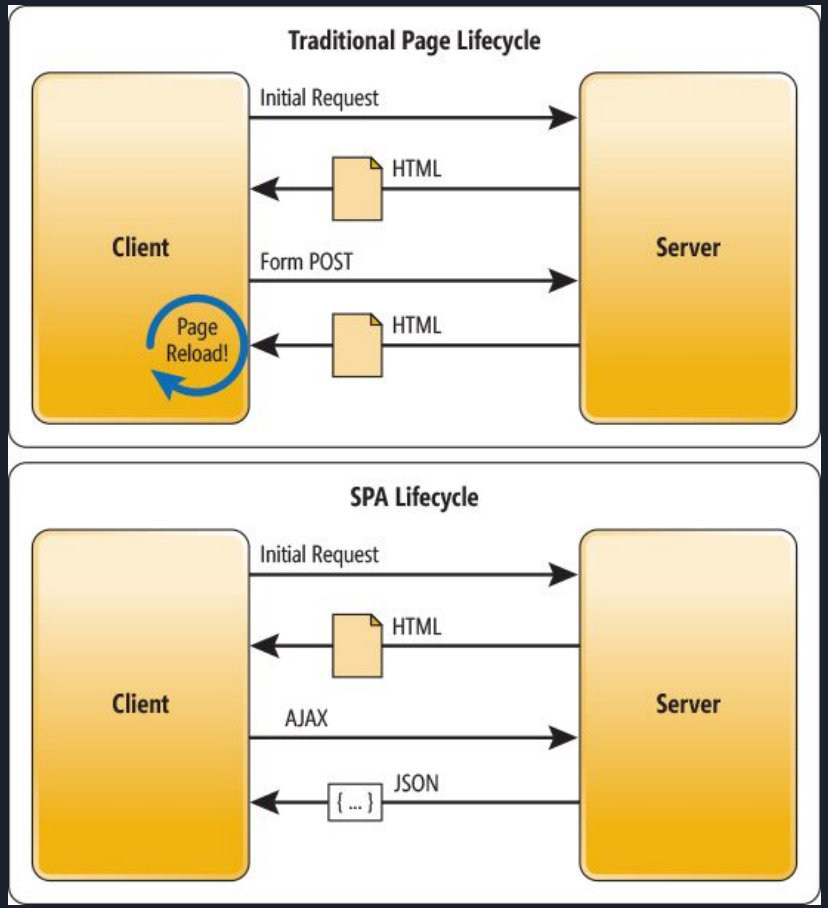
Introducción al FrontEnd

Desarrollo. Web

Single-Page Applications (SPAs) are Web **apps** that load a **single** HTML **page** and dynamically update that **page** as the user interacts with the **app**.

Microservices is a specialisation of an implementation approach for service-oriented architectures (SOA) used to build flexible, independently deployable software systems.

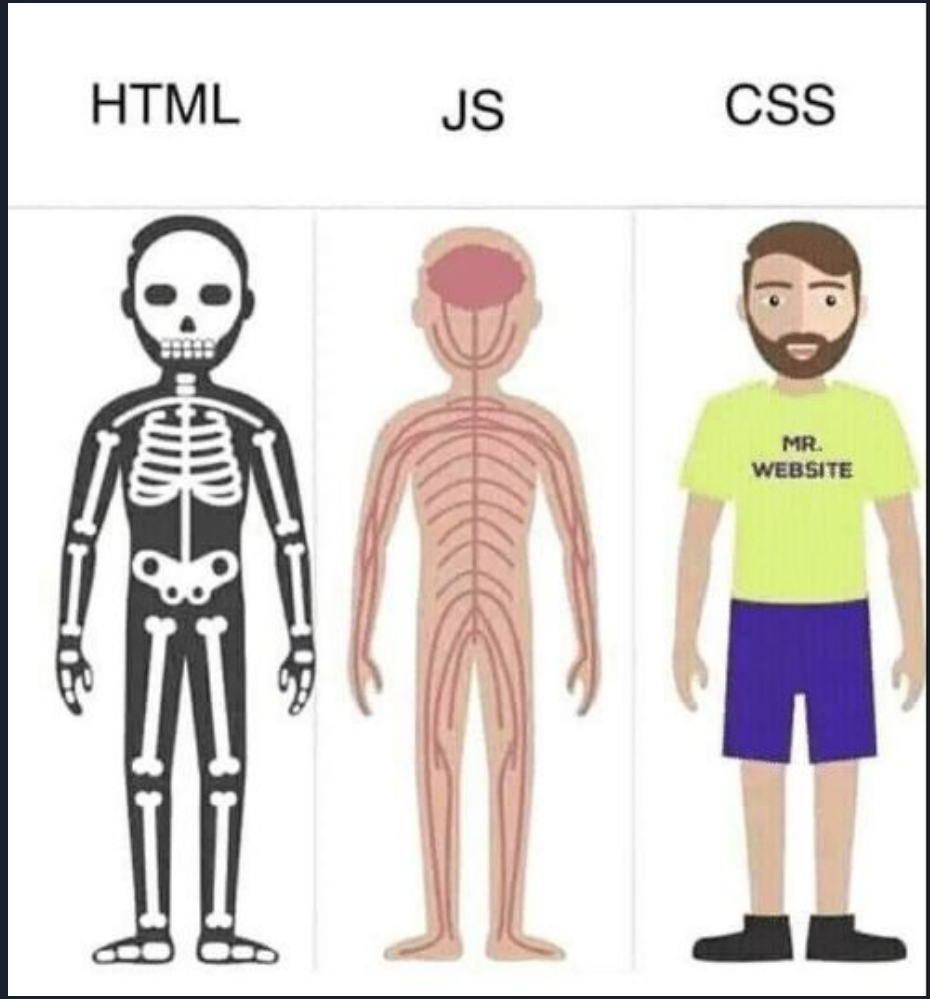
Representational State Transfer (**REST**) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web **service** induce desirable properties, such as performance, scalability, and modifiability, that enable**services** to work best on the Web.



