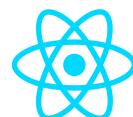


# React

# Índice

Introducción de React en JavaScript .....	<b>3</b>
Componente .....	5
Props y State .....	6
Renderizado condicional .....	8
Proyecto en React .....	<b>9</b>
Componente .....	13
Props .....	16
Children .....	18
Estilos .....	19
Íconos .....	22
Eventos .....	23
Formularios .....	24
Arrays .....	25
JSON .....	26
Hooks .....	27
State .....	27
Effect .....	28
Context .....	29
Ref y LayoutEffect .....	32
Memo .....	33
Callback .....	34
Reducer .....	35
Otros Hooks .....	36
React Router (navegación entre páginas SPA) .....	37
createBrowserRouter .....	38
Routes .....	40
React Redux (gestor de estados globales) .....	42
React Query (gestor de peticiones externas) .....	48
Build .....	54
GitHub pages .....	55
Modularización .....	56
React Native .....	<b>58</b>
Widgets .....	60
Explorar galería .....	63
Share .....	
Splash Screen & Icon .....	
Deploy .....	



## Introducción de React en JavaScript

React permite la construcción de interfaces de usuario tanto para aplicaciones web como apps nativas.

Es un framework que cambia el tipo imperativo de JS por un tipo declarativo.

Está basado en componentes.

Para comenzar a usarlo en nuestra página web, es necesario importarlo en nuestro <head> con los siguientes scripts:

React Core:

```
<script src="https://unpkg.com/react@18/umd/react.development.js"></script>
```

React para Páginas Web:

```
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

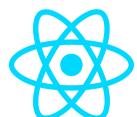
JavaScript XML (**JSX**) para pruebas de funcionamiento (sólo sirve para testing porque enlentece la página). Para que funcione, debes usar el atributo `type="text/babel"` en el archivo de JS de nuestro código para transformarlo en código válido de React / JavaScript.

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

Para que este último script funcione, se debe usar el atributo `type="text/babel"` en el archivo de JS de nuestro código para transformarlo en código válido de React / JavaScript.

Se pueden hacer pruebas de cómo transforma código aquí: <https://babeljs.io/repl>

<pre>1 const Avatar = params =&gt; { 2   const src = `https://randomuser.me/api/portraits/women/\${ 3     params.id 4   }.jpg`; 5 6   return &lt;img src={src} /&gt; 7 };</pre>	<p>JSX</p>	<pre>1 "use strict"; 2 3 var Avatar = function Avatar(params) { 4   var src = "https://randomuser.me/api/portraits/women/".concat(pa rams.id, ".jpg"); 5   return /*#___PURE__*/React.createElement("img", { 6     src: src 7   }); 8 };</pre>	<p>ReactJS</p>
---	------------	--	----------------



Sin React, procedímos de la siguiente manera:

```
<h1>Women in tech</h1>
<div id="app"></div>
```

```
const $app = document.getElementById("app");

const Avatar = params => {
  const src = `https://randomuser.me/api/portraits/women/${params.id}.jpg`;
  return `<picture>
    
    <em>${params.name}</em>
  </picture>`;
};

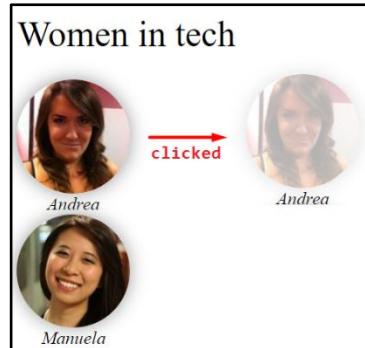
$app.innerHTML += Avatar({ id: 1, name: "Andrea" });
$app.innerHTML += Avatar({ id: 2, name: "Manuela" });

// AL tocar una imagen, se deshabilita o habilita
$app.querySelectorAll('img').forEach(img => {
  img.addEventListener("click", () => {
    img.classList.toggle("disabled");
  });
});
```

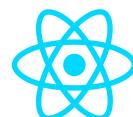
```
picture {
  display: flex;
  flex-direction: column;
  align-items: center;
  width: 100px;
}

img {
  box-shadow: 0 0 15px #999;
  border-radius: 50%;
  width: 100px;
}

img.disabled {
  opacity: 0.3;
}
```



Esto implica crear las imágenes una a una, y luego recorrerlas a todas para aplicar, en cada una, el evento que ocurrirá al clickear sobre ellas para cambiarles la clase “disabled”. Esta forma de trabajar se denomina “imperativa” ya que ordenas a cada elemento tras ser creado una acción / evento.



## Componente

React cambia esta forma de trabajo de JavaScript por una más declarativa y, junto con el uso de JSX, más amigable a la vista del desarrollador.

Podemos crear componentes (porciones de interfaz) de la siguiente manera

```
const $app = document.getElementById("app");

// JSX que será transformado luego por babel a código React válido en JS
const Avatar = params => { // Componente
  const src = `https://randomuser.me/api/portraits/women/${params.id}.jpg`;
  return (
    <picture>
      <img src={src}>
      <em>{params.name}</em>
    </picture>
  );
}

// Crea los elementos en el DOM (con el parámetro 'id')
ReactDOM.render(
  <div>
    <Avatar id={10} name="Andrea"/>
    <Avatar id={11} name="Manuela"/>
  </div>,
  $app // ---> será el htmlElement contenedor
);
```

Y, para que tengan funcionalidad, se declara el método `onClick` al momento de crearlos (*aclaración*: no es lo mismo que `onclick` de **HTML**, la cual es una mala práctica)

En este caso, se puede utilizar así:

```
const Avatar = params => {
  const src = `https://randomuser.me/api/portraits/women/${params.id}.jpg`;
  return (
    <picture>
      <img onClick={event => event.target.classList.toggle("disabled")} src={src}>
      <em>{params.name}</em>
    </picture>
  );
}
```

Obtiene el htmlElement

Sin embargo, esta forma sigue siendo imperativa, ya que se le está “ordenando” una acción a partir de un evento.



## Props y State

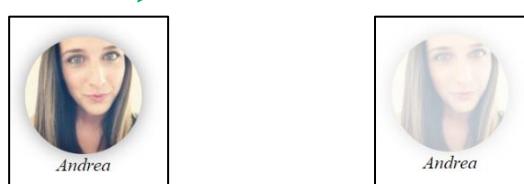
Las **props** son las propiedades de un componente, en general son los parámetros que recibe un componente al momento de ser creado.

Define también comportamientos y otros atributos de un componente.

```
const Avatar = params => { → const Avatar = props => {
```

El **state** es el estado en que se encuentra un componente, por ejemplo, si un componente se encuentra activo o inactivo.

En este caso:



Cada vez que el **prop** o el **state** de un componente cambia, React vuelve a renderizar el componente para que se visualice actualizado, haciendo las modificaciones mínimas para evitar usar recursos excesivos para volver a cargar todo.

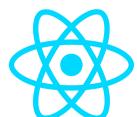
```
const $app = document.getElementById("app");
const {useState} = React; // useState = React.useState;

// JSX que será transformado luego por babel a código React válido en JS
const Avatar = props => {
  const [enabled, setEnabled] = useState(true) // valor inicial del estado (devuelve un array)
  const imgClassName = enabled ? '' : 'disabled';

  const src = `https://randomuser.me/api/portraits/women/${props.id}.jpg`;
  return (
    <picture>
      <img
        onClick={() => setEnabled(!enabled)}
        className={imgClassName}
        src={src}
      />
      <em>{props.name}</em>
    </picture>
  );
}

// Crea los elementos en el DOM (con el parámetro 'id')
ReactDOM.render(
  <div>
    <Avatar id={10} name="Andrea"/>
  </div>,
  $app // ---> será el htmlElement contenedor
);
```

El atributo **class** de **HTML** se llama **className** en **JSX** ya que **class** en **JavaScript** está siendo usado para otra función (creación de clases / objetos).



- Desestructuración de props

Se pueden desestructurar parámetros (**props**) al crear un componente.

```
const Avatar = props => {
```

```
const Avatar = ({id, name}) => {
```

De esta manera, se usarán solo las propiedades de **id** y **name** en la creación del componente.

Además, ya no será necesario usar la palabra *props* para acceder a la propiedad.

```
<em>{props.name}</em>
```

```
→ <em>{name}</em>
```

- Default props

Teniendo las **props** desestructuradas, se le pueden asignar valores por defecto.

```
const Avatar = ({id, name = 'un nombre'}) => {
```

Otra forma más antigua de crear un Default prop es mediante la declaración de los valores por defecto por fuera de la función.

```
} // end Avatar function  
  
Avatar.defaultProps = {  
  name: "un nombre"  
};
```



## Renderizado condicional

Se trata de renderizar una cosa u otra según las **props** o **state** que se tengan declaradas.

Siguiendo el ejemplo anterior, si se creara un Avatar sin nombre, tenemos definida una prop por defecto. Sin embargo, si se creara un Avatar sin id, se crearía un componente extraño.

Para evitar que esto suceda, se pueden añadir condicionales como el siguiente:

```
const $app = document.getElementById("app");
const {useState} = React; // useState = React.useState;

const Avatar = ({id, name = 'un nombre'}) => {

    const [enabled, setEnabled] = useState(true) // valor inicial del estado
    const imgClassName = enabled ? '' : 'disabled';

    const src = `https://randomuser.me/api/portraits/women/${id}.jpg`;

    return (
        <picture>
            {
                id ? (
                    <img
                        onClick={() => setEnabled(!enabled)}
                        className={imgClassName}
                        src={src}
                    />
                ) : (
                    <i>Sin imagen</i>
                )
            }
            <em>{enabled ? name : "Desactivada"}</em>
        </picture>
    );
}
```

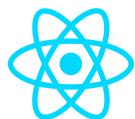
Se ejecutará esto si la **id** no está definida

Si el estado está activo, se mostrará el nombre. Si no, “desactivada”.

