
REACT
www.postparaprogramadores.com

Contenido

ReactJS - Descripción general

ReactJS es una biblioteca de JavaScript utilizada para construir componentes de IU reutilizables. De acuerdo con la documentación oficial de React, la siguiente es la definición:

React es una biblioteca para construir interfaces de usuario componibles. Fomenta la creación de componentes de IU reutilizables, que presentan datos que cambian con el tiempo. Mucha gente usa React como V en MVC. Reaccione los extractos del DOM de usted, ofreciendo un modelo de programación más simple y un mejor rendimiento. React también puede renderizar en el servidor usando Node, y puede alimentar aplicaciones nativas usando React Native. React implementa un flujo de datos reactivo unidireccional, lo que reduce la resistencia y es más fácil razonar que el enlace de datos tradicional.

Características de reacción

- **JSX** - JSX es una extensión de sintaxis de JavaScript. No es necesario usar JSX en el desarrollo de React, pero se recomienda.
- **Componentes** : React se trata de componentes. Debe pensar en todo como un componente. Esto lo ayudará a mantener el código cuando trabaje en proyectos de mayor escala.
- **Flujo de datos unidireccional y flujo** : React implementa un **flujo de datos unidireccional** que facilita razonar sobre su aplicación. Flux es un patrón que ayuda a mantener sus datos unidireccionales.
- **Licencia** : React tiene licencia de Facebook Inc. La documentación tiene licencia de CC BY 4.0.

Ventajas de reacción

- Utiliza DOM virtual que es un objeto de JavaScript. Esto mejorará el rendimiento de las aplicaciones, ya que el DOM virtual de JavaScript es más rápido que el DOM normal.
- Se puede usar en el lado del cliente y del servidor, así como con otros marcos.
- Los patrones de componentes y datos mejoran la legibilidad, lo que ayuda a mantener aplicaciones más grandes.

Limitaciones de reacción

- Cubre solo la capa de vista de la aplicación, por lo tanto, aún debe elegir otras tecnologías para obtener un conjunto completo de herramientas para el desarrollo.
- Utiliza plantillas en línea y JSX, lo que puede parecer incómodo para algunos desarrolladores.

ReactJS - Configuración del entorno

En este capítulo, le mostraremos cómo configurar un entorno para el desarrollo exitoso de React. Tenga en cuenta que hay muchos pasos involucrados, pero esto ayudará a acelerar el proceso de desarrollo más adelante. Necesitaremos **NodeJS** , por lo que si no lo tiene instalado, consulte el enlace de la siguiente tabla.

No Señor.	Software y descripción
1	**NodeJS y NPM** NodeJS es la plataforma necesaria para el desarrollo de ReactJS. Consulte nuestra configuración del entorno NodeJS .

Después de instalar con éxito NodeJS, podemos comenzar a instalar React con npm. Puede instalar ReactJS de dos maneras

- Usando webpack y babel.
- Usando el comando **create-react-app** .

Instalación de ReactJS usando webpack y babel

Webpack es un paquete de módulos (gestiona y carga módulos independientes). Toma módulos dependientes y los compila en un solo paquete (archivo). Puede usar este paquete mientras desarrolla aplicaciones usando la línea de comandos o configurándolo usando el archivo webpack.config.

Babel es un compilador y transpilador de JavaScript. Se utiliza para convertir un código fuente a otro. Con esto, podrá usar las nuevas funciones de ES6 en su código, donde babel lo convierte en un viejo ES5 que se puede ejecutar en todos los navegadores.

Paso 1: crear la carpeta raíz

Cree una carpeta con el nombre **reactApp** en el escritorio para instalar todos los archivos necesarios, utilizando el comando mkdir.

```
C:\Users\username\Desktop>mkdir reactApp
C:\Users\username\Desktop>cd reactApp
```

Para crear cualquier módulo, es necesario generar el archivo **package.json** . Por lo tanto, después de crear la carpeta, necesitamos crear un archivo **package.json** . Para hacerlo, debe ejecutar el comando **npm init** desde el símbolo del sistema.

```
C:\Users\username\Desktop\reactApp>npm init
```

Este comando solicita información sobre el módulo, como el nombre del paquete, la descripción, el autor, etc. Puede omitirlos utilizando la opción –y.

```
C:\Users\username\Desktop\reactApp>npm init -y
Wrote to C:\reactApp\package.json:
{
   "name": "reactApp",
   "version": "1.0.0",
   "description": "",
   "main": "index.js",
   "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
   },
   "keywords": [],
   "author": "",
   "license": "ISC"
}
```

Paso 2 - instala React y react dom

Dado que nuestra tarea principal es instalar ReactJS, instálelo y sus paquetes dom, usando los comandos **install** **react** y **react-dom** de npm

respectivamente. Puede agregar los paquetes que instalamos al archivo **package.json** usando la opción **--save** .

```
C:\Users\Tutorialspoint\Desktop\reactApp>npm install react --
save
C:\Users\Tutorialspoint\Desktop\reactApp>npm install react-
dom --save
```

O bien, puede instalarlos todos en un solo comando como:

```
C:\Users\username\Desktop\reactApp>npm install react react-
dom --save
```

Paso 3 - Instalar webpack

Dado que estamos usando webpack para generar el paquete de instalación de webpack, webpack-dev-server y webpack-cli.

```
C:\Users\username\Desktop\reactApp>npm install webpack --save
C:\Users\username\Desktop\reactApp>npm install webpack-dev-
server --save
C:\Users\username\Desktop\reactApp>npm install webpack-cli --
save
```

O bien, puede instalarlos todos en un solo comando como:

```
C:\Users\username\Desktop\reactApp>npm install webpack
webpack-dev-server webpack-cli --save
```

Paso 4 - Instala babel

Instale babel y sus complementos babel-core, babel-loader, babel-preset-env, babel-preset-react y, html-webpack-plugin

```
C:\Users\username\Desktop\reactApp>npm install babel-core --
save-dev
C:\Users\username\Desktop\reactApp>npm install babel-loader -
-save-dev
C:\Users\username\Desktop\reactApp>npm install babel-preset-
env --save-dev
C:\Users\username\Desktop\reactApp>npm install babel-preset-
react --save-dev
C:\Users\username\Desktop\reactApp>npm install html-webpack-
plugin --save-dev
```

O bien, puede instalarlos todos en un solo comando como:

```
C:\Users\username\Desktop\reactApp>npm install babel-core
babel-loader babel-preset-env
   babel-preset-react html-webpack-plugin --save-dev
```

Paso 5 - Crea los archivos

Para completar la instalación, necesitamos crear ciertos archivos, a saber, index.html, App.js, main.js, webpack.config.js y,. ***babelrc*** . Puede crear estos archivos manualmente o mediante el **símbolo del sistema** .

```
C:\Users\username\Desktop\reactApp>type nul > index.html
C:\Users\username\Desktop\reactApp>type nul > App.js
C:\Users\username\Desktop\reactApp>type nul > main.js
C:\Users\username\Desktop\reactApp>type nul >
webpack.config.js
C:\Users\username\Desktop\reactApp>type nul > .babelrc
```

Paso 6 - Establecer compilador, servidor y cargadores

Abra el archivo **webpack-config.js** y agregue el siguiente código. Estamos configurando el punto de entrada del paquete web para que sea main.js. La ruta de salida es el lugar donde se servirá la aplicación incluida. También estamos configurando el servidor de desarrollo en el puerto **8001** . Puedes elegir el puerto que quieras.

