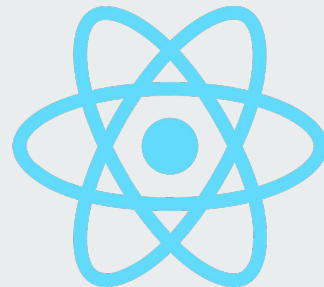




UD-6. REACTJS: UN FRAMEWORK DE JAVASCRIPT.

Una biblioteca de JavaScript para construir interfaces de usuario.





Glosario.

Componente. La programación orientada a componentes es un tipo de programación en la que los sistemas se descomponen en componentes funcionales o presentacionales (relativos a la interfaz). El nivel de abstracción es más alto que el de los objetos.

Callback. Llamada que hace un componente hijo a un método de un componente padre. Es la manera que tiene un componente hijo de comunicarse con su componente padre.

Elemento. Pequeña porción de código en una aplicación React.

JSX. Extensión de JavaScript utilizada para programar componentes React.

Props. Sistema de comunicación entre componentes que utiliza React. Los componentes padre envían información a los componentes hijos a través de las props.

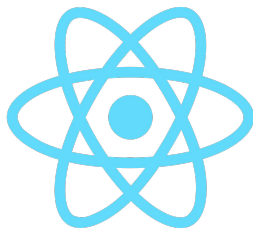


Glosario.

Renderizar. Proceso por el cual el componente se dibuja en el interfaz (navegador).

State. Estado de un componente donde se almacena la información relativa a este. Al igual que los objetos tienen atributos, un componente tiene estado.

Virtual DOM. DOM virtual utilizado por React. Al utilizar un DOM virtual los cambios en la interfaz son más ágiles y rápidos.





6.1. Introducción a React.

ReactJS es un *framework* o librería que permite a los programadores realizar aplicaciones web utilizando programación declarativa.

Las aplicaciones React están formadas por componentes y React es capaz de renderizar dichos componentes.

React utiliza programación orientada a componentes. Estos componentes gestionan su propio estado, son autónomos y su reutilización es muy alta y sencilla.



6.1. Introducción a React.

React utiliza una sintaxis parecida al XML llamada JSX. Podemos decir que: ***JSX = JS + XML***.

React fue diseñado por Jordan Walke y su primera versión se utilizó en las newsfeed de Facebook en el 2011. Este framework se diseñó para que fuese rápido, simple y escalable.



6.1. Introducción a React.

Resumiendo:

- Es una **librería de JavaScript** para crear interfaces de usuario.
- Utiliza **programación declarativa** y no imperativa como puede ser Java.
- Utiliza **programación basada en componentes** (component-based).
- Interactúa bien con cualquier arquitectura. No obliga a utilizar una arquitectura determinada (Technology stack agnostic).



6.1.1. Creación de la primera aplicación React.

Vamos a seguir los pasos del libro de texto *“Desarrollo Web en Entorno Cliente”* (Juan Carlos Moreno Pérez, Edt.Síntesis), en la página 166.

Antes de comenzar deberemos instalar **Node.js** para poder trabajar con React:

1. `apt install nodejs`
2. `apt install npm`



6.1.1. Creación de la primera aplicación React.

También debemos instalar **YARN**, que es una herramienta para gestionar paquetes de aplicaciones **Node.js**. Existen varios métodos para instalar Yarn, pero uno muy directo sería utilizando NPM:

```
3. npm install yarn -g
```

```
4. npm install -g create-react-app
```

```
5. create-react-app introduccion
```




6.1.1. Creación de la primera aplicación React.

Una vez creada la aplicación, ahora debemos ejecutarla, para ello escribimos el siguiente comando:

```
6. npm start o yarn start.
```

Una vez ejecutado el comando, se abrirá el navegador por defecto y se ejecutará la aplicación.

A continuación vamos a analizar los archivos más importantes que se han creado a partir de la aplicación.



6.2. Análisis del contenido de la primera aplicación.

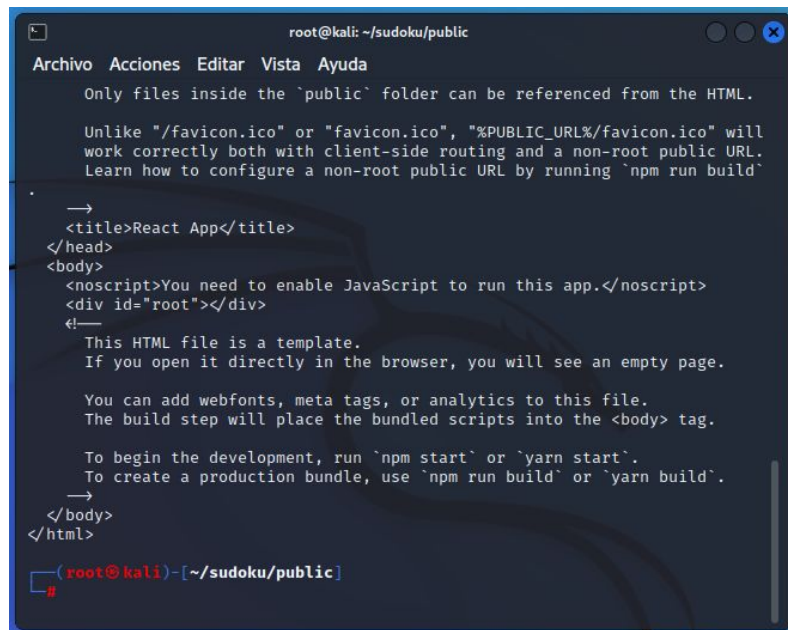
Los archivos más importantes de la aplicación son los siguientes:

- El archivo HTML *public/index.html*.
- El archivo de script `src/App.js`.
- El fichero de script `src/index.js`

Los componentes React comienzan con letra mayúscula y el resto del nombre escrito en minúsculas.

Las etiquetas HTML deberán estar en minúsculas en el código JSX.