HTML + CSS + JS

index.html > ...

```html
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8" />
        <title>Front End 101</title>
        <!-- Import Styles -->
        <link rel="stylesheet" href="./assets/styles.css" />
    </header>
    <body>
        <div id="app" />
    </body>
</html>
<script>
    console.log("Follow the white rabbit!");
    const app = document.getElementById("app");
    console.log(app);

    const inputText = document.createElement("input");
    inputText.type = "text";

    const button = document.createElement("button");
    button.textContent = "Enviar";
    // button.onclick = () => alert("Hola");
    button.onclick = () => {
        fetch("https://api.github.com/users/" + inputText.value)
            .then((response) => response.json())
            .then((content) => {
                console.log(content);
                const userInfo = `<div><h3>${content.login}</h3><img src="${content.avatar_url}"/></div>`;
                const result = document.createElement("div");
                result.innerHTML = userInfo;
                app.appendChild(result);
            });
    };

    app.appendChild(inputText);
    app.appendChild(button);
</script>
```



Instalación

Desde scripts:

```
<!-- ... HTML existente ... -->

<div id="like_button_container"></div>

<!-- ... HTML existente ... -->
```

```
  <!-- ... más HTML ... -->

  <!-- Cargar React. -->
  <!-- Nota: cuando se despliegue, reemplazar "development.js" con "production.min.js". -->
  <script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>

  <!-- Cargamos nuestro componente de React. -->
  <script src="like_button.js"></script>

</body>
```

Instalación

Desde scripts:

```
<body>
    <div id="app">
        <!-- Contenedor para los elementos de React -->
    </div>
```

```
ReactDOM.render(
    // Creamos un div con con varios myName en su interior
    // Donde a cada myName le enviamos un objeto
    // Estos objetos son los props
    React.createElement(
        "div",
        null,
        React.createElement(myName, { name: "Juanjo" }, null),
        React.createElement(myName, { name: "Laura" }, null),
        React.createElement(myName, { name: "Pablo" }, null)
    ),
    $app
);
```

Instalación

Usando Create-React-App

```
npx create-react-app my-app --template typescript
```

Necesitarás tener Node >= 6 y npm >= 5.2 instalados

https://facebook.github.io/create-react-app/docs/getting-started

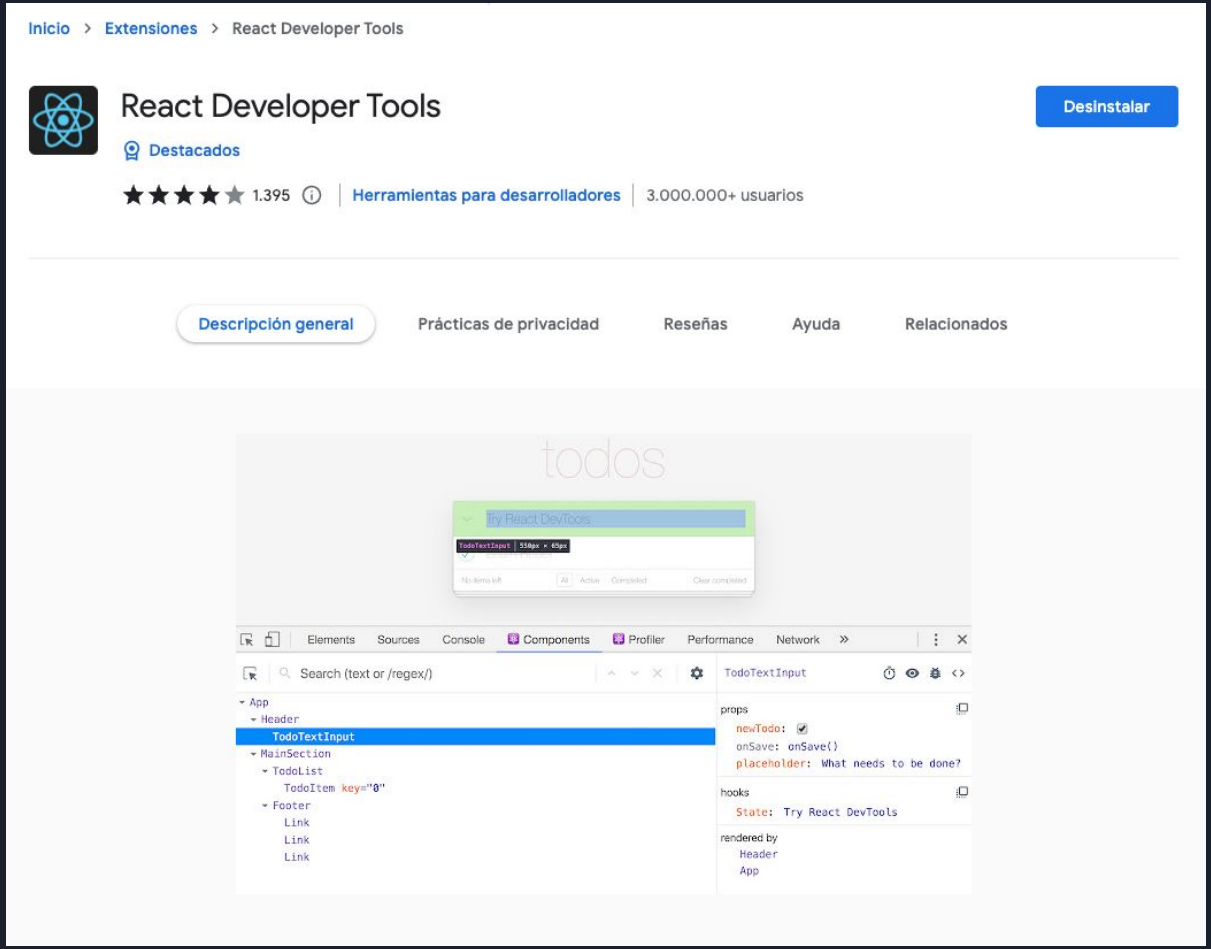
Instalación

Usando Create-React-App

```
npm start
  Starts the development server.

npm run build
  Bundles the app into static files for production.

npm test
  Starts the test runner.

npm run eject
  Removes this tool and copies build dependencies, configuration files
  and scripts into the app directory. If you do this, you can’t go back!
```

