

FRONTEND

III

Certified Tech Developer

Primer Año
Tercer Bimestre

Contenido

Módulo 1: Primeros pasos a REACT	6
Introducción a React.....	6
¿Cómo luce un programa en React?	7
Agregando REACT a una página existente	8
Create Element.....	10
Estructura de React.createElement	10
Datos importantes	10
El DOM y el Virtual DOM	11
Agregando JSX	11
¿Qué es JSX?.....	11
Datos Importantes	12
Intención de JSX	12
Reglas para escribir JSX	12
Integración de JSX en React	13
Introducción a React Developer Tools.....	15
El inspector	15
Definir vs. usar componentes	16
Glosario de la semana I	16
Create React App	18
¿Qué es linting?	18
Beneficios de Create React App	18
¿Cómo ejecutamos Create React App?.....	18
Principales herramientas que instalamos con Create React App	20
Estructura de carpetas	22
Módulo 2: Componentes reutilizables	23
Componentes	23
Características	23
Componentes de clase vs. Funcionales.....	23
Ejemplos.....	24
¿Cómo interactúan los componentes entre ellos?.....	24
Props	26
Fragment y Children	28
Fragment.....	28
Children.....	30
Glosario de la semana II	31

Componentes Dinámicos	32
Map	32
Keys	34
CSS en componentes	36
Componentes con Estado.....	38
Componentes Stateful	39
State y setState	40
Clases y Objetos.....	41
Manejo del estado componentes de clase	42
Herencia, especialización y composición	44
Clases en React.....	47
Dato de color	49
Tips de la Semana III.....	50
Tips para CSS.....	50
Tips para map y keys	50
Tips para estructura de componentes de clase.....	51
Tips para state y setState.....	51
Ciclo de vida de un componente	52
Métodos más usados	53
Constructor.....	53
Render.....	53
componentDidMount().....	54
componentDidUpdate()	54
componentWillUnmount()	55
Otros métodos	55
getDerivedStateFromProps(props, state).....	55
shouldComponentUpdate(nextProps, nextState)	55
getSnapshotBeforeUpdate(prevProps, prevState).....	55
Eventos de Usuario	56
Implementar eventos en un componente de clase.....	57
Forma 1.....	57
Forma 2	57
Forma 3	58
Componentes controlados vs. No Controlados	58
Otros eventos: onSubmit y onChange	58
handleChange.....	58

handleSubmit	59
Validación.....	59
Alternativas en formularios y modales	59
Formik	59
SweetAlert	59
Glosario de la semana IV	60
Módulo 3: APIs y enrutamiento dinámico.....	62
APIs y enrutamiento dinámico	62
Efectos Secundarios	62
Rastreando nuestras peticiones.....	63
Asincronía	64
APIs	64
APIs y React	65
Fetch	65
Axios.....	66
Fetch vs Axios.....	68
Glosario de la semana V	69
Enrutamiento.....	70
Enrutamiento en el FrontEnd y SPA	71
Enrutamiento estático en React.....	72
Páginas anidadas y parámetros	77
Enrutamiento dinámico	80
Enrutamiento dinámico y páginas anidadas	84
React Router v5	85
Creando las páginas de la aplicación	86
Usando React Router v5	87
Diferencias entre enrutamiento estático y dinámico	89
Tips de la semana VI	90
Enrutamiento estático	90
Enrutamiento dinámico.....	91
Módulo 4: Introducción a Hooks	92
Introducción a Hooks	92
¿Qué son los Hooks?	92
¿Por qué Hooks?	92
Problemas en los componentes de clase.....	93
Hooks	94

Convivencia entre componentes de Clase y Hooks.....	95
Hooks nativos en React.....	95
¿Qué es useState?.....	96
Estructura de useState.....	97
Actualización	97
Ciclo useState	98
useEffect ()	98
Funcionamiento.....	100
Características	101
Casos de uso	101
Función de limpieza.....	103
Glosario de la semana VII	103

Módulo 1: Primeros pasos a REACT

Introducción a React

Imaginemos que tenemos que escribir una aplicación web y llegamos a un momento donde requerimos inputs, es decir, entradas de una persona usuaria para que se la aplicación actualice lo que se muestra la interfaz. En pocas palabras actualizar la UI.

La solución tradicional sería enviar los datos al backend y enseguida traer desde el backend el HTML con los cambios y dejar que el navegador se encargue solamente demostrarlos. Ahora la aplicación envía esos cambios en la UI al backend el cual nos devuelve más datos que deben actualizarse en diferentes secciones de la interfaz y, estas acciones, deben tener la misma información en forma sincronizada. Sin embargo, son decenas de variables las que tienen que actualizar sus valores y las secciones no muestran y manejan los datos de la misma manera, sino que cada sección tiene su propia forma de mostrar y manejar la información. ¿Cómo manejamos esto?

Una alternativa es manejar las actualizaciones de estado del lado del navegador disminuyendo la cantidad de veces que se traen los datos desde el backend y evitando totalmente las recargas de la página.

¿QUÉ SIGNIFICA UNA ACTUALIZACIÓN DE ESTADO? El estado es una representación del sistema en un momento, se refiere a los datos almacenados en la aplicación en forma de Strings, arrays, objetos, etcétera. Cada vez que una persona usuaria cambia algo se dispara un evento o cada vez que el backend inserta datos en el navegador el estado de la aplicación cambia con los nuevos valores, en este caso, manejar el estado con JS, HTML y CSS sería una tarea muy propensa a errores y muy costosa de mantener porque hay que rastrear constantemente todo lo que cambia y propagar los cambios en toda la interfaz. Es por eso que existe REACT.

REACT es una tecnología que fue creada en 2011 por el ingeniero de software de Facebook Jordan Walker y fue lanzada públicamente en 2013 por la necesidad de mantener una UI coherente en cuanto a su estado y donde las secciones estén pensadas como componentes reutilizables en vez de etiquetas HTML y código JS aislado.

¿QUÉ ENTENDEMOS POR COMPONENTES? Los componentes son una parte autosuficiente dentro de la interfaz de usuario, es decir, secciones que contienen toda la lógica de JS, enmarcado de HTML y CSS que se necesitan para mostrar una parte de la UI en una aplicación.

Podríamos pensar en una analogía como un tablero instrumentos, siendo el tablero la aplicación y los componentes los instrumentos que lo conforman. Si bien cada instrumento está conectado con los demás, cada uno tiene toda su funcionalidad y su interfaz de usuario auto contenido. Una característica elemental de REACT es que el interfaz de los componentes se expresa usando JSX que es una sintaxis declarativa tipo XML que se usa en JS internamente pero que se asemeja los tags de HTML. Funciona como una biblioteca o library de JS.

¿PORQUE USAR UNA LIBRARY Y NO UN FRAMEWORK? La principal diferencia entre un framework y una library está sobre el control del programa. Cuando se usa una library los desarrolladores están a cargo del flujo del programa y es el desarrollador quién escoge cuando y donde llamar a las funciones de la library. En cambio, cuando se usa un framework, es éste el que está a cargo del flujo del programa. El framework nos indica lo que debemos hacer y provee los lugares para insertar el código del usuario y llama al código cuando lo necesita.

Los dos frameworks más utilizados en la industria son ANGULARJS desarrollado por Google en 2010 y VUE.JS desarrollado por Even You en 2014. Si los comparamos en términos de uso y satisfacción por opinión de desarrolladores, veremos que REACT ha ido ganando preponderancia en el terreno durante los últimos años. La realidad es que, si pensamos en desarrollar una aplicación web sucesora de Facebook, Instagram, Netflix o Airbnb, por mencionar algunas, la comunidad desarrolladora Nos está dando un veredicto de conformidad muy claro.



¿Cómo luce un programa en React?

Estas son unas constantes típicas de JS utilizadas sobre elementos JSX “name” en una constante con el nombre “Heisenberg”. Mientras que element es un elemento de REACT expresado con JSX que encierra entre etiquetas la palabra “hello” seguido de una coma, seguido de una constante “name”.

La constante es interpretada por el parser o analizador de JSX gracias a las llaves que indican que lo que está dentro debe ser tratado como código JS. Por lo tanto, la expresión entre llaves debe ser una expresión de JS válida o tendremos un error.

```

1  const name = 'Heisenberg';
2      const element = <h1>Hello,
3      {name}</h1>;
4

```

En este segundo ejemplo notaremos una función típica de JS utilizada en un elemento JSX. “sayMyName” es una función de JS que toma un parámetro name y devuelve un String que utiliza este parámetro. Mientras que element un elemento de REACT expresado como JSX que encierra entre etiquetas la frase “Say my name” seguida de puntos suspensivos, seguida de una invocación a la función sayMyName definida arriba con un parámetro.

```

1  function sayMyName(name) {
2      return "You are " + name;
3  };
4
5  const element = <div>Say my
6  name... { sayMyName("Heisenberg")
7  }</div>;
8
9

```

Mantener una UI compleja no es una tarea fácil de hacer, pero gracias a REACT lo podemos llevar adelante de la forma más práctica, rápida y robusta.

Agregando REACT a una página existente

¿Cómo se crean los elementos en REACT y como se expresan? Actualmente REACT usa una sintaxis propia llamadas JSX pero antes de su existencia la creación de elementos se parecía más un código JS invocando la función createElement.

`createElement` es una función de REACT que básicamente crea un nuevo elemento. El primer argumento esta función representa el tipo de elemento, el segundo argumento es opcional y se conoce props por properties; este es un objeto de JS para especificar los atributos de ese elemento. El tercer argumento, también es opcional, es un array de los elementos que estarán dentro del elemento padre.

```
1 React.createElement(  
2   type,  
3   [props],  
4   [...children]  
5 );
```

Es importante resaltar que si bien createElement acepta un componente REACT como primer argumento esto sólo crea el componente, pero el componente en sí, ya debe estar definido como una función o clase en alguna otra parte del código.

[¿CÓMO USAMOS REACT.CREATEELEMENT?](#) Antes de empezar con REACT escribiremos un código esencial conocido como **boilerplate** independientemente de REACT y de cualquier otra biblioteca o framework.

Tenemos una estructura básica de HTML, necesitamos tener REACT disponible en el navegador, también es necesario poder imprimir los elementos creados con REACT en el navegador. En general **imprimir** se conoce como **renderizar**. Agregaremos el código mínimo para lograr esto qué consiste en vincular dos Script de JS a la página. El primer script descarga el paquete de REACT que da acceso a la biblioteca de REACT y el otro, descarga el paquete de REACT DOM que da acceso a la API de REACT DOM.

```
1 <!DOCTYPE html>  
2 <html lang="es">  
3  
4 <head>  
5   <meta charset="UTF-8"/>  
6   <meta name="viewport" content="width=device-width,  
7 initial-scale=1"/>  
8   <title>Creando Elementos con React</title>  
9 </head>  
10  
11 <body>  
12 <script crossorigin  
13 src="https://unpkg.com/react@17/umd/react.development.js"></script>  
14 <script crossorigin  
15 src="https://unpkg.com/react-dom@17/umd/react-dom.development.js">  
16 </script>  
17  
18 </body>
```

La única función que utilizaremos de REACT DOM es render. Ahora crearemos un <div></div> al que le asignaremos el id “root” o “raíz”, si bien el id puede ser cualquier String o número entero la convención es identificarlo como root. Llegamos a la parte en que usaremos createElement, para esto crearemos dos párrafos y un <div> anidando los dos párrafos dentro del mismo. Por último, renderizamos este árbol dentro del div al que identificamos como root.

Primero veamos cómo crear los elementos, usaremos la función React.createElement y le pasaremos 3 parámetros: el string “p”, luego un objeto de JS con un atributo llama **key** qué es el necesario para que REACT pueda mantener la referencia correcta a este elemento cuando se usa como un hijo de otro elemento y, por último, pasaremos el contenido del párrafo que en este caso es sólo un string.

```
1 const p_1 = React.createElement(  
2     "p",  
3     { key: 1 },  
4     "Soy un paragraph"  
5 );
```

El segundo párrafo se crea de manera similar la diferencia está en el valor del atributo key que debe ser distinto al anterior. De hecho, ningún valor asignado a key puede repetirse en el elemento dentro del mismo parent. El valor de atributo puede ser un número entero o un string.

Por último, crearemos un elemento, para esto, pasaremos como primer parámetro el string <div> no pasaremos ningún objeto con atributos por lo que debemos pasar null y por último pasaremos un array con los hijos que tendrá el <div> que son los dos elementos recién creados.

```
1 const div_el = React.createElement(  
2     "div",  
3     null,  
4     [p_1, p_2]  
5 );
```

El paso final es renderizar estos elementos dentro del <div> identificado como root para esto, invocaremos la función de render de ReactDOM con los dos parámetros: el primero que es el elemento que contiene todo el árbol de elementos que deseamos renderizar y el segundo que es el <div> dentro del que queremos renderizar ese árbol.

Convenientemente hemos ido almacenando el retorno de React.createElement en cada caso que lo hemos utilizado, nuestro primer parámetro para la función render será el elemento que contiene todo el árbol, para obtener el segundo parámetro utilizaremos la función **getElementsByld** que le pertenece al DOM. A esta función se le debe pasar el ID del elemento que queremos obtener. Este código se representa generalmente en una sola línea.

```
ReactDOM.render(div_el, document.getElementById("root"));
```

En resumen, lo más importante de conocer React.createElement radica en que permite prescindir de la configuración de la compilación de JSX en nuestro ambiente desarrollo. El otro factor importante es que en realidad el navegador no entiende JSX sino que tras bastidores, en un proceso conocido como **transpilación**, el código JSX es transformado en llamadas a la función React.createElement

Create Element

Actualmente React usa una sintaxis propia llamada JSX, pero antes de su existencia, la creación de elementos se parecía más a un código JavaScript invocando la función createElement.

createElement es una función de React que crea:

Un elemento	Un componente	Un fragmento
<ul style="list-style-type: none">• El cual puede ser un string que se refiera a una etiqueta HTML, como por ejemplo 'div', 'p' o 'span'	<ul style="list-style-type: none">• Ya sea una clase o una función	<ul style="list-style-type: none">• Que es un contenedor genérico

Es importante resaltar que, si bien createElement acepta un componente de React como primer argumento, el componente en sí debe ya estar definido como una función o class en otra parte del código.

Estructura de React.createElement

```
React.createElement(  
  type,  
  [props],  
  [...children]  
)
```

PRIMER ARGUMENTO: representa el tipo de elemento.

SEGUNDO ARGUMENTO: es opcional y se conoce como props (por properties) y es un objeto JavaScript para especificar los atributos de ese elemento.

TERCER ARGUMENTO: también opcional, es un array con los elementos que estarán dentro del elemento padre.

[¿POR QUÉ ES IMPORTANTE?](#) Quizá lo más importante de conocer createElement radica en que permite prescindir de configurar la compilación de JSX en nuestro ambiente de desarrollo. El otro factor importante es que, en realidad, el navegador no entiende JSX, sino que, tras bastidores, en un proceso conocido como transpilación, el código JSX es transformado en llamadas a la función React.createElement, por lo que es importante entender qué es lo que en realidad sucede.

Datos importantes

Podemos crear una referencia a la función React.createElement y usar esa referencia como atajo para escribir menos:

```
const create = React.createElement;  
  
const myP = create('p', null, 'Hello World');  
const myDiv = create('div', null, myP);
```

En el código de arriba, en lugar de escribir React.createElement cada vez que lo necesitemos, creamos una constante que apunta a la función y luego usamos esa constante para crear nuestros elementos. El primer elemento es un tag <p>, sin props (pasamos null), y como hijo le pasamos el

archiconocido string “Hello World”; el segundo elemento es un <div>, sin props (nuevamente pasamos null), y como hijo le pasamos el paragraph recién creado.

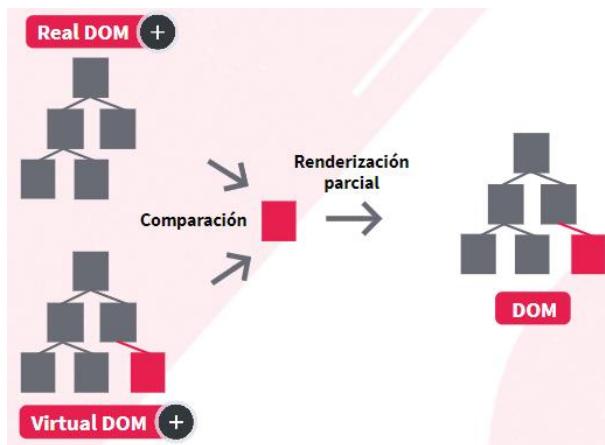
Recordemos que la función getElementById no es parte de la especificación ECMAScript, sino que está disponible dentro de document que es una propiedad del objeto global window que ofrece el DOM. El DOM (Document Object Model) es una API para trabajar con documentos HTML y XML.

Clase 2: Herramientas y conocimientos útiles

El DOM y el Virtual DOM

DOM: el DOM es una interfaz de objetos que interpreta nuestro HTML. Es el mismo que estudiamos cuando vimos JavaScript en el FrontEnd. Sirve para acceder a nuestros elementos y poder manipularlos.

VIRTUAL DOM: el Virtual DOM es una interpretación liviana del DOM. La biblioteca de JavaScript, React, permite la comparación del Virtual DOM con el DOM, y hace modificaciones solo en los elementos que hayan cambiado. Este proceso requiere menor procesamiento ya que no precisa volver a renderizar todo el DOM, sino que solamente actualiza los elementos que hayan sufrido cambios.



Agregando JSX

JSX son las siglas de JavaScript XML, y es una extensión de la sintaxis de JavaScript, es decir, que nos permite escribir JavaScript de otra forma diferente a la manera estándar. JSX no es un requisito para usar React, pero es muy conveniente usar esta sintaxis porque se puede escribir HTML directamente dentro de código JavaScript.

En el corazón de React está la idea de que en realidad la lógica de renderizado (o representación visual) está intrínsecamente acoplada con la lógica encargada de cómo manejar los eventos, cómo cambiar el estado de la interfaz con el tiempo y cómo preparar los datos para su visualización.

Por eso, React separa los aspectos (concerns) en unidades débilmente acopladas llamadas "componentes" que contienen tanto el marcado (HTML) como la lógica (JavaScript), en lugar de separarlos artificialmente en archivos .js y HTML.

¿Qué es JSX?

JSX es una extensión a ECMAScript que provee una forma más conveniente (syntactic sugar) de crear elementos que llaman a la función React.createElement. JSX, es una sintaxis similar a XML/HTML que extiende ECMAScript.

La comunidad de desarrolladores que trabajan con React prefiere utilizar JSX en lugar de createElement, por lo que el código más actual en la industria estará totalmente, o casi en su totalidad, escrito con JSX. Por lo tanto, es necesario entenderlo.

Datos Importantes

1. JSX significa JavaScript XML y no es una extensión estándar. También se conoce como JavaScript Syntax Extension, pero el nombre más aceptado es JavaScript XML.
2. JSX permite crear estructuras tipo árbol compuestas por HTML o elementos de React como si fueran variables de JavaScript.
3. En realidad, JSX no es absolutamente necesario, pero es la forma actual favorecida por la comunidad porque es más cómoda (syntactic sugar) para crear elementos, mostrar errores y advertencias.

No es necesario usar JSX para programar en React, y quizás lo más llamativo es que en realidad, desde React v17.0 tampoco es necesario usar React para poder usar JSX. Para [transpilar JSX a JavaScript](#) estándar se pueden usar varios transpiladores:

- **React JSX:** Crea elementos de React usando JSX.
- **Jsx-transform:** Implementación configurable de JSX desacoplada de React.
- **Babel:** Un “traductor” / compilador de ES2015 y posteriores al ECMAScript de ahora con soporte para JSX.

Intención de JSX

La intención con JSX es que sea utilizado por transpiladores que conviertan los tokens JSX en JavaScript estándar.

Reglas para escribir JSX

Los componentes definidos por el usuario deben empezar con mayúscula. De hecho, React recomienda nombrar los componentes con una letra mayúscula. Si se tiene un componente que comienza con una letra minúscula, debe asignarse a una variable en mayúscula antes de usarlo con sintaxis JSX. Por ejemplo:

```
/* ;No hacer! Este es un componente, por lo que la primera letra
   debe ser mayúscula */
function miComponente() {
    /* El uso de div en minúscula sí es correcto
       porque div es una etiqueta válida de HTML */
    return <div>Soy un componente React</div>;
}

function miFuncion() {
    /* ;No hacer! React piensa que <miComponente /> es una etiqueta HTML
       porque no empieza con mayúscula */
    return <miComponente />;
}
```

Para los elementos HTML que tienen etiquetas de cierre automático como ``, `<hr>`, `<input>` y `
`, la barra diagonal antes del corchete angular de cierre es obligatoria en JSX (aunque es opcional en HTML). Por ejemplo, esto funcionará: `<input />`

Las expresiones entre llaves se evalúan como JavaScript, por lo que deben ser expresiones válidas de JavaScript. Por ejemplo, la siguiente línea produce un paragraph con el contenido $2 + 3 = 5$ porque la porción entre llaves `{2 + 3}` evalúa a 5: `<p> 2 + 3 = {2 + 3} </p>`

JSX acepta anidación, pero la expresión debe tener solo un elemento externo. Por ejemplo, esto funciona:

```
const headings = (
  <div id = "outermost-element">
    <h1>Soy un encabezado h1</h1>
    <h2>Soy un encabezado h2</h2>
  </div>
);
```

Las propiedades (props) en JSX utilizan la convención de nomenclatura de camelCase en lugar de la usada para nombres de atributos en HTML, porque se basan en la API DOM, no en las especificaciones del lenguaje HTML. Por ejemplo, se debe usar `className` y `tabIndex` en lugar de `classname` y `tabindex` respectivamente, igual que cuando usamos JavaScript para manipular el DOM. Por ejemplo, esto funciona:

```
const hola = <h1 className="welcome"> Hola Mundo </h1>
```

Esto también es cierto para los manejadores de eventos en JSX. Por ejemplo, esto funciona:

```
<button onClick = {handleClick}>Hazme click</button>
```

Integración de JSX en React

JSX es una [sintaxis declarativa](#) que facilita expresar estructura de árbol para organizar componentes en la interfaz de usuario. Al usar JSX se pueden escribir estructuras similares a HTML Y XML, como, por ejemplo, estructuras de árbol similares al DOM el mismo archivo donde se escribe el código JavaScript

Partiendo de una estructura básica de HTML, agregamos REACT para poder usarlo en el navegador. El código mínimo para lograr esto consiste en vincular 3 script de JavaScript a la página. El primer script descarga el paquete de REACT que da acceso a la biblioteca de REACT y el otro, descarga el paquete de REACT DOM que da acceso a la API de REACT DOM. La única función que utilizaremos de REACT DOM es `render`. El tercer script permite utilizar JSX en el navegador puesto que descarga Babel que es uno de los compiladores de JavaScript más utilizados y éste se encarga de transpilar JSX a JavaScript estándar.

```
1  <!DOCTYPE html>
2  <html lang="es">
3
4  <head>
5    <meta charset="UTF-8">
6    <meta name="viewport" content="width=device-width, initial-scale=1">
7    <title>Creando Elementos con JSX</title>
8  </head>
9
10 <body>
11  <script crossorigin
12  src="https://unpkg.com/react@17/umd/react.development.js"></script>
13  <script crossorigin
14  src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"></script>
15  <script
16  src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
17 </body>
18
19 </html>
20
```

Ahora crearemos un `<div></div>` al que le asignaremos el id “root” o “raíz”, si bien el id puede ser cualquier String o número entero la convención es identificarlo como root. Llegamos a la parte en que usaremos JSX. Es esencial que para que nuestro navegador pueda entender JSX escribimos el código dentro un Script de tipo text/Babel (`<script type="text/babel"></script>`) en lugar de text/JavaScript. De no hacer esto, el código JSX no funcionará.

Ya estamos listos para crear nuestros elementos.

1. Crear dos `<p>` y un `<div>`
2. Anidar los `<p>` al `<div>`
3. Renderizar el árbol dentro del `<div id="root">`

¿CÓMO CREAR ESTOS ELEMENTOS CON JSX? Al elemento le debemos pasar un atributo “key” que es necesario para que React pueda mantener una referencia correcta a este elemento cuando se usa como hijo de otro elemento y, además, le pasaremos algo de contenido al párrafo. Ambos se crean de manera similar, la única diferencia radica en el valor de la key. Por último, creamos el div.

Ningún elemento dentro del mismo parente puede tener el mismo valor de key

Ahora bien, JSX nos ofrece una sintaxis similar HTML para meter los elementos dentro del elemento. La forma más aceptable y elegante es usar paréntesis y cerrar con un punto y coma. En JavaScript el punto y coma es opcional excepto en pocas ocasiones, pero es una buena práctica utilizarlo. Hay otras formas válidas de anidar componentes.

El paso final es renderizar estos elementos dentro del `<div id="root">`. Para esto invocaremos a la función render de React DOM con los dos parámetros.

ReactDOM.render(div_el, document.getElementById("root"));

El primero es elemento que contiene todo el árbol de elementos que deseamos renderizar y el segundo es el div dentro del que queremos renderizar ese árbol. Convenientemente almacenamos el `<div>` en una constante, esta será nuestro primer parámetro para la función render.

Para obtener el segundo parámetro utilizaremos la función getElementById que le pertenece al DOM. El código se expresa normalmente en una sola línea.

JSX facilita muchísimo entender y leer un código. Por qué utiliza sintaxis HTML para crear objetos de JavaScript a diferencia de React.createElement que utiliza sintaxis de función estándar de JS. Además, con JSX no hace falta tener archivos HTML por un lado y archivos JS por el otro, logrando su propósito de simplificar la creación de nodos de React y sus atributos.

Introducción a React Developer Tools

A la hora de depurar errores y entender como el navegador arma los componentes desde la perspectiva propia de React, es importante poder ver más allá del código HTML por esta razón es muy conveniente aprender a utilizar las herramientas de desarrollo de React.

Las herramientas de desarrollo de React mejor conocidas como [React Developer Tools](#) o React DevTools son una extensión para navegador diseñada para la biblioteca React. Estas instalan un inspector para los componentes de React junto a las herramientas de desarrollo nativas del navegador. De esta manera, el navegador muestra dos pestañas nuevas en las herramientas de desarrollo propias del navegador llamadas **componentes** y **profiler**.

PESTAÑAS COMPONENTES. Esta muestra los componentes de React que se renderizaron en la página, así como los subcomponentes que estos renderizaron. Al seleccionar uno de los componentes en el árbol se pueden inspeccionar y editar sus propiedades actuales y su estado. En la ruta de navegación se puede inspeccionar el componente seleccionado, el componente que lo creó, el componente que creó a este y así sucesivamente.

PESTAÑA PROFILER. Permite registrar información del rendimiento de los componentes.

Sin la ayuda de una herramienta como React DevTools el desarrollador puede apoyarse en las herramientas de desarrollo propias de su navegador. Sin embargo, el navegador mostrara los renderizado por los componentes y no los componentes en sí.

Las React Devtools son muy valiosas porque permiten al desarrollador ver a los componentes en el navegador de la forma en que se escribieron, es decir, manteniendo la estructura de React, en vez de “traducidos” a elementos HTML, lo cual facilita mucho el desarrollo y depuración.

El inspector

La extensión funciona como una aplicación independiente dentro de las herramientas de desarrollo del navegador en la que se instale y proporciona un inspector, similar al inspector de HTML nativo del navegador, pero para inspeccionar la jerarquía de componentes de React.

El inspector revela el árbol de componentes de React que construye la página y para cada componente se pueden verificar sus propiedades, su estado, qué componente lo renderizó, qué eventos lo re renderizan, y permite evaluar la performance de los componentes.

Definir vs. usar componentes

Para finalizar los temas vistos esta semana, queremos hablar de un último tema: la diferencia entre definir componentes y usar o crear componentes.

Usar un componente es similar a usar una función o una clase. Para usar una función se escribe su nombre y se le pone paréntesis al final, y si necesita parámetros, se pasan dentro de los paréntesis. La función debe ya existir, es decir, debe estar definida, y su nombre debe estar accesible dentro del código que la quiere invocar.

Esto también sucede con un componente que se ha definido con una clase. La única diferencia es que se debe anteponer la palabra clave `new` al nombre de la función. Para que este código funcione la clase debe ya existir: debe estar definida y su nombre debe estar accesible dentro del código que la quiere usar.

Usar componentes en React no es muy distinto a usar funciones u objetos creados con el operador `new` en JavaScript. Las reglas son las mismas.

Tanto JSX como `createElement` son formas de usar los componentes, no de definirlos. Para definir los componentes se usan funciones y clases. Ahora, internamente, dentro de la función o clase sí se puede usar JSX o `createElement` para crear los elementos que queramos que nuestros componentes tengan.

Glosario de la semana I

UI: User Interface o interfaz de usuario. Una interfaz de usuario es el espacio donde ocurren las interacciones entre humanos y máquinas. No está limitada de ninguna manera a aplicaciones de software pues la necesidad de una interfaz de usuario existe en el campo del diseño industrial y donde sea necesaria la interacción humano-máquina. En software, la interfaz de usuario comprende los botones, cajas y áreas de texto, casillas de verificación (checkboxes), botones de radio (radio-buttons), y en general todas las formas de ingresar información. También incluye las partes donde se muestra información al usuario como etiquetas, gráficos, imágenes, iconos, etc.

UX: User eXperience o experiencia de usuario. Va más allá de la interfaz de usuario, y engloba los aspectos relacionados con la facilidad y rendimiento de la aplicación: los tiempos de respuesta; los mensajes al usuario; facilidad para encontrar la información; la cantidad de pasos para llegar a partes críticas de la aplicación; colores, tipo y tamaño de las fuentes, etc.