6.2.1. El fichero index.html.



The screenshot shows a code editor window titled "root@kali: ~/sudoku/public". The editor displays the content of the "index.html" file. At the top, there is a menu bar with "Archivo", "Acciones", "Editar", "Vista", and "Ayuda". Below the menu, there is a warning message: "Only files inside the 'public' folder can be referenced from the HTML. Unlike '/favicon.ico' or 'favicon.ico', '%PUBLIC_URL%/favicon.ico' will work correctly both with client-side routing and a non-root public URL. Learn how to configure a non-root public URL by running 'npm run build'". The main content of the file is an HTML template with a title "React App", a body containing a noscript message and a root div, and a footer with instructions on how to run the application. The file ends with the closing tags for the body and html elements. The editor's status bar at the bottom shows the current file path: "(root@kali)-[~/sudoku/public]".

```
root@kali: ~/sudoku/public
Archivo Acciones Editar Vista Ayuda
Only files inside the `public` folder can be referenced from the HTML.
Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
work correctly both with client-side routing and a non-root public URL.
Learn how to configure a non-root public URL by running `npm run build`
.
  →
  <title>React App</title>
</head>
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.

    You can add webfonts, meta tags, or analytics to this file.
    The build step will place the bundled scripts into the <body> tag.

    To begin the development, run `npm start` or `yarn start`.
    To create a production bundle, use `npm run build` or `yarn build`.
  -->
</body>
</html>
(root@kali)-[~/sudoku/public]
```



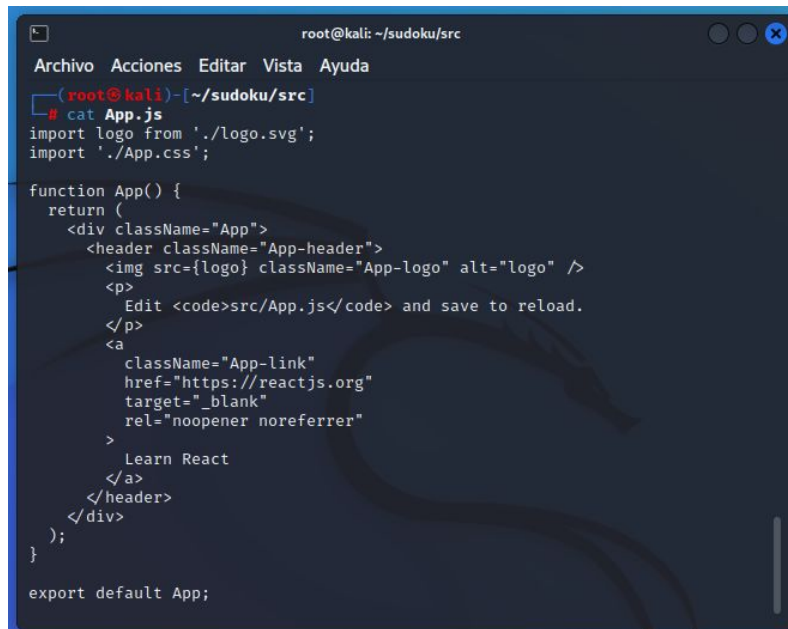
6.2.1. El fichero index.html.

La siguiente línea es la más interesante de este archivo:

```
<div id="root"></div>
```

Aquí vemos que el elemento `<div>` nos indica dónde se va a renderizar la aplicación de React, en el root.

6.2.2. El fichero src/App.js.

A terminal window with a dark blue background and light blue text. The window title is 'root@kali: ~/sudoku/src'. The terminal shows the command 'cat App.js' and its output, which is the content of the App.js file. The code includes imports for 'logo.svg' and 'App.css', a function 'App()' that returns a JSX element, and an 'export default App;' statement. The JSX element has a 'div' with 'className="App"' containing a 'header' with 'className="App-header"'. Inside the header, there is an 'img' tag for the logo, a paragraph with instructions to edit the file and save to reload, and a link to 'https://reactjs.org' with 'className="App-link"'.

```
root@kali: ~/sudoku/src
Archivo Acciones Editar Vista Ayuda
(root@kali)~[~/sudoku/src]
$ cat App.js
import logo from './logo.svg';
import './App.css';

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

export default App;
```



6.2.2. El fichero `src/App.js`.

Este archivo es donde residirá la **lógica de la aplicación**. Es importante saber que **App** extiende **Component**, luego la aplicación tendrá un componente principal **React** llamado **App**.

El método `render()` en un componente React es el encargado de mostrar el componente en la interfaz web.

La estructura visual de la aplicación está en el método `render()`, pues es el que devuelve el contenido HTML y el que dibuja el componente de la aplicación. El componente **App** se exporta para poder ser utilizado después por otras partes de la aplicación:

```
export default App;
```

6.2.3. El fichero index.js.

```
(root@kali)-[~/sudoku/src]
# cat index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

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

// 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();
```



6.2.3. El fichero index.js.

En este fichero, se **importan los distintos elementos** de los que consta la aplicación y que son imprescindibles para ella. Crea el enlace entre la aplicación y el elemento **root** definido en ***index.html*** mediante la siguiente línea:

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

En esta orden renderiza la aplicación en el elemento de index.html con el “id=root”.



6.3. Introducción a JSX.

Es una extensión de JavaScript.

JSX genera elementos React (muchos elementos HTML tienen una correspondencia con un elemento React).

Una de las ventajas que tiene JSX es qué está **íntimamente ligado con el manejo de eventos**, con cómo el estado de los componentes cambian durante su vida y cómo se formatean los datos para ser mostrados en la interfaz.

Los componentes de React contienen la lógica y la parte de interfaz en el mismo componente.

React no necesita utilizar JSX, pero es sumamente útil tener el HTML dentro del código JavaScript.

```
const element=(  
<h1 className="saludos">  
  Hola Mundo!  
</h1>  
);
```

JSX

```
const element={  
  type: 'h1',  
  props:{  
    className:"saludos",  
    children; "Hola, Mundo!"  
  }  
};
```

Es igual

```
const element =  
(React.createElement(  
  'h1',  
  {className:"saludos"},  
  'Hola, Mundo!'  
));
```

JavaScript

Genera



6.4. Componentes React.

- Las **interfaces** de usuario estarán formadas de **componentes** que son independientes uno de otros.
- Los **componentes** son piezas **reutilizables**.
- Un componente devolverá una serie de elementos que aparecerán en la **ventana del navegador**.
- En React, se pueden definir diferentes clases o tipos de componentes.
- Los componentes realizados por el programador deben tener la **primera letra en mayúsculas** obligatoriamente. Esos componentes serán compilados con el método *React.createElement(...)*.
- Las **etiquetas** que empiezan por minúsculas se tratarán como etiquetas del **DOM**.
- También hay que tener en cuenta que existen **componentes predefinidos**.



6.4.1. Class components y function components.

Cuando se crean interfaces en ReactJS, existe la posibilidad de crear un *class component* o un *function component*.

CLASS COMPONENTS => STATEFUL COMPONENTS (Tienen estado)

FUNCTION COMPONENTS => STATELESS COMPONENTS (No tiene estado).

