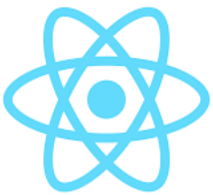
CURSO DE PROGRAMACIÓN FULL STACK

GUIA DE REACT



EGG

GUIA DE REACT

¿QUÉ ES NODE?

Node.js es un entorno de tiempo de ejecución de JavaScript (de ahí su terminación en .js haciendo alusión al lenguaje JavaScript). Este **entorno de tiempo** de ejecución en tiempo real incluye todo lo que se necesita para ejecutar un programa escrito en JavaScript.

Node.js fue creado por los **desarrolladores originales de JavaScript.** Lo transformaron de algo que solo podía ejecutarse en el navegador en algo que se podría ejecutar en los ordenadores como si de aplicaciones independientes se tratara. Gracias a Node.js se puede ir un paso más allá en la programación con JavaScript no solo creando sitios web interactivos, sino teniendo la capacidad de hacer cosas que otros lenguajes de secuencia de comandos como Python pueden crear.

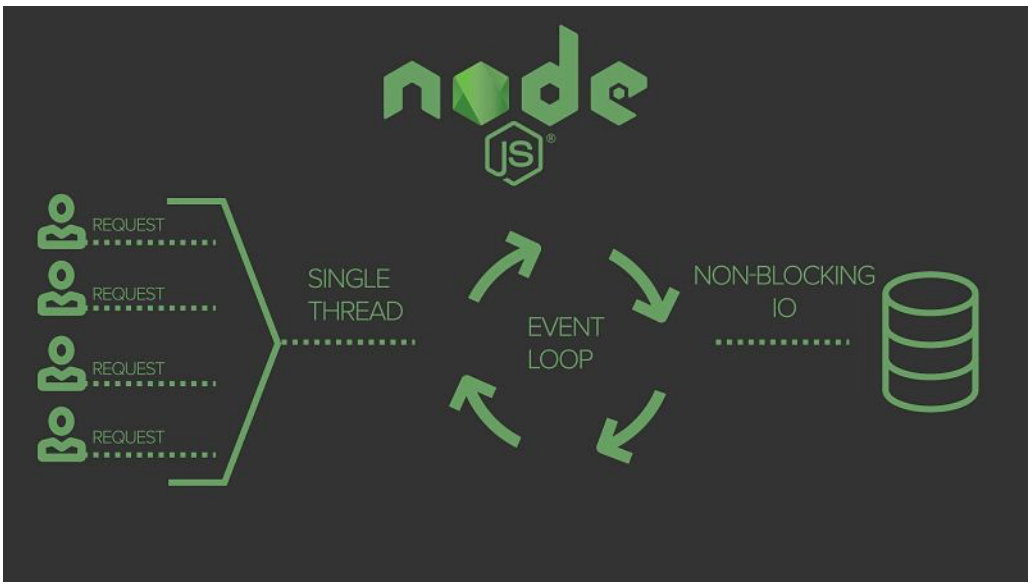
¿CÓMO FUNCIONA NODE.JS?

Para entender como funciona NodeJS, haremos uso de una metáfora:

Imagina que vas a un restaurante. El mesero te atiende, toma tu orden y la lleva a la cocina, luego va y atiende a otra mesa, mientras el cocinero prepara tu orden. Entonces, una misma persona puede atender múltiples mesas, no tiene que esperar que el cocinero termine de preparar una orden antes de pasar a atender a otra mesa. A esta naturaleza o arquitectura la llamamos **asíncrona** o **no bloqueante**. Así es como funcionan las aplicaciones de Node.js. El camarero viene a ser un **hilo** o **thread** asignado para manejar solicitudes. De esta forma un sólo hilo puede manejar múltiples solicitudes.

Imaginemos el caso de una aplicación como Twitter donde se tienen múchisimas conexiones simultáneas, para cada una se debe generar un hilo, y naturalmente, en algún punto, no vamos a tener mas hilos o memoria disponibles para atender mas peticiones. Entonces las nuevas peticiones deben esperar hasta que se libere algún hilo, se libere memoria, o debemos agregar mas hardware.

En Node.js no tenemos ese inconveniente, ya que **un solo hilo se encarga de procesar todas las peticiones**. Cuando llega una petición, el hilo de Node.js se encarga de procesarla. Si necesitamos hacer una petición a una base de datos, nuestro hilo no tiene que esperar hasta que la BD devuelva una respuesta. Mientras la petición a la base de datos se esta procesando, nuestro hilo puede atender otras peticiones. Cuando la respuesta de la base de datos está lista, se coloca en algo llamado ***Event Queue*** o **Cola de Eventos**. Node.js está constantemente monitoreando esta cola de eventos a mediante el ***Event Loop***, de tal manera que cuando la respuesta a alguna petición pendiente esta lista, Node.js la toma, la procesa y la devuelve.



- **Request:** petición
- **Single Thread:** único hilo
- **Event Loop:** cola de eventos
- **Non-Blocking I/O**(Input/Output): IO no bloqueante

Éste tipo de arquitectura hace a Node.js **ideal para construir aplicaciones** que involucran una gran cantidad de accesos a disco, peticiones de red (consultar bases de datos, consultar servicios web), etc. Podemos atender una gran cantidad de clientes sin la urgencia de disponer de más hardware. Y es por ésto que las aplicaciones en Node.js son altamente escalables.

NODE Y REACT

Nodejs es la plataforma más conveniente para alojar y ejecutar un servidor web para una aplicación React. Es por dos razones principales:

- Utilizando un NPM (Node Package Manager), Node trabaja junto con el registro de NPM para instalar fácilmente cualquier paquete a través de la CLI de NPM.
- Node agrupa una aplicación React en un solo archivo para una fácil compilación usando webpack y varios otros módulos de Node.

¿QUÉ ES NPM?

npm es el Node Package Manager que viene incluido y ayuda a cada desarrollo asociado a Node.

Esta herramienta funciona de dos formas:

- Como un repositorio ampliamente utilizado para la publicación de proyectos Node.js de código abierto. Lo que significa que es una plataforma en línea donde cualquiera puede publicar y compartir herramientas escritas en JavaScript.

- Como una herramienta de línea de comandos que ayuda a interactuar con plataformas en línea, como navegadores y servidores. Esto ayuda a instalar y desinstalar paquetes, gestión de versiones y gestión de dependencias necesarias para ejecutar un proyecto.

Para usarlo, debes instalar **node.js**, ya que están desarrollados de forma agrupada.

La utilidad de línea de comando del administrador de paquetes Node permite que **node.js** funcione correctamente.

Para usar paquetes, tu proyecto debe contener un archivo llamado **package.json**. Dentro de ese archivo, encontrarás metadatos específicos para los proyectos.

Los metadatos muestran algunos aspectos del proyecto en el siguiente orden:

- El nombre del proyecto
- La versión inicial
- Descripción
- El punto de entrada
- Comandos de prueba
- El repositorio git
- Palabras clave
- Licencia
- Dependencias
- Dependencias de desarrollo

Los metadatos ayudan a identificar el proyecto y actúan como una línea de base para que los usuarios obtengan información al respecto.

NPX

Desde la versión 5.2.0 de npm, npx está preinstalado con npm. Así que es más o menos estándar hoy en día.

npx es también una herramienta CLI cuyo propósito es facilitar la instalación y la gestión de las dependencias alojadas en el registro npm.

Ahora es muy fácil ejecutar cualquier tipo de ejecutable basado en Node.js que normalmente se instalaría a través de npm.

NPX VS NPM

npx nos ayuda a evitar el versionado, los problemas de dependencia y la instalación de paquetes innecesarios que sólo queremos probar.

También proporciona una forma clara y fácil de ejecutar paquetes, comandos, módulos e incluso listas y repositorio de GitHub.

¿QUÉ ES REACT?

React es una librería Javascript focalizada en el desarrollo de interfaces de usuario. Así se define la propia librería y evidentemente, esa es su principal área de trabajo. Sin embargo, lo cierto es que en React encontramos un excelente aliado para hacer todo tipo de aplicaciones web, SPA (Single Page Application) o incluso aplicaciones para móviles. Para ello, alrededor de React existe un completo ecosistema de módulos, herramientas y componentes capaces de ayudar al desarrollador a cubrir objetivos avanzados con relativamente poco esfuerzo.

Por tanto, React representa una base sólida sobre la cual se puede construir casi cualquier cosa con Javascript. Además facilita mucho el desarrollo, ya que nos ofrece muchas cosas ya listas, en las que no necesitamos invertir tiempo de trabajo.

El nombre de React proviene de su capacidad de crear interfaces de usuario reactivas, la cual es la capacidad de una aplicación para actualizar toda la interfaz gráfica en cadena, como si se tratara de una formula en Excel, donde al cambiar el valor de una celda automáticamente actualiza todas las celdas que depende del valor actualizado y esto se repite con las celdas que a la vez dependía de estas últimas. Esto se lo conoce como programación reactiva y react lo hará mediante componentes, que los veremos más adelante.

