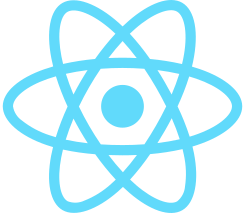
**Curso Práctico de React JS**

**React**



React.svg

**React** (también llamada **React.js** o ReactJS) es una biblioteca [Javascript](https://es.wikipedia.org/wiki/JavaScript) de [código abierto](https://es.wikipedia.org/wiki/C%C3%B3digo_abierto) diseñada para crear [interfaces de usuario](https://es.wikipedia.org/wiki/Interfaz_de_usuario) con el objetivo de facilitar el desarrollo de [aplicaciones en una sola página](https://es.wikipedia.org/wiki/Single-page_application). Es mantenido por [Facebook](https://es.wikipedia.org/wiki/Facebook) y la comunidad de [software libre](https://es.wikipedia.org/wiki/Software_libre), han participado en el proyecto más de mil [desarrolladores](https://es.wikipedia.org/wiki/Desarrollador_de_software) diferentes. [1](https://es.wikipedia.org/wiki/React#cite_note-infoworld-1)​

React intenta ayudar a los desarrolladores a construir [aplicaciones](https://es.wikipedia.org/wiki/Aplicaci%C3%B3n_inform%C3%A1tica) que usan datos que cambian todo el [tiempo](https://es.wikipedia.org/wiki/Tiempo). Su objetivo es ser sencillo, declarativo y fácil de combinar. React sólo maneja la [interfaz de usuario](https://es.wikipedia.org/wiki/Interfaz_de_usuario) en una aplicación; React es la **Vista** en un contexto en el que se use el patrón [MVC](https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador) (Modelo-Vista-Controlador) o [MVVM](https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93modelo_de_vista) (Modelo-vista-modelo de vista). También puede ser utilizado con las extensiones de React-based que se encargan de las partes no-UI (que no forman parte de la interfaz de usuario) de una [aplicación web](https://es.wikipedia.org/wiki/Aplicaci%C3%B3n_web).

Según el servicio de análisis Javascript (en inglés "javascript analytics service"), Libscore, React actualmente está siendo utilizado en las páginas principales de Imgur, Bleacher Informe, [Feedly](https://es.wikipedia.org/wiki/Feedly), [Airbnb](https://es.wikipedia.org/wiki/Airbnb), SeatGeek, HelloSign, y otras.

**Historia**

React fue creada por Jordan Walke, un ingeniero de software en Facebook, inspirado por los problemas que tenía la compañía con el mantenimiento del código de los anuncios dentro de su plataforma. Enfocado en la experiencia del usuario y la eficiencia para sus programadores, influenciado por XHP (un marco de componentes de [HTML](https://es.wikipedia.org/wiki/HTML) para [PHP](https://es.wikipedia.org/wiki/PHP)), nace el prototipo ReactJS.

**DOM virtual y detalles de implementación**

**¿Qué es el DOM virtual?**

El DOM virtual (VDOM) es un concepto de programación donde una representación ideal o “virtual” de la IU se mantiene en memoria y en sincronía con el DOM “real”, mediante una biblioteca como ReactDOM. Este proceso se conoce como [reconciliación](https://es.reactjs.org/docs/reconciliation.html).

Este enfoque hace posible la API declarativa de React: le dices a React en qué estado quieres que esté la IU, y se hará cargo de llevar el DOM a ese estado. Esto abstrae la manipulación de atributos, manejo de eventos y actualización manual del DOM que de otra manera tendrías que usar para construir tu aplicación.

Ya que “DOM virtual” es más un patrón que una tecnología específica, las personas a veces le dan significados diferentes. En el mundo de React, el término “DOM virtual” es normalmente asociado con [elementos de React](https://es.reactjs.org/docs/rendering-elements.html) ya que son objetos representando la interfaz de usuario. Sin embargo, React también usa objetos internos llamados “fibers” para mantener información adicional acerca del árbol de componentes. Éstos pueden ser también considerados como parte de la implementación de “DOM virtual” de React.

**¿Es el Shadow DOM lo mismo que el DOM virtual?**

No, son diferentes. El Shadow DOM es una tecnología de los navegadores diseñada principalmente para limitar el alcance de variables y CSS en componentes web. El DOM virtual es un concepto que implementan bibliotecas en JavaScript por encima de las APIs de los navegadores.

**Create React App**

[Create React App](https://github.com/facebookincubator/create-react-app) es un ambiente cómodo para **aprender React**, y es la mejor manera de comenzar a construir **una nueva** [**aplicación de página única**](https://es.reactjs.org/docs/glossary.html#single-page-application) usando React.

**Create React App** configura tu ambiente de desarrollo de forma que puedas usar las últimas características de Javascript, brindando una buena experiencia de desarrollo, y optimizando tu aplicación para producción. Necesitarás tener Node >= 8.10 y npm >= 5.6 instalados en tu máquina. Para crear un proyecto ejecuta:

npx create-react-app my-app

cd my-app

npm start

Nota

En la primera línea npx no es un error de escritura: Es una [herramienta de ejecución de paquetes que viene con npm 5.2+](https://medium.com/@maybekatz/introducing-npx-an-npm-package-runner-55f7d4bd282b).

Create React App no se encarga de la lógica de backend o de bases de datos; tan solo crea un flujo de construcción para frontend, de manera que lo puedes usar con cualquier backend. Para ello internamente usa [Babel](https://babeljs.io/) y [webpack](https://webpack.js.org/), pero no necesitas saber nada de estas herramientas para usar Create React App.

Cuando estés listo para desplegar a producción, ejecuta npm run build lo cual crea una compilación optimizada de tu aplicación en el directorio build. Puedes aprender más acerca de **Create React App** [en su archivo README](https://github.com/facebookincubator/create-react-app#create-react-app--) y en la [Guía del Usuario](https://facebook.github.io/create-react-app/).

**Hola mundo**

El más pequeño de los ejemplos de React se ve así:

ReactDOM.render(

<h1>Hello, world!</h1>,

document.getElementById('root')

);

Este muestra un encabezado con el texto “Hello, world!” en la página.

[**Stateful and Stateless Components in React**](https://programmingwithmosh.com/javascript/stateful-stateless-components-react/)

**Componentes con estado y sin estado**

Hoy, revisaremos qué componentes con estado y sin estado se encuentran en React, cómo puede notar la diferencia y el complejo proceso de decidir si los componentes tienen estado o no.