```
// Crea los elementos en el DOM
ReactDOM.render(
    <div>          sin id
        <Avatar name="Andrea"/>
    </div>,
    $app // ---> será el htmlElement contenedor
);
```

*Sin imagen*  
Andrea

```
i {
    border: 1px solid #ccc;
    border-radius: 50%;
    display: block;
    width: 100px;
    height: 100px;
    margin: auto;
    text-align: center;
}
```



## Proyecto en React

Para iniciar un nuevo proyecto de React, es necesario **NodeJS**

<https://nodejs.org/en/download>

Downloads

Latest LTS Version: **18.16.1** (includes npm 9.5.1)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features
 Windows Installer node-v18.16.1-x64.msi	 macOS Installer node-v18.16.1.pkg
	 Source Code node-v18.16.1.tar.gz

La versión LTS es la más estable, mientras que Current es la última sin soporte actualizado.

Una vez descargado e instalado, comprobar que la instalación fue exitosa abriendo la consola del sistema y verificar la versión instalada.

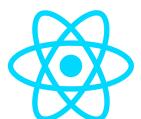
```
cmd Command Prompt
Microsoft Windows [Version 10.0.17758.1]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Admin>Node --version
v14.17.3

C:\Users\Admin>npm --version
6.14.13

C:\Users\Admin>
```

– Verificación de la instalación de Node.js en Windows.



Teniendo instalado **NodeJS**, se puede iniciar un nuevo proyecto desde la terminal, estando ubicados en el directorio donde se quiere crear el repositorio.

Es necesario tener conexión a internet para descargar todos los paquetes necesarios.

```
Símbolo del sistema
Microsoft Windows [Versión 10.0.19045.3324]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\Guill>cd C:\xampp\htdocs

C:\xampp\htdocs> npx create-react-app nombre-de-mi-app-sin-espacios
```

Creado el repositorio, se puede ingresar a la carpeta creada desde la consola para tener disponibles los siguientes comandos:

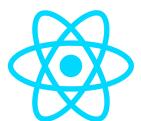
- `npm start` Inicia el servidor local con el proyecto (Ctrl + C para cerrarlo)
- `npm run build` Compila el proyecto en la carpeta “build” (para publicarse)
- `npm test` Inicia el modo test
- `npm run ejects` Compila el proyecto y elimina este directorio (no hay vuelta atrás)

Además, la carpeta creada con el repositorio contendrá varias carpetas con distintos archivos:

The screenshot shows a file explorer on the left and a browser window on the right. The file explorer displays the directory structure of a React application:

- `node_modules` → **archivos necesarios para React**
- `public` → **código HTML autogenerado**
  - `favicon.ico`
  - `index.html`
  - `logo192.png`
  - `logo512.png`
  - `manifest.json`
  - `robots.txt`
- `src` → **nuestro código**
  - `App.css`
  - `App.js`
  - `App.test.js`
  - `index.css`
  - `index.js`
  - `logo.svg`
  - `reportWebVitals.js`
  - `setupTests.js`
  - `.gitignore`

The browser window shows the React App with the React logo. Below the browser, there is a message: **Edit `src/App.js` and save to reload.** and a link: [Learn React](#).



El archivo **index.js** es donde comenzará la ejecución del código. Por defecto, React importa dentro de **index.js** el módulo **App.js**, y es allí dónde comenzaremos a escribir nuestro código.

```
✓ src
  App.css
  App.js
  App.test.js
  App.css
  index.css
  index.js
  logo.svg
  reportWebVitals.js
  setupTests.js
  .gitignore
  package-lock.json
  package.json
  README.md
```

```
1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  import './index.css';
4  import App from './App';
5  import reportWebVitals from './reportWebVitals';
6
7  const root = ReactDOM.createRoot(document.getElementById('root'));
8  root.render(
9    <React.StrictMode>
10   <App />
11   </React.StrictMode>
12 );
13
14 // If you want to start measuring performance in your app, pass a function
15 // to Log results (for example: reportWebVitals(console.log))
```

Por defecto, React crea el módulo **App.js** de la siguiente forma:

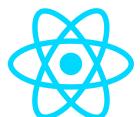
```
JS App.js
```

```
nombre-de-mi-app-sin-espacios > src > JS App.js > ...
1  import logo from './logo.svg';
2  import './App.css';
3
4  function App() {
5    return (
6      <div className="App">
7        <header className="App-header">
8          <img src={logo} className="App-logo" alt="logo" />
9          <p>
10            Edit <code>src/App.js</code> and save to reload.
11          </p>
12          <a
13            className="App-link"
14            href="https://reactjs.org"
15            target="_blank"
16            rel="noopener noreferrer"
17          >
18            Learn React
19          </a>
20        </header>
21      </div>
22    );
23  }
24
25 export default App;
```

*Indica que se exportará todo el módulo*

Es una buena práctica que los módulos a utilizar de React tengan extensión JSX.





## ViteJS

Permite crear proyectos de una forma más eficiente que el modo por defecto que brinda react-create-app.

Teniendo **NodeJS** ya instalado, se debe instalar desde la consola, ubicados en el directorio donde se creará el proyecto.

```
npm create vite
C:\xampp\htdocs> npm create vite
Need to install the following packages:
  create-vite@4.4.1
Ok to proceed? (y)
```



Al intentar crear un nuevo proyecto, la consola misma te dirá que debes instalar el package necesario, para lo cual accedes y se procede a la instalación.

Finalizada la instalación, se pedirá que especifiques el nombre del proyecto, el cual no debe contener ningún espacio.

```
npm create vite
C:\xampp\htdocs> npm create vite
? Project name: » vite-project
```

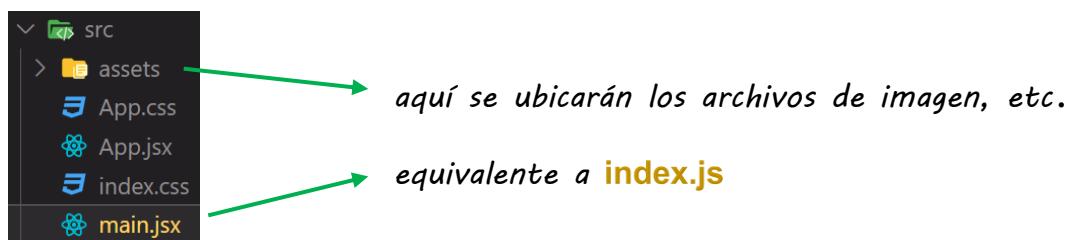
Luego de esto, se preguntará cuál es el framework con el que se desea trabajar, en este caso será react, seguido de si se quiere usar JavaScript o TypeScript.

```
C:\Windows\system32\cmd.exe
? Select a framework: » - Use arrow-keys. Return to submit.
  Vanilla
  Vue
  React
  Preact
  Lit
  Svelte
  Solid
  Qwik
  Others
```

```
C:\Windows\system32\cmd.exe
✓ Select a framework: » React
? Select a variant: » - Use arrow-keys. Return to submit.
  TypeScript
  TypeScript + SWC
  > JavaScript
  JavaScript + SWC
```

Finalmente se creará el directorio del proyecto.

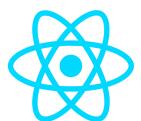


En consola se describen los siguientes pasos para finalizar la creación del proyecto.

```
C:\Windows\system32\cmd.exe
Scaffolding project in C:\xampp\htdocs\vite-project...
Done. Now run:
  cd vite-project      entrar al directorio
  npm install           instalar packages
  npm run dev            iniciar servidor local
```

12

```
npm run dev -- --host
```



## Componente

React intenta crear una página web a partir de partes de interfaz. Un componente es una porción de interfaz que puede replicarse, cada una con sus propiedades (**props**) y estados (**state**), con su respectivo código HTML, CSS y JavaScript.

### Componente root

js index.js

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

El componente root es similar al `<body>` de **HTML**, es decir, es el componente sobre el cual se crean todos los componentes que compondrán la página.

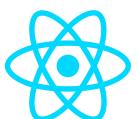
### Creación de un componente

Los componentes siempre son una función que retornan porciones de interfaz, y dicha función siempre está escrita **con la primera letra en mayúscula**.

```
const HelloWorld = () => {
  return <h1>Hola mundo</h1>
}
```

Estos componentes no necesariamente deben tener un solo htmlElement

```
const ComponenteEjemplo = () => {
  return <div>
    <h1>Soy un componente</h1>
    <p>con más de un elemento</p>
  </div>
}
```



## Renderización de componentes

Para renderizar el componente creado, tantas veces como se desee, se utiliza el método `render` del componente root creado.

Hay que tener en cuenta que los componentes deben estar contenidos en un `htmlContainer` (como un `div` u otro). De lo contrario, no puede renderizarse.

```
root.render(  
  <div>  
    |   <HelloWorld/>  
  </div>  
)
```

*Self Closing Tag*

```
root.render(  
  <HelloWorld/>  
  <HelloWorld/>  
)
```

JSX expressions must have one parent element. ts(2657)  
`const HelloWorld: () => React.JSX.Element`  
View Problem (Alt+F8) Quick Fix... (Ctrl+.)

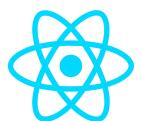
Para evitar esto, podemos utilizar una etiqueta especial de JSX llamada `Fragment` que actúe como contenedor, la cual debe expresarse como `<> ... </>`.

```
root.render(<>  
  <HelloWorld/>  
  <HelloWorld/>  
</>);
```

## Clases de componentes

Otra forma de crear un componente es con una class, aunque no es la forma recomendada de hacerlo.

```
import {Component} from 'react'  
  
export class HelloWorld extends Component {  
  render() {  
    return <h1>Hello world</h1>  
  }  
}
```



## Interpretación de código en componentes

Para añadir variables / constantes a un componente se utilizan las llaves, las cuales permiten la interpretación de código.

```
const ComponenteEjemplo = () => {
  const unNombre = "Guillermo"
  return <h1>Hola {unNombre}</h1>
}
```

También permite la implementación de condicionales con operadores ternarios

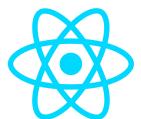
```
const ComponenteEjemplo = () => {
  const estaCasado = false;
  return <h1>Hola { estaCasado ? "ya me casé" : "sigo soltero" }</h1>
} // => "Hola sigo soltero"
```

y el uso de funciones

```
const ComponenteEjemplo = () => {
  function sumar(a, b) {
    return a + b;
  }
  return <h1>La suma da: {sumar(1, 2)}</h1>
} // => La suma da: 3
```

Si sólo se desea mostrar en caso de que la condición sea verdadera (o falsa):

```
const ComponenteEjemplo = () => {
  const edad = 19;
  return <p>Hola { edad > 18 && <span>señor</span> } </p>
}
```



## Props

Las **props** son las propiedades (o parámetros) que recibe un componente, los cuales se pasan como un objeto por parámetro a la función que actúa como creador del componente.

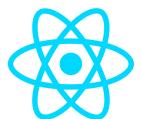
```
const Componente = props => {
  return <p>Yo tengo props como {props.unString}</p>
}
```

O bien, se puede desestructurar para manejar directamente las props esperables.