CARACTERÍSTICAS DE REACT

Al igual que otros marcos web JavaScript, React.SJ tiene muchas características principales que podrían ser su consideración. Son:

- Composición de componentes.
- Desarrollo Declarativo Vs Imperativo.
- Performance gracias al DOM Virtual.
- Isomorfismo.
- Elementos y JSX.
- Componentes con y sin estado.
- Ciclo de vida de los componentes.
- Ideal para aplicaciones de alta demanda
- Permite el desarrollo de aplicaciones móviles

En comparación a otros marcos web de JavaScript, que lo hacen de los mejores:

- Fácil de aprender, tiene la curva de aprendizaje más rápida en comparación a otros marcos web.
- Programación reactiva de manera nativa

Estos son los que consideramos son las princípiales ventajas de React por sobre otros marcos web.

CREATE REACT APP (PRIMEROS PASOS EN REACT)

A no ser que seas un experto desarrollador frontend, la mejor alternativa para dar los primeros pasos con React es usar el paquete **create-react-app**. Te permitirá empezar muy rápido y ahorrarte muchos pasos de configuración inicial de un proyecto.

Generalmente cuando se construye un sitio o app web se tiene que lidiar con una serie de herramientas que forman parte del tooling de un frontend developer, como gestores de paquetes, de tareas, transpiladores, linters, builders, live reload, etc. Toda esta serie de herramientas pueden tener su complejidad si se quieren aprender con el suficiente detalle como para comenzar a usarlas en un proyecto, pero son esenciales para un buen workflow.

Por tanto, si queremos ser detallistas y proveernos de las herramientas necesarias para ser productivos y eficientes, se puede hacer difícil la puesta en marcha en un proyecto Frontend en general. Ahí es donde entra **Create React App, ofreciéndonos todo lo necesario para comenzar una app con React, pero sin tener que perder tiempo configurando herramientas.** Comenzaremos una app con un par de comandos sencillos, obteniendo muchos beneficios de desarrolladores avanzados.

Dentro del CMD en Windows y el Terminal en Mac. **Instalamos Create React App** con el siguiente comando de npm:

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

Una vez lo hemos instalado de manera global, **nos paramos con el CMD o Terminal en na carpeta creada para nuestro proyecto React** y lanzamos el comando para comenzar una nueva aplicación:

```
npx create-react-app mi-app
```

"mi-app" será el nombre de tu aplicación React. Obviamente, podrás cambiar ese nombre por uno de tu preferencia.

Mediante el anterior comando se cargarán los archivos de un proyecto vacío y todas las dependencias de npm para poder contar con el tooling que hemos mencionado. Una vez terminado el proceso, que puede tardar un poco, podemos entrar dentro de la carpeta de nuestra nueva app.

```
cd mi-app/
```

Y una vez dentro hacer que comience la magia con el comando:

```
npm start
```

Observarás que, una vez lanzas el comando para iniciar la app, se abre una página en tu navegador con un mensaje de bienvenida. Ese es el proyecto que acabamos de crear. No obstante, en el propio terminal nos indicarán la URL del servidor web donde está funcionando nuestra app, para cualquier referencia posterior. De manera predeterminada será el http://localhost:3000/, aunque el puerto podría cambiar si el 3000 está ocupado.

CARPETAS DE NUESTRA APP REACT

El listado de nuestra app recién creada es bastante sencillo. Observarás que tenemos varias carpetas:

- node_modules: con las dependencias npm del proyecto
- public: esta es la raíz de nuestro servidor donde se podrá encontrar el index.html, el archivo principal y el favicon.ico que sería el icono de la aplicación.
- src: aquí es donde vamos a trabajar principalmente para nuestro proyecto, donde vamos a colocar los archivos de nuestros componentes React.

Además encontrarás archivos sueltos como:

- README.md que es el readme de Create React App, con cantidad de información sobre el proyecto y las apps que se crean a partir de él.
- package.json, que contiene información del proyecto, así como enumera las dependencias de npm, tanto para desarrollo como para producción. Si conoces npm no necesitarás más explicaciones.
- .gitignore que es el típico archivo para decirle a git que ignore ciertas cosas del proyecto a la hora de controlar el versionado del código.
- yarn.lock Este archivo no se debe tocar, puesto que es código generado por Yarn. Su utilidad es ofrecer instalaciones de dependencias consistentes a lo largo de todas las instalaciones de un proyecto.

COMPONENTE RAÍZ DEL PROYECTO CON CREATE REACT APP

Para nuestros primeros pasos con React no necesitamos más que entrar en la carpeta src y empezar a editar su código. De entre todos los archivos que encuentras en la carpeta src por ahora nos vamos a quedar con uno en concreto, src/App.js, en el que se crea el componente principal de nuestra app.

De momento en nuestra aplicación sólo tenemos un componente, el mencionado componente raíz localizado en src/App.js. Sin embargo a medida que se vaya desarrollando la aplicación se irán creando nuevos componentes para realizar tareas más específicas, e instalando componentes de terceros, que nos ayuden a realizar algunas tareas sin necesidad de invertir tiempo en programarlas de nuevo. Así es como llegamos a la arquitectura frontend actual, basada en componentes.

HOLA MUNDO EN REACT

En el mencionado archivo src/App.js nos encontraremos poco código, pero para no despistarnos en nuestro primer "Hola Mundo", podemos borrar gran parte y quedarnos solamente con esto:

```
import React, { Component } from 'react';
class App extends Component {
  render() {
    return (
      <div>
        <h1>Hola Mundo!</h1>
        <p>Bienvenidos a los primeros pasos con React</p>
      </div>
    );
  }
}
export default App;
```

En la primera línea "import React..." se está importando la librería React y la clase Component, que es la que usamos para, en la segunda línea, extender la clase App.

Las clases que usamos para implementar componentes React solo necesitan un método render para poder funcionar. En nuestro caso observarás que el componente que estamos creando, llamado App (como el nombre de la clase), solamente tiene ese método render().

```
render() {
  return (
    <div>
      <h1>Hola Mundo!</h1>
      <p>Bienvenidos a los primeros pasos con React</p>
    </div>
  );
}
```

Todo el código que coloques dentro del método render() será el marcado que usará el componente para su representación en la página.

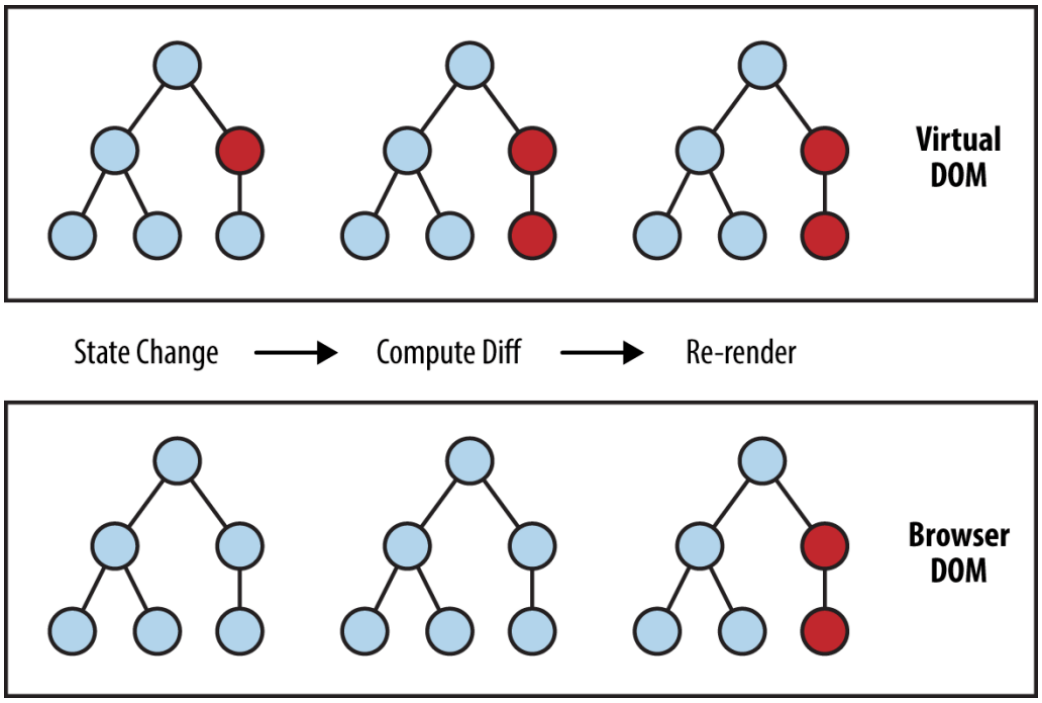
Guardando el archivo podrás observar como el navegador refresca la página automáticamente, gracias a las herramientas incluidas de serie dentro de Create React App.

DOM VIRTUAL

El *Virtual DOM* es una representación en memoria del *DOM real* (que es el DOM que conocemos de JS) que actúa de intermediario entre el estado de la aplicación y el DOM de la interfaz gráfica que está viendo el usuario.