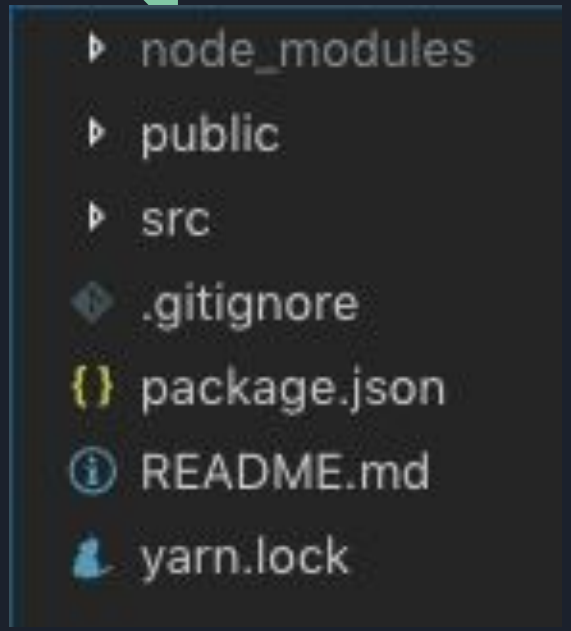
Instalación

React Developer Tools

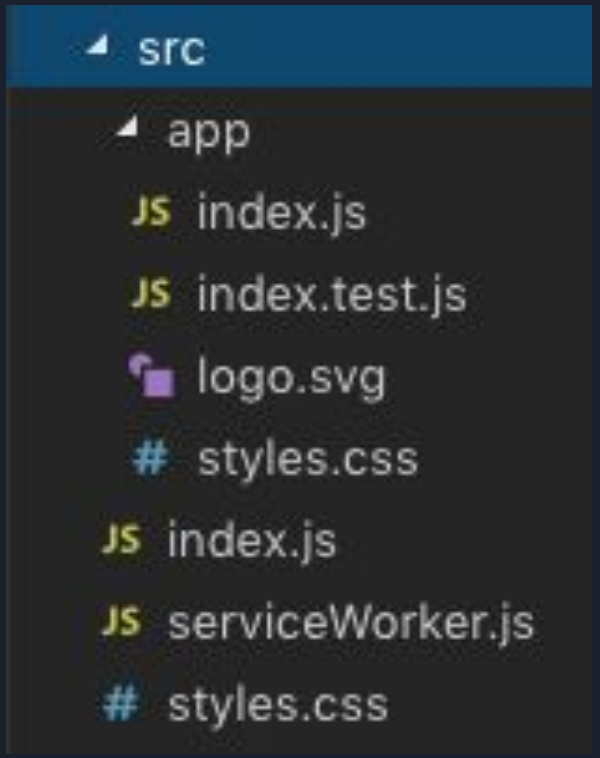


https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=es

Estructura.


node_modules
public
src
.gitignore
package.json
README.md
yarn.lock


public
favicon.ico
index.html
manifest.json


src
app
index.js
index.test.js
logo.svg
styles.css
index.js
serviceWorker.js
styles.css

Instalación. Ejercicio

Crear una Aplicación usando Create-React-App.

Instalar node

npx create-react-app <nombre-app> --template typescript

Trabajar en la app: npm start

Ejecutar test: npm test

Construir la app: npm build

Abrir la app en el navegador

HelloWorld.

```
1  // src/index.js
2  import React from 'react';
3  import ReactDOM from 'react-dom';
4  import './index.css';
5  import App from './app';
6  import * as serviceWorker from './serviceWorker';
7
8  ReactDOM.render(<App />, document.getElementById('root'));
9
10 // If you want your app to work offline and load faster, you can change
11 // unregister() to register() below. Note this comes with some pitfalls.
12 // Learn more about service workers: https://bit.ly/CRA-PWA
13 serviceWorker.unregister();
14
```

HelloWorld.

```
// src/app/index.js
import React, { Component } from 'react';
import './styles.css';

class App extends Component {
    render() {
        return (
            <div className="App">
                Hello World!
            </div>
        );
    }
}

export default App;
```

```
1  // src/index.js
2  import React from 'react';
3  import ReactDOM from 'react-dom';
4  import './index.css';
5  import App from './app';
6  import * as serviceWorker from './serviceWorker';
7
8  ReactDOM.render(<App />, document.getElementById('root'));
9
10 // If you want your app to work offline and load faster, you can change
11 // unregister() to register() below. Note this comes with some pitfalls.
12 // Learn more about service workers: https://bit.ly/CRA-PWA
13 serviceWorker.unregister();
14
```

```
const container = document.getElementById( elementId: 'root');
if (container) {
    const root = createRoot(container);
    // @ts-ignore
    root.render(<App />);
    // If you want to start measuring performance in your app, pass a function
    // to log results (for example: reportWebVitals(console.log))
    // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
    reportWebVitals();
}
```