Ambos componentes son similares en comportamiento, pero estructuralmente tienen diferencias. Vamos a ver un ejemplo, a partir de la primera aplicación creada anteriormente:

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header
        className="App-header">
        <img
          src={logo}
          className="App-logo"
          alt="logo"/>
        <p>HOLA</p>
      </header>
    </div>
  );
}

export default App;
```

```
import logo from './logo.svg';
import './App.css';
import {Component} from 'react'; ← 1

class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      saludo: "HOLA",
    };
  }
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo"
            alt="logo"/>
          <p>{this.state.saludo}</p>
        </header>
      </div>
    );
  }
}

export default App;
```

Diagram illustrating the mapping between Function Component and Class Component code:

- 1: `import {Component} from 'react';` (Class Component) corresponds to the `Component` base class in the `class App extends Component` declaration.
- 2: `function App() {` (Function Component) corresponds to the `class App` declaration (Class Component).
- 3: `return (` (Function Component) corresponds to the `constructor(props) {` block (Class Component).
- 4: `render() {` (Function Component) corresponds to the `render() {` block (Class Component).
- 5: `<p>{this.state.saludo}</p>` (Function Component) corresponds to `<p>{this.state.saludo}</p>` (Class Component).



6.4.1. Class components y function components.

1. En un componente de tipo clase, lo primero es **importar** la librería `"Component"`. Que es la principal de React.
2. En el encabezado, vemos claramente que uno empieza por `class` y el otro por `function`.
3. El componente clase tiene **constructor** y dentro se especifica el **estado**. Ésto es muy importante.
4. Los componentes de tipo clase tienen un método que se denomina `render()`.
5. El estado de los componentes de tipo clase, al tener estado, se pueden referenciar mediante una llamada del tipo `this.state`.



6.4.1. Class components y function components.

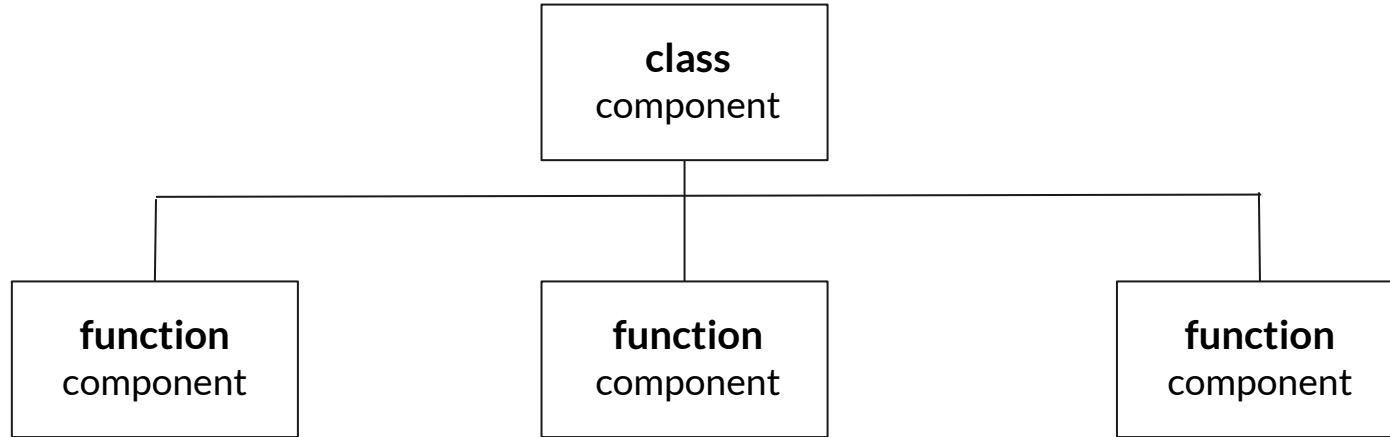
El método `render()` en un componente React es el encargado de **mostrar el componente** en la interfaz web.

La **estructura visual** de la aplicación está basada en el método `render()`.

El método `render` es el que devuelve el contenido HTML, el que **dibuja el componente** de la aplicación.

Es uno de los métodos más importantes, puesto que va a **definir o mostrar la funcionalidad** del componente.

6.4.1. Class components y function components.



Esquema de componentes en una aplicación.



6.4.1. Class components y function components.

En muchas aplicaciones pequeñas es normal encontrarnos con una estructura como la de la diapositiva anterior (24), donde tendremos un **componente principal de tipo clase** para almacenar la información de la aplicación, y una serie de componentes de tipo función, que serán hijos del componente principal. Éstas funciones hijas solamente muestran dicha información.

Esto no es que sea estrictamente así siempre, pero sí que normalmente componentes principales suelen ser de tipo clase para poder gestionar el estado de la aplicación.



6.4.2. Componentes contenedores y presentacionales.

En React los componentes se clasifican dependiendo de su uso o de cómo se utilizan, dando lugar a dos tipos principales: **contenedores** y **presentacionales**.

- **Los componentes *presentacionales*:**
 - Están orientados a renderizar la vista. Utilizan estilos predefinidos para mejorar la visualización.
 - Renderizan la vista dependiendo de los datos que se les proporcione.
 - No es necesario que gestionen su propio estado. No manejan datos, solamente los necesarios para realizar la presentación de la interfaz.



6.4.2. Componentes contenedores y presentacionales.

- **Los componentes *contenedores*:**
 - Responsables de que la aplicación **funcione correctamente**. Consultas a bases de datos, actualizar estado de los componentes, etc.
 - Realizan **llamadas a los componentes presentacionales** para mostrar la interfaz. También envían datos a esos componentes para que los muestren en el navegador.
 - Generalmente, los componentes **presentacionales** están embebidos, en forma de **<divs>**, en éstos componentes contenedores.
 - Responsables de comunicarse con la base de datos u otros orígenes de datos.



6.4.2. Componentes contenedores y presentacionales.

Flux y **Redux** son las tecnologías que utiliza Facebook para realizar sus interfaces con ReactJS

- Flux es un patrón de diseño, mientras que Redux (que es más complejo) es una librería.
- Flux es el nombre que se le da al patrón de diseño *observer*. Que se modificó levemente para utilizarlo en React.
- En Flux el almacenamiento de los componentes puede tener varias ubicaciones, mientras que en Redux el almacenamiento está centralizado. (Esta es una de las claves por las que al trabajar con React se utiliza Redux).

Este tipo de aplicaciones tendrán un **estado inicial** y va a ir **modificándose** en base a los eventos acaecidos.

6.4.3. Creando el componente principal. Conversor euro-dólares.

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

```
class App extends Component{
```

```
  constructor(props) {
```

```
    super(props);
```

```
    this.state={
```

```
      euros:0,
```

```
      factor:1.09,
```

```
    } }
```

Uso del 'state'

```
  render() {return (
```

```
    <div className="App">
```

```
      <header className="App-header">
```

```
        {this.state.euros}Euro<button>+</button><button>-</button>
```

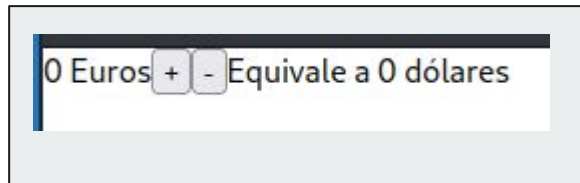
```
        Equivale a {this.state.euros*this.state.factor}dolares
```

```
      </header>
```

```
    </div>
```

```
  ); }
```

Usar llaves para que lo interprete como código y no como texto.



