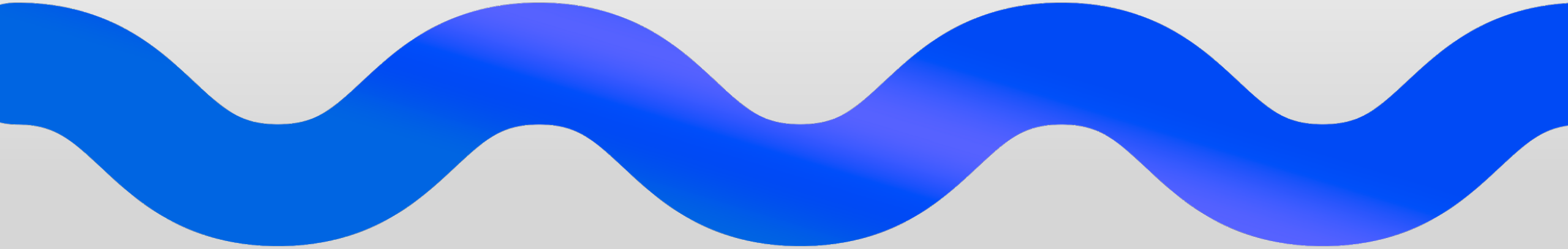
Lagash™

Introducción a React



1. ¿Cómo instalar React?

a.Primero se debe instalar la última versión LTS disponible de node.js (https://nodejs.org/es/)

b. Luego abrir la terminal de nuestro sistema operativo y pararse en la carpeta donde quiero generar el proyecto

c. Una vez en la carpeta correspondiente, correr el siguiente comando en la terminal: ***npx create-react-app nombre-de-nuestra-app.*** Npx permite acceder a los ejecutables almacenados en el registro de npm sin instalar dependencias. Create-react-app es una herramienta que permite levantar proyectos de React sin configurarlos manualmente.

d. Una vez instalado nos situamos dentro de la carpeta y corremos el siguiente comando para inicializar la app: ***npm start***

2. Estructura

Al crear un proyecto de react, se crean los siguientes archivos dentro de la carpeta **src** donde vamos a estar trabajando.

- **App.js**: Es un componente, el cual se renderiza en la página.
- **App.css**: hoja de estilo
- **App.test.js**: donde se testean los componentes, que veremos más adelante
- **index.js**: Aquí se carga y se importa todo lo principal de react es donde se renderiza el componente principal de la aplicación

3.1 Componentes

Los componentes son piezas que nos permiten dividir nuestra UI y que podremos reutilizar para construir nuestra app.

Los componentes pueden ser funciones o clases, pueden recibir parámetros (llamados props) desde otros componentes, y siempre deben retornar un elemento jsx.

```jsx
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Para crear un componente debemos:

1. Importar React al comienzo del archivo.
2. Definir el componente ya sea una función o una clase.
3. Retornar jsx.
4. exportar el componente al final (Se pueden exportar varios componentes por archivo pero es una buena practica crear uno nuevo para cada uno).

En la imagen podemos ver como esta compuesto el componente App que se crea automáticamente cuando generamos el proyecto.

3.2 JSX

Es una sintaxis que nos permite combinar etiquetas HTML con código Javascript. El compilador usa esta sintaxis para generar “elementos” (que simplemente son objetos) que luego React utilizara para construir el DOM a travez de un proceso llamado “render”.

```
return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
```

En el mismo componente mostrado anteriormente podemos ver el jsx y como se definen las clases y atributos de cada elemento y como también se pueden asignar valores de variables como en el caso de la img.

3.3 Props

Las props son valores que recibe un componente hijo de un componente padre. En el componente hijo las props se reciben por parámetro en forma de objeto y siempre son valores de solo lectura. Se pueden pasar por props tanto valores como funciones.

