beneficio es el de poder integrar a otros miembros de tu equipo que no sean programadores en el desarrollo. Es “fácil” leer el código si ellos están acostumbrados a leer HTML.

React no requiere usar JSX, pero la mayoría de la gente lo encuentra útil como ayuda visual cuando trabajan con interfaz de usuario dentro del código Javascript. Esto también permite que React muestre mensajes de error o advertencia más útiles.

Insertando expresiones en JSX

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

En el ejemplo declaramos una variable llamada name y luego la usamos dentro del JSX envolviendola dentro de llaves.

```
function formatName(user) {
    return user.firstName + ' ' + user.lastName;
}
const user = {
    firstName: 'Harper',
    lastName: 'Perez'
};
const element = (
    <h1>
        Hello, {formatName(user)}!
    </h1>
);
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

En el ejemplo insertamos el resultado de llamar la función de JavaScript, formatName(user), dentro de un elemento <h1>.

Especificando atributos con JSX

Podemos utilizar comillas para especificar strings literales como atributos:

```
const element = <div tabIndex="0"></div>;
```

También puedes usar llaves para insertar una expresión JavaScript en un atributo:

```
const element = <img src={user.avatarUrl}></img>;
```

Podemos usar comillas rodeando llaves cuando insertamos una expresión JavaScript en un atributo. Debemos utilizar comillas (para los valores de los strings) o llaves (para las expresiones), pero no ambas en el mismo atributo.

Advertencia:

Dado que JSX es más cercano a JavaScript que a HTML, React DOM usa la convención de nomenclatura camelCase en vez de nombres de atributos HTML.

Por ejemplo, class se vuelve className en JSX, y tabindex se vuelve tabIndex**.**

create-react-app

create-react-app es un ambiente cómodo para aprender React, y es la mejor manera de comenzar a construir una nueva aplicación de página única usando React.

create-react-app configura tu ambiente de desarrollo de forma que puedas usar las últimas características de Javascript, brindando una buena experiencia de desarrollo, y optimizando tu aplicación para producción. Necesitarás tener Node mayor o igual a la versión 10.16 y npm mayor o igual a la versión 5.6 instalados en tu máquina.

Para crear un proyecto ejecuta:

```
create-react-app nombredemiproyecto cd nombredemiproyecto npm start
```

create-react-app no se encarga de la lógica de backend o de bases de datos; tan solo crea un flujo de construcción para frontend, de manera que lo puedes usar con cualquier backend

Componentes

Componentes de clase vs. componentes de función

Los componentes de React pueden definirse de dos formas distintas: como funciones o como clases.

```
class Saludo extends React.Component {
    render() {
        return <h1>Hola, {this.props.name}</h1 >;
    }
}
```

Ejemplo de componente de clase

```
function Saludo(props) {
    return <h1>Hola, {props.name}</h1>;
}
```

Ejemplo de componente de función

Ambos componentes son equivalentes, sin embargo la forma más sencilla, y la más utilizada en las últimas actualizaciones es de función.

El ejemplo de función es un componente de React válido porque acepta un solo argumento de objeto “props” (que proviene de propiedades) con datos y devuelve un elemento de React. Llamamos a dichos componentes “funcionales” porque literalmente son funciones JavaScript.

Creación de componentes

Como vimos anteriormente podemos definir un componente usando una simple función, y su valor de retorno será el contenido HTML que se muestre. Así mismo podemos incluir todos los componentes que necesitemos dentro del valor de retorno de otros componentes. Este concepto se llama composición.

```
function Saludo(props) {
    return <h1>Hola, {props.name}</h1>;
}
function App() {
    return (
        <div>
            <Saludo name="Sara" />
            <Saludo name="Cahal" />
            <Saludo name="Edite" />
        </div>
    );
}
```

Por lo general, las aplicaciones de React nuevas tienen un único componente App en lo más alto. Sin embargo, si se integra React en una aplicación existente, se podría empezar de abajo hacia arriba con un pequeño componente como Button y poco a poco trabajar el camino a la cima de la jerarquía de la vista.

Es importante que en cada archivo en el que vayamos a definir un componente (ya sea de clase o funcional recordemos importar React al principio de nuestro archivo js y exportarlo al final.

```
// components/MiComponente.js
import React from 'react'
const MiComponente = (props) => {
    return <p>Soy un componente</p>;
};
export default MiComponente;
```

Asumiendo que este componente se encuentre en la carpeta componentes, dentro del archivo MiniComponente.js podríamos importarlo dentro de otro componente en la misma carpeta de la siguiente forma:

```
// components/OtroComponente.is
import React from 'react'
import MiComponente from ' ./MiComponente'
const OtroComponente = (props) => {
    return (
        <>
            <h1>Soy Otro Componente</h1>
            <MiComponente />
        </>
    )
}
export default OtroComponente;
```

En caso de ser archivos locales usamos siempre la ruta relativa desde el archivo en el que estamos escribiendo. No es necesario agregar la extensión a los archivos js al importarlos.

Bibliografía utilizada y sugerida

Artículos de revista en formato electrónico:

- **MDN Web Docs.** Disponible desde la URL: https://developer.mozilla.org/
- **React**.Disponible desde la URL: https://es.reactjs.org/

Libros y otros manuscritos:

- **Alex Banks & Eve Porcello.** Learning React: desarrollo web funcional con React y Redux. 2017.

2. Instalación y Configuración de React JS

Bloques temáticos:

- Virtual DOM
- Ventajas de utilizar React JS
- Node JS
- Crear una aplicación utilizando el CLI
- Componentes
- Tipos de componentes
- JSX
- Componentes con clases
- Componentes con funciones
- Estructura de directorio

2.1. Instalación y Configuración de React JS

Virtual DOM

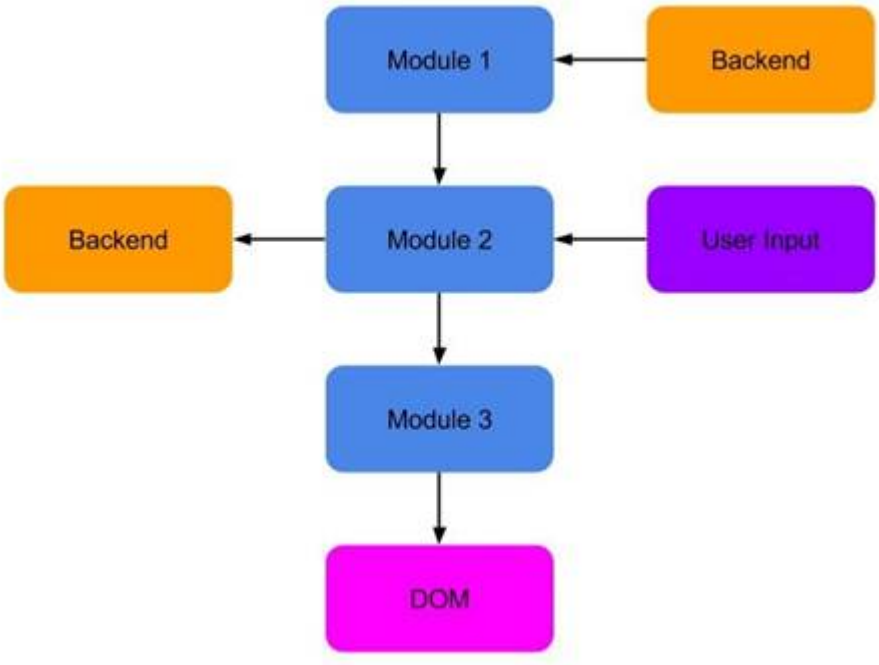
La clave de React JS

El secreto de ReactJS para tener un performance muy alto, es que implementa algo llamado Virtual DOM y en vez de renderizar todo el DOM en cada cambio, que es lo que normalmente se hace, este hace los cambios en una copia en memoria y después usa un algoritmo para comparar las propiedades de la copia en memoria con las de la versión del DOM y así aplicar cambios exclusivamente en las partes que varían.

Esto puede sonar como mucho trabajo, pero en la práctica es mucho más eficiente que el método tradicional pues si tenemos una lista de dos mil elementos en la interfaz y ocurren diez cambios, es más eficiente aplicar diez cambios, ubicar los componentes que tuvieron un cambio en sus propiedades y renderizar estos diez elementos, que aplicar diez cambios y renderizar dos mil elementos.

Son más pasos a planear y programar, pero ofrece una mejor experiencia de usuario y una planeación muy lineal.

Una característica importante de ReactJS es que promueve el flujo de datos en un solo sentido, en lugar del flujo bidireccional típico en Frameworks modernos, esto hace más fácil la planeación y detección de errores en aplicaciones complejas, en las que el flujo de información puede llegar a ser muy complejo, dando lugar a errores difíciles de ubicar.



¿Cómo funciona el virtual DOM?

JavaScript puro es el lenguaje declarativo utilizado para crear aplicaciones web interactivas, React es una biblioteca de JavaScript que ayuda a los desarrolladores a construir interfaces de usuario interactivas y reutilizables.

Imagina que tienes un objeto que es un modelo en torno a una persona. Tienes todas las propiedades relevantes de una persona que podría tener, y refleja el estado actual de la persona. Esto es básicamente lo que React hace con el DOM.

Ahora piensa, si tomamos ese objeto y le hacemos algunos cambios. Se ha añadido un bigote, unos bíceps y otros cambios. En React, cuando aplicamos estos cambios, dos cosas ocurren:

- En primer lugar, React ejecuta un algoritmo de “diffing”, que identifica lo que ha cambiado.
- El segundo paso es la reconciliación, donde se actualiza el DOM con los resultados de diff.

La que hace React, ante estos cambios, en lugar de tomar a la persona real y reconstruirla desde cero, sólo cambiaría la cara y los brazos. Esto significa que si usted tenía el texto en una entrada y una actualización se llevó a cabo, siempre y cuando nodo padre de la entrada no estaba programado para la actualización, el texto se quedaría sin ser cambiado.

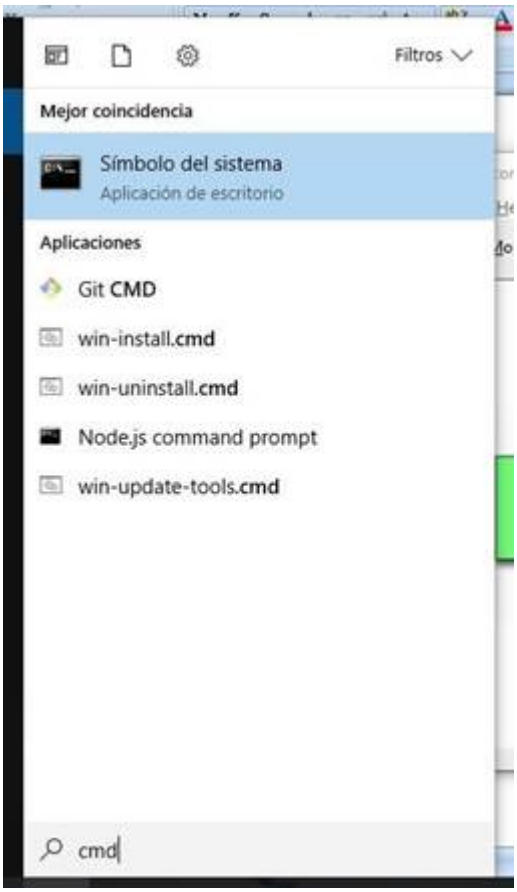
Ventajas de utilizar React JS

- Performance con DOM Virtual
- Flujo unidireccional de datos y eventos a través de una jerarquía de componentes modulares con punto único de entrada (programación reactiva)
- Binding unidireccional entre el view (vista) y modelo, se evita complejidad y explosión de eventos difíciles para debugging.
- Se usa realmente por sus creadores
- Se integra directamente con implementaciones del paradigma también reactivo de gestión de datos Flux (como Redux) que maneja los datos de la aplicación en una división también jerárquica de componentes contenedores vinculados al estado de los datos en el store, y que pasan estos datos y funciones gestores de esos datos a componentes puros que demuestran los datos en forma predecible sin efectos secundarios y ofrecen puntos de interacción al usuario.
- Es fácil incluir rendering (proyección del componente en el DOM virtual como nodos de elementos con estilo y comportamiento, o sea HTML + CSS + JavaScript) en el lado del servidor con funciones sencillas, para lograr las llamadas aplicaciones universales (antes se llamaban isomórficas).

Crear una aplicación utilizando el CLI

Trabajar con la consola de windows

Abrir la consola



Ubicarnos en un directorio específico

Con el comando cd podemos ingresar al directorio sobre el cual vamos a crear nuestra aplicación en react. Con cd.. Volvemos al directorio anterior.

```
G:\sites>cd react
```

Luego estamos dentro del directorio react

```
G:\sites\react>
```

Para ver más sobre el uso de consola en Windows:

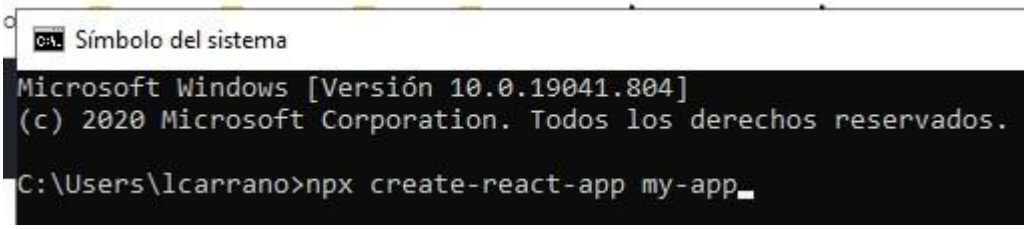
https://computerhoy.com/paso-a-paso/software/comandos-esenciales-consola-windowscmd-que-debes-conocer-77943

Para ver más sobre el uso de la consola en Linux / mac:

https://swcarpentry.github.io/shell-novice-es/02-filedir/index.html

Crear una aplicación

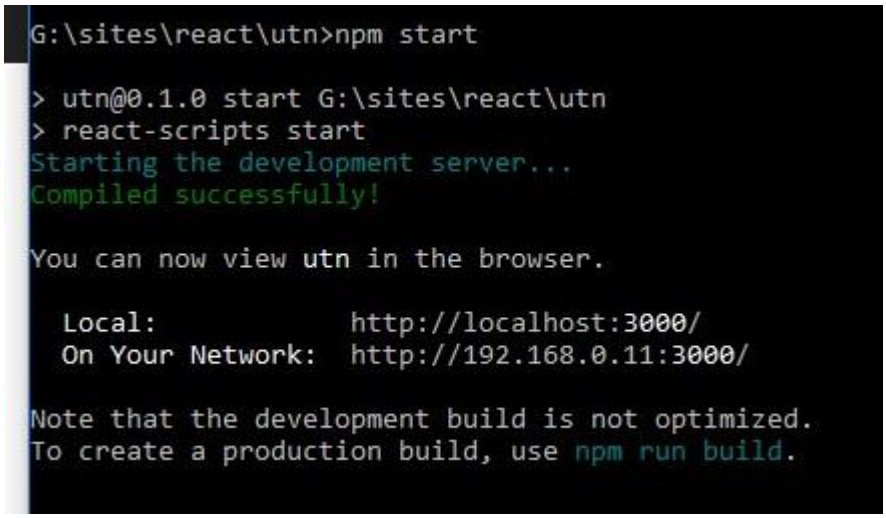
Para crear nuestra aplicación debemos ejecutar npx create-react-app my-app



Ejecutar aplicación en el navegador

Para ejecutar una aplicación y poder acceder desde el navegador debemos ejecutar

(dentro del directorio de la aplicación creada) npm start



El resultado obtenido será el siguiente:

Componentes

En pocas palabras

Comenzamos un tema importante en React, ya que en esta librería realizamos un desarrollo basado en componentes. Básicamente quiere decir que, mediante anidación de componentes podremos crear aplicaciones completas de una manera modular, de fácil mantenimiento a pesar de poder ser complejas.

En React JS existen dos categorías recomendadas para los componentes: los componentes de presentación y los componentes contenedores.

Los **componentes de presentación** son aquellos que simplemente se limitan a mostrar datos y tienen poca o nula lógica asociada a manipulación del estado, es preferible que la mayoría de los componentes de una aplicación sean de este tipo porque son más fáciles de entender y analizar.

Los **componentes contenedores** tienen como propósito encapsular a otros componentes y proporcionarles las propiedades que necesitan, además se encargan de modificar el estado de la aplicación por ejemplo usando Flux o Redux para despachar alguna acción y que el usuario vea el cambio en los datos.

Ventajas del enfoque

- Favorece la separación de responsabilidades, cada componente debe tener una única tarea.
- Al tener la lógica de estado y los elementos visuales por separado es más fácil reutilizar los componentes.
- Se simplifica la tarea de hacer pruebas unitarias.
- Puede mejorar el rendimiento de la aplicación.
- La aplicación es más fácil de entender.

Composición de componentes

Así como en programación funcional (paradigma de programación que se basa en la idea de que los programas deben ser escritos en términos de funciones matemáticas puras) se pasan funciones como parámetros para resolver problemas más complejos, creando lo que se conoce como composición funcional, en ReactJS podemos aplicar este mismo patrón mediante la composición de componentes

Las aplicaciones se realizan con la composición de varios componentes. Estos componentes encapsulan un comportamiento, una vista y un estado. Pueden ser muy complejos, pero es algo de lo que no necesitamos preocuparnos cuando estamos desarrollando la aplicación, porque el comportamiento queda dentro del componente y no necesitamos complicarnos por él una vez se ha realizado.

En resumen, al desarrollar crearemos componentes para resolver pequeños problemas, que por ser pequeños son más fáciles de resolver y en adelante son más fáciles de visualizar y comprender. Luego, unos componentes se apoyarán en otros para resolver problemas mayores y al final la aplicación será un conjunto de componentes que trabajan entre sí. Este modelo de trabajo tiene varias ventajas, como la facilidad de mantenimiento, depuración, escalabilidad, etc.