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
   entry: './main.js',
   output: {
      path: path.join(__dirname, '/bundle'),
      filename: 'index_bundle.js'
   },
   devServer: {
      inline: true,
      port: 8001
   },
   module: {
      rules: [
         {
            test: /\.jsx?$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            query: {
               presets: ['es2015', 'react']
            }
         }
      ]
   },
   plugins:[
      new HtmlWebpackPlugin({
         template: './index.html'
      })
   ]
}
```

Abra la **package.json** y borrar **"prueba" "echo \" Error: ninguna prueba especifica \" && salida 1"** interior **'guiones'** objeto. Estamos eliminando esta línea ya que no haremos ninguna prueba en este tutorial. Agreguemos los comandos de **inicio** y **compilación en su** lugar.

```
"start": "webpack-dev-server --mode development --open --
hot",
"build": "webpack --mode production"
```

Paso 7 - index.html

Esto es solo HTML normal. Estamos configurando **div id = "app"** como elemento raíz de nuestra aplicación y agregando script **index_bundle.js** , que es nuestro archivo de aplicación incluido.

```
<!DOCTYPE html>
<html lang = "en">
   <head>
      <meta charset = "UTF-8">
      <title>React App</title>
   </head>
   <body>
      <div id = "app"></div>
      <script src = 'index_bundle.js'></script>
   </body>
</html>
```

Paso 8: App.jsx y main.js

Este es el primer componente React. Explicaremos los componentes de React en profundidad en un capítulo posterior. Este componente representará **Hello World** .

App.js

```
import React, { Component } from 'react';
class App extends Component{
   render(){
      return(
         <div>
            <h1>Hello World</h1>
         </div>
      );
   }
}
export default App;
```

Necesitamos importar este componente y representarlo en nuestro elemento de **aplicación** raíz , para que podamos verlo en el navegador.

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

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

Nota : siempre que desee utilizar algo, primero debe **importarlo** . Si desea que el componente se pueda utilizar en otras partes de la aplicación, debe **exportarlo** después de la creación e importarlo en el archivo donde desea usarlo.

Cree un archivo con el nombre **.babelrc** y copie el siguiente contenido.

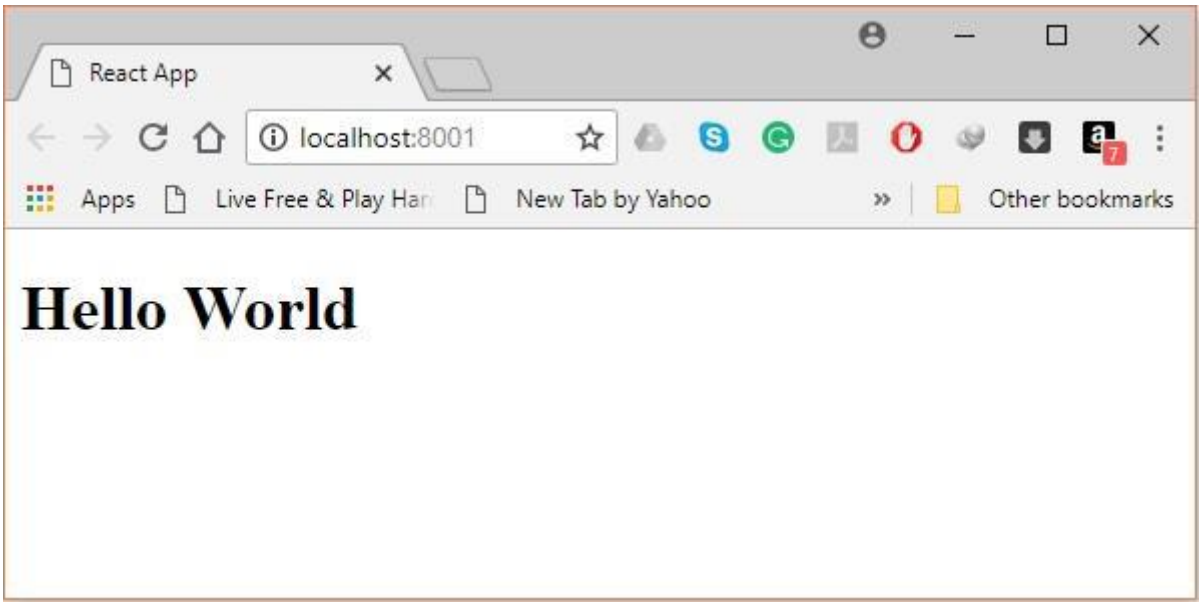
```
{
   "presets":["env", "react"]
}
```

Paso 9: ejecutar el servidor

La configuración está completa y podemos iniciar el servidor ejecutando el siguiente comando.