**Revisión del estado**

Primero, repasemos qué estado es.

En un componente, el estado son datos que importamos, generalmente para mostrar al usuario, que están sujetos a cambios. Podría cambiar porque la base de datos de la que estamos obteniendo puede actualizarse, el usuario la modificó, ¡hay muchas razones por las que los datos cambian!

import React, {Component} from 'react'

class Pigeons extends Component {

constructor() {

super()

this.state = {

pigeons: []

}

}

render() {

return (

<div>

<p>Look at all the pigeons spotted today!</p>

<ul>

{this.state.pigeons.map(pigeonURL => {

return <li><img src={pigeonURL} /></li>

})}

</ul>

</div>

)

}

}

Por lo general, también tendríamos un componenteDidMount () que tomaría nuestros datos de una base de datos, pero el ejemplo anterior debería darle una idea básica: tenemos estado y podemos representar cosas desde el estado.

**Componentes con estado y sin estado**

Los componentes con estado y sin estado tienen muchos nombres diferentes.

También son conocidos como:

* Componentes de contenedor vs presentación
* Componentes inteligentes vs tontos

La diferencia literal es que uno tiene estado y el otro no. Eso significa que los componentes con estado realizan un seguimiento de los datos cambiantes, mientras que los componentes sin estado imprimen lo que se les da a través de accesorios, o siempre representan lo mismo.

Componente con estado / contenedor / inteligente:

class Main extends Component {

constructor() {

super()

this.state = {

books: []

}

}

render() {

<BooksList books={this.state.books} />

}

}

Sin estado / Presentacional / Componente tonto:

const BooksList = ({books}) => {

return (

<ul>

{books.map(book => {

return <li>book</li>

})}

</ul>

)

}

Observe que el componente sin estado se escribe como una función. Tan bueno como es el estado, siempre debe tratar de hacer que sus componentes sean lo más simples y sin estado posible, de modo que los diferentes componentes se puedan reutilizar como piezas de Lego, incluso si no tiene planes inmediatos para reutilizar un componente. ¡Los con estado deberían sentirse afortunados de ser así!

**Componentes Visuales**

Conocidos en inglés como Presentational Components. Este tipo de componentes solo deben centrase y enfocar sus esfuerzos en como debe renderizarse la UI. Este tipo de componentes puede componerse de otros elementos visuales y suele incluir estilos y clases. Todos los datos implicados en su renderización se deben recibir a través de props, por lo que deben ser independientes de llamadas a servicios externos. Este tipo de componentes suelen ser de tipo Stateless ya que no necesitan estado, y deben de gestionar las acciones pasándoselas a componentes padre a través de sus props.

***Ejemplo:***

class Item extends React.Component {

render () {

return (

<li><a href='#'>{ this.props.valor }</a></li>

);

}

}

**Presentando JSX**

Considera la declaración de esta variable:

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

Esta curiosa sintaxis de etiquetas no es ni un string ni HTML.

Se llama JSX, y es una extensión de la sintaxis de JavaScript. Recomendamos usarlo con React para describir cómo debería ser la interfaz de usuario. JSX puede recordarte a un lenguaje de plantillas, pero viene con todo el poder de JavaScript.

JSX produce “elementos” de React. Exploraremos como renderizarlos en el DOM en la [siguiente sección](https://es.reactjs.org/docs/rendering-elements.html). A continuación puedes encontrar lo básico de JSX que será necesario para empezar.

**¿Por qué JSX?**

React acepta el hecho de que la lógica de renderizado está intrínsecamente unida a la lógica de la interfaz de usuario: cómo se manejan los eventos, cómo cambia el estado con el tiempo y cómo se preparan los datos para su visualización.

En lugar de separar artificialmente *tecnologías* poniendo el maquetado y la lógica en archivos separados, React [separa *intereses*](https://es.wikipedia.org/wiki/Separaci%C3%B3n_de_intereses) con unidades ligeramente acopladas llamadas “componentes” que contienen ambas. Volveremos a los componentes en [otra sección](https://es.reactjs.org/docs/components-and-props.html), pero si aún no te sientes cómodo maquetando en JS, [esta charla](https://www.youtube.com/watch?v=x7cQ3mrcKaY) podría convencerte de lo contrario.

React [no requiere](https://es.reactjs.org/docs/react-without-jsx.html) usar JSX, pero la mayoría de la gente lo encuentra útil como ayuda visual cuando trabajan con interfaz de usuario dentro del código Javascript. Esto también permite que React muestre mensajes de error o advertencia más útiles.

Con eso fuera del camino, ¡empecemos!

**Insertando expresiones en JSX**

En el ejemplo a continuación, declaramos una variable llamada name y luego la usamos dentro del JSX envolviéndola dentro de llaves:

const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>;

ReactDOM.render(

element,

document.getElementById('root')

);

Puedes poner cualquier [expresión de JavaScript](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Expressions_and_Operators) dentro de llaves en JSX. Por ejemplo, 2 + 2, user.firstName, o formatName(user) son todas expresiones válidas de Javascript.

En el ejemplo a continuación, insertamos el resultado de llamar la función de JavaScript, formatName(user), dentro de un elemento <h1>.

function formatName(user) {

return user.firstName + ' ' + user.lastName;

}

const user = {

firstName: 'Harper',

lastName: 'Perez'

};

const element = (

<h1>

Hello, {formatName(user)}!

</h1>

);

ReactDOM.render(

element,

document.getElementById('root')

);

[**Pruébalo en CodePen**](https://es.reactjs.org/redirect-to-codepen/introducing-jsx)

Dividimos JSX en varias líneas para facilitar la lectura. Aunque no es necesario, cuando hagas esto también te recomendamos envolverlo entre paréntesis para evitar errores por la [inserción automática del punto y coma](https://stackoverflow.com/q/2846283).

**JSX también es una expresión**

Después de compilarse, las expresiones JSX se convierten en llamadas a funciones JavaScript regulares y se evalúan en objetos JavaScript.