Puesto que cada elemento es un nodo en el árbol del DOM, cada vez que se produce un cambio en cualquiera de estos elementos (o un nuevo elemento es añadido) se genera un nuevo Virtual DOM con el árbol resultante. Dado que este DOM es *virtual,* la interfaz gráfica aún no es actualizada, sino que se compara el DOM real con este DOM virtual con el objetivo de calcular la forma más óptima de realizar las cambios (es decir, de *renderear* los menos cambios posibles). De este modo se consigue reducir el coste en términos de rendimiento de actualizar el DOM real.

Este grafico ejemplifica el proceso:



1. State Change:En este primer paso, se produce un cambio en el estado del nodo de color de rojo, lo cual provoca que se genere en memoria un Virtual DOM con el árbol resultante tras ese cambio.

2. Compute diff: A continuación se realiza la comparación entre el árbol del Virtual DOM y el del navegador (DOM real) con el fin de detectar los cambios producidos. Como veis, el cambio afecta a toda la rama descendiente del nodo cuyo estado cambió.

3. Re-render: Finalmente, se consolida el cambio en el DOM real y la interfaz gráfica es actualizada de golpe.

¿CÓMO USA REACT EL VIRTUAL DOM?

En React, cada pieza de la UI es un componente y cada componente posee un estado interno. Este estado es *observado* por la librería con el fin de detectar cambios en él de modo que, cuando se produce un cambio, React actualiza el árbol de su Virtual DOM y sigue el mismo proceso para trasladar los cambios resultantes a la interfaz presentada en el navegador. Esto le permite tener un rendimiento mejor que las librerías que manipulan el DOM directamente, pues React sólo actualiza aquellos objetos en los que ha detectado cambios durante el proceso de *diffing.*

Además, React traslada los cambios al DOM de la interfaz gráfica de forma masiva lo cual también incrementa el rendimiento. Esto se debe a que en vez de enviar múltiples cambios, React los junta todos en uno para reducir el número de veces que la interfaz gráfica debe pintarse.

Finalmente, otra de las ventajas que nos proporciona React es la abstracción de todo el proceso de actualización del DOM que nos proporciona. Es decir, nos permite olvidarnos de actualizar los atributos o el valor de los nodos que componen nuestra interfaz gráfica ya que todo esto sucede a nivel interno.

JSX

JSX es una extensión de la sintaxis de JavaScript para su uso en React. JSX, produce elementos de React. Un elemento es un bloque más de las aplicaciones de React y describe lo que quieres ver en pantalla.

Se puede utilizar JSX dentro de declaraciones IF y bucles FOR, asignarlo a variables, aceptarlo como argumento y retornarlo desde dentro de funciones.

ATRIBUTOS CON JSX

Puedes utilizar comillas para especificar Strings como atributos:

```
const elemento = <div tabIndex="0"></div>
```

También usar llaves para insertar una expresión JS:

```
const elemento = <img src={user.avatarURL}></img>
```

JSX REPRESENTA OBJETOS

Estos objetos son llamados "Elementos de React". React lee estos objetos y los usa para construir el DOM.

Supongamos que tenemos el siguiente elemento:

```
const elemento = (<h1 className = 'saludo' > Hola Mundo </h1>);
```

Este elemento va a ser idéntico al siguiente elemento

```
const elemento = React.createElement('h1',
{className = 'saludo'},
Hola Mundo );
```

RENDERIZAR UN ELEMENTO

Supongamos que tenemos las siguiente variables:

```
const name = 'Lionel Messi',
const elemento = <h1> Hola, {name} </h1>
```

Para renderizarlo, imagina que tenemos el siguiente div en alguna parte del HTML:

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

Para renderizar un elemento de React en un nodo raíz del DOM, pasa ambos a ReactDOM.render()

```
ReactDOM.render(

element.
document.getById("root")

);
```

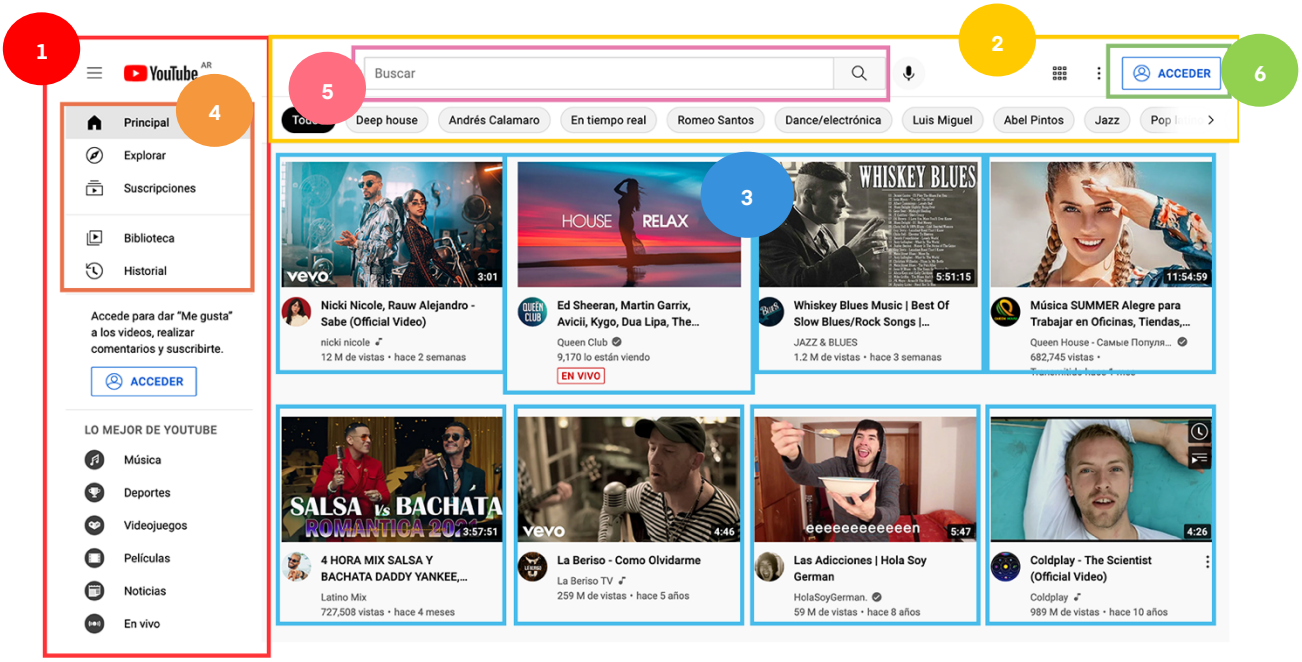
```
Muestra: Hola Lionel Messi
```

ACTUALIZANDO EL ELEMENTO

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

COMPONENTES

React, como dijimos es una librería JavaScript para crear interfaces de usuario dependiendo de su estado. Las aplicaciones React se construyen **mediante componentes**, los cuales **son elementos independientes y pueden ser reutilizados**, además, describen cómo tienen que visualizarse y cómo tienen que comportarse. Los componentes permiten separar la interfaz de usuario en piezas independientes.



Dentro de una pagina como Youtube, podemos marcar varios componentes:

1) Un SideBar

2) Un Header

3) Cada carta es su propio componente

También dentro de cada componente existen otros componentes:

4) Los botones Principal, Explorar, Suscripciones, Biblioteca e Historial. Cada uno de esos botones es un componente, que vive dentro del componente Sidebar.

5) El buscador es un componente dentro del Header

6) El botón de login es otro componente dentro del Header

Y en esta pagina podríamos nombrar muchos componentes más.

Cada aplicación React comienza con un componente de raíz y se compone de muchos componentes en una formación de árbol. Los componentes en React son "funciones" que dejan la UI basada en la data (apoyo y estado) que recibe.

COMPONENTES EN REACT

Conceptualmente, los componentes son como las funciones de JavaScript. Aceptan entradas arbitrarias (llamadas "props") y devuelven a React elementos que describen lo que debe aparecer o se debe ver en la pantalla. Los props son las propiedades o información que viaja de un componente padre a un hijo.

Además existen dos tipos de componentes en React:

- Componentes funcionales: Solo tienen propiedades.
- Componentes de clase: Tienen propiedades, ciclos de vida y estado.