```
const Componente = ({unString = "valor por defecto"}) => {
  return <p>Yo tengo props como {unString}</p>
}
```

Para que un componente reciba dichas **props**, al momento de renderizar se definen como si fueran atributos de HTML.

```
root.render(<>
  <Componente unString = "Los strings no usan llaves"/>
  <Componente unNumero = { 28 }/>
  <Componente unBool = { true }/>
  <Componente unArray = { [1, 6, 1995] }/>
  <Componente unObjeto = { {calle: "Av. Mitre", altura: 2000} }/>
  <Componente unaFuncion = { function(){ alert('una funcion') } }/>
</>);
```



Así se pueden crear componentes más complejos tales como

```
const UserCard = ({id, name="User", married=false, points=[], ubication={street:"No"}) => {
  return (<div>
    {
      id ? (<> // el id está definido
        <p>Mi nombre es {name}</p>
        <p>Estoy {married ? "casado" : "soltero"}</p>
        <p>El máximo de mis puntajes es { Math.max(...points) }</p>
        <p>Vivo en la calle {ubication.street}</p>
      ) : ( <p>Mi ID no está definida</p> )
    }
  </div>);
}
```

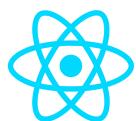
```
root.render(<>
  <UserCard
    id = {1234}
    name = "Guillermo"
    married = { false }
    points = { [7, 2, 5, 1] }
    ubication = { {street: "San Nicolas", number: 368} }/>
</>);
```

Mi nombre es Guillermo  
Estoy soltero  
El máximo de mis puntajes es 7  
Vivo en la calle San Nicolas

## Prop types

Existe una librería de NPM que permite establecer qué tipos de datos puede recibir un componente (como ayuda al desarrollador).

1:29 FAZT



## Children

Es una prop especial que permite acceder a los elementos hijos de un componente.



## Estilos

Hay varias formas de añadir estilos a componentes.

### Estilos en línea

Como en HTML, se escribe directamente en la etiqueta.

Para que JSX lo reconozca como un estilo en línea, debe escribirse entre llaves y con formato de objeto.

Estos estilos están siendo trabajados como código Javascript (un objeto json), por lo que puede guardarse en una variable separada para que no ocupe el mismo espacio en la etiqueta.

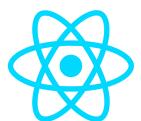
```
const UserCard = ({name, age}) => {
  return (<div style={{
    display: 'inline-block',
    border: '1px solid #0e0e0e',
    margin: '2px 8px'
  }>
    <p>Mi nombre es {name}</p>
    <p>Tengo {age} años</p>
  </div>);
}
```

```
const myStyle = {
  display: 'inline-block',
  border: '1px solid #0e0e0e',
  margin: '2px 8px'
}
```

```
const UserCard = ({name, age}) => {

  return (<div style={myStyle}>
    <p>Mi nombre es {name}</p>
    <p>Tengo {age} años</p>
  </div>);

}
```



## Estilo de un archivo externo

Se utiliza un archivo **CSS** separado del **JSX** donde se ubica nuestro componente.

```
UserCard.jsx
import './UserCard.css'; // estilos

const UserCard = ({name, age}) => {

  return (<div className='card-div'>
    <p>Mi nombre es {name}</p>
    <p>Tengo {age} años</p>
  </div>);

}

export default UserCard;
```

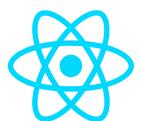
```
UserCard.css
.card-div {
  display: inline-block;
  border: 1px solid #0e0e0e;
  margin: 2px 8px;
  color: #rgb(205, 17, 205);
}
```

```
index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import UserCard from './UserCard';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <UserCard name="Guillermo" age={28}/>
  </React.StrictMode>
);
```

Mi nombre es Guillermo  
Tengo 28 años

El atributo **class** de **HTML** se llama **className** en **JSX** ya que **class** en **JavaScript** está siendo usado para otra función (creación de clases / objetos).



## Modularización

Se trata de crear archivos CSS con extensión module para poder importar los estilos como un objeto.

```
import styles from './Title.module.css';

export default function Title() {
  <h1 className={styles.text}>Hello world</h1>
}
```

```
.text {
  font-size: 32px;
}
```

```
components
  title
    Title.jsx
    Title.module.css
```

Se pueden concatenar className utilizando template strings

```
<div className={`${styles.clase1} ${styles.clase2}`}>
```

## Composición

Utilizando el sistema de módulos, se puede hacer una composición de estilos mediante la palabra reservada **composes**.

CSS modules also allow you to combine multiple classes through the **composes** keyword. For example, consider the following .btn class above. You could “extend” that class in other classes using **composes**.

For a submit button, you could have:

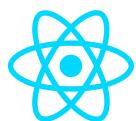
```
.btn {
  /* styles */
}

.submit {
  composes: btn;
  background-color: green;
  color:#FFFFFF
}
```

This combines the .btn and the .submit classes. You can also compose styles from another CSS module like this:

```
.submit {
  composes:primary from './colors.css';
  background-color: green;
}
```

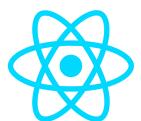
Note that you must write the **composes** rule before other rules.



## Íconos

Existen librerías con Íconos especialmente creados para React.

2:17



## Event handler

Son los eventos relacionados a un componente, como es `addEventListener` en JS.

Existen diversos eventos que pueden aplicarse, todos tienen prefijo `on` y van variando según el `htmlElement` al que se le quiera aplicar:

```
const Componente = () => {
  return <button onBoton>Botón</button>
}

// onAbort?
// onAbortCapture?
// onAnimationEnd?
// onAnimationEndCapture?
// onAnimationIteration?
// onAnimationIterationCapture?
// onAnimationStart?
// onAnimationStartCapture?
// onAuxClick?
// onAuxClickCapture?
// onBeforeInput?
// onBeforeInputCapture?
```

nota: no es lo mismo que eventos en línea de [HTML](#), los cuales son una mala práctica

Cada evento tiene un parámetro por defecto, el cual devuelve el mismo evento con información del `htmlElement` que lo disparó.

```
// Se ejecuta con cada tipo de usuario
const handleChange = event => {
  console.log(event);
  console.log(event.target); // input del evento
  console.log(event.target.value); // texto del input
}

// Se ejecuta cuando el botón es tocado
const handleClick = event => {
  // code
}

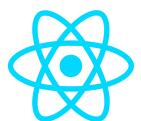
// Componente JSX
const Button = ({text = "Send"}) => {

  return (<div>
    <input onChange={ handleChange }/>
    <button onClick={ handleClick }>{text}</button>
  </div>);
}
```

```
SyntheticBaseEvent {_reactName: 'onChange', _targetInst: null, type: "change", nativeEvent: InputEvent, target: input, ...} 
  bubbles: true
  cancelable: false
  currentTarget: null
  defaultPrevented: false
  eventPhase: 3
  isDefaultPrevented: false
  isPropagationStopped: false
  isTrusted: true
  nativeEvent: InputEvent {isTrusted: true, data: "a", isComposing: false}
  target: input
  timeStamp: 22342
  type: "change"
  _reactName: "onChange"
  _targetInst: null
  [[Prototype]]: Object
```

Si se quieren pasar parámetros adicionales a la función, se utiliza una función flecha.

```
onClick={(event)=>showArticleInfo(event, article)}
```



## Formularios

Al igual que en HTML, los formularios están diseñados para enviar datos al servidor / backend, por lo que por defecto se intentará hacer esta acción y, al no encontrar, se recargará la misma página enviando por parámetros los contenidos de los input del form.

React App

localhost:3000/?name=Guille&amount=28&textarea=Al+tocar+el+botón+de+Send+de+

Traductor Cursos Outlook Gmail Gestión de Personal WhatsApp Discord

FORM TITLE

Name Amount

Guille 28

Description

Al tocar el botón de Send de un form, por defecto se recargará la misma página con parámetros de este formulario.

Send

Este comportamiento por defecto se puede quitar haciendo un manejo del evento.

```
import './Formulario.css';

const handleSubmit = e => {
  e.preventDefault(); // Cancela la acción por defecto
  console.log(e);
}

const Formulario = () => {
  return (
    <div className="form-container">
      <form id="form" onSubmit={handleSubmit}>
        <fieldset>
          <legend>Form title</legend>

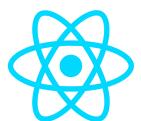
          <div className="form-container__inputcontainer"> ...
          </div>

          <div className="form-container__submitcontainer">
            <button id="submit">Send</button>
          </div>

        </fieldset>
      </form>
    </div>
  );
}
```

```
Formulario.jsx:5
SyntheticBaseEvent {reactName: 'onSubmit', _targetInst: null, type: 'submit', nativeEvent: SyntheticEvent {cancelBubble: true, cancelable: true, currentTarget: null, defaultPrevented: true, eventPhase: 3, isDefaultPrevented: functionThatReturnsTrue(), isPropagationStopped: functionThatReturnsFalse(), isTrusted: true, nativeEvent: SubmitEvent {isTrusted: true, submitter: button#submit, type: 'submit', target: form#form}, target: form#form, 0: fieldset, 1: input#input-name.input, 2: input#input-amount.input, 3: textarea#input-text.input, 4: button#submit}, _reactFiber$qvllle162ie: FiberNode {tag: 5, key: null, elementType: 'form', type: 'form', acceptCharset: "", accessKey: "", action: "http://localhost:3000/?name=&amount=&textarea="}}
```

En este caso, el **target** contiene todos los input del formulario



## Arrays

Los arrays son listas de cualquier tipo de datos, desde variables primitivas hasta objetos más complejos.

En **JavaScript** se utilizan bucles para recorrerlos, como los siguientes:

```
// Un array de objetos
const unArray = [
  { name: "Guille", age: 28 },
  { name: "Marce", age: 45 },
  { name: "Josi", age: 44 }
]
```

```
unArray.forEach((element, index) => {
  console.log(element, index); // imprime elementos de un array
});
```

usado para  
**NodeList**

```
for (const element of unArray) {
  console.log(element); // imprime elementos de una Lista
}
```

usado para  
**HTMLCollection**

```
for (const index in unArray) {
  console.log(unArray[index]); // imprime por index
}
```

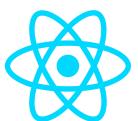
```
/* Crea un nuevo array de Lo que devuelva la
   función dentro de map en cada elemento      */
const nombres = unArray.map((element, index) => element.name);
console.log(nombres); // ['Guille', 'Marce', 'Josi']
```

**React** permite aprovechar los recorridos que ya existen para arrays y utilizarlos dentro de los componentes:

```
// Componente JSX
const Lista = () => {
  return (<>
    {
      unArray.map((element, index) =>
        <p key={index}>Nombre: {element.name}</p>
      )
    }
  </>);
}
```

Nombre: Guille  
Nombre: Marce  
Nombre: Josi

*El atributo `key` no es necesario realmente, pero si no se usa, se generará un warning en la consola. Esto no sucede al compilar la página para producción por lo que puede ser ignorado.*



## JSON

En React se pueden crear archivos JSON e importarlos directamente en el código sin necesidad de acceder a ellos mediante fetch o axios.