```
function Welcome(props) {
    return <h1>Hello, {props.name}</h1>;
}
function App() {
    return (
        <div>
            <Welcome name="Sara" />
            <Welcome name="Cahal" />
            <Welcome name="Edite" />
        </div>
    );
    ReactDOM.render(
        <App />,
        document.getElementById('root')
    );
```

En este caso vemos como el componente app inyecta al componente welcome

Tipos de componentes

Características de los componentes de presentación

- Orientados al aspecto visual
- No tienen dependencia con fuentes de datos (e.g. Flux)
- Reciben callbacks por medio de props
- Pueden ser descritos como componentes funcionales.
- Normalmente no tienen estado

Ejemplo de componente de presentación

```
// Componentes de presentación
class Item extends React.Component {
    render() {
        return (
            <li>
                <a href='#'>{this.props.valor}</a>
            </li>
        );
    }
}
class Input extends React.Component {
    render() {
        return (
            <input type='text' placeholder={this.props.placeholder} />
        );
    }
}
class Titulo extends React.Component {
    render() {
        return (
            <h1>{this.props.nombre}</h1>
        );
    }
}
```

En este fragmento de código definimos algunos componentes de presentación (Header, Item e Input) que son mostrados en la página dentro de un componente contenedor.

Los componentes de presentación no tienen estado y sólo contienen código para mostrar información, y para hacer el código más simple dichos componentes se crean con una función en vez de una clase. Además reciben de su componente padre las propiedades necesarias.

También es posible escribir estos mismos componentes de forma funcional, es decir, en vez de que sean definidos con la palabra class, serán creados como simples funciones que regresan el contenido que se mostrará:

```
// Componentes de presentación de forma funcional
const Titulo = ({ nombre } = props) => (
    <h1>{nombre}</h1>);
const Item = (props) => (
    <li><a href='#'>{props.valor}</a></li>);
const Input = (props) => (
    <input type='text' placeholder={props.placeholder} />);
```

Usando esta sintaxis las propiedades se reciben como parámetros de la función e incluso es posible aplicar destructuring para obtener las variables que nos interesan por separado.

Características de los componentes contenedores

- Orientados al funcionamiento de la aplicación
- Contienen componentes de presentación y/o otros contenedores
- Se comunican con las fuentes de datos
- Usualmente tienen estado para representar el cambio en los datos

```
// Contenedor
class AppContainer extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            temas: ['JavaScript', 'React JS', 'Componentes']
        };
    }
    render() {
        const items = this.state.temas.map(t => (<Item valor={t} />));
        return (
            <div>
                <Titulo nombre='List Items' />
                <ul>{items}</ul>
                <Titulo nombre='Inputs' />
                <div>
                    <Input placeholder='Nombre' /><br />
                    <Input placeholder='Apellido' />
                </div>
            </div>
        );
    }
}
```

Por otra parte el componente contenedor define los datos contenidos en la aplicación y también los manipula, después crea los componentes hijos y los muestra en el método render.

No hay una diferencia sintáctica entre ambos componentes, la misma es 100% conceptual

JSX

JSX es una extensión de JavaScript creada por Facebook para el uso con su librería React. Sirve de preprocesador (como Sass o Stylus a CSS) y transforma el código a JavaScript.

Te puede parecer que estás mezclando código HTML dentro de tus ficheros JavaScript, pero nada más lejos de la realidad. A continuación te lo explico. JSX puede recordarte a un lenguaje de plantillas, pero viene con todo el poder de JavaScript.

React al basar el desarrollo de apps en componentes, necesitamos crear elementos HTML que definen nuestro componente, por ejemplo , <p>, <img>.

¿Por qué usar JSX?

React acepta el hecho de que la lógica de renderizado está intrínsecamente unida a la lógica de la interfaz de usuario: cómo se manejan los eventos, cómo cambia el estado con el tiempo y cómo se preparan los datos para su visualización.

En lugar de separar artificialmente tecnologías poniendo el maquetado y la lógica en archivos separados, React separa intereses con unidades ligeramente acopladas llamadas “componentes” que contienen ambas.

React no requiere usar JSX, pero la mayoría de la gente lo encuentra útil como ayuda visual cuando trabajan con interfaz de usuario dentro del código Javascript. Esto también permite que React muestre mensajes de error o advertencia más útiles.

Puedes utilizar comillas para especificar strings literales como atributos:

```
const element = <div tabIndex="0"></div>;
```

También puedes usar llaves para insertar una expresión JavaScript en un atributo:

```
const element = <img src={user.avatarUrl}></img>;
```

No pongas comillas rodeando llaves cuando insertes una expresión JavaScript en un atributo. Debes utilizar comillas (para los valores de los strings) o llaves (para las expresiones), pero no ambas en el mismo atributo

Dado que JSX es más cercano a JavaScript que a HTML, React DOM usa la convención de nomenclatura camelCase en vez de nombres de atributos HTML.

Por ejemplo, class se vuelve className en JSX, y tabindex se vuelve tabIndex.

Ejemplo

Crear un componente sin utilizar JSX sería algo como esto

```
var image = React.createElement('img', {
    src: 'react-icon.png',
    className: 'icon-image'
});
var container = React.createElement('div', {
    className: 'icon-container'
}, image);
var icon = React.createElement('Icon', {
    className: 'avatarContainer'
}, container);
ReactDOM.render(
    icon,
    document.getElementById('app'));
```

Obteniendo el siguiente resultado:

```
<div class='icon-container'>
    <img src='icon-react.png' class='icon-image' />
</div>
```

```
.icon-image {
    width: 100px
}
.icon-container {
    background-color: #222;
    width: 100px
}
```

```
var Icon = (
    <div className='icon-container'>
    <img
    src='Icon-react.png'
    className='icon-image'
    />
    </div>)
ReactDOM.render (Icon, document.getElementById('app'))
```

Ver más en https://es.reactjs.org/docs/jsx-in-depth.html

Componentes con clases

Son componentes declarados como clases. Al declarar estos componentes los mismos deben heredar de la clase component