Esto significa que puedes usar JSX dentro de declaraciones if y bucles for, asignarlo a variables, aceptarlo como argumento, y retornarlo desde dentro de funciones:

function getGreeting(user) {

if (user) {

return <h1>Hello, {formatName(user)}!</h1>;

}

return <h1>Hello, Stranger.</h1>;

}

**Especificando atributos con JSX**

Puedes utilizar comillas para especificar strings literales como atributos:

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

También puedes usar llaves para insertar una expresión JavaScript en un atributo:

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

No pongas comillas rodeando llaves cuando insertes una expresión JavaScript en un atributo. Debes utilizar comillas (para los valores de los strings) o llaves (para las expresiones), pero no ambas en el mismo atributo.

**Advertencia:**

Dado que JSX es más cercano a JavaScript que a HTML, React DOM usa la convención de nomenclatura camelCase en vez de nombres de atributos HTML.

Por ejemplo, class se vuelve [className](https://developer.mozilla.org/es/docs/Web/API/Element/className) en JSX, y tabindex se vuelve [tabIndex](https://developer.mozilla.org/es/docs/Web/API/HTMLElement/tabIndex).

**Especificando hijos con JSX**

Si una etiqueta está vacía, puedes cerrarla inmediatamente con />, como en XML:

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

Las etiquetas de Javascript pueden contener hijos:

const element = (

<div>

<h1>Hello!</h1>

<h2>Good to see you here.</h2>

</div>

);

**JSX previene ataques de inyección**

Es seguro insertar datos ingresados por el usuario en JSX:

const title = response.potentiallyMaliciousInput;

// Esto es seguro:

const element = <h1>{title}</h1>;

Por defecto, React DOM [escapa](https://stackoverflow.com/questions/7381974/which-characters-need-to-be-escaped-on-html) cualquier valor insertado en JSX antes de renderizarlo. De este modo, se asegura de que nunca se pueda insertar nada que no esté explícitamente escrito en tú aplicación. Todo es convertido en un string antes de ser renderizado. Esto ayuda a prevenir vulnerabilidades [XSS (cross-site-scripting)](https://es.wikipedia.org/wiki/Cross-site_scripting).

**JSX representa objetos**

Babel compila JSX a llamadas de React.createElement().

Estos dos ejemplos son idénticos:

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

);

React.createElement() realiza algunas comprobaciones para ayudarte a escribir código libre de errores, pero, en esencia crea un objeto como este:

// Nota: Esta estructura está simplificada

const element = {

type: 'h1',

props: {

className: 'greeting',

children: 'Hello, world!'

}

};

Estos objetos son llamados “Elementos de React”. Puedes pensar en ellos como descripciones de lo que quieres ver en pantalla. React lee estos objetos y los usa para construir el DOM y mantenerlo actualizado.

Vamos a explorar el renderizado de los elementos de React al DOM en la siguiente sección.

**Tip:**

Recomendamos usar la [Definición del lenguaje en “Babel”](https://babeljs.io/docs/editors) en tu editor de elección para que tanto el código en ES6 como el código en JSX sea resaltado apropiadamente. Este sitio web utiliza el esquema de color [Oceanic Next](https://labs.voronianski.com/oceanic-next-color-scheme/), el cual es compatible con esto.

**Estilo y CSS**

**¿Cómo agrego clases CSS a los componentes?**

Pasa una string como la prop className:

render() {

return <span className="menu navigation-menu">Menu</span>

}

Es común que las clases CSS dependan de las props o del estado del componente:

render() {

let className = 'menu';

if (this.props.isActive) {

className += ' menu-active';

}

return <span className={className}>Menu</span>

}

Tip

Si a menudo escribes código como este, el paquete [classnames](https://www.npmjs.com/package/classnames#usage-with-reactjs) puede hacerlo más simple.

**¿Puedo usar estilos en línea?**

Sí, ve la documentación sobre estilo [aquí](https://es.reactjs.org/docs/dom-elements.html#style).

**¿Los estilos en línea son malos?**

Las clases CSS son generalmente mejores para el rendimiento que los estilos en línea.

**¿Qué es CSS-in-JS?**

“CSS-in-JS” se refiere a un patrón donde el CSS se compone usando JavaScript en lugar de definirlo en archivos externos. Lee una comparación de las bibliotecas CSS-in-JS [aquí](https://github.com/MicheleBertoli/css-in-js).

*Ten en cuenta que esta funcionalidad no es parte de React, sino que es proporcionada por bibliotecas de terceros.* React no tiene una opinión sobre cómo se definen los estilos; en caso de dudas, un buen punto de partida es definir tus estilos en un archivo \*.css separado como de costumbre y referirse a ellos usando [className](https://es.reactjs.org/docs/dom-elements.html#classname).

**¿Puedo hacer animaciones en React?**

React puede usarse para potenciar animaciones. Revisa [React Transition Group](https://reactcommunity.org/react-transition-group/) y [React Motion](https://github.com/chenglou/react-motion) o [React Spring](https://github.com/react-spring/react-spring), por ejemplo.

**Componentes y propiedades**

Los componentes permiten separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada.Esta página proporciona una introducción a la idea de los componentes. Puedes encontrar una [API detallada sobre componentes aquí](https://es.reactjs.org/docs/react-component.html).

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

**Componentes funcionales y de clase**

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

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

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

También puedes utilizar una [clase de ES6](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes) para definir un componente:

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

Los dos componentes anteriores son equivalentes desde el punto de vista de React.

Las clases tienen algunas características adicionales que veremos en las [próximas secciones](https://es.reactjs.org/docs/state-and-lifecycle.html). Hasta entonces, usaremos componentes funcionales por su brevedad.

**Renderizando un componente**

Anteriormente, sólo encontramos elementos de React que representan las etiquetas del DOM:

const element = <div />;

Sin embargo, los elementos también pueden representar componentes definidos por el usuario:

const element = <Welcome name="Sara" />;

Cuando React ve un elemento representando un componente definido por el usuario, pasa atributos JSX a este componente como un solo objeto. Llamamos a este objeto “props”.

Por ejemplo, este código muestra “Hello, Sara” en la página:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

const element = <Welcome name="Sara" />;

ReactDOM.render(

element,

document.getElementById('root')

);

[**Pruébalo en CodePen**](https://es.reactjs.org/redirect-to-codepen/components-and-props/rendering-a-component)

Recapitulemos lo que sucede en este ejemplo:

1. Llamamos a ReactDOM.render() con el elemento <Welcome name="Sara" />.
2. React llama al componente Welcome con {name: 'Sara'} como “props”.
3. Nuestro componente Welcome devuelve un elemento <h1>Hello, Sara</h1> como resultado.
4. React DOM actualiza eficientemente el DOM para que coincida con <h1>Hello, Sara</h1>.