```
import jsonData from './archivo.json';
```

Además, se puede importar sólo una clave específica del JSON en caso de no necesitar el archivo completo.

```
import { clave1 } from './data.json';
```

```
json
{
  "clave1": "valor1",
  "clave2": "valor2",
  "clave3": "valor3"
}
```

### Iteración

A diferencia de un Array, los objetos JSON no pueden recorrerse de cualquier manera, sino que sólo puede hacerse con un **for-in**:

```
for in
for (const key in jsonObject) {
  const value = jsonObject[key];
  print(value);
}
```

O bien, tratándolos como objetos

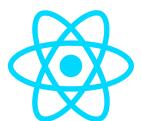
```
const keys = Object.keys(jsonObject);
keys.forEach(key => {
  const value = jsonObject[key];
  print(value);
})
```

```
const values = Object.values(jsonObject);
values.forEach(value => { // no key
  print(value);
})
```

```
Object.entries(jsonObject).forEach(([key, value]) => {
  print(key + ": " + value);
});
```



## Hooks

Son funciones que permiten guardar datos, cambiarlos y usar **states**.

- **useState**

```
import {useState} from 'react';
```

Se utiliza para guardar variables / estados de un componente.

Para comenzar a usarlo, es necesario importarlo y luego crear el objeto dentro del componente.

```
const Componente = props => {
  const [nombreVariable, setterVariable] = useState( );
  ↓           ↓           ↓
  Esta es la variable o   Esta es una función que   Aquí se puede
  state que creamos       permite setear un nuevo   establecer el
                           valor a la variable o   valor con el
                           state creada            que inicia la
                                         variable o
                                         state a crear
```

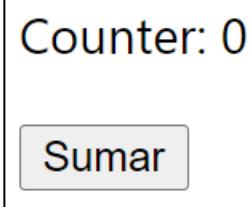
De esta manera, ya se pueden usar variables / estados dentro de un componente.

Por ejemplo:

```
import {useState} from 'react';
const Contador = () => {

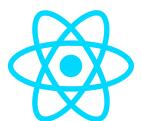
  const [numero, setNumero] = useState(0);

  return (
    <div>
      <p>Counter: {numero}</p>
      <button onClick={ () => setNumero(numero + 1) }>Sumar</button>
    </div>
  );
}
```



Luego del click →





## ➤ **useEffect**

```
import {useEffect} from 'react';
```

Se utiliza cuando se realizan cambios en un componente de forma asíncrona. Puede ser de mucha utilidad para solicitudes de red.

Existen varias formas de usarse, dependiendo los parámetros que le pasemos a la función dentro del componente.

- ✓ Se ejecuta sólo cuando el componente es creado

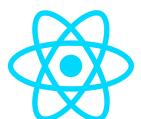
```
const Componente = props => {
  useEffect( () => {/* una función */}, []);
  ...
}
```

- ✓ Se ejecuta siempre que haya un cambio en el componente

```
const Componente = props => {
  useEffect( () => {/* una función */});
  ...
}
```

- ✓ Se ejecuta siempre que haya un cambio en un **state** específico del componente

```
const Componente = props => {
  const [unState, setState] = useState("valor inicial");
  useEffect( () => {/* una función */}, [unState]);
  ...
}
```



## ➤ useContext

Permite la creación de un componente que englobe al resto y que, por lo tanto, contenga la información de todos los componentes hijos (*children*).

Para hacerlo, es necesario crear dicho componente global de la siguiente forma:

### ComponentContext.jsx

```
import { createContext } from "react";

// Contexto
export const ComponentContext = createContext();

// Componente global
export function ComponentContextProvider(props) {

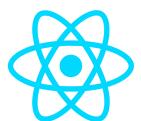
    return (
        <ComponentContext.Provider>
            {props.children}
        </ComponentContext.Provider>
    );
}
```

Luego, se debe incorporar el componente global en el renderizado de nuestro **root component**.

### index.js

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'
import { ComponentContextProvider } from './ComponentContext.jsx'

ReactDOM.createRoot(document.getElementById('root')).render(
    <React.StrictMode>
        <ComponentContextProvider>
            <App />
        </ComponentContextProvider>
    </React.StrictMode>,
)
```



Para poder asignar valores al componente global (**context**) creado, es necesario crear dicha funcionalidad de la siguiente manera:

ComponentContext.jsx

```
import { createContext } from "react";

// Contexto
export const ComponentContext = createContext();

// Componente global
export function ComponentContextProvider(props) {

    let variableEjemplo = "Variable Ejemplo";   1

    return (
        <ComponentContext.Provider value={variableEjemplo}>
            <h1>Component context</h1>
            {props.children}
        </ComponentContext.Provider>
    );
}
```

Así, el componente global contiene una variable a la cual todos sus componentes hijos pueden acceder:

ComponentChildren.jsx

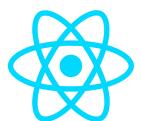
```
import { ComponentContext } from "./ComponentContext";
import { useContext } from "react";

const ComponentChildren = () => {

    const value = useContext(ComponentContext);   2

    return (
        <p> { value } </p>
    );
}

export default ComponentChildren;
```



Dentro del `value` de **Provider** del componente global (**context**) se pueden enviar desde pequeñas variables (como un **state**) hasta funciones, y si se desean enviar más de una, se las puede englobar en un **objeto**.

```
import { createContext } from 'react';
import { useState } from 'react';

// Contexto
export const ComponentContext = createContext();

// Componente global
export function ComponentContextProvider(props) {

    let [unState, setState] = useState();

    function funcionDelContexto(param) {
        // code
    }

    return (<>
        <ComponentContext.Provider
            value = {
                {
                    unState: unState,
                    unaFuncion: funcionDelContexto
                }
            }
        >
            {props.children}
        </ComponentContext.Provider>
    </>);
}
```

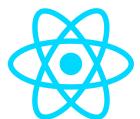
En `value` hay  
un **objeto**

Una forma de resumir el objeto con clave-valor de mismo nombre es escribirlo directamente así:

```
{
    unState,
    funcionDelContexto
}
```

Para obtener sólo uno de los elementos pasados en el `value` del contexto desde un componente `children`, se utilizan las llaves con el nombre del elemento.

```
// Obtener sólo La función del contexto
const { funcionDelContexto } = useContext(ComponentContext);
```



## ➤ useRef

Su uso es similar al de un **querySelector**. Permite obtener un elemento específico del Componente.

## ➤ useLayoutEffect

Funciona exactamente igual que useEffect, excepto que este hook se ejecuta de forma síncrona.

Puede ser útil para medir elementos del DOM.

```
import { useLayoutEffect, useRef, useState } from 'react';

const Component = (props) => {

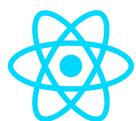
  const divRef = useRef();

  const [ height, setHeight ] = useState();

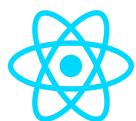
  useLayoutEffect(() => {
    setHeight(divRef.current.clientHeight)
  }, []);

  console.log(height); // Altura que ocupa el div

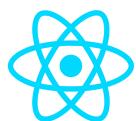
  return (
    <div ref={divRef}>
      {props.children}
    </div>
  )
}
```



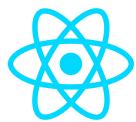
➤ **useMemo**



➤ **useCallback**



➤ **useReducer**



## Otros Hook

- [useImperativeHandle](#)
- [useDebugValue](#)
- [useTransition](#)
- [useId](#)
- [useSyncExternalStore](#)
- [useInsertionEffect](#)

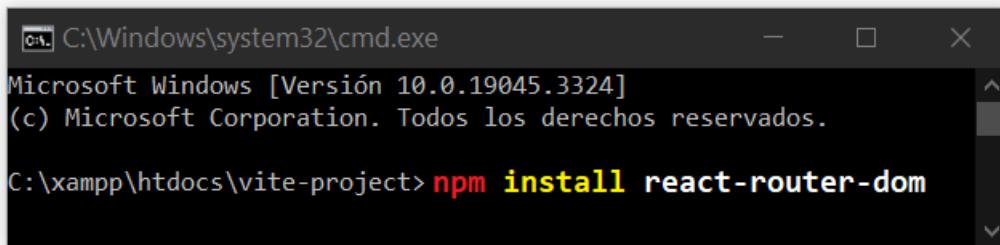


## React Router

Permite la navegación entre páginas en React con un formato de SPA (una sola página que actualiza su contenido simulando el paso de una página a otra pero manteniendo las funciones de navegación normales como las flechas adelante y atrás).