```
import React from "react";
import Component from "./Component";

function App() {
  return (
    <Component
      value="Hello World!"
      clicked={() => {
        console.log("click");
      }}
    />
  );
}

export default App;
```

```
import React from "react";

// El objeto props se puede destructurar para hacer mas claro el c
ódigo
function Component({ value, clicked }) {
  return (
    <>
      <h1>{value}</h1>
      <button onClick={clicked}>Click me!</button>
    </>
  );
};

export default Component;
```

3.4 Componente principal

En el archivo index.js podemos ver como React genera el DOM que veremos en el navegador a partir del componente App, por lo que es buena practica usarlo como principal y sobre el se irán anidando otros componentes.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can chan
ge
// unregister() to register() below. Note this comes with some pitfal
ls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

En App podemos ir agregando todos los componentes que importamos arriba y componen la pantalla de nuestra aplicación, de esta forma se abstrae la lógica de cada parte a un componente por separado y luego se unifica todo. Como vemos, todos los componentes pueden ser padres de otros componentes

```
import React from 'react';
import Navbar from './components/Navbar'; // Ejemplo
import Container from './components/Container'; // Ejemplo
import Title from './components/Title'; // Ejemplo
import Footer from './components/Footer'; // Ejemplo
import './App.css';

function App() {
  return (
    <div className="App">
      <Navbar />
      <Container>
        <Title />
      </Container>
      <Footer />
    </div>
  );
}

export default App;
```

Ejercicios

Ejercicio 1:
Crear una app que imprima el saludo “Hello World!”.

Click aquí para ver las resoluciones de los ejercicios.
(cada ejercicio está resuelto en un branch del repositorio)

Ejercicios

Ejercicio 2:
Modificar el ejercicio anterior para que el saludo “Hello World!” lo haga desde un componente llamado “Hello”.

3.5 Class Components

Como dijimos antes, los componentes se pueden declarar como funciones o clases, con estas ultimas podemos tener acceso a la api de React.Component que nos permite crear un state y nos provee métodos para controlar la performance del componente.

```
import React from "react";

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {};
  }

  render() {
    return <h1>Hello World</h1>;
  }
}

export default App;
```

El componente funciona igual que los vistos antes, solo que se implementa la api Component, las props se toman en un constructor y el jsx que devolvemos primero tiene que estar dentro de la función render.

3.6 State

El state es un objeto que podemos definir en el constructor de la clase, cada vez que cambie una de sus propiedades se va a ejecutar nuevamente la función render y se actualizara la pantalla con los nuevos cambios.

Para cambiar los valores del state se usa el método setState, que recibe como parámetro un objeto con las propiedades que queremos cambiar.

```
constructor(props) {
    super(props);

    this.state = {
      text: "Counter",
      counter: 0
    };
  }

  handleClick = () => {
    this.setState({
      counter: this.state.counter + 1
    });
  };

  render() {
    return (
      <>
        <h1> {this.state.text + ": " + this.state.counter} </h1>
        <button onClick={this.handleClick}> + 1 </button>
      </>
    );
  }
```

3.7 Functional components

Los functional components son componentes declarados como funciones, al no tener state ni poder acceder a la api Component, generalmente se usan solo para recibir datos por props y presentar información a través de jsx.

Para crearlos solo hay que declararlos como functions o arrow functions.

```jsx
import React from "react";

const Post = ({ title, subtitle, content }) => {
  return (
    <div>
      <h1>{title}</h1>
      <h3>{subtitle}</h3>
      <p>{content}</p>
    </div>
  );
};

export default Post;
```

Ejercicios

Ejercicio 3:
Modificar el ejercicio anterior para que el componente reciba por props el nombre a imprimir. Ejemplo: “Hello Juan!” (Juan es la variable recibida por props).

Ejercicios

Ejercicio 4:
Modificar el ejercicio anterior para que la variable “name” sea un estado y pueda modificarlo desde un input.

4. Hooks

Son funciones que se usan para agregar mayor funcionalidad a los functional components.

Podemos crear states locales y globales, o definir funciones que se ejecuten ante ciertos cambios. Son muy útiles cuando queremos agregar, modificar o dividir lógica de negocio, es mas simple implementar hooks en componentes que tener que convertirlos en clases.

¿Los hooks reemplazan a los class components? No, ambos pueden existir juntos y tenemos la libertad de decidir cuando conviene usar cada uno.

4.1 useState