JSX: JavaScript XML.

LIBRARY: se traduce como biblioteca y es el sitio donde se alojan colecciones de documentos para ser usados públicamente. En nuestro caso, una biblioteca de software es donde se aloja la funcionalidad común para que pueda ser compartida por múltiples aplicaciones, en lugar de estar repetida por cada aplicación. Por eso la traducción correcta es biblioteca y no librería (lugar donde se compran libros).

FRAMEWORK: es una estructura básica de soporte para una construcción. En nuestro caso en particular, un framework de desarrollo es una estructura fija que sirve de marco para insertar la funcionalidad particular de la aplicación.

VANILLA JAVASCRIPT: el término `vanilla` (`vainilla` en castellano) se utiliza con mucha frecuencia en el mundo del software para referirse a algo puro sin agregados ni modificaciones.

COMPONENTES: la programación orientada a componentes es una técnica de desarrollo de aplicaciones de software basada en la combinación de componentes nuevos y preexistentes, como en una fábrica. No es nueva en lo absoluto: similar a otros paradigmas modernos como la programación orientada a objetos (data de la década de los sesenta) y la programación funcional (data de la década de los cincuenta), la programación orientada a componentes data de la década de los sesenta. La idea esencial del componente es que este debe encapsular y controlar su propio estado.

ESTADO DE LA APLICACIÓN: es el conjunto de los valores de cada variable que maneja la aplicación en un momento dado. Por ejemplo, en un videojuego, el estado serían los valores de las variables que almacenan información como las vidas, la fuerza, las armas, las municiones, etc., tanto del personaje del jugador como de los enemigos, y también todo lo referente al nivel y al mapa.

FRONTEND: es el código y el procesamiento de datos que ocurre en el navegador. No se limita a mostrar los datos en pantalla y a la parte visual, sino que también incluye llamados a APIs y el manejo del estado de toda la aplicación.
BackEnd: es el código y el procesamiento de datos que se hace fuera del navegador. El BackEnd no es simplemente un servidor. Puede comprender varias computadoras o incluso máquinas virtuales que simulan computadoras reales.

ARQUITECTURA: en FrontEnd en general, la arquitectura se refiere al modelo de la aplicación como un sistema. Incluye todos los componentes, cómo interactúan entre sí, el entorno en el que operan (contenedores, máquinas virtuales, plataformas en la nube) y los principios y decisiones utilizados para diseñarlos (OOP, FP, estándares). Normalmente, la arquitectura se separa en varias vistas o aspectos (concerns) destinadas a todos los stakeholders, desde los dueños del producto hasta los ingenieros.

NPM: es, a la vez, un registro público y gratuito de código JavaScript y un manejador de paquetes de Node.

YARN: es un manejador de paquetes de Node que se puede utilizar para acceder al registro público de npm.

PROPERTIES: props de un componente. Son pares de clave/valor que se le pasan a los componentes. Son equivalentes los atributos que se le pasa a los componentes HTML como, por ejemplo, `<div id="1">`, la expresión `id="1"` es un par clave/valor, donde la clave es `id` y el valor es `1`.

BOILERPLATE: se trata de secciones de código (o cualquier texto en general) que se repiten en varios lugares sin cambios significativos, y que se pueden reutilizar en nuevos contextos o aplicaciones.

ECMASCRIPT: es un lenguaje de programación de propósito general, estandarizado por Ecma International. Se trata de un estándar de JavaScript destinado a garantizar la interoperabilidad de las páginas web en diferentes navegadores web.
DOM: el modelo de objetos de documento es una interfaz multiplataforma e independiente de lenguaje de programación que trata a un documento XML o HTML como una estructura de árbol en la que cada nodo es un objeto que representa una parte del documento. El DOM representa un documento con un árbol lógico.

TRANSPILADOR: también llamado transcompilador, es un traductor que toma código fuente escrito en un lenguaje de programación como entrada y produce código fuente equivalente en el mismo lenguaje de programación o en uno diferente de salida.

COMPILADOR: es un programa que traduce código de un lenguaje de programación a otro. A diferencia de un transpilador, el nombre compilador se utiliza principalmente para programas que

traducen código fuente de un lenguaje de programación de alto nivel a uno de bajo nivel (comprendido por la máquina) para crear un programa ejecutable.

RENDER: la función render es donde React cambia el DOM. Para esto recopila la salida de las funciones createElement.

Clase 4: Creando una aplicación con React

Create React App

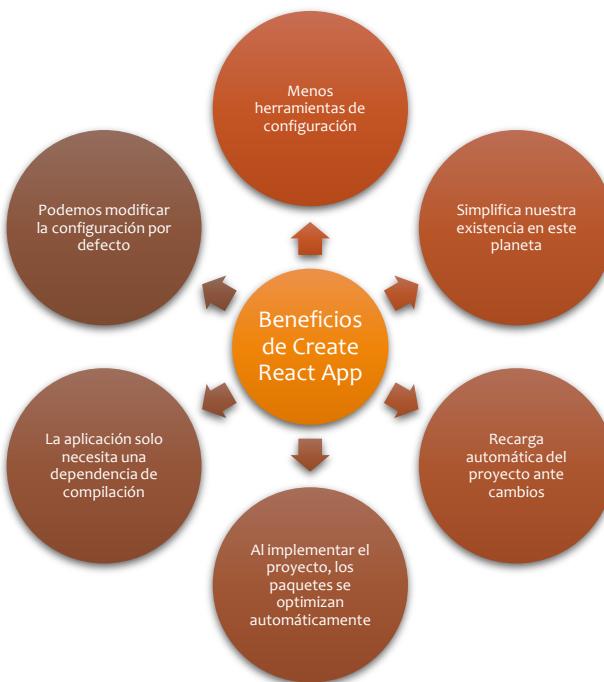
Create React App es un creador oficial de aplicaciones de React desarrollado por Facebook. CRA configura, mediante un solo comando, un ambiente de desarrollo de forma que podamos usar las últimas características de JavaScript, permitiendo una gran y simple experiencia de desarrollo, y optimizando nuestra aplicación para producción. Asimismo, genera proyectos con compilación y linting preestablecidos. También viene con un servidor de desarrollo y soporte de aplicación web progresiva (PWA) de primera clase.

¿Qué es linting?

Seguro se están preguntando qué es linting. Se trata de la detección de ciertos errores en el momento de desarrollo de aplicaciones (es decir, antes de su ejecución). Entre estos podemos encontrar:

- Errores de sintaxis.
- Código poco intuitivo o difícil de mantener.
- Uso de "malas prácticas".
- Estilos de código inconsistentes.

Beneficios de Create React App



¿Cómo ejecutamos Create React App?

Create React App se ejecuta mediante un comando. Esto quiere decir que Create React App es un **CLI (interfaz de línea de comandos)** que se puede ejecutar en una terminal independiente, en una

Powershell o CMD de Windows, o mismo desde la terminal provista en Visual Code, entre numerosas opciones.

El comando básico

El comando básico puede correrse con npx:

```
λ npx create-react-app my-app
npx: installed 114 in 4.308s

Creating a new React app in ~/my-app .

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts ...

yarn add v1.2.1
info No lockfile found.
[1/4] 🔎 Resolving packages...
[2/4] 🚀 Fetching packages...
[3/4] ⚙️ Linking dependencies...
[4/4] 🎯 Building fresh packages...
```

Un par de doritos después...

```
Happy hacking!
λ cd my-app
λ npm start

> my-app@0.1.0 start ~/my-app
> react-scripts start

Starting the development server...

Compiled successfully!

You can now view my-app in the browser.

  Local:          http://localhost:3000/
  On Your Network:  http://192.168.37.106:3000/

Note that the development build is not optimized.
To create a production build, use yarn build.

Compiling...
Compiled successfully!
```

O yarn:

```
yarn create react-app [nombre del proyecto]
```

Alternativamente, para usuarios más avanzados podemos crear una aplicación React en Typescript:

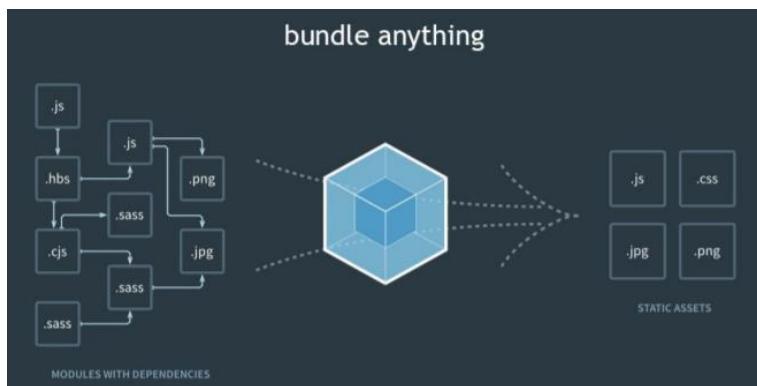
```
npx create-react-app [nombre del proyecto] --template typescript
```

```
yarn create react-app [nombre del proyecto] --template typescript
```

Principales herramientas que instalamos con Create React App¹

Webpack

Webpack es un empaquetador de aplicaciones que convierte archivos y módulos en un estático distribuible. En términos más simples, es un bundle, un conglomerado de varios recursos que una página web necesita para funcionar.

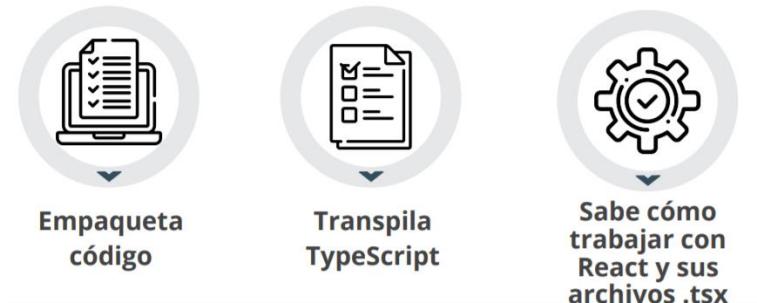


El componente obvio de un bundle es Js, pero en algunos casos también incluye CSS e imágenes, codificadas para su uso en data URLs. El bundle frecuentemente es solo un archivo.

Beneficios de un bundle:

- Solo requiere una solicitud a la red.
 - Puede superponer otras optimizaciones como la minificación y compresión de código.

Cuando se le dice a Webpack que empaquete una aplicación, este mira todos los archivos que encuentra, examina y determina qué archivos dependen de otros. Crea lo que se llama un gráfico de dependencias bundles de forma inteligente para tener la cantidad mínima de código requerida para que un sitio web o aplicación web funcione.



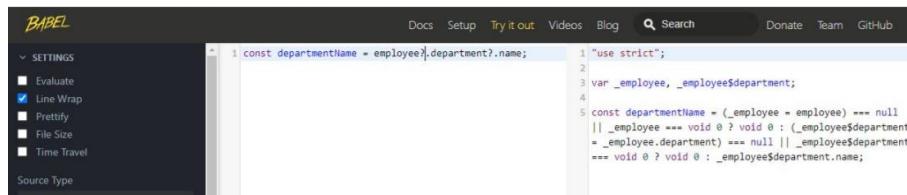
Lo mejor de Create React App es que nos evita tener que configurar una aplicación por nosotros mismos. Veamos un ejemplo de todo lo que hace Create React App con Webpack sin que tengamos que preocuparnos.

Por defecto, este código está oculto. Pero, para poder acceder a él, podemos ejecutar `npm run eject`.

¹ (además de React y ReactDOM)

Babel

Babel es un transpilador/traductor que al mismo tiempo transforma y compila cualquier código JavaScript escrito en versiones superiores a ECMAScript 2015+ a versiones de Javascript que los distintos navegadores del mercado puedan leer.

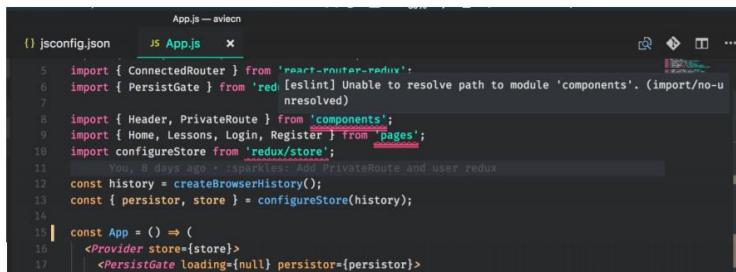


Cuando aparece una nueva versión de ECMAScript, los navegadores y el motor Js de Node necesitan tiempo para absorberla. Babel permite esto: tomará el código y generará un polyfill (refactorizar una característica nueva de un lenguaje utilizando características más antiguas de este).

¿QUÉ ES ECMASCRIPT? Es un estándar publicado por Ecma International. Contiene la especificación para un lenguaje de scripting de propósito general. Es decir, sirve como guía o referencia para estandarizar los avances del código de Javascript.

ESLint

ESLint es una herramienta de linting. Realiza análisis de código estático para identificar patrones problemáticos encontrados en el código JavaScript. Fue creado por Nicholas C. Zakas en 2013.

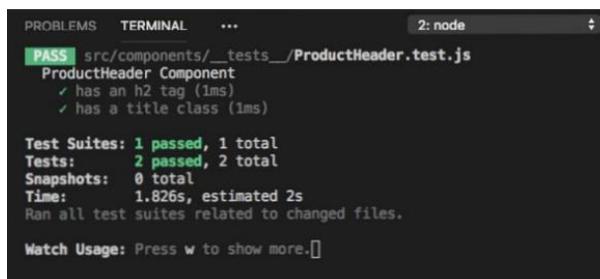


Jest

Jest es un marco de prueba de JavaScript mantenido por Facebook, Inc. diseñado y construido por Christoph Nakazawa con un enfoque en la simplicidad y el soporte para grandes aplicaciones web.

@testing-library

React Testing Library es una solución muy liviana para probar componentes de React. Proporciona funciones de utilidad ligeras con el fin de fomentar mejores prácticas de testeo. Su principio rector principal es: “Cuanto más se asemejen sus pruebas a la forma en que se utiliza su software, más confianza le pueden dar.”



Estructura de carpetas

Si bien existen distintas alternativas de creación de aplicaciones web React, pasando desde lo artesanal hasta bibliotecas similares a Create React App (CRA), actualmente CRA es amplio dominador como creador de aplicaciones React en el mundo empresarial. Por eso es importante recorrer y conocer su organización con detenimiento.

Comencemos a desarmar parte por parte todos los elementos que componen un proyecto en React en términos de carpetas y archivos relevantes. Así se debería ver el directorio de nuestro proyecto una vez ejecutados los comandos npm init react-app “mi-proyecto”, luego cd “mi-proyecto”, y por último npm start.

NODE_MODULES: Node modules almacena todos los paquetes npm que requiere nuestro proyecto de React para poder funcionar.

PACKAGE.JSON: Este archivo es el **manifiesto** del proyecto y le provee a node y a npm, nuestro gestor de paquetes, toda la info necesaria sobre los módulos que están instalados en el proyecto. También podemos ver el nombre, la versión, si es privado o público, las dependencias incluidas por default, los scripts de ejecución.

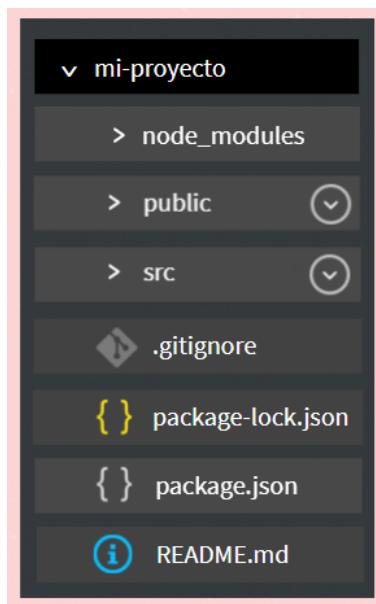
PACKAGE-LOCK.JSON: Este archivo se encarga de indicar Cómo desglosar todas las versiones de paquetes de node, es decir, describe las subdependencias de una dependencia, su integridad y la versión de node compatible con la misma.

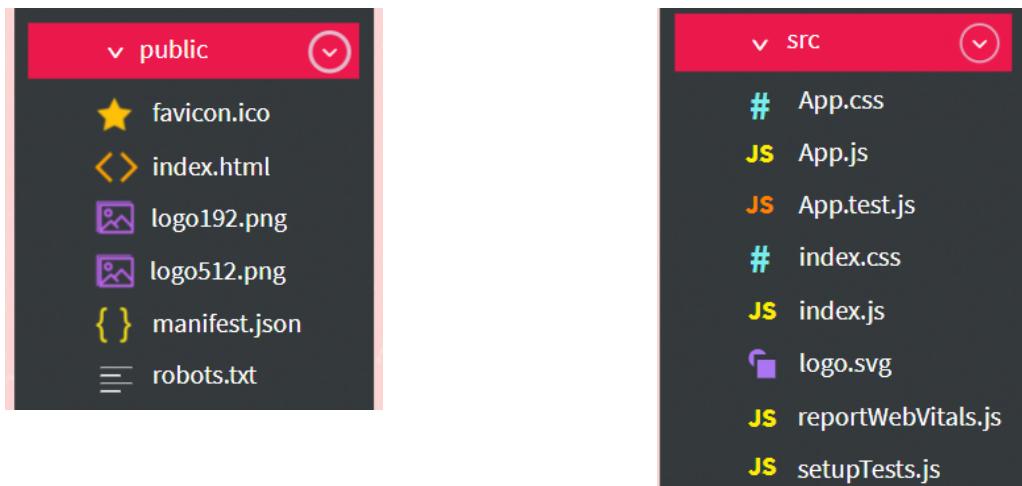
README: Nos da instrucciones e información útil sobre el proyecto, por ejemplo, cuando enviamos un proyecto a plataformas como Git Hub o Git Lab. En este archivo generalmente es donde se muestra la información sobre el proyecto. Puede incluir datos sobre scripts útiles para correr, probar o lanzar el proyecto. Además, incluye información relevante adicional para cualquier desarrollador qué puede utilizar el proyecto.

.GITIGNORE: Este nos muestra todos los archivos y carpetas que no deben ser agregados a un repositorio git. Cuando nosotros llevamos a GitHub o GitLab nuestro repositorio, mediante este archivo podemos excluir carpetas o algunos otros archivos.

PUBLIC: Contiene archivos estáticos de la aplicación. La carpeta public almacena principalmente el archivo index.html que va a ser el que se cargue cuando el usuario entre a la URL. Dentro de este archivo se encuentra un div con id=”root”, donde se va a insertar nuestra aplicación de React.

SRC: Es donde alojamos nuestra aplicación. Todo el trabajo en React se hace principalmente dentro de esta carpeta. Dentro del archivo index.js (principal) es que renderizamos el div con id=”root”. App.js se utiliza para implementar componentes de React y es donde vamos a iniciar la estructura de componentes en la aplicación.





Clase 5: Trabajando con componentes

Módulo 2: Componentes reutilizables

Componentes

Quizás una de las palabras que más se escucha nombrar al momento de introducirse en el mundo de React es "componente". Pero ¿qué es un componente?

Los componentes son piezas funcionales y fundamentales de la aplicación, ya que nos van a permitir separar las distintas partes que conforman la estructura de un sitio web en pequeñas piezas independientes y reutilizables. Estas están pensadas para trabajar de forma aislada, pero haciendo parte de un “todo”.

Técnicamente, los componentes son *funciones nativas de JavaScript* que se ejecutan cada vez que sea necesario. Como cualquier función, pueden estar atadas, o no, a una serie de argumentos que nos permiten generar que este bloque de código pueda ser reutilizable, estableciendo que genere partes de la interfaz de usuario con información realmente dinámica.

Un componente es una clase o una función que devuelve HTML a través de sintaxis JSX. Una vez definido, este se inscribe dentro del árbol de componentes de la aplicación. Aunque React tiene un fuerte enfoque sobre los componentes como una biblioteca, el ecosistema circundante lo convierte en un marco flexible.

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

Características

Anidación: Un componente puede ser mostrado dentro de otro.

Reusabilidad: Un componente bien construido puede reutilizarse en cualquier aplicación.

Configuración: Permite la posibilidad de configurarse en su creación.

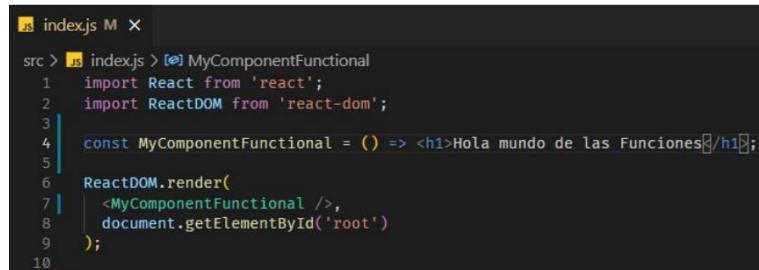
Componentes de clase vs. Funcionales

A partir de la versión 16.8 de React los componentes funcionales implementaron el uso de Hooks. Esta novedad permitió a los componentes funcionales poder brindar la misma utilidad que los de clase con algunas ventajas relacionadas a su mayor simpleza y legibilidad y su menor peso, entre otras.

DE CLASE	FUNCIONALES
Es una clase de JavaScript que se extiende de React.Component	Es una función
Sintaxis menos breve y clara	Sintaxis más breve y clara
Uso this	No usa this
Uso constructor	No requiere constructor
Tiene ciclo de vida a través de métodos	Usa hooks a partir de la versión 16.8
Menos facilidad de reutilización	Más reutilizables

Tabla 1: Componentes de clase vs funcionales

Ejemplos

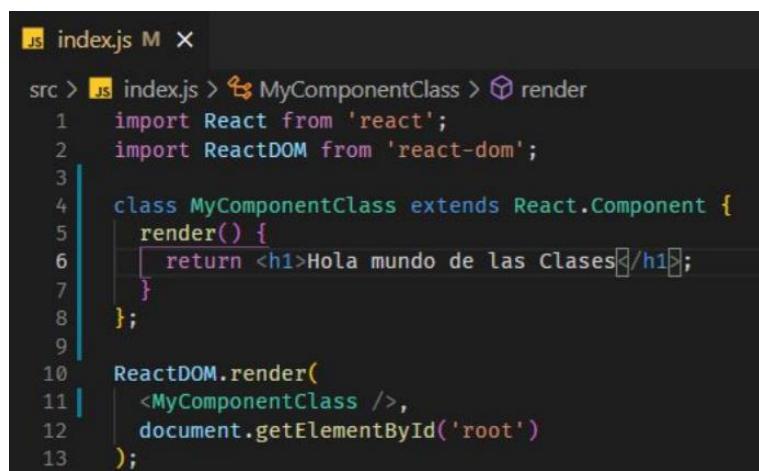


```

index.js M X
src > index.js > MyComponentFunctional
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 const MyComponentFunctional = () => <h1>Hola mundo de las Funciones</h1>;
5
6 ReactDOM.render(
7   <MyComponentFunctional />,
8   document.getElementById('root')
9 );
10

```

Ilustración 1: Componente de Clase



```

index.js M X
src > index.js > MyComponentClass > render
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 class MyComponentClass extends React.Component {
5   render() {
6     return <h1>Hola mundo de las Clases</h1>;
7   }
8 }
9
10 ReactDOM.render(
11   <MyComponentClass />,
12   document.getElementById('root')
13 );
14

```

Ilustración 2: Componente Funcional

¿Cómo interactúan los componentes entre ellos?

Así como en HTML tenemos etiquetas o tags que pueden tener hijos, en REACT también tienen esta relación sólo que se suelen separar en distintos archivos lo cual otorga reusabilidad y nos permite trabajar en el código de manera más ordenada y con un control mucho más detallado. Cuando hablamos de padres e hijos en REACT notaremos que los hijos son componentes que se exportan para ser utilizados dentro de los componentes padre estamos hablando en sí de **jerarquías**.

Supongamos que tenemos una verdulería y la dividimos en componentes: tenemos el componente verdulería qué es el que contiene todo y que, en este caso, sería el padre de todos los componentes. A su vez tendremos un componente cajón que contiene distintas variedades el componente fruta, en este caso el componente padre cajón es el que va a determinar qué tipo de fruta será el



componente hijo que, por su parte, tendrá como única función ser una fruta dentro del cajón. Por sí solo no sabemos qué fruta es y eso se lo indicará el cajón, como componente padre.

¿CÓMO SE VE ESTA RELACIÓN DE COMPONENTES EN EL CÓDIGO? Veremos al componente inicial de la aplicación llamado app el cual renderiza el componente padre. Podríamos decir que app es el abuelo de la familia en este ejemplo.

```
1 import Padre from
2 './components/Padre';
3
4 function App() {
5   return (
6     <Padre />
7   );
8 }
9
10 export default App;
11
```

Ilustración 3: El abuelo

Una vez en el componente padre vemos en principio la importación del componente hijo para ser utilizado y la declaración de dos constantes, la primera un array de frutas con cinco elementos y luego un título.

```
1 import React from "react";
2
3 import Hijo from "./Hijo";
4
5 const frutas = ["anán", "banana", "pera",
6 "manzana", "naranja"];
7 const titulo = "Listado de Frutas";
8
9
```

Ilustración 4: El padre

Si continuamos vemos el componente padre el cual retorna una etiqueta fragment que engloba una etiqueta que imprime la constante título y una lista desordenada dentro de la cual se llama a la función hijos. Esta simplemente mapea el array frutas y por cada elemento del mismo renderiza un componente hijo y le asigna atributos fruta como prop.

```
1 const Padre = () => {
2   const hijos = () =>
3     frutas.map((fruta) => <Hijo
4       fruta={fruta} />);
5   return (
6     <>
7       <h1>{titulo}</h1>
8       <ul>{hijos()}</ul>
9     </>
10   );
11 };
12
13 export default Padre;
14
```

Ilustración 5: Padre cont.

Cada componente hijo recibirá su fruta específica como prop y la renderizará dentro de una etiqueta y así el padre renderizará un hijo por cada fruta de la array.

```
1 import React from "react";
2
3 const Hijo = ({ fruta }) =>
4   <li>{fruta}</li>;
5
6 export default Hijo;
7
```

Gracias a React Developer Tools podemos ver esto desde el inspector de los elementos del navegador y hacemos click en la pestaña “componentes”. Allí podremos ver la estructura que creamos en el código.

Props

Las props son los datos internos de un componente. Representan información que es enviada al momento en el que un componente es utilizado. Estas permitirán que la información interna del componente sea variable para que podamos tener estructuras HTML realmente dinámicas y 100% reutilizables.

Las props representan información a las que un componente puede acceder para saber cómo o qué renderizar. Props es un objeto de JavaScript presente en cualquier componente gracias a que React las incluye como segundo argumento en `React.createElement()`.

Como primero y central que podemos decir de las props es que permiten el pasaje de información o atributos a los componentes. Estos atributos son pasados desde componentes padres a los hijos como un solo objeto, props. Esto refleja también el *carácter unidireccional* de los datos desde arriba hacia abajo.

Para especificar atributos o props en un componente podemos hacer algo como esto:

```
1  render() {
2    return(
3      <Hija nombre="Isabella" edad="24" />
4      <Hijo nombre="Marcelo" edad="12" />
5    )
6  );
7}
```

En este caso hemos pasado los atributos nombre y edad. El énfasis está en que podemos ver una especie de *relación de composición* donde un componente más específico representa uno más genérico y se configura con las props. Esta información se pasa de manera simple, algo similar a ponerle una id a un `<div>`.

Ahora veamos, por ejemplo, como el componente hijo recibe las props. Alternativamente podríamos desestructurar el objeto props y como resultado veremos lo siguiente:

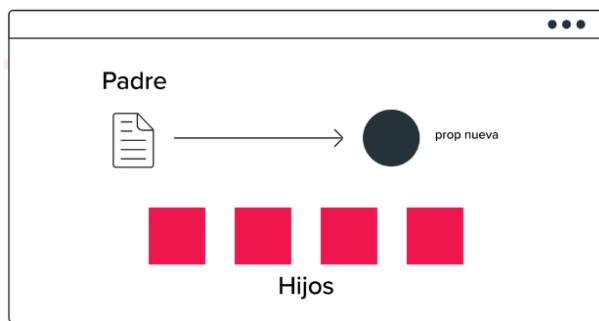
```
1  function Hija({nombre, edad}) {
2    return <h1>Hola, mi nombre es
3    {nombre} y mi edad es {edad}</h1>;
4  }
5
```

Para alcanzar esto, React utiliza un sistema de herencia entre la definición y la implementación del componente. Gracias a esto que ocurre tras bambalinas podemos tener acceso de forma directa al objeto props en la definición de un componente. Las props poseen cuatro características principales:

1. **SON INMUTABLES:** Como vimos en principio, las props se generan en el componente padre al pasar atributos a los hijos, por esta razón, React nos indica que debemos entenderlas como datos que el componente hijo no puede cambiar. Es por esto que sólo son de lectura.
2. **SON RECIBIDAS:** Teniendo en cuenta el sistema de herencia, los hijos las reciben y las utilizan.

3. **FACILITAN LA REUTILIZACIÓN DE COMPONENTES:** Ya que, en vez de tener un componente por hijo, podemos tener un solo componente hijos y reutilizarlo.
4. **SON PASADAS AL COMPONENTE HIJO CUANDO ÉSTE SE ESTÁ CREANDO.**

¿QUÉ PASA SI LAS PROPS SON MODIFICADAS DESDE EL MISMO COMPONENTE RAÍZ QUE LAS PROVEE? Un cambio de las props en un componente ameritaría su recreación en la interfaz gráfica. Es decir, el componente iniciaría un nuevo ciclo de vida: se va a recrear y se va a insertar en el DOM esperando por su desmontaje o destrucción.