**Nota:** Comienza siempre los nombres de componentes con una letra mayúscula.

React trata los componentes que empiezan con letras minúsculas como etiquetas del DOM. Por ejemplo, <div /> representa una etiqueta div HTML pero <Welcome /> representa un componente y requiere que Welcome esté definido.

Para saber más sobre el razonamiento detrás de esta convención, puedes consultar [JSX en profundidad](https://es.reactjs.org/docs/jsx-in-depth.html#user-defined-components-must-be-capitalized).

**Composición de componentes**

Los componentes pueden referirse a otros componentes en su salida. Esto nos permite utilizar la misma abstracción de componente para cualquier nivel de detalle. Un botón, un cuadro de diálogo, un formulario, una pantalla: en aplicaciones de React, todos son expresados comúnmente como componentes.

Por ejemplo, podemos crear un componente App que renderiza Welcome muchas veces:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

function App() {

return (

<div>

<Welcome name="Sara" />

<Welcome name="Cahal" />

<Welcome name="Edite" />

</div>

);

}

ReactDOM.render(

<App />,

document.getElementById('root')

);

[**Pruébalo en CodePen**](https://es.reactjs.org/redirect-to-codepen/components-and-props/composing-components)

Por lo general, las aplicaciones de React nuevas tienen un único componente App en lo más alto. Sin embargo, si se integra React en una aplicación existente, se podría empezar de abajo hacia arriba con un pequeño componente como Button y poco a poco trabajar el camino a la cima de la jerarquía de la vista.

**Extracción de componentes**

No tengas miedo de dividir los componentes en otros más pequeños.

Por ejemplo, considera este componente Comment:

function Comment(props) {

return (

<div className="Comment">

<div className="UserInfo">

<img className="Avatar"

src={props.author.avatarUrl}

alt={props.author.name}

/>

<div className="UserInfo-name">

{props.author.name}

</div>

</div>

<div className="Comment-text">

{props.text}

</div>

<div className="Comment-date">

{formatDate(props.date)}

</div>

</div>

);

}

[**Pruébalo en CodePen**](https://es.reactjs.org/redirect-to-codepen/components-and-props/extracting-components)

Acepta author (un objeto), text (un string), y date (una fecha) como props, y describe un comentario en una web de redes sociales.

Este componente puede ser difícil de cambiar debido a todo el anidamiento, y también es difícil reutilizar partes individuales de él. Vamos a extraer algunos componentes del mismo.

Primero, vamos a extraer Avatar:

function Avatar(props) {

return (

<img className="Avatar"

src={props.user.avatarUrl}

alt={props.user.name}

/>

);

}

El Avatar no necesita saber que está siendo renderizado dentro de un Comment. Esto es por lo que le dimos a su propiedad un nombre más genérico: user en vez de author.

Recomendamos nombrar las props desde el punto de vista del componente, en vez de la del contexto en el que se va a utilizar.

Ahora podemos simplificar Comment un poquito:

function Comment(props) {

return (

<div className="Comment">

<div className="UserInfo">

<Avatar user={props.author} />

<div className="UserInfo-name">

{props.author.name}

</div>

</div>

<div className="Comment-text">

{props.text}

</div>

<div className="Comment-date">

{formatDate(props.date)}

</div>

</div>

);

}

A continuacion, vamos a extraer un componente UserInfo que renderiza un Avatar al lado del nombre del usuario:

function UserInfo(props) {

return (

<div className="UserInfo">

<Avatar user={props.user} />

<div className="UserInfo-name">

{props.user.name}

</div>

</div>

);

}

Esto nos permite simplificar Comment aun más:

function Comment(props) {

return (

<div className="Comment">

<UserInfo user={props.author} />

<div className="Comment-text">

{props.text}

</div>

<div className="Comment-date">

{formatDate(props.date)}

</div>

</div>

);

}

[**Pruébalo en CodePen**](https://es.reactjs.org/redirect-to-codepen/components-and-props/extracting-components-continued)

Extraer componentes puede parecer un trabajo pesado al principio, pero tener una paleta de componentes reutilizables vale la pena en aplicaciones más grandes. Una buena regla en general es que si una parte de su interfaz de usuario se usa varias veces (Button, Panel, Avatar), o es lo suficientemente compleja por sí misma (App, FeedStory, Comment), es buen candidato para ser un componente reutilizable.

**Las props son de solo lectura**

Ya sea que declares un componente [como una función o como una clase](https://es.reactjs.org/docs/components-and-props.html#function-and-class-components), este nunca debe modificar sus props. Considera esta función sum :

function sum(a, b) {

return a + b;

}

Tales funciones son llamadas [“puras”](https://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional#Funciones_puras) por que no tratan de cambiar sus entradas, y siempre devuelven el mismo resultado para las mismas entradas.

En contraste, esta función es impura por que cambia su propia entrada:

function withdraw(account, amount) {

account.total -= amount;

}

React es bastante flexible pero tiene una sola regla estricta:

**Todos los componentes de React deben actuar como funciones puras con respecto a sus props.**

**Estado y ciclo de vida**

Esta página introduce el concepto de estado y ciclo de vida en un componente de React. Puedes encontrar una [referencia detallada de la API de un componente aquí](https://es.reactjs.org/docs/react-component.html).

Consideremos el ejemplo del reloj de [una de las secciones anteriores](https://es.reactjs.org/docs/rendering-elements.html#updating-the-rendered-element). En [Renderizando elementos](https://es.reactjs.org/docs/rendering-elements.html#rendering-an-element-into-the-dom), aprendimos solo una forma de actualizar la interfaz de usuario. Invocamos a ReactDOM.render() para que cambie el resultado renderizado.

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(

element,

document.getElementById('root')

);

}

setInterval(tick, 1000);

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/gwoJZk?editors=0010)

En esta sección, aprenderemos como hacer al componente Clock verdaderamente reutilizable y encapsulado. Configurarás tu propio temporizador y se actualizará cada segundo.