```
}export default App;
```



6.5. Los componentes. El *state*.

El **state** representa el estado del componente y las **props** (propiedades) son los parámetros que le pasa un componente padre a su componente subordinado o hijo.



6.5.1. El state y los métodos. Reglas:

1. Cada componente tiene su información local registrada en su estado (**state**). Sólo puede ser controlada por el componente, no puede ser accedida directamente por otro componente.
2. Ese estado se le puede pasar a los componentes hijos como **props**. De esa forma un ancestro (**ancestor**) puede pasar información a toda su descendencia (**descendant**).
3. Solamente los componentes de tipo clase (class components) pueden tener estado local (state).



6.5.1. El state y los métodos. Reglas:

A continuación vamos a actualizar el **state**, de la aplicación anterior, y veremos cómo evoluciona el programa.


```
import {Component} from 'react';
class App extends Component {
  constructor (props){
    super (props);
    this.state = {
      euros:0,
      factor:1.09,
    };
  }
  aumentar(){
    var auxeuro = this.state.euros+1;
    this.setState({euros:auxeuro});
  }
  disminuir(){
    var auxeuro = this.state.euros-1;
    if (auxeuro ≥ 0) this.setState({euros:auxeuro});
  }
  render(){
    return (
      <div className="App">
        <br/>
        {this.state.euros} euros <button onClick={()⇒this.aumentar()}> + </button>
        <button onClick={()⇒this.disminuir()}> - </button>
        Equivale a {Math.round(this.state.euros * this.state.factor*100)/100 } dólares<br/>
      </div>
    );
  }
}
export default App;
```



6.5.1. El state y los métodos. Reglas:

Cuando se modifica el estado con `setState()` la interfaz se renderizará automáticamente. Si no utilizas `setState()` y modificas el estado directamente, la interfaz no se renderiza de forma automática.

El estado (state) solamente puede ser modificado utilizando el método `setState()`.

NO debemos actualizar el componente así: `this.state.euros +=1;`

Hacerlo así, implica que la interfaz no se actualizará automáticamente.



6.5.2. Los componentes hijo y la comunicación padre-hijo: las props.

JSX pasa información de un componente a otro mediante **props**, las cuales son tratadas como un objeto individual. Puede accederse a esta información dentro del componente mediante sus **props** y se puede pasar más de una propiedad (props).

Un componente no puede modificar las propiedades. Si se quiere modificar la información que viene vía **props**, lo más lógico es copiar el contenido en el state del componente y luego modificar lo que haga falta.

Por **props** puedes enviar tanto variables como funciones. El envío de funciones es muy útil cuando quieres comunicar el hijo con el padre.

```

import {Component} from 'react';
class App extends Component {
  constructor (props){
    super (props);
    this.state = {
      euros:0,
      factor:1.09,
    };
  }
  aumentar(){
    var auxeuro = this.state.euros+1;
    this.setState({euros:auxeuro});
  }
  disminuir(){
    var auxeuro = this.state.euros-1;
    if (auxeuro ≥ 0) this.setState({euros:auxeuro});
  }
  render(){
    return (
      <div className="App">
        <br/>
        <button onClick={()⇒this.aumentar()}>+
        </button><button onClick={()⇒this.disminuir()}>-</button><br/>
        <Mostrar euros={this.state.euros} dolares={Math.round(this.state.euros*this.state.factor*100)/100}/>
      </div>
    );
  }
}

function Mostrar(props){
  return (
    <div>
      <h2>{props.euros} Euros equivalen a {props.dolares} Dólares</h2>
    </div>
  );
}

export default App;

```

Componente
App

props
euros
dolares

Componente
Mostrar



6.5.2. Los componentes hijo y la comunicación padre-hijo: las props.

Aunque en la aplicación anterior se han incluido dos componentes en un mismo fichero para una mejor comprensión, generalmente cada componente irá codificado en un fichero aparte. Por ejemplo, el componente Mostrar se ubicará en el fichero ***MostrarComponent.js***. Recuerda que el nombre de los componentes siempre empiezan por mayúsculas.

El componente **App** le pasa vía props dos parámetros (*euros* y *dolares*). El componente **Mostrar** lo podrá referenciar invocando *props.euros* y *props.dolares*.



6.5.2. Los componentes hijo y la comunicación padre-hijo: las props.

También hay que tener en cuenta que en el evento *onClick* del botón se le pasa el método `disminuir()` de la siguiente manera:

```
onClick = { () => this.disminuir ()}
```

```
onClick = {this.disminuir ()}
```



6.6. El DOM virtual de React.

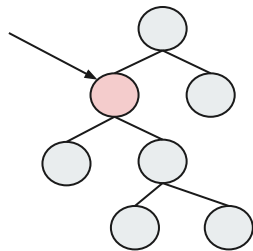
Como ya se ha indicado, el DOM (*Document Object Model*) es un objeto del navegador. Al igual que el navegador, React utiliza un objeto que mantiene un DOM virtual, que es una representación ligera del DOM del navegador.

React tiene mapeados los objetos del DOM a objetos virtuales suyos. El DOM virtual se almacena como un árbol en memoria donde cada nodo será un objeto que representará a un elemento. Al residir en memoria, este DOM virtual es muy rápido y las modificaciones del mismo son mucho más ágiles que las del DOM del navegador.

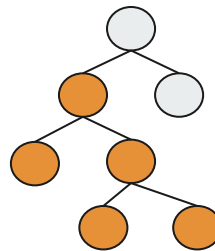
6.6.1. Renderización y cambios.

React tiene un algoritmo que detecta los nodos que cambian y se encarga de re-renderizarlos. Además, si cambia un nodo y se renderiza, los nodos descendientes tienen que renderizarse de nuevo, puesto que un ancestro ha cambiado y dichos cambios podrían afectar.

Cambia este
nodo



Renderización
del subárbol



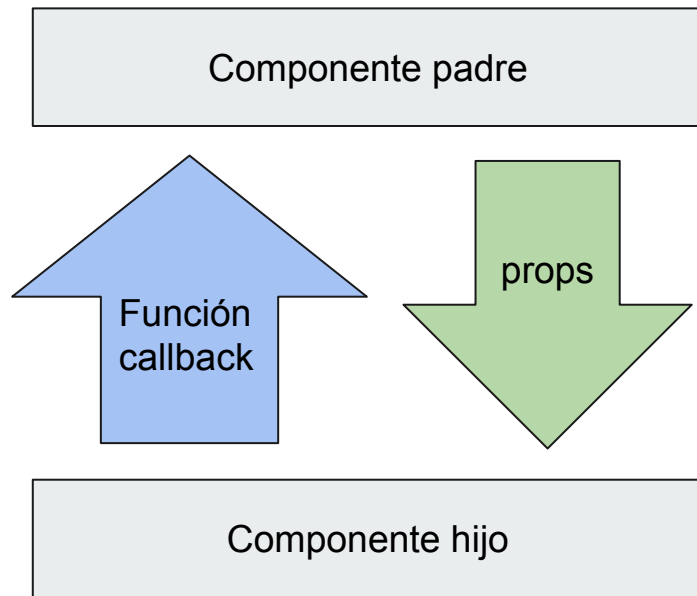


6.7. Los callbacks.

Un caso frecuente es cuando un componente necesita enviar una información a otro componente subordinado (hijo) y necesita cierta información de vuelta (feedback).

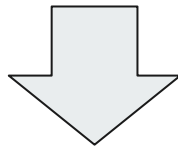
Este envío de vuelta de información (del hijo al padre) se denomina *callbacks* y es una función que se le pasa al componente subordinado (hijo) para que la ejecute con los datos del mismo (hijo). Esta función modificará el estado (state) del componente padre.

6.7. Los callbacks.