```
import React, { Component } from 'react';
class Contacto extends Component {
    constructor(props) {
        super(props)
        console.log(this.props)
    }
    render() {
        let prueba = "dsadas"
        return (
            <div className="App">
                {this.props.data.map(d => <div>{d}</div>)}
                Contacto
            </div>);
    }
}
export default Contacto;
```

Render

Todo componente de clase en React, tiene un método Render que es el que se encarga de renderizar en el navegador el HTML correspondiente al componente.

Este método se llama automáticamente cuando se crea un componente y cuando el estado del componente se actualiza.

En este método es donde usamos JSX para facilitar el desarrollo y creación de elementos HTML. Veamos un ejemplo:

```
import React from 'react'
class MyComponent extends React.Component {
    constructor() {
        super()
    }
    render() {
        return (
            <div>
                <span>Hola!, soy un componente</span>
            </div>
        )
    }
}
```

React DOM compara el elemento y sus hijos con el elemento anterior, y solo aplica las actualizaciones del DOM que son necesarias para que el DOM esté en el estado deseado.

Componentes con funciones

Son componentes de react declarados como funciones, por defecto el componente app viene declarado como un componente funcional

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import Home from './Home'
import Contacto from './Contacto'
import Menu from './Menu'
function App() {
    return (
        <div className="App">
            Hola Mundo
        </div>);
}
export default App;
```

En el caso de las funciones las mismas no disponen de un método o función para el renderizado. Se renderiza lo que se encuentre declarado dentro del return

Estructura de directorio

Si observamos el código de la aplicación creada veremos lo siguiente en el archivo **index.js:**

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';
ReactDOM.render(<App />, document.getelementById('root'));
registerServiceWorker();
```

Acá podemos observar que estamos renderizando el componente <App /> sobre el elemento con id root del DOM. ¿Que tiene el componente <App />?

Para responder a esta pregunta debemos ir al archivo **App.js**

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';
class App extends Component {
    render() {
        var app = <div className="App">
            <header className="App-header">
                <img src={logo} className="App-logo" alt="logo" />
                <h1 className="App-title">Welcome to React</h1>
            </header>
            <p className="App-intro">
                To get started, edit <code>src/App.js</code> and save to
reload.
            </p>
        </div>;
        return (app);
    }
}
export default App;
```

Acá podemos observar como el componente App renderiza una imagen de un párrafo.

También podemos ver como el src de la imagen lo obtiene utilizando el **import** de javascript y no definiendo su ruta absoluta y/o relativa en el código.

Volviendo al archivo **index.js**, ¿en donde está el elemento del DOM con id **root**?

Veamos el archivo **index.html**

```
<html><body>
    <noscript>
        You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
    <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.
    You can add webfonts, meta tags, or analytics to this file.
    The build step will place the bundled scripts into the <body> tag.
    To begin the development, run 'npm start' or 'yarn start'.
    To create a production bundle, use 'npm run build' or "yarn build'
    -->
</body>
</html>
```

Este será el html que se renderiza en nuestro navegador. Luego utilizando react se renderizada el contenido del componente **App** sobre el elemento con id **root**

En caso de querer modificar los metas de nuestra página debemos hacerlo sobre el archivo **index.html**

Bibliografía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt.

https://reactjs.org/tutorial/tutorial.html https://reactjs.org/docs/hello-world.html https://carlosvillu.com/introduccion-a-reactjs/ https://platzi.com/blog/react-js-de-javascript/ http://code.ezakto.com/react/introduccion-a-react.html https://es.reactjs.org/docs/jsx-in-depth.html

3. React JS Componentes y Virtual DOM

Bloques temáticos:

- Map
- Trabajando con nuestra aplicación
- API REST
- Promise
- Aplicando una consulta API a nuestro código
- Aplicando axios en lugar de fetch

3.1. React JS Componentes y Virtual DOM

Map

El método map() crea un nuevo array con los resultados de la llamada a la función indicada aplicados a cada uno de sus elementos. Por ejemplo:

```
{this.props.cervezas.map(cerveza=><li>{cerveza}</li>)}
```

Cervezas es un array enviado como propiedad desde el llamado al componente:

```
class App extends Component {
    render() {
        return (
            <div className="App">
                <h1>Bienvenido a Red Social UTN FRBA</h1>
                <Productos cervezas={['Quilmes', 'Brahma']} />
            </div>
        );
    }
}
```

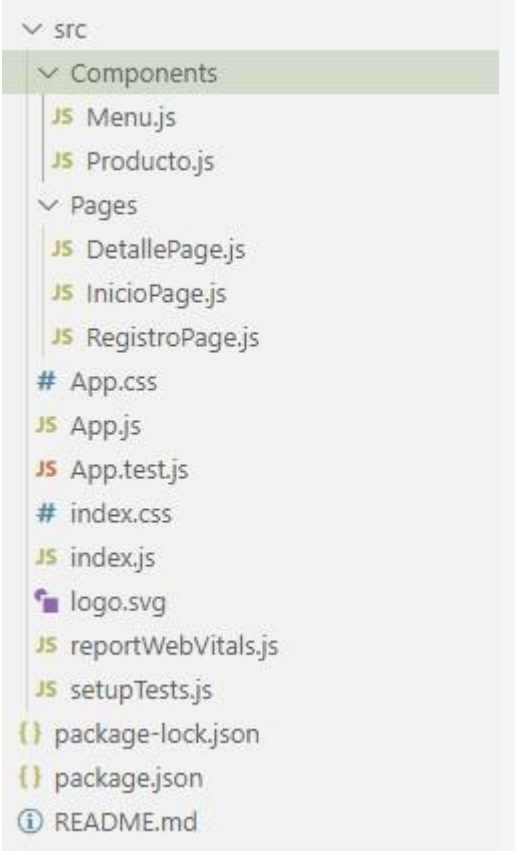
Podemos ver que el primer parámetro recibido por la función dentro de map hace referencia a cada objeto devuelto por la iteración del array.

Trabajando con nuestra aplicación

Para mejorar la organización de nuestro código, podemos crear dos directorios:

- Pages En este directorio ubicaremos aquellos componentes que sean una página completa. Por ejemplo, el componente Home
- Components En este directorio colocaremos todo componente que sea parte de una página, pero no sea una página completa. Por Ejemplo, el componente Menú

Nos queda de la siguiente manera:

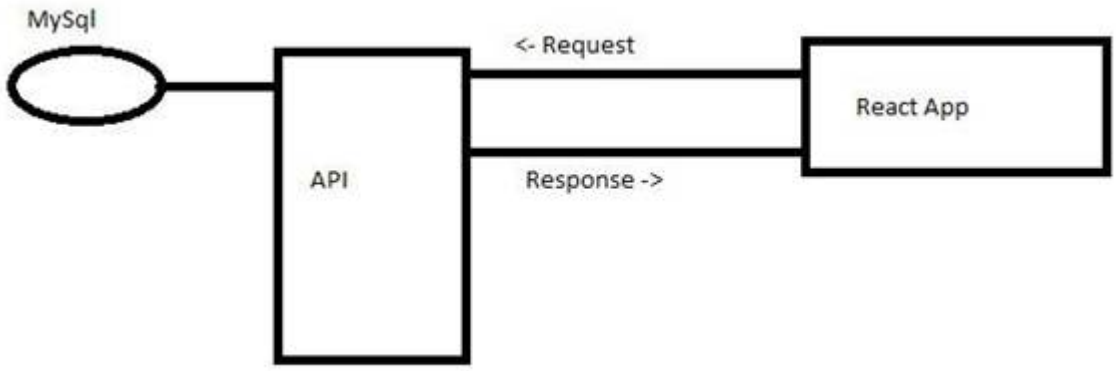


Es una estructura propuesta, cada alumno puede utilizar la estructura deseada.

API REST

REST es cualquier interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON.

Es una alternativa en auge a otros protocolos estándar de intercambio de datos como SOAP (Simple Object Access Protocol), que disponen de una gran capacidad, pero también mucha complejidad. **A veces es preferible una solución más sencilla de manipulación de datos como REST.**



Ventajas

- **Separación entre el cliente y el servidor:** el protocolo REST separa totalmente la interfaz de usuario del servidor y el almacenamiento de datos. Eso tiene algunas ventajas cuando se hacen desarrollos.
Por ejemplo, mejora la portabilidad de la interfaz a otro tipo de plataformas, aumenta la escalabilidad de los proyectos y permite que los distintos componentes de los desarrollos se puedan evolucionar de forma independiente.