HelloWorld.

```
export const AuthApp: React.FC<{}> = () => {
    const styles = useStyles();
    const { showLoading, showLogin, showNoService, invalidReason } = useAPIProviderStatus();

    if (showLoading) {
        return (
            <div aria-label='authapp validating' className={styles.container}>
                <div className={styles.dialog}>
                    <CircularProgress size={180} color='inherit' />
                    <div className={styles.message}>
                        <Typography className={styles.messageText}>Checking API KEY, please wait a moment...</Typography>
                    </div>
                </div>
            </div>
        );
    }

    if (showNoService) {
        return <ServiceNotFoundPage />;
    }

    if (showLogin) {
        return <LoginPage invalidReason={invalidReason} />;
    }
    return <App />;
};

export default AuthApp;
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import CssBaseline from '@material-ui/core/CssBaseline';
import { ThemeProvider } from '@material-ui/core/styles';
import { BrowserRouter } from 'react-router-dom';
import { MainMenuProvider } from './services/main-menu-provider';
import { APIProvider } from './services/api-provider';
import * as serviceWorker from './serviceWorker';

import AuthApp from './auth-app';
import theme from './theme';
import './index.css';
ReactDOM.render(
    <React.StrictMode>
        <ThemeProvider theme={theme}>
            <CssBaseline />
            <APIProvider>
                <BrowserRouter>
                    <MainMenuProvider>
                        <AuthApp />
                    </MainMenuProvider>
                </BrowserRouter>
            </APIProvider>
        </ThemeProvider>
    </React.StrictMode>,
    document.getElementById('root')
);

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

HelloWorld. Ejercicio

Ejecutar aplicación: npm start

Modificar el contenido del componente App para que solo muestre vuestro nombre

Ejecutar la App: npm start

Construir aplicación: npm run build

JSX.

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

```
const element = <h1>Hello, world!</h1>;
```

JSX. Expresiones

```
// src/app/index.js
import React, { Component } from 'react';
import './styles.css';

class App extends Component {
    render() {
        const message = <span>Hello World!</span>
        return (
            <div className="App">
                {message}
            </div>
        );
    }
}

export default App;
```

```
const ErrorMessage: React.FC<{ error: ServiceError | undefined }> = ({ error }) => {
    if (!error) return null;
    return (
        <div aria-label='error message'>
            <Typography variant='h3'>{error.message}</Typography>
        </div>
    );
};
```

JSX. Expresiones

```
function formatName(user) {
    return <p>{user.name + ' ' + user.lastname}</p>
}
class App extends Component {
    render() {
        const user = {name: 'Juanjo', lastname: 'Franco'}
        const message = <span>Hello {formatName(user)}</span>
        return (
            <div className="App">
                {message}
            </div>
        );
    }
}
```

JSX. Expresiones

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

JSX. Expresiones

```
export type ELEMENTS = (EventType | Target | Rule)[];
const DeleteList: React.FC<{
    show: boolean;
    elements?: ELEMENTS;
    isLoading: boolean;
}> = ({ show, elements, isLoading }) => {
    const styles = useStyles();
    if (!show) return null;
    isLoading = true;
    return (
        <div aria-label='elements to delete' className={styles.elementList}>
            {elements!.map(e => (
                <div key={e.id} aria-label='element to delete' className={styles.elementItem}>
                    <div>
                        <Typography variant='h5'>{e.name}</Typography>
                    </div>
                    <div>
                        <Typography variant='caption' className={styles.captionText}>
                            {e.id}
                        </Typography>
                    </div>
                </div>
            ))}
        </div>
    );
};
```

JSX. Atributos

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

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

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

```jsx
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

```jsx
<div
    className='SensorValues'>
    <div
        className='SensorValue'>
        <div className='SensorValueData'>{sensor.RPMBomba / 10}</div>
        <div className='SensorValueName'>RPM</div>
    </div>
    <div
        className='SensorValue'>
        <div className='SensorValueData'>{sensor.VelocidadLinea}</div>
        <div className='SensorValueName'>Velocidad</div>
    </div>
    <div
        className='SensorValue'>
        <div className='SensorValueData'>{sensor.PresionBomba / 10}</div>
        <div className='SensorValueName'>Presión</div>
    </div>
    <div
        className='SensorValue'>
        <div className='SensorValueData'>{sensor.Contrapresion}</div>
        <div className='SensorValueName'>ContraPresión</div>
    </div>
</div>
```

JSX. Ejercicio

Mostrar en pantalla el nombre y el apellido de una persona, formateado a partir de un objeto JSON.

Mostrar por pantalla una lista de nombres de personas famosas.

Mostrar por pantalla una imagen y su nombre.

Añadir CSS a los elementos creados anteriormente.

Renderizado

React se basa en repintar la pantalla (renderizar) en función de los elementos (JSX) que se devuelven en su método render.

En cada “Repintado” React calcula los cambios que debe realizar en el DOM para actualizar del estado anterior al nuevo estado.

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

Renderizado. Ejercicio

Mostrar en pantalla la hora actual, actualizándose cada segundo.

Mostrar en pantalla un mensaje que contenga un número y un texto que diga si es un número par o impar, el número debe actualizarse cada medio segundo.

Componentes

Los componentes permiten separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada.

Los componentes son como las funciones de JavaScript. Aceptan entradas arbitrarias (llamadas “**props**”) y devuelven a React elementos que describen lo que debe aparecer en la pantalla.

Componentes

Los componentes pueden ser funciones que aceptan un solo parámetro, que es el objeto “prop” y devuelve el JSX describiendo el componente.