### Instalación

Para usarse, se requiere instalar la librería **react-router-dom** de NPM.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 10.0.19045.3324]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\xampp\htdocs\vite-project> npm install react-router-dom
```

### Routes

Se crea un directorio (**routes**) que contenga todos los componentes que se renderizarán en cada ruta de la página web.

- **Index (layout)**: componente principal que se renderizará al iniciar la página web y, mediante **Link** se pueden acceder a cada ruta.



routes

- About.jsx
- ErrorRoute.jsx
- Index.jsx
- User.jsx
- Users.jsx

Index.js

```
import { Link } from "react-router-dom";

export default function Index() {
  return <>
    <h1>Este es el index</h1>
    <Link to="/">Index</Link>
    <Link to="/about">About</Link>
    <Link to="/users">Users</Link>
  </>
}
```

localhost:5173

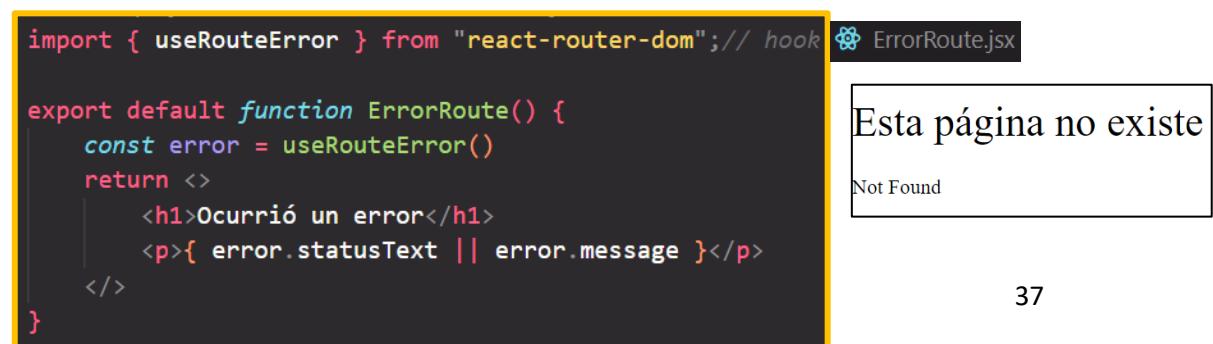
# Este es el index

[Index](#)

[About](#)

[Users](#)

- **Ruta de error**: componente que se renderizará cuando ocurra un error.



```
import { useRouteError } from "react-router-dom"; // hook

export default function ErrorRoute() {
  const error = useRouteError()
  return <>
    <h1>Ocurrió un error</h1>
    <p>{ error.statusText || error.message }</p>
  </>
}
```

ErrorRoute.jsx

Esta página no existe

Not Found



## Configuración de las rutas con createBrowserRouter

En el archivo **App.js** se configuran todas las rutas del proyecto.

App.js

```
import { createBrowserRouter, RouterProvider } from 'react-router-dom';

import Index from './routes/Index';
import About from './routes/About';
import ErrorRoute from './routes/ErrorRoute';
import Users from './routes/Users';
import User from './routes/User';

const routes = createBrowserRouter([
  {
    path: "*", // todas las rutas no definidas van al index
    element: <Index/>, // este es el componente que se renderizará
    errorElement: <ErrorRoute/> // aquí se redirigirá en caso de error
  },
  {
    path: "/about", // en esta ruta, se renderizará el componente About
    element: <About/>,
    errorElement: <ErrorRoute/>
  },
  {
    path: "/users",
    element: <Users/>,
    errorElement: <ErrorRoute/>,
  }
])

export default function App() {
  return <RouterProvider router={routes}>
}
```

### ▪ Basename

Si, por algún motivo, se requiere que todas las rutas tengan una subruta previa desde la raíz, se puede usar **basename** como parte de las opciones disponibles.

```
return <RouterProvider router={
  createBrowserRouter([ ... ],
  {
    basename: "/subruta-anterior"
  })
}/>;
```



localhost:5173/subruta-anterior/



## ▪ Children Routes

Se pueden establecer sub rutas dentro de una página, con la posibilidad de crear una ruta dinámica, es decir, que podrá cargar una cosa u otra según un parámetro (definido luego de dos puntos).

```
{
  path: '/users',
  element: <Users/>,
  errorElement: <ErrorRoute/>,
  children: [ // subruta
    {
      path: 'user/:id',
      element: <User/>
    }
  ]
}
```

Un ejemplo de uso es el de mostrar elementos que fueron cargados de una fuente externa, como un JSON.

```
{
  "users": [
    { "id": 0, "name": "Guillermo", "age": 28 },
    { "id": 1, "name": "Marcela", "age": 45 },
    { "id": 2, "name": "José", "age": 44 },
    { "id": 3, "name": "Sol", "age": 24 },
    { "id": 4, "name": "Jana", "age": 1 }
  ]
}
```

Users.jsx

```
import { Outlet, useNavigate } from 'react-router-dom' // hook
import { users } from '../data/data.json' // cargar JSON

export default function Users() {
  const navigate = useNavigate();
  const showUser = id => navigate(`/users/user/${id}`) // redirección

  return <>
    <h1>Usuarios</h1>
    {
      users.map(user =>
        <button onClick={() => showUser(user.id)}>{user.name}</button>
      )
    }
    <Outlet /> // Donde se mostrará la subruta
  </>
}
```

Permite acceder a la  
subruta sin usar Link

localhost:5173/users

## Usuarios

- Guillermo
- Marcela
- Jose
- Sol
- Jana

User.jsx

```
import { useParams } from 'react-router-dom' // hook
import { users } from '../data/data.json'

export default function User() {
  const { id } = useParams(); // Leer parámetros

  return <>
    <h2>Usuario {id}</h2>
    <p>Nombre: {users[id].name}</p>
    <p>Edad: {users[id].age}</p>
  </>
}
```

Usuario 1

Nombre: Marcela  
Edad: 45

en donde está la etiqueta **Outlet** es donde se cargará el contenido de la subruta



## Configuración de las rutas con la etiqueta Routes

main.jsx

En el archivo **index.jsx** (donde se renderiza el root de la página web) se importa el componente **BrowserRouter**.

```
localhost:5173
En el layout se pueden mostrar los nav links permanentemente

Este es el index

Index
About
Users
```

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'
import { BrowserRouter } from "react-router-dom"

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>,
)
```

Se crea el archivo **Layout.jsx** que representará el header / nav de la página web. Debe contener el elemento **Outlet** que es donde se cargará el resto de la página.

```
// Component
export default function Layout() {
  return <>
    <p>En el layout se pueden mostrar los nav links permanentemente </p>
    <Outlet/>
  </>
}
```

En el archivo **app.jsx** se importan las primeras subrutas del index, es decir, las que corresponden a la ruta / y sus primeras subrutas.

```
import { Routes, Route, Outlet } from 'react-router-dom';

import Index from './routes/Index';
import About from './routes/About';
import ErrorRoute from './routes/ErrorRoute';
import Users from './routes/Users';

export default function App() {
  return (
    <Routes>
      <Route path="/" element={<Layout />}> // componente que siempre se mostrará primero
        <Route index element={<Index />} /> // componente que representa el índice de la página, es decir, cuando no haya ninguna subruta cargada

        <Route path="about" element={<About />} /> // subruta /about
        <Route path="users" element={<Users />} /> // subruta /users
        <Route path="*" element={<ErrorRoute />} /> // cualquier otra subruta no configurada
      </Route>
    </Routes>
  )
}
```



## ▪ Children Routes

Se pueden establecer subrutas directamente dentro del componente que representa una ruta específica.

```
import { Routes, Route, Outlet, useNavigate } from 'react-router-dom' // hook
import { users } from '../data/data.json' // cargar JSON

// Routes
import User from './User';
import ErrorRoute from './ErrorRoute';

export default function Users() {
  return (
    <Routes>
      <Route index element={<Content />} />
      <Route path="user/:id" element={<User />} /> // subruta
      <Route path="*" element={<ErrorRoute />} /> // otra subrutas
    </Routes>
  );
}

const Content = () => {
  const navigate = useNavigate();
  const showUser = id => navigate(`/users/user/${id}`) // redirección

  return <>
    <h1>Usuarios</h1>
    {
      users.map(user =>
        <button onClick={() => showUser(user.id)}>{user.name}</button>
      )
    }
    <Outlet/>
  </>
}
```

De esta manera, el contenido de la subruta se mostrará como una página completamente nueva.

Si desea que la subruta sea un componente completamente que se renderiza justo debajo del contenido principal, se cambia la estructura de **Routes** de la siguiente forma:

```
<Routes>
  <Route path="/" element={<Content />}>
    <Route path="user/:id" element={<User />} />
    <Route path="*" element={<ErrorRoute />} />
  </Route>
</Routes>
```



## React Redux

Proporciona un flujo de datos unidireccional facilitando la gestión del estado global, lo cual evita la necesidad de pasar props a través de múltiples componentes.

Es similar al useContext, sin embargo, redux permite más funcionalidades.

### Instalación

Para usarse, se requiere instalar la librería **react-redux** de NPM.



```
C:\...\API Rest> npm install react-redux redux @reduxjs/toolkit
```

Es importante instalar la extensión de navegador de Redux para poder visualizar el comportamiento de los stores de redux.





## Actions

Objetos literales que están compuestos por dos props.

- Type              nombre con el que se identifica la acción.
- payLoad            información que se quiere enviar al estado para que se actualice.

Ejemplo de un contador

- Type: "Incrementar"
- payLoad: +1

Esto será procesado por el **reducer**

## Action creators

Funciones que retornan los Actions. Permiten recibir el payLoad de forma dinámica. Muchas veces, se definen los Actions dentro del mismo Action creator.

```
const addTodoAction = {  
  type: 'todos/todoAdded',  
  payload: 'Buy milk'  
}
```

```
const addTodo = text => {  
  return {  
    type: 'todos/todoAdded',  
    payload: text  
  }  
}
```

## Reducer

Es una función que toma dos argumentos y se encarga de actualizar el estado.

- Estado previo
- Action

## Store

Donde se almacenan los estados globales de la aplicación.

Cuando la página se recarga, se reinician todos los datos del Store.