```
C:\Users\username\Desktop\reactApp>npm start
```

Mostrará el puerto que necesitamos abrir en el navegador. En nuestro caso, es **http: // localhost: 8001 /** . Después de abrirlo, veremos el siguiente resultado.

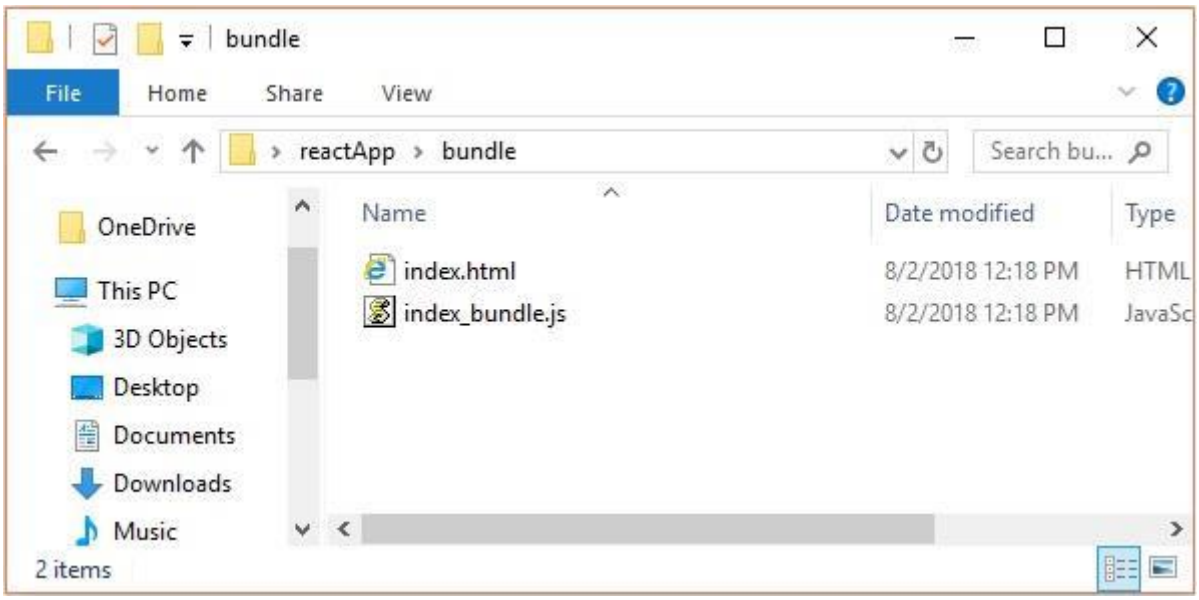


Paso 10 - Generando el paquete

Finalmente, para generar el paquete, debe ejecutar el comando de compilación en el símbolo del sistema como:

```
C:\Users\Tutorialspoint\Desktop\reactApp>npm run build
```

Esto generará el paquete en la carpeta actual como se muestra a continuación.



Usando el comando create-react-app

En lugar de usar webpack y babel, puede instalar ReactJS más simplemente instalando **create-react-app** .

Paso 1 - instala create-react-app

Navegue por el escritorio e instale la aplicación Create React usando el símbolo del sistema como se muestra a continuación:

```
C:\Users\Tutorialspoint>cd C:\Users\Tutorialspoint\Desktop\
C:\Users\Tutorialspoint\Desktop>npx create-react-app my-app
```

Esto creará una carpeta llamada my-app en el escritorio e instalará todos los archivos necesarios.

Paso 2: elimina todos los archivos de origen

Navegue a través de la carpeta src en la carpeta my-app generada y elimine todos los archivos en ella como se muestra a continuación:

```
C:\Users\Tutorialspoint\Desktop>cd my-app/src
C:\Users\Tutorialspoint\Desktop\my-app\src>del *
C:\Users\Tutorialspoint\Desktop\my-app\src\*, Are you sure
(Y/N)? y
```

Paso 3 - Agregar archivos

Agregue archivos con los nombres **index.css** e **index.js** en la carpeta src como -

```
C:\Users\Tutorialspoint\Desktop\my-app\src>type nul >
index.css
C:\Users\Tutorialspoint\Desktop\my-app\src>type nul >
index.js
```

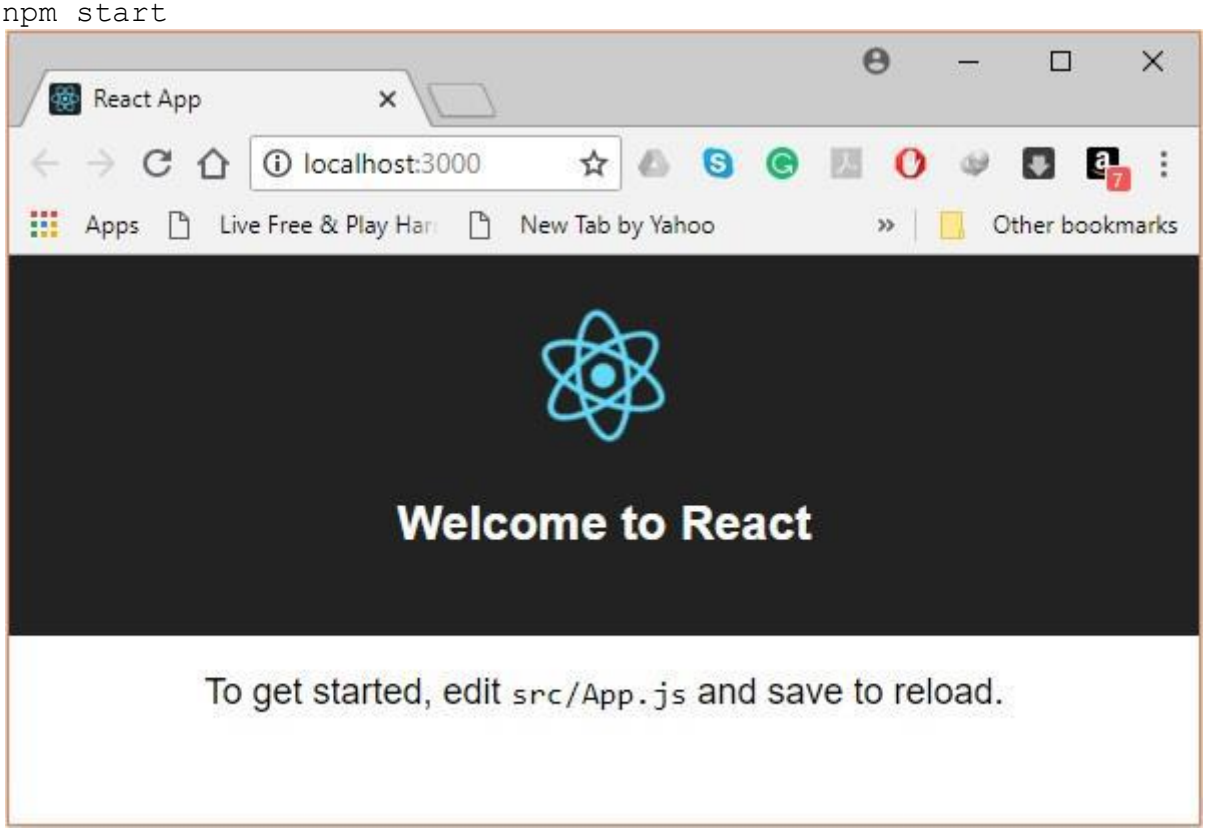
En el archivo index.js agregue el siguiente código

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
```

Paso 4: ejecuta el proyecto

Finalmente, ejecute el proyecto usando el comando de inicio.

```
npm start
```



ReactJS - JSX

React usa JSX para crear plantillas en lugar de JavaScript normal. No es necesario usarlo, sin embargo, a continuación hay algunos pros que vienen con él.

- Es más rápido porque realiza la optimización al compilar código en JavaScript.
- También es de tipo seguro y la mayoría de los errores pueden detectarse durante la compilación.
- Hace que sea más fácil y rápido escribir plantillas, si está familiarizado con HTML.

Usando JSX

JSX parece un HTML normal en la mayoría de los casos. Ya lo usamos en el capítulo Configuración del entorno. Mire el código de **App.jsx** donde **devolvemos div** .

App.jsx

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            Hello World!!!
         </div>
      );
   }
}
export default App;
```

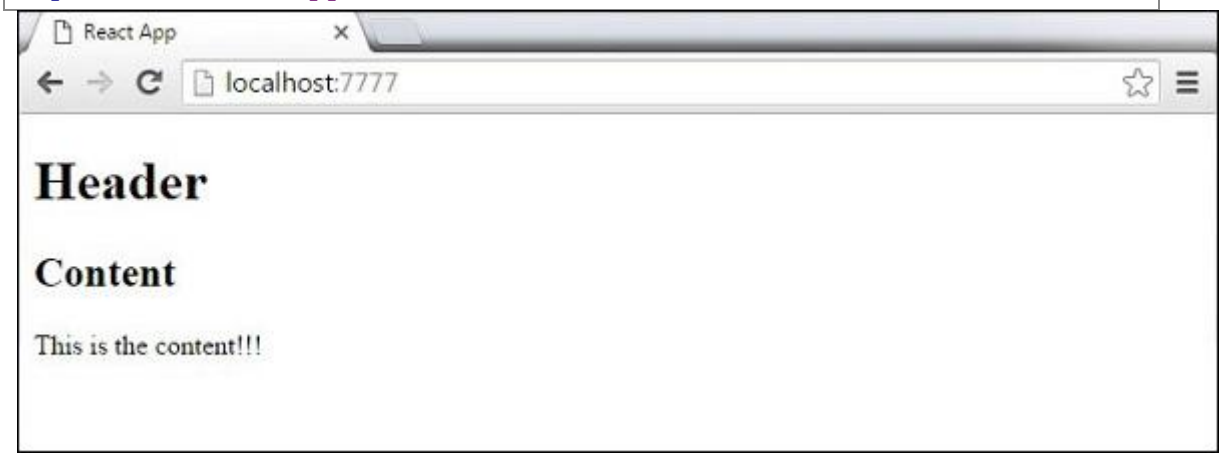
Aunque es similar al HTML, hay algunas cosas que debemos tener en cuenta al trabajar con JSX.

Elementos anidados

Si queremos devolver más elementos, necesitamos envolverlo con un elemento contenedor. Observe cómo estamos usando **div** como envoltorio para los elementos **h1** , **h2** y **p** .

App.jsx

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <h1>Header</h1>
            <h2>Content</h2>
            <p>This is the content!!!</p>
         </div>
      );
   }
}
export default App;
```



Atributos

Podemos usar nuestros propios atributos personalizados además de las propiedades y atributos HTML normales. Cuando queremos agregar un atributo personalizado, necesitamos usar el prefijo de **datos** . En el siguiente ejemplo, agregamos **data-myattribute** como un atributo del elemento **p** .

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <h1>Header</h1>
            <h2>Content</h2>
            <p data-myattribute = "somevalue">This is the
content!!!</p>
         </div>
      );
   }
}
export default App;
```

Expresiones JavaScript

Las expresiones de JavaScript se pueden usar dentro de JSX. Solo necesitamos envolverlo con llaves **{}** . El siguiente ejemplo representará **2** .

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <h1>{1+1}</h1>
         </div>
      );
   }
}
export default App;
```

No podemos usar sentencias **if else** dentro de JSX, en su lugar podemos usar expresiones **condicionales (ternarias)** . En el siguiente ejemplo, la variable **i** es igual a **1,** por lo que el navegador se convertirá en **verdadero** . Si lo cambiamos a otro valor, se convertirá en **falso** .

```
import React from 'react';

class App extends React.Component {
   render() {
      var i = 1;
      return (
         <div>
            <h1>{i == 1 ? 'True!' : 'False'}</h1>
         </div>
      );
   }
}
export default App;
```

Estilo

React recomienda usar estilos en línea. Cuando queremos establecer estilos en línea, necesitamos usar la sintaxis **camelCase** . React también agregará **px** automáticamente después del valor numérico en elementos específicos. El siguiente ejemplo muestra cómo agregar **myStyle en** línea al elemento **h1** .

```
import React from 'react';

class App extends React.Component {
   render() {
      var myStyle = {
         fontSize: 100,
         color: '#FF0000'
      }
      return (
         <div>
            <h1 style = {myStyle}>Header</h1>
         </div>
      );
   }
}
```

```
export default App;
```

Comentarios

Cuando escribimos comentarios, necesitamos poner llaves **{}** cuando queremos escribir comentarios dentro de la sección secundaria de una etiqueta. Es una buena práctica usar siempre **{}** al escribir comentarios, ya que queremos ser coherentes al escribir la aplicación.

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <h1>Header</h1>
            {//End of the line Comment...}
            {/*Multi line comment...*/}
         </div>
      );
   }
}
export default App;
```

Convenio de denominación

Las etiquetas HTML siempre usan nombres de etiquetas en **minúsculas** , mientras que los componentes React comienzan con **mayúsculas** .

Nota : debe usar **className** y **htmlFor** como nombres de atributos XML en lugar de **class** y **for** .

Esto se explica en la página oficial de React como:

Como JSX es JavaScript, los identificadores como **class** y **for** **no** se recomiendan como nombres de atributos XML. En cambio, los componentes React DOM esperan nombres de propiedad DOM como **className** y **htmlFor** , respectivamente.

En este capítulo, aprenderemos cómo combinar componentes para que la aplicación sea más fácil de mantener. Este enfoque permite actualizar y cambiar sus componentes sin afectar el resto de la página.

Ejemplo sin estado

Nuestro primer componente en el siguiente ejemplo es la **aplicación** . Este componente es propietario de **Encabezado** y **Contenido** . Estamos creando **encabezado** y **contenido por** separado y solo lo agregamos dentro del árbol JSX en nuestro componente de **aplicación** . Solo se debe exportar el componente de la **aplicación** .

App.jsx

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <Header/>
            <Content/>
         </div>
      );
   }
}
class Header extends React.Component {
   render() {
      return (
         <div>
            <h1>Header</h1>
         </div>
      );
   }
}
class Content extends React.Component {
   render() {
      return (
         <div>
            <h2>Content</h2>
            <p>The content text!!!</p>
         </div>
      );
   }
}
export default App;
```

Para poder representar esto en la página, necesitamos importarlo en el archivo **main.js** y llamar a **reactDOM.render ()** . Ya hicimos esto mientras configuramos el entorno.

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

El código anterior generará el siguiente resultado.

Ejemplo con estado

En este ejemplo, estableceremos el estado del componente propietario (**Aplicación**). El componente **Encabezado** se acaba de agregar como en el último ejemplo, ya que no necesita ningún estado. En lugar de etiqueta de contenido, estamos creando elementos de **tabla** y **cuerpo** , donde insertaremos dinámicamente **TableRow** para cada objeto de la matriz de **datos** .

Se puede ver que estamos usando la sintaxis de flecha EcmaScript 2015 (=>) que se ve mucho más limpia que la sintaxis JavaScript anterior. Esto nos ayudará a crear nuestros elementos con menos líneas de código. Es especialmente útil cuando necesitamos crear una lista con muchos elementos.

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor() {
      super();
      this.state = {
         data:
         [
            {
               "id":1,
               "name":"Foo",
               "age":"20"
            },
```

```
            {
               "id":2,
               "name":"Bar",
               "age":"30"
            },
            {
               "id":3,
               "name":"Baz",
               "age":"40"
            }
         ]
      }
   }
   render() {
      return (
         <div>
            <Header/>
            <table>
               <tbody>
                  {this.state.data.map((person, i) =>
<TableRow key = {i}
                     data = {person} />)}
               </tbody>
            </table>
         </div>
      );
   }
}
class Header extends React.Component {
   render() {
      return (
         <div>
            <h1>Header</h1>
         </div>
      );
   }
}
class TableRow extends React.Component {
   render() {
      return (
         <tr>
            <td>{this.props.data.id}</td>
            <td>{this.props.data.name}</td>
            <td>{this.props.data.age}</td>
         </tr>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Nota : tenga en cuenta que estamos usando la **tecla = {i} dentro de la función map ()** . Esto ayudará a React a actualizar solo los elementos necesarios en lugar de volver a representar la lista completa cuando algo cambie. Es un gran aumento de rendimiento para un mayor número de elementos creados dinámicamente.

ReactJS - Estado

El estado es el lugar de donde provienen los datos. Siempre debemos tratar de hacer que nuestro estado sea lo más simple posible y minimizar la cantidad de componentes con estado. Si tenemos, por ejemplo, diez componentes que necesitan datos del estado, deberíamos crear un componente contenedor que mantenga el estado de todos ellos.

Usando estado

El siguiente código de muestra muestra cómo crear un componente con estado utilizando la sintaxis EcmaScript2016.

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor(props) {
      super(props);

      this.state = {
         header: "Header from state...",
         content: "Content from state..."
      }
   }
```

```
    render() {
       return (
          <div>
             <h1>{this.state.header}</h1>
             <h2>{this.state.content}</h2>
          </div>
       );
    }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Esto producirá el siguiente resultado.

ReactJS - Resumen de accesorios

La principal diferencia entre estado y accesorios es que los **accesorios** son inmutables. Esta es la razón por la cual el componente contenedor debe definir el estado que se puede actualizar y cambiar, mientras que los componentes secundarios solo deben pasar datos del estado mediante accesorios.

Usando accesorios

Cuando necesitamos datos inmutables en nuestro componente, podemos agregar accesorios a la función **reactDOM.render ()** en **main.js** y usarlos dentro de nuestro componente.

App.jsx

```
import React from 'react';
```

```
class App extends React.Component {
   render() {
      return (
         <div>
            <h1>{this.props.headerProp}</h1>
            <h2>{this.props.contentProp}</h2>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

ReactDOM.render(<App headerProp = "Header from props..."
contentProp = "Content
   from props..."/>, document.getElementById('app'));

export default App;
```

Esto producirá el siguiente resultado.

Atrezzo predeterminado

También puede establecer valores de propiedad predeterminados directamente en el constructor del componente en lugar de agregarlo al elemento **reactDom.render ()** .

App.jsx

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
```

```
            <div>
               <h1>{this.props.headerProp}</h1>
               <h2>{this.props.contentProp}</h2>
            </div>
         );
      }
}
App.defaultProps = {
   headerProp: "Header from props...",
   contentProp:"Content from props..."
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

La salida es la misma que antes.

Estado y accesorios

El siguiente ejemplo muestra cómo combinar **estado** y accesorios en su aplicación. Estamos configurando el estado en nuestro componente principal y pasándolo al árbol de componentes usando **accesorios** . En el interior del **render** función, estamos estableciendo **headerProp** y **contentProp** utilizado en componentes hijos.

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor(props) {
      super(props);
      this.state = {
```

```
            header: "Header from props...",
            content: "Content from props..."
        }
    }
    render() {
        return (
            <div>
                <Header headerProp = {this.state.header}/>
                <Content contentProp = {this.state.content}/>
            </div>
        );
    }
}
class Header extends React.Component {
    render() {
        return (
            <div>
                <h1>{this.props.headerProp}</h1>
            </div>
        );
    }
}
class Content extends React.Component {
    render() {
        return (
            <div>
                <h2>{this.props.contentProp}</h2>
            </div>
        );
    }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

El resultado será nuevamente el mismo que en los dos ejemplos anteriores, lo único que es diferente es la fuente de nuestros datos, que ahora provienen originalmente del **estado** . Cuando queremos actualizarlo, solo necesitamos actualizar el estado, y todos los componentes secundarios se actualizarán. Más sobre esto en el capítulo Eventos.

ReactJS - Validación de accesorios

La validación de propiedades es una forma útil de forzar el uso correcto de los componentes. Esto ayudará durante el desarrollo para evitar errores y problemas futuros, una vez que la aplicación se haga más grande. También hace que el código sea más legible, ya que podemos ver cómo se debe usar cada componente.

Validar accesorios

En este ejemplo, estamos creando un componente de **aplicación** con todos los **accesorios** que necesitamos. **App.propTypes** se usa para la validación de accesorios. Si algunos de los accesorios no usan el tipo correcto que asignamos, recibiremos una advertencia de la consola. Después de especificar los patrones de validación, configuraremos **App.defaultProps** .

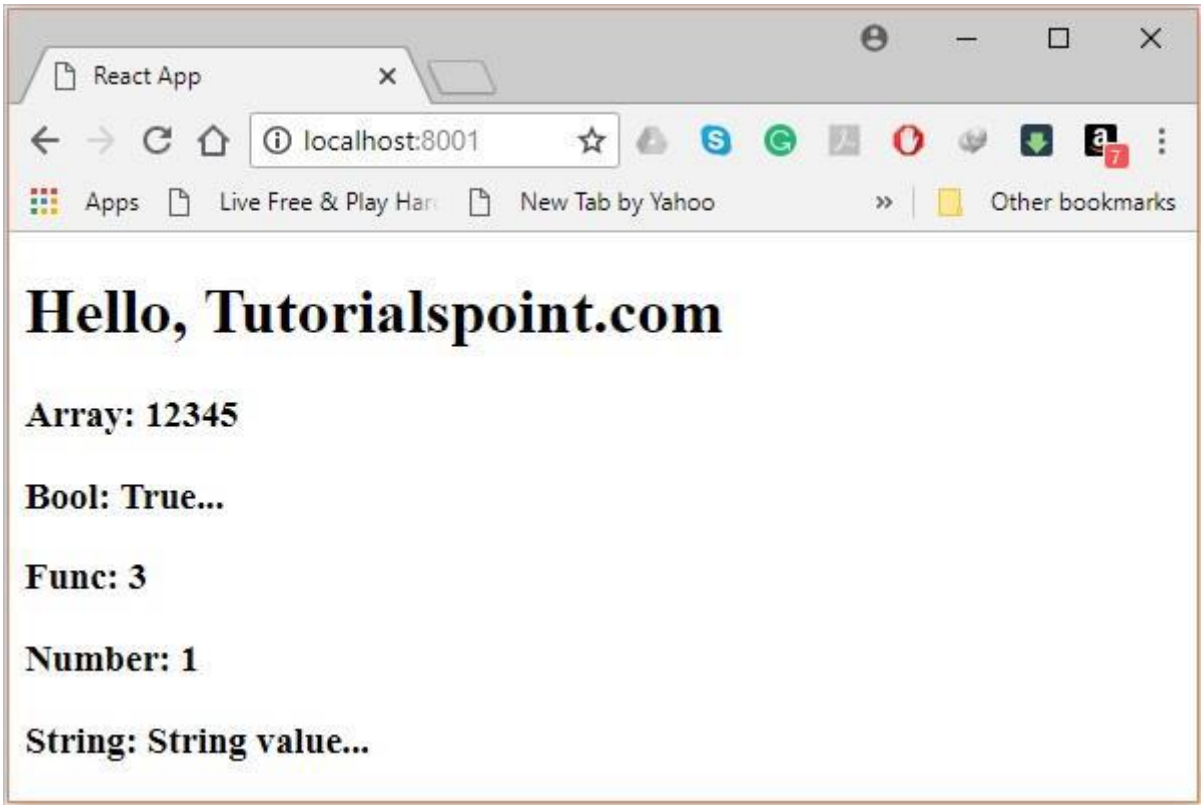
App.jsx

```
import React from 'react';

class App extends React.Component {
   render() {
      return (
         <div>
            <h3>Array: {this.props.propArray}</h3>
            <h3>Bool: {this.props.propBool ? "True..." :
"False..."}</h3>
            <h3>Func: {this.props.propFunc(3)}</h3>
            <h3>Number: {this.props.propNumber}</h3>
            <h3>String: {this.props.propString}</h3>
            <h3>Object:
{this.props.propObject.objectName1}</h3>
            <h3>Object:
{this.props.propObject.objectName2}</h3>
            <h3>Object:
{this.props.propObject.objectName3}</h3>
         </div>
      );
   }
```

```
}

App.propTypes = {
   propArray: React.PropTypes.array.isRequired,
   propBool: React.PropTypes.bool.isRequired,
   propFunc: React.PropTypes.func,
   propNumber: React.PropTypes.number,
   propString: React.PropTypes.string,
   propObject: React.PropTypes.object
}

App.defaultProps = {
   propArray: [1,2,3,4,5],
   propBool: true,
   propFunc: function(e){return e},
   propNumber: 1,
   propString: "String value...",

   propObject: {
      objectName1:"objectValue1",
      objectName2: "objectValue2",
      objectName3: "objectValue3"
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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



ReactJS - API de componentes

En este capítulo, explicaremos la API del componente React. Discutiremos tres métodos: **setState (), forceUpdate** y **ReactDOM.findDOMNode ()** . En las nuevas clases de ES6, tenemos que vincular esto manualmente. Usaremos **this.method.bind (this)** en los ejemplos.

Establecer estado

El método **setState ()** se utiliza para actualizar el estado del componente. Este método no reemplazará el estado, sino que solo agregará cambios al estado original.

```
import React from 'react';

class App extends React.Component {
   constructor() {
      super();

      this.state = {
         data: []
      }

      this.setStateHandler = this.setStateHandler.bind(this);
   };
   setStateHandler() {
      var item = "setState..."
      var myArray = this.state.data.slice();
         myArray.push(item);
```

```
      this.setState({data: myArray})
   };
   render() {
      return (
         <div>
            <button onClick = {this.setStateHandler}>SET
STATE</button>
            <h4>State Array: {this.state.data}</h4>
         </div>
      );
   }
}
export default App;
```

Comenzamos con una matriz vacía. Cada vez que hacemos clic en el botón, el estado se actualizará. Si hacemos clic cinco veces, obtendremos el siguiente resultado.

Actualización de fuerza

A veces es posible que queramos actualizar el componente manualmente. Esto se puede lograr usando el método **forceUpdate ()** .

```
import React from 'react';

class App extends React.Component {
   constructor() {
      super();
      this.forceUpdateHandler =
this.forceUpdateHandler.bind(this);
   };
   forceUpdateHandler() {
      this.forceUpdate();
   };
   render() {
      return (
         <div>
            <button onClick = {this.forceUpdateHandler}>FORCE
UPDATE</button>
            <h4>Random number: {Math.random()}</h4>
         </div>
      );
   }
```

```
}
export default App;
```

Estamos configurando un número aleatorio que se actualizará cada vez que se haga clic en el botón.

Encontrar el nodo Dom

Para la manipulación DOM, podemos usar el método **ReactDOM.findDOMNode ()** . Primero necesitamos importar **react-dom** .

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
   constructor() {
      super();
      this.findDomNodeHandler =
this.findDomNodeHandler.bind(this);
   };
   findDomNodeHandler() {
      var myDiv = document.getElementById('myDiv');
      ReactDOM.findDOMNode(myDiv).style.color = 'green';
   }
   render() {
      return (
         <div>
            <button onClick = {this.findDomNodeHandler}>FIND
DOME NODE</button>
            <div id = "myDiv">NODE</div>
         </div>
      );
   }
}
export default App;
```

El color del elemento **myDiv** cambia a verde, una vez que se hace clic en el botón.

Nota : desde la actualización 0.14, la mayoría de los métodos de API de componentes más antiguos están en desuso o se eliminan para acomodar ES6.

ReactJS - Ciclo de vida del componente

En este capítulo, discutiremos los métodos del ciclo de vida de los componentes.

Métodos de ciclo de vida

- **componentWillMount** se ejecuta antes de la representación, tanto en el servidor como en el lado del cliente.
- **componentDidMount** se ejecuta después del primer render solo en el lado del cliente. Aquí es donde deben ocurrir las solicitudes AJAX y las actualizaciones DOM o de estado. Este método también se usa para la integración con otros marcos de JavaScript y cualquier función con ejecución retrasada, como **setTimeout** o **setInterval** . Lo estamos utilizando para actualizar el estado para que podamos activar los otros métodos de ciclo de vida.
- **componentWillReceiveProps** se invoca tan pronto como se actualizan los accesorios antes de que se llame a otro render. Lo **activamos** desde **setNewNumber** cuando actualizamos el estado.
- **shouldComponentUpdate** debería devolver **un** valor **verdadero** o **falso** . Esto determinará si el componente se actualizará o no. Esto se establece en **verdadero** de forma predeterminada. Si está seguro de que el componente no necesita renderizarse después de actualizar el **estado** o los **accesorios** , puede devolver **un** valor **falso** .
- Se llama a **componentWillUpdate** justo antes de la representación.
- Se llama a **componentDidUpdate** justo después de la representación.
- Se llama a **componentWillUnmount** después de desmontar el componente del dom. Estamos de desmontar nuestro componente en **main.js** .

En el siguiente ejemplo, estableceremos el **estado** inicial en la función constructora. El **setNewnumber** se utiliza para actualizar el **estado** . Todos los métodos del ciclo de vida están dentro del componente Contenido.

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor(props) {
```

```
      super(props);

      this.state = {
         data: 0
      }
      this.setNewNumber = this.setNewNumber.bind(this)
   };
   setNewNumber() {
      this.setState({data: this.state.data + 1})
   }
   render() {
      return (
         <div>
            <button onClick =
{this.setNewNumber}>INCREMENT</button>
            <Content myNumber = {this.state.data}></Content>
         </div>
      );
   }
}
class Content extends React.Component {
   componentWillMount() {
      console.log('Component WILL MOUNT!')
   }
   componentDidMount() {
      console.log('Component DID MOUNT!')
   }
   componentWillReceiveProps(newProps) {
      console.log('Component WILL RECIEVE PROPS!')
   }
   shouldComponentUpdate(newProps, newState) {
      return true;
   }
   componentWillUpdate(nextProps, nextState) {
      console.log('Component WILL UPDATE!');
   }
   componentDidUpdate(prevProps, prevState) {
      console.log('Component DID UPDATE!')
   }
   componentWillUnmount() {
      console.log('Component WILL UNMOUNT!')
   }
   render() {
      return (
         <div>
            <h3>{this.props.myNumber}</h3>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

setTimeout(() => {

ReactDOM.unmountComponentAtNode(document.getElementById('app'
));}, 10000);
```

Después del render inicial, obtendremos la siguiente pantalla.

ReactJS - Formularios

En este capítulo, aprenderemos cómo usar formularios en React.

Ejemplo simple

En el siguiente ejemplo, estableceremos un formulario de entrada con **valor = {this.state.data}** . Esto permite actualizar el estado cada vez que cambia el valor de entrada. Estamos utilizando el evento **onChange** que observará los cambios de entrada y actualizará el estado en consecuencia.

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor(props) {
      super(props);

      this.state = {
         data: 'Initial data...'
      }
      this.updateState = this.updateState.bind(this);
   };
   updateState(e) {
      this.setState({data: e.target.value});
   }
   render() {
      return (
         <div>
```

```
            <input type = "text" value = {this.state.data}
               onChange = {this.updateState} />
            <h4>{this.state.data}</h4>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Cuando el valor del texto de entrada cambia, el estado se actualizará.

Ejemplo complejo

En el siguiente ejemplo, veremos cómo usar formularios del componente hijo. **El** método **onChange** activará la actualización de estado que se pasará al **valor de** entrada secundario y se representará en la pantalla. Un ejemplo similar se utiliza en el capítulo Eventos. Siempre que necesitemos actualizar el estado del componente secundario, debemos pasar la función que manejará la actualización (**updateState**) como un accesorio (**updateStateProp**).

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor(props) {
      super(props);

      this.state = {
         data: 'Initial data...'
      }
      this.updateState = this.updateState.bind(this);
   };
   updateState(e) {
      this.setState({data: e.target.value});
```

```
   }
   render() {
      return (
         <div>
            <Content myDataProp = {this.state.data}
               updateStateProp =
{this.updateState}></Content>
         </div>
      );
   }
}
class Content extends React.Component {
   render() {
      return (
         <div>
            <input type = "text" value =
{this.props.myDataProp}
               onChange = {this.props.updateStateProp} />
            <h3>{this.props.myDataProp}</h3>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Esto producirá el siguiente resultado.

ReactJS - Eventos

En este capítulo, aprenderemos cómo usar eventos.

Ejemplo simple

Este es un ejemplo simple donde solo usaremos un componente. Solo estamos agregando el evento **onClick** que activará la función **updateState** una vez que se **haga** clic en el botón.

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor(props) {
      super(props);

      this.state = {
         data: 'Initial data...'
      }
      this.updateState = this.updateState.bind(this);
   };
   updateState() {
      this.setState({data: 'Data updated...'})
   }
   render() {
      return (
         <div>
            <button onClick =
{this.updateState}>CLICK</button>
            <h4>{this.state.data}</h4>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Esto producirá el siguiente resultado.

Eventos infantiles

Cuando necesitamos actualizar el **estado** del componente primario desde su elemento secundario, podemos crear un controlador de eventos

(**updateState**) en el componente primario y pasarlo como un accesorio (**updateStateProp**) al componente secundario donde simplemente podemos llamarlo.

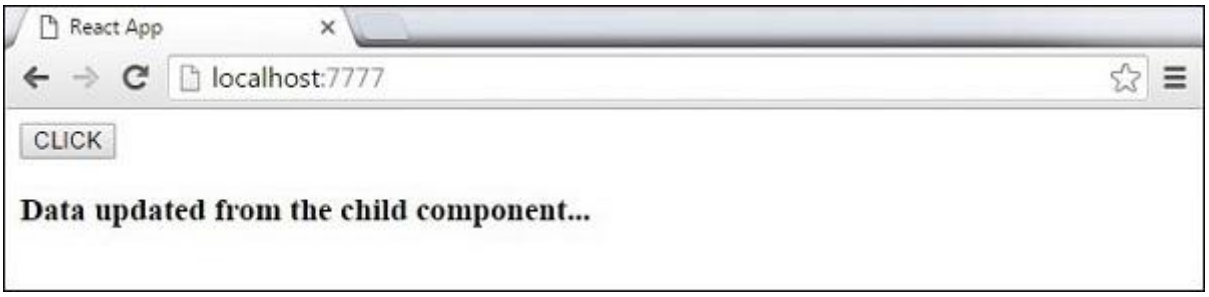
App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor(props) {
      super(props);

      this.state = {
         data: 'Initial data...'
      }
      this.updateState = this.updateState.bind(this);
   };
   updateState() {
      this.setState({data: 'Data updated from the child
component...'})
   }
   render() {
      return (
         <div>
            <Content myDataProp = {this.state.data}
               updateStateProp =
{this.updateState}></Content>
         </div>
      );
   }
}
class Content extends React.Component {
   render() {
      return (
         <div>
            <button onClick =
{this.props.updateStateProp}>CLICK</button>
            <h3>{this.props.myDataProp}</h3>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Esto producirá el siguiente resultado.



ReactJS - Refs

La **referencia** se usa para devolver una referencia al elemento. **Las referencias** deben evitarse en la mayoría de los casos, sin embargo, pueden ser útiles cuando necesitamos mediciones DOM o para agregar métodos a los componentes.

Usando referencias

El siguiente ejemplo muestra cómo usar referencias para borrar el campo de entrada. **La** función **ClearInput** busca el elemento con el valor **ref = "myInput"** , restablece el estado y le agrega foco después de hacer clic en el botón.

App.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
   constructor(props) {
      super(props);

      this.state = {
         data: ''
      }
      this.updateState = this.updateState.bind(this);
      this.clearInput = this.clearInput.bind(this);
   };
   updateState(e) {
      this.setState({data: e.target.value});
   }
   clearInput() {
      this.setState({data: ''});
      ReactDOM.findDOMNode(this.refs.myInput).focus();
   }
   render() {
      return (
         <div>
            <input value = {this.state.data} onChange =
{this.updateState}
```

```
                ref = "myInput"></input>
             <button onClick =
{this.clearInput}>CLEAR</button>
             <h4>{this.state.data}</h4>
          </div>
       );
    }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Una vez que se hace clic en el botón, la **entrada** se borrará y se enfocará.

ReactJS - Keys

Las **claves de** reacción son útiles cuando se trabaja con componentes creados dinámicamente o cuando los usuarios alteran sus listas. Establecer el valor **clave** mantendrá sus componentes identificados de forma única después del cambio.

Usando llaves

Creemos dinámicamente elementos de **contenido** con índice único (i). La función de **mapa** creará tres elementos de nuestra matriz de **datos** . Como el valor de la **clave** debe ser único para cada elemento, asignaremos i como clave para cada elemento creado.

App.jsx

```
import React from 'react';

class App extends React.Component {
   constructor() {
      super();
```

```
      this.state = {
         data:[
            {
               component: 'First...',
               id: 1
            },
            {
               component: 'Second...',
               id: 2
            },
            {
               component: 'Third...',
               id: 3
            }
         ]
      }
   }
   render() {
      return (
         <div>
            <div>
               {this.state.data.map((dynamicComponent, i) =>
<Content
                  key = {i} componentData =
{dynamicComponent}/>)}
            </div>
         </div>
      );
   }
}
class Content extends React.Component {
   render() {
      return (
         <div>
            <div>{this.props.componentData.component}</div>
            <div>{this.props.componentData.id}</div>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

Obtendremos el siguiente resultado para los valores clave de cada elemento.

Si agregamos o eliminamos algunos elementos en el futuro o cambiamos el orden de los elementos creados dinámicamente, React usará los valores **clave** para realizar un seguimiento de cada elemento.

ReactJS - Enrutador

En este capítulo, aprenderemos cómo configurar el enrutamiento para una aplicación.

Paso 1 - Instalar un enrutador React

Una forma sencilla de instalar el **enrutador reactivo** es ejecutar el siguiente fragmento de código en la ventana del **símbolo del sistema** .

```
C:\Users\username\Desktop\reactApp>npm install react-router
```

Paso 2 - Crear componentes

En este paso, crearemos cuatro componentes. El componente de la **aplicación** se utilizará como un menú de pestañas. Los otros tres componentes **(Inicio), (Acerca de)** y **(Contacto)** se representan una vez que la ruta ha cambiado.

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, Link, browserHistory, IndexRoute }
from 'react-router'

class App extends React.Component {
   render() {
      return (
         <div>
            <ul>
            <li>Home</li>
            <li>About</li>
            <li>Contact</li>
            </ul>
            {this.props.children}
```

```
            </div>
        )
    }
}
export default App;

class Home extends React.Component {
    render() {
        return (
            <div>
                <h1>Home...</h1>
            </div>
        )
    }
}
export default Home;

class About extends React.Component {
    render() {
        return (
            <div>
                <h1>About...</h1>
            </div>
        )
    }
}
export default About;

class Contact extends React.Component {
    render() {
        return (
            <div>
                <h1>Contact...</h1>
            </div>
        )
    }
}
export default Contact;
```

Paso 3: agregar un enrutador

Ahora, agregaremos rutas a la aplicación. En lugar de representar el elemento de la **aplicación** como en el ejemplo anterior, esta vez se representará el **enrutador** . También estableceremos componentes para cada ruta.

main.js

```
ReactDOM.render((
    <Router history = {browserHistory}>
        <Route path = "/" component = {App}>
            <IndexRoute component = {Home} />
```

```
            <Route path = "home" component = {Home} />
            <Route path = "about" component = {About} />
            <Route path = "contact" component = {Contact} />
         </Route>
      </Router>
), document.getElementById('app'))
```

Cuando se inicia la aplicación, veremos tres enlaces en los que se puede hacer clic que se pueden usar para cambiar la ruta.

ReactJS - Concepto de flujo