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

Componentes

Los componentes pueden ser clases de ES6 que tienen un método render que devuelve el JSX que describe al componente. Reciben el objeto “props” como parte de la instancia

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

Componentes

El objeto “props” es un objeto que contiene todos los atributos que enviamos al componente cuando lo renderizamos.

```
// Componente tipo Clase
class App extends Component {
    render() {
        return (
            <Welcome
                name='Juanjo'/>
        );
    }
}
```

```
// Componente tipo función
function Welcome(props){
    // props contiene {name: 'Juanjo'}
    return (
        <div>Welcome {props.name} to ReactJS!</div>
    );
}
```

```
class Welcome extends Component {
    // this.props contiene {name: 'Juanjo'}
    render() {
        return <div>Welcome {this.props.name} to ReactJS!</div>;
    }
}
```

Componentes

```
return (
    <div className={styles.root}>
        <div aria-label='rule search bar' className={styles.searchBar}>
            <SearchBar hint='Enter a rule name...' minLength={0} onSearchFor={changeFilter} />
        </div>
        <Fab color='primary' aria-label='add rule' className={styles.fabAddRule} onClick={openDialog}>
            <AddIcon />
        </Fab>
```

```
export const SearchBar: React.FC<SearchBarProps> = function SearchBar({
    hint = 'search for...',
    onSearchFor = NOOP,
    delay = 500,
    minLength = 3
}) {
    const styles = useStyles();
    const filter = React.useCallback(value => !!value && value.length >= minLength, [minLength]);
    const [searchText, setSearchText] = useDebounce({
        callback: onSearchFor,
        initialValue: '',
        delay: delay,
        filterDispatch: filter
    });
    const onTextChange = React.useCallback((ev: React.ChangeEvent<HTMLInputElement>) => setSearchText(ev.target.value), [setSearchText]);
    return (
        <Paper component='div' className={styles.searchContainer}>
            <InputBase
                className={styles.searchInput}
                placeholder={hint}
                value={searchText}
                onChange={onTextChange}
                inputProps={{ 'aria-label': 'search input' }}
            />
            <div className={styles.searchButton} aria-label='search icon'>
                <SearchIcon color='primary' />
            </div>
        </Paper>
    );
};

export default SearchBar;
```

Componentes

Los componentes pueden referirse a otros componentes en su salida.
Esto nos permite utilizar la misma abstracción de componente para cualquier nivel de detalle.
Un botón, un cuadro de diálogo, un formulario, una pantalla, etc..
En aplicaciones de React, todo se expresa como componentes.

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

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

Componentes

Aplicamos el patrón “Divide y vencerás”

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

Componentes

Aplicamos el patrón “Divide y vencerás”

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Componentes

Aplicamos el patrón “Divide y vencerás”

```
export const HelpDialog: React.FC<{
    showDialog: boolean;
    closeDialog: () => void;
}> = ({ showDialog, closeDialog }) => {
    const styles = useStyles();
    if (!showDialog) return null;
    return (
        <Dialog
            open={true}
            onClose={closeDialog}
            id='help-dialog'
            scroll='paper'
            aria-labelledby='help-dialog-title'
            aria-describedby='help-dialog-content'
        >
            <DialogTitle aria-label='payload creator help' id='help-dialog-title'>
                Sorry, no default payload!
            </DialogTitle>
            <DialogContent dividers={true} id='help-dialog-content' className={styles.addFieldDialogContent}>
                Sorry, this eventType has not been received any data, yo can copy the url from the EventType and fire your first request,
                then you will be able to download its schema.
            </DialogContent>
            <DialogActions>
                <Button aria-label='payload creator help button close' onClick={closeDialog}>
                    Close
                </Button>
            </DialogActions>
        </Dialog>
    );
};
```

Componentes

Extraer componentes puede parecer un trabajo pesado al principio, pero tener una paleta de componentes reutilizables vale la pena en aplicaciones más grandes.

La regla general es que si una parte de la interfaz de usuario se usa varias veces (Button, Panel, Avatar), o es lo suficientemente compleja por sí misma (App, FeedStory, Comment), debe ser un componente.

Componentes

Todos los nombres de componentes, tanto funciones como clases deben comenzar con mayúscula. De lo contrario React los confundirá con elementos del DOM.

Todos los componentes de React deben actuar como funciones puras con respecto a sus props. (El objeto “prop” no puede modificarse)

Componentes

Podemos pasar otros componentes como “props”, esto se hace en un campo especial llamado “**children**”.

Podemos pasar funciones como props para comunicar los componentes hijos con los padres.

```
class App extends Component {
    hablaAlAbuelo(msgFromChildren) {
        console.log('Mensaje del hijo', msgFromChildren);
    }
    render() {
        return (
            <div className='App'>
                <Welcome
                    name='Juanjo'/>
                <Padre hablaAlAbuelo={this.hablaAlAbuelo}>
                    <div>Soy el Hijo</div>
                </Padre>
            </div>
        );
    }
}
```

```
1  import React from 'react';
2
3  export const Padre = (props) => {
4      props.hablaAlAbuelo('Hola Abuelo');
5      return (
6          <div>
7              <div>Yo soy tu PADRE</div>
8              {props.children}
9          </div>
10     );
11 };
```

Componentes. Ejercicio

Crear un componente Reloj Digital.

El componente debe actualizarse cada segundo.

El componente debe recibir la fecha actual en su objeto “prop”

El componente debe contener al menos dos componentes. El componente Fecha y el componente Hora.

Crear 2 componentes Reloj Digital, cada uno con una zona horaria, debe indicar el nombre de la zona horaria.

Crear un componente que muestre sus componentes hijos dentro de un recuadro

Estado.

Es la forma de actualizar un solo componente. Sin renderizar toda la aplicación como con ReactDOM.render() / createRoot

El estado es similar a las props, pero es privado y está completamente controlado por el componente.

El estado está presente en los componentes de clases ES6 y en los componentes de función usando Hooks.