Es importante reafirmar que para un componente hijo las propiedades no cambian después el render inicial del mismo, este tipo de flujo de arriba hacia abajo facilita la previsibilidad en el manejo de datos de la aplicación.

Caso de uso

Vemos un componente padre llamado `PokemonInList` el cual recibe de su propio padre la props “ítem” y que renderiza un `<div>` contenedor el cual, dentro del mismo, llama a la función `renderItems`. Esta última, mapea los ítems recibidos como props en el componente y asigna dos atributos: `id` y `name` a cada ítem que renderizaremos en un componente hijo, `PokeCard`.

<pre> 1 import React from 'react'; 2 3 import PokeCard from './PokeCard'; 4 import './PokemonInList.css'; 5 6 const PokemonInList = ({items}) => { 7 const renderedItems = () => 8 items.results.map(item => { 9 const id = item.url.split('/')[6]; 10 return <PokeCard id={id} name={item.name}> 11 }); 12 } 13 return <div 14 className="container">{renderedItems()}</div>; 15} 16 17 export default PokemonInList; 18 19 const renderedItems = () => 20 items.results.map(item => { 21 const id = item.url.split('/')[6]; 22 return <PokeCard id={id} name={item.name}> 23 }); 24 25 </pre>	<pre> 1 import React from "react"; 2 3 const url = 4 "https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/other/dream-world/"; 5 6 const PokeCard = ({ id, name }) => (7 <div className="card" key={id}> 8 <div className="pokemon"> 9 13 <div> 14 <h3> 15 {name} N°{id} 16 </h3> 17 </div> 18 </div> 19 </div> 20); 21 22 export default PokeCard; 23 24 </pre>
---	--

Digital house >

El componente `PokeCard` recibe las props de su padre y renderiza una card cada vez que se ha creado y el resultado final las 5 `PokeCards` se terminan visualizando así:



Lo maravilloso de las props es que permiten a la arquitectura y componentes de React evitar llamadas redundantes facilitando la creación de componentes encapsulados y reutilizables.

Fragment y Children

Fragment

Cuando se trabaja con componentes en React, es necesario retornarlos dentro de una etiqueta que los envuelva. Esto produce que en un HTML se creen etiquetas vacías innecesarias y lo resuelve Fragment. Una posible solución es agrupar todo en una <div>, pero generaría etiquetas extra una y otra vez. Para ello React nos da la posibilidad de usar Fragment y así utilizar un envoltorio que finalmente no generará un nodo extra en nuestro DOM.

Aquí un ejemplo de un array map que retorna un y un <p> envueltos en un <div> sin Fragment:

```
index.js M ✘ logo.svg
src > index.js > ...
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 const numbers = [1, 2, 3, 4, 5, 6, 7, 8];
5
6 const listItems = numbers.map((number) =>
7   

8     <span>Nº </span>
9     <p>{number}</p>
10    </div>
11  );
12
13 ReactDOM.render(
14   <div>{listItems}</div>,
15   document.getElementById('root')
16 );
17
18
19


```

```

•▼ <body cz-shortcut-listen="true"> == $0
  <noscript>You need to enable JavaScript to run this app.</noscript>
  ▼<div id="root">
    ▼<div>
      ▼<div>
        <span>Nº </span>
        <p>1</p>
      </div>
      ▼<div>
        <span>Nº </span>
        <p>2</p>
      </div>
      ▼<div>
        <span>Nº </span>
        <p>3</p>
      </div>
      ▼<div>
        <span>Nº </span>
        <p>4</p>
      </div>
      ▼<div>
        <span>Nº </span>
        <p>5</p>
      </div>
      ▼<div>
        <span>Nº </span>
        <p>6</p>
      </div>
      ▼<div>
        <span>Nº </span>
        <p>7</p>
      </div>
      ▼<div>
        <span>Nº </span>
        <p>8</p>
      </div>
    </div>
  <!!--

```

Aquí un ejemplo usando Fragment:

```

js index.js > ...
import React from 'react';
import ReactDOM from 'react-dom';

const numbers = [1, 2, 3, 4, 5, 6, 7, 8];

const listItems = numbers.map((number) =>
  <React.Fragment>
    <span>Nº </span>
    <p>{number}</p>
  </React.Fragment>
);

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

```

```

▼<body cz-shortcut-listen="true">
  <noscript>You need to enable JavaScript to run this app.</noscript>
  ▼<div id="root">
    <span>Nº </span>
    <p>1</p>
    <span>Nº </span>
    <p>2</p>
    <span>Nº </span>
    <p>3</p>
    <span>Nº </span>
    <p>4</p>
    <span>Nº </span>
    <p>5</p>
    <span>Nº </span>
    <p>6</p> == $0
    <span>Nº </span>
    <p>7</p>
    <span>Nº </span>
    <p>8</p>
  </div>

```

Es notorio el ahorro en líneas y la simpleza del código con el uso de Fragment. Por último, hay 2 formas de implementar Fragment: `<></>` o `<React.Fragment></React.Fragment>`

Children

Seguramente van a existir casos en los que tengamos un componente y queramos tan dinámico su uso que no va a ser suficiente con el simple uso de las props. Es aquí en donde cobran un particular protagonismo los children. A través de ellos vamos a tener la capacidad de enviar (como si fuera una prop) cualquier tipo de estructura HTML.

¿CUÁNDΟ USAMOS UN CHILDREN? En caso de no saber qué contenido puede llegar a haber dentro de un componente padre, es común que los componentes actúen como cajas o contenedores genéricos de otros componentes. En estos casos es recomendable usar la prop special children para pasar elementos hijos directamente en el resultado. Esto permite que otros componentes pasen hijos arbitrarios anidando el JSX.

```

js index.js > ...
import React from "react";
import ReactDOM from "react-dom";

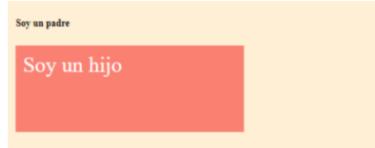
const Padre = (props) => {
  return (
    <div
      style={{
        width: "75%",
        background: "#FFEFD5",
        height: "200px",
        padding: "20px",
      }>
      <h5>Soy un padre</h5>
      {props.children}
    </div>
  );
};

const Hijo = (props) => {
  return (
    <div
      style={{
        width: "50%",
        background: "#FA8072",
        height: "100px",
        padding: "10px",
        color: "white",
        fontSize: "30px",
      }>
      {props.autor}
    </div>
  );
};

const App = () => {
  return (
    <Padre>
      <Hijo autor="Soy un hijo" />
    </Padre>
  );
};

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

```



Cualquier elemento que pasemos dentro de la etiqueta JSX se pasará a Padre como `{props.children}`. Como resultado obtendremos al Hijo envuelto en el contenedor Padre.

Clase 6: Revisión y Práctica II

Glosario de la semana II

CREATE REACT APP O CRA: Create React App es un creador oficial de aplicaciones de React desarrollado por Facebook.

BABEL: Babel es un transpilador, transforma cualquier código JavaScript escrito en versiones superiores a ECMAScript 2015+ a versiones de JavaScript que los distintos navegadores del mercado puedan leer.

WEBPACK: Es un empaquetador de aplicaciones que convierte archivos y módulos en un estático distribuible. En términos más simples, es un bundle, un conglomerado de varios recursos que una página web necesita para funcionar. El componente obvio de un bundle es Js, pero en algunos casos también incluye CSS e imágenes, codificadas para su uso en data URLs. Sin embargo, el bundle frecuentemente es solo un archivo. El beneficio más importante de un bundle es que, por

lo general, solo requiere una solicitud a la red. Además, un bundle puede superponer otras optimizaciones como la minificación y compresión de código.

ESLINT: Es una herramienta de linting. Realiza análisis de código estático para identificar patrones problemáticos encontrados en el código JavaScript.

JEST: Es un marco de prueba de JavaScript mantenido por Facebook, Inc., diseñado y construido por Christoph Nakazawa con un enfoque en la simplicidad y el soporte para grandes aplicaciones web.

@TESTING-LIBRARY: Es una solución muy liviana para probar componentes de React. Proporciona funciones de utilidad ligeras con el fin de fomentar mejores prácticas de testeo. Su principio rector principal es: “Cuanto más se asemejen sus pruebas a la forma en que se utiliza su software, más confianza le pueden dar”.

NPX: Es una herramienta de CLI que nos permite ejecutar paquetes de npm de forma mucho más sencilla.

COMPONENTE: Un componente es una clase o una función que devuelve HTML a través de sintaxis JSX. Una vez definido un componente, este se inscribe dentro del árbol de componentes de la aplicación. Aunque React tiene un fuerte enfoque sobre los componentes como una biblioteca, el ecosistema circundante lo convierte en un marco flexible. Permite separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada.

- **Fragment**: Cuando se trabaja con componentes en React, es necesario retornar los mismos dentro de una etiqueta que los envuelva. Esto produce que en un HTML se creen etiquetas vacías innecesarias. Esto lo resuelve Fragment.
- **Children**: Permite enviar (como si fuera una prop) cualquier tipo de estructura HTML. Cuando no sabemos exactamente qué contenido puede llegar a haber dentro de un componente padre, es común que los componentes actúen como cajas o contenedores genéricos de otros componentes. En estos casos, es recomendable usar la prop especial children para pasar elementos hijos directamente en el resultado. Esto permite que otros componentes pasen hijos arbitrarios anidando el JSX.
- **Props**: representan información a las que un componente puede acceder para saber cómo o qué renderizar. Las props son un objeto de JavaScript presente en cualquier componente, gracias a que React las incluye como segundo argumento en React.createElement().

Clase 7: Componentes Dinámicos

Componentes Dinámicos

Map

La función map de Array (Array.prototype.map) se usa para crear un nuevo array con los resultados de llamar a una función aplicada sobre cada elemento de un array. Dicho de otra forma, map aplica la función que se le indique en cada uno de los elementos del array y devuelve un nuevo array de resultados.

La función map nos permite mapear, o transformar, los elementos de un array en otros elementos para un nuevo array.

Veamos su sintaxis en forma de función arrow:

```
map((element) => { ... })
map((element, index) => { ... })
```

Y en forma de callback inline:

```
map(function callbackFn(element) { ... })
map(function callbackFn(element, index) { ... })
```

La función map acepta un array como tercer parámetro, pero la forma en que más se utiliza solo necesita los dos primeros parámetros.

Ejemplo

Dijimos que map se llama desde un array y que retorna un nuevo array de resultados. Supongamos que tenemos un array de personajes de Among Us y queremos tener un array que nos diga cuáles personajes no eran el impostor y cuál sí era el impostor. El array tiene objetos literales con el nombre del personaje y una bandera (flag) que indica si es tripulante o impostor:

```
const tripulantes = [
  {nombre: "Mr. Poindibags", esImpostor: true},
  {nombre: "Bombom", esImpostor: false},
  {nombre: "Tito", esImpostor: false},
  {nombre: "X-Ray", esImpostor: false},
  {nombre: "Fixfox", esImpostor: false},
]
```

Nuestra función map podría simplemente utilizar la información de cada tripulante para crear una cadena (string) con una frase como: “Mr. Poindibags era el impostor” o “Fixfox no era el impostor”. Podríamos utilizar un lazo for para crear un nuevo array a partir del primero, o podríamos usar map. La implementación con el lazo for podría ser así:

```
// Creamos un nuevo array
let tripulacion = [];

// Iteramos sobre cada elemento en el array tripulantes
for(let i=0; i<tripulantes.length; ++i) {
  let tripulante =
    `${ tripulantes[i].nombre } ${tripulantes[i].esImpostor ? '' : 'no '} era el impostor`;

  // Metemos cada tripulante en el nuevo array
  tripulacion.push(tripulante);
}
```

Por otro lado, para hacer esto con map, debemos invocar la función sobre el array tripulantes y pasamos como primer argumento una función que toma cada elemento (en este caso lo nombramos tripulante) y usa una plantilla literal (template literal) para leer los datos del elemento tripulante.nombre y tripulante.esImpostor y crear la frase:

```

let tripulacion = tripulantes.map(
  tripulante =>
    `${ tripulante.nombre } ${tripulante.esImpostor ? '' : 'no ' }era el impostor`
);

```

En cualquiera de los casos, el resultado será un array con los cinco elementos.

Como vemos, usar el lazo for es mucho más verboso, aunque muy eficiente, pero usar map nos permite escribir código más simple y es un buen ejemplo de programación funcional.

Keys

En React la palabra clave key es un atributo especial de tipo string que se debe incluir al crear listas de elementos. React lo utiliza para identificar cuáles elementos han cambiado, se han agregado o se han eliminado.

The diagram shows a code snippet for a list component named `Milista`. The component takes `props` and returns a `<div>` containing an `` element. Inside the ``, the `props.items.map` function is used to generate list items. Each item has a `key` attribute set to `{i+item}`. To the right of the code, a callout box contains the following text:

Las **keys** deben ser dadas a los elementos dentro del mapeo del **array** para darles una identidad única y estable.

Es muy importante asignar la propiedad key a los elementos dentro de una lista para que React pueda identificarlos con precisión. De no hacerlo, recibiremos una advertencia de que se debe proporcionar una clave para los elementos de la lista.

Ejemplo

Veamos el mismo ejemplo anterior, pero ahora con React. Supongamos que tenemos un componente de tipo `StatusTripulante` que muestra las frases y que deseamos formar una lista. Si usáramos un lazo for, tendríamos algo así:

```

const tripulantes = [
  {nombre: "Mr. Poindibags", esImpostor: true},
  {nombre: "Bombom", esImpostor: false},
  {nombre: "Tito", esImpostor: false},
  {nombre: "X-Ray", esImpostor: false},
  {nombre: "Fixfox", esImpostor: false},
];
// Creamos un nuevo array
let tripulacion = [];
for(let i=0; i<tripulantes.length; ++i) {
  let tripulante = <StatusTripulante {...tripulantes[i]} />;
}

```

```
    tripulacion.push(tripulante);
}

lista = <ul>{ tripulacion }</ul>;
```

La expresión {...tripulantes[i]} lo que hace es expandir la información dentro de cada tripulante y pasársela como props. Como vemos, usar for produce un código más largo y más bien complejo. En la industria se prefiere usar map de la siguiente manera:

```
const tripulantes = [
  {nombre: "Mr. Poindibags", esImpostor: true},
  {nombre: "Bombom", esImpostor: false},
  {nombre: "Tito", esImpostor: false},
  {nombre: "X-Ray", esImpostor: false},
  {nombre: "FixFox", esImpostor: false}
];

let tripulacion = tripulantes.map(
  tripulante => <StatusTripulante {...tripulante}>
)

lista = <ul>{tripulacion}</ul>
```

Nuevamente, la expresión {...tripulante} lo que hace es expandir la información dentro de cada tripulante y pasársela como props.

Como dijimos, si dejamos el código de esta manera recibiremos una advertencia sobre la falta de keys, pero no un error. ¿Qué sucede si hacemos caso omiso de la advertencia?

¡IMPORTANTE!

Lo que puede suceder es una de dos cosas: podríamos tener un muy bajo rendimiento de la aplicación al hacer cambios en la lista, o podríamos tener resultados inesperados (bugs) dependiendo del tipo de operaciones que realicemos sobre la lista.

Esto ocurre porque cuando React aplica su algoritmo de diferenciación (diffing algorithm) sobre dos árboles del DOM, primero compara los dos elementos raíz, y el comportamiento es diferente dependiendo del tipo de elementos raíz:

1. Siempre que los elementos raíz tengan diferentes tipos, React derribará el árbol actual y construirá el nuevo árbol desde cero.
2. Si los elementos raíz son del mismo tipo, React observará los atributos de ambos y mantendrá el mismo nodo DOM subyacente. Entonces, React solo actualizará los atributos modificados y enseguida recorrerá a los hijos de los elementos.

React recorre al nodo actual y al nodo nuevo al mismo tiempo y generará una mutación siempre que haya una diferencia entre los elementos.

Es en este punto donde se puede hacer una diferencia, porque si el usuario agrega, elimina, o actualiza algunos elementos, no se necesita recrearlos a todos, pero si React no identifica a cada uno de los elementos, los cambiará a todos en lugar de darse cuenta de que puede mantener intactos algunos, incluso quizás la mayoría.

Es por esto que React soporta el atributo key. Cuando los elementos de un mismo tipo dentro de un subárbol tienen keys, React las usa para hacer coincidir los elementos en el árbol original con los elementos del nuevo árbol. Esto es lo que le permite **ahorrar cambios innecesarios**.

¿Y qué podemos utilizar como key? Los atributos key pueden ser una propiedad ID que agreguemos, o alguna composición hecha con algunas partes del contenido del elemento. La key solo tiene que ser única entre hermanos, no globalmente. En nuestro ejemplo si tuviéramos un parámetro ID para nuestros tripulantes, nuestro código con **map** y **key** podría ser así:

```
let tripulacion = tripulantes.map(  
  tripulante => <StatusTripulante key={(tripulante.id)} {...tripulante}/>  
)  
  
lista = <ul>{ tripulacion }</ul>;
```

¿Pero qué sucede en el caso de que no contemos con un ID para cada elemento y no haya una forma fácil de crear un atributo key? En nuestro caso, cada tripulante tiene solo dos datos: un nombre y un flag llamado `esImpostor`.

No hay ID ni nada que podamos usar como tal, porque los nombres no tienen garantía de ser únicos. Es en estos casos que podemos usar el parámetro `index` del elemento dentro del array como key. Esto lo logramos usando el argumento `index` de la función que pasamos a `map`:

```
let tripulacion = tripulantes.map(  
  (tripulante, index) => <StatusTripulante key={index} {...tripulante}/>  
)  
  
lista = <ul>{ tripulacion }</ul>;
```

Usar el parámetro `index` del elemento dentro del array como key puede llegar a funcionar bien si los elementos de la lista nunca se reordenan, ya que los reordenamientos serán lentos. Además, tengamos en cuenta que cuando se utiliza el parámetro `index` del elemento dentro del array como key, los reordenamientos también pueden causar problemas con el estado de los componentes. Si el atributo key es el índice del elemento, entonces mover un elemento lo cambia a los ojos de React porque le modifica la key. Esto podría causar actualizaciones en formas inesperadas (bugs) en los componentes.

CSS en componentes

Tradicionalmente el modelo de desarrollo de una aplicación nos propone tener los diferentes tipos de código en archivos o incluso en directorios separados. Pero, en la medida en que los navegadores más poderosos, sumado a que podemos ejecutar frameworks y bibliotecas en los mismos se empezaron a utilizar algunas formas más convenientes. Uno de estos cambios ocurre en la forma en que se trata al CSS (hojas de estilos de cascada).

Existen dos aproximaciones modernas para trabajar con CSS: módulos y componentes estilizados, también conocidos como “style components”.

Los módulos CSS permiten escribir estilos en archivo CSS pero que son consumidos como objetos JS. Estos son muy populares ya que hacen que los nombres de clases y de animaciones sean únicos por lo que, no hay que preocuparse por las colisiones en nombres de los selectores.

Por su parte, los componentes estilizados, explotan las plantillas etiquetadas o “tagged templates” que introdujo ECMAScript para poder escribir códigos CSS real en los componentes. Esto elimina el mapeo entre componentes y estilos.

¿CÓMO SE IMPLEMENTAN ESTAS NUEVAS DINÁMICAS? Aunque ambos enfoques son válidos, trabajaremos con módulos ya que es CSS nativo, por lo que, la carga de estos módulos se hará en paralelo y además nos permitirá utilizar preprocesadores como SaSS. En cambio, style components es una biblioteca JS por lo que el navegador debe parsear código JS para aplicar los estilos.

Supongamos que tenemos una aplicación que muestra un listado de películas y, además, tiene un encabezado y un pie de página. Tiene sentido nombrar los títulos como `tittle` y por eso en nuestros estilos usamos la palabra `tittle` para todo lo que se refiere a títulos. Entonces, la clase CSS `tittle` las teníamos utilizando tanto en el encabezado como en el pie de página, como también en el listado.

```
1 const Encabezado = () => {
2   return (
3     <header>
4       <h1 className="title">Encabezado</h1>
5     </header>
6   );
7 }
8
```

Ademas, al ser un componente único, tiene sentido que usemos un tag `<h1>`. El problema que tendremos es que la clase `tittle` y particularmente la etiqueta `<h1>` tiene muchas definiciones. Una para el encabezado, otra para el pie de página y otra para el listado. Incluso podríamos tener otros componentes con secciones en las que también podríamos necesitar etiquetas `<h1>` y, al ser títulos, tendría sentido llamarlos `tittle`. Pero, debido a las reglas de CSS, la última definición de la clase `tittle` que el navegador cargue para las etiquetas `<h1>` será la que sobrescriba a todas.

Para resolver esto existen varias soluciones posibles que implicarían cierta redundancia de código por lo que, la mejor forma es utilizando módulo de CSS.

¿CÓMO RESCRIBIR NUESTROS COMPONENTES CON ESTA TECNOLOGÍA? Lo que haremos es crear distintos archivos de estilo. Por convención usamos el prefijo `module` antes de la extensión del archivo. Esto no es esencial, pero nos sirve para que al configurar webpacks podamos distinguir entre los estilos que serán tratados como módulos y los que no.

```
1 import styles from
2 './encabezado.module.scss';
3
4 const Encabezado = () => {
5   return (
6     <header>
7       <h1 className={styles.title}>Encabezado</h1>
8     </header>
9   );
10 }
11
```

Luego, cada componente importa su archivo de CSS bajo un nombre. Esto crea un objeto JS con el que nos referiremos a las distintas clases y selectores que hayamos creado en nuestros archivos de clase.

Por último, accedemos a los nombres de los selectores por medio del objeto `styles`.

```

1 import styles from
2 './lista.module.scss';
3
4 const Lista = (props) => {
5
6     // Código que prepara la lista
7
8     return (
9         <div className={styles.lista} role="main">
10            <h1 className={styles.title}>Películas</h1>
11            { lista }
12        </div>
13    );
14 }
15

```

Como vemos, esto va entre llaves ya que debe ser una expresión válida de JS dentro de nuestro JSX.

En la industria sucede que, a menudo, estamos sobre el tiempo de entrega y tendremos que trabajar sobre una parte del código con la que no estamos familiarizados, necesitando agregar CSS sin tiempo para probar toda la aplicación. Puede pasarnos también que no sepamos exactamente qué parte de la aplicación está usando ciertos estilos o, incluso, si todos están siendo utilizados.

Todas estas situaciones conllevan el riesgo de que estemos rompiendo la UI, por eso, soluciones como los módulos de CSS o, en el caso particular de REACT, los componentes estilizados son sumamente importantes.

Clase 8: Componentes con estado

Componentes con Estado

Comencemos por preguntarnos, ¿qué es el estado? En programación, al hablar de **ESTADO**, nos referimos a *programas que manejan sus propios datos internos* y, de alguna manera, los *protegen para que no puedan ser modificados* desde otras partes del mismo o por programas externos. Una de las formas de resolver el manejo del estado es usar clases. El concepto de clase viene del paradigma de la programación orientada a objetos.

Por ejemplo, el videojuego Counter Strike, el estado del programa en un momento dado es el conjunto de las coordenadas de cada personaje, cada contador de tiempos o municiones, cada variable que define el nivel y toda la información relevante para el juego.

Los datos del estado pueden ser guardados en alguna de las siguientes formas: variables, constantes, referencias ya sea a objetos o arrays, o expresiones de funciones también conocidas como Function Expression.

```
1 let soyUnaVariable = "Soy una variable";
2
3 const SOY_UNA_CONSTANTE = 100;
4
5 let soy_una_referencia_a_un_objeto = {};
6
7 let soy_una_referencia_a_un_array = [];
8
9 const soyUnaFunctionExpression = function() {};
10
11
```

Una de las formas para resolver el manejo de estado es usando clases. El concepto de clase viene del paradigma de la POO, el cual se destaca por introducir el concepto de clases para definir la forma de comportamiento de nuevos tipos de datos. Así como existen enteros, floats, constantes, arrays o strings, en la POO se usan clases para crear tipos nuevos.

En su estructura de código una clase tiene dos partes diferenciadas pero interrelacionadas: datos internos y funciones que operan sobre estos datos. Los datos internos definidos en la *clase son el estado* y las funciones permiten cambiar y actualizar los datos internos o Estado.

Para ilustrar que es una clase y qué son los objetos usaremos, por ejemplo, una plantilla para manejar la información de las películas que más nos gustan. La plantilla representa a la clase mientras que los objetos son representados por las fichas de cada película que crearemos con la plantilla. Las clases sirven para crear lo que se conoce como objetos y, en el caso de React, las clases son una de las dos formas que se usan para crear componentes.

Componentes Stateful

Los componentes stateful —o componentes con estado— son aquellos que están atentos a los eventos que los rodean y saben reaccionar a los mismos. En función de esos eventos, el componente podrá cambiar su contenido interno.

Estos componentes ya no serán funciones nativas de JavaScript, sino que los trabajaremos como un nuevo tipo de dato: una clase.

```
import React, { Component } from 'react';
class NombreComponente extends Component {
  render () {
    return (
      // código a renderizar ...
    );
  }
}
export default NombreComponente;
```

`import React, {Component} from 'react'`. Importamos React, aclarando que queremos traer el objeto Component.

`class NombreComponente extends Component`. Usamos la palabra reservada class para empezar a definir nuestro component.

`render () {}`. Para poder renderizar los elementos visuales del componente, usamos el método render().

`return (código a renderizar ...)`. Llamamos al método `return` y definimos los elementos que queremos renderizar.

`export default NombreComponente`. Exportamos el componente.

Solemos utilizar los componentes **stateful** cuando esperamos cierta interacción del usuario y, en base a eso, queremos que el interior de nuestro componente cambie.

Los componentes **stateful** son reactivos y por ende el DOM se actualiza cuando es necesario

State y setState

El estado de un componente es aquel que permite que el mismo pueda guardar información internamente. Al estado de un componente lo llamamos “state”. Este es un objeto literal (clave/valor) que almacenará la información que deseemos.

Por otro lado, el `setState()`² es un método que nos va a permitir actualizar el estado cuando lo estimemos necesario, logrando así que sea el mismo componente quien se encargue de administrar esta información.

En su momento, dijimos que una de las razones por las cuales React nos beneficia es la actualización por componentes del DOM. También, indicamos que los componentes con estado eran reactivos a las interacciones con el usuario y que, en base a eso, se actualizaban o no.

Vamos a ver cómo usamos `setState` para cambiar el estado de nuestros componentes, ya sea ante eventos de usuario, cambios en el servidor o cambios en los props.

Código

El **método constructor** es necesario para poder definir la estructura de un componente.

La **función super** en el constructor es necesaria en React, ya que de esa forma podemos utilizar las props que hereda de su componente padre.



Ilustración 6: Constructor

² <https://es.reactjs.org/docs/faq-state.html#what-does-setstate-do>

```

class contador extends Component{
  constructor(props){
    super(props);
    this.state = {
      valor:props.value,
    }
  }
}

```

Podemos recibir las **props** en el **constructor**. Es buena práctica utilizarlas al llamar al **super**.

Ilustración 7: Props

En todos los métodos que no sean el constructor debemos utilizar `this.setState()`.

```

class contador extends Component{
  // Aquí va el constructor
  incrementar(){
    this.setState({
      valor: this.state.valor + 1
    });
  }
  render(){
    return (
      <button
        onClick={this.incrementar}>
      </button>
    );
  }
}

```

Con el evento **onClick** vamos a estar modificando, a través del método `incrementar`, el estado de nuestro componente.

Ilustración 8: setState

Clases y Objetos

Para ilustrar qué es una clase y qué son los objetos, usaremos como ejemplo una plantilla para manejar información de las películas que más nos gustan. La plantilla representa a la clase, mientras que los objetos son representados por las fichas de cada película que crearemos con la plantilla.



Es equivalente a:



Supongamos que la información que nos interesa es: el título, el año, el director, los actores, el género y el puntaje en IMDB.com. Estos serán los campos preestablecidos que deberán ser llenados luego, al momento de crear las fichas concretas de cada película.

Con respecto a las funciones, solo queremos dos:

- Poder ir desde la ficha de cada película a la dirección de IMDB donde está la información de la película

- Poder actualizar la imagen o foto de la película en la ficha fácilmente, sin tener que alterar la ficha una vez creada a partir de la plantilla.

Nuestra plantilla tendrá el nombre Película porque cada documento creado a partir de ella representará a una sola película, no a varias. Nuestra plantilla luce así para empezar:

Película
Título
Año
Director
Actores
Género
Puntaje

La primera función la implementaremos vinculando la URL de la película en IMDB al documento. Así, al hacer clic se abrirá el navegador automáticamente, mostrándonos la sinopsis de la película con toda la información. La segunda función la implementaremos vinculando la imagen con una ruta, en lugar de pegar la imagen en la ficha. Así, con solamente reemplazar la imagen en el directorio, el documento se actualizará automáticamente para mostrar la nueva imagen.

Nuestra plantilla final luce así:

Película
<carga-automática-de-la-imagen-de-la-película>
Título
Año
Director
Actores
Género
Puntaje
<click-para-abrir-en-imdb>

Manejo del estado componentes de clase

Existen momentos claves en los que se debe manipular un estado. Uno es luego de montar y el otro es cuando se cambia alguna de sus propiedades.

Implementemos como ejemplo el armado de fichas técnicas para películas, donde las variables de estado serán: título, año, director, género, elenco principal y el puntaje. Lo que queremos es que estas variables de estado se ajusten cuando el componente se monte. Luego, si alguna de estas variables cambia, buscaremos que se reflejen esos cambios.