## Configuración inicial

Dentro de la carpeta donde se ubicará nuestro redux, se crea un archivo store.js

```
(o: store.js)
import { configureStore } from '@reduxjs/toolkit'

export default configureStore({
  reducer: {}
})
```

src  
└── redux  
 └── reducers  
 └── store.js

Luego, en el index.js (o main.js) se importa el Store y el Provider de nuestro Redux para poder englobar toda la aplicación.

```
(o: main.jsx)
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";

// Redux
import store from './redux/store';
import { Provider } from "react-redux";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <Provider store={store}>
      <App/>
    </Provider>
  </React.StrictMode>
);
```

Finalmente, se pueden comenzar a crear los reducers, y “estados globales”

```
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  name: '',
  age: 0,
  live: true
}

export const userSlice = createSlice({
  name: 'user', // nombre del "estado global"
  initialState, // valor inicial del "estado global" ←
  reducers: { // funciones/métodos que el "estado global" puede usar
    addAge: (state) => {
      state.age += 1
    },
    die: (state) => {
      state.live = false
    },
    setUser: (state, action) => {
      state.name = action.payload.name
      state.age = action.payload.age
      state.live = action.payload.live
    },
  },
})
// Funciones / Métodos (reducers) de este estado
export const { addAge, die, setUser } = userSlice.actions
export default userSlice.reducer
```

Action creators



Tras crear un reducer (“estado global”), se lo debe guardar en el Store creado.

```
import { configureStore } from '@reduxjs/toolkit'

// Reducers
import userReducer from './reducers/user/userSlice'

export default configureStore({
    reducer: {
        user: userReducer
    }
})
```

nombre del “estado global”

## useSelector

Este Hook proporcionado por react-redux permite leer los datos del Store en un componente de React.

```
import { useSelector } from 'react-redux';

// JSX Component
export default (props) => {
    const { name, age, live } = useSelector(state => state.user);

    return (
        <p>Hola, soy {name} y tengo {age} años</p>
    )
}
```



## useDispatch

Este Hook proporcionado por react-redux permite actualizar datos del Store desde un componente de React.

```
import { useDispatch } from 'react-redux';

// Action creators
import { setUser } from '../../redux/reducers/user/userSlice';

// JSX Component
export default (props) => {
  const dispatch = useDispatch();

  const handleClick = () => {
    dispatch(setUser({ // setState global
      name: "Guille",
      age: 28,
      live: true
    }))
  }

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

The diagram shows a code snippet where a component uses the `useDispatch` hook to dispatch an action. The action is `setUser` with a payload object. An annotation labeled "payload" points to the payload object in the code. Another annotation labeled "initialState" points to the initial state object defined in the `userSlice`.

Para ejecutar una función que no requiere parámetros (payload), podemos usar el siguiente ejemplo:

```
import { useDispatch } from 'react-redux';

// Action creators
import { addAge } from '../../redux/reducers/user/userSlice';

// JSX Component
export default (props) => {
  const dispatch = useDispatch();

  const handleClick = () => {
    dispatch(addAge()) // setState global sin payload
  }

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



## Selector

Cuando se tiene un Slice con un array en el initialState, se pueden crear selectores para obtener sólo un elemento de dicho array.

Para ello, se crea una función que reciba el ID como parámetro y acceda state.

```
// Importa createSelector de 'reselect'  
import { createSelector } from 'reselect';  
  
// Crea un selector para obtener un elemento específico por su 'id'  
export const selectItemById = (itemId) =>  
  createSelector(  
    (state) => state.mySlice.items,  
    (items) => items.find((item) => item.id === itemId)  
  );
```

```
import { createSlice } from '@reduxjs/toolkit';  
  
const mySlice = createSlice({  
  name: 'mySlice',  
  initialState: {  
    items: [  
      { id: 1, name: 'Item 1' },  
      { id: 2, name: 'Item 2' },  
      { id: 3, name: 'Item 3' },  
    ],  
  },  
  reducers: {  
    // Tus otras acciones aquí...  
  },  
});  
  
export default mySlice.reducer;
```

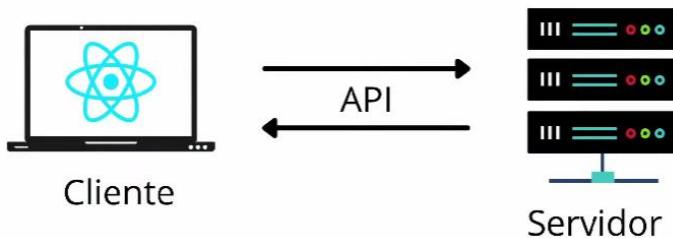
Con la función definida, se la puede trabajar desde el componente deseado.

```
import { useSelector } from 'react-redux';  
import { selectItemById } from './mySlice';  
  
function ItemDetail({ itemId }) {  
  const item = useSelector(selectItemById(itemId));  
  
  if (!item) {  
    return <div>Item not found</div>;  
  }  
  
  return (  
    <div>  
      <h2>{item.name}</h2>  
      /* Otros detalles del elemento */  
    </div>  
  );  
}
```



## React Query

Es una biblioteca que simplifica la gestión de datos obtenidas desde fuentes externas como APIs. Administra una caché automática para agilizar solicitudes repetidas.



### Instalación

Para usarse, se requiere instalar la librería **react-query** de NPM.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 10.0.19045.3324]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\xampp\htdocs\vite-project> npm install react-query
```

### Axios

En general, para solicitar datos a una API, se utiliza **fetch** o **axios** para obtener información del objeto en formato JSON.

```
export const getUsers = async () => {
  const { data } = await axios(`${API}/users`);
  return data;
};

export const getUser = async (id) => {
  const { data } = await axios.get(`${API}/users/${id}`);
  return data;
};
```

Y para cargar datos nuevos con la API, se usa el método **post**.

```
export const createUser = async (post) => {
  const { data } = await axios.post(`${API}/users`, post);
  return data;
};
```



## Ejemplo de uso sin React Query

Como ejemplo, se usará una “base de datos” simple basada en JSON-Server.

```
▼ src
  ▼ api
    ○; api.js
  ▼ component
    ▼ user
      ● User.jsx
      ○ User.module.css
    ▼ users
      ● Users.jsx
      ○ Users.module.css
  ○ App.css
  ● App.jsx
```

```
import User from './component/user/User' // component
import Users from './component/users/Users' // component

export default function App() {
  const [ userSelected, setUserSelected ] = useState(null)
  const deselectUser = () => setUserSelected(null)

  return userSelected ? // Se enviará el setter como parámetro
    <User id={userSelected} deselectUser={deselectUser}/> :
    <Users setUser={setUserSelected} />
}
```

● DataBase example

```
{
  "users": [
    {
      "id": 1,
      "image": "https://randomuser.me/api/portraits/men/1.jpg",
      "name": "Guillermo",
      "description": "Estudiante de programación",
      "followers": 3030,
      "follows": 4
    }
  ]
}
```

○; api.js

```
export const getUsers = async () => {
  const { data } = await axios(`$API}/users`);
  return data;
};
```

Teniendo la base, se prepara la carga de datos:

```
import { getUsers } from '../api/api'

export default function Users({ setUser }) {
  const [ error, setError ] = useState()
  const [ users, setUsers ] = useState()
  const [ isLoading, setIsLoading ] = useState(true)
  console.log("Users:", setUser)

  useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true)
      try {
        const data = await getUsers()
        setUsers(data)
        setError(null)
      } catch (error) {
        setUsers(null)
        setError(error)
      } finally {
        setIsLoading(false)
      }
    }

    fetchData()
  }, [])

  return (
    isLoading ? <p className={styles.loading}>Cargando...</p> :
    error ? <p className={styles.error}>{error}</p> :
    <section>
      <h2>Users:</h2>
      <ul className={styles.ul}>
        {users.map((user) => (
          <li key={user.id}>
            <a onClick={() => setUser(user.id)} href="#">
              {user.name}
            </a>
          </li>
        )))
      </ul>
    </section>
  )
}
```

se puede modularizar

○; apis

```
// FetchData
export default async (
  setData,
  setLoading,
  setError,
  fetchFunction,
  ...params ) => {
  setLoading(true)
  try {
    const data = await fetchFunction(params)
    setData(data)
    setError(null)
  } catch (error) {
    setData(null)
    setError(error)
  } finally {
    setLoading(false)
  }
}
```

```
import fetchData, { getUsers } from './api/api'

export default function Users({ setUser }) {
  const [ error, setError ] = useState()
  const [ users, setUsers ] = useState()
  const [ isLoading, setIsLoading ] = useState(true)

  useEffect(() => {
    fetchData(
      {
        setUsers,
        setIsLoading,
        setError,
        getUsers
      }
    ), []
  })
}
```



## Configuración de React Query

main.jsx

```
import { QueryClient, QueryClientProvider } from 'react-query'

const queryClient = new QueryClient();

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <QueryClientProvider client={queryClient}>
      <App />
    </QueryClientProvider>
  </React.StrictMode>,
)
```

Ahora se puede usar en nuestros componentes por medio de **useQuery** para hacer una petición a la API.

```
const { data:users, error, isLoading } = useQuery(["users"], getUsers)
```

clave que identifica de forma única a la consulta a realizar    ↙  
promise que devuelva los datos o un error en caso de problemas (axios, fetch, etc)    ↓

Así, se simplifica el código utilizado:

```
import { getUsers } from '../../../../../api/api'
import { useQuery } from 'react-query'

export default function Users({ setUser }) {

  const { data:users, error, isLoading } = useQuery(["users"], getUsers)

  return (
    isLoading ? <p className={styles.loading}>Cargando...</p> :
    error ? <p className={styles.error}>{error.message}</p> :
    <section>
      <h2>Users:</h2>
      <ul className={styles.ul}>
        {users.map((user) => (
          <li key={user.id}>
            <a onClick={() => setUser(user.id)} href="#">
              {user.name}
            </a>
          </li>
        ))}
      </ul>
    </section>
  )
}
```

Se realizarán 4 intentos antes de lanzar el resultado “error”



Al usar `useQuery`, éste devuelve un objeto con los siguientes datos:

```
data: undefined
dataUpdatedAt: 0
error: null
errorUpdateCount: 0
errorUpdatedAt: 0
failureCount: 0
isError: false
isFetched: false
isFetchedAfterMount: false
isFetching: true
isIdle: false
isLoading: true
isLoadingError: false
isPlaceholderData: false
isPreviousData: false
isRefetchError: false
isRefetching: false
isStale: true
 isSuccess: false
▶ refetch: f ()
▶ remove: f ()
status: "loading"
```

## Refetch

Al usar `useQuery`, los datos obtenidos se guardan en caché y, la próxima vez que se ejecute la misma query (con el mismo identificador '`users`'), se mostrarán los datos de la caché.

Al mismo tiempo se realizará una revalidación. Es decir, se hará una nueva petición a la API para actualizar los datos mostrados desde caché en tiempo real.

Se sabe si se está realizando esta operación con `isFetching`.

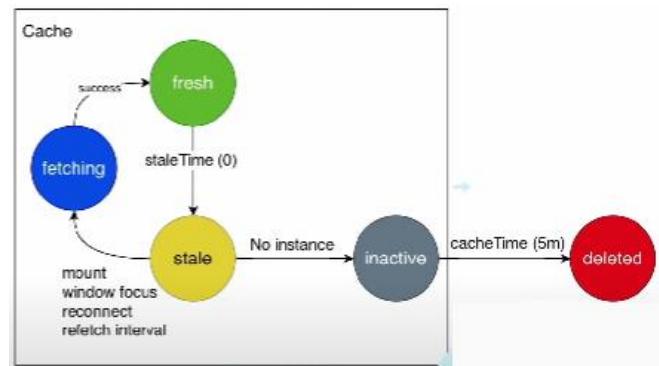
```
ading, isFetching} = useQuery
```

Esto se realiza tanto cuando se sale y vuelve a poner el foco en la página o pestaña web, y cuando se renderiza de nuevo el componente.

Esto se puede configurar con un tercer parámetro en el `useQuery`.