Se representa como “this.state” en las clases y con “useState” en hooks

Estado.

¿Necesito reescribir todos mis componentes que ya sean clases?

No. No hay planes de eliminar las clases de React — todos debemos seguir lanzando productos y no nos podemos dar el lujo de reescribir. Recomendamos usar Hooks en tu código nuevo.

¿Qué tanto de mi conocimiento de React se mantiene relevante?

Los Hooks son una manera más directa de usar la características de React que ya conoces — como el estado, ciclo de vida, contexto, y las referencias (refs). No cambian de manera fundamental el funcionamiento de React, y tu conocimiento de componentes, props, y el flujo de datos de arriba hacia abajo sigue siendo igual de relevante.

Estado.

Se puede convertir un componente de función en una clase en cinco pasos:

1. Crear una clase ES6 con el mismo nombre que herede de React.Component.
2. Agregar un único método vacío llamado render().
3. Mover el cuerpo de la función al método render().
4. Reemplazar props con this.props en el cuerpo de render().
5. Borrar el resto de la declaración de la función ya vacía.

Estado.

```jsx
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

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

Estado.

El método render se invocará cada vez que ocurre una actualización

Siempre y cuando repintemos el componente en el mismo nodo del DOM

Se usará solo una única instancia de la clase del componente.

Esto nos permite utilizar características adicionales como el estado local y los métodos de ciclo de vida.

Estado.

Para dotar de estado a una clase componente debemos mover su objeto “prop” al objeto “state”.

Añadir un constructor a la clase e inicializar el objeto “state” en dicho constructor.

Eliminar las propiedades de la declaración del componente.

Llamamos a “this.setState()” con el nuevo estado, para generar un repintado

Estado.

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}
```

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Estado.

Añadir ciclos de vida a un componente.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {

  }

  componentWillUnmount() {

  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Estado.

Añadir ciclos de vida a un componente.

El método **componentDidMount()** se ejecuta después que la salida del componente ha sido renderizada en el DOM.

El método **componentWillUnmount()** se ejecuta justo antes de que el componente se elimina del DOM.

El método **componentDidUpdate()** se ejecuta justo después de que el componente haya sido repintado.

Estado.

Prohibido **MUTAR** el estado directamente

```
// Incorrecto
this.state.comment = 'Hello';
```

```
// Correcto
this.setState({comment: 'Hello'});
```

Las actualizaciones del estado pueden ser asíncronas

```
// Incorrecto
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

```
// Correcto
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Estado.

Las actualizaciones del estado se fusiona

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

Estado. Ahora con Hooks!

Los Hooks resuelven una amplia variedad de problemas aparentemente desconectados en React que hemos encontrado durante más de cinco años de escribir y mantener decenas de miles de componentes.

Ya sea que estés aprendiendo React, usándolo diariamente o incluso prefieras una librería diferente con un modelo de componentes similar, es posible que reconozcas algunos de estos problemas.

- Es difícil reutilizar la lógica de estado entre componentes
- Los componentes complejos se vuelven difíciles de entender
- Las clases confunden tanto a las personas como a las máquinas

La gestión del ESTADO se mantiene, cambiamos la sintaxis

useState. this.setState()

```
class ExampleUseState extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0};
    }

    render() {
        return (
            <div>
                <div>{this.count}</div>
                <button onClick={() => this.setState({count: this.state.count + 1})}>Increment</button>
            </div>
        );
    }
}
```

useState. this.setState()

Sirve para mantener un estado local al componente entre repintados y ofrecernos el valor más reciente en cada repintado, además de ofrecernos un mecanismo para actualizar dicho estado.

Permite mantener una variable de estado en un componente función como haríamos con this.state en clases.
const [state, setState] = useState(initialState)

```
const ExampleUseState = () => {
    const [count, setCount] = React.useState(0);
    return (
        <div>
            <div>{count}</div>
            <button onClick={() => setCount(count + 1)}>Increment</button>
        </div>
    );
};
```

useState. this.setState()

Su único argumento es el valor inicial que tendrá su estado, a diferencia de las clases no tiene porqué ser un objeto.

Este argumento puede ser una función, que se ejecutará solo la primera vez.

Este argumento puede ser una función, que se ejecutará solo la primera vez

```
const ExampleUseState = () => {
    const [count, setCount] = React.useState(0);
    return (
        <div>
            <div>{count}</div>
            <button onClick={() => setCount(count + 1)}>Increment</button>
        </div>
    );
};
```

useState. this.setState()

La llamada a la función updater (setState) con el mismo valor no hará un repintado.

Podemos usar tantos useStates dentro de un componente como queramos/necesitemos.

Devuelve una pareja de valores, el valor de la variable y su función actualizadora.

La función actualizadora repintará el componente cada vez que se invoque con un nuevo valor.

```
const ExampleUseState = () => {
    const [count, setCount] = React.useState(0);
    return (
        <div>
            <div>{count}</div>
            <button onClick={() => setCount(count + 1)}>Increment</button>
        </div>
    );
};
```

useState. this.setState()

La función actualizadora puede recibir una función, cuyo argumento es el estado actual.

A diferencia de las clases, si el valor es una clase, esta no se fusiona y la variable de estado tendrá ese nuevo valor durante su repintado.

```
const ExampleUseState1 = () => {
    const initialValue = () => {
        console.log('ExampleUseState', 'Instanciando valor inicial', 5);
        return 5;
    };
    const [state, setState] = React.useState(initialValue);
    console.log('Repintando con cambio de state', state);
    return (
        <div>
            <div>{state}</div>
            <button onClick={() => setState(state => state + 1)}>Increment</button>
        </div>
    );
};
```

useEffect. componentdidMount, componentDidUpdate, componentWillUnmount

```
class ExampleUseEffect extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0};
    }

    componentDidMount() {
        document.title = 'Effect count' + this.count;
    }

    componentDidUpdate() {
        document.title = 'Effect count' + this.count;
    }

    render() {
        return (
            <div>
                UseEffect, Ejemplo 1
                <button onClick={() => this.setState({count: this.state.count + 1})}>Change title</button>
            </div>
        );
    }
}
```

useEffect. componentdidMount, componentDidUpdate, componentWillUnmount

Sirve para ejecutar modificaciones sobre el DOM que no pueden ser realizadas durante el repintado.