```
import React, { useState } from "react";

const Post = () => {
  const [text, setText] = useState("Hola mundo");

  return (
    <div>
      <h1>{text}</h1>
      <input
        type="text"
        onChange={event => setText(event.target.value)}
      ></input>
    </div>
  );
};

export default Post;
```

Este hook nos permite crear sencillamente un state y una función que se encargara de actualizarlo.

Primero importamos el hook useState.

Cuando llamamos la función le podemos pasar opcionalmente como parametro un valor inicial y esta nos retornara un array con dos valores, el primero es una referencia al valor del state, el segundo es la función que nos permite actualizar el valor del state.

4.2 useEffect

```jsx
import React, { useState, useEffect } from "react";

const Post = () => {
  const [text, setText] = useState("");
  const [count, setCount] = useState(0);

  useEffect(() => {
    setText(`count ${count}`);
  }, [count]);

  return (
    <div>
      <h1>{text}</h1>
      <button onClick={() => setCount(count + 1)}>Agregar +1</button>
    </div>
  );
};

export default Post;
```

useEffect nos permite en el primer parámetro definir una función que se ejecutara cada vez que un valor en el segundo parámetro cambie. Se suele llamar a estas funciones como “side effects”.

Si en el segundo parámetro se pasa un array sin valores el side effect se ejecutara una sola vez luego de que el componente se renderice por primera vez.

Si no se pasa ningún valor como segundo parámetro el side effect se ejecutara cada vez que el componente se renderice nuevamente por algún cambio en un state.

4.3 useLayoutEffect

```jsx
import React, { useState, useLayoutEffect } from "react";

const Post = () => {
  const [text, setText] = useState("");
  const [count, setCount] = useState(0);

  useLayoutEffect(() => {
    setText(`count ${count}`);
  }, [count]);

  return (
    <div>
      <h1>{text}</h1>
      <button onClick={() => setCount(count + 1)}>Agregar +1</button>
    </div>
  );
};

export default Post;
```

Es muy similar al useEffect, la diferencia es que useEffect primero corre el side effect luego de que se renderice el componente y se asegure que el efecto no bloquee el browser.

UseLayoutEffect lo hace antes de que el browser muestre los cambios.

4.4 useRef

```
import React, { useState, useRef } from "react";

const Post = () => {
  const text = useRef(null);
  const counterRef = useRef(0);
  const [flag, setFlag] = useState(false);

  const handleClick = () => {
    counterRef.current++;
    text.current.innerText = "counterRef:" + counterRef.current;
  };

  return (
    <div>
      {console.count("render")}
      <h1 ref={text}>counterRef: 0</h1>
      <h3>flag value: {flag ? "true" : "false"}</h3>
      <button onClick={() => handleClick()}>Agregar texto</button>
      <button onClick={() => setFlag(true)}>Cambiar flag</button>
    </div>
  );
};

export default Post;
```

Esta función nos retorna una referencia mutable que puede mantener su valor a pesar de los re renders ocasionados por cambios en el state. Por ejemplo podemos asignar esa referencia a un elemento html y modificarlo sin tener que hacer un nuevo render, o podemos asignarle un valor que no queremos perder.

La ref es simplemente un objeto al que podemos acceder a su valor a travez de la propiedad current.

5. Context

```jsx
import React, { createContext, useContext } from "react";

const stateContext = createContext();

const TextProvider = ({ children }) => {
  const state = {
    text: "Hola mundo"
  };

  return (
    <stateContext.Provider value={state}>{children}</stateContext.Provider>
  );
};

const Text = () => {
  const { text } = useContext(stateContext);

  return <p>{text}</p>;
};

class App extends React.Component {
  render() {
    return (
      <TextProvider>
        <Text />
      </TextProvider>
    );
  }
}

export default App;
```

Para no tener que pasar siempre variables a través de props, React ofrece una forma de trasmitir datos entre componentes llamada Context.

Primero se declara un context con createContext(), luego dentro de un componente definimos las variables que queremos compartir y en el return dejamos que nuestro context provea a nuestros componentes hijos a través de un value al cual le pasaremos las variables elegidas.

En los componentes que queremos consumir el context usamos la función useContext y nos retornara el value que pasamos en el componente padre.

6. Promises