Podemos comenzar por encapsular cómo se ve el reloj:

function Clock(props) {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {props.date.toLocaleTimeString()}.</h2>

</div>

);

}

function tick() {

ReactDOM.render(

<Clock date={new Date()} />,

document.getElementById('root')

);

}

setInterval(tick, 1000);

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/dpdoYR?editors=0010)

Sin embargo, se pierde un requisito crucial: el hecho de que Clock configure un temporizador y actualice la interfaz de usuario cada segundo debe ser un detalle de implementación de Clock.

Idealmente, queremos escribir esto una vez y que Clock se actualice a sí mismo:

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

Para implementar esto, necesitamos agregar «estado» al componente Clock.

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

**Convertir una función en una clase**

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

1. Crear una [clase ES6](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes) con el mismo nombre que herede de React.Component.
2. Agregar un único método vacío llamado render().
3. Mover el cuerpo de la función al método render().
4. Reemplazar props con this.props en el cuerpo de render().
5. Borrar el resto de la declaración de la función ya vacía.

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.props.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/zKRGpo?editors=0010)

Clock ahora se define como una clase en lugar de una función.

El método render se invocará cada vez que ocurre una actualización; pero, siempre y cuando rendericemos <Clock /> en el mismo nodo del DOM, se usará solo una única instancia de la clase Clock. Esto nos permite utilizar características adicionales como el estado local y los métodos de ciclo de vida.

**Agregar estado local a una clase**

Moveremos date de las props hacia el estado en tres pasos:

1. Reemplazar this.props.date con this.state.date en el método render():

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

1. Añadir un [constructor de clase](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes#Constructor) que asigne el this.state inicial:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

Nota cómo pasamos props al constructor base:

constructor(props) {

super(props);

this.state = {date: new Date()};

}

Los componentes de clase siempre deben invocar al constructor base con props.

1. Eliminar la prop date del elemento <Clock />:

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

Posteriormente regresaremos el código del temporizador al propio componente.

El resultado es el siguiente:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/KgQpJd?editors=0010)

A continuación, haremos que Clock configure su propio temporizador y se actualice cada segundo.

**Agregar métodos de ciclo de vida a una clase**

En aplicaciones con muchos componentes, es muy importante liberar recursos tomados por los componentes cuando se destruyen.

Queremos [configurar un temporizador](https://es.reactjs.org/docs/%60https:/developer.mozilla.org/es/docs/Web/API/WindowTimers/setInterval%60) cada vez que Clock se renderice en el DOM por primera vez. Esto se llama «montaje» en React.

También queremos [borrar ese temporizador](https://developer.mozilla.org/es/docs/Web/API/WindowTimers/clearInterval) cada vez que el DOM producido por Clock se elimine. Esto se llama «desmontaje» en React.

Podemos declarar métodos especiales en la clase del componente para ejecutar algún código cuando un componente se monta y desmonta:

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

componentDidMount() {

}

componentWillUnmount() {

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

Estos métodos son llamados «métodos de ciclo de vida».

El método componentDidMount() se ejecuta después que la salida del componente ha sido renderizada en el DOM. Este es un buen lugar para configurar un temporizador:

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

Nota como guardamos el ID del temporizador en this (this.timerID).

Si bien this.props es configurado por el mismo React y this.state tiene un significado especial, eres libre de añadir campos adicionales a la clase manualmente si necesitas almacenar algo que no participa en el flujo de datos (como el ID de un temporizador).

Eliminaremos el temporizador en el método de ciclo de vida componentWillUnmount():

componentWillUnmount() {

clearInterval(this.timerID);

}

Finalmente, implementaremos un método llamado tick() que el componente Clock ejecutará cada segundo.

Utilizará this.setState() para programar actualizaciones al estado local del componente.

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {date: new Date()};

}

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

componentWillUnmount() {

clearInterval(this.timerID);

}

tick() {

this.setState({

date: new Date()

});

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/amqdNA?editors=0010)

Ahora el reloj cambia cada segundo.

Repasemos rápidamente lo que está sucediendo y el orden en que se invocan los métodos:

1. Cuando se pasa <Clock /> a ReactDOM.render(), React invoca al constructor del componente Clock. Ya que Clock necesita mostrar la hora actual, inicializa this.state con un objeto que incluye la hora actual. Luego actualizaremos este estado.
2. React invoca entonces al método render() del componente Clock. Así es como React sabe lo que se debe mostrar en pantalla. React entonces actualiza el DOM para que coincida con la salida del renderizado de Clock.
3. Cuando la salida de Clock se inserta en el DOM, React invoca al método de ciclo de vida componentDidMount(). Dentro de él, el componente Clock le pide al navegador que configure un temporizador para invocar al método tick() del componente una vez por segundo.
4. Cada segundo el navegador invoca al método tick(). Dentro de él, el componente Clock planifica una actualización de la interfaz de usuario al invocar a setState() con un objeto que contiene la hora actual. Gracias a la invocación a setState(), React sabe que el estado cambió e invoca de nuevo al método render() para saber qué debe estar en la pantalla. Esta vez, this.state.date en el método render() será diferente, por lo que el resultado del renderizado incluirá la hora actualizada. Conforme a eso React actualiza el DOM.
5. Si el componente Clock se elimina en algún momento del DOM, React invoca al método de ciclo de vida componentWillUnmount(), por lo que el temporizador se detiene.

**Usar el estado correctamente**

Hay tres cosas que debes saber sobre setState().

**No modifiques el estado directamente**

Por ejemplo, esto no volverá a renderizar un componente:

// Incorrecto

this.state.comment = 'Hello';

En su lugar utiliza setState():

// Correcto

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

El único lugar donde puedes asignar this.state es el constructor.

**Las actualizaciones del estado pueden ser asíncronas**

React puede agrupar varias invocaciones a setState() en una sola actualización para mejorar el rendimiento.

Debido a que this.props y this.state pueden actualizarse de forma asincrónica, no debes confiar en sus valores para calcular el siguiente estado.

Por ejemplo, este código puede fallar en actualizar el contador:

// Incorrecto

this.setState({

counter: this.state.counter + this.props.increment,

});

Para arreglarlo, usa una segunda forma de setState() que acepta una función en lugar de un objeto. Esa función recibirá el estado previo como primer argumento, y las props en el momento en que se aplica la actualización como segundo argumento:

// Correcto

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

Anteriormente usamos una [función flecha](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/Arrow_functions), pero se podría haber hecho igualmente con funciones comunes:

// Correcto

this.setState(function(state, props) {

return {

counter: state.counter + props.increment

};

});

**Las actualizaciones de estado se fusionan**

Cuando invocas a setState(), React combina el objeto que proporcionaste con el estado actual.