La forma más sencilla de definir un componente es escribir una función de JavaScript:

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

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

También puedes utilizar una clase de ES6 para definir un componente:

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

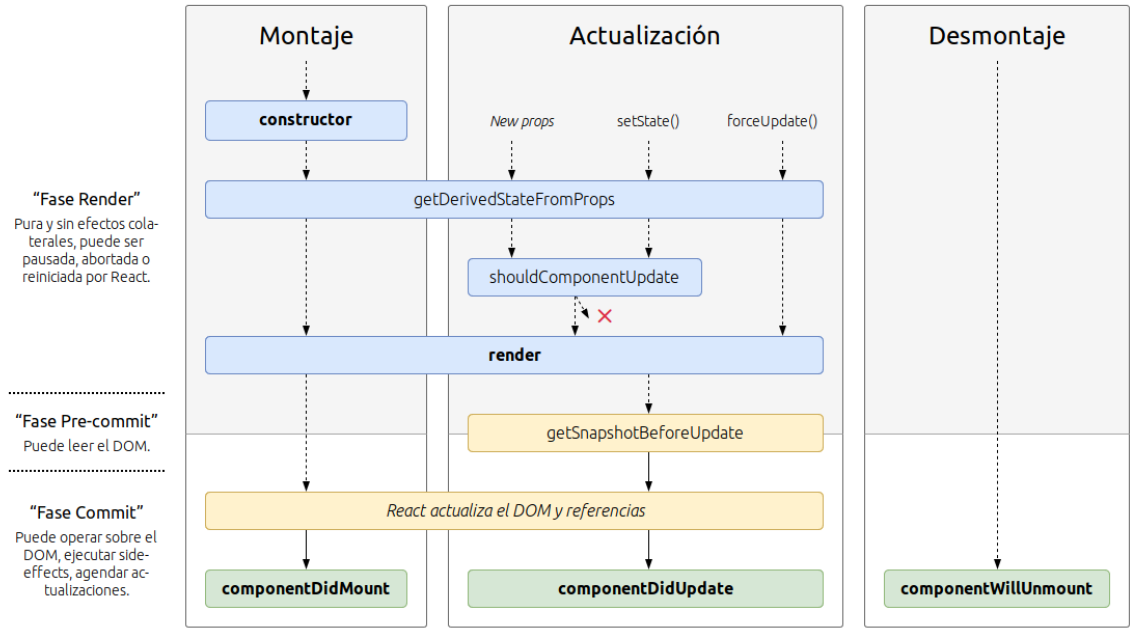
CICLO DE VIDA DE UN COMPONENTE

Vimos que **existen** dos formas de **crear componentes** (*funcionales* y de *clases*).

Cuando creamos un **componente de clase**, tenemos la **ventaja**, que al **extenderlo** de *React.Component* nos va a **proporcionar** más *métodos* y *propiedades*, que van a **mejorar** de manera exponencial **el componente**.

```
class Title extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!!!</h1>;
  }
}
ReactDOM.render(<Title name="Mauricio" />,
document.getElementById("root"));
```

Para entender mejor el ejemplo, es necesario saber qué **React.Component**, tiene tres *ciclos de vida:*



a. Montaje

Cuando se crea una **instancia** de un **componente** y **se inserta en el DOM**.

b. Actualización

Cuando **sufre** algún **cambio las propiedades** (props) o **el estado** (state) **del componente**.

c. Desmontaje

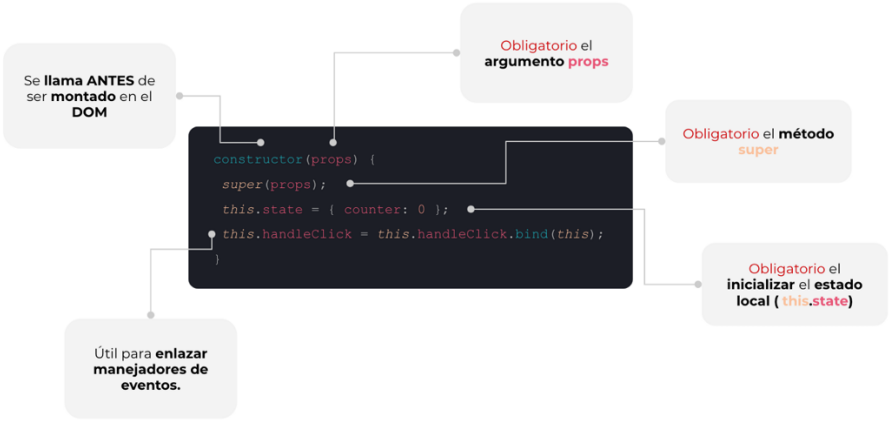
Cuando el **componente** se **elimina** del DOM.

MÉTODOS DE CICLO DE VIDA

A su vez cada **ciclo de vida,** nos va a **proporcionar** diferentes **métodos de ciclo de vida** para el **componente**, **dependiendo** en qué **fase** se encuentre, vamos a poder utilizar:

a. Montaje

1. constructor(props):



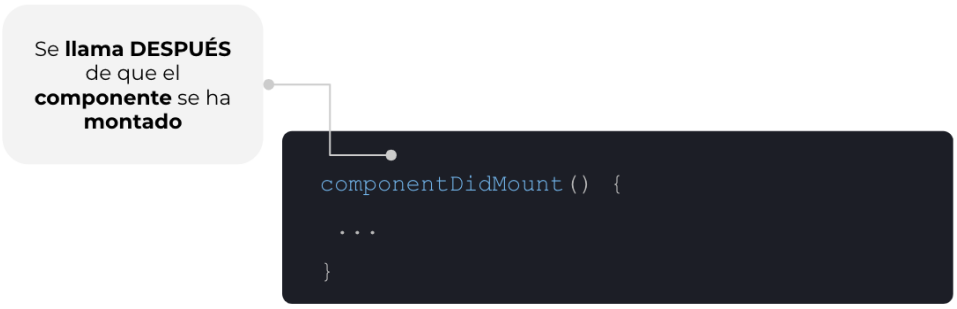
- El **constructor es llamado antes de ser montado el componente** en el DOM.
- Es **OBLIGATORIO** pasar el **argumento props.**
- Al ser **extendiendo React.Component**, es **OBLIGATORIO** ejecutar el método **super(props) pasandole el argumento props**, antes que cualquier otra **instrucción**, ya que de no ser así, puede ocasionar errores.
- Es útil para **inicializar** el **estado local** (this.state).
- Es útil para **enlazar manejadores de eventos** a una instancia.

2. render():

- El **único método obligatorio** en un **componente de clase.**
- Puede **retornar** un **elemento** en **React** a través de JSX (Ej. , <MyComponent />, y es renderizado como un nodo DOM.
- Puede **retornar fragmentos** de **elementos** (útil cuando creamos tablas)

- Puede **retornar** el **renderizado** de un **hijo** como otro **nodo DOM**, llamados *portales* (es útil para sacar sacarlo de un contenedor, ej. modales, tooltips).
- Puede **retornar datos** tipo **String** (cadenas de texto o números), que son **renderizados** como **nodos de texto en el DOM**.
- Se **recomienda que la función sea pura**, esto quiere decir que **no modifique variables, propiedades o estado del componente**, debe **devolver siempre el mismo resultado cada vez que se invoca.**

3. componentDidMount():

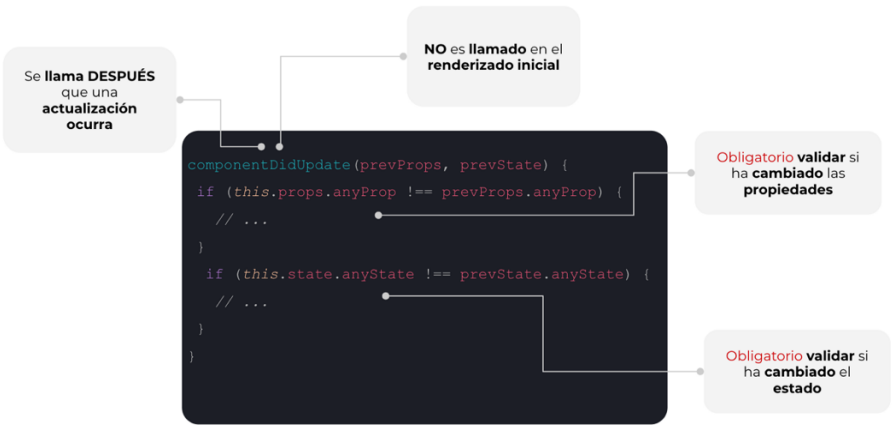


- Se **invoca** inmediatamente **después** que el **componente** se **monte**.
- Carga de **datos de alguna API.**
- Podemos modificar el **estado** (*setState()*), donde **activará** un **renderizado extra**; hay que tener **cuidado** de cómo ocuparlo, ya que puede **afectar** al **rendimiento** de la **aplicación**, es **útil** cuando debemos saber **propiedades** de algún **nodo DOM** (tamaño, posición) como un modal o tooltip.

b. Actualización

1. render():
   - Todas las **opciones** que tiene cuando se **monta**.
   - Cuando sufre algún **cambio el componente** y se necesita **renderizar el componente** (cuando se cambia el estado en componentDidMount()).