```
render() {
  return (
    <div className="App">
      <br/>
      <button onClick={()=>this.aumentar()}>+
    </button><button onClick={()=>this.disminuir()}>-</button><br/>
      <Mostrar euros={this.state.euros}
dolares={Math.round(this.state.euros*this.state.factor*100)/100}/>
    </div>
  )
}
```

Modificamos el código anterior para utilizar callbacks:



```
render() {
  return (
    <div className="App">
      <br/>
      {this.state.euros} euros
      <Botonera f1={()=> this.aumentar()} texto1={"+"} f2={()=> this.disminuir()}
texto2={"-"}/>
      Equivalen a {Math.round(this.state.euros*this.state.factor*100)/100} dólares<br/>
    </div>
  )
}
```

```
render() {
```

```
  return (
```

```
    <div className="App">
```

```
    <br/>
```

```
    {this.state.euros} euros
```

componente padre

```
    <Botonera f1={() => this.aumentar()} texto1={"+"} f2={() => this.disminuir()}
```

```
    texto2={"-"}/>
```

```
    Equivalen a {Math.round(this.state.euros*this.state.factor*100)/100} dólares<br/>
```

```
  </div>
```

```
);
```

```
}}
```

componente hijo

```
function Botonera(props) {
```

```
  return (
```

```
    <span>
```

```
      <button onClick={() => props.f1()}>{props.texto1}</button>
```

```
      <button onClick={() => props.f2()}>{props.texto2}</button>
```

```
    </span>
```

```
  );
```

```
export default App;
```



6.7. Los callbacks.

El componente **App** renderiza al componente hijo, y en esa llamada le pasa vía **props** cuatro parámetros, los dos textos de los botones y dos funciones **callback**, **aumentar()** y **disminuir()**, que se llamarán en el componente hijo Botonera **f1()** y **f2()**.

Cuando se llame a la función **f1()** se estará ejecutando en el padre la función **aumentar()**, y lo mismo ocurrirá en el caso de **f2()** y **disminuir()**.

La forma más común de pasar las funciones **callback** como parámetros es las siguientes:

```
f1 = { () => this.aumentar() }
```



6.7. Los callbacks.

Al enlazar o pasar una función mediante una **función flecha** estás creando una nueva función cada vez que el componente se renderiza. Utilizar la función flecha es la **forma más utilizada** y sencilla de enviar una función *callback* vía props.

Al enlazar o utilizar una **función flecha** te aseguras que el método que se va a invocar es realmente el del componente padre



6.8. Práctica guiada. Los reyes Godos.

Se requiere realizar una pequeña aplicación que muestre información sobre los reyes godos que dominaron la península ibérica.

Se proporcionará al programador un array en formato JSON, que podrá incorporar a su aplicación de la siguiente manera:



6.8. Práctica guiada. Los reyes Godos.

Leovigildo

Fue el rey visigodo más importante y admirado de la historia.

Atanagildo

Luchó contra Agila I para ser el rey lo cual supuso una crisis económica para el reino.

Suintila

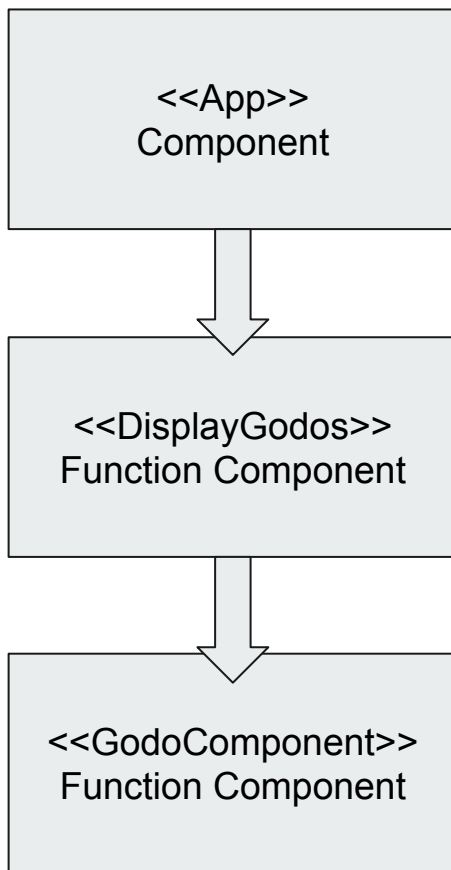
Derrotó a los vascones y dominó toda la península.

Recaredo

Hijo de Leovigildo que se convirtió al catolicismo.

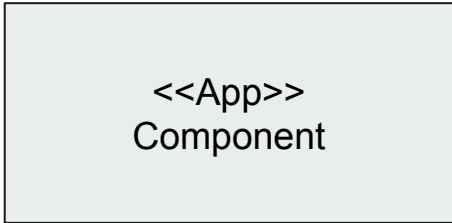

```
const GODOS = [  
  {  
    id:0,  
    nombre:"Leovigildo",  
    texto:"Fue el rey visigodo más importante y admirado de la historia."  
  },  
  {  
    id:1,  
    nombre:"Atanagildo",  
    texto:"Luchó contra Agila I para ser el rey lo cual supuso una crisis económica para el reino."  
  },  
  {  
    id:2,  
    nombre:"Suintila",  
    texto:"Derrotó a los vascones y dominó toda la península."  
  },  
  {  
    id:3,  
    nombre:"Recaredo",  
    texto:"Hijo de Leovigildo que se convirtió al catolicismo."  
  },  
]
```

Jerarquía de componentes de la aplicación:



```
import { Component } from 'react';
class App extends Component{
  constructor(props) {
    super(props);
    this.state = {
      godos: GODOS,
    };
  }
  render(){
    return (
      <div className="App">
        <header className='App-header'>
          <DisplayGodos reyes =
{this.state.godos}/>
        </header>
      </div>
    );
  }
}
```

El componente principal se llamará “**App**”, y será de tipo *class component*, puesto que va a tener en su estado la lista de los reyes godos.



El componente **App** hace una llamada al componente **DisplayGodos**, pasándole por props la lista de reyes godos.