Por ejemplo, tu estado puede contener varias variables independientes:

constructor(props) {

super(props);

this.state = {

posts: [],

comments: []

};

}

Luego puedes actualizarlas independientemente con invocaciones separadas a setState():

componentDidMount() {

fetchPosts().then(response => {

this.setState({

posts: response.posts

});

});

fetchComments().then(response => {

this.setState({

comments: response.comments

});

});

}

La fusión es superficial, asi que this.setState({comments}) deja intacto a this.state.posts, pero reemplaza completamente this.state.comments.

**Los datos fluyen hacia abajo**

Ni los componentes padres o hijos pueden saber si un determinado componente tiene o no tiene estado y no les debería importar si se define como una función o una clase.

Por eso es que el estado a menudo se le denomina local o encapsulado. No es accesible desde otro componente excepto de aquel que lo posee y lo asigna.

Un componente puede elegir pasar su estado como props a sus componentes hijos:

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

Esto también funciona para componentes definidos por el usuario:

<FormattedDate date={this.state.date} />

El componente FormattedDate recibiría date en sus props y no sabría si vino del estado de Clock, de los props de Clock, o si se escribió manualmente:

function FormattedDate(props) {

return <h2>It is {props.date.toLocaleTimeString()}.</h2>;

}

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/zKRqNB?editors=0010)

A esto comúnmente se le llama flujo de datos «descendente» o «unidireccional». Cualquier estado siempre es propiedad de algún componente específico, y cualquier dato o interfaz de usuario derivados de ese estado solo pueden afectar los componentes «debajo» de ellos en el árbol.

Si imaginas un árbol de componentes como una cascada de props, el estado de cada componente es como una fuente de agua adicional que se le une en un punto arbitrario, pero también fluye hacia abajo.

Para mostrar que todos los componentes están verdaderamente aislados, podemos crear un componente App que represente tres componentes <Clock>:

function App() {

return (

<div>

<Clock />

<Clock />

<Clock />

</div>

);

}

ReactDOM.render(

<App />,

document.getElementById('root')

);

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/vXdGmd?editors=0010)

Cada Clock configura su propio temporizador y se actualiza de forma independiente.

En las aplicaciones de React, si un componente tiene o no estado se considera un detalle de implementación del componente que puede cambiar con el tiempo. Puedes usar componentes sin estado dentro de componentes con estado y viceversa.

**Manejando eventos**

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

* Los eventos de React se nombran usando camelCase, en vez de minúsculas.
* Con JSX pasas una función como el manejador del evento, en vez de un string.

Por ejemplo, el HTML:

<button onclick="activateLasers()">

Activate Lasers

</button>

En React es algo diferente:

<button onClick={activateLasers}>

Activate Lasers

</button>

Otra diferencia es que en React no puedes retornar false para prevenir el comportamiento por defecto. Debes, explícitamente, llamar preventDefault. Por ejemplo, en un HTML plano, para prevenir el comportamiento por defecto de un enlace de abrir una nueva página, puedes escribir:

<a href="#" onclick="console.log('The link was clicked.'); return false">

Click me

</a>

En cambio en React, esto podría ser:

function ActionLink() {

function handleClick(e) {

e.preventDefault();

console.log('The link was clicked.');

}

return (

<a href="#" onClick={handleClick}>

Click me

</a>

);

}

Aquí, e es un evento sintético. React define estos eventos sintéticos acorde a las [especificaciones W3C](https://www.w3.org/TR/DOM-Level-3-Events/), entonces no debes preocuparte por la compatibilidad a tráves de los navegadores. Mira la guía de referencia [SyntheticEvent](https://es.reactjs.org/docs/events.html) para aprender más.

Cuando estás utilizando React, generalmente, no debes llamar addEventListener para agregar escuchadores de eventos a un elemento del DOM después de que este es creado. Por el contrario, solo debes proveer un manejador de eventos cuando el elemento es inicialmente renderizado.

Cuando defines un componente usando una [clase de ES6](https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Classes), un patrón muy común es que los manejadores de eventos sean un método de la clase. Por ejemplo, este componente Toggle renderiza un botón que permite al usuario cambiar el estado entre “ENCENDIDO” y “APAGADO”:

class Toggle extends React.Component {

constructor(props) {

super(props);

this.state = {isToggleOn: true};

// Este enlace es necesario para hacer que `this` funcione en el callback

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState(state => ({

isToggleOn: !state.isToggleOn

}));

}

render() {

return (

<button onClick={this.handleClick}>

{this.state.isToggleOn ? 'ON' : 'OFF'}

</button>

);

}

}

ReactDOM.render(

<Toggle />,

document.getElementById('root')

);

[**Pruébalo en CodePen**](https://codepen.io/gaearon/pen/xEmzGg?editors=0010)

Tienes que tener mucho cuidado en cuanto al significado de this en los callbacks de JSX. En JavaScript, los métodos de clase no están [ligados](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_objects/Function/bind) por defecto. Si olvidas ligar this.handleClick y lo pasas a onClick, this será undefined cuando se llame la función.

Esto no es un comportamiento especifico de React; esto hace parte de [como operan las funciones JavaScript](https://www.smashingmagazine.com/2014/01/understanding-javascript-function-prototype-bind/). Generalmente, si refieres un método sin usar () después de este, tal como onClick={this.handleClick}, deberías ligar ese método.

Si te molesta llamar bind, existen dos maneras de evitarlo. Si usas la sintaxis experimental [campos públicos de clases](https://babeljs.io/docs/plugins/transform-class-properties/), puedes usar los campos de clases para ligar los callbacks correctamente:

class LoggingButton extends React.Component {

// Esta sintaxis nos asegura que `this` está ligado dentro de handleClick

// Peligro: esto es una sintaxis \*experimental\*

handleClick = () => {

console.log('this is:', this);

}

render() {

return (

<button onClick={this.handleClick}>

Click me

</button>

);

}

}

Esta sintaxis está habilitada por defecto en [Create React App](https://github.com/facebookincubator/create-react-app).