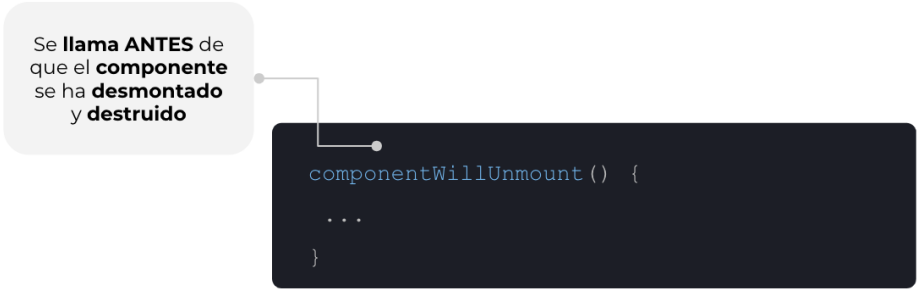
2. componentDidUpdate(prevProps, prevState):



- Se **invoca** inmediatamente **después** de que la **actualización** ocurra.
- Este método **NO** es **llamado** en el **renderizado inicial**.
- Es **útil** para para hacer **solicitudes API**, **siempre** y cuando se **compare** las **propiedades** (*this.props*) con el **argumento prevProps**, esto con el fin de que si no hay cambios, no afecte.
- Podemos **modificar** el **estado** (*setState*), pero, al igual que el punto anterior se **debe comparar** el actual ***this.state*** con el **anterior** ***prevState***, ya que de lo contrario puedes **generar un bucle infinito**.
- Es **útil** cuando queremos **realizar** un **efecto secundario** (ej. obtención de datos o animaciones) en una respuesta debido a un **cambio** en los *props*.

c. Desmontaje

1. componentWillUnmount():



- Se **invoca** inmediatamente **antes de desmontar** y **destruir el componente**.
- Es **útil** para **realizar limpieza** de **tareas** (*temporizadores*, *solicitudes de red*, *suscripciones*)
- **PROHIBIDO modificar el estado** (*setState*), ya que **el componente se vuelve a renderizar.**
- Una vez que una instancia de componente sea desmontada, **NUNCA será montada de nuevo**, si volvemos a montar el componente **va a ser una NUEVA instancia de él**.

PROPIEDADES (PROPS) Y ESTADO (STATE) EN EL CICLO DE VIDA (MONTAJE, ACTUALIZACIÓN, DESMONTAJE)

Si hemos puesto atención a los conceptos y ejemplos de arriba, he mencionado mucho las palabras **propiedades (props)** y **estado (state)** que **tiene** un **componente;** estos conceptos, tienen una **función** específica **dentro** del **componente** y al igual que los métodos, **dependiendo** en el **ciclo de vida** en el que se **encuentren**, son los usos que podemos darle.

1. this.props
    - Son los **atributos** que se **envían** desde el **elemento**.
    - Son solo de **LECTURA**, por lo que **NUNCA** deben **modificarse**
    - Está considerado dentro del **grupo** de **propiedades** de **instancia** de un **componente**
2. this.state
    - Es un **objeto** de **propiedades** (*key/value*), que van a ser **privadas** y que son **controladas COMPLETAMENTE** por el **componente**
    - Se ocupan cuando queremos "***observar***" alguna **propiedad** y que **afecte** el **componente**
    - El **único** lugar para **crearlo** y **modificarlo directamente** como ***this.state*** es el **método constructor**, para todos los demás usar ***this.setState()***
    - **NO se recomienda agregar un estado**, si **no va a cambiar** o cuando **no queremos** que **afecte al nodo DOM.**
    - Para el manejo de **propiedades internas** "*normales*" es mejor **agregarlas** dentro de ***this*** *(this.anyVar)*
    - Está considerado dentro del **grupo** de **propiedades** de **instancia** de un **componente**
3. this.setState({key: value})

- **Método** que vamos a **ocupar** cuando queremos **actualizar** un **valor** de una **propiedad** del **objeto estado** (***this.state***).
- Para **cambiar** una **propiedad NUNCA usar directamente** *this.state*
- Cuando se **actualiza** un **estado**, **React vuelve** a **procesar** el **componente y** sus **elementos**.
- Lo ocupamos para **actualizar** la **interfaz de usuario** (ej. respuesta de un manejador de evento, respuesta de una API, validación)
- Es importante saber que, **internamente React puede retrasar** la **actualización** del **componente**, **esperando** a que **otros componentes puedan** ser **afectados**, por lo que **NO garantiza que los cambios de estado sean inmediatos**.
- Para **garantizar** el **cambio inmediato**, se **recomienda** hacerlo en el **método componentDidUpdate**() o **setState** con un **callback** (veremos un ejemplo más abajo

FUNCTIONAL COMPONENT (REACT HOOKS)

Anteriormente, habíamos hablado de los componentes "funcionales" que son funciones JavaScript, y también, habíamos dicho que la forma más sencilla de definir un componente es escribir una función de JavaScript. Es aquí, donde entran los **React Hooks.**

Los Hooks son una nueva característica en React 16.8. Estos te permiten usar el estado y otras características de React sin escribir una clase. Los vamos a utilizar junto con los componente funcionales y de esa manera, evitar usar clases y solo trabajar con un solo tipo de componente.

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

function Example() {

  // Declara una nueva variable de estado, la cual llamaremos "count"
const [count, setCount] = useState(0);

  return (

    <div>

      <p>You clicked {count} times</p>

      <button onClick={() => setCount(count + 1)}>

        Click me

      </button>

    </div>

  );

}
```

Esta nueva función **useState** es el primer "Hook" que vamos a aprender, pero este ejemplo es solo una introducción. ¡No te preocupes si aún no tiene sentido!

Antes de continuar, debes notar que los Hooks son:

- **Completamente opcionales.** Puedes probar Hooks en unos pocos componentes sin reescribir ningún código existente. Pero no tienes que aprender o usar Hooks ahora mismo si no quieres.
- **100% compatibles con versiones anteriores.** Los Hooks no tienen cambios con rupturas con respecto a versiones existentes.
- **Disponibles de inmediato.** Los Hooks ya están disponibles con el lanzamiento de la versión v16.8.0.

¿PERO QUÉ ES UN HOOK?

Los **Hooks son funciones que te permiten "enganchar" el estado de React y el ciclo de vida desde componentes de función**. Los hooks no funcionan dentro de las clases — te permiten usar React sin clases. (No recomendamos reescribir tus componentes existentes de la noche a la mañana, pero puedes comenzar a usar Hooks en los nuevos si quieres).

React proporciona algunos Hooks incorporados como useState. También puedes crear tus propios Hooks para reutilizar el comportamiento con estado entre diferentes componentes. Primero veremos los Hooks incorporados.

HOOK DE ESTADO

Este ejemplo renderiza un contador. Cuando haces click en el botón, incrementa el valor:

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

function Example() {
  // Declara una nueva variable de estado, que llamaremos "count".  const
[count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Aquí, **useState es un Hook** (hablaremos de lo que esto significa en un momento). Lo llamamos dentro de un componente de función para agregarle un estado local. React mantendrá **este estado entre re-renderizados**. useState devuelve **un par: el valor de estado actual y una función que le permite actualizarlo**. Puedes llamar a esta función desde un **controlador de eventos** o **desde otro lugar**. Es **similar a this.setState** en una clase, excepto que no combina el estado antiguo y el nuevo.

El único argumento para useState es el estado inicial. En el ejemplo anterior, es 0 porque nuestro contador comienza desde cero. Ten en cuenta que a diferencia de this.state, el estado aquí no tiene que ser un objeto — aunque puede serlo si quisieras. El argumento de estado inicial solo se usa durante el primer renderizado.

HOOK DE EFECTO

Es probable que hayas realizado recuperación de datos, suscripciones o modificación manual del DOM desde los componentes de React. Llamamos a estas operaciones "efectos secundarios" (o "efectos" para abreviar) porque pueden afectar a otros componentes y no se pueden hacer durante el renderizado.

El Hook de efecto, useEffect, agrega la capacidad de realizar efectos secundarios desde un componente de función. **Tiene el mismo propósito que componentDidMount, componentDidUpdate y componentWillUnmount** en las clases React, **pero unificadas en una sola API.**

Por ejemplo, este componente establece el título del documento después de que React actualiza el DOM:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);
  // Similar a componentDidMount y componentDidUpdate:
useEffect(() => {
// Actualiza el título del documento usando la Browser API
document.title = `You clicked ${count} times`;
});
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Cuando llamas a useEffect, le estás diciendo a React que **ejecute tu función de "efecto" después de vaciar los cambios en el DOM**. Los **efectos se declaran dentro del componente** para que tengan **acceso a sus props y estado**. De forma **predeterminada**, React **ejecuta los efectos después de cada renderizado** — incluyendo el primer renderizado.

En el ejemplo que tenemos del useEffect, va a llamar las funciones sobre los componentes de manera infinita, esto a veces nos puede ser útil, pero también nos puede generar un bucle infinito, entonces supongamos que queremos cargar unas cartas, estas se van a cargar de manera indefinida y va a romper nuestra pagina.

1. Para ejecutar en cada reenvió del componente:

```
useEffect(() => {
})
```

Este es el que nos puede generar un bucle infito.

2. Para ejecutar algo solo una vez después del montaje del componente (se procesará una vez), debe usar:

```
useEffect(() => {
// hará algo solo una vez si pasas un arreglo vacío []
// tener en cuenta que ese componente se renderizará una vez (con los valores predeterminados) antes de que lleguemos aquí
  }, [])
```

3. Para ejecutar cualquier cosa una vez en el montaje de componentes y en el cambio de datos / datos2 :