Le indica a React que tiene que hacer después de haber repintado el DOM.

React recordará la función que le pasemos y la ejecutará después del repintado del DOM.

```
const ExampleUseEffect = () => {
    const [count, setCount] = React.useState(0);
    React.useEffect(
        () => {
            document.title = 'Effect count' + count;
        }
    );
    return (
        <div>
            UseEffect, Ejemplo 1
            <button onClick={() => setCount(count + 1)}>Change title</button>
        </div>
    );
};
```

useEffect. componentdidMount, componentDidUpdate, componentWillUnmount

Se ejecutará después del primer repintado y tras cada repintado, garantizando que el DOM ya se ha repintado.

A diferencia de cdm y cdu, no bloqueará el repintado del DOM, es un proceso asíncrono.

Es un escape a código imperativo.

React ejecuta los efectos justo después de repintar el DOM.

Los efectos tienen acceso a las props y states del componente.

Los efectos, por defecto, se ejecutan en cada repintado, incluyendo el primero.

Puedes usar tantos efectos como necesites dentro de un componente

Estado.

Realizar peticiones al backend. Fetch

```
componentDidMount() {
    // Simple GET request using fetch
    fetch('https://api.npms.io/v2/search?q=react')
        .then(response => response.json())
        .then(data => this.setState({ totalReactPackages: data.total }));
}
```

```
useEffect(() => {
    // GET request using fetch inside useEffect React hook
    fetch('https://api.npms.io/v2/search?q=react')
        .then(response => response.json())
        .then(data => setTotalReactPackages(data.total));

// empty dependency array means this effect will only run once (like componentDidMount in clas
}, []);
```

Estado.

Realizar peticiones al backend. Fetch

```
async componentDidMount() {
    // GET request using fetch with async/await
    const response = await fetch('https://api.npms.io/v2/search?q=react');
    const data = await response.json();
    this.setState({ totalReactPackages: data.total })
}
```

Estado.

Realizar peticiones al backend. Fetch

```
componentDidMount() {
    // GET request using fetch with error handling
    fetch('https://api.npms.io/v2/invalid-url')
        .then(async response => {
            const data = await response.json();

            // check for error response
            if (!response.ok) {
                // get error message from body or default to response statusText
                const error = (data && data.message) || response.statusText;
                return Promise.reject(error);
            }

            this.setState({ totalReactPackages: data.total })
        })
        .catch(error => {
            this.setState({ errorMessage: error.toString() });
            console.error('There was an error!', error);
        });
}
```

Estado. Ejercicios

Transformar el ejemplo anterior en un componente con estado que gestione su actualización cada segundo.

1. Mover setInterval a componentDidMount
2. Eliminar intervalo en componentWillUnmount

Crear un componente que obtenga mi perfil de GitHub y lo muestre.

1. Hacer la petición en componentDidMount
2. Actualizar el estado al recibir la respuesta de la petición
3. Pintar el perfil de usuario en pantalla

Estado. Ejercicios

Transformar el ejemplo anterior en un componente con estado que gestione su actualización cada segundo usando Hooks

Crear un componente que obtenga mi perfil de GitHub y lo muestre usando Hooks

Eventos.

Manejar eventos en elementos de React es muy similar a manejar eventos con elementos del DOM. Hay algunas diferencias de sintaxis:

1. Los eventos de React se nombran usando camelCase, en vez de minúsculas.
2. Con JSX pasas una función como el manejador del evento, en vez de un string.
3. Los manejadores no pueden devolver false para evitar comportamientos por defecto

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Eventos.

Manejar eventos en elementos de React es muy similar a manejar eventos con elementos del DOM. Hay algunas diferencias de sintaxis

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

Eventos.

Necesita bindearse cuando usamos clases

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // Este enlace es necesario para hacer que `this` funcione en el callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      isToggleOn: !state.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

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

Eventos.

O usar la sintaxis de campos públicos de clases de ES7

```
class LoggingButton extends React.Component {
  // Esta sintaxis nos asegura que `this` está ligado dentro de handleClick
  // Peligro: esto es una sintaxis *experimental*
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

Eventos.

O usar arrow function en la función render para definir los manejadores de eventos

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // Esta sintaxis nos asegura que `this` esta ligado dentro de handleClick
    return (
      <button onClick={(e) => this.handleClick(e)}>
        Click me
      </button>
    );
  }
}
```

Eventos. Ejercicios

1 Crear un componente Cronómetro, que tenga la opción de comenzar, pausar y reiniciar el tiempo.

2 Añadir un campo de texto y un botón al componente GitHub que nos permita buscar un nuevo usuario y muestre su avatar y algo de información.

3 Crear una calculadora con las operaciones básicas, Suma, Resta, Multiplicación y División.

Renderizado II.

En React, podemos crear distintos componentes que encapsulan el comportamiento que necesitamos.
Podemos renderizar solamente algunos de ellos, dependiendo del estado del componente.

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}
```

Renderizado II.

Podemos usar variables para almacenar elementos.
Esto permite renderizar condicionalmente una parte del componente mientras el resto del resultado no cambia

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  let button;
  if (isLoggedIn) {
    button = <LogoutButton onClick={this.handleLogoutClick} />;
  } else {
    button = <LoginButton onClick={this.handleLoginClick} />;
  }
  return (
    <div>
      <Greeting isLoggedIn={isLoggedIn} />
      {button}
    </div>
  );
}
```

Renderizado II.