- **Visibilidad, fiabilidad y escalabilidad.** La separación entre cliente y servidor tiene una ventaja evidente y es que cualquier equipo de desarrollo puede escalar el producto sin excesivos problemas.
Se puede migrar a otros servidores o realizar todo tipo de cambios en la base de datos, siempre y cuando los datos de cada una de las peticiones se envíen de forma correcta.
Esta separación facilita tener en servidores distintos el front y el back y eso convierte a las aplicaciones en productos más flexibles a la hora de trabajar.

- **La API REST siempre es independiente del tipo de plataformas o lenguajes**: la API REST siempre se adapta al tipo de sintaxis o plataformas con las que se estén trabajando, lo que ofrece una gran libertad a la hora de cambiar o probar nuevos entornos dentro del desarrollo.
Con una API REST se pueden tener servidores PHP, Java, Python o Node.js. Lo único que es indispensable es que las respuestas a las peticiones se hagan siempre en el lenguaje de intercambio de información usado, normalmente XML o JSON.

-

Promise

Una Promise (promesa en castellano) es un objeto que representa la terminación o el fracaso eventual de una operación asíncrona. Una promesa puede ser creada usando su constructor. Sin embargo, la mayoría de la gente son consumidores de promesas ya creadas devueltas desde funciones.

Esencialmente, una promesa es un objeto devuelto al cual enganchas las funciones callback, en vez de pasar funciones callback a una función.

Por ejemplo, en vez de una función del viejo estilo que espera dos funciones callback, y llama a una de ellas en caso de terminación o fallo:

```
function exitoCallback(resultado) {
    console.log("Tuvo éxito con " + resultado);
}
function falloCallback(error) {
    console.log("Falló con " + error);
}
hazAlgo(exitoCallback, falloCallback);
```

Las funciones modernas devuelven una promesa a la que puedes enganchar tus funciones de retorno

```
hazAlgo().then(exitoCallback, falloCallback);
```

A diferencia de las funciones callback pasadas al viejo estilo, una promesa viene con algunas garantías:

- Las funciones callback nunca serán llamadas antes de la terminación de la ejecución actual del bucle de eventos de JavaScript.
- Las funciones callback añadidas con .then serán llamadas después del éxito o fracaso de la operación asíncrona, como arriba.
- Pueden ser añadidas múltiples funciones callback llamando a .then varias veces, para ser ejecutadas independientemente en el orden de inserción.
- Pero el beneficio más inmediato de las promesas es el encadenamiento.

```
hazAlgo().then(function (resultado) {
    return hazAlgoMas(resultado);
})
    .then(function (nuevoResultado) {
        return hazLaTerceraCosa(nuevoResultado);
    })
    .then(function (resultadoFinal) {
        console.log('Obtenido el resultado final: ' + resultadoFinal);
    })
    .catch(falloCallback);
```

Aplicando una consulta API a nuestro código

Modificaremos nuestro código anterior para consultar los datos de nuestros usuarios a una API REST

Componente función

En el componente Home creamos los siguientes hooks:

```
const [loading, setLoading] = useState(true)
const [productos, setProductos] = usestate([])
```

Luego aplicamos el hook useEffect para la ejecución de la consulta al servicio al renderizarse el componente

```
useEffect(
    () => {
        fetch("https://jsonfy.com/items")
        .then(res => res.json())
        .then(data => {
            console.log("data", data)
            setLoading(false)
            setProductos(data)
        })
    },
    []
)
```

Agregamos condiciones para mostrar el listado de usuarios solo en el caso de que se haya completado la respuesta desde el api. En caso contrario mostramos el loader o bien un error.

```
if (loading) {
    return (
        <div> loading...
        </div>
    )
} else {
    return (
        <div>
            <h1>Productos</h1>
            <button onCLick={() => setDestacados(true)}> Ver
Destacados</button>
            <button onclick={() => setDestacados(false)}>Ocultar
Destacados</button>
            {productos.map(producto => <Producto datos={producto}
destacados={true} />)}
        </div >
    )
}
```

Este es el resultado obtenido:

MSI B450-A PRO MAX
91.79
Ver Detalle
HP ProBook 455R G6 9VX50ES R3-3200U 8GB/256GB SSD 15
559
Ver Detalle
Microsoft Office 2019 Home & Business
239.99
Ver Detalle

Componente de clase

En el componente Home realizaremos lo siguiente:

```
constructor(props) {
    super(props)
    this.state = {
        error: null,
        isLoaded: false,
        perfiles: []
    };
}
```

Definimos en el estado del componente las variables error, isLoaded, perfiles

Luego modificamos el ciclo de vida utilizado anteriormente por componentDidMount

```
componentDidMount(){
    fetch("https://jsonplaceholder.typicode.com/users")
        .then(res => res.json())
        .then(
            (result) => {
                console.log(result)
                this.setState({
                    isLoaded: true,
                    perfiles: result
                });
            },
            // Note: it's important to handle errors here
            // instead of a catch() block so that we don't swallow
            // exceptions from actual bugs in components.
            (error) => {
                console.log(error)
                this.setState({
                    isLoaded: true,
                    error
                });
            })
}
```

Dentro de este método incluimos el llamado a la API REST, como vemos debemos trabajar la respuesta con el promise (then)

```
render() {
    const { error, isLoaded, perfiles } = this.state;
    if (error) {
        return <div>Error: {error.message}</div>;
    } else if (!isLoaded) {
        return <div>Loading...</div>;
    } else {
        return (
            <ul>
                {perfiles.map(
                    perfil => <Perfil datos={perfil} />)
                }
            </ul>
        );
    }
}
```

En el render agregamos condiciones para mostrar el listado de usuarios solo en el caso de que se haya completado la respuesta desde la api.

En caso contrario mostramos el loader o bien un error. Este es el resultado obtenido:

Aplicando axios en lugar de fetch

Es una librería JavaScript que puede ejecutarse en el navegador y que nos permite hacer sencillas las operaciones como cliente HTTP, por lo que podremos configurar y realizar solicitudes a un servidor y recibiremos respuestas fáciles de procesar.

Axios es una alternativa que nos brinda multitud de ventajas:

- La API es unificada para las solicitudes Ajax.
- Está optimizado para facilitar el consumo de servicios web, API REST y que devuelvan datos JSON.
- De fácil utilización y como complemento perfecto para las páginas convencionales.
- Pesa poco, apenas 13KB minimizado. Menos aún si se envía comprimido al servidor.
- Compatibilidad con todos los navegadores en sus versiones actuales.

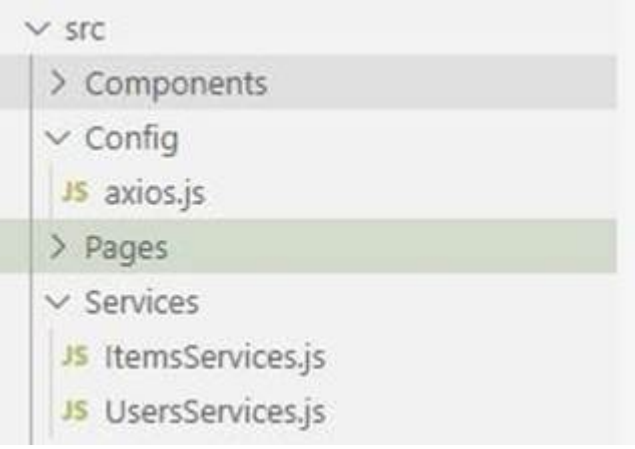
Instalar axios

Debemos ingresar por consola al directorio raíz de nuestro proyecto y ejecutar: **npm install axios**

```
C:\sites\react\pwa2c>pm install axios
```

Una vez instalado podemos consumir directamente sus métodos, a continuación, veremos cómo hacerlo creando un directorio y concepto de services.