Como podemos ver en el código, para acceder al constructor de la clase base se usa super y se le pasan los parámetros que espera, es decir, props. Esto permite inicializar correctamente a la clase base de React de la que estamos heredando. Luego definimos el estado, la sentencia `this.state` espera un objeto literal con las variables que definirán el estado. En este contexto, todos los datos que queremos reflejar de las películas forman parte del objeto literal.

```
1  constructor(props) {
2    super(props);
3
4    this.state = { titulo:"",
5      año:0, director:"", actores:[],
6      genero:"", puntaje:0 };
7  }
8
```

Por simplicidad supongamos que estos datos se guardan un archivo json llamado “peliculas.json” dentro del directorio data que se puede importar de la siguiente manera:

```
1  import * as peliculas from
2  './data/peliculas.json';
3
```

Este archivo es una array con los datos que nos interesan. Ahora llegó el momento de asignar valores al estado. Para manipular al estado en un componente de clase React es imprescindible utilizar la función `setState` provista por la clase base de React.

Esta función la podremos aplicar dentro de otras funciones de clase que estarán en uso durante la vida del componente luego de montarse (`componentDidMount`), luego de actualizarse (`componentDidUpdate`) y antes de destruirse (`componentWillUnmount`).

Con la función `componentDidMount` utilizamos `this.setState` para actualizar los valores de estado cuando las propiedades cambian de sus valores iniciales a los valores que leeremos desde el archivo json.

```
1  componentDidMount() {
2    this.setState({...this.props});
3  }
```

Notemos como estamos utilizando el operador spread, que son los puntos suspensivos que utilizamos para expandir las props dentro de un par de llaves y que queden dentro del objeto literal. De esta manera, cada props es asignada a cada una de las variables de estado.

Por último, dentro de la función render retornamos los datos de las películas como elementos jsx. En este caso en particular, cada película tendrá dos elementos: `` y `<div>` al mismo nivel y por eso necesita etiquetas fragment.

```

1  render() {
2    return (
3      <>
4        <img src={this.state.image}
5          alt={this.state.titulo + " image"}
6          width="auto" height="200"
7        />
8        <div>
9          <p> <b>Titulo:</b> {this.state.titulo}</p>
10         <p> <b>Año:</b> {this.state.anio}</p>
11         <p> <b>Director:</b> {this.state.director}</p>
12       </div>
13       <p> <b>Actores:</b>
14       {this.state.actores.join(', ')}.</p>
15       <p> <b>Genero:</b> {this.state.genero}</p>
16     </div>
17     <p> <b>Puntaje:</b> {this.state.puntaje}</p>
18   </div>
19 )
20 );
21 );
22 }
23

```

En resumen, cuando se crea un componente de clase sucede lo siguiente:

1. React invoca al constructor de la clase base y luego al constructor de la clase del componente. Dentro del constructor se debe utilizar `this.state` para definir las variables de estado.
2. React monta al componente dentro del DOM y se ejecuta `componentDidMount`, dentro de este se utiliza `setState` si deseamos actualizar el estado. `SetState` dispara un renderizado extra, pero sucede antes de que el navegador actualice la pantalla. Así que, aunque se llame dos veces el `render`, y usuario no verá el estado intermedio.
3. React ejecuta `componentDidUpdate` cuando se actualizan las propiedades del componente. Dentro de este se debe utilizar `setState` si se desea actualizar el estado. Hay que tener cuidado de comprobar si hubo un cambio en las `props` o tendremos un bucle infinito.
4. React ejecuta `componentWillUnmount` antes de destruir el componente. Dentro de este no se debe utilizar `setState` ni tampoco `this.setState` porque este componente nunca se volverá a renderizar.

Herencia, especialización y composición

Las clases pueden derivarse de otras clases y también pueden componerse de varias clases. Seguro te preguntarás: ¿por qué queríamos衍生 classes de otras classes o componer classes con otras classes?, ¿por qué no utilizar la misma clase para todos los tipos o por qué no crear una clase nueva para cada nuevo tipo? La respuesta a estas preguntas es: **especialización y reutilización**.

Volvamos a la analogía de las plantillas para explorar este tema. Supongamos que deseamos tener fichas de películas organizadas en subgéneros. Podríamos cambiar la plantilla creada anteriormente y agregarle datos y funciones cada vez que queramos agregar otro subgénero. El problema es que estos agregados seguramente serán irrelevantes para los otros géneros y subgéneros, por lo que terminaríamos con una plantilla confusa que mezcla géneros y subgéneros y que crece constantemente con cada agregado.

Ahora, supongamos que decidimos crear una plantilla para cada subgénero. Copiaríamos la información común en cada plantilla, y agregaríamos los datos y funciones relevantes según el subgénero. El problema con esta solución es que terminaríamos duplicando información y cada vez que queramos agregar datos y funciones que son independientes del subgénero deberemos

actualizar todas las plantillas o tendremos inconsistencias. Por ejemplo, supongamos que decidimos agregar el idioma de la película y tenemos 20 tipos de plantillas. Entonces, deberemos recordar agregar ese dato a las 20 plantillas, y así para cada nuevo dato o función que sea común a todas.

La programación orientada a objetos provee soluciones para estos problemas, conocidas como: **herencia, especialización y composición**. La herencia y la especialización van de la mano, mientras que la composición es la alternativa a estas.

Así, una solución es crear una clase base con datos y funciones comunes para todas las clases. Luego, crear subclases a partir de esta y que solo agreguen los nuevos tipos de datos y funciones según necesiten. Esto es lo que se conoce como herencia y especialización.

La alternativa es crear clases independientes y, cuando necesitemos datos o funcionalidad de algunas de estas clases, simplemente utilizarlas desde dentro de la clase que la requiera. Esto es composición.

Volvamos a nuestro ejemplo con plantillas y veamos cómo resolver el tema de los subgéneros.

Herencia y especialización

Supongamos que queremos crear dos subgéneros: terror y ciencia ficción. La solución consiste en crear las dos plantillas desde la plantilla base y a estas nuevas plantillas agregarles los datos nuevos. De esa manera, nuestras plantillas finales serían creadas automáticamente y lucirían así:

Película
<carga-automática-de-la-imagen-de-la-película>
Título
Año
Director
Actores
<input type="text"/>
<input type="text"/>
<input type="text"/>
Puntaje
<click-para-abrir-en-imdb>

Película
<carga-automática-de-la-imagen-de-la-película>
Título
Año
Director
Actores
<input type="text"/>
<input type="text"/>
<input type="text"/>
Puntaje
<click-para-abrir-en-imdb>

a. Subplantilla de película de terror.

b. Subplantilla de película de ciencia-ficción.

Básicamente, eliminamos el campo género de la plantilla Película y lo pasamos a las subplantillas para que lo predefinan. Esta es una práctica muy común en la programación orientada a objetos. Además, agregamos un nuevo campo llamado subgénero que será definido en cada ficha.

Ahora, cada subplantilla puede reutilizar las dos funciones de la plantilla base, acceder a los campos de esta (Título, Año, etc.), y puede definir funciones propias que solo estarán disponibles para las fichas creadas a partir de la subplantilla. Derivar una plantilla a partir de otra es un ejemplo de herencia clásica.

Composición

Retomemos nuestro ejemplo con plantillas y supongamos que deseamos tener una lista de películas.

En lugar de crear una plantilla a partir de otra plantilla, creando una relación plantilla=>subplantilla, podríamos incluir dentro de una plantilla otras plantillas. En nuestro ejemplo, la composición nos permite crear un componente que funciona como aplicación, y este crea los componentes películas. En este caso, no hay subplantillas, no hay especificidad. Componer una plantilla a partir de otras es un ejemplo de composición.

Otro aspecto de vital importancia es que cada plantilla es responsable de presentar los campos, de darle significado a cada campo, y del acceso a cada campo.

Nuestro ejemplo de plantilla tiene todo lo que una clase tiene: estado y funciones que interactúan con esos datos. Cambiemos la palabra plantilla por clase, y ya entramos (casi sin darnos cuenta) en el mundo de la programación orientada a objetos.

Resumen

HERENCIA

La programación orientada a objetos introdujo el concepto de herencia. En React y, por lo tanto, en JavaScript, existen dos formas de herencia: la herencia clásica y la herencia basada en prototipo. Esta última ha estado desde la creación misma de JavaScript, pero recién con la incorporación de las clases en ECMAScript 2015 la herencia clásica entró a formar parte de JavaScript.

La herencia basada en clases es simplemente un mecanismo para crear nuevas clases a partir de clases existentes. En JavaScript y React, se utiliza la palabra clave extends. Por ejemplo, para crear un nuevo componente de clase en React se utiliza la clase base Component así:

```
class App extends React.Component {}
```

Ahora nuestra clase App tiene todo lo que tiene la clase Component de React.

ESPECIALIZACIÓN

La especialización es adaptar una clase para hacerla más afín con nuestras necesidades. Por ejemplo, nuestra clase App tiene lo que heredó de Component, en este sentido, es genérica aún. Pero ahora podemos agregar datos y funciones para especializarse en el trabajo que queremos realizar con la clase App.

REUTILIZACIÓN

La reutilización es simplemente poder hacer uso de lo que ya tenemos a disposición, en lugar de tener que reescribirlo o copiarlo. En nuestro caso, al heredar de la clase base de React Component nos libramos de tener que escribir esa funcionalidad nosotros, o de copiarla desde la biblioteca a nuestro código.

COMPOSICIÓN

La composición es otra forma de reutilización y puede tener distintas formas: composición de funciones, composición de objetos y composición de clases, pero la idea es la misma: escribir funcionalidad en forma separada para poder combinarlas. Con esto, se logran estructuras menos

rígidas porque las relaciones que se forman no son jerarquías del tipo “es un”, sino que son del tipo “usa un”.

Clases en React

Antes de que ECMAScript 2015 (o ES6) incluyera clases, React proveía su propia definición de clases. Sin embargo, al estar disponibles las clases en forma nativa, el equipo de desarrollo de React se encargó de que estas estuvieran soportadas desde React v0.13 (lanzada en 2015), y recomienda utilizar las clases ES6 en lugar de su propia definición de clases.

Habiendo dicho esto, es importante destacar que React no está basado en el paradigma de orientación a objetos. De hecho, el componente base de las clases es un componente de tipo funcional, es decir, es una función, no una clase.

Una clase en React sigue las mismas reglas de las clases en ECMAScript, pero tiene una estructura propia y tiene algunas reglas que conviene entender y recordar.

Veamos cuáles son:

1. La biblioteca React debe estar en Scope. Esto lo logramos empezando por importar React.
2. El nombre de la clase debe empezar con mayúscula y extender la clase base `React.Component`.
3. Se debe implementar la función de clase `render` retornando algo, aunque sea un `null`.

Esto es lo mínimo que React espera en una clase, pero así no nos sirve de mucho. La idea de escribir una clase es crear un componente que maneje su estado. De hecho, si el componente no va a manejar su propio estado, usar clases para definir el componente es prácticamente innecesario. Incluso, si el componente está pensado para manejar su propio estado, la forma preferida es usar componentes funcionales, pero por ahora concentrémonos en analizar la estructura de una clase en React.

Si nuestro componente recibirá propiedades, la manera de indicárselo a la clase es escribir un **constructor** que tome esas propiedades como argumento. Al hacer esto, es imprescindible que invoquemos al constructor de la clase base `React.Component` y le pasemos las propiedades. Esto hace que nuestra clase luzca así:

```
1 import React from "react";
2
3 class App extends React.Component {
4   render() {
5     return (null);
6   }
7 }
8
```

Como podemos ver, la clase se define con la palabra clave **class**, para indicar que la clase es una especialización de la clase `React.Component` utilizaremos la palabra clave `extends`. Dentro de la clase hay una sola función que debe implementarse obligatoriamente, esta es la función **render()** y debe tener un retorno.

La otra función importante para una clase es la **función “constructor”**, dentro de esta función es donde definiremos las variables de estado. De hecho, hay dos tipos de código que encontraremos dentro del constructor: uno es para crear el estado; el otro es para declarar las funciones que serán llamadas desde otros componentes, es decir, funciones que, si bien serán definidas dentro de la clase del componente, están pensadas para ser invocadas por otros componentes que serán creados desde nuestra clase.

Estas dos funciones se deben declarar dentro del constructor utilizando bind. JavaScript es un lenguaje muy distinto a los demás lenguajes (como C++, JAVA, PHP, etc.), por esta razón debemos tener en cuenta que el contexto al que nos referimos cuando invocamos una función sea el correcto.

Para asignar el contexto correcto, una forma es usar `Function.prototype.bind()`. Hay otro mecanismo basado en arrow functions, pero bind es muy conveniente porque **permite asignar el contexto al objeto** al cual le queremos pasar la invocación de la función y será el mismo contexto sin importar cuántas veces se renderice el componente, es decir, no se destruirá y volverá a recrearse con cada renderizado. Veamos cómo queda el constructor:

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

  this.state = {
    imagen: "",
    titulo: "",
    año: 0,
    director: "",
    actores: [],
    genero: "",
    puntaje: 0,
    imdb_URL: ""
  };

  this.seleccionarPelícula = this.seleccionarPelícula.bind(this);

  this.verEnIMDB = this.verEnIMDB.bind(this);
}
```

Gracias a bind, se garantiza que estas dos funciones podrán ser invocadas por cualquier elemento o componente al que se le pasen y ejecutadas correctamente sin que salte un error. Ahora debemos definir las funciones dentro de la clase con la funcionalidad que deseamos.

Las funciones que no están pensadas para que sean ejecutadas por otros componentes o elementos no necesitan ser vinculadas con bind. Simplemente basta con definirlas dentro de la clase sin la respectiva línea en el constructor.

Veamos ahora la función render. Por el momento estamos retornando null, pero la idea es retornar un elemento JSX con la funcionalidad que hemos escrito en la clase. En nuestro ejemplo, queremos mostrar la información de cada película y queremos cierta funcionalidad al hacer clic sobre el componente. Entonces, nuestro render podría retornar elementos HTML y usar las funciones de la siguiente manera:

```

render() {
  return (
    <li onClick={this.seleccionarPelicula}>
      <img src={this.state.image}
        alt={this.state.titulo + " image"}
        width="auto" height="200"
      />
      <div onClick={this.verEnIMDB}>
        <p> <b>Título:</b> {this.state.titulo} </p>
        <p> <b>Año:</b> {this.state.año} </p>
        <p> <b>Director:</b> {this.state.director} </p>
        <p> <b>Actores:</b> {this.state.actores.join(', ')} </p>
        <p> <b>Género:</b> {this.state.genero} </p>
        <p> <b>Puntaje:</b> {this.state.puntaje} </p>
      </div>
    </li>
  );
}

```

Existen otras funciones que forman parte de la estructura de la clase y se usan para actualizar el estado del componente. Estas funciones forman parte del llamado ciclo de vida del componente. Este ciclo lo estudiaremos en detalle más adelante, por ahora basta mencionar que forma parte de la estructura de una clase React. Las dos funciones que se utilizan para actualizar el estado del componente son: `componentDidMount` y `componentDidUpdate`.

Por último, y no menos importante, debemos exportar nuestra clase para que sea accesible como un módulo, por otros módulos de la aplicación. La exportación de la clase puede tener dos formas.

- Con nombre: `export { Pelicula };`
- Sin nombre: `export default Pelicula;`

En el primer caso, debe ser importada con los corchetes angulares y con el nombre exacto. Por ejemplo:

```
import { Pelicula } from "./Pelicula.jsx";
```

En el segundo caso, debe ser importada desde otros módulos sin los corchetes angulares y puede tener el nombre original o cualquier otro nombre. Con el nombre original sería así:

```
import Pelicula from "./Pelicula.jsx";
```

Dato de color

¿Qué es ciudadano de primera clase?

El término “ciudadano de primera clase” se usa en programación para indicar que lo que sea denominado con ese término es tratado como un tipo de dato. Por ejemplo, en la programación orientada a objetos, los objetos son ciudadanos de primera clase y se les puede dar nombres, se pueden devolver desde funciones o pueden pasarse como parámetros a funciones.

De la misma manera, en JavaScript —y en la programación funcional en general—, las funciones son ciudadanas de primera clase. Esto en la práctica lo que implica es que una función se puede almacenar con un nombre (referencia), se puede devolver desde dentro de otra función o incluso puede pasarse como parámetro a otras funciones.

React es una biblioteca de JavaScript, por lo tanto, disfruta de todo el poder que la última versión de ECMAScript aporta, ya sea en forma nativa por el navegador o gracias a los transpiladores que convierten la última versión de ECMAScript en una versión que el navegador entienda.

Clase 9: Revisión y Práctica III

Tips de la Semana III

Tips para CSS

- Con módulos de CSS, cada componente puede tener su propio archivo CSS (o SCSS). Normalmente el archivo CSS reside en el mismo directorio donde reside el componente para un acceso rápido y fácil.
- La importación CSS como módulos es posible gracias a webpack.
- La forma en que se compondrán los nombres modificados de las clases se debe hacer en la configuración de webpack.
- Cada componente importa su archivo de estilos bajo un nombre. Esto crea un objeto JavaScript con el que nos referiremos a las distintas clases y selectores que hayamos creado en nuestros archivos de clase.
- La palabra clave `className` es propia de JavaScript, no de React.

Tips para map y keys

- La función `map` devuelve un nuevo array con los resultados de una función aplicada sobre cada elemento del array original.
- La función `map` no modifica el array original.
- La función `map` no reemplaza a los bucles `for`, simplemente es una alternativa más conveniente.
- React utiliza el atributo `key` para identificar cuáles elementos han cambiado, se han agregado o se han eliminado.
- Si no usamos `key` en una lista, podríamos tener un muy bajo rendimiento de la aplicación al hacer cambios en ella, o podríamos tener resultados inesperados (bugs) dependiendo del tipo de operaciones que realicemos sobre la lista.
- Los atributos `key` pueden ser una propiedad ID del elemento de la lista, o alguna composición hecha con algunas partes del contenido del elemento. Por ejemplo, si el elemento no tiene ID, pero tiene una descripción única, podríamos usar alguna función hash sobre el texto para evitar espacios y caracteres no alfanuméricos. Pero esto solo tiene sentido si la descripción es única.
- La `key` solo tiene que ser única entre hermanos, no es necesario que sea única globalmente.
- En casos en que no tengamos nada para usar como una `key`, podemos usar el parámetro `index` del elemento dentro del array. Esto no es óptimo, pero se utiliza cuando la lista no será modificada por el usuario.

Tips para estructura de componentes de clase

- La clase más básica que podemos escribir en React solamente necesita derivarse de React.Component e implementar render con un return que podría ser de tipo null.
- Las clases de React son las mismas que las clases de JavaScript.
- En JavaScript los constructores se escriben con la primera letra en mayúsculas únicamente por convención. En React, es necesario para la compatibilidad con XML de modo que no se confunda con una etiqueta HTML.
- Antes de usar this dentro del constructor debemos haber llamado a super que es el constructor de la clase base.
- Las funciones que están pensadas para que sean ejecutadas por otros componentes o elementos necesitan ser vinculadas con bind dentro del constructor.
- Las funciones que no están pensadas para que sean ejecutadas por otros componentes o elementos no necesitan ser vinculadas con bind.
- Debemos exportar nuestra clase para que sea accesible como un módulo por otros módulos de la aplicación. Hay dos formas de exportar, una con nombre y otra sin nombre. Dependiendo de la que usemos, debemos importar usando el nombre preciso o podemos usar cualquier nombre que queramos.
- En React, y por lo tanto en JavaScript, hay dos formas de herencia: la herencia clásica y la herencia basada en prototipo.
- La herencia basada en clases es simplemente un mecanismo para crear nuevas clases a partir de clases existentes. En JavaScript y React se utiliza la palabra clave extends.

Tips para state y setState

- Para manipular el estado en un componente de clase en React se utiliza la función setState provista por la clase base de React.
- Hay momentos clave en los que se debe manipular el estado. Uno es justo después de montar el componente, otro es justo después de que cambia alguna de sus propiedades.
- Cuando React crea un componente de clase sucede lo siguiente: invoca al constructor de la clase base; enseguida invoca al constructor de la clase del componente; el componente se hace accesible en el DOM (se monta); se ejecuta componentDidMount; cada vez que las propiedades del componente se actualizan se ejecuta componentDidUpdate; y justo antes de destruirse el componente se ejecuta componentWillUnmount.
- No se debe llamar a setState en componentWillUnmount porque el componente está próximo a ser destruido y ya no se volverá a renderizar.
- El operador spread (puntos suspensivos) se usa para expandir arrays u objetos JavaScript. Al usarse para pasar props, lo que hacemos es expandir ese objeto y —al mismo tiempo— meter esos objetos dentro de un objeto literal, es decir las dos llaves. De esta manera, cada prop es asignada a cada uno de los objetos expandidos.
- Trabajar con funciones en lugar de clases es lo recomendado no solo por la comunidad de React, sino también por la comunidad de JavaScript en general.

- El término ciudadano de primera clase se usa en programación para indicar que lo que sea denominado con ese término es tratado como un tipo de dato. En JavaScript —y en la programación funcional en general— las funciones son ciudadanas de primera clase. Esto en la práctica lo que significa es que una función se puede almacenar con un nombre (referencia), se puede devolver desde adentro de otra función, o incluso puede pasarse como parámetro a otras funciones.

Clase 10: Ciclo de Vida

Ciclo de vida de un componente

¿Cuáles son los estadios o fases por los cuales atraviesa un componente durante su ciclo de vida? En sus principios de diseño, React manifiesta que el armado de aplicaciones mediante componentes es su característica clave. Esto quiere decir que los componentes significan mucho para React y tal como se menciona antes, los *componentes tienen ciclo de vida*. Es más, no solo tienen vida, sino que también pasarán por varios estadios.

Pensemos en un videojuego de acción, a comenzar el juego iniciamos la acción del personaje, tal vez disparamos, cambiamos armas, avanzamos hasta que alguien sin empatía nos líquida. En este instante nuestro personaje entra en juego nuevamente, con una nueva vida. Es decir, se crea una nueva instancia del mismo personaje. De este modo y a grandes rasgos nos hemos aproximado a un entendimiento de las tres fases del ciclo de vida de un componente. Técnicamente estos son: montaje, actualización y desmontaje.



Podemos definir que un componente nace cuando su instancia es montada en el DOM. Del mismo modo podemos entender que su instancia morirá cuando sea eliminada o desmontada del DOM. ¿Qué sucede en su fase de actualización? Los componentes de clase proporcionan una serie de métodos, funciones, que nos permiten lidiar con estas situaciones.

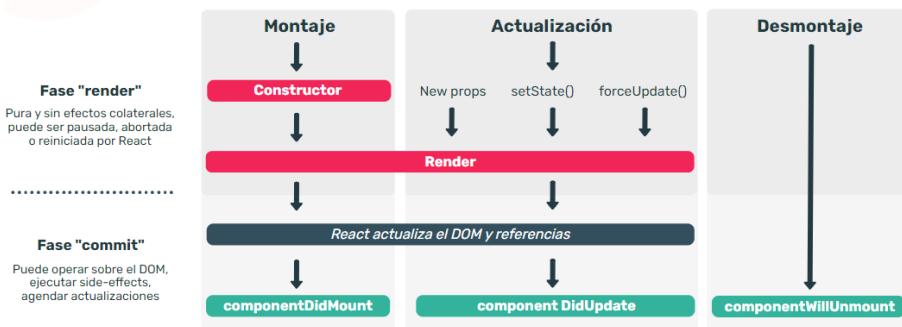
De todos los *métodos* el único *obligatorio* será *render()* siendo los demás opcionales y a utilizar según la situación. Los componentes no sólo pueden representar contenido estático, sino que también, son capaces de lidiar con datos propios o externos. Los cambios en estos datos potencialmente actualizarán lo que representan o declaran estos componentes en el DOM.

Por ejemplo, podemos recibir información una API externa que nos diga el partido que se está jugando y el puntaje del marcador. Teniendo un componente partido y otro puntaje les damos ciertas instrucciones: cuando hacen un gol el componente puntaje, ya montado, se actualiza. Y cuando termina el partido, el componente partido se desmonta y luego se montará nuevamente con información del nuevo partido y el puntaje de o.

Como podemos ver, el ciclo de vida de un componente es un proceso fundamental en la arquitectura de componentes en React. Nos permite entender cómo reaccionan dichos componentes al flujo de datos en una aplicación.

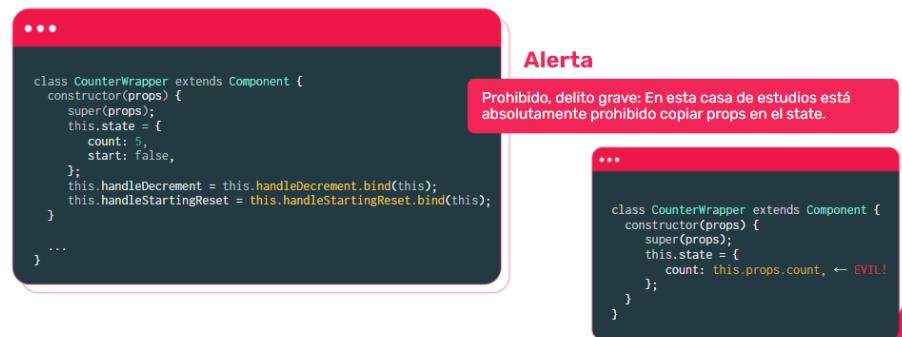
Métodos más usados

En el video se mencionó que las fases del ciclo de vida de React están asociadas a métodos específicos. La aproximación más rápida y que cubre la amplia mayoría de casos con los que nos enfrentaremos profesionalmente puede verse en este gráfico:



Constructor

El constructor es llamado antes de montarse el componente. Si bien no es obligatorio, es usado para iniciar el estado y enlazar manejadores de eventos a una instancia —en la próxima clase hablaremos de estos eventos—. Al usar el constructor en una clase que extiende `React.Component` deberá llamarse a `super(props)`, de este modo, `this.props` quedará definido en el constructor.



Render

`render()` es el único método obligatorio en un componente de clase. Render es una función pura, no interviene sobre el estado ni interactúa con el navegador, solo representará lo que reciba. Al ser llamado, examinará `this.props` y `this.state` e inmediatamente retornará alguna de las siguientes opciones:

- Elementos de React creados con JSX
- Arrays
- Fragments
- Portals (lo veremos más adelante)
- String, numbers, booleans o null

```
...  
  
render() {  
  return (  
    <main>  
      {this.state.start === true ?  
        (<Counter  
          decrement={this.handleDecrement}  
          reset={this.handleStartingReset}  
          count={this.state.count} />  
      ) : (<Start starting={this.handleStartingReset} />)  
    </main>  
  );  
}
```

componentDidMount()

Se invoca solo una vez cuando el componente se monta en el DOM —aparece en el HTML—. Este método no vuelve a ejecutarse, salvo que el componente se destruya/desmonte y vuelva a ser insertado/montado en el DOM. Es utilizado para realizar suscripciones —este tema lo veremos más adelante en el curso—. También es posible llamar a setState antes de que la pantalla se actualice.

```
...  
  
componentDidMount() {  
  console.log('El componente CounterWrapper se montó en el DOM');  
}
```

componentDidUpdate()

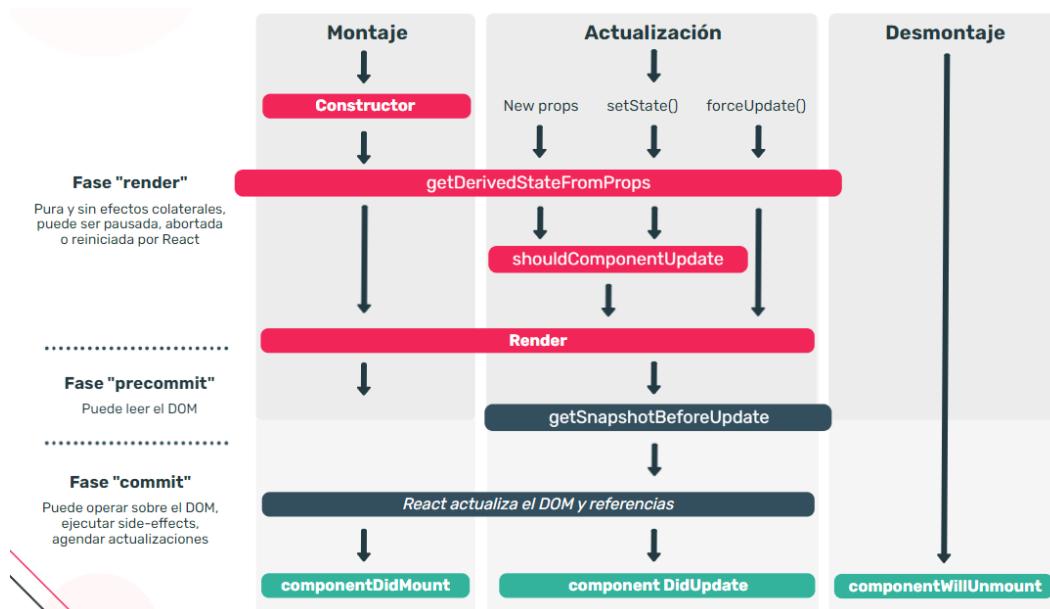
Es invocado ante cada actualización de state o props que suceda luego del renderizado inicial. Permite comparar valores previos con actuales y tomar decisiones en base a condiciones.