```jsx
import React, { useState } from "react";

const App = () => {
  const [text, setText] = useState("");

  const textPromise = new Promise((resolve, reject) => {
    setTimeout(() => {
      if (true) resolve("Hola ya termine");
      else reject("No pude terminar");
    }, 2000);
  });

  const handleClick = () => {
    textPromise
      .then(value => {
        setText(value);
      })
      .catch(err => {
        setText(err);
      });
  };
  return (
    <div>
      <h3>{text}</h3>
      <button onClick={handleClick}>Traer texto</button>
    </div>
  );
};

export default App;
```

Una promesa es un valor que no se conoce en este momento, pero puede conocerse en el futuro. Se puede crear una promise para representar un valor asincrónico o consumir una promise para usar el resultado de una operación asincrónica.

En la creación de una promise esta se encuentra en un estado “pending”, hay que definir un una funcion que eventualmente resuelva con un resolve() el valor o sino devuelva un error con reject().

Para consumir una promise se utiliza el método then() que espera que se resuelva el valor, cuando eso pasa se ejecuta la función con el valor recibido como argumento. Si la promise llega a devolver un error este se puede manejar con el método catch().

7. Fetch y renderizar listas

```
import React, { useState, useEffect } from "react";

const Pokemon = ({ name }) => {
  return <li>{name}</li>;
};

const App = () => {
  const [pokemonList, setPokemonList] = useState([]);

  useEffect(() => {
    fetch("https://pokeapi.co/api/v2/type/3/")
      .then(res => res.json())
      .then(json => setPokemonList(json.pokemon))
      .catch(err => console.log(err));
  }, []);

  return (
    <div>
      {pokemonList.map(pokemon => (
        <Pokemon key={pokemon.pokemon.url} name={pokemon.pokemon.name}
/>
      ))}
    </div>
  );
};

export default App;
```

Jsx nos permite ingresar variables o funciones que devuelvan valores entre etiquetas html, para eso debemos ingresar el código dentro de llaves.

Por ejemplo podemos hacer un fetch que nos traiga una lista, esa lista la loopeamos con map que nos devuelve un nuevo array de componentes. React automáticamente se encarga de renderizar ese resultado como una lista.

Ejercicios

Ejercicio 5:

Crear una app para gestionar una lista de compras.

Tips:

- crear un componente que muestre los items
- crear un componente que me permita agregar nuevos items

Ejercicios

Ejercicio 6:
Modificar el ejercicio anterior para adaptarlo a un Todo-List
(debo poder eliminar tareas de la lista y seleccionar las que ya completé)

8. React-router-dom

React-router-dom nos permite renderizar condicionalmente ciertos componentes dependiendo de la ruta usada en la url. Esto nos permite crear aplicaciones con varias paginas de manera sencilla.

Lo primero que hay que hacer es, como esta librería no es parte de react, correr el siguiente comando:

npm install react-router-dom

8.1 Usando react router

```
import React from "react";
import { BrowserRouter, Switch, Route, Link } from "react-router-dom";

const User = () => <h1>User page</h1>;
const About = () => <h1>About page</h1>;
const Home = () => (
  <ul>
    <li>
      <Link to="/user">User</Link>
    </li>
    <li>
      <Link to="/about">About</Link>
    </li>
  </ul>
);

const App = () => {
  return (
    <BrowserRouter>
      <Switch>
        <Route path="/" component={Home} exact />
        <Route path="/user" component={User} />
        <Route path="/about" component={About} />
      </Switch>
    </BrowserRouter>
  );
};

export default App;
```

Una vez instalada la librería debemos importar los componentes que vamos a usar.

- BrowserRouter se encarga de toda la lógica de mostrar los componentes que se le proveen, por eso debe envolver nuestra App.
- Switch se encarga de asegurarse que se renderizara un solo componente a la vez.
- Route serán los links para mostrar nuestros componentes. Se le debe pasar un atributo ***path*** con la dirección y otro atributo ***component*** que define el componente que queramos usar. El atributo ***exact*** se usa cuando hay varios paths similiares, de esta forma se asegura que Home se renderice solo cuando la ruta es “/” y no cuando es ”/user”.
- Link nos permite crear links clickeables para navegar por nuestra app. A este componente le debemos pasar un atributo ***to*** con la dirección a la que queremos dirigir.

Ejercicios

Ejercicio 7:
Integrar el ejercicio anterior con el backend.

Lagash™

Let’s keep in touch

www.lagash.com