Si no estas usando la sintaxis de campos públicos de clases, puedes usar una [función flecha](https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions/Arrow_functions) en el callback:

class LoggingButton extends React.Component {

handleClick() {

console.log('this is:', this);

}

render() {

// Esta sintaxis nos asegura que `this` esta ligado dentro de handleClick

return (

<button onClick={(e) => this.handleClick(e)}>

Click me

</button>

);

}

}

El problema con esta sintaxis es que un callback diferente es creado cada vez que LogginButton es renderizado. En la mayoría de los casos, esto está bien. Sin embargo, si este callback se pasa como una propiedad a componentes más bajos, estos componentes podrían renderizarse nuevamente. Generalmente, recomendamos ligar en el constructor o usar la sintaxis de campos de clases, para evitar esta clase de problemas de rendimiento.

**Pasando argumentos a escuchadores de eventos**

Dentro de un bucle es muy común querer pasar un parámetro extra a un manejador de eventos. Por ejemplo, si id es el ID de una fila, cualquiera de los códigos a continuación podría funcionar:

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

Las dos líneas anteriores son equivalentes, y utilizan [funciones flecha](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) y [Function.prototype.bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind) respectivamente.

En ambos casos, el argumento e que representa el evento de React va a ser pasado como un segundo argumento después del ID. Con una función flecha, tenemos que pasarlo explícitamente, pero con bind cualquier argumento adicional es pasado automáticamente.

**Git init**

git-init: crea un repositorio de Git vacío o reinicializa uno existente

**SINOPSIS**

\_git init\_ [-q | --quiet] [--bare] [--template = <template\_directory>] [--separate-git-dir <git dir>] [--shared [= <permissions>]] [directorio]

**DESCRIPCIÓN**

Este comando crea un repositorio vacío de Git - básicamente un .git directorio con subdirectorios para objects, refs/heads, refs/tags, y los archivos de plantilla. HEADTambién se crea un archivo inicial que hace referencia al HEAD de la rama maestra.

Si se establece la $GIT\_DIRvariable de entorno, especifica una ruta a utilizar en lugar de ./.gitpara la base del repositorio.

Si el directorio de almacenamiento de objetos se especifica a través de la $GIT\_OBJECT\_DIRECTORYvariable de entorno, los directorios sha1 se crean debajo; de lo contrario $GIT\_DIR/objects , se utiliza el directorio predeterminado .

Ejecutar *git init* en un repositorio existente es seguro. No sobrescribirá las cosas que ya están allí. La razón principal para volver a ejecutar *git init* es recoger plantillas recién agregadas (o mover el repositorio a otro lugar si se da --separate-git-dir).

**EJEMPLOS**

Inicie un nuevo repositorio Git para una base de código existente

$ cd / ruta / a / my / codebase

$ git init \*\*(1)\*\*

$ git add. \*\*(2)\*\*

$ git commit \*\*(3)\*\*

1. Cree un directorio /path/to/my/codebase/.git.
2. Agregue todos los archivos existentes al índice.
3. Registre el estado prístino como el primer compromiso en la historia.

[**npm-init**](https://docs.npmjs.com/cli/init.html)

Crea un archivo package.json

[**SINOPSIS**](https://docs.npmjs.com/cli/init#synopsis)

npm init [--force|-f|--yes|-y|--scope]

npm init <@scope> (same as `npx <@scope>/create`)

npm init [<@scope>/]<name> (same as `npx [<@scope>/]create-<name>`)

[**EJEMPLOS**](https://docs.npmjs.com/cli/init#examples)

Cree un nuevo proyecto basado en React usando [create-react-app](https://npm.im/create-react-app):

$ npm init react-app ./my-react-app

Cree un nuevo esmpaquete compatible con [create-esm](https://npm.im/create-esm):

$ mkdir my-esm-lib && cd my-esm-lib

$ npm init esm --yes

Genere un paquete antiguo simple.json usando init heredado:

$ mkdir my-npm-pkg && cd my-npm-pkg

$ git init

$ npm init

Generarlo sin tener que hacer preguntas:

$ npm init -y

[**DESCRIPCIÓN**](https://docs.npmjs.com/cli/init#description)

npm init <initializer> se puede usar para configurar un paquete npm nuevo o existente.

initializeren este caso, se llama un paquete npm create-<initializer>, que será instalado por , y luego se ejecutará su bin principal, presumiblemente creando o actualizando y ejecutando cualquier otra operación relacionada con la inicialización.[npx](https://docs.npmjs.com/cli/npx)``package.json

El comando init se transforma en una npxoperación correspondiente de la siguiente manera:

* npm init foo -> npx create-foo
* npm init @usr/foo -> npx @usr/create-foo
* npm init @usr -> npx @usr/create

Cualquier opción adicional se pasará directamente al comando, por lo que se npm init foo --helloasignará a npx create-foo --hello.

Si se omite el inicializador (simplemente llamando npm init), init recurrirá al comportamiento init heredado. Le hará un montón de preguntas y luego escribirá un package.json para usted. Intentará hacer conjeturas razonables basadas en los campos existentes, las dependencias y las opciones seleccionadas. Es estrictamente aditivo, por lo que mantendrá todos los campos y valores que ya se hayan establecido. También puede usar -y/ --yespara omitir el cuestionario por completo. Si aprueba --scope, creará un paquete con ámbito.

**Estructura de directorios estandar**



**Babel (compilador)**

**Babel** es un [compilador de](https://en.wikipedia.org/wiki/Compiler) [JavaScript](https://en.wikipedia.org/wiki/JavaScript) [gratuito](https://en.wikipedia.org/wiki/Free_software) y [de código](https://en.wikipedia.org/wiki/Open-source_software) [abierto](https://en.wikipedia.org/wiki/JavaScript) que se utiliza principalmente para convertir el [código ECMAScript 2015+](https://en.wikipedia.org/wiki/ECMAScript#6th_Edition_-_ECMAScript_2015) (ES6 +) en una versión de [JavaScript](https://en.wikipedia.org/wiki/JavaScript) compatible con versiones anteriores que pueden ejecutar los [motores JavaScript](https://en.wikipedia.org/wiki/JavaScript_engine) más antiguos . Babel es una herramienta popular para usar las funciones más nuevas del lenguaje de programación JavaScript. [[3]](<https://en.wikipedia.org/wiki/Babel_(compiler)#cite_note-3>)

**Instalación**

Usaremos las siguientes dependencias:

// package.json

{

"name": "react-curso",

"version": "1.0.0",

"description": "",

"main": "index.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "ISC",

"dependencies": {

"@babel/core": "^7.6.0",

"@babel/preset-env": "^7.6.0",

"@babel/preset-react": "^7.0.0",

"babel-loader": "^8.0.6",

"react": "^16.9.0",

"react-dom": "^16.9.0"

}

}

Usamos npm install para instalar todo esto.