```
...  
  
componentDidUpdate(prevProps, prevState) {  
  console.log("El componente CounterWrapper se ha actualizado");  
  if(prevState.start !== this.state.start) {  
    if(this.state === false) {  
      console.log('El contador se ha reseteado');  
      console.log('El componente Start se montó en el DOM');  
      console.log('El componente Counter se desmontó del DOM');  
    } else {  
      console.log('El contador se ha reiniciado');  
      console.log('El componente Start se montó en el DOM');  
      console.log('El componente Counter se desmontó del DOM');  
    }  
  }  
  
  if(prevState.count !== this.state.count && this.state.count === 0) {  
    swal("You are magic!")  
      .then(() => {  
        this.handleStartingReset();  
      });  
  }  
}
```

`componentWillUnmount()`

Es el único método que se llama cuando un componente está por desmontarse. Si bien React desmonta el componente por sí sólo, puede desconocer especificidades del componente por lo cual no sabrá exactamente cómo limpiar el componente removido. Esa es la utilidad de `componentWillUnmount()`. La limpieza es útil cuando el componente ha realizado llamadas asíncronas. El problema básico a abordar es si el componente realizó una llamada a una API asíncronamente y antes de recibir la respuesta es desmontado. Si bien, no pasará nada grave, se registrará una advertencia y no se establecerá el estado. Es útil registrar esta advertencia para poder rastrear errores. Las acciones asíncronas deben ser cancelables.

Otros métodos



`getDerivedStateFromProps(props, state)`

Este método es invocado justo antes de `render()`. Devolverá un objeto en el caso de tener que actualizar el estado, o null en caso contrario. Se usa en casos raros en los cuales se le permite a un componente actualizar su estado interno como resultado de cambios en las props. Es un método estático porque se espera que se comporte como una función pura y no tenga efectos secundarios.

`shouldComponentUpdate(nextProps, nextState)`

Este al recibir un false muere ahí.

Cuando recibe un true, al ser distinto dispara el render y el component did update

Compara nuevos estados o props con los previos. Entonces, puede devolver false si no es necesario volver a renderizar el componente. Este método se utiliza para hacer que los componentes sean más eficientes.

`getSnapshotBeforeUpdate(prevProps, prevState)`

Facilita hacer operaciones sobre los elementos del DOM antes de que el componente sea realmente comprometido con el DOM. Por ejemplo, la posición exacta del scroll en un momento determinado. La diferencia entre este método y `render()` es que `getSnapshotBeforeUpdate()` no es asíncrono. Con `render()`, puede pasar que la estructura DOM cambie entre el momento en que se llama y cuándo se realizan los cambios en el DOM. Cualquier valor que se devuelva en este método de ciclo de vida se pasará como parámetro a `componentDidUpdate()`.

Eventos de Usuario

En una aplicación se puede ejecutar diferentes acciones por ejemplo hacer clic en un botón, cargar información sobre un campo, o hacer hover en una imagen, pero ¿quién las lleva a cabo? las personas usuarias.

Los eventos de usuarios son un pilar fundamental en cualquier aplicación FrontEnd, a través de ellos las personas usuarias pueden interactuar con las capas visuales y de datos de cualquier aplicación. Estos **eventos en React** se declaran con el **prefijo on** de modo muy parecido al de JavaScript Vanilla y HTML. React describe los eventos utilizando sintaxis camelCase dentro de etiquetas jsx. Recordemos que en jsx se pasan funciones dentro de llaves para manejar un evento en vez de Strings.

```
1 <button onClick={handleClick}
2   type="button">Love me</button>
3
```

Notemos la particularidad aquí con el onSubmit, estaremos usando un manejador de eventos o handler llamado handleSubmit el cual ponemos entre llaves. Más arriba en el código vemos que al declarar esta función manejadora de eventos, básicamente al ejecutarse, imprime por consola 'You clicked submit'.

```
1 function handleSubmit(e) {
2   e.preventDefault();
3   console.log('You clicked
4   submit.');
5 }
6 render {
7   return (
8     <form
9     onSubmit={handleSubmit}>
10       <button
11       type="submit">Submit</button>
12     </form>
13   );
14 }
15 }
16 }
```

¿QUÉ ES UN MANEJADOR DE EVENTOS? Es un método que ejecutará las instrucciones declaradas en su interior una vez que el evento haya sido disparado. Al disparar un evento con un manejador se le pasará a este una instancia de **SyntheticEvent**, esto nos provee un contenedor al que no le importará que navegador se esté usando por lo que permitirá que los eventos funcionen de igual modo en todos.

De esta manera vimos que un evento usuario es una respuesta que ejecuta el navegador y nuestro código ante el input o interacción del usuario con el Front que creamos. Es absolutamente esencial que la respuesta a un evento de usuario funcione y se muestre de la forma adecuada tanto para nosotros como para el usuario en sí, ya que de otra forma este último no tendrá una idea clara de lo que está pasando en la aplicación.

Implementar eventos en un componente de clase

React, junto con la continua evolución de JavaScript (ECMAScript), brindan variadas y flexibles formas de desarrollar, actualizar o debuggear una aplicación. En este caso explicaremos tres formas muy extendidas de implementar eventos, todas válidas.

Forma 1

En este caso tenemos un botón con un evento onClick que llama al manejador handleChangeColor. En esta función se setea la pieza de estado de red a blue.

```
class Eventos extends Component {
  constructor(props) {
    super(props);
    this.state = { color: "red" }
    this.handleChangeColor = this.handleChangeColor.bind(this);
  }

  handleChangeColor () {
    this.setState({ color: "blue" });
  }

  render() {
    return (
      <button onClick={this.handleChangeColor}>{this.state.color}</button>
    );
  }
}
```

Ahora bien, este armado de por sí no funcionará. En un componente de clase, el valor de this dentro de una función dependerá de cómo fue invocada la función, su contexto. En el caso de ser llamada desde un evento en JSX, this apuntará a undefined y no a la clase. Como resultado obtendremos el siguiente error: **Cannot read property 'setState' of undefined**.

Para solucionar esto debemos unir cada función handler a la clase, transformándola en métodos de la misma.

```
this.handleChangeColor = this.handleChangeColor.bind(this);
```

Forma 2

En este caso, nos ahorraremos de declarar el constructor y de unir la función con bind. Para lograr esto hacemos uso de funciones arrow, las cuales tienen la ventaja de que su scope (ámbito/alcance) es léxico.

```
class Eventos extends Component {
  state = { color: "red" }

  handleChangeColor = () => {
    this.setState({ color: "blue" });
  }

  render() {
    return (
      <button onClick={this.handleChangeColor}>{this.state.color}</button>
    );
  }
}
```

¿Qué queremos decir con esto? En una función de flecha, el valor de this es determinado por su ámbito que, en este caso, es la clase.

Forma 3

En este último caso, la sintaxis declarativa de JSX nos permite —en la misma etiqueta ()— y —desde el evento— llamar directamente a una función arrow anónima que ejecutará las instrucciones.

```
class Eventos extends Component {
  state = { color: "red" }

  render() {
    return (
      <button onClick={() => this.setState({ color: "blue" })}>
        {this.state.color}
      </button>
    );
  }
}
```

Componentes controlados vs. No Controlados

Cómo sabemos los formularios son una gran puerta de entrada para qué personas usuarias de una aplicación introduzcan datos, imágenes o archivos adjuntos. Ahora bien, hay que cuidar esos datos y tener un control de su flujo.

Cuando hablamos de *componentes controlados y no controlados* en React nos referimos a *formularios*. Los datos en un formulario se introducen a través de inputs, en este caso no hacemos referencia solo a la etiqueta, sino que cuando hablamos de inputs queremos decir entradas. Estos son los elementos que controlaremos. ¿Qué queremos decir con controlar? Es sencillo, llevar un registro de qué pasa a cada momento en un input, tanto sus valores como sus actualizaciones con React. Estamos controlando los datos asegurando desde el comienzo su integridad antes de guardarlos en una base de datos.

Ahora bien, ¿Por qué existen componentes no controlados? Para los componentes no controlados el input siempre será registrado e igualmente controlado como todo elemento insertado en el DOM por defecto.

Entonces, si los componentes son controlados por defecto desde que se montan en el DOM ¿Por qué React dice el input no está siendo controlado? Esto es así porque React propone controlar los inputs para hacer un manejo eficiente de lo que está cambiando en la aplicación. De este modo puede trabajar con datos en su estado y ¿cómo lo hace? por medio de State y Props, es decir, teniendo en cuenta el código tendremos un input controlado. React tomará ese input y lo controlará para su estado con herramientas como funciones handlers, también conocidas como manejadores, que se encargará de actualizar o implementar el estado cuando corresponda.

En definitiva, los componentes controlados son contados por React y los no controlados son controlados por el DOM.

Otros eventos: onSubmit y onChange

handleChange

Se encarga de actualizar name. Esto permite que podamos ver lo que vamos escribiendo en el <input>. Además, en este caso el manejador escribirá un mensaje en consola cada vez que el valor en el input cambie.

```
handleChange = (event) => {
  this.setState({
    name: event.target.value,
    error: false
  });
  console.log('Has cambiado el nombre');
}
```

handleSubmit

Es el manejador que controla lo que ocurre cuando el evento onSubmit es disparado. Si nosotros damos submit a una etiqueta form, por más que el mismo esté controlado por React, se impondrá el comportamiento del evento del DOM. Es por esto que hacemos un event.preventDefault(). Esto evitará que se refresque la pantalla del navegador.

```
handleSubmit = (event) => {
  event.preventDefault();
}
```

Validación

En este caso realizaremos una pequeña validación para que, al dar submit, el input tenga que tener algún carácter, de lo contrario se mostrará un mensaje de error.

```
handleSubmit = (event) => {
  event.preventDefault();

  if(this.state.name === "")
    this.setState({ error: true });

  return ({this.state.error && <span>This field is required</span>})
}
```

Finalmente, de pasar la validación, se producirá el evento

```
handleSubmit = (event) => {
  event.preventDefault();

  if(this.state.name === "")
    this.setState({ error: true });
  else {
    Swal.fire('Hello ' + this.state.name);
    this.setState({ name: '' });
  }
}
```

Alternativas en formularios y modales

Formik

Formik es una biblioteca de código abierto que nos permite trabajar en React con formularios.

Características de Formik

- Simplifica el seguimiento de valores/errores/campos visitados.
- Maneja las validaciones y submits.
- Simplifica radicalmente la configuración de estados y controladores.
- Facilita la depuración, las pruebas y el razonamiento sobre sus formularios.

SweetAlert

Es el generador de modales (rectángulos emergentes) que usamos esta semana. Es muy simple de instalar y configurar. Para instalarlo debemos ingresar por consola: `npm install --save sweetalert2`.

Glosario de la semana IV

CICLO DE VIDA: React, en sus principios de diseño, manifiesta que el armado de aplicaciones mediante componentes es su característica clave. Los componentes tienen un ciclo de vida, es decir, pasan por varios estadios. Un componente se instancia cuando es montado en el DOM (modelo de objetos del documento). Del mismo modo, podemos entender que su instancia puede eliminarse al ser desmontada del DOM.

FASES DEL CICLO DE VIDA: montaje, actualización y destrucción o desmontaje.

MÉTODOS EN EL CICLO DE VIDA EN COMPONENTES DE CLASE: los componentes de clase proporcionan una serie de métodos. Habrá métodos específicos para cada estadio del componente.

CONSTRUCTOR: este es llamado antes de montarse el componente. Si bien no es obligatorio, es usado para iniciar el estado y enlazar manejadores de eventos a una instancia (en la próxima clase hablaremos de estos eventos). Al usar el constructor en una clase que extiende React.Component deberá llamarse a super(props), de este modo, this.props quedará definido en el constructor

RENDER: es el único método obligatorio en un componente de clase. Se trata de una función pura, no interviene sobre el estado ni interactúa con el navegador, solo representará lo que reciba. Al ser llamado, examinará this.props y this.state e inmediatamente retornará alguna de las siguientes opciones:

- Elementos de React creados con JSX
- Arrays
- Fragments
- Portals (lo veremos más adelante)
- String, numbers, booleans o null

COMPONENTDIDMOUNT(): se invoca una vez que el componente se montó en el DOM (aparece en el HTML) y no vuelve a ejecutarse. Es utilizado para realizar suscripciones. También es posible llamar a setState antes de que la pantalla se actualice.

COMPONENTDIDUPDATE(): es invocado ante cada actualización de state o props que suceda luego del renderizado inicial. Permite comparar valores previos con actuales y tomar decisiones sobre la base de condiciones.

COMPONENTWILLUNMOUNT(): es el único método que se llama cuando un componente está por desmontarse. Si bien React desmonta el componente por sí solo, puede desconocer especificidades del componente, por lo cual no sabrá exactamente cómo limpiar el componente removido. Esa es la utilidad de este método.

La limpieza es útil cuando el componente ha realizado llamadas asincrónicas. El problema básico a abordar es si el componente realizó una llamada a una API asincrónicamente y antes de recibir la respuesta es desmontado. Si bien, no pasará nada grave, se registrará una advertencia y no se establecerá el estado. Es útil registrar esta advertencia para poder rastrear errores. Las acciones asincrónicas deben ser cancelables.

STATIC GETDERIVEDSTATEFROMPROPS(Props, State): este método es invocado justo antes de render(). Devolverá un objeto en el caso de tener que actualizar el estado o null en caso contrario. Se usa en casos raros en los cuales se le permite a un componente actualizar su estado interno

como resultado de cambios en las props. Es un método estático porque se espera que se comporte como una función pura y no tenga efectos secundarios.

SHOULDCOMPONENTUPDATE(NEXTPROPS, NEXTSTATE): compara nuevos estados o props con los previos. Entonces puede devolver false si no es necesario volver a renderizar el componente. Este método se utiliza para hacer que los componentes sean más eficientes.

GETSNAPSHOTBEFOREUPDATE(PREVPROPS, PREVSTATE): facilita hacer operaciones sobre los elementos del DOM antes de que el componente sea realmente comprometido con el DOM. Por ejemplo, la posición exacta del scroll en un momento determinado. La diferencia entre este método y render() es que getSnapshotBeforeUpdate() no es asíncrono. Con render() puede pasar que la estructura DOM cambie entre el momento en que se llama y cuándo se realizan los cambios en el DOM. Cualquier valor que se devuelva en este método de ciclo de vida se pasará como parámetro a componentDidUpdate().

EVENTOS EN REACT: permiten a los usuarios interactuar con la capa visual y de datos de la aplicación. ¿A qué llamamos eventos de usuario? Por ejemplo, cuando hacemos clic en un botón, modificamos un input (change) o hacemos hover sobre una imagen, entre una gran cantidad de opciones. Se declaran en las etiquetas JSX en el render del componente.

MANEJADOR DE EVENTOS: es una función o método que ejecutará las instrucciones declaradas en su interior una vez que el evento haya sido disparado.

SYNTHETICEVENT: haciendo una aproximación ligera al tema, al instanciar un manejador de eventos (al disparar un evento), se le pasará a este una instancia de SyntheticEvent. ¿Para qué sirve esto? Simplificando, podríamos decir que provee un contenedor al que no le importará en lo más mínimo qué navegador se esté usando, por lo que permitirá que los eventos funcionen de igual modo en todos los navegadores.

COMPONENTES CONTROLADOS: los componentes controlados son controlados por React y los no controlados son controlados por el DOM.

¿QUÉ QUEREMOS DECIR CON CONTROLAR?: llevar un registro de qué pasa a cada momento en un input, sus valores, sus actualizaciones. Estamos controlando los datos, asegurando desde el comienzo la integridad de los mismos antes de, por ejemplo, guardarlos en una base de datos.

FORMIK: es una biblioteca de código abierto que nos permite trabajar en React con formularios. Se encarga de simplificar el seguimiento de valores / errores / campos visitados, manejar las validaciones y submits. Simplifica radicalmente la configuración de estados y controladores y facilita la depuración, las pruebas y el razonamiento sobre sus formularios.

SWEETALERT: generador de modales.

Módulo 3: APIs y enrutamiento dinámico

APIs y enrutamiento dinámico

Efectos Secundarios

Por ahora solo hemos implementado funciones puras —o algo muy aproximado— dentro de nuestros componentes. Las funciones puras provienen del paradigma de programación funcional y se definen como aquellas que:

- Siempre producen el mismo resultado cuando se le dan los mismos argumentos.
- No producen efectos secundarios.

Estas dos características las hacen *predecibles, optimizables y reutilizables*. Cualidades que identifican también a los componentes de React.

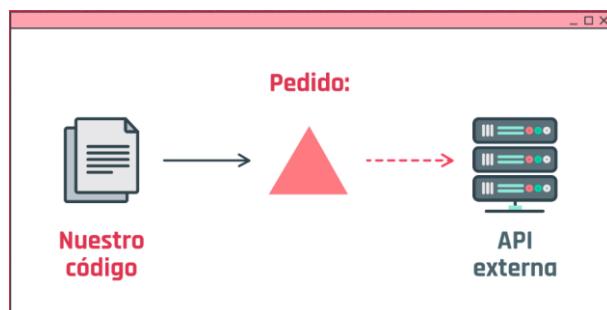
¿CÓMO LAS USA REACT? Si bien las funciones puras provienen del paradigma de programación funcional, React las aplica perfectamente en sus componentes de clase. En este ejemplo, el nombre siempre dependerá solo del valor pasado como prop:

```
class CualEsMiNombre extends React.Component {
  ...
  render() {
    return Mi nombre es { this.props.nombre };
  }
}
```

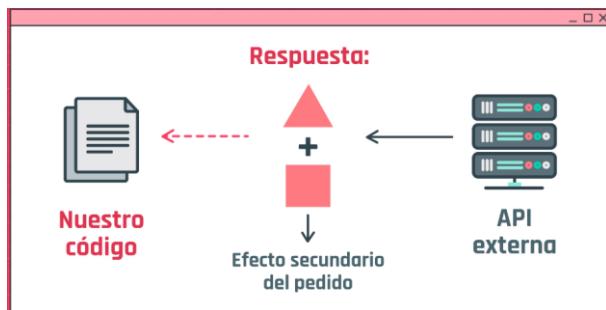
Al implementar funciones puras en componentes estos se vuelven predecibles, optimizables y reutilizables, sin embargo, esto no consigue cubrir el amplio espectro de componentes de React puesto que puede, en muchos casos, tener efectos secundarios.

¿QUÉ ES UN EFECTO SECUNDARIO? podríamos decir que se trata de una consecuencia sobre algo que iniciamos pero que no controlamos en su desarrollo. En React un efecto secundario es una operación invocada dentro de un *ámbito o scope específico* pero que transcurre *fuera del mismo* por lo que su resultado podría ser difícil de manejar afectando y haciendo más imprevisible nuestra aplicación.

Por ejemplo, cuando dentro de un método del ciclo de vida hacemos una petición o request a una API externa alojada en un servidor de un continente distinto al servidor de nuestra aplicación, en esta solicitud de red podemos ver claramente que el *resultado dependerá de sucesos externos* al método que lo invoca, a la clase que lo contiene, a la misma aplicación y hasta nuestro servidor.



Este Request será un intento de comunicación con un tercero al que no controlamos pues su comportamiento es independiente de nuestra aplicación. De esta situación se desprenden 2 elementos: por un lado, existe la posibilidad de errores en la respuesta API y a su vez, la respuesta puede demorar es decir ser asíncrona.



Los efectos secundarios, hablando técnicamente, vieron el [principio de responsabilidad única](#) que establece que una clase o un módulo debe tener solo una razón para cambiar. Esto es porque el retorno asíncrono de datos por parte de una API puede provocar cambios de estado al activar acciones adicionales y asíncronas. Ante esto REACT debe poder hacer predecibles y controlables los efectos secundarios.

¿CÓMO LO HACE? en su ciclo de vida. Luego del render inicial, los efectos secundarios pasarán por distintas fases a través de algunos métodos de montaje, actualización y desmontaje. Si bien existen otros tipos de efectos secundarios posibles, los efectos secundarios de una API suelen ser los más recurrentes.

Veamos el siguiente caso: internet se comunica principalmente por tierra y mar por medio de cables, es decir, que centenares de miles de kilómetros de cables comunican continentes a través de océanos. Sin conciencia de esto, en 2011 una mujer georgiana de 75 años en el afán de buscar cobre para venderlo como chatarra, cortó accidentalmente un cable subterráneo dejando sin servicio de internet a todo Armenia, ampliaciones de Georgia y algunas zonas de Azerbaiyán. Como resultado 3,2 millones de personas quedaron sin internet entre 5 a 12 horas.

Como podemos ver, los efectos secundarios pueden repercutir de diversas magnitudes y tener conocimiento de las diversas implicancias nos ayudarán a una programación más consistente y con menos fallos o imprevistos

Rastreando nuestras peticiones

Como ejemplo final, para acercarnos a la magnitud de la infraestructura técnica involucrada en un simple request, aquí podemos ver el recorrido de una petición para acceder a una página en China, con el comando traceroute. Cada número en la secuencia es un salto de un artefacto (IP) a otro. Partiendo desde la IP de nuestra computadora, el Gateway, nuestra conexión con el mundo a través de los cables submarinos de Las Toninas y luego, al salir de la Argentina, pasando por Miami, Dallas, Los Ángeles y Singapur. Finalmente, encontraremos un montón de ubicaciones ocultas, las cuales superan a 100 IPs protegidas con los símbolos: * * *.

```

[isabela@Isabela ~]$ traceroute -m 100 mi.com
traceroute to mi.com (161.117.97.92), 100 hops max, 60 byte packets
 1 _gateway (192.168.1.1)  0.434 ms  0.511 ms  0.793 ms
 2
 3  81.173.106.61 (81.173.106.61)  6.499 ms 213.140.39.117 (213.140.39.117)  6.614 ms 81.173.106.
61 (81.173.106.61)  6.640 ms
 4 * 213.140.39.118 (213.140.39.118)  6.642 ms *
 5  176.52.255.29 (176.52.255.29)  29.984 ms 176.52.248.79 (176.52.248.79)  30.671 ms 94.142.97.5
9 (94.142.97.59)  31.046 ms
 6  5.53.3.165 (5.53.3.165)  139.862 ms * *
 7  213.140.43.206 (213.140.43.206)  136.973 ms 94.142.119.188 (94.142.119.188)  132.280 ms 94.14
2.118.184 (94.142.118.184)  132.705 ms
 8  ix-ae-16-0.tcore2.mln-miami.as6453.net (66.110.72.30)  133.378 ms 131.422 ms 132.241 ms
 9  if-ae-56-3.tcore2.dt8-dallas.as6453.net (66.110.57.145)  363.899 ms 364.502 ms ix-ae-16-0.tco
ore2.mln-miami.as6453.net (66.110.72.30)  134.355 ms
10 if-ae-34-2.tcore1.lvw-losangeles.as6453.net (66.110.57.21)  395.986 ms 370.540 ms 370.355 m
s
11 if-ae-2-2.tcore2.lvw-losangeles.as6453.net (66.110.59.2)  366.190 ms if-ae-34-2.tcore1.lvw-lo
sangeles.as6453.net (66.110.57.21)  369.789 ms if-ae-2-2.tcore2.lvw-losangeles.as6453.net (66.110
.59.2)  364.544 ms
12 if-et-53-4.hcore2.kv8-chiba.as6453.net (64.86.252.59)  299.346 ms if-ae-28-2.tcore2.av3-toyoh
ashi.as6453.net (64.86.252.33)  364.027 ms if-ae-27-2.tcore2.av3-toyohashi.as6453.net (180.87.28.
128)  366.398 ms
13 if-ae-27-3.tcore2.av3-toyohashi.as6453.net (180.87.28.136)  364.898 ms if-ae-27-2.tcore2.av3-
toyohashi.as6453.net (180.87.28.128)  367.647 ms if-et-53-4.hcore2.kv8-chiba.as6453.net (64.86.25
2.59)  300.248 ms
14 if-ae-28-2.tcore2.svw-singapore.as6453.net (180.87.3.129)  373.351 ms if-ae-2-2.tcore2.svw-si
ngapore.as6453.net (180.87.12.2)  372.903 ms 373.934 ms
15 * * *
16 * * *
17 * * *
18 * * *
19 * * *
20 * * *
21 * * *
22 * * *
23 * * *
24 * * *
25 * * *
26 * * *
27 * * *
28 * * *
29 * * *
30 * * *
31 * * *
32 * *^C

```

Asincronía

En FrontEnd II se trató el tema de las llamadas asincrónicas AJAX. Estas, en general, permiten el acceso a datos mediante el objeto XMLHttpRequest. Esta es una interfaz empleada para realizar peticiones HTTP y HTTPS introducida al mercado por Microsoft para Internet Explorer 5, a finales del siglo XX.

Vimos que AJAX facilitaba las peticiones mediante un código funcional, rápido y eficiente, con el beneficio adicional de evadir la recarga de una página para actualizar su información. Recordemos que, dentro de esta estructura de comunicación, hablamos de:

- **request:** cada vez que el cliente le solicita un recurso al servidor.
- **response:** cada vez que el servidor le devuelve una respuesta al cliente.

También conocimos otras formas de manejar la asincronía a través de Promises con [fetch\(\)](#).

APIs

Una APPI (application programming interface) es básicamente una URL que no devuelve una página web como estamos acostumbrados a ver si no que lo que devuelves información para qué otro sistema lo consuma.

Las APIs son desarrolladas para que dos sistemas puedan comunicarse ya que si un sistema se aísla no tiene futuro. Como la información es para otro éste debe saber utilizarla es por esto que las APIs suelen tener su propia documentación.

Hay APIs que son públicas, otras privadas o también las hay semipúblicas. Las públicas son APIs para las cuales no necesitamos nada para consumirlas, tal vez necesitemos registrarnos, pero son gratuitas y podemos consultar su información. Por otro lado, tenemos las APIs semipúblicas donde

se establece un límite de información y si requiero más debo pagar. Por último, existen las APIs privadas, en este caso los endpoints no están disponibles para nuestro consumo, son solamente para productos de la organización que las crea.

¿QUÉ ES UN ENDPOINT? Es un punto de conexión donde necesitamos apuntar para obtener la información que queremos. Es decir, son las URL que debemos utilizar para obtener la información de un servidor a través de una API.

Profundizando un poco, una API (interfaz de programación de aplicaciones) es un conjunto de subrutinas, definiciones, protocolos, funciones y procedimientos, que puede ser usado por otro software como una capa de abstracción. Permite comunicar aplicaciones que no hablan el mismo idioma. Como esas aplicaciones de traducción que usaban los temerarios hinchas de fútbol que viajaron a la inhóspita Rusia en tiempos del mundial de fútbol para pedir una hamburguesa. Además, habilitan acceso a recursos sin descuidar la seguridad y el control.

APIs y React

React es compatible con cualquier biblioteca AJAX. Desde bibliotecas con soporte nativo en navegadores actuales —como Fetch— o bibliotecas third party —como Axios—. Esto permite una gran flexibilidad a la hora de desarrollar peticiones al servidor. No hay ningún gran misterio aquí. Recordemos lo aprehendido: las llamadas a APIs en un componente de clase de React se realizan durante el ciclo de vida del mismo. Estas podrán ser efectuadas luego de que el componente haya sido montado o cuando el componente se haya actualizado debido a algún cambio en sus props o state. Es decir, utilizando los métodos `componentDidMount()` y `componentDidUpdate()`. Finalmente, para limpiar los efectos de la petición si el componente es desmontado antes de obtener el response, usaremos `componentWillUnmount()`.

Fetch

Fetch es una API JavaScript nativa para navegadores (no hay que instalar nada) que permite realizar peticiones HTTP asíncronas por medio de promises. Fetch simplifica enfoques previos basados en objetos XMLHttpRequest con un código más legible y práctico.

La sintaxis de Fetch es la siguiente:

```
fetch("/noBombardeenLasToninas.api").then(function(respuesta) { ... });
```

De este modo, retorna una promesa que solo es rechazada si ocurre un fallo de red o una abuela georgiana se cruza en el camino.

```
fetch(url)
  .then((res) => res.json())
  .then((res) => console.log(res));
```

El primer método `then()` recibe un string por lo que se le aplica el método nativo `json()` para convertirlo en un objeto legible por JavaScript. En el segundo `then()` usaremos la lógica que deseemos con los datos obtenidos.

Options

La función Fetch tiene un segundo parámetro: un objeto opcional que permite configurar la petición. En este código podemos ver cómo hacer la petición con Fetch:

```

const options = {
  method: "POST",
  credentials: "include",
  headers: { "Content-Type": "application/json" },
  body: datos ? JSON.stringify(datos) : null,
}

fetch(url, options);

```

JSON es el acrónimo de JavaScript Object Notation y, como su nombre lo indica, es muy similar al objeto literal que ya conocemos. Veamos las diferencias:

Options	
method	Método HTTP. Por ejemplo: POST, GET, PUT, DELETE y HEAD.
credentials	Permite cambiar el modo en el que se realiza el request. Por ejemplo, habilitar el envío de cookies al servidor.
headers³	Son las cabeceras HTTP que permiten —tanto al cliente como al servidor— enviar información extra. Existen un gran número de opciones.
body	Cuerpo de la petición HTTP. Datos que enviamos al servidor.

Tabla 2: Tabla comparativa

Compatibilidad en navegadores



Axios

Axios es una biblioteca third party (hay que instalarla). Es un cliente HTTP que admite la API Promise y que facilita la configuración y ejecución —tanto desde el lado del cliente como del lado del servidor— a través de un cliente HTTP basado en promesas para Node.js. En el servidor utiliza el módulo HTTP de Node.js y en el navegador usa XMLHttpRequest.

Las principales ventajas de Axios son:

- Interceptar requests y responses.
- Cancelar solicitudes.
- Protección contra XSRF.

³ <https://developer.mozilla.org/es/docs/Web/HTTP/Headers>

- Transformar datos de requests y responses.
- Transformar automáticamente datos JSON.

Compatibilidad en navegadores

Latest ✓	Latest ✓	Latest ✓	Latest ✓	Latest ✓	11 ✓
87 ✅ 7 ✖	89 ✅ 7 ✖	11 ✅ 8.1 ✖	89 ✅ 10 ✖	9 ✖ 10.11 ✖	
				10 ✖ 10.12 ✖	
				11 ✖ 10.13 ✖	

Práctica

Veamos cómo usar Axios, su implementación, dos peticiones (GET Y POST) y cómo refactorizar Axios para separar su configuración de su instancia base.

Método Get

```

1 import React, { Component } from 'react';
2 import axios from 'axios';
3
4 class Beauties extends Component {
5
6   state = {
7     ornitorrincos: []
8   }
9
10  componentDidMount() {
11    axios.get(`https://get.loveliesornitorrincos.firebaseio.com`)
12      .then(res => {
13        const ornitorrincos = res.data;
14        this.setState({ ornitorrincos })
15      })
16  }
17
18  render() {
19    return (
20      <div className="App">
21        <ul>
22          {this.state.ornitorrincos.map(
23            ornitorrindo => <li>{ornitorrindo.speakVeryWell}</li>
24          )}
25        </ul>
26      </div>
27    )
28  }
29}
30
31 export default Beauties;
32

```

Método Post

```
componentDidMount(){
  axios.post('/proceres',{
    firstName:'Jose',
    lastName:'de San Martin'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
}
```

Instancia base en Axios

Las instancias base en Axios permiten refactorizar la configuración en Axios para ser usada en el momento de instanciación, ahorrando la escritura de código repetitivo.