```
const [data, setData] = useState(false)
  const [data2, setData2] = useState('default value for first render')
  useEffect(() => {
// si pasas alguna variable, el componente se volverá a procesar después del montaje del componente una vez y una segunda vez si esto (en mi caso, data o data2) se cambia
  }, [data, data2] )
```

Según que tenemos que hacer elegiríamos una opción o la otra.

¿QUÉ ES REACT ROUTER?

React Router es una colección de componentes de navegación la cual podemos usar como ya lo mencione tanto en web o en móvil con React Native. Con esta librearía vamos a obtener un enrutamiento dinámico gracias a los componentes, en otras palabras tenemos unas rutas que renderizan un componente.

BENEFICIOS DE REACT ROUTER

- Establecer rutas en nuestra aplicación ej: Home, About, User.
- Realizar redirecciones
- Acceso al historial del navegador
- Manejo de rutas con parámetros
- Páginas para el manejo de errores como 404

COMPONENTES PILARES DE REACT ROUTER

BrowserRouter

Este componente es el encargado de envolver nuestra aplicación dándonos acceso al API historial de HTML5 (pushState, replaceState y el evento popstate) para mantener su UI sincronizada con la URL.

Switch

Este componente es el encargado de que solo se renderice el primer hijo **Route o Redirect** que coincide con la ubicación. Si no usar este componente todos los componentes Route o Redirect se van a renderizar mientras cumplan con la condición establecida.

Route

Con Route podemos definir las rutas de nuestra aplicación, quizás sea el componente más importante de React Router para llegar a comprender todo el manejo de esta librería. Cuando definimos una ruta con Route le indicamos que componente debe renderizar.

Este componente cuanta con algunas propiedades.

Path: la ruta donde debemos renderizar nuestro componente podemos pasar un string o un array de string.

Exact: Solo vamos a mostrar nuestro componente cuando la ruta sea exacta. Ej: /home === /home.

Strict: Solo vamos a mostrar nuestro componente si al final de la ruta tiene un slash. Ej: /home/ === /home/

Sensitive: Si le pasamos true vamos a tener en cuenta las mayúsculas y las minúsculas de nuestras rutas. Ej: /Home === /Home

Component: Le pasamos un componente para renderizar solo cuando la ubicación coincide. En este caso el componente se monta y se desmonta no se actualiza.

Render: Le pasamos una función para montar el componente en línea.

¿QUE ES UNA URL?

Una cosa importante para trabajar con React Router es entender que es un URL y las partes que componen a una URL. Ya que vamos a trabajar con dichas partes de la URL con React Router.

Un URL (Uniform Resource Locator) hace referencia más comúnmente a páginas web (HTTP), pero también puede aplicarse a la transferencia de archivos (FTP), correo electrónico (mailto), acceso a bases de datos (ODBC) y muchas otras aplicaciones. Una URL HTTP también puede describir la ubicación de recursos externos como una imagen, hoja de estilo, script e incluso una sección específica dentro de un documento HTML. Las URL completas o al menos parciales aparecen en la barra de direcciones de la mayoría de los navegadores web.

PARTES DE UNA URL

1) **Protocolo o esquema (Scheme):** define como el recurso se va a obtener.
2) **Subdominio (SubDomain):** www es el más común pero no es necesario.
3) **Dominio (Domain):** valor único dentro del dominio de nivel superior.
4) **Dominio de nivel superior (Top-level Domain):** existen cientos de opciones.
5) **Puerto (Port):** si es omitido, HTTP se conectará al puerto 80, HTTPS al 443.
6) **Ruta (Path):** especifica y capaz encuentra el recurso requerido por el usuario.
7) **Parámetro (Query String):** datos pasados al servidor, si está presente
8) **Etiqueta (Fragment Identifier):** especifica un lugar concreto de una pagina HTML

1. Protocolo

El esquema suele ser el nombre de un protocolo, que define cómo se obtendrá el recurso. Sin embargo, esquemas como *"archivo"* y *"mailto"* no especifican un protocolo. Los clientes, como los navegadores web, se conectan a sitios web a través del Protocolo de transferencia de Hipertexto, también conocido como HTTP.

2. Subdominio

Aunque www es el más común, los subdominios se pueden configurar con cualquier valor que consista en caracteres ASCII alfanuméricos que no distinguen entre mayúsculas y minúsculas. Se permiten guiones si están rodeados por otros caracteres ASCII u otros guiones. Los subdominios también se pueden eliminar, acortando y simplificando toda la URL.

3-4. Dominio + nivel superior

En octubre de 1984, el conjunto original de dominios de nivel superior se definió como:

- .com
- .edu
- .gov
- .mil
- .org
- .neto

Desde 1984 se crearon un puñado de nuevos dominios de nivel superior, pero el cambio más significativo se produjo en 2012 cuando ICANN (Corporación de Internet para la Asignación de Nombres y Números) autorizó la creación de casi dos mil nuevos dominios de nivel superior.

5. Puerto

El propósito de un puerto es identificar de forma única diferentes procesos o aplicaciones que se ejecutan en un solo servidor. Los números de puerto permiten que cada proceso o aplicación comparta una conexión de red. Las URL públicas a menudo no incluyen un número de puerto. En esos casos, se utiliza el puerto predeterminado del esquema. Los puertos predeterminados para HTTP y HTTPS son 80 y 443, respetuosamente.

6. Ruta

Una ruta describe una ubicación específica dentro de un sistema de archivos. En el contexto de una URL HTTP, una ruta describe un recurso específico, como un documento HTML.

7. Parámetro

Los parámetros contienen datos que se enviarán al servidor para procesamiento adicional. Puede contener pares de nombre / valor separados por un ampersand (&), así:

```
?first_name=Alfred&last_name=Pennyworth
```

En el ejemplo anterior, los datos serían:

"Nombre: Alfred"

"Apellido: Pennyworth"

Luego, estos datos se pasan a los scripts del lado del servidor para su procesamiento. Si esta es una consulta válida, el servidor devolverá información pertinente a Alfred Pennyworth.

8. Etiqueta

Las etiquetas en una URL aparecen después del hashtag #.

Su función, entre otras cosas, consiste en permitir hacer scroll hasta un elemento en concreto. Por ejemplo, si mandamos a alguien una URL que contenga una etiqueta, ésta le dirigirá a la parte exacta de la página en cuestión.

INSTALACIÓN DE REACT ROUTER

Para instalar la librería solo tenemos que ir a la terminal estar ubicados en la raíz de nuestro proyecto y ejecutar el siguiente comando.

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

TRABAJANDO CON REACT ROUTER

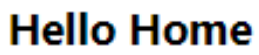
Teniendo todo listo ahora si vamos a nuestro editor de código y abrimos el archivo App.js que está ubicado en src/App.js acá vamos a limpiar muchas cosas hasta que al final tengamos algo como el siguiente código.

```
import React from 'react';
import './App.css';
import {
  BrowserRouter as Router,
  Route
} from "react-router-dom";
import Home from './pages/Home'
function App() {
  return (
    <Router>
      <div className="App">
          <Route exact path="/" component={Home} />
      </div>
    </Router>
  );
}
export default App;
```

Importamos nuestro componente BrowserRouter le damos un nombre Router también importamos Route de react-router-dom. Envolvemos nuestra aplicación con Router y definimos nuestra primera ruta en este caso nuestro home le indicamos que debe ser exacta la ruta y que haga render de nuestro componente Home pero donde esta nuestro componente Home bueno vamos a crearlo

```
// Home.js
import React from 'react'
const Home = () => (
  <section className="Home">
    <h3>Hello Home</h3>
  </section>
)
export default Home
```

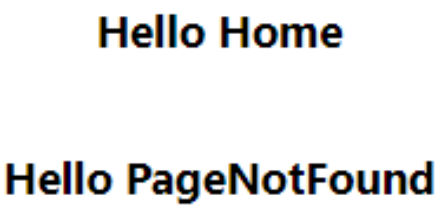
Este es nuestro componente Home que está dentro de una carpeta llamada pages/Home y solo tenemos un h3 con un texto. Si vamos al navegador a http://localhost:3000/ vamos a ver solo el texto.



Ahora vamos a crear otras páginas y una página para manejar el error 404.

```
import React from 'react';
import './App.css';
import {
  BrowserRouter as Router,
  Route
} from "react-router-dom";
import Home from './pages/Home'
import About from './pages/About'
import PageNotFound from './pages/PageNotFound'
function App() {
return (
 <Router>
 <div className="App">
  <Route exact path="/" component={Home} />
  <Route exact path="/about" component={About} />
  <Route component={PageNotFound} />
      </div>
    </Router>
  );
}
export default App;
```

Si vamos al navegador después de agregar estas nuevas rutas vamos a tener un pequeño problema y es que también vamos a tener el render de nuestro componente PageNotFound para solucionar esto lo que tenemos que hacer es envolver nuestras rutas con el componente Switch.