```
} = useQuery(["posts"], getPosts, {
  refetch
  refetchInterval... (property) refetchInterval
  refetchIntervalInBackground?
  refetchOnMount?
  refetchOnReconnect?
  refetchOnWindowFocus?
```

Se pueden desactivar



```
} = useQuery(["posts"], getPosts, {
  refetchOnWindowFocus: false
})
```

Además, se puede configurar para que se actualicen los datos cada cierto intervalo de tiempo sin necesidad de que el usuario realice ninguna acción.

`refetchInterval: 2000` ms

Otra configuración posible es la de establecer el tiempo en que los datos se considerarán como actualizados (0 por defecto). `staleTime: 60000` ms

Con `staleTime: Infinity` se define que no se actualicen los datos.

La información en caché se borrará por defecto luego de 5 minutos de dejar de mostrarse el componente donde fue cargado. `cacheTime: 3000` ms



## Configuraciones globales

Las configuraciones de Refetch se pueden realizar directamente en la creación del QueryClient en el index de la app.

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: Infinity,
      retry: 1, // reintentos de petición
      retryDelay: 1000 // ms entre reintentos
    }
  }
});
```

## Idle

Se pueden deshabilitar las peticiones automáticas que realiza useQuery al cargarse el componente y hacerlo “manualmente”.

```
, isIdle, refetch } = useQuery(["users"], getUsers, { enabled: false })
if (isIdle) return <button onClick={refetch}>Realizar petición</button>
```

La función “refetch” es la que reanuda las peticiones a la API.

## useQueryClient

Permite acceder a los datos que se encuentran almacenados en caché.

```
export default function Users({ setUser }) {

  const queryClient = useQueryClient();
  const { data: users, error, isLoading } = useQuery(["users"], getUsers)

  if (queryClient.getQueryData(["user", 1]))
    console.log("La info del usuario 1 está en caché")

  return (
    <table>
      <thead>
        <tr>
          <th>Nombre</th>
          <th>Apellido</th>
        </tr>
      </thead>
      <tbody>
        {users.map(user => (
          <tr key={user.id}>
            <td>{user.firstName}</td>
            <td>{user.lastName}</td>
          </tr>
        ))}
      </tbody>
    </table>
  )
}

export default function User({id, deselectUser}) {

  const { data: user, error, isLoading } = useQuery(["user", id], () => getUser(id))

  return (
    <div>
      <h3>{user.firstName} {user.lastName}</h3>
      <button onClick={deselectUser}>Deselección</button>
    </div>
  )
}
```

En este caso, al cargar los datos del user 1, dichos datos se almacenarán en caché. Al volver al componente Users, detectará que ya existen datos del user 1 en caché.



## useMutation

0: api.js

Permite la manipulación de los datos almacenados en caché, útil para realizar un **post** a la base de datos con nuevos datos.

```
export const createUser = async (post) => {
  const { data } = await axios.post(`${API}/users`, post);
  return data;
};
```

```
const { mutate, error, isLoading, isSuccess } = useMutation(createUser)

// Submit form
const handleSubmit = async (e, name, message) => {
  //e.preventDefault(); // prevents the page reload
  mutate({ name, message });
}
```

En este caso, el parámetro **post** es el objeto:

```
▶ {name: '123', message: '123'}
```

Además, es posible ejecutar callbacks en caso de que sucedan ciertos eventos.

```
const queryClient = useQueryClient();
const { mutate, error, isLoading, isSuccess } = useMutation(createUser, {
  onError: () => console.error("No se pudo guardar los datos."),
  onSuccess: () => {
    console.log("Datos guardados.")
    queryClient.invalidateQueries(["users"]) // Refetch
  }
})
```

Estos callbacks también pueden definirse en la función **mutate** como segundo parámetro. Es útil para cambiar estados locales.

```
mutate({ name, message },
{
  onSuccess: () => { /* setStates */ }
});
```

Para resetear todos los estados del useMutation, es posible utilizar la función **reset**.

```
s, reset } = useMutation(
```

Se pueden añadir datos en la caché con su id específica para agilizar el proceso de refetch para el usuario.

```
const queryClient = useQueryClient();
const { mutate, error, isLoading, isSuccess } = useMutation(createUser, {
  onError: () => console.error("No se pudo guardar los datos."),
  onSuccess: (user) => {
    console.log("Datos guardados: ", user)
    queryClient.setQueryData(["users"], prevUsers => {
      prevUsers.concat(user) // Agrega el nuevo user a La caché
    });
    queryClient.invalidateQueries(["users"]) // Refetch
  }
})
```



## Hook personalizado

La declaración del useMutation es bastante extensa en el código y puede modularizarse creando un nuevo hook.

```
import { useMutation, useQueryClient } from "react-query";
import { createUser } from '../api/api'

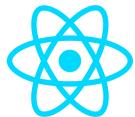
export const useMutateUsers = () => {
  const queryClient = useQueryClient()

  return useMutation(createUser, {
    onError: () => console.error("No se pudo guardar los datos."),
    onSuccess: (user) => {
      console.log("Datos guardados: ", user)
      queryClient.setQueryData(["users"], prevUsers => {
        prevUsers.concat(user) // Agrega el nuevo user a la caché
      });
      queryClient.invalidateQueries(["users"]) // Refetch
    }
  })
}
```

```
const { mutate, error, isLoading, isSuccess } = useMutatePost() ←

// Submit form
const handleSubmit = async (e, name, message) => {
  //e.preventDefault(); // prevents the page reload

  mutate({ name, message },
  {
    onSuccess: () => { /* setStates */ }
  });
}
```



## Build

Todo el código creado en React se hace en un entorno de trabajo, pero este no es el que debe publicarse en un servidor.

Para generar el código que puede publicarse en un servidor se debe ejecutar el comando **npm run build**.

## Carpeta PUBLIC

Dentro del proyecto de React existe una carpeta llamada **public**.

Esta carpeta se “construirá” tal cual como está en el **index** del proyecto.

Es útil especialmente para guardar archivos de imagen que pueden accederse desde el código y que este pueda variar. Así como archivos de código y otros.

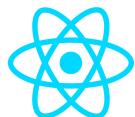
## XAMPP

Por lo general, en React se utiliza Express de NodeJS para construir el servidor, pero si se quiere usar XAMPP, se debe publicar el contenido de **dist** (donde se “construye” el proyecto) en la carpeta **htdocs**.



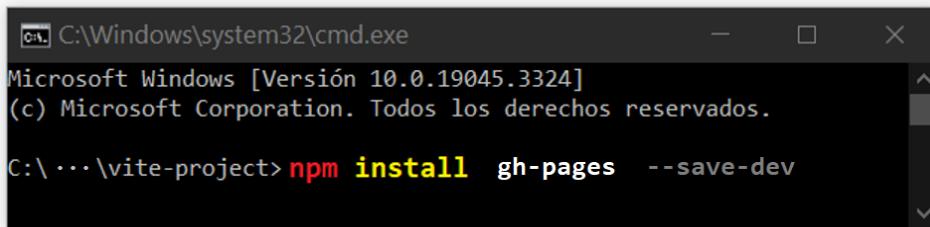
Además, si se usa **react-router**, es necesario configurarlo para que pueda reconocer las rutas alternativas. Se debe crear un archivo en **htdocs** llamado **.htaccess** con el siguiente código de PERL:

```
RewriteEngine On  
RewriteBase /  
RewriteRule ^index\.html$ - [L]  
RewriteCond %{REQUEST_FILENAME} !-f  
RewriteCond %{REQUEST_FILENAME} !-d  
RewriteRule . /index.html [L]
```



## GitHub Pages

Se puede publicar un proyecto construído con React en GitHub, desde un repositorio. Para esto, es requerido instalar un package.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 10.0.19045.3324]
(c) Microsoft Corporation. Todos los derechos reservados.

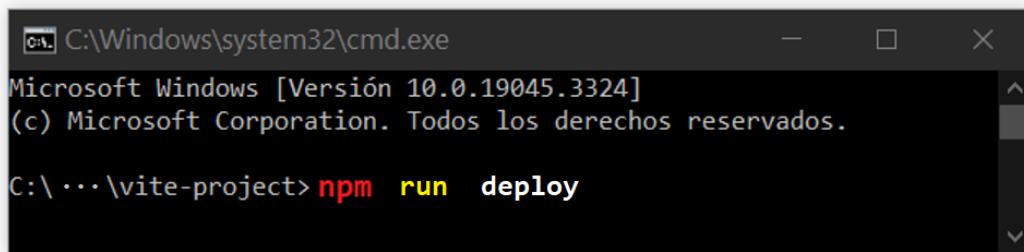
C:\...\vite-project> npm install gh-pages --save-dev
```

Luego, será necesario modificar el archivo **package.json** del proyecto, agregando las siguientes líneas:



```
type: "module",
"homepage": "https://USUARIO.github.io/NOMBRE-REPOSITORIO", ←
"scripts": {
  "predeploy": "npm run build",
  "deploy": "gh-pages -d dist -r https://github.com/USUARIO/NOMBRE-REPOSITORIO.git", ←
  "dev": "vite"
```

Finalmente, se procede a ejecutar el comando requerido para publicar la página.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 10.0.19045.3324]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\...\vite-project> npm run deploy
```

Así ya podrás acceder a la página con la URL

<https://USUARIO.github.io/NOMBRE-REPOSITORIO>

Si se está usando **react-routes** con varias páginas, puede ser necesario crear un archivo html en el root del proyecto llamado **404.html** que redirija al usuario al **home** de la app. Esto es porque, al recargar la página estando en una subruta, Github Pages no reconocerá la ruta como válida y ejecutará el error 404.