Vamos a crear un directorio config y dentro un archivo llamado axios.js

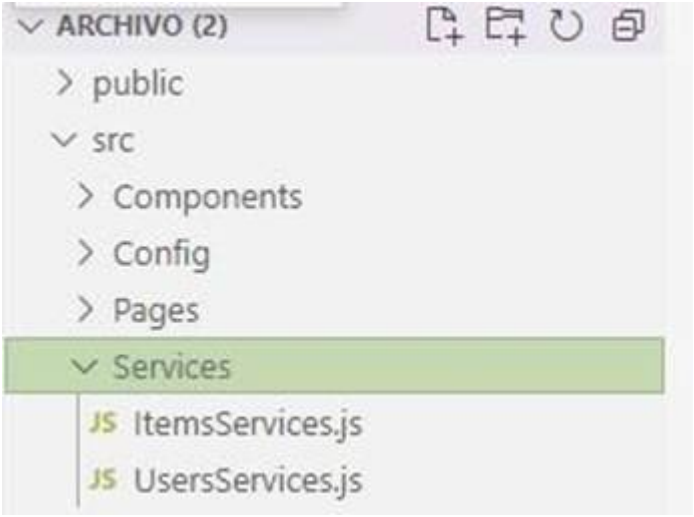


Dentro de dicho archivo vamos a realizar un requiere de la biblioteca y crear una instancia en axios la cual vamos a exportar:

```
import axios from "axios"
export default axios.create({
    baseURL: "https://jsonfy.com/"
})
```

Vemos que créate recibe un parámetro el cual nos permite definir la url base

Por otro lado, dentro de src creamos un directorio llamado “services” y dentro de dicho directorio crearemos un archivo js por cada “entidad”. Ejemplo si consultamos ítems creamos ítems.js. Si la entidad es compras, compras.js



ItemServices.js tendrá el siguiente contenido:

```
import instance from "../Config/axios"
export function getAll(query = "") {
    return instance.get("items" + query)
}
export function getById(id) {
    return instance.get("items/" + id)
}
```

En este caso lo que estamos haciendo es un import del archivo axios.js, para poder importar la instancia creada.

Por otro lado, creamos dos métodos, uno que nos permite consultar todos los ítems retornados por jsonfy y el otro un ítem dado su id.

Por último, en el componente en el cual queramos usar alguno de estos métodos hacemos un import de la función a utilizar y el llamado al método. Por ejemplo:

```
import { getAll } from "../Services/ItemsServices"
//Definicion de clase
function InicioPage() {
    const [loading, setloading] = useState(true)
    const [productos, setProductos] = useState([])
    const [destacados, setDestacados] = useState(false)
    console.log("Database", firebase.db)
    useEffect(
        () => {
            getAll()
                .then(({ data }) => {
                    console.log("data", data)
                    setLoading(false)
                    setProductos(data)
                })
        }, [])
}
```

Axios nos permite tener algunas funcionalidades extra respecto de fetch, por ejemplo, la definición de la url base o el uso de interceptors. No es necesaria su utilización, ya que con fetch podemos realizar una app react completa.

Para ver más acerca de axios: https://github.com/axios/axios

Bibliografía y Webgrafía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt.

https://reactjs.org/docs/faq-ajax.html https://github.com/axios/axios

4. React JS Eventos

Bloques temáticos:

- Eventos con react
- Cómo declarar los eventos
- Eventos disponibles en react
- Cómo recibir parámetros en los eventos
- Ejemplo aplicado a nuestro código
- Ejemplo aplicado a formularios
- React hook forms

4.1. React JS Eventos

Eventos con React

React tiene sus propios eventos, que cuentan con la misma interfaz de los eventos nativos del navegador, con la ventaja que los eventos de React tienen un comportamiento compatible con la mayoría de los navegadores.

Estos eventos reciben una función, o manejadores de eventos. Lo que hacen estas funciones es definir el comportamiento de la aplicación si se corre X o Y evento.

Algo muy común en React, es declarar los manejadores de eventos como funciones dentro de una clase. Cuando hacemos esto, debemos tener cuidado, ya que this no se conecta por defecto a la clase. Para resolver esto, tenemos tres opciones: hacer la conexión en el render(), en el constructor(), o usar arrow functions.

Cómo declarar los eventos

En **Javascript** la declaración de eventos la realizamos de la siguiente manera:

```
<div onclick="handleClick () ">click me</div>
function handleClick() {
    alert('clicked');
    return false:
}
```

Como vemos utilizamos el onclick, en el cual hacemos referencia a una función declarada.

En **React** lo haremos de la siguiente manera:

```
<div onClick={handleClick}>click me</div>
function handleClick(event) {
    alert('clicked');
    event.preventDefault();
    event.stopPropagation();
}
```

Como vemos también utilizamos el evento onClick, pero a diferencia de lo realizado en Javascript solo declaramos el nombre de la función.

La función declarada sólo recibe el argumento evento, sobre el cual podremos realizar la acción de stop propagation (detener propagación o comportamiento por defecto.

Consideraciones:

- El nombre del evento tiene que ser **camelCase** y no minúscula sostenida.
- Al evento se le pasa la función y no una cadena de texto.
- En react si quieres prevenir un comportamiento por defecto o la propagación de un evento debes hacerlo explícitamente llamando los métodos preventDefault() y stopPropagation() respectivamente.

Eventos sintéticos

En este caso event es un evento sintético de React, en React todos los manejadores de eventos son instancias de SyntheticEvents. Los eventos sintéticos son una envoltura de los eventos nativos del navegador, por lo que estos eventos cuentan con la misma interfaz de los eventos nativos, como por ejemplo preventDefault() y stopPropagation(), con la ventaja de que todos estos eventos funcionan idénticamente en la mayoría de los navegadores.

Eventos disponibles en react

React soporta los siguientes eventos:

- **Mouse**: onClick, onContextMenu, onDoubleClick, onMouseDown onMouseEnter onMouseLeave onMouseMove onMouseOut onMouseOver onMouseUp,
- **Drag & Drop**: onDrag, onDragEnd, onDragEnter, onDragExit, onDragLeave, onDragOver, onDragStart, onDrop.
- **Focos y formularios**: onFocus, onBlur, onChange, onInput, onSubmit.
- **Touchs**: onTouchCancel, onTouchEnd, onTouchMove, onTouchStart.
- **Cortar y pegar**: onCopy, onCut, onPaste.
- **Scrolls**: onScroll, onWheel.

Cómo recibir parámetros en los eventos

Si queremos pasar un parámetro a la función llamada en la ejecución de un evento en React debemos hacerlo utilizando la función arrow de ES6. Por ejemplo:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
```

Aquí vemos que cuando se ejecuta el onClick, se llama a la función arrow que tiene como parámetro a **e.** Luego esta función llama en su ejecución a la función deleteRow, la cual debe estar declarada dentro del componente.

Ejemplo aplicado a nuestro código

Aplicaremos la utilización de eventos en nuestro componente **Login**, quedando de la siguiente manera:

```
class Login extends Component {
    ingresar(e) {
        alert("entre");
    }
    render() {
        return (
            <div>
                <form>
                    <div>
                        <label>Usuario</label>
                        <input type="text" placeholder="Introduzca su
usuario" />
                    </div>
                    <div>
                        <label>Contraseña</label>
                        <input type="text" placeholder="Introduzca su
contraseña" />
                    </div>
                    <button onclick={this.ingresar}>Ingresar</button>
                </form>
            </div>
        )
    }
}
```

Como vemos en el onClick debemos utilizar el this si el método lo tenemos declarado a nivel clase.

Ahora si declaramos la función ingresar dentro del render, lo hacemos sin el this**.** Quedando de la siguiente manera:

```
render() {
    ingresar(e){
        alert("entre");
    }
    return (
        <div>
            <form>
                <div>
                    <label>Usuario</label>
                    <input type="text" placeholder="Introduzca su
usuario" />
                </div>
                <div>
                    <label>Contraseña</label>
                    <input type="text" placeholder="Introduzca su
contraseña" />
                </div>
                <button onClick={ingresar}> Ingresar</button>
            </form>
        </div>
    )
}
```

En el caso de las funciones, el llamado se realiza sin this y la misma debe estar declara en el componente:

JSX