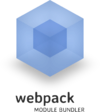
Luego configuramos .babelrc para implementar el uso de @babel/preset-env que es para usar js moderno y @babel/preset-react que es para usarlo junto con React.

{

"presets": ["@babel/preset-env", "@babel/preset-react"]

}

**Webpack**



Webpack.png

**Webpack** (estilizado **webpack** ) es un [código abierto](https://en.wikipedia.org/wiki/Open-source_software) [de JavaScript](https://en.wikipedia.org/wiki/JavaScript) módulo bundler. [[2]](https://en.wikipedia.org/wiki/Webpack#cite_note-2) [[3]](https://en.wikipedia.org/wiki/Webpack#cite_note-3) [[4]](https://en.wikipedia.org/wiki/Webpack#cite_note-4) [[5]](https://en.wikipedia.org/wiki/Webpack#cite_note-5) [[6]](https://en.wikipedia.org/wiki/Webpack#cite_note-6) Es un paquete de módulos principalmente para JavaScript, pero puede transformar activos front-end como HTML, CSS e imágenes si se incluyen los complementos correspondientes. [[7]](https://en.wikipedia.org/wiki/Webpack#cite_note-7) Webpack toma módulos con dependencias y genera activos estáticos que representan esos módulos. [[8]](https://en.wikipedia.org/wiki/Webpack#cite_note-8)

**Dependencias a usar**

"devDependencies": {

"html-loader": "^0.5.5",

"html-webpack-plugin": "^3.2.0",

"webpack-cli": "^3.3.9"

}

Vamos a configurar **webpack** con el archivo webpack.config.js en la raíz de nuestro proyecto.

const path = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

entry: "./src/index.js",

output: {

path: path.resolve(\_\_dirname, "dist"),

filename: "bundle.js"

},

resolve: {

extensions: [".js", ".jsx"]

},

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_module/,

use: {

loader: "babel-loader"

}

},

{

test: /\.html$/,

use: [

{

loader: "html-loader"

}

]

}

]

},

plugins: [

new HtmlWebpackPlugin({

template: './public/index.html',

filename: './index.html'

})

]

};

Configuramos en nuestro package.json el siguiente *script*:

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1",

"build": "webpack --mode production"

}

Corremos nuestro copilador con npm run build para que se aloje el la carpeta dist.

**webpack-dev-server**

Use [webpack](https://webpack.js.org/) con un servidor de desarrollo que proporcione recarga en vivo. Esto debe usarse **solo** para el **desarrollo** .

Utiliza [webpack-dev-middleware](https://github.com/webpack/webpack-dev-middleware) debajo del capó, que proporciona acceso rápido en memoria a los activos del paquete web.

**Empezando**

Primero lo primero, instale el módulo:

npm install webpack-dev-server --save-dev

*Nota: Si bien puede instalar y ejecutar webpack-dev-server a nivel mundial, recomendamos instalarlo localmente. webpack-dev-server siempre usará una instalación local sobre una global.*

**Uso**

Hay dos métodos principales recomendados para usar el módulo:

**Con la CLI**

La forma más fácil de usarlo es con la CLI. En el directorio donde webpack.config.jsestá, ejecute:

node\_modules / .bin / webpack-dev-server

***Nota*** *: Muchas opciones de CLI están disponibles con webpack-dev-server. Explore este* [*enlace*](https://webpack.js.org/configuration/dev-server/) *.*

**Con scripts NPM**

Los scripts package.json de NPM son un medio conveniente y útil para ejecutar binarios instalados localmente sin tener que preocuparse por sus rutas completas. Simplemente defina un script como tal:

" scripts " : { " start: dev " : " webpack-dev-server " }

Y ejecute lo siguiente en su terminal / consola:

npm run start: dev

NPM hará referencia automática al binario node\_modulespara usted y ejecutará el archivo o comando.

**El resultado**

Cualquiera de los métodos iniciará una instancia del servidor y comenzará a escuchar las conexiones desde localhostel puerto 8080.

webpack-dev-server está configurado de manera predeterminada para admitir la recarga en vivo de archivos mientras edita sus activos mientras el servidor se está ejecutando.

Consulte [**la documentación**](https://webpack.js.org/configuration/dev-server/#devserver) para obtener más casos de uso y opciones.

**sass-loader**

Carga un archivo Sass / SCSS y lo compila en CSS.

**Empezando**

Para comenzar, deberá instalar sass-loader:

npm install sass-loader nodo-sass webpack --save-dev

El sass-loader requiere que instales [Node Sass](https://github.com/sass/node-sass) o [Dart Sass](https://github.com/sass/dart-sass) por tu cuenta (puedes encontrar más documentación a continuación). Esto le permite controlar las versiones de todas sus dependencias y elegir qué implementación de Sass usar.

* [node sass](https://github.com/sass/node-sass)
* [dat sass](http://sass-lang.com/dart-sass)

Encadene el sass-loader con el [css-loader](https://github.com/webpack-contrib/css-loader) y el [style-loader](https://github.com/webpack-contrib/style-loader) para aplicar inmediatamente todos los estilos al DOM o al [mini-css-extract-plugin](https://github.com/webpack-contrib/mini-css-extract-plugin) para extraerlo en un archivo separado.

Luego agregue el cargador a su webpackconfiguración. Por ejemplo:

**file.js**

import style from './style.scss';

**file.scss**

$body-color: red;

body {

color: $body-color;

}

**webpack.config.js**

module.exports = {

module: {

rules: [

{

test: /\.s[ac]ss$/i,

use: [

// Creates `style` nodes from JS strings

'style-loader',

// Translates CSS into CommonJS

'css-loader',

// Compiles Sass to CSS

'sass-loader',

],

},

],

},

};

Y corre a webpacktravés de tu método preferido.