```
function DisplayGodos (props) {  
  const losreyes = props.reyes;  
  const lista = losreyes.map((rey)=>  
    <GodoComponent key={rey.id.toString()}  
      titulo = {rey.nombre}  
      texto = {rey.texto}  
    />);  
  return(  
    <ul>  
      {lista}  
    </ul>  
  );  
}
```

<<DisplayGodos>>
Function Component

Al no tener que almacenar ningún dato, puede ser un *function component*.

El componente almacena dos datos, uno con la **lista de los reyes** pasada por props y otro en el que se va a generar un **array de componentes**.

El prop **key** se usa cuando se genera un número determinado de componentes iguales (en este caso GodoComponent).

Al listado de componentes se le pasan dos props, el **título** o nombre del rey y el **texto** que se desea que aparezca en el navegador.

```
const GodoComponent = (props)=>{  
  return(  
    <div>  
      <h1>{props.titulo}</h1>  
      <p>{props.texto}</p>  
    </div>  
  );  
};
```

Este será un componente presentacional de tipo función. Se ha cambiado la palabra *function* por una función flecha.

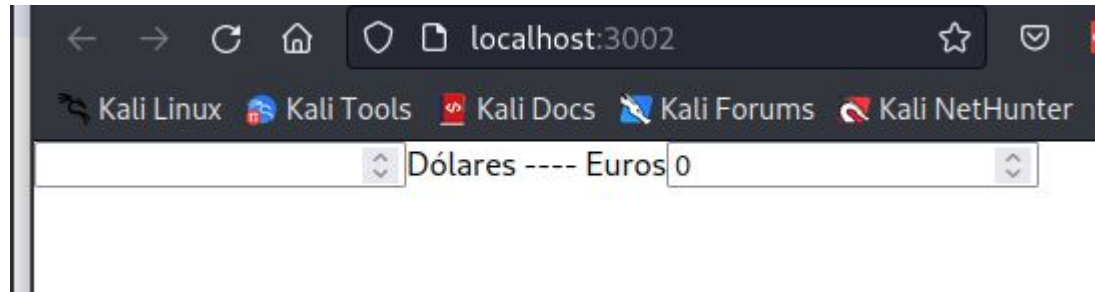
GodoComponent, será el encargado de visualizar el nombre y texto de un rey godo.

<<GodoComponent>>
Function Component

6.9. Práctica guiada. Conversor dólares a euros y viceversa.

Se requiere realizar un conversor de euros a dólares y viceversa. El cambio actual es $1\text{€} = 1,07\$$.

Debe tener éste aspecto:



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

class App extends Component{
  constructor(props) {
    super(props);
    this.state = {
      dolares: 0,
      euros: 0,
    };
  }
}
```

Este será el componente principal, que es muy parecido al del ejemplo, con la diferencia de que ahora los valores iniciales están a '0'

```
convertir (moneda, valor){  
    if (moneda === "dolar"){  
        this.setState({dolares:valor});  
        this.setState({euros:valor/1.07});  
    }  
    if (moneda === "euro"){  
  
this.setState({dolares:valor*1.07});  
        this.setState({euros:valor});  
    }  
}
```

En esta parte del código se asignan los estados a las variables según la conversión que queremos hacer.


```
render() {  
  return (  
    <div className="App">  
      <Moneda moneda="dolar" value={this.state.dolares}  
convertir={ (moneda, valor) => this.convertir(moneda, valor) } />  
      Dólares ---- Euros  
      <Moneda moneda="euro" value={this.state.euros}  
convertir={ (moneda, valor) => this.convertir(moneda, valor) } />  
    </div>  
  );  
}
```



Código utilizado para visualizar (renderizar) en el navegador el resultado de llamar al componente hijo **Moneda**.

Componente Hijo (MONEDA)

```
class Moneda extends Component {  
  handleChange (evt) {  
    this.props.convertir (this.props.moneda, evt.target.value);  
  }  
  render () {  
    return (  
      <input type="number"  
        value = {this.props.value}  
        onChange={this.handleChange.bind(this)}  
      />  
    );  
  }  
}
```

Declaración del componente hijo **Moneda**. Aquí se enlaza (bind) el evento **onChange** con la función **handleChange()**, la cual hace una llamada al método **convertir** enviado vía props. Este método o función tiene dos parámetros. El primero se llama “dolar” o “euro” para conocer el campo que ha sido modificado y el segundo el valor actual de dicho campo.

Este valor se recoge en el atributo **evt.target.value** de la función **handleChange()**. El objeto **evt** recoge el evento producido, el objeto **target** recoge el objeto que ha producido dicho evento y en **value** se recogerá el número del input utilizado.



MÁS APUNTES SOBRE REACT.

Algunos conceptos más sobre React, que no vienen en el libro.



1. Retornar elementos con Fragment.

A partir del código:

```
export const HelloWorldApp = () => {  
  return (  
    <h1>Hola Mundo</h1>  
  )  
}
```

Si quisiéramos devolver más de un elemento, podríamos hacerlo con un `<div>`, pero no sería del todo correcto. React tiene un componente llamado `Fragment` con el cual podemos devolver todos los elementos que queramos. Para ello, primero tendríamos que importarlo de la siguiente manera:



1. Retornar elementos con Fragment.

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

export const HelloWorldApp = () => {
  return (
    <Fragment>
      <h1>Hola Mundo</h1>
      <p>Soy un párrafo</p>
    </Fragment>
  )
}
```

Este código es equivalente:

```
<>
  <h1>HelloWordlApp</h1>
  <p>Soy un párrafo</p>
</>
```



2. Impresión de variables en HTML.

En React podemos manejar expresiones dentro de las etiquetas html, (no pueden ser un objeto), por ejemplo:

```
export const ImpresionVbles = () => {  
  return (  
    <>  
      <h1>{1+3}</h1>  
    </>  
  )  
}
```



2. Impresión de variables en HTML.

Otro ejemplo:

```
const getResult = (a,b) =>{
  return a+b;
}

export const ImpresionVbles = () => {
  return (
    <>
    <h1>{getResult(1,2)}</h1>
    <p>Soy un párrafo</p>
    </>
  )
}
```



3. Colocar estilos de CSS.

Para crear estilos dentro de mi código de React, deberemos irnos a la carpeta /src y crear un archivo que contenga todos los estilos, del tipo *styles.css*.