```
<button onClick={handleClick}>Registrarse</button>
```

Función declarada en el componente

```
const handleClick = (event) => {
    console.log("handleSubmit", form)
    event.preventDefault()
}
```

Ejemplo aplicado a formularios

En el siguiente ejemplo veremos cómo aplicar el uso de eventos a un formulario en un componente tipo función. Vamos a declarar un hook para almacenar los datos introducidos en el formulario:

```
const [form, setform] = useState({ nombre: '', apellido: '', email: '',
password: '' })
```

Por otro lado vamos a declarar una función que se ejecutará con el evento **submit** del formulario:

```
const handleSubmit = (event) => {
    console.log("handleSubmit", form)
    event.preventDefault()
}
```

El console.log nos retornará los datos que el usuario introdujo en el formulario.

Vemos el formulario (jsx) desarrollado:

```
return (
    <div> <form onSubmit={handleSubmit}>
        <div> <label>Nombre</label>
            <input type="text" name="nombre" value={form.nombre}
onChange={handleChange}></input>
        </div>
        <div> <label>Apellido</label>
            <input type="text" name="apellido" value={form.apellido}
onChange={handleChange}></input>
        </div>
        <div>
            <label>Email</label>
            <input type="email" name="email" value={form.email}
onChange={handleChange}></input>
        </div>
        <div> <label>Contraseña</label>
            <input type="password" name="password" value={form.password}
onChange={handleChange}></input>
        </div>
        <button type="submit"> Registrarse</button>
    </form>
    </div >
)
```

En React el flujo de datos es unidireccional, por lo cual debemos hacer una doble relación de los datos. En este caso asociamos el **value del input** a **la propiedad del hook form** correspondiente.

Ejemplo el **input nombre** queda asociado a form.nombre, esto implica que cuando se modifique dicha propiedad en el hook se modificara el **value en el input**.

Por otro lado capturamos el evento onchange y llamamos a la función handleChange, en la misma realizamos la modificación del hook form de la siguiente manera:

```
const handleChange = (event) => {
    const name = event.target.name
    const value = event.target.value
    console.log("hand leChange", name, value)
    setForm({ ... form, [name]: value })
}
```

Para que esto funcione el nombre de cada propiedad del hook form tiene que llamarse igual que el atributo name del elemento, mediante esta función identificamos el elemento que está siendo modificado y actualizamos el valor del hook.

El componente completo queda de la siguiente manera:

```
import React, { useState } from "react"
function RegistroPage() {
    const [form, setform] = useState({ nombre: '', apellido: '', email: '',
password: '' })
    const handleSubmit = (event) => {
        console.log("handleSubmit", form)
        event.preventDefault()
    }
    const handleChange = (event) => {
        const name = event.target.name
        const value = event.target.value
        console.log("handleChange", name, value)
        setForm({ ...form, [name]: value })
    }
    return (
```

```
        <div> <form onSubmit={handleSubmit}>
            <div> <label>Nombre</label>
                <input type="text" name="nombre" value={form.nombre}
onChange={handleChange}></input>
            </div>
            <div> <label>Apellido</label>
                <input type="text" name="apellido" value={form.apellido}
onChange={handleChange}></input>
            </div>
            <div>
                <label>Email</label>
                <input type="email" name="email" value={form.email}
onChange={handleChange}></input>
            </div>
            <div> <label>Contraseña</label>
                <input type="password" name="password" value={form.password}
onChange={handleChange}></input>
            </div>
            <button type="submit"> Registrarse</button>
        </form>
        </div >
    )
}
export default RegistroPage;
```

React hook forms

Es una biblioteca para React, en la cual nos permite realizar construcción y validaciones de formularios de manera sencilla. https://react-hook-form.com/

En el siguiente video se puede observar cómo aplicar la misma:

Bibliografía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt. https://medium.com/@shmesa23/eventos-en-react-b53179ec9683 https://www.mmfilesi.com/blog/react-4-eventos-y-estados/ https://reactjs.org/docs/handling-events.html http://blog.codeando.org/articulos/manejadores-de-eventos-con-react.html https://react-hook-form.com/ https://es.reactjs.org/docs/handling-events.html

5. Administración del estado en React JS

Bloques temáticos:

- ¿Por qué utilizar un administrador de estado?
- React Context
- Objeto Context
- Provider
- Consumer
- Actualización del estado mediante Context

5.1. Administración del estado en React JS

¿Por qué utilizar un administrador de estado?

En una aplicación típica de React, los datos se pasan de arriba hacia abajo (de padre a hijo) a través de *props*, pero esto puede ser complicado para ciertos tipos de *props* (por ejemplo, localización, el tema de la interfaz) que son necesarios para muchos componentes dentro de una aplicación.

Además, a medida que nuestra aplicación vaya creciendo tendremos complejas cadenas de propiedades siendo pasadas de componente a componente.

La idea de utilizar un administrador de estado es solucionar este tipo de problemas, no hay una única solución ya que tenemos distintas librerías (react context, redux, mobx, etc.) para lograrlo y dependiendo la alternativa que elijamos cada una tendrá algunos conceptos particulares y el enfoque puede variar, pero algo que todas van a mantener el principio de única fuente de verdad (SSOT – Single Source Of Truth).

React Context

Context provee una forma de pasar datos a través del árbol de componentes sin tener que pasar *props* manualmente en cada nivel.

Entonces vamos a utilizar context para compartir datos que pueden considerarse “globales” sin la necesidad de pasar *props* a través de elementos intermedios.

Debemos tener en cuenta aplicarlo con moderación porque hace que la reutilización de los componentes sea más difícil.

Consiste en 3 bloques que veremos en detalle:

- Objeto Context
- Provider
- Consumer

Objeto Context

El objeto Context al fin y al cabo es un objeto donde podremos almacenar cualquier tipo de datos, los cuales serán compartidos a otros componentes.

Se puede crear de la siguiente forma para luego proveerlo y consumirlo desde otros componentes.

```
import React from "react";
const myContext = React.createContext({
    /* Valor inicial*/
});
export default myContext;
```

Provider

Una vez creado el contexto podemos proveer a todos los componentes que necesitaran interactuar con él, es decir, que deberíamos proveerlo en un componente que envuelva todos los niveles inferiores que eventualmente necesiten acceder al contexto.

Un componente proveedor puede estar conectado a muchos consumidores.

```
import React, { Component } from "react";
import myContext from "./Context"
class App extends Component {
    /* Inicializamos un estado*/
    state = {
        usuarios: []
    };
    componentDidMount() {
        /*Guardamos un array de usuarios en el estado*/
        this.setState({ usuarios: ["user1", "user2", "user3"] });
    }
    render() {
        return (
            <myContext.Provider value={{
                state: this.state.usuarios
            }}>
                {/* Componentes de niveles inferiores */}
            </myContext.Provider>);
    }
}
export default App;
```

En el código de ejemplo, se puede ver la creación de un componente App, la inicialización de un estado para ese componente y la modificación del estado en el método componentDidMount del ciclo de vida del componente.

Dentro del renderizado de nuestro componente vamos a envolver todo con el componente proveedor, en este caso myContext.Provider. La propiedad value definirá los valores que enviaremos a los componentes hijos, y si el valor cambia, también cambiará en los mismos.

Consumer

El consumidor será un componente que se suscriba a los cambios en el contexto. De esta forma si el contexto cambia, el componente será renderizado nuevamente.

Es importante saber que para poder consumir el contexto no es necesario que el proveedor sea el padre directo.

Podemos realizarlo de 2 formas:

- Utilizando el componente myContext.Consumer

Con este componente vamos a envolver nuestro componente consumidor para poder utilizar el contexto provisto.

```
import React, { Component } from "react";
import myContext from "./Context";
class ChildComponent extends Component {
    render() {
        return (
            <myContext.Consumer>
                {context => (
                    <div>
                        {/* Renderiza algo basado en el contexto */}
                        {context.usuarios}
                    </div>
                )}
            </myContext.Consumer>
        );
    }
}
export default ChildComponent;
```

Este componente necesita como hijo una función, la cual recibe el valor del contexto actual y devuelve un nodo de React. Cuando nos referimos al valor del contexto actual será igual al props value del componente proveedor.

- Utilizando static contextType

También podremos acceder al contexto asignando una propiedad estática en nuestra clase. A diferencia del anterior, que puede utilizarse en componentes funcionales o basados en clase, este solo puede utilizarse en componentes basados en clase.