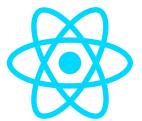
```
<meta http-equiv="refresh" content="0;URL=https://USUARIO.github.io/NOMBRE-REPOSITORIO/">
```

En caso de estar usando Vite, será necesario configurarlo adecuadamente.



```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [
    react(),
  ],
  base: '/NOMBRE-REPOSITORIO/',
})
```



# Modularización

Un proyecto se debe modularizar para un mejor entendimiento de cualquier programador que visualice el código.

## Estructura por tipo de fichero

Diferenciación entre componentes compartidos y páginas. Generalmente utilizada en proyectos pequeños.



## Estructura por funcionalidad

La idea principal es que cada módulo que definimos tenga todo el código relacionado con este y solo se importe código del módulo en sí. No se puede compartir código de módulos distintos.

```
src/
|-- components/
|   |-- Avatar/
|   |   |-- Avatar.jsx
|   |   |-- Avatar.test.js
|   |-- Button/
|   |   |-- Button.jsx
|   |   |-- Button.test.js
|   |-- TextField/
|   |   |-- TextField.jsx
|   |   |-- TextField.test.js
|-- contexts/
|   |-- UserContext/
|   |   |-- UserContext.js
|-- hooks/
|   |-- useMediaQuery/
|   |   |-- useMediaQuery.js
|-- features/
|   |-- Home/
|   |   |-- components/
|   |   |   |-- SomeUserProfileComponent/
|   |   |   |   |-- SomeUserProfileComponent.jsx
|   |   |   |   |-- SomeUserProfileComponent.test.js
|   |   |-- utils/
|   |   |-- services/
|   |   |-- hooks/
|   |   |-- contexts/
|   |   |-- views/
|   |   |   |-- HomeView.jsx
|   |   |-- pages/
|   |   |   |-- HomePage.jsx
|   |   |-- index.js
|-- utils/
|   |-- some-common-util/
|   |   |-- index.js/
|   |   |-- index.test.js
|-- services/
|   |-- some-common-service/
|   |   |-- index.js/
|   |   |-- some-common-service.js/
|   |   |-- index.test.js
|-- App.jsx
|-- index.js
```

```
src/
|-- components/
|   |-- Avatar/
|   |   |-- Avatar.jsx
|   |   |-- Avatar.test.js
|   |-- Button/
|   |   |-- Button.jsx
|   |   |-- Button.test.js
|   |-- TextField/
|   |   |-- TextField.jsx
|   |   |-- TextField.test.js
|-- contexts/
|   |-- UserContext/
|   |   |-- UserContext.js
|-- hooks/
|   |-- useMediaQuery/
|   |   |-- useMediaQuery.js
|-- pages/
|   |-- UserProfile/
|   |   |-- components/
|   |   |   |-- SomeUserProfileComponent/
|   |   |   |   |-- SomeUserProfileComponent.jsx
|   |   |   |   |-- SomeUserProfileComponent.test.js
|   |   |-- UserProfile.jsx
|   |   |-- UserProfile.test.js
|   |-- index.js
|-- routes/
|   |-- routes.jsx
|   |-- routes.test.js
|-- utils/
|   |-- some-util/
|   |   |-- index.js
|   |   |-- someUtil.js
|   |   |-- index.test.js
|-- services/
|   |-- some-service/
|   |   |-- index.js
|   |   |-- someService.js/
|   |   |-- index.test.js
|-- App.jsx
|-- index.js
```



## React Native

Framework que permite la creación de aplicaciones android, ios y desktop utilizando código JavaScript y React, el cual realiza un “bridge” con el código nativo de los respectivos sistemas operativos para desplegar la aplicación.

Para comenzar a programar en React Native es necesario tener **NodeJS**.

### Expo

Permite la visualización de un dispositivo virtual para poder ejecutar el programa de React Native de forma sencilla.

Para instalarlo, es necesario abrir la consola y ejecutar los siguientes comandos:

```
C:\Users\Guill> npm install -g expo-cli
ESPERAR INSTALACIÓN...
C:\Users\Guill> expo init nombre-sin-espacios
```

Una vez finalizado el proceso, se preguntará por el tipo de proyecto que se desea crear. En este caso, crearemos un proyecto en blanco para comenzar desde el inicio.

Finalmente se creará el directorio con el nombre del proyecto indicado.

```
C:\Windows\system32\cmd.exe
Choose a template: » blank
Downloaded template.
Using npm to install packages.
Installed JavaScript dependencies.

Your project is ready!

To run your project, navigate to the directory and run one of the following npm commands.

- cd react-native-app
- npm start # you can open iOS, Android, or web from here, or run them directly with the commands below.
- npm run android
- npm run ios # requires an iOS device or macOS for access to an iOS simulator
- npm run web

C:\xampp\htdocs>
```



Usando el comando **npm start**, se iniciará el localhost para la app.

Puede ser necesario permitir el acceso en windows defender

The screenshot shows the Windows Defender Firewall settings. On the left, there's a sidebar with links like 'Ventana principal del Panel de control', 'Permitir que una aplicación o una característica a través de Firewall de Windows Defender', 'Cambiar la configuración de notificaciones', 'Activar o desactivar el Firewall de Windows Defender', 'Restaurar valores predeterminados', and 'Configuración avanzada'. A red arrow points to the 'Configuración avanzada' link. On the right, there's a main panel titled 'Ayudar a proteger el equipo con Firewall de Windows Defender' with sections for 'Redes privadas' and 'Estado de Firewall de Windows Defender'. Below that is a table with rows for 'Conexiones entrantes', 'Redes privadas activas', and 'Estado de notificaciones'. To the right of the main panel is another window titled 'Windows Defender Firewall con seguridad avanzada' showing a list of 'Reglas de entrada' with two entries: 'Node.js JavaScript Runtime' and 'Node.js JavaScript Runtime' under 'Supervisión'.

Una vez iniciado, descargar la APP de Expo en el dispositivo móvil y escanear el código QR que aparecerá en la consola.

The screenshot shows a terminal window titled 'C:\Windows\system32\cmd.exe'. It displays a QR code in the center. Below the QR code, the terminal output includes the following text:  
> Metro waiting on exp://127.0.0.1:8081  
> Scan the QR code above with Expo Go (Android) or the Camera app (iOS)  
> Using Expo Go  
> Press s | switch to development build  
> Press a | open Android  
> Press w | open web  
> Press j | open debugger  
> Press r | reload app  
> Press m | toggle menu  
> Press o | open project code in your editor  
> Press ? | show all commands  
Logs for your project will appear below. Press Ctrl+C to exit.

De esta forma, se accede a la app del proyecto con el dispositivo móvil.



## Widgets

En el desarrollo para dispositivos móviles, varias etiquetas que normalmente son usadas en desarrollo web son reemplazadas por otras.

- **<View>** es el equivalente a **<div>**
- **<Text>** es el equivalente a **<p>**
- **<Separator>** es el equivalente a **<hr>**
- **<TextInput>** es el equivalente a **<input>**
- **<Button>** es el equivalente a **<button>**
- **<Image>** es el equivalente a **<img>**
- **<ScrollView>** crea un bloque con barra de scroll
- **<Switch>** es el equivalente a un radio button

Estas etiquetas deben importarse desde **react-native**.

Además, los estilos ya no pueden estar en archivos CSS, sino que pasan a estar escritos en objetos **StyleSheet**.

```
JS App.js
import { StyleSheet, Text, View } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.paragraph}>Hello world</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  paragraph: {
    fontSize: 30,
  },
});
```





## Imágenes

Permite agregar imágenes locales (almacenadas en **assets**) o de internet.

```
import { StyleSheet, Text, View, Image } from 'react-native';
import logo from './assets/enel-logo.webp'; // importar imagen Local

export default function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.paragraph}>Hello world</Text>

      <Image /* Imagen de internet */ 
        source={{uri:'https://picsum.photos/200/200'}}
        style={{height:200, width:200}}
      />

      <Image /* Imagen Local (assets) */ 
        source={logo}
        style={{height:200, width:200}}
      />

    </View>
  );
}
```

Puede usarse StyleSheet para los estilos.



## Botones

Existen diversos tipos de botones.

- ✓ <button/> poco personalizable

```
<Button
  onPress={() => Alert.alert("Soy un dialog")}
  title="Nombre"
  color="red" → background
  accessibilityLabel="Descripción de la función"
/>
```

- ✓ <TouchableOpacity> se comporta como un View / Div que ejecuta una función al ser clickeado y posee capacidad de recibir estilos.

```
import { StyleSheet, Text, View, Alert, TouchableOpacity } from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.paragraph}>Hello world</Text>

      <TouchableOpacity
        onPress={() => Alert.alert("Soy un dialog")}
        style={buttonStyle.container} >
        <Text style={buttonStyle.text}>Texto del botón</Text>
      </TouchableOpacity>

    </View>
  );
}

const buttonStyle = StyleSheet.create({
  container: {
    backgroundColor: 'blue',
    padding: 7,
    marginTop: 10,
    borderRadius: 10
  },
  text: {
    color: '#fff',
    fontSize: 20
  }
})
```



## Explorar galería

React-Native no posee por defecto la posibilidad de explorar la galería de un dispositivo móvil.

Es necesario instalar una librería de terceros para hacerlo.

Desde Expo, se puede instalar por consola: `expo install expo-image-picker`

VER 0:57



**Share**



## **Splash Screen & Icon**

1:31

Los íconos deben estar en formato png.



## SharedPreferences

Permite el almacenado de pequeña información (como configuraciones) en la memoria del dispositivo para que pueda ser utilizada al iniciar la app.

**npm install @react-native-async-storage/async-storage**

Luego, en código ya se podrá utilizar.

```
import AsyncStorage from '@react-native-async-storage/async-storage';

export default class SharedPreferences {
    static create = (key, value) => new Promise(async(response, reject) => {
        try { await AsyncStorage.setItem(key, value); response(true); }
        catch(err) { reject(err); }
    });

    static get = (key, defaultValue=undefined) => new Promise(async(response, reject) => {
        try {
            const value = await AsyncStorage.getItem(key);
            if (value !== null) response(value);
            else response(defaultValue ? defaultValue : null);
        } catch(err) { reject(err); }
    });

    static remove = (key) => new Promise(async(response, reject) => {
        try { await AsyncStorage.removeItem(key); response(true); }
        catch(err) { reject(false); }
    });
}
```



## Deploy

Compilación el código para ser usado tanto en Web, como en Android y IOs.

Con Expo, se puede hacer con el siguiente comando en consola: `expo build:android`

Será necesario tener una cuenta en la página de Expo para esta función.