```
html, body{  
  background-color: #21232A;  
  color: white;  
  font-family: Arial, Helvetica, sans-serif;  
  font-size: 1.3rem;  
  padding: 70px;  
}
```




3. Colocar estilos de CSS.

Pero para que estos estilos se apliquen a todos los componentes que creemos en nuestro proyecto, deberemos importarlos en nuestro componente principal, en el *main.jsx*, de la siguiente forma:

```
import './styles.css';
```



4. Comunicación entre componentes (Props).

Los **props** van a ser las propiedades que le mandamos a nuestra función. Estas “*props*” son un objeto:

```
export const FirstApp = ({props}) => {  
  return (  
    <>  
    <h1>{props.title}</h1>  
    <p>Soy un párrafo</p>  
    </>  
  )  
}
```



4. Comunicación entre componentes (Props).

Normalmente las propiedades nos las encontraremos desestructuradas en la función, de esta forma:

```
export const FirstApp = ({title}) => {  
  return (  
    <>  
    <h1>{title}</h1>  
    <p>Soy un párrafo</p>  
    </>  
  )  
}
```



4. Comunicación entre componentes (Props).

En el componente padre, es decir, en el *main.jsx* tendremos:

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    <FirstApp title='Hola, Soy Gustavo' subTitle='123' />  
  </React.StrictMode>  
);
```

Estos “props” son los que nos van a permitir la comunicación entre el padre (main.jsx) y el componente hijo (FirstApp). Es información que fluye desde el componente padre hacia el componente hijo.



4. Comunicación entre componentes (Props).

En el caso de que desde el componente padre no tengamos valor en la propiedad “title”, podemos hacer que siempre devuelva algo nuestro componente hijo, así:

```
export const FirstApp = ({title = 'Hola Mundo'}) =>
{
  return (
    <>
      <h1>{title}</h1>
      <p>Soy un párrafo</p>
    </>
  )
}
```



5. PropTypes. Tipo de las propiedades (props).

Si hemos empezado a trabajar con **CRA** (*Create-React-App*) ya viene por defecto. Sin embargo en **Vite**, debemos instalarlo e importarlo de forma manual:

En el terminal, instalamos las prop-types: `% yarn add prop-types`

```
import PropTypes from 'prop-types';
```

Las *propTypes* nos va a permitir definir el tipo de la propiedades de los componentes:



5. PropTypes. Tipo de las propiedades (props).

Dentro de nuestro componente (FirstApp), podemos definir el tipo de las propiedades del mismo:

```
FirstApp.propTypes = {  
  title: PropTypes.string.isRequired,  
  subTitle: PropTypes.number,  
}
```

Hay que tener en cuenta que si el tipo de alguna propiedad debe de pasarse siempre, deberemos añadir al final de la línea la condición *isRequired*.



5. PropTypes. Tipo de las propiedades (props).

En el archivo *main.jsx* (componente padre) tendríamos que:

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode>  
    <FirstApp title='Hola, Soy Gustavo' subTitle= '123' />  
  </React.StrictMode>  
);
```

Aquí hay un error ¿sabes cuál es? ¿Funcionará nuestro código?



6. Default Props. Propiedades por defecto.

Para definir los valores por defecto que deben tener las propiedades de mi componente, haremos lo siguiente:

```
FirstApp.defaultProps = {  
  title: "No hay título",  
  subTitle: "No hay subtítulo",  
};
```

Las propiedades por defecto, ***defaultProps***, van a ser renderizadas antes que las ***propTypes***. Es decir, el navegador primero consultará las propiedades que se han declarado por defecto y luego el resto.

Tarea 6.8: Counter-App.

Crear un nuevo componente dentro de la carpeta SRC llamado **CounterApp**

El **CounterApp** debe ser un ***Functional Component***

El contenido del **CounterApp** debe de ser:

```
<h1>CounterApp</h1>
```

```
<h2> { value } </h2>
```

Donde "value" es una propiedad enviada desde el padre hacia el componente **CounterApp** (Debe ser numérica validada con PropTypes)

Reemplazar en el ***index.js*** ó ***main.jsx*** el componente de `<FirstApp />` por el componente `<CounterApp />`

Asegúrense de no tener errores ni warnings (Cualquier warning no usado, comentar el código)



7. Evento onClick (Eventos en general).

A continuación vamos a ver cómo se manejan los eventos en React. Os dejo un enlace a la documentación completa sobre el tema: <https://es.reactjs.org/docs/events.html>

Siguiendo con el ejemplo anterior, para añadir un botón a nuestro componente **CounterApp** haríamos lo siguiente:

```
<h1>CounterApp</h1>
  <h2>{value}</h2>
  <button>
    +1
  </button>
```



7. Evento onClick (Eventos en general).

Ahora quiero darle funcionalidad al botón, quiero añadirle el evento **onClick** (cuando dé click que haga algo). Para eso deberíamos escribir, por ejemplo:

```
<h1>CounterApp</h1>
  <h2>{value}</h2>
  <button onClick={function () { console.log( 'Hola Mundo' ) } }>
    +1
  </button>
```

Esto es una expresión de JavaScript: es una función que se va a llamar cuando yo haga click.



7. Evento onClick (Eventos en general).

Hay que tener en cuenta que el argumento que estamos recibiendo en nuestra función es opcional, por eso hemos puesto “()”, pero ahí tendríamos un evento del tipo **MouseEvent**. Este **MouseEvent** tendrá información acerca de “**dónde se hizo click**” y mucha más información.

```
<h1>CounterApp</h1>
  <h2>{value}</h2>
  <button onClick={function (event) { console.log(event) }}>
    +1
  </button>
```



7. Evento onClick (Eventos en general).

Normalmente no vamos a tener declarada la función dentro de nuestro código jsx del componente en cuestión, porque eso **dificulta la lectura y comprensión** del mismo. Lo que se hará es sacarlo fuera para que se entienda mejor:

```
function botonContador (event) {  
  console.log('event')  
}
```



7. Evento onClick (Eventos en general).

Y dentro de nuestro componente tendríamos ya algo más claro el código. Vemos que al dar onClick, mediante una función flecha hacemos la llamada a la función “botonContador”.

```
<h1>CounterApp</h1>
  <h2> {value} </h2>
  <button onClick={ (event) => botonContador(event) }>
    +1
  </button>
```



7. Evento onClick (Eventos en general).

En el código anterior tenemos una función (**onClick**) que recibe un (**evento**), y ese evento se lo pasa directamente a otra función (**botonContador**). Recibimos un argumento y para lo único que sirve es para pasárselo a una función. Cuando sucede esto, sabemos que podemos obviar pasarle el argumento a la función **onClick** y sólo indicar la referencia a la función **botonContador**:

```
<h1>CounterApp</h1>
  <h2> {value} </h2>
  <button onClick={ botonContador }>
    +1
  </button>
```




7. Evento onClick (Eventos en general).

Con esto vemos que el primer argumento, o argumentos, que vengan al **onClick** van a pasar de la misma forma (orden) a los argumentos definidos en la función **botonContador**.

Si ahora, por ejemplo, queremos o necesitamos mandar algo a nuestra función (**botonContador**) cuando pinchemos en el botón, es decir, cuando nos llegue un “evento” (**onClick**), sí deberemos indicarlo mediante la función flecha como al principio.