```
<Switch>
  <Route exact path="/" component={Home} />
  <Route exact path="/about" component={About} />
  <Route exact path="/users" render={() => <Users name='John Serrano' />}
/>
  <Route component={PageNotFound} />
</Switch>
```

Listo con esto ya solucionamos ese pequeño problema. Antes de seguir vamos a crear un menú de navegación para precisamente eso recorrer todas nuestras páginas y transformar nuestro proyecto en una SPA para esto vamos a usar el componente Link de React Router.

LINK

Con Link vamos a poder navegar por nuestra aplicación, este componente recibe las siguientes propiedades.

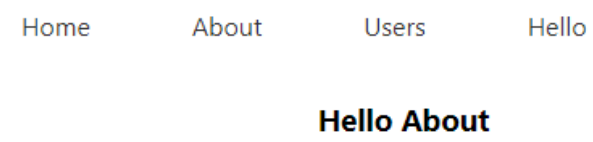
To: le podemos pasar un string, object o una function para indicarle la ruta a la cual queremos navegar.

Replace: cuando es verdadero, y hacemos clic en el enlace reemplazará la entrada actual en la pila del historial en lugar de agregar una nueva.

```
<nav>
  <ul>
    <li>
      <Link to="/">Home</Link>
    </li>
<li>
  <Link to="/about">About</Link>
</li>
<li>
  <Link to="/users">Users</Link>
</li>
<li>
  <Link to="/hola-mundo">Hello</Link>
</li>
  </ul>
</nav>
```

Agregamos el siguiente código en nuestro componente *App.js* dentro del **div** con la clase App, ahora podemos ver un menú de navegación en nuestra aplicación. Me falto mencionar debemos hacer el **import** de Link.

```
// App.js
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from "react-router-dom";
```



Hello About

REDIRECT

Con este componente podemos causar un redireccionamiento a una ruta diferente a la ruta actual reemplazando el **location** actual y el historial de navegación. Tiene las siguientes propiedades.

From: le pasamos un string u object para indicarle desde donde se va hacer el redireccionamiento.

To: le pasamos un string u object hacia dónde vamos a realizar el redireccionamiento.

Push: si es verdadero no modifica el location del historial por el contrario agrega esta nueva locación al historial.

```
<Redirect from="/redirect" to="/about"/>
```

Agregamos ese componente dentro de **Switch**, ahora si vamos al navegador y en la barra de dirección escribimos lo siguiente http://localhost:3000/redirect vamos a ver que nos hace un redirect a la ruta /about. No olvides hacer el **import** de Redirect.

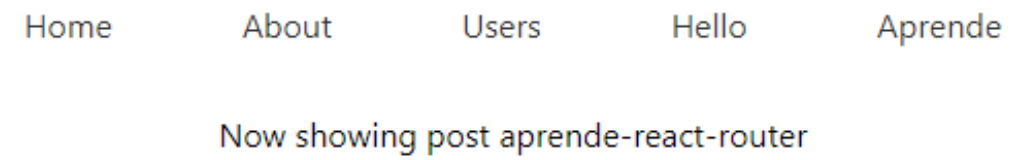
HOOK USEPARAMS

Ahora vamos a ver unos Hooks que nos van a ayudar a realizar operaciones con los componentes del navegador.

Con este **Hook** podemos acceder a los **params** de las rutas veamos un ejemplo para eso debemos crear un nuevo componente el cual se va encargar de usar el hook y debemos agregar un nuevo link al menú de navegación al igual que un Route

```
// App.js
import Blog from './pages/Blog'
// App.js
<li>
  <Link to="/blog/aprende-react-router">Aprende</Link>
</li>
<Route exact path="/blog/:slug" component={Blog}></Route>
// pages/Blog.js
import React from 'react'
import { useParams } from 'react-router-dom'
const Blog = () => {
  let { slug } = useParams()
  return (
    <section className="Blog">
      <p>Now showing post {slug}</p>
    </section>
  )
}
export default Blog
```

Lo primero es importar el hook y obtener el params que definimos en el Route, con este hook es muy fácil acceder al params que indicamos en el Route si vamos al navegador y damos click en el nuevo link que tiene como nombre aprende vamos a ver lo siguiente.



Now showing post aprende-react-router

HOOK USELOCATION

El hook useLocation devuelve el objeto location, este representa la URL actual. Puedes pensar en un objeto como un useState que devuelve una nueva location cada vez que cambia la URL.

Esto podría ser realmente útil, por ejemplo, en una situación en la que nos gustaría activar un nuevo evento de "vista de página" utilizando su herramienta de análisis web cada vez que se carga una nueva página, como en el siguiente ejemplo:

```
import React from "react";
import ReactDOM from "react-dom";
import {
  BrowserRouter as Router,
  Switch,
  useLocation
} from "react-router-dom";
function usePageViews() {
  let location = useLocation();
  React.useEffect(() => {
    ga.send(["pageview", location.pathname]);
  }, [location]);
}
function App() {
  usePageViews();
  return <Switch>...</Switch>;
}
ReactDOM.render(
  <Router>
    <App />
  </Router>,
  node
);
```

HOOK USECONTEXT

El hook **useContext nos sirve para poder implementar la API de Context** que ya existía en React desde antes de los hooks.

```
const value = useContext(MyContext);
```

Para quien no esté familiarizado, Context nos permite comunicar props en un árbol de componentes sin necesidad de pasarlos manualmente a través de props.

Algunos ejemplos de cuándo utilizar Context son:

- Un Tema de la UI (light theme, dark theme, etc).
- Autenticación del usuario.
- Idioma preferido.

Pero no es limitado a los ejemplos anteriores. Puedes aplicar Context a las necesidades de tu aplicación según tu criterio.

En otras palabras, no es obligatorio que Context maneje un "estado global" de la aplicación: Puede abarcar sólo una parte del estado si lo deseas.

A continuación vamos a ver ejemplos y comparaciones de cómo utilizar este hook para que puedas aprovechar todo su potencial.

HOOK USECONTEXT VS CLASE

Primero implementaremos un Context para aplicar un tema en componentes usando la API tradicional y luego lo refactorizaremos usando useContext.

Esto es sólo para poder identificar las semejanzas y diferencias entre ambas opciones. No quiere decir que la manera tradicional sea obsoleta ni nada por el estilo.

Dicho lo anterior, comenzaremos creando un Context para manejar el tema del siguiente modo:

```
// theme-context.js

  export const themes = {
  light: {
      color: "#555555",
      background: "#eeeeee"
    },
   dark: {
      color: "#eeeeee",
     background: "#222222"
   },
   vaporwave: {
     color: "#ffffff",
     background: "#ff71ce"
   }
 };

 export const ThemeContext = React.createContext(themes.light);
```

En este caso tenemos un objeto para representar temas diferentes que podemos usar en nuestra UI.

El tema por defecto es themes.light que es pasado por parámetro a createContext.

Ahora necesitamos proveer este context a nuestro árbol de componentes:

```
export default function App() {
   const [currentTheme, setCurrentTheme] =
     useState(themes.light);
   return (
     <div className="App">
       <h1>React Context</h1>

       <ThemeContext.Provider value={currentTheme}>
        <MyButton>Hello World!</MyButton>
      </ThemeContext.Provider>
    </div>);}
```

Estamos creando una variable de estado con useState para obtener el valor actual del tema.

Usamos ThemeContext.Provider value={currentTheme} para asignar el context y a partir de aquí, todo el árbol de componentes hijos van a poder tener acceso a context si lo desean. Aquí estamos haciendo uso de un Provider.

Cada objeto Context viene con un componente Provider de React que permite que los componentes que lo consumen se suscriban a los cambios del contexto.

El componente Provider acepta una prop value que se pasará a los componentes consumidores que son descendientes de este Provider. Un Provider puede estar conectado a muchos consumidores. Los Providers pueden estar anidados para sobreescribir los valores más profundos dentro del árbol.

Para usarlo en un componente lo hacemos del siguiente modo:

```
class MyButton extends Component {
   render() {
     const theme = this.context;
     const style = {
       backgroundColor: theme.background,
       color: theme.color,
       border: "1px solid",
       borderRadius: 5
     };
    return <button style={style} {...this.props} />;
  }
}

// Esto es importante (asignar el context)
MyButton.contextType = ThemeContext;
```

Y con esto ya tenemos nuestro context funcionando.

Ahora si queremos modificar el context, en este ejemplo basta con hacer lo que sigue:

```
export default function App() {
    const [currentTheme, setCurrentTheme] = useState(themes.light);

    return (
      <div className="App">
        <h1>React Context</h1>
        <ThemeContext.Provider value={currentTheme}>
          <MyButton
            onClick={() => setCurrentTheme(themes.dark)}>
           Dark Theme
         </MyButton>
         <MyButton
           onClick={() => setCurrentTheme(themes.vaporwave)}>
           Vaporwave Theme
         </MyButton>
       </ThemeContext.Provider>
     </div>
   );
 }
```

Con esto ya funciona pero por lo común vas a tener la necesidad de modificar el context desde un componente hijo.

Para eso podemos modificar nuestro context para proveer su valor y además una función que sirva para modificar su valor.

Primero modificamos nuestro context:

```
export const ThemeContext = React.createContext({
   theme: themes.light,
   updateTheme: () => {}
});
```

Ahora tendrá un objeto con las dos propiedades antes dichas.

En donde hacemos uso de Provider necesitamos modificar también su valor:

```
 export default function App() {
    const [currentTheme, setCurrentTheme] = useState(themes.light);

    return (
      <div className="App">
        <h1>React Context</h1>

        {/* Nuevo value definido */}
        <ThemeContext.Provider
         value={{
           theme: currentTheme,
           updateTheme: setCurrentTheme
         }}>


         <MyButton
           onClick={() => setCurrentTheme(themes.dark)}>
           Dark Theme
         </MyButton>
```

```
          <MyButton
            onClick={() => setCurrentTheme(themes.vaporwave)}>
            Vaporwave Theme
          </MyButton>

         {/* Nuevo Boton sin onClick */}
          <MyButton>Light Theme</MyButton>
        </ThemeContext.Provider>
      </div>
    );
  }
```

En el componente MyButton vamos a hacer que si no recibe el prop de onClick, pondremos por default updateThemeque obtenemos de ThemeContext:

```
class MyButton extends Component {
    render() {
      const { theme, updateTheme } = this.context;
      const style = {
        backgroundColor: theme.background,
        color: theme.color,
        border: "1px solid",
        borderRadius: 5
      };

      const updateLightTheme = () => {
        updateTheme(themes.light);
      };

      const onClick = this.props.handleClick ||
        updateLightTheme;

      return <button
        onClick={onClick}
        style={style}
        {...this.props}
      />;
    }
  }
```

Con esto ya estamos actualizando context desde un componente hijo o de manera directa en el componente donde hacemos el provider.

USECONTEXT

Ahora vamos a hacer una nueva versión del código anterior para hacer uso de useContext y puedas apreciar las diferencias.

La definición de ThemeContext no necesitamos modificarla, ni el Provider.

Lo que nos va a servir useContext es para poder acceder al valor de un context en un componente funcional.

```
const MyNewButton = (props) => {
    const { theme, updateTheme } = useContext(ThemeContext);

    const style = {
      backgroundColor: theme.background,
      color: theme.color,
```

```
      border: "1px solid",
      borderRadius: 5
    };

  const updateLightTheme = () => {
    updateTheme(themes.light);
  };

  const onClick = props.handleClick || updateLightTheme;

  return <button onClick={onClick} {...props} style={style} />;
};
```

Como puedes ver, lo único que cambia es que ahora podemos obtener nuestro valor de context usando const { theme, updateTheme } = useContext(ThemeContext);.

Con esto cubrimos el tema de Context en sí mismo y el hook useContext.

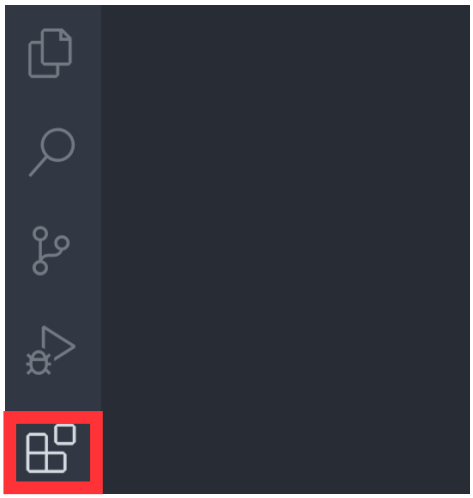
EJERCICIOS DE APRENDIZAJE

¡¡Vamos a ponernos manos a obra!! Ahora vamos a practicar lo visto en la guía con unos ejercicios de React. **Nota:** recomendamos que para cualquier vista que vayan a realizar usen BootStrap, de está manera podemos tener vistas de manera "rápida" y podemos trabajar con React.

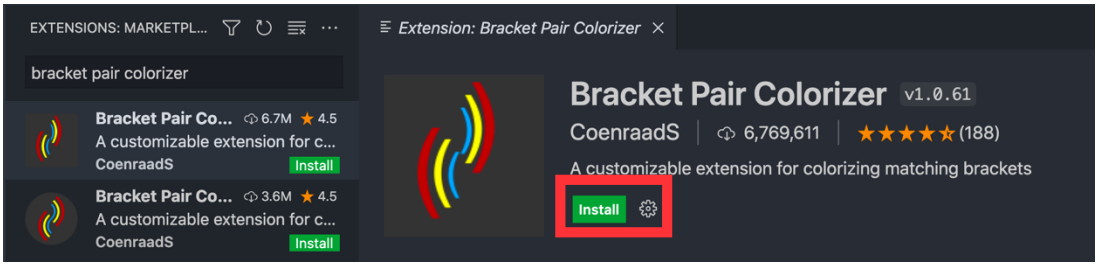
VIDEOS: Te sugerimos ver los videos relacionados con este tema, antes de empezar los ejercicios, los podrás encontrar en tu aula virtual o en nuestro canal de YouTube.

Importante: Para la realización de los ejercicios deberemos instalar las siguientes extensiones en Visual Studio Code: Bracket Pair Colorizer – Bootstrap 4, Font awesome 4, Font Awesome 5 Free & Pro snippets – Auto Close Tag – ES7 React/Redux/GraphQL/React-Native snippets – Error Lens – Auto Import – Auto Import ES6, TS, JSX, TSX.

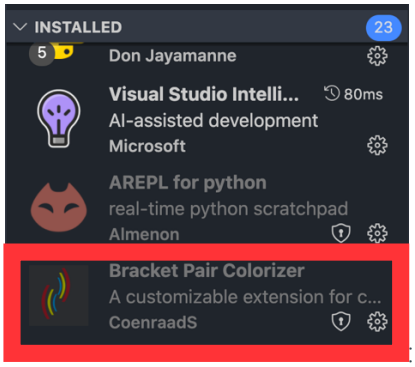
Para instalar una extensión en Visual Studio Code, haremos click en el siguiente botón:



Esto nos abrirá una pestaña que nos muestra las extensiones instaladas y nos da un buscador para instalar más extensiones. En ese buscador, buscaremos alguna de las extensiones mencionadas, cuando la encontremos, le daremos a Install.



Una vez que la hayamos instalado nos va a aparecer en nuestras extensiones instaladas



También deberemos instalar las siguientes extensiones en Google Chrome o el navegador que usen: Allow Cors – Json Formatter – React Developer Tools – Redux DevTools. Hecho esto ya podemos empezar con los ejercicios.

1. Crear un proyecto compuesto de un solo Functional Component. En dicho componente mostrar al menos dos datos, como por ejemplo titulo y subtitulo.

   El componente debe ser llamado desde App, a continuación, se propondrá la jerarquía del árbol de componentes y de como es el llamado desde index.js

   - Index.js
     - App
       - Ejemplo

```
export const Ejemplo = () => {

  const titulo = "Hello Dog";
  const subTitulo = "Sub titulo";

  return (
    <div>
      <h1>{titulo}</h1>
      <h2>{subTitulo}</h2>
    </div>
  );
};
```

2. Crear un proyecto compuesto por tres componentes bajo la misma jerarquía. Crear un Navbar, Main y Footer.

- Index.js
  - App
    - Navbar
    - Main
    - Footer

Necesitamos hacer que Footer, Main y Navbar muestren al menos un dato, de la misma manera que el ejercicio anterior.

3. Crear un Componente Main el cual llame dos veces a un mismo componente, es decir, que Main anide a Hijo e Hijo.

- Index.js
  - App
    - Main
      - Hijo
      - Hijo

Al primer Componente anidado pasarle como props el nombre Chiquito y al segundo el nombre Filomena. Desde los componentes Hijos atrapar los valores mediante las props.

```
//Envio dato desde donde lo llamo
<Hijo nombre="Chiquito" />;
<Hijo nombre="Filomena" />;

//Recibo dato
export function Hijo(props) {
  return <h1>Hola, {props.nombre}</h1>;
}
```

4. Crear un proyecto compuesto por 4 componentes bajo la siguiente jerarquía.

- Index.js
  - App
    - Navbar
    - Main1 o Main2
    - Footer

Al hacer click sobre las dos posibles opciones en el NavBar, se deberá cambiar entre Main1 y Main2 dependiendo de la navegación. Para lograrlo se deberá instalar y usar React Router Dom.

A continuación, se propone como será la jerarquía de los componentes

```
<div >
  <NavBar />
  <Switch>
    <Route exact path={"main1"} component={Main1} />
    <Route exact path={"main2"} component={Main2} />
  </Switch>
  <Footer />
</div>
```

En la siguiente imagen se vera como establecer la navegación. Se sugiere ver los videos explicativos que encontrarán en el canal de Youtube de Egg.

```
<Link to={"/main1"}>
  Main1
</Link>
```

5. Crear un proyecto compuesto de un solo componente y hacer uso de **useState** y mostrar el **state** del componente.

   Se podrá crear un **contador de clicks** o crear un input que mediante **onChange** cambie el valor del **state**. Cualquiera de las dos opciones, son validas para este ejercicio.

6. Crear un proyecto compuesto de un **solo componente y un servicio**, quien deberá ser capaz de llamar desde el servicio, mediante la **funcionalidad Fetch,** a la API de Rick and Morty (https://rickandmortyapi.com/api/character) .

   Una vez llamado los datos desde el servicio, hacer uso de useEffect en el componente creado, deberá mostrar una lista compuesta de los nombres de los 20 primeros personajes.