```
import React, { Component } from "react";
import myContext from "./Context";
class ChildComponent extends Component {
    static contextType = myContext;
    render() {
        return (
            <div>
                {/* Renderiza algo basado en el contexto */}
                {this.context.usuarios}
            </div>);
    }
}

export default ChildComponent;
```

La ventaja de utilizar esta forma es que podremos utilizar el contexto en cualquier lugar de nuestro componente, por ejemplo, en el método componentDidMount.

Actualización del estado mediante Context

De esta forma tenemos un contexto y lo proveemos a componentes hijos que lo muestran. Pero, ¿qué pasa si queremos modificar esa información? ¿Cómo actualizamos el contexto?

Recordemos que en el Provider teníamos un props value, además de enviar nuestro estado, podríamos enviar métodos para modificarlo.

De esta forma nuestro Consume no solo recibiría el estado, sino también los métodos para modificarlo.

```
import React, { Component } from "react";
import myContext from "./Context";
class App extends Component {
    /*Inicializamos un estado*/
    state = {
        usuarios: []
    };
    componentDidMount() {
        /* Guardamos un arroy de usuarios en el estado */
        this.setState({ usuarios: ["user1", "user2", "user3"] });
    }
    agregarUsuario = () => {
        this.setState({ usuarios: this.state.usuarios.concat(["user4"])
});
    };
    render() {
        return (
            <myContext.Provider
                value={{
                    state: this.state.usuarios,
                    agregarUsuario: this.agregarUsuario
                }}
            > {/* Componentes de niveles inferiores */}
            </myContext.Provider>);
    }
}
export default App;
```

En el ejemplo, creamos un método para agregar un usuario a nuestro estado. Y luego lo enviamos como referencia dentro de props value de nuestro Provider. De esta forma el Consumer podrá acceder al método.

```
import React, { Component } from "react";
import myContext from "./Context".
class ChildComponent extends Component {
    static contextType = myContext;
    render() {
        return (
            <div>
                {/* Renderiza algo basado en el contexto */}
                {this.context.usuarios}
                {/* Renderiza un boton para agregar un usuario */}
                <button
onClick={this.context.agregarUsuario.bind(this)}>
                    Agregar Usuario
                </button>
            </div >);
    }
}
export default ChildComponent;
```

De esta forma podríamos, por ejemplo, crear un botón para que al hacer click, agregue un usuario a nuestro estado.

Bibliografía utilizada y sugerida

https://es.reactjs.org/docs/context.html

https://www.academind.com/learn/react/redux-vs-context-api/

6. ReactJS Manejo de rutas

Bloques temáticos:

- React router
- Configurar ruteo en app.js
- Configurar enlaces de dirección
- Recibir parámetros por URL
- useHistory
- NavLink
- Redirect
- Not Found

6.1. ReactJS Manejo de rutas

React router

El componente router no es un módulo oficial de React. Nos permitirá hacer una web navegable, es decir poder redirigir entre distintas páginas.

Instalación

Debemos ubicarnos dentro del directorio en el cual está nuestra aplicación y ejecutar:

```
npm install react-router-dom
```

Configurar ruteo en app.js

En el archivo **app.js** debemos realizar lo siguiente:

```
import HomePage from ". /Pages/HomePage"
import RegistroPage from './Pages/RegistroPage';
import Menu from ' ./Components/Menu';
import {
    BrowserRouter as Router,
    Switch,
    Route
} from "react-router-dom";
function App() {
    return (
        <div className="App">
            <Router>
                <Menu />
                <Switch>
                    <Route path="/registro" >
                        <RegistroPage />
                    </Route>
                    <Route path="/">
                        <HomePage />
                    </Route>
                </Switch>
            </Router>
        </div >
    );
}
export default App;
```

Como vemos debemos incluir los módulos **Route, Router (BrowserRouter) y Switch**

Si ejecutamos nuestra aplicación ingresara al contenido del componente App y si en la barra de direcciones colocamos / ingresara al contenido del componente HomePage

Si en la barra de direcciones colocamos /registro entonces se visualizará el contenido del componente RegistroPage

Vemos que el elemento route recibe 1 propiedad path (indica la url por la que se ingresara a ese componente).

Configurar enlaces de dirección

Cuando queramos armar los enlaces (lo que sería <a href) lo debemos hacer usando el componente Link:

```
import React from "react"
import {
    Link
} from "react-router-dom"
function Menu() {
    return (
        <div>
            <ul> <li>
                <Link to="/">Home</Link>
            </li>
                <li>

                    <Link to="/registro">Registro</Link>
                </li>
            </ul>
        </div>
    )
}
export default Menu;
```

En este ejemplo vemos el componente Link aplicado al menú, pero podemos aplicarlo a cualquier componente.

En la propiedad to colocamos la url destino (sería el href del componente Link)

Recibir parámetros por URL

En caso de querer recibir parámetros por URL vamos a indicar en el componente route lo siguiente:

```
<Route path="/detalle/:id" >
    <DetallePage />
</Route>
```

En caso de ser un componente de tipo clase este elemento :id se enviará al componente DetallePage como una propiedad, de la siguiente manera:

```
constructor(props){
    super(props)
    console.log(this.props.params.id)
}
```

En caso de ser un componente tipo función podemos utilizar el hook useParams() de la siguiente manera:

```
import React from "react"
import {
    useParams
} from "react-router-dom";
function DetallePage() {
    const { id } = useParams()
    console.log(id)
    return (
        <div>DetallePage</div>
    )
}
export default DetallePage;
```

En ambos casos se hace referencia al elemento id porque se identificó el parámetro de esa manera en el route (:id)

useHistory

Hook que permite acceder al histórico de navegación. Podemos utilizar para realizar una redirección (push) desde la lógica del componente. Por ejemplo:

```
import React from "react"
import {
    useHistory
} from "react-router-dom";
function RegistroPage() {
    let history = useHistory();
    function handleClick() {
        history.push("/home");
    }
    return (
        <div>
            Registro
            <button type="button" onClick={handleClick}>
                Go home
            </button>
        </div>
    )
}
export default RegistroPage;
```

En este caso al hacer click en **Go To Home** llamamos a la función handleClick y en dicha función hacemos un history.push("/home") el cual nos redirigirá a la página de inicio.

NavLink

Este componente tiene una serie de propiedades utilizadas a la hora de armar un navbar en nuestra aplicación. Ejemplo de utilización en el Menú:

```
function Menu() {
    return (
        <div>
            <NavLink to="/" exact >Home</NavLink>
            <NavLink to="/registro">Registro</NavLink>
        </div>)
}
export default Menu;
```

Se puede ver la totalidad de las propiedades en https://reactrouter.com/web/api/NavLink

Redirect

Permite realizar redirecciones en la lógica de ruteo. Lo aplicamos en el Switch en el cual se realiza la comparación de las direcciones:

```
<Redirect from=" /home" to="/" />
<Route path="/">
<HomePage />
</Route>
<Route path="/" >
<HomePage />
</Route>
```

En el ejemplo cuando recibamos /home redirigirá a /

https://reactrouter.com/web/api/Redirect

Not Found

Para hacer la lógica de ruteo de nuestra página 404 Not Found debemos declarar la siguiente regla:

```
<Router>
    <Menu />
    <Switch>
        <Route path="/" exact >
            <HomePage />
        </Route>
        <Route path="/registro" exact >
            <RegistroPage />
        </Route>
        <Route path="/detalle/:id" exact >
            <DetallePage />
        </Route>
        <Redirect from="/home" to="/" exact />
        <Route path="*">
            <NotFoundPage />
        </Route>
    </Switch>
</Router>
```

De esta manera estaremos accediendo a cualquier url no identificada en reglas anteriores.

Bibliografía y Webgrafía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt.

https://medium.com/@pshrmn/a-simple-react-router-v4-tutorial-7f23ff27adf
https://reactrouter.com/web/guides/quick-start