Podemos embeber cualquier expresión en JSX envolviendo dicha función en llaves. Esto incluye al operador lógico (&&) de JavaScript.

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
```

Renderizado II.

Otro método para el renderizado condicional de elementos en una línea es usar el operador condicional **condición ? true : false** de JavaScript.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

Renderizado II.

También podemos usarlo para expresiones más complejas.
Pero pierde legibilidad!

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn ? (
        <LogoutButton onClick={this.handleLogoutClick} />
      ) : (
        <LoginButton  onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Renderizado II.

Podemos hacer que el componente se oculte así mismo devolviendo **null**

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }
  return (
    <div className="warning">
      Warning!
    </div>
  );
}
```

```
class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }
  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }
  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}
```

Renderizado II. Ejercicios

Añadir un componente <Loading/> al componente <GitHub/> para que se muestre mientras se esté realizando la petición.

Añadir un componente <Warning/> al componente <GitHub/> que se muestre si la petición devuelve un error.

Mostrar un mensaje en el componente <GPS/> si el navegador no soporta la funcionalidad de GPS.

Mostrar un mensaje en el componente <Calculadora/> si se intenta hacer una división por cero.

Listas.

Renderizado de múltiples componentes.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

```
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

Listas.

Renderizado de múltiples componentes.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Listas.

Renderizado de múltiples componentes. Necesitamos indicar el atributo **KEY**

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

Listas.

Cuidado dónde ponemos la KEY.

```jsx
function ListItem(props) {
  const value = props.value;
  return (
    // Mal! No hay necesidad de especificar la key aquí:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Mal! La key debería haber sido especificada aquí:
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

Listas. Ejercicios

Crear un componente <Agenda/> que podamos insertar tareas, y ver el listado de todas las tareas insertadas.

Usando el componente <Almanaque/> crear un componente que reciba un array de zonas horarias y muestre un <RelojDigital/> por cada zona horaria.

Formularios.

Los elementos de formularios en HTML funcionan un poco diferente a otros elementos del DOM en React, debido a que los elementos de formularios conservan algún estado interno y en React nosotros mantenemos el estado en nuestros componentes.

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Formularios.

En HTML, los elementos de formularios como los **<input>**, **<textarea>** y el **<select>** normalmente mantienen sus propios estados y los actualizan de acuerdo a la interacción del usuario.

En React, el estado mutable es mantenido normalmente en la propiedad estado de los componentes, y solo se actualiza con **setState()**.

Un campo de un formulario cuyos valores son controlados por React de esta forma es denominado “componente controlado”.

Formularios.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange=
{this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Formularios.

La etiqueta **<textarea>**

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Formularios.

La etiqueta **<Select>**

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Formularios.

La etiqueta **<Select>**

Puedes pasar un array al atributo value, permitiendo que selecciones múltiples opciones.

```
<select multiple={true} value={['B', 'C']}>
```

Formularios.

Manejando múltiples inputs

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }
```

```
  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Formularios. Ejercicios

Crear un formulario de reservas que gestione el estado de la reserva (aceptada/cancelada), el nombre del titular de la reserva y el número de invitados.

Crear un formulario de Acceso, que solicite el nombre de usuario y su contraseña, debe validar que el nombre de usuario sea un email y que la contraseña tenga una longitud de al menos 6 caracteres.

Conectar el formulario con un API de backend

Sincronización.

Usualmente, muchos componentes necesitan reflejar el mismo cambio en los datos. En React es un patrón habitual.

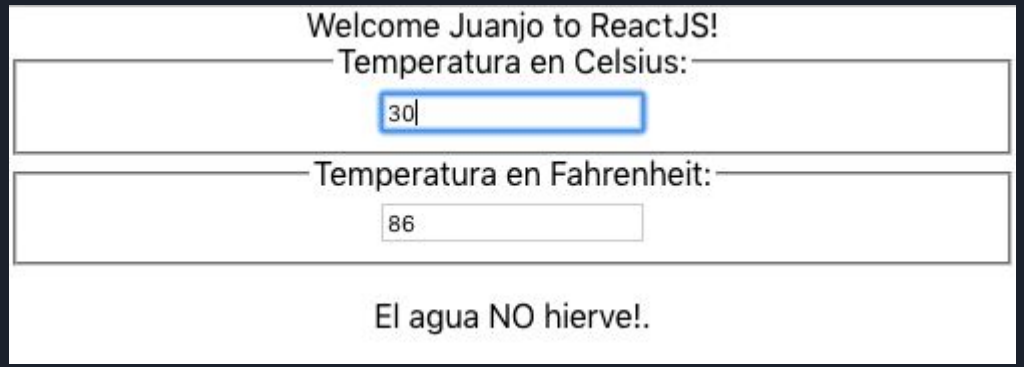
Simplemente necesitamos llevar el estado compartido al componente padre común más cercano.

El componente padre mantiene el estado y envía las propiedades a cada hijo para que las renderize de la manera adecuada.

Podemos pensar en este patrón como las dos caras de la misma moneda

Sincronización. Ejercicio

Crear un formulario de conversión de temperaturas entre grados centígrados y farenheit, que mantiene sincronizadas ambas temperaturas. El formulario debe indicar cuando el agua está hirviendo, según la temperatura introducida.



CRUD. Ejercicio

Crear una web para gestionar una base de datos de usuarios:

- Crear nuevos usuarios
- Listar todos los usuarios
- Ver los datos de un usuario al pinchar en la lista de usuarios
- Modificar un usuario
- Eliminar un usuario