```
<button onClick={ (event) => botonContador (event, 'Hola Mundo') }>
```



7. Evento onClick (Eventos en general).

Y en nuestra función tendríamos, por ejemplo:

```
function botonContador (event, newValue) {  
  
  console.log(newValue);  
}
```



7. Evento onClick (Eventos en general).

La declaración anterior está bien, y es correcta, pero normalmente veremos éste tipo de funciones escritas como una constante y usando la función flecha de la siguiente forma:

```
function botonContador (event, newValue) {  
  
  console.log(newValue);  
}
```

CORRECTA

```
const botonContador = (event, newValue) =>  
{  
  
  console.log(newValue);  
}
```

MEJOR



8. useState - Hook.

Hooks son una nueva característica en React desde su versión 16.8. Estos te permiten usar el estado y otras características de React sin escribir una clase.

Lo primero que debemos hacer es importar el hook de react, para ello:

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

Al escribir “*use...*” le estamos indicando a React que es un **hook**, y cuando definamos nuestros propios **hooks** deberemos seguir la misma nomenclatura.



8. useState - Hook.

Vamos a declarar el primer hook:

```
const [ contador ] = useState (0);
```

¿Cómo deberíamos hacer para incrementar nuestro contador?

```
const botonContador = () => {  
    console.log(contador ++);  
};
```



8. useState - Hook.

El código anterior no va a funcionar porque nuestro “**contador**” está declarado como constante, y una constante no se puede modificar. Por eso, los hooks se declaran como constantes para evitar que se cambie su estado por accidente. La forma de hacerlo es usando un segundo argumento en la declaración que siempre va a comenzar con la palabra “**set**”, y seguida del nombre de nuestro hook:

```
const [ contador, setContador ] = useState (0);
```



8. useState - Hook.

De esta forma, ahora sí podemos manejar el estado de nuestro contador:

```
export const HookUseState = () => {  
  
  const [ contador, setContador ] = useState (0);  
  
  const botonContador = () => {  
    setContador (contador + 1);  
  };  
}
```

~~contador++~~




8. useState - Hook.

Podemos hacer ésto mismo pero sin usar la constante “contador”, en el caso de que no se pueda por cualquier motivo. La solución más correcta sería:

```
const [ contador, setContador ] = useState (0);
```

```
const botonContador = () => {  
  setContador ( (c) => c+1 );  
};
```




Mediante una nueva variable y una función flecha.

8. useState - Hook.

El estado inicial del “*contador*” se le puede pasar como propiedad, a partir de la llamada del padre, en el archivo “**main.jsx**”. De esta forma, le estamos asignando el valor indicado a nuestro contador:

```
ReactDOM.createRoot(document.getElementById('root')).render(  
  <React.StrictMode  
    <HookUseState value={20} />  
  </React.StrictMode>  
) ;
```



main.jsx

A través de la propiedad “value” lo inicializo a “20”.



8. useState - Hook.

Y en nuestro componente hijo, se vería así:

```
export const HookUseState = ({value}) => {  
  const [ contador, setContador ] = useState (value);  
  const botonContador = ()=> {  
    setContador(contador + 1);  
  };  
};
```

Tarea 6.9.: Aumenta-Disminuye-Reset

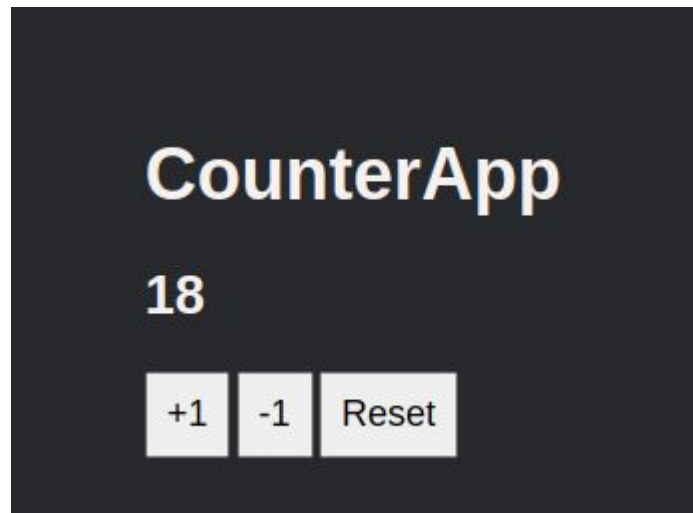
A partir de lo aprendido hasta ahora sobre los **hooks**, intenta realizar la siguiente tarea:

El valor inicial de **“value”** será igual a **“0”**.

El valor irá aumentando al pulsar **“+1”**.

Disminuirá al pulsar **“-1”**.

Volverá al valor **“0”** al pulsar **“Reset”**.





10. Pruebas unitarias y de integración.

Antes de lanzar una aplicación se deben realizar ciertas pruebas para comprobar que todo funciona correctamente. Existen dos tipos de pruebas principalmente:

- UNITARIAS: Enfocadas en pequeñas funcionalidades.
- INTEGRACIÓN: Enfocadas en cómo reaccionan varias piezas en conjunto.



10. Pruebas unitarias y de integración.

Características de las pruebas:

- Fáciles de escribir.
- Fáciles de leer.
- Confiables.
- Rápidas.
- Principalmente unitarias.



10. Pruebas unitarias y de integración.

Los pasos a seguir para realizar las pruebas deben tener en cuenta el concepto de **AAA**, que es el acrónimo de **Arreglar**, **Actuar** y **Afirmar** (*Arrange*, *Act* y *Assert* en inglés):

ARREGLAR (Arrange) - Preparar el estado inicial del sujeto a probar:

- Inicializamos variables.
- Importaciones necesarias.

Preparamos el ambiente a probar.



10. Pruebas unitarias y de integración.

ACTUAR (Act) - Aplicamos acciones o estímulos al sujeto:

- Llamar métodos.
- Simular clicks.
- Realizar acciones sobre el paso anterior.

AFIRMAR (Assert) - Observar el comportamiento resultante:

- Son los resultados esperados o no.
- Ej: Que algo cambie, algo incremente o bien que no pase nada.



10. Pruebas unitarias y de integración.

MITOS y BULOS acerca de las pruebas:

- Hacen que mi aplicación no tenga errores.
- Las pruebas no pueden fallar.
- Hacen más lenta mi aplicación.
- Es una pérdida de tiempo.
- Hay que probar todo.