```
1 // api.js
2 import axios from 'axios';
3 export default axios.create({
4   baseURL: `http://autosdelsiglo22.com/~`,
5   headers: { 'X-Custom-Header': 'foobar' }
6 });
7
8 // Componente
9 import React from 'react';
10 import axios from 'axios';
11 import APIFUTURISTA from '../api';
12 export default class Futuro extends React.Component {
13   handleSubmit = event => {
14     event.preventDefault();
15     APIFUTURISTA.get('voladores')
16       .then(res => {
17         console.log(res);
18         console.log(res.data);
19       })
20   }
21 }
```

Fetch vs Axios

fetch	Axios
Es una API nativa. No se instala	Es un paquete de terceros. Se instala.
Sencillo de usar.	Es aún más sencillo de usar.
No funciona en IE 11.	Funciona en IE 11.
Hay que convertir los datos recibidos a JSON.	Conversión automática.
No usa XMLHttpRequest.	Usa XMLHttpRequest.
Funciona en Node.js si se instala la dependencia node-fetch.	Funciona en Node.js

Tabla 3: Tabla Comparativa

Glosario de la semana V

FUNCIONES PURAS: provienen del paradigma de programación funcional y se definen como aquellas que siempre producen el mismo resultado cuando se les dan los mismos argumentos y no producen efectos secundarios.

EFFECTO SECUNDARIO (SIDE EFFECTS): en React, es una operación invocada dentro de un ámbito o scope específico, pero que transcurre fuera del mismo. Debido a esto, su resultado podría ser difícil de manejar, afectando y haciendo más imprevisible nuestra aplicación, porque puede generar errores en el estado global de la aplicación.

CARACTERÍSTICAS DE LOS EFECTOS SECUNDARIOS: violan el principio de responsabilidad única que establece que una clase o un módulo debe tener solo una razón para cambiar. En el caso de peticiones a APIs, el retorno asincrónico de datos puede provocar cambios de estado al activar acciones adicionales y asincrónicas. Sin embargo, debemos recordar que, si bien los efectos secundarios de peticiones a APIs suelen ser los más recurrentes, no son los únicos que existen.

MANEJO DE EFECTOS SECUNDARIOS EN REACT: nuestra aplicación React tiene que hacer predecibles y controlables los efectos secundarios a través de su ciclo de vida. Luego del render inicial, los efectos secundarios pasarán por distintas fases, y serán manejados a través de los métodos: componentDidMount, componentDidUpdate y componentWillUnmount.

TRACEROUTE: es un comando de diagnóstico de redes para mostrar las posibles rutas o caminos de los paquetes y medir las latencias de tránsito y los tiempos de ida y vuelta a través de redes de protocolo de Internet. Permite seguir la pista de los paquetes que vienen desde un punto de red.

ASINCRONÍA: JavaScript utiliza AJAX para realizar llamadas asincrónicas. Estas, en general, permiten el acceso a datos mediante el objeto XMLHttpRequest (interfaz empleada para realizar peticiones HTTP y HTTPS introducida al mercado por Microsoft para Internet Explorer 5 a finales del siglo XX). AJAX facilitaba las peticiones mediante un código funcional, rápido y eficiente, con el beneficio adicional de evadir la recarga de una página para actualizar su información.

REQUEST: cada vez que el cliente solicita un recurso al servidor.

RESPONSE: cada vez que el servidor devuelve una respuesta al cliente.

API (INTERFAZ DE PROGRAMACIÓN DE APLICACIONES): es un conjunto de subrutinas, definiciones, protocolos, funciones y procedimientos, que puede ser usado por otro software como una capa de abstracción. Permite comunicar aplicaciones que no hablan el mismo idioma. Además, habilitan acceso a recursos sin descuidar la seguridad y el control.

APIS Y REACT: React es compatible con cualquier biblioteca AJAX. Desde bibliotecas con soporte nativo en navegadores actuales —como Fetch— o bibliotecas third party —como Axios—. Eso permite una gran flexibilidad a la hora de desarrollar peticiones al servidor.

REQUEST EN REACT: las llamadas a APIs en un componente de clase de React se realizan durante el ciclo de vida del mismo. Estas podrán ser efectuadas luego de que el componente haya sido montado o cuando el componente se haya actualizado debido a algún cambio en sus props o state. Es decir, utilizando los métodos componentDidMount() y componentDidUpdate(). Finalmente, para limpiar los efectos de la petición, si el componente es desmontado antes de obtener el response, usaremos componentWillUnmount().

FETCH: es una API JavaScript nativa para navegadores (no hay que instalarla) que permite realizar peticiones HTTP asíncronas por medio de promises que simplifica enfoques previos basados en objetos XMLHttpRequest, con un código más legible y práctico.

PARÁMETRO OPTIONS EN FETCH: Fetch tiene un segundo parámetro, un objeto opcional que permite configurar la petición.

Método	Método HTTP. Por ejemplo: POST, GET, PUT, DELETE y HEAD.
Credentials	Permite cambiar el modo en el que se realiza el request. Por ejemplo, habilitar el envío de cookies al servidor.
Headers ⁴	Son las cabeceras HTTP que permiten tanto —al cliente como al servidor— enviar información extra. Existe un gran número de opciones.
Body	Cuerpo de la petición HTTP. Datos que enviamos al servidor.

AXIOS: es una biblioteca third party (hay que instalarla). Es un cliente HTTP que admite la API Promise y que facilita la configuración y ejecución —tanto desde el lado del cliente como del lado del servidor— a través de un cliente HTTP basado en promesas para Node.js. En el servidor utiliza el módulo HTTP de Node.js y en el navegador usa XMLHttpRequest.

Sus beneficios son:

- Intercepta requests y responses.
- Transforma datos de requests y responses.
- Cancela solicitudes.
- Transforma automáticamente datos JSON.
- Brinda protección contra CSRF.

INSTANCIAS BASES EN AXIOS: estas permiten refactorizar la configuración en Axios para ser usada en el momento de instancia, ahorrando la escritura de código repetitivo.

Clase 16: React Router

Enrutamiento

El enrutamiento es un mecanismo que abarca varias funciones, pero lo más importante que se logra con él es leer las URLs, interpretarlas y pasar esta información a la parte correspondiente de la aplicación que se encarga de renderizar la vista correspondiente.

URL

- Una URL es una indicación de cómo acceder a ciertos contenidos en un sitio web.
- Una vista es lo que el usuario ve.
- Las URLs pueden venir desde la barra de direcciones del navegador o desde eventos disparados desde la aplicación.
- Una URL no es en realidad una ruta, es más bien una indicación de cómo se deben armar y conseguir los recursos. Esta indicación es interpretada por el enrutador de la aplicación.

⁴ <https://developer.mozilla.org/es/docs/Web/HTTP/Headers>

Recurso

- Un recurso es cualquier cosa a la que se pueda acceder a través de la web. Por ejemplo: texto, imágenes, audio, video, scripts, correos electrónicos o páginas web.
- Con un enrutador se logra el desacople de los recursos utilizados para generar una página web y la URL que se presenta al mundo.

Enrutamiento en el FrontEnd y SPA

SPA (*Single page applications*)

Hoy en día, gracias a la potencia de los navegadores, a las bibliotecas y frameworks de front end (como React, Angular o Vue) y a tecnologías propias de JavaScript (como la API de Fetch y AJAX), podemos conectar el navegador con el back end sin necesidad de recargar la página. Esto dio paso a las single page applications (SPA).

Lo que damos por sentado

Sin embargo, esta nueva forma de hacer las cosas trajo como consecuencia la pérdida de ciertos aspectos del comportamiento natural del navegador.

- Esperamos que dentro de un sitio web cada página tenga su propia URL. Así podemos guardar bookmarks y visitar esa página en el futuro sin tener que volver a buscar en qué parte del sitio web está.
- Si algún link nos lleva fuera del sitio web o a otra página del mismo sitio, esperamos ver una indicación de que la página está cargando los contenidos.
- En documentos muy largos, esperamos que, al hacer clic en un link, el navegador nos lleve exactamente al contenido dentro de la misma página. Esto lo hacen los navegadores con los identificadores de fragmento (hash).
- Queremos poder retroceder o adelantar usando las flechas de navegación del historial.

Problemas con las SPA

Todo el contenido se muestra en una sola página, por lo que veremos una única URL para todas las páginas. Con esto perdemos la capacidad de marcar nuestras páginas favoritas, y de que las flechas del historial nos faciliten la navegación.

Si la SPA simula la navegación entre páginas manipulando el identificador de fragmentos de la URL, esto puede entrar en conflicto con la función natural del navegador de poder usar el identificador de fragmentos para llevarnos a partes de la página identificadas con hashes.

Concordancia

En general, necesitamos mantener la concordancia entre una SPA y las funciones tradicionales del navegador. Esto significa que:

- El usuario aún debe poder navegar con URLs usando la barra de direcciones.
- Hay que mantener un historial de cambios de las URLs.
- Hay que manejar los hashes (#) que llevan a partes específicas dentro de una misma página.
- Hay que manejar las transiciones entre estados, lo que antes era la recarga de la página.
- Hay que entretenér e informar al usuario mientras se actualiza el estado.

Aspectos importantes

Hay dos aspectos más de mucha importancia desde el punto de vista del negocio. Es importante que las URLs sean:

- SEO friendly
- Semánticas

Esto significa que queremos que las URLs sean fáciles de leer para el usuario y fáciles de ser indexadas correctamente por los webs crawlers o spiders

Para solventar estas necesidades, los frameworks modernos reintrodujeron el concepto de enrutador, pero ahora en el lado del front end.

Un **web crawler** es un tipo de **bot** que normalmente es **operado por los motores de búsqueda**. Su propósito es indexar el contenido de los sitios web en Internet para que esos sitios web puedan aparecer en los resultados de los motores de búsqueda.

Enrutamiento estático en React

Tradicionalmente el enrutamiento es parte de la inicialización o configuración de la aplicación, y se hace antes de mostrar algún contenido. A este tipo de enrutamiento se le llama enrutamiento estático.

En el **enrutamiento estático** se deben definir de antemano todas las rutas en una ubicación centralizada. Luego, estas rutas están disponibles en el nivel más externo de la aplicación, antes de que suceda cualquier renderizado.

En React Router versión 3, el enrutamiento se resuelve de forma estática, es decir, que las rutas se definen de antemano en una sola ubicación centralizada.

Para instalar React Router v3 debemos importar react-router-3.

NPM: npm install react-router-3

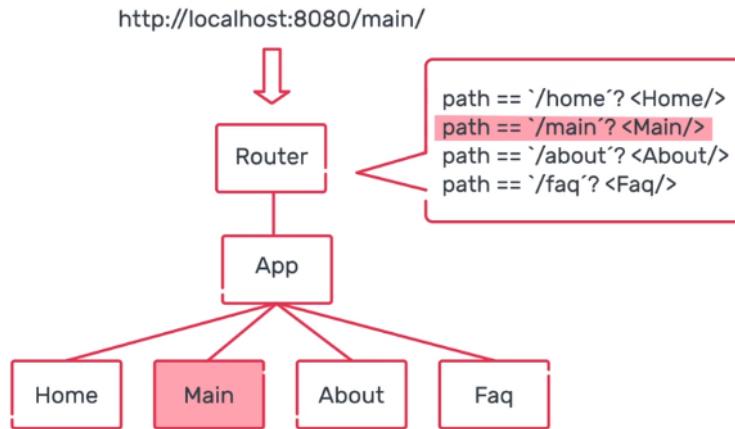
YARN: yarn add react-router-3

Anteriormente el BackEnd era quién decidía que vista cargar según su URL y enviaba la vista al navegador. Gracias a los avances que gozamos hoy podemos entablar una conversación con el BackEnd sin recargar la página y así consultar los recursos específicos y necesarios para mostrar laUrls. Con el objetivo de solventar estas necesidades los frameworks modernos reintrodujeron el concepto de enrutador o router pero ahora desde el lado del FrontEnd.

Enrutar es el proceso que sigue un navegador cuando lee, interpreta y manipula una URL ya sea que venga desde la barra de búsqueda o, desde un evento disparado en la aplicación para cargar una vista al navegador. La vista exhibe los recursos que pueden ser texto, imágenes, videos, scripts de html, correos electrónicos u otras páginas web.

Tradicionalmente el enrutamiento se hace como parte de la inicialización o configuración de la aplicación antes de mostrar algún contenido, a esto se lo conoce como enrutamiento estático. En el enrutamiento estático, se deben definir de antemano todas las rutas en una ubicación centralizada luego estás están disponibles en el nivel más externo de la aplicación antes de que suceda cualquier renderizado.

Para entender mejor cómo funciona el enrutamiento estático en React, escribiremos una aplicación muy simple con cuatro páginas para empezar (/home, /main, /about y /faq). Luego, anidaremos una página dentro de la página /main, así: /main/article. Después, veremos cómo trabajar con parámetros y pasaremos un parámetro a la página /main/article, así: /main/article/title.



Cada una de las páginas será un componente de React. Es decir que para las URLs: /home, /main, /article, /about y /faq; tendremos <Home>, <Main>, <Article> <About> y <Faq> respectivamente.

Utilizaremos React Router v3 para manejar nuestras páginas. Empezaremos por definir nuestros componentes. Serán todos componentes funcionales, no hará falta crear componentes de clase.

El primer componente que crearemos es el que contendrá todas las rutas. En este componente importaremos tres componentes desde react-router-3: Router, Route y browserHistory. Existe una alternativa a browserHistory llamada hashHistory, pero esta administra el historial de enrutamiento con el hash de la URL. Nosotros usaremos browserHistory. Veamos cómo se ve el componente App.jsx en principio:

```

import React from 'react';
import { Home, Main, Article, Faq, About } from "./index.jsx";
import { Router, Route, browserHistory } from "react-router-3";

class App extends React.Component {
  render() {
    return (
      <Router history={browserHistory}>
        {null}
      </Router>
    );
  }
}

export default App;

```

Ilustración 9: App.jsx

Ahora, veamos cómo declaramos las rutas. Dentro del par <Router></Router>, usaremos el componente Route con dos atributos llamados path y component para manejar cada ruta. Cuando el atributo path coincide con la URL entonces renderizará al componente referenciado por su atributo component.

Lo otro que haremos será meter todas las rutas dentro de la ruta raíz (/). Esto es importante para nuestro ejemplo porque queremos usar enlaces a las URLs y queremos que estos enlaces estén

disponibles en todas las páginas que se carguen. De manera que el componente que se mostrará al cargarse la ruta raíz estará siempre presente y tendrá estos enlaces.

Esto es lo que pondremos dentro del Router de App por ahora:

```
<Route path="/" component={Home}>
  <Route path="/main" component={Main}>
    <Route path="/faq" component={Faq}>
      <Route path="/about" component={About}>
    </Route>
  </Route>
```

Nuestro componente App por ahora luce así:

```
import React from 'react';
import { Home, Main, Article, Faq, About } from "./index.jsx";
import { Router, Route, browserHistory } from "react-router-3";

class App extends React.Component {
  render() {
    return (
      <Router history={browserHistory}>
        <Route path="/" component={Home}>
          <Route path="/main" component={Main}>
            <Route path="/faq" component={Faq}>
              <Route path="/about" component={About}>
            </Route>
          </Route>
        </Router>
    );
  }
}
export default App;
```

Ilustración 10: App.jsx

El componente **Home** que se renderiza en la ruta raíz tendrá los enlaces de navegación, y como además es el que contiene al resto de los componentes, es esencial que incluyamos sus componentes children: { props: children }.

Los enlaces (links) los crearemos usando un componente importado desde react-router-3 llamado Link (y su atributo to). Link es similar a la etiqueta excepto que es consciente del enrutador en el que se procesó y no dispara una recarga de página como hacen las anclas (<a>).

Creemos una lista desordenada (), ítems de listas (), y usaremos al componente Link para crear los vínculos a las páginas.

```

import React from 'react';
import { Link } from 'react-router-dom';

const Home = (props) => {
  return (
    <div>
      <ul role="nav">
        <li><Link to="/main">Main</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/faq">Faq</Link></li>
      </ul>
      <hr/>
      { props.children }
    </div>
  );
};

export default Home;

```

Ilustración 11: Home.jsx

Ahora veamos al componente Main:

```

import React from "react";

const Main = () => {
  return (
    <div>
      <h1>Main</h1>
    </div>
  );
};

export default Main;

```

Ilustración 12: Main.jsx

<Main/> por ahora será muy simple. Solo retornará un elemento de encabezado de sección h1 que dice “Main”, todo dentro de un div. Más adelante trabajaremos con este componente para que anide al componente <Article/> definido más abajo.

Ahora veamos al componente Article:

```

import React from "react";

const Article = () => {
  return (
    <div>
      <h1>Article</h1>
    </div>
  );
};

```

Ilustración 13: Article.jsx

El componente <Article /> también es sencillo. Retorna un div con un h1 adentro que dice “Article”. Este componente estará anidado dentro del componente <Main />.

```
import React from "react";

const About = () => {
  return (
    <div>
      <h1>About</h1>
    </div>
  );
};

export default About;
```

Ilustración 14: About.jsx

<About /> es también simple, en el texto dice “About”

```
import React from "react";

const Faq = () => {
  return (
    <div>
      <h1>FAQ</h1>
    </div>
  );
};

export default Faq;
```

Ilustración 15: Faq.jsx

<Faq /> es similar a todos los demás. En el texto dice “FAQ”.

Hasta acá tenemos cinco componentes que representan cinco páginas: el componente Home que se mostrará en la ruta raíz y que contiene a los otros cuatro; el componente Main que se mostrará en la página /main; el componente Article que se muestra en la página /main/article; el componente About que se mostrará en la página /about y el componente Faq que se mostrará en la página /faq:

- /home
- /main/article
- /about
- /faq

ENRUTAMIENTO USANDO PROPS. Cómo podemos ver, importamos router y browser history desde react-router-3 y usamos una función a la que llamamos routes para configurar nuestros paths y que así devuelvan componentes. El retorno de la función routes lo pasamos como una prop a Router junto con history.

```

import React from "react";
import { Router, browserHistory } from 'react-router-3';

class App extends React.Component {
  routes = () => (
    path: '/',
    indexRoute: { component: Home },
    childRoutes: [
      { path: 'home', component: Home },
      { path: 'main', indexRoute: { component: Article } },
      { path: 'about', component: About },
      { path: 'faq', component: Faq },
      { path: '*', component: NotFound },
    ]
  )
  render() {
    return (
      <div>
        <h1>App</h1>
        <Router history={browserHistory} routes={this.routes()} />
      </div>
    )
  }
}

```

ENRUTAMIENTO USANDO JERARQUÍAS. Si utilizamos jerarquías de componentes, debemos pasar los componentes al router como children de la siguiente manera:

```

import React from "react";
import { Router, Route, IndexRoute, browserHistory } from 'react-router-3';

class App extends React.Component {
  render() {
    return (
      <>
        <h1>App</h1>
        <Router history={browserHistory}>
          <Route path="/">
            <IndexRoute component={home} />
            <Route path="/main">
              <IndexRoute component={Article} />
            </Route>
            <Route path="/about" component={About} />
            <Route path="/faq" component={Faq} />
            <Route path="/" component={NotFound} status={404} />
          </Route>
        </Router>
      </>
    )
  }
}

```

Como vemos las dos formas son equivalentes: tenemos una ruta principal, podemos pasar rutas por defecto con Index Route y tenemos una while cut para capturar cualquier ruta que no hayamos definido. En este caso, ante una ruta inexistente montaremos un componente al que llamamos “Not Found”.

Así podemos hacer que cada componente se comporte como una página de manera que al cambiar la URL se muestre la vista correspondiente. Sin algo como un enrutador debemos escribir nosotros mismos la lógica para sincronizar los componentes que representan páginas con sus respectivas URLs, gestionar el histórico de las URLs y mantener el estado de la aplicación entre URLs.

Actualmente, React ha dejado de lado el enrutamiento estático por el rodamiento dinámico, un mecanismo que está más acorde con la filosofía orientada a componentes propia de React.

Páginas anidadas y parámetros

Ahora veamos cómo anidar la URL /article dentro de /main. Para esto, debemos modificar nuestro componente App para que incluya esta ruta:

```

<Route path="/main" component={Main}>
  <Route path="/main/article/" component={Article}/>
</Route>

```

Cómo podemos ver, es simplemente anidar un componente dentro de otro, al estilo React. Veamos cómo queda App con este cambio:

```
import React from 'react';
import { Home, Main, Article, Faq, About } from './index.jsx';
import { Router, Route, browserHistory } from "react-router-3";
class App extends React.Component {
  render() {
    return (
      <Router history={browserHistory}>
        <Route path="/" component={Home}>
          <Route path="/main" component={Main}>
            <Route path="/main/article/" component={Article} />
          </Route>
          <Route path="/faq" component={Faq} />
          <Route path="/about" component={About} />
        </Route>
      </Router>
    );
  }
}
export default App;
```

Ilustración 16: App.jsx

Lo otro que debemos hacer es incluir la URL en el componente Home que provee los enlaces de navegación:

```
<ul role="nav">
  <li><Link to="/main">Main</Link></li>
  <li><Link to="/about">About</Link></li>
  <li><Link to="/faq">Faq</Link></li>
  <li><Link to="/main/article">Article</Link></li>
</ul>
```

El componente Home ahora queda así:

```
import React from 'react';
import { Link } from 'react-router-3';
const Home = (props) => {
  return (
    <div>
      <ul role="nav">
        <li><Link to="/main">Main</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/faq">Faq</Link></li>
        <li><Link to="/main/article">Article</Link></li>
      </ul>
      <hr />
      {props.children}
    </div>
  );
}
export default Home;
```

Ilustración 17: Home.jsx

También debemos modificar a Main para que renderice al componente Article agregándole {props.children}

```
import React from "react";
const Main = (props) => {
  return (
    <div>
      <h1>Main</h1>
      {props.children}
    </div>
  );
}
export default Main;
```

Ilustración 18: Main.jsx

Por último, veamos cómo pasar parámetros a nuestras rutas. Supongamos que tenemos varios artículos. Podríamos tener una lista de artículos con metainformación (como nombre, ID, descripción, etc.), y podríamos leer esa información desde una base de datos, desde un archivo JSON en el back end, un archivo XML, etc. Lo importante es que supongamos que disponemos de una forma de encontrar un artículo por medio de un ID.

En nuestro ejemplo, queremos que dentro de Main se visualice Article como página. Es decir, que Article tenga su URL y además que acepte un ID como parámetro de la URL para poder conseguir el artículo.

Entonces, dentro de App, usaremos Route (encerrando al componente Article) con el atributo path apuntando a la URL anidada y le pasaremos el ID como parámetro:

```
<Route path="/main" component={Main}>
  <Route path="/main/article/:titleId" component={Article}>/>
</Route>
```

En el código de arriba, estamos pasando un identificador en el path llamado id para el artículo. Esa es la forma en que pasamos parámetros a una ruta con React Router v3.

El componente App queda finalmente así:

```
import React from 'react';
import { Home, Main, Article, Faq, About, NotFound } from './index.jsx';
import { Router, Route, browserHistory } from "react-router-3";
class App extends React.Component {
  render() {
    return (
      <Router history={browserHistory}>
        <Route path="/" component={Home}>
          <Route path="/main" component={Main}>
            <Route path="/main/article/:titleId"
              component={Article}>/>
          </Route>
          <Route path="/faq" component={Faq}>/>
          <Route path="/about" component={About}>/>
          <Route path="/" component={NotFound}>/>
        </Route>
      </Router>
    );
  }
}
export default App;
```

Ilustración 19: App.jsx

Ahora, agreguemos un enlace a la página /article dentro del componente Home que tiene la navegación:

```
<ul role="nav">
  <li><Link to="/main">Main</Link></li>
  <li><Link to="/about">About</Link></li>
  <li><Link to="/faq">Faq</Link></li>
  <li><Link to="/main/article/101">Article</Link></li>
</ul>
```

En el código de arriba, la línea que pasa el parámetro para la ruta nueva es la última:

```
<li><Link to="/main/article/101">Article</Link></li>
```

El código de Home queda finalmente así:

```

import React from 'react';
import { Link } from 'react-router-dom';

const Home = (props) => {
  return (
    <div>
      <ul role="nav">
        <li><Link to="/main">Main</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/faq">Faq</Link></li>
        <li><Link
          to="/main/article/101">Article</Link></li>
      </ul>
      <hr />
      {props.children}
    </div>
  );
}
export default Home;

```

Ilustración 20: Home.jsx

Por último, debemos obtener el parámetro en el componente Article que pasamos por la URL:

```

import React from 'react';

const Article = (props) => {
  return (
    <div>
      <hr />
      <h1>Article</h1>
      <div> {props.params.titleId} </div>
    </div>
  );
}
export default Article;

```

Ilustración 21: Article.jsx

Ahora, cuando entremos a la página /main, se renderizará el componente Main y cuando hagamos clic en el enlace /main/article/101, el componente Article se renderizará mostrando el parámetro que pasamos en la URL: el ID 101. Por supuesto, en una aplicación real este ID lo usaríamos para algo más complejo que simplemente mostrarlo.

Al hacer clic en cualquiera de los enlaces, veremos el componente respectivo y su URL en la barra de direcciones.

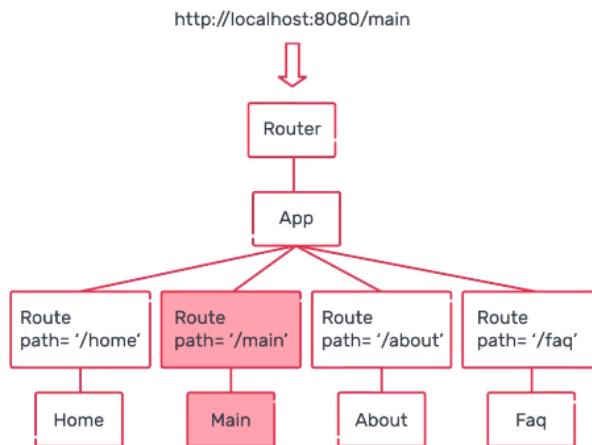
Clase 17: React Router II

Enrutamiento dinámico

Los creadores de React se sintieron frustrados y limitados por la dirección en que habían llevado a react-router. Se dieron cuenta de que estaban reimplementando partes de React como el ciclo de vida y que la API no coincidía con el modelo mental que propone React para componer la interfaz del usuario. Entonces ¿cuál es el tipo de enrutamiento moderno que utiliza React actualmente? ¿Qué ventajas tiene?

En React Router antes de la versión 4 las rutas se declaraban antes de que la aplicación escuchara las peticiones, entonces las rutas eran parte de la inicialización de la aplicación y se definían antes de que se llevaran a cabo cualquier renderizado. Por eso, en el enrutamiento estático las rutas se agrupan sin separación clara de aspectos, se ponen todas juntas en un archivo porque simplemente son rutas. En este sentido, el enrutamiento dinámico es una gran mejora.

Cuando hablamos de enrutamiento dinámico nos referimos a aquel que tiene lugar mientras la aplicación se está renderizando no en una configuración, una convención por fuera de la aplicación que está en ejecución. Como se puede ver en el enrutamiento dinámico no existe una lista de rutas que deban ser accedidas en forma centralizada, la idea es que *la rutas no queden agrupadas* por motivos de implementación técnica sino por *aspectos de negocio*.



Por ejemplo, podría ser que la regla de negocio de la aplicación requiera que solo ciertos roles de usuarios puedan acceder un tipo de URL. Para ver esto con mayor claridad construyamos una aplicación SPA minimalista.

Escribiremos nuestro componente principal que es el que contendrá a la aplicación. Será otro componente funcional, al que llamaremos `<App />`.

En las primeras líneas importamos la biblioteca React, luego nuestros componentes que funcionarán como páginas, y finalmente los componentes de React Router: BrowserRouter, Route, Switch y Link. La aplicación debe estar contenida dentro de BrowserRouter.

Veamos la estructura inicial de nuestro componente `<App />`:

```
import React from 'react';
import Home from './Home.jsx';
import Main from './Main.jsx';
import About from './About.jsx';
import Faq from './Faq.jsx';
import { BrowserRouter, Route, Switch, Link } from 'react-router-dom';

function App () {
    return (
        <BrowserRouter>
            {null}
        </BrowserRouter>
    );
}
export default App;
```

React Router requiere que la aplicación esté contenida o bien dentro del componente llamado `<BrowserRouter />` o dentro de un componente llamado `<HashRouter />`. Aunque es posible utilizar el componente `<HashRouter />` en lugar de `<BrowserRouter />`, a menos que haya alguna razón muy particular, lo mejor es trabajar con `BrowserRouter`.

- **ROWSERROUTER** (pushState, replaceState y popstate event): Usa la API del historial de HTML5. <http://localhost:8080/main>
- **HASHROUTER**(usa `window.location.hash`): Usa la parte del hash de la URL. <http://localhost:8080/#/main>

`BrowserRouter` es preferible porque crea URL ordinarias mientras que `HashRouter` creará URL con `#`. Al envolver nuestra aplicación con `BrowserRouter` creamos una instancia de API history para nuestro componente.

La API history permite administrar fácilmente el historial de sesiones, un objeto de tipo historial abstracto las diferencias en varios entornos y proporciona una API mínima que permite administrar la pila del historial, navegar y conservar el estado entre sesiones.

Ahora pasemos a escribir la lógica de la aplicación. Como mencionamos más arriba, React Router requiere que metamos la lógica de la aplicación dentro de un par `<BrowserRouter>` `</BrowserRouter>`. Además, debemos encerrar cada componente que representa a una página dentro de un componente `Route`.

Cada componente `Route` tiene un atributo llamado `path`, y cuando el atributo `path` coincide con la URL de la página entonces se renderiza el componente que está adentro.

Esto es lo que pondremos dentro de `BrowserRouter` por ahora:

```
<BrowserRouter>
  <Route exact path="/"><Home /></Route>
  <Route path="/main"><Main /></Route>
  <Route path="/about"><About /></Route>
  <Route path="/faq"><Faq /></Route>
</BrowserRouter>
```

Notemos que en el código de arriba hemos utilizado el atributo `exact`. El atributo `exact` le indica a `Route` que la URL debe coincidir exactamente con el `path`. Por ejemplo, en el código de arriba, de no utilizar `exact`, se renderizará al mismo tiempo tanto `<Home />` como el componente de la página que estemos visitando porque todos empiezan con `"/"`. Así, si la URL fuera `http://localhost:8080/about`, la aplicación renderizaría tanto `Home` como `About` al mismo tiempo.

Es importante entender que solo porque una URL coincidió con el `path` de un `Route`, no significa que los otros `Route` cuyos atributos `paths` coincidan también con la URL dejarán de renderizar a sus componentes. Recordemos esto: `Route` siempre renderizará algo, ya sea un componente, si la URL coincide con su `path`, o `null`. Debemos pensar en los componentes `Route` anidados de la misma forma en que pensamos en otros componentes anidados en React.

Veamos otro componente esencial para el manejo correcto de rutas con React Router: el componente `Switch`. Como ya sabemos, cuando una URL coincide con el `path` de algún componente `Route`, el componente encerrado por `Route` será renderizado. Sin embargo, también sabemos que si no usamos `exact`, podríamos renderizar más de un componente al mismo tiempo.

Pero ¿podemos resolver esto siempre con exact? La respuesta es que tendremos casos en los que no podremos usar exact para obtener el resultado correcto. Acá es donde el componente Switch es esencial.

SWITCH garantiza que solo el primer componente cuyo path coincida con la URL será renderizado, mientras que todos los demás cuyos paths también coincidan con la URL —si los hubiera— serán ignorados.

La forma de usar Switch es envolviendo a los componentes Route dentro de un par <Switch></Switch>:

```
<BrowserRouter>
  <Switch>
    <Route exact path="/"><Home /></Route>
    <Route path="/main"><Main /></Route>
    <Route path="/about"><About /></Route>
    <Route path="/faq"><Faq /></Route>
  </Switch>
</BrowserRouter>
```

Nuestro componente App ahora es completamente funcional, y luce así:

```
import React from 'react';
import Home from './Home.jsx';
import Main from './Main.jsx';
import About from './About.jsx';
import Faq from './Faq.jsx';
import { BrowserRouter, Switch, Route, Link } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route exact path="/"><Home /></Route>
        <Route path="/main"><Main /></Route>
        <Route path="/about"><About /></Route>
        <Route path="/faq"><Faq /></Route>
      </Switch>
    </BrowserRouter>
  );
}
export default App;
```

Si bien nuestro código está completo desde el punto de vista funcional y ya podemos navegar a través de la barra de direcciones, aún podríamos facilitar la navegación agregando vínculos (links) a las páginas que creamos.

Normalmente en una aplicación multipágina usaríamos los componentes ancla (<a>) con su atributo href apuntando a la URL, pero esto nos resulta inconveniente para una SPA porque las anclas disparan una recarga de página por defecto. En su lugar, utilizaremos el componente Link y su atributo to que React Router introdujo para estos casos.

Utilizaremos una lista desordenada (), ítems de listas (), y el componente Link de React Router para crear los vínculos a las páginas de esta manera:

```

<ul>
  <li>
    <Link to="/">Home</Link>
  </li>
  <li>
    <Link to="/main">Main</Link>
  </li>
  <li>
    <Link to="/about">About</Link>
  </li>
  <li>
    <Link to="/faq">FAQ</Link>
  </li>
</ul>

```

Nuestro código ahora luce así:

```

import React from 'react';
import Home from './Home.jsx';
import Main from './Main.jsx';
import About from './About.jsx';
import Faq from './Faq.jsx';
import { BrowserRouter, Switch, Route, Link } from 'react-router-dom';

function App() {
  return (
    <BrowserRouter>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/main">Main</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/faq">FAQ</Link>
        </li>
      </ul>
      <Switch>
        <Route exact path="/"><Home /></Route>
        <Route path="/main"><Main /></Route>
        <Route path="/about"><About /></Route>
        <Route path="/faq"><Faq /></Route>
      </Switch>
    </BrowserRouter>
  );
}

export default App;

```

Enrutamiento dinámico y páginas anidadas

La idea más importante que debemos entender es que con el enrutamiento dinámico ya no necesitamos un archivo con rutas predeterminadas. Pero esto también significa una cosa más: ¡podemos usar componentes Route y Link según lo necesitemos en cualquier componente! Por supuesto, esto funciona siempre y cuando el componente esté dentro del árbol de BrouserRouter y ya sea parte de una ruta, pero ya no es necesario que esté todo en un solo componente o archivo centralizado. Esto es lo que pone el nombre dinámico a enrutamiento dinámico. Veamos esto en la práctica.

Recordemos que queremos que dentro de Main se visualicen Blog y Vlog como páginas, es decir, cada uno con su URL. Para esto necesitamos meter componentes Route dentro de Main para que muestren a los componentes si sus paths concuerdan con la URL. Esto es exactamente lo que hemos venido haciendo, solo que ahora no lo estamos haciendo desde App, sino desde el componente Main que ya está dentro de una ruta.

Para indicarle a React Router que renderice las nuevas rutas usaremos Route (encerrando a los componentes Blog y Vlog) con los atributos paths apuntando a las URLs respectivas, tal como hemos venido haciendo hasta ahora:

```
<Route path={'/main/blog'}> <Blog /> </Route>
<Route path={'/main/vlog'}> <Vlog /> </Route>
```

Ahora, aunque no es estrictamente necesario, agreguemos los links a las páginas /blog y /vlog dentro del componente Main:

```
<ul>
  <li> <Link to={'/main/blog'}>Blog</Link> </li>
  <li> <Link to={'/main/vlog'}>Vlog</Link> </li>
</ul>
```

Nuestro código final luce así:

```
import React from "react";
import Blog from "./Blog.jsx";
import Vlog from "./Vlog.jsx";
import { Route, Link } from 'react-router-dom';
import styles from "./main.scss";
function Main() {
  return (
    <div className={styles.content}>
      <h1>Main</h1>
      <ul>
        <li> <Link to={'/main/blog'}>Blog</Link> </li>
        <li> <Link to={'/main/vlog'}>Vlog</Link> </li>
      </ul>
      <hr />
      <Route path={'/main/blog'}> <Blog /> </Route>
      <Route path={'/main/vlog'}> <Vlog /> </Route>
    </div>
  );
}
export default Main;
```

Ahora, cuando entremos a la página /main, se renderizará el componente Main que mostrará el texto Main y debajo tendremos los dos links a las páginas /blog y /vlog. Al hacer clic en cualquiera de ellos, veremos el componente respectivo y su URL en la barra de direcciones.

React Router v5

React Router es la biblioteca de enrutamiento de facto para React, y aunque no fue desarrollada por Facebook, es muy popular por su diseño y simplicidad. Ya hemos visto los conceptos esenciales sobre enrutamiento, como: URLs, aplicaciones de una sola página (SPA), enrutamiento estático, enrutamiento dinámico y rutas anidadas.

Instalemos en nuestro proyecto la versión 5 de React Router. Como estamos trabajando con navegadores web, es aconsejable instalar react-router-dom.

NPM: npm install react-router-dom

YARN: yarn add react-router-dom

Luego, en nuestro proyecto, podremos importar los componentes de React Router desde react-router-dom.

Utilizaremos los siguientes componentes provistos por React Router: BrowserRouter, Route, Switch y Link.

Creando las páginas de la aplicación

En una aplicación multipágina tradicional, cada página corresponde a un archivo que tiene la información que estamos consultando. En el caso de una SPA con React Router y enrutamiento dinámico esto no es muy distinto. Simplemente las páginas son componentes renderizados en el momento, de forma muy similar a como venimos haciendo hasta ahora.

Vamos a construir una aplicación que tendrá seis páginas. Nuestra página inicial se llamará /home; llamaremos /main a la página principal, y esta tendrá anidadas dos páginas: una llamada /blog y otra /vlog; escogeremos /about como nombre de la página sobre nosotros; y la página de preguntas frecuentes se llamará /faq.

Escribamos primero los componentes para estas páginas. Serán todos componentes funcionales, no hace falta crear componentes de clase.



```
import React from "react";
import styles from "./home.scss";

const Home = () => {
  return (
    <div className={styles.content}>
      <h1>Home</h1>
    </div>
  );
};

export default Home;
```

El componente <Home /> es muy simple, solo retorna un div con un elemento de encabezado de sección h1 que dice Home.

Ilustración 22: Home.jsx



```
import React from "react";
import styles from "./main.scss";

const Main = () => {
  return (
    <div className={styles.content}>
      <h1>Main</h1>
    </div>
  );
};

export default Main;
```

<Main /> por ahora solo retornará un elemento de encabezado de sección h1 que dice Main, todo dentro de un div. Más adelante trabajaremos con este componente para que anide a los componentes <Blog /> y <Vlog /> definidos más abajo.

Ilustración 23: Main.jsx



```
import React from "react";
import styles from "./blog.scss";

const Blog = () => {
  return (
    <div className={styles.content}>
      <h1>Blog</h1>
    </div>
  );
};

export default Blog;
```

El componente <Blog /> también es sencillo: retorna un div con un h1 dentro que dice Blog. Este componente estará anidado dentro del componente <Main />.

Ilustración 24: Blog.jsx

```

import React from "react";
import styles from "./vlog.scss";

const Vlog = () => {
  return (
    <div className={styles.content}>
      <h1>Vlog</h1>
    </div>
  );
}

export default Vlog;

```

Similar al componente <Blog />, el componente <Vlog /> retorna un div con un h1 adentro que dice Vlog. Este componente estará también anidado dentro del componente <Main />.

Ilustración 25: Vlog.jsx

```

import React from "react";
import styles from "./faq.scss";

const Faq = () => {
  return (
    <div className={styles.content}>
      <h1>FAQ</h1>
    </div>
  );
}

export default Faq;

```

<Faq /> es otro componente sencillo: solo retorna un div con un elemento h1 que dice FAQ.

Ilustración 26: Faq.jsx

```

import React from "react";
import styles from "./about.scss";

const About = () => {
  return (
    <div className={styles.content}>
      <h1>About</h1>
    </div>
  );
}

export default About;

```

Por último, <About /> es similar a todos los demás. En el texto dice About.

Ilustración 27: About.jsx

Usando React Router v5

Podemos instalar en nuestro proyecto la versión 5 de React Router con NPM. Utilizaremos los componentes provistos por React Router: BrowserRouter, Route, Switch y Link. Además, veremos cómo requerir que la ruta sea precisa con `exact path` también cómo pasar parámetros y responder entre páginas no existentes.

Empezamos importando nuestros componentes que actuarán como páginas, también importamos los componentes provistos por React Router y creamos nuestro componente de clase App. Por ahora dejamos que el render retorne null, por último, exportamos la clase App.jsx

```

import React from 'react';
import Home from './Home.jsx';
import Main from './Main.jsx';
import About from './About.jsx';
import Faq from './Faq.jsx';
import Article from './Article.jsx';
import NotFound from './NotFound.jsx';

import { BrowserRouter, Route, Switch, Link } from 'react-router-dom';

class App extends React.Component {

    render () {
        return (null);
    }
}

export default App;

```

Ahora escribimos la lógica de render, es esencial que usemos `BrowserRouter` para englobar el resto de los componentes que importamos de React Router. Utilizamos el componente `<Link />` y su atributo `to` de formar similar a las etiquetas `<a>` y su atributo `href`. La ventaja de `Link` sobre la etiqueta `<a>` es que no dispara una recarga de página. Por último, también dentro de `<BrowserRouter />`, utilizamos un componente `<Switch>` que englobe los componentes `<Route>` con sus respectivos atributos `path`. `<Switch />` nos garantiza que solamente se seleccionará el primer componente cuyo `path` coincide con la URL.

```

<Switch>
    <Route exact path="/"><Home /></Route>
    <Route path="/main"><Main /></Route>
    <Route path="/about"><About /></Route>
    <Route path="/faq"><Faq /></Route>
    <Route path="/article/:id"><Article /></Route>
    <Route path="/"><NotFound /></Route>
</Switch>

```

Analicemos el código:

- Para el caso de la ruta `/home` nos interesa que solo cuando la ruta sea exactamente la raíz se muestra el componente `home`. Por eso utilizamos el atributo `exact`, para modificar el atributo `path`.
- Para las rutas `/main`, `/about`, `/faq` no usaremos `exact` porque no hay ambigüedad.

Ahora notemos como estamos pasando un `id` al componente `article` en las rutas `/article`. Dentro de este componente podemos acceder al parámetro que pasamos por medio de la función `useParams()` de react router dom.

Esta función nos devuelve un objeto con pares clave-valor de parámetros dinámicos de la URL.

Analicemos el código del componente `Article.jsx`

```

import React from "react";
import { useParams } from
'react-router-dom';
import styles from "./article.scss";

function Article () {
    let { id } = useParams();

    return (
        <div className={styles.content}>
            <div>Article: {id} </div>
        </div>
    );
}
export default Article;

```

En este componente usamos **desestructuración** para obtener el par de clave-valor del id que en nuestro caso serían el número 1, 2 que estamos pasando con los vínculos con link to. Lo hacemos con: `let {id}=useParams()`.

Para finalizar, para capturar URLs que no coincidan con algún path de algún componente Route utilizamos una wildCard o comodín. Esto lo hacemos en el componente App, donde declaramos los componentes Route. Instanciamos un componente NotFound para indicar a la persona usuaria que la vista que está pidiendo no está en la aplicación.

En general las personas usuarias esperan que haya una concordancia entre la URL y la vista cargada en el navegador. Además, las personas esperan tener disponible las funciones básicas del navegador debemos poder navegar con URLs en la barra de direcciones, que cada vista tiene una URL que especifica de manera única esa vista, que se pueden usar marcadores (Bookmarks), que se mantenga el historial de cambios en las URLs y se pueden usar los identificadores de fragmentos o hash que llevan a partes específicas dentro de una misma página.

Las transiciones de estado no implican recarga de páginas, pero, en el caso de que tomen algo de tiempo, deben ser entretenidas. Es decir, hay que entretenere a las personas usuarias mientras se actualice el estado.

Si bien React es muy popular por su diseño y simplicidad, es sólo una biblioteca para manejar la UI. Al no ser un framework a menudo necesitaremos otras soluciones distintos aspectos del sistema.

El enrutamiento Dinámico es la forma estándar de manejar las rutas en React porque nos ayude a cumplir con todos los requisitos de forma muy simple y robusta.

Diferencias entre enrutamiento estático y dinámico

ENRUTAMIENTO ESTÁTICO: El router decide cuál vista se debe renderizar en forma centralizada antes de que App instancie al componente. En este caso la ruta /main debe instanciar la página Main.

ENRUTAMIENTO DINÁMICO: El Router decide cuál página se debe renderizar después de que App se haya renderizado. Nuevamente la ruta /main debe instanciar la página Main.

	Enrutamiento Estático	Enrutamiento Dinámico
Versión	Hasta la versión 3.	Desde la versión 4 en adelante.
Componentes de React Router.	Router, Route, IndexRoute.	BrowserRouter, HashRouter, Switch, Route, Link.
Rutas responsivas.	No	Si. Lo que se renderiza puede decidirse según las dimensiones de la pantalla.
Rutas semánticas y SEO-friendly.	Si	Si
Permite la separación clara de aspectos.	No, las rutas se ponen todas juntas en un archivo simplemente porque son rutas.	Sí, las rutas quedan agrupadas por aspectos del negocio, no por motivos de implementación técnica

Tabla 4: Diferencias Clave

Clase 18: Revisión y práctica VI

Tips de la semana VI

- Gracias a la potencia de los navegadores, a las bibliotecas y frameworks de FrontEnd (como React, Angular o Vue) y a tecnologías propias de JavaScript (como la API de Fetch y AJAX) podemos conectar el navegador con el BackEnd sin necesidad de recargar la página. Esto dio paso a las single page applications (SPAs).
- Los frameworks modernos reintrodujeron el concepto de enrutador, pero ahora en el lado del FrontEnd.
- React Router utilizó el concepto tradicional de enrutamiento estático, pero a partir de la versión 5 introdujo el concepto de enrutamiento dinámico.
- Es muy importante entender que una URL no es en realidad una ruta. Es más bien una indicación de cómo se deben armar y conseguir los recursos. Esta indicación es interpretada por el enrutador de la aplicación.

Enrutamiento estático

En el enrutamiento estático se deben definir de antemano todas las rutas en una ubicación centralizada. Luego, estas rutas están disponibles en el nivel más externo de la aplicación, antes de que suceda cualquier renderizado.

En el enrutamiento estático las rutas se definen juntas en un solo archivo, por ejemplo:

```
<Router history={browserHistory}>
  ...
  <Route path="/" component={Home}>
    <Route path="/main" component={Main}>
      <Route path="/main/article/" component={Article} />
    </Route>
    <Route path="/faq" component={Faq} />
    <Route path="/about" component={About} />
  </Route>
</Router>
```

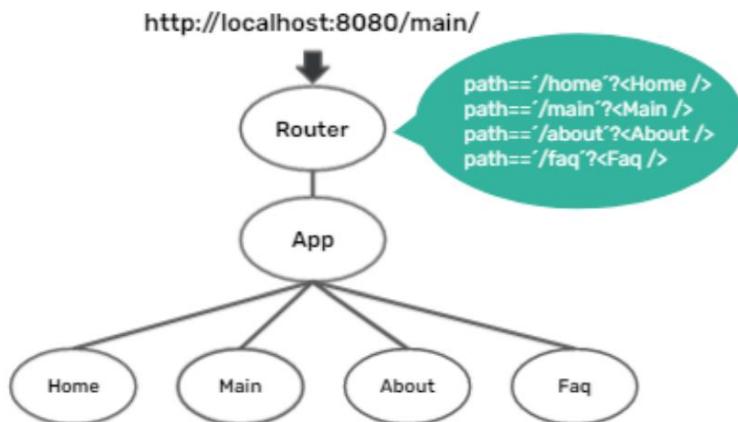


Ilustración 28: Flujo de una aplicación que utiliza enrutamiento estático

Enrutamiento dinámico

Con el enrutamiento dinámico ya no necesitamos un archivo con rutas predeterminadas. Además, podemos usar componentes Route y Link según lo necesitemos en cualquier componente.

Con este enrutamiento, las rutas se pueden definir en el componente donde tenga sentido hacerlo. Por ejemplo:

```
<BrowserRouter>
  <Switch>
    <Route exact path="/"><Home /></Route>
    <Route path="/main"><Main /></Route>
    <Route path="/about"><About /></Route>
    <Route path="/faq"><Faq /></Route>
  </Switch>
</BrowserRouter>
```

Y en otro componente, por ejemplo, en el componente Main, también podemos definir rutas.

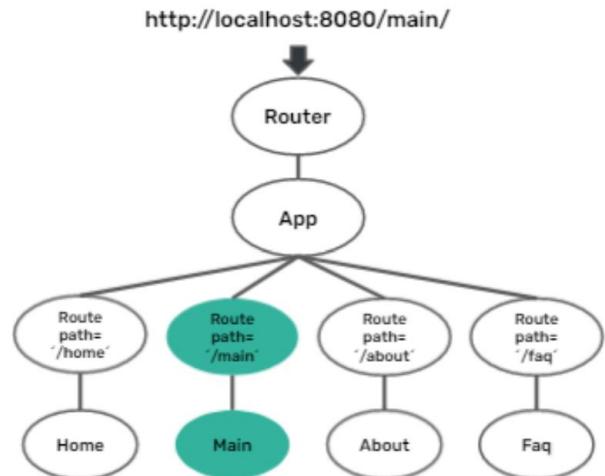


Ilustración 29: Flujo de una aplicación que utiliza enrutamiento dinámico

	Enrutamiento Estático	Enrutamiento Dinámico
Versión	Hasta la versión 3.	Desde la versión 4 en adelante.
Componentes de React Router.	Router, Route, IndexRoute.	BrowserRouter, HashRouter, Switch, Route, Link.

Rutas responsivas.	No	Si. Lo que se renderiza puede decidirse según las dimensiones de la pantalla.
Rutas semánticas y SEO-friendly.	Sí	Sí
Permite la separación clara de aspectos.	No, las rutas se ponen todas juntas en un archivo simplemente porque son rutas.	Sí, las rutas quedan agrupadas por aspectos del negocio, no por motivos de implementación técnica

Tabla 5: Diferencias entre enrutamiento estático y dinámico

Clase 19: Hooks

Módulo 4: Introducción a Hooks

Introducción a Hooks

¿Qué son los Hooks?

“Un Hook es una función especial que permite ‘conectarse’ a características de React”.

Los *Hooks son básicamente funciones*. Permiten a componentes funcionales incorporar características de React, previamente reservadas solo a componentes de clase. Entre otras utilidades, dotan a los componentes funcionales de estado interno y ciclo de vida. Además —y no menos relevante— resuelven problemas que React ha enfrentado por años.

Dato adicional: un componente de orden superior —conocido como HOC (higher-order component)— es una técnica avanzada en React para el reuso de la lógica de componentes. Es un patrón de React. Una función que recibe un componente y devuelve un nuevo componente.

¿Por qué Hooks?

Como sabemos, las tecnologías de Internet se han desarrollado exponencialmente en los últimos años a través del éxito irrefrenable de las FAANG⁵. Por ejemplo, el caso de Facebook que pasó de ser —en su etapa germinal (2003)— un sitio llamado FaceMatch (creado por Mark Zuckerberg para el ámbito de la Universidad de Harvard), en el que se calificaba rostros universitarios; hasta llegar a hoy —tras casi 20 años— a ser una de las más grandes corporaciones de alcance quasi global: propietaria de Instagram, WhatsApp, Oculus, creadora de Internet.org, entre otras miles de iniciativas. El avance continuo de estas corporaciones se da en el marco de una competencia permanente por crear una Internet a espejo de sus necesidades y definir las formas de comunicación más usadas en occidente.

Este es el contexto en el que surge React, como así también Angular (Google). El de la búsqueda de controlar el rumbo de Internet y, con ello, debido a la necesidad implícita de sofisticación y optimización permanente de las tecnologías de Internet, la necesidad de mejora continua en términos de estos frameworks/bibliotecas.

⁵ FAANG se refiere a las cinco empresas de tecnología estadounidenses prominentes: Facebook, Amazon, Apple, Netflix y Alphabet (GOOG)

¿Pero cómo se llegó a Hooks?

La liberación de React de la mano de Facebook, allá por el año 2013, fue muy exitosa. Su enfoque basado en una arquitectura de datos fluyendo hacia abajo, composición en componentes y una superrápida actualización del DOM de forma declarativa —gracias al uso del Virtual Dom— se viralizó en el mundo front end. Fue una estrategia exitosa. Lo sigue siendo hoy.

Problemas

A pesar de esto, la historia de React no estuvo exenta de problemas y adecuaciones. Principalmente respecto a la implementación de sus componentes, su pieza maestra. Cuando React fue lanzado a la comunidad, JavaScript no contaba aún con un sistema de clases, así que React optó por crear la API `React.createClass` para la creación de componentes. Fue un camino muy eficiente.

No obstante, la felicidad duró poco. Con el lanzamiento de la versión 6 de ECMAScript, en el año 2015, JavaScript introdujo oficialmente la palabra reservada `Class` y, ante la disyuntiva de seguir su propio camino o adecuarse a ECMAScript para no perder seguidores, React optó por adoptar las clases de JavaScript.

Sin embargo, esto introdujo nuevos problemas generados por las particularidades del funcionamiento de las clases en JavaScript. Este nuevo camino exigió la necesidad de crear subclases extendidas de `React.Component`, declarar el estado dentro de un constructor, usar `this`, `bind` y `super(props)`. Podríamos igualmente convenir que, más allá de ser un poco molesto, usar `bind` y `super(props)` no eran en sí problemas. Igualmente, a nadie le gusta programar años de su vida con cierta molestia.

Al detectar el fastidio en la comunidad front end, React trabajó en encontrar soluciones y presentó la sintaxis Campos de Clase. Con esta nueva característica, accedimos a la posibilidad de declarar el estado inicial de un componente sin declarar el constructor y, además, evitamos `bind` mediante la implementación de arrow functions en los handlers.

```
class Mejorando extends React.Component {
  state = { goal: '' };
  handleGoalChange = (goal) => this.setState({ goal });
  render() {
    const { goal } = this.state;
    return <Goal onGoalChange={this.handleGoalChange} goal={goal} />
  }
}
export default Mejorando;
```

En el mes de octubre del año 2018 —en la React Conf, celebrada en Henderson, Nevada, EEUU—, Sophie Alpert y Dan Abramov, miembros del equipo de Facebook encargado de desarrollar React (por ese entonces), presentaron una nueva característica revolucionaria para el mundo front end: React Hooks. Más tarde, en febrero de 2019, esta propuesta se hizo realidad y React liberó su versión 16.8 incluyendo la API de Hooks.

Problemas en los componentes de clase

Gracias a React la división de una aplicación compleja en componentes permite dosificar a nuestro gusto su complejidad. Sin embargo, es en esta instancia en dónde surge el verdadero problema de la implementación de las clases en React y este es la división del código respecto al ciclo de vida.

Volviendo a su definición, React se ve a sí mismo como una biblioteca de JavaScript para construir interfaces de usuario, pero cuando hablamos de composición no solo nos referimos a piezas de la interfaz de usuario. En ciertas ocasiones también nos encontramos frente a la lógica no visual como expresa la siguiente función en la cual la vista es función del Estado: `view=fn(state)`.

Históricamente React ha tenido problemas con su refactorización, como ejemplo podemos pensar en el problema de crear un nuevo componente de clase que comparte el ciclo de vida con un componente existente. Para afrontar esto React, en principio, se enfocó en ofrecer soluciones cómo utilizar componentes de orden superior conocidos como HOC (higher order componente) o Render Props.

Sin embargo, a mayor crecimiento de la complejidad de la aplicación, estos enfoques se tomaron muy incómodos. La aplicación se convierte en un conjunto de componentes con más envolturas que una mini mamushka, es decir, React no tenía una buena primitiva para compartir lógica no visual. En resumen, nos encontramos frente a tres problemas con los componentes de clase de React:

- La lógica de estado no es fácil de reutilizar
- Los componentes complejos son difíciles de comprender
- La implementación de clases puede ser confusa y molesta.

La lógica de estado no es fácil de reutilizar porque React no dispone de una forma directa de compartir lógica de estado entre sus componentes y las soluciones más utilizadas para afrontar esto, RenderProps y HOCs, requieren de un nivel de restructuración y abstracción de los componentes que dificultan su implementación y su seguimiento.

Entonces ¿podemos decir que los componentes complejos son difíciles de comprender? En cualquier aplicación que desarrollamos el tiempo, las necesidades del producto y de los clientes demandarán modificaciones como resultado los componentes seguirán complejizando en términos de estado, ciclo de vida y efectos secundarios. Los métodos del ciclo de vida comenzaran a mezclar código no relacionado incorrectamente lo que favorecerá a qué emergan errores e inconsistencias en la aplicación. La lógica de estado será confusa y dispersa complejizando la refactorización de los componentes en componentes más pequeños. Por ende, el testing de esos componentes se tornaron experiencia tortuosa a niveles cósmicos.

Mientras tanto las clases de JavaScript incluidas en React tienen ciertos inconvenientes, por ejemplo, this, constructor, superProps y su necesidad de unir mediante bean a los handlers. Además, los componentes de clase pueden fomentar patrones involuntarios que hacen que las optimizaciones sean más lentas. Si, las clases son confusas. En síntesis, estos problemas estructurales a nivel técnico forzaron a React a pensar en alternativas que finalmente derivaron en Hooks.

Hooks

Los Hooks son un modo simplificado de insertar características de React como: estado, ciclo de vida, contexto y referencias en componentes funcionales sin alterar el funcionamiento y las bases de React, como son la importancia de los componentes, el flujo de datos y las props. Además, permiten:

- Una mejor y más simple reutilización, composición y testeo del código.
- Extraer lógica de un componente para ser reutilizada y compartida.

Beneficios

Los beneficios de los Hooks son los siguientes:

- Menor cantidad de código.

- Código más organizado
- Utilización de funciones reutilizables.
- Mayor facilidad para testear.
- No llama a super() en un constructor de clases.
- No trabaja con this y bind en el uso de manejadores.
- Simplifica la vida y el ciclo de vida.
- El estado local está dentro del alcance de los handlers y las funciones de efectos secundarios.
- Componentes más reducidos que facilitan el trabajo de React.

Reglas de Hooks

1. No llamar Hooks dentro de condicionales, ciclos o funciones anidadas
2. Llamar a los Hooks solo dentro de los componentes de React
3. Llamar a los Hooks en el principio de la función del componente.

Convivencia entre componentes de Clase y Hooks

Si bien React hace énfasis en las ventajas de Hooks sobre Class, no eligió una estrategia radical de migración a partir de su versión 16.8. Esto se explica comprendiendo que antes de Hooks (y con el lento, pero continuo declive de jQuery), React era el líder indiscutido dentro del universo de las bibliotecas/frameworks front-end. Es decir, había muchísimo código escrito en clases enviado a producción y estable o, dicho de otro modo, mucho capital invertido en clases de React. Por lo tanto, no era conveniente generar un caos en la comunidad y en las empresas que pudiera hacer perder porción de mercado a React.

La decisión, por el contrario, fue una convivencia entre ambos tipos de programación. Es decir, que en una misma aplicación pudieran convivir componentes de ambos tipos, para dar tiempo a un cambio de mentalidad gradual. Como ejemplo de esta decisión, Facebook conserva una enorme cantidad de código escrito en clases. Finalmente, debemos aclarar que no existe la libertad absoluta entre ambos tipos de componentes. No podemos usar Hooks dentro de un componente de clase.

Hooks nativos en React

UseDebugValue: Permite aplicar una etiqueta a los hooks personalizados visible mediante React DevTools.

- **USESTATE**: Declara variables de estado y las actualiza.
- **USEEFFECT**: Permite incluir y manejar efectos secundarios.
- **USECONTEXT**: Permite crear un proveedor de datos globales para que puedan estos ser consumidos por los componentes envueltos por el proveedor.
- **USEREDUCER**: Es una variación de useState, resulta conveniente utilizarlo cuando nos encontramos ante un gran número de piezas de estado.
- **USEIMPERATIVEHANDLE**: Personaliza el valor de instancia que se expone a los componentes padres cuando se usa ref.
- **USEMEMO**: Devuelve un valor memorizado. También es útil para evitar renderizados innecesarios. Optimiza el rendimiento.

- **USERREF**: Permite utilizar programación imperativa. Devuelve un objeto mutable cuyos cambios de valor no volverán a renderizar el componente. Por el contrario, se mantendrá persistente durante la vida completa del componente.
- **USECALLBACK**: Devuelve un callback que es memorizado para evitar renderizados innecesarios.
- **USELAYOUTEFFECT**: Es similar a useEffect, pero se dispara de forma síncrona después de todas las mutaciones de DOM.

¿Qué es useState?

En React, la interfaz de usuario representa el estado en el momento de renderización inicial. Si alguna pieza de ese estado actualiza su valor, React volverá a renderizar los componentes involucrados con dicha pieza de estado tras hacer una comparación entre el virtual DOM y el DOM guardado en memoria. El objetivo de esto es que la actualización sea eficiente y basada en la diferencia entre ambos DOM.

Desde el lanzamiento de Hooks, los componentes funcionales pueden declarar y actualizar su estado interno mediante el hook useState. Este es una función que crea una variable que permite almacenar una pieza de estado interno y, a la vez, actualizar dicho valor en un componente funcional.

```
const [job, setJob] = useState("Seeking");
```

Cada componente tiene asociado una lista interna de “celdas de memoria”. Estas celdas de memoria se traducen en objetos literales de JavaScript en donde incluimos nuestros datos. Cuando invocamos useState, la función va leyendo las celdas una a una. De este modo, múltiples llamados a useState obtienen múltiples estados locales independientes.

En teoría, esto nos permitiría declarar infinitas const de useState.

```
const [count, setCount] = useState(0);
const [phone, setPhone] = useState("");
const [username, setUsername] = useState("");
const [email, setEmail] = useState("");
const [password, setPassword] = useState("");
const [confirmPassword, setConfirmPassword] = useState("");
const [otaku, setOtaku] = useState(false);
const [loveLevel, setLoveLevel] = useState({day: "orange", night: "blue"});
```

Podemos incluir useState las veces que necesitemos, la única condición es que sea llamada desde un nivel superior de código, no en un bloque.

```
//Esto está mal
const Counter = () => {
  const estoEstaMal = () => {
    const [count, setCount] = useState(0);
  }
  return {estoEstaMal};
};

//Esto está bien
const Counter = () => {
  const [count, setCount] = useState(0);
  return {count};
};
```

Estructura de useState

Un array desestructurado es una manera de extraer elementos de arrays o propiedades de objetos en distintas variables. En nuestro ejemplo, count y setCount son elementos de useState declarándolos así obtenemos acceso directo a los mismos sin tener que volver a invocar a useState.

```
const [count, setCount] = useState(24)
```

Aquí podemos ver un ejemplo con un conjunto de frutas desestructurándose en frutas específicas que podemos comer directamente.

```
const [🍇,🍊,🍉,🍊,🍋,🍌] = FRUTERÍA
```

La fórmula anterior es equivalente a esto:

```
const arrayCount = React.useState(24)
const count = arrayCount[0]
const setCount = arrayCount[1]

const count = React.useState(24) [0]
const setCount = React.useState(24) [1]
```

A simple vista podemos entender que está implementación simplifica el código y lo hace más entendible, useState recibe un argumento que devuelve una **TUPLA** con dos elementos.

Una tupla es una array en el cual sus elementos respetan un orden específico entonces el primer elemento será el valor de la variable que figura como count y el segundo es la modificación para modificarla, setCount.

Cuando React renderiza, o bien, re renderiza nuestro componente este solicita a React el último valor de estado, en este caso de count, con el fin de dibujar la interfaz de usuario y también la función para actualizar el valor de estado. Si se ejecuta la función para modificar el valor, React volverá a renderizar la interfaz. Entonces React comparará la interfaz de usuario generada con la versión almacenada y actualizará la pantalla del modo más eficiente. De este modo, la función de actualización ha actualizado el valor de count.

Actualización

Es importante remarcar que cuando llamamos a la función set de un useState, se sobrescribe el contenido de la variable.

Diferencias en la implementación respecto a clases

En los componentes de clase, el estado es un objeto literal, es decir, todo el estado va en ese objeto. Con useState, si bien puede usarse el mismo patrón, tenemos la posibilidad de implementar diversas llamadas a useState para dividir nuestro estado en piezas.

```
const [state, setState] = useState({ radio: 0, perimetro: 0, superficie: 0 });
```

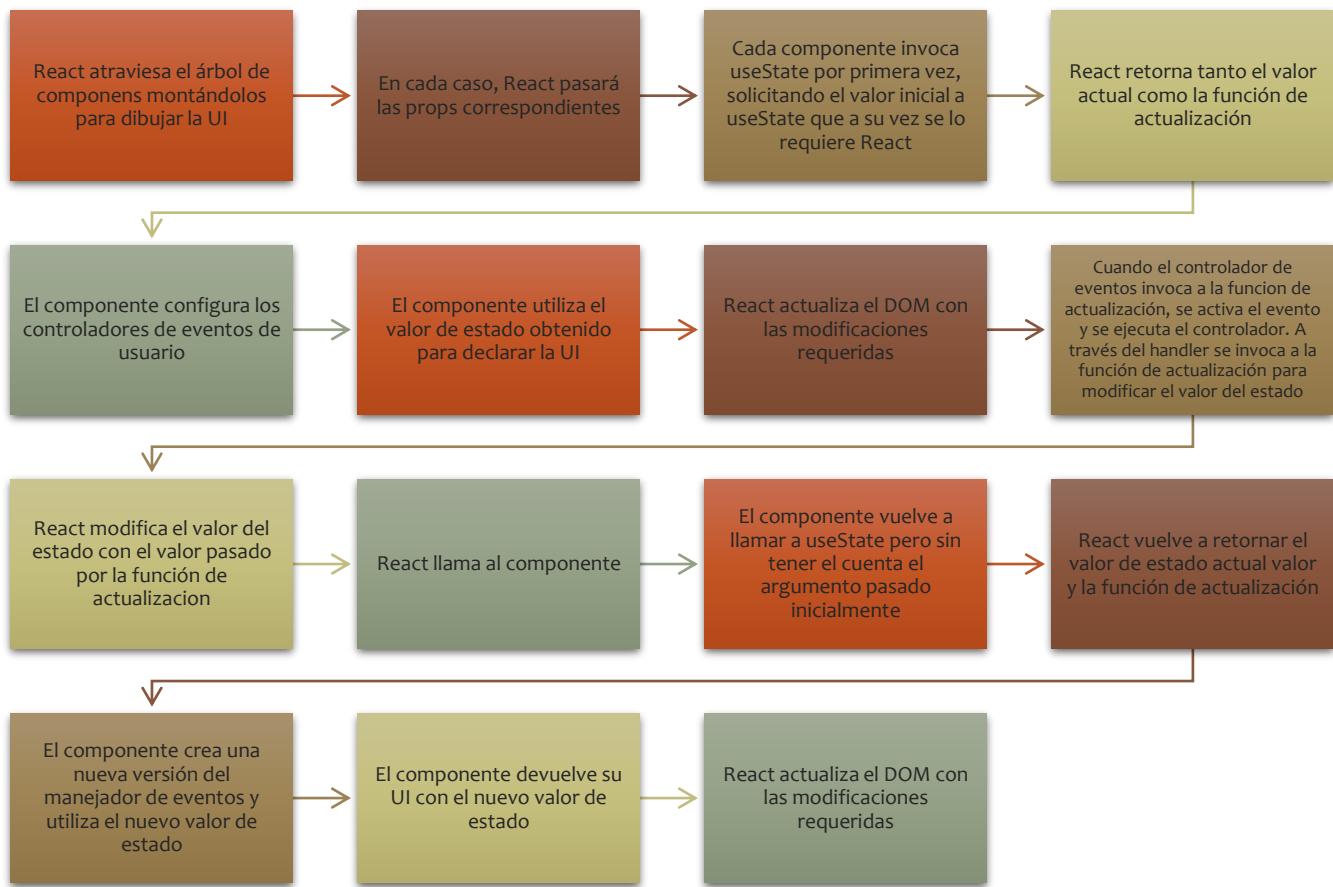
Ilustración 30: Ejemplo similar al manejo en componentes de clase:

Este enfoque actualizará todos los valores incluidos en el objeto cuando cambie cualquiera de sus valores internos.

Actualización modificando solo radio y perímetro: `setState(state => ({ radio: 3, perimetro: 4, superficie: 0 }));`

En el caso de modificar un solo valor de los 3 posibles: `setState(state => ({ ...state, superficie: 2 }));`

Ciclo useState



Clase 20: Hooks adicionales

useEffect ()

Es el Hook que se encarga de configurar y gestionar los efectos secundarios en el ciclo de vida de un componente funcional. Los hace:

- Predecibles
- Controlables

Es el equivalente, sintetizado en una sola función, de lo que podemos hacer con los métodos `componentDidMount`, `componentDidUpdate` y `componentWillUnmount` en un componente de clase.

La primera vez que miramos una implementación del Hook `useEffect` puede sorprendernos ganas de salir. ¿Podemos implementar todo el ciclo de vida en un componente funcional? tengamos fe porque la respuesta es Sí.

```
1 useEffect(() => {
2   console.log("Soy un pato, hago lo
3   que quiero!!!");
4 }, []);
5
```

Si hacemos clic en VS Code sobre la función nos llevará un archivo que nos mostrará esto:

```
1 function useEffect(effect:
2 EffectCallback, deps?: DependencyList):
3 void;
4
```

Ilustración 31: Declaración de la función técnica

En este código podemos ver que la función tiene dos argumentos: **effect** y **deps**. El primer argumento es una función callback es en donde el efecto secundario sucede, donde el código debe ser escrito. Volviendo a la implementación si incluimos la función callback como argumento en `useEffect` el resultado será el siguiente:

```
1 () => {
2   document.title = "Me gusta más este
3   título";
4 }
5
```

Ilustración 32: Función callback

```
1 useEffect(() => {
2   console.log("Soy un pato, hago lo
3   que quiero!!!");
4 });
5
```

Ilustración 33: Resultado

¿Cómo podemos simular las distintas situaciones que abarcan `componentDidMount` y `componentDidUpdate`? Con el segundo argumento: `deps`. El mismo nos brinda las opciones de sincronización, si revemos el código y los argumentos de `useEffect` nos encontramos con que `deps` termina con un signo interrogación. ¿Por qué lleva ese signo? es un operador introducido en ecmaScript 2020 que puede estar como no. Esto nos indica que una propiedad puede estar definida o indefinido (`undefined`). Si `deps` está definido nos encontraremos ante un array que puede estar vacío o incluir elementos o, mejor dicho, dependencias.

Las tres opciones de definición e indefinición de `deps` permitirán que administremos el ciclo de vida y sincronicemos nuestros componentes en términos de efectos secundarios:

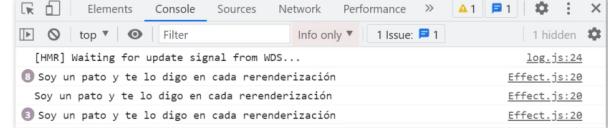
1. Ejecución por render
2. Ejecución tras el primer render
3. Correr en el primer render y luego lo hace únicamente si la variable incluida como dependencia dentro del segundo argumento cambia

Funcionamiento

useEffect ejecutará su función callback por defecto cuando el componente sea montado. Las posibilidades de posteriores ejecuciones de useEffect serán controladas por el argumento deps y acá nuevamente contamos con tres opciones:

EJECUCIÓN POR RENDER: en este caso no se ha definido el segundo argumento, como resultado useEffect se ejecutará con cada render.

```
1  useEffect(() => {
2    console.log("Soy un pato y te lo digo
3 en cada rerenderización");
4  );
5
```



EJECUCIÓN TRAS EL PRIMER RENDER: En este caso, solo se ejecuta tras el primer render. El array sin elementos o dependencias da entender que useEffect se ejecutará solo una vez tras el render inicial.

```
1  useEffect(() => {
2    console.log("Soy un pato y te lo digo
3 una sola vez");
4  }, []);
5
```



CORRER EN EL PRIMER RENDER Y LUEGO LO HACE ÚNICAMENTE SI LA VARIABLE INCLUIDA COMO DEPENDENCIA DENTRO DEL SEGUNDO ARGUMENTO CAMBIA: Cuando React monta un componente conserva un registro del array de dependencias de useEffect si se ha dado un cambio dentro de ese array, es decir si alguna dependencia modifica su valor, se volverá ejecutar useEffect. En caso contrario no ocurrirá nada.

```
1  useEffect(() => {
2    async function fetchMiVida() {
3      const response = await
4      fetch('http://mividia.com/' + year);
5      const json = await
6      response.json();
7      setMiVida(json);
8    }
9
10   fetchMiVida();
11 }, [year]);
```

En el caso de incluir dependencias dentro de useEffect es conveniente que las funciones necesarias en el efecto estén incluidas dentro del callback, es decir, dentro del primer argumento con el fin de no perder perspectiva de cómo son afectadas esas dependencias.

Características

1. Tiene acceso a las variables dentro del componente puesto que comparten el mismo ámbito.
2. Se ejecuta después del primer render y también después de cada render posterior, a no ser que se lo configure en sentido contrario.
3. Nos permite pensar de otro modo al ciclo de vida, en términos de sincronización de piezas lógicas.

Casos de uso

Podemos encontrarnos con diferentes casos de uso de useEffect. Estos son:

Establecer el título de una página de forma imperativa

El título de una página se encuentra fuera del nodo raíz de React (#root)

```
<!DOCTYPE html>
<html lang="es">

  <head>
    <title>Título</title>
  </head>

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

</html>

import React from "react";
import ReactDOM from "react-dom";
import { App } from "./components/App";
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById("root")
);
```

Sin embargo, es accesible a través de la interfaz document⁶

```
import { useEffect, useState } from "react";
const ChangeTitleExample = () => {
  const [title, setTitle] = useState(
    "Certified Tech Developer | Digital House"
  );
  useEffect(() => {
    document.title = title;
  }, [title]);
};
export default ChangeTitleExample;
```



Trabajar con temporizadores: setInterval o setTimeout

Si trabajamos con temporizadores, aquí deben introducirse:

⁶ La interfaz document representa cualquier página web cargada en el navegador y sirve como punto de entrada al contenido de la página (el árbol DOM).

Más info: <https://developer.mozilla.org/es/docs/Web/API/Document>.

```

import { useEffect, useState } from "react";
const Interval = () => {
  const [cantidad, setCantidad] = useState(2);
  useEffect(() => {
    const interval = setInterval(() => {
      setCantidad(prevState => prevState + 1);
    }, 1000);
  }, []);
  return <div>Quiero {cantidad} chocolates</div>;
};
export default Interval;

```

Leer ancho, alto o posición de elementos del DOM y registrar mensajes (console.log)

Aquí un ejemplo donde detectamos el ancho y alto de la ventana al redimensionarla y registramos un mensaje en consola.

```

import { useEffect } from "react";
const GetResizeFromWindow = () => {
  useEffect(() => {
    function handleResize() {
      console.log(
        "Redimensionar: ",
        window.innerWidth,
        "x",
        window.innerHeight
      );
    }
    window.addEventListener("resize", handleResize);
  });
  return <div />;
};
export default GetResizeFromWindow;

```

```

Redimensionar: 888 x 594
Redimensionar: 889 x 594
Redimensionar: 891 x 594
Redimensionar: 894 x 594
Redimensionar: 899 x 596
Redimensionar: 900 x 596
Redimensionar: 904 x 598
Redimensionar: 907 x 599

```

Establecer u obtener valores de almacenamiento local

Podemos usar el almacenamiento local del navegador para una gran cantidad de situaciones. Por ejemplo, en casos de corroborar si un usuario está logueado.

```

import React, { useState, useEffect } from "react";
import Login from "./Login";
import Home from "./Home";

const UserLogged = () => {
  const [user, setUser] = useState();
  useEffect(() => {
    function checkUser() {
      const item = localStorage.getItem("User");
      if (item) setUser(item);
    }
    window.addEventListener("storage", checkUser);
    return () => {
      window.removeEventListener("storage", checkUser);
    };
  }, []);
  return <>(user ? <Home /> : <Login />)</>;
};
export default Userlogged;

```

Realizar peticiones a servicios

Cuando realizamos una petición a una API, en la función Fetch o Axios, debemos declarar `async` y `await` debido a que el callback de `useEffect` es sincrónico.

```

useEffect(() => {
  async function getCandy() {
    const resp = await fetch("https://alotofcandy.iumi");
    const data = await resp.json();
    setCandy(data);
  }
  getCandy();
}, []);

```

Función de limpieza

Cuando implementamos componentes que incluyen efectos secundarios es posible que —en algunos casos— el usuario, al navegar por la aplicación, desmonte componentes antes que tales efectos secundarios finalicen. Ya sea, por ejemplo, el retorno de datos de una petición a una API externa o una función temporizadora. Esto producirá errores debido a que la aplicación buscará actualizar valores de estado de un componente que se ha eliminado.

useEffect incluye un mecanismo de limpieza, el cual es una función que se ejecutará cuando se desmonte el componente. Siempre que el componente se vuelva a renderizar, React llamará a la función de limpieza antes de ejecutar useEffect si el efecto se ejecuta de nuevo. Si hay varios efectos que necesitan ejecutarse de nuevo, React llamará a todas las funciones de limpieza para esos efectos. Una vez finalizada la limpieza, React volverá a ejecutar las funciones de efectos según sea necesario.

Clase 21: Revisión y Práctica VII

Glosario de la semana VII

HOOK: “Un Hook es una función especial que permite ‘conectarse’ a características de React.” Los Hooks son básicamente funciones. Permiten a componentes funcionales incorporar características de React, previamente reservadas solo a componentes de clase. Entre otras utilidades, dotan a los componentes funcionales de estado interno y ciclo de vida. Además —y no menos relevante— resuelven problemas que React ha enfrentado por años. Los Hooks son un modo simplificado de insertar características de React —como por ejemplo: estado, ciclo de vida, contexto, referencias, en componentes funcionales—; sin alterar el funcionamiento y las bases de React como son la importancia de los componentes, el flujo de datos y las props. Además, permiten una mejor y más simple reutilización, composición y testeo del código. Lo brillante de Hooks es que nos permiten extraer lógica de un componente para ser reutilizada y compartida.

PROBLEMAS DE LOS COMPONENTES DE CLASE:

- La lógica de estado no es fácil de reutilizar.
- Los componentes complejos son difíciles de comprender.
- La implementación de clases puede ser confusa y molesta.

SURGIMIENTO DE HOOKS: en el mes de octubre del año 2018 —en la React Conf celebrada en Henderson, Nevada, EEUU—, Sophie Alpert y Dan Abramov, miembros del equipo de Facebook encargado de desarrollar React (por ese entonces), presentaron una nueva característica revolucionaria para el mundo front end: React Hooks. Más tarde, en febrero de 2019, esta propuesta se hizo realidad y React liberó su versión 16.8 incluyendo la API de Hooks.

BENEFICIOS HOOKS:

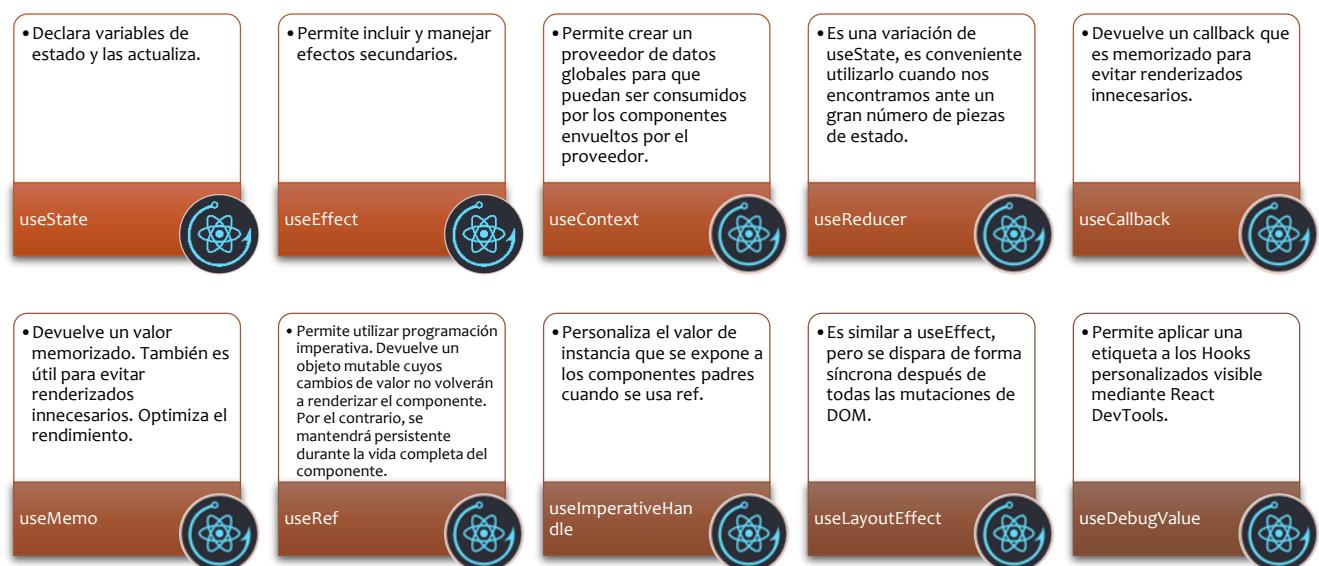
- Menor cantidad de código.
- Código más organizado.
- Utilización de funciones reutilizables.

- Mayor facilidad para testear.
- No llama a super() en un constructor de clases.
- No trabaja con this y bind en el uso de manejadores.
- Simplifica la vida y el ciclo de vida.
- El estado local está dentro del alcance de los handlers y las funciones de efectos secundarios.
- Componentes más reducidos que facilitan el trabajo de React.

CONVIVENCIA ENTRE COMPONENTES DE CLASE Y HOOKS: en React conviven ambos tipos de programación. Es decir, que en una misma aplicación pueden convivir componentes de ambos tipos, para dar tiempo a un cambio de mentalidad gradual.

Como ejemplo de esta decisión, Facebook conserva una enorme cantidad de código escrito en clases. Finalmente, debemos aclarar que no existe la libertad absoluta entre ambos tipos de componentes, ya que no podemos usar Hooks dentro de un componente de clase.

HOOKS NATIVOS



USESTATE: desde el lanzamiento de Hooks, los componentes funcionales pueden declarar y actualizar su estado interno mediante el hook useState. Este es una función que crea una variable que permite almacenar una pieza de estado interno y, a la vez, actualizar dicho valor en un componente funcional.

ESTRUCTURA DE USESTATE: este recibe un argumento y devuelve un array con 2 elementos. El primero será el valor de la variable y el segundo la función para modificarla.

FUNCIONAMIENTO DE USESTATE: cuando React renderiza o rerenderiza nuestro componente, este solicita a React el último valor de estado con el fin de dibujar la interfaz de usuario y también la función para actualizar el valor de estado. Si se ejecuta la función para modificar el valor, React rerenderizará la interfaz. React comparará la interfaz de usuario generada con la versión almacenada y actualizará la pantalla del modo más eficiente.

CICLO USESTATE:

1. React atraviesa el árbol de componentes montándolos para dibujar la UI. En cada caso, React pasará las props correspondientes.
2. Cada componente invoca useState por primera vez, solicitando el valor inicial a useState que a su vez se lo requiere a React.
3. React retorna tanto el valor actual como la función de actualización.
4. El componente configura los controladores de eventos de usuario.
5. El componente utiliza el valor de estado obtenido para declarar la UI.
6. React actualiza el DOM con las modificaciones requeridas.
7. Cuando el controlador de eventos invoca a la función de actualización. Se activa un evento y se ejecuta el controlador. A través del handler, se invoca a la función de actualización para modificar el valor del estado.
8. React modifica el valor del estado con el valor pasado por la función de actualización.
9. React llama al componente.
10. El componente vuelve a llamar a useState, pero sin tener en cuenta el argumento pasado inicialmente.
11. React vuelve a retornar el valor de estado actual y la función de actualización.
12. El componente crea una nueva versión del manejador de eventos y utiliza el nuevo valor de estado.
13. El componente devuelve su IU con el nuevo valor de estado.
14. React actualiza el DOM con las modificaciones requeridas.

USEEFFECT: es el Hook que se encarga de configurar y gestionar los efectos secundarios en el ciclo de vida de un componente funcional. Los hace predecibles, controlables. Es el equivalente, sintetizado en una sola función, de lo que podemos hacer con los métodos componentDidMount, componentDidUpdate y componentWillUnmount en un componente de clase.

CARACTERÍSTICAS DE USEEFFECT:

- Tiene acceso a las variables dentro del componente puesto que comparten el mismo ámbito.
- Se ejecuta después del primer render y también después de cada render posterior, a no ser que se lo configure en sentido contrario.
- Nos permite pensar de otro modo al ciclo de vida, en términos de sincronización de piezas lógicas.

ESTRUCTURA DE USEEFFECT: la función tiene 2 argumentos: effect y deps. El primer argumento es una función callback. Es en donde el efecto secundario sucede, donde el código debe ser escrito. El segundo argumento nos brinda las opciones de sincronización.

OPCIONES DE SINCRONIZACIÓN DE USEEFFECT:

- Se ejecuta en cada render

- Solo se ejecuta tras el primer render.
- Corre en el primer render y luego solo si varía la variable incluida como dependencia dentro del segundo argumento.

CASOS DE USO DE USEEFFECT:

- Establecer el título de una página de forma imperativa.
- Trabajar con temporizadores: setInterval o setTimeout.
- Leer ancho, alto o posición de elementos del DOM.
- Registrar mensajes.
- Establecer u obtener valores de almacenamiento local.
- Realizar peticiones a servicios.

FUNCIÓN DE LIMPIEZA DE USEEFFECT: cuando implementamos componentes que incluyen efectos secundarios es posible —en algunos casos— que el usuario al navegar por la aplicación desmonte componentes antes que tales efectos secundarios finalicen. Ya sea, por ejemplo, el retorno de datos de una petición a una API externa o una función temporizadora. Esto producirá errores debido a que la aplicación buscará actualizar valores de estado de un componente que se ha eliminado. useEffect incluye un mecanismo de limpieza, el cual es una función que se ejecutará cuando se desmonte el componente. Siempre que el componente se vuelva a renderizar, React llamará a la función de limpieza antes de ejecutar useEffect si el efecto se ejecuta de nuevo. Si hay varios efectos que necesitan ejecutarse de nuevo, React llamará a todas las funciones de limpieza para esos efectos. Una vez finalizada la limpieza, React volverá a ejecutar las funciones de efectos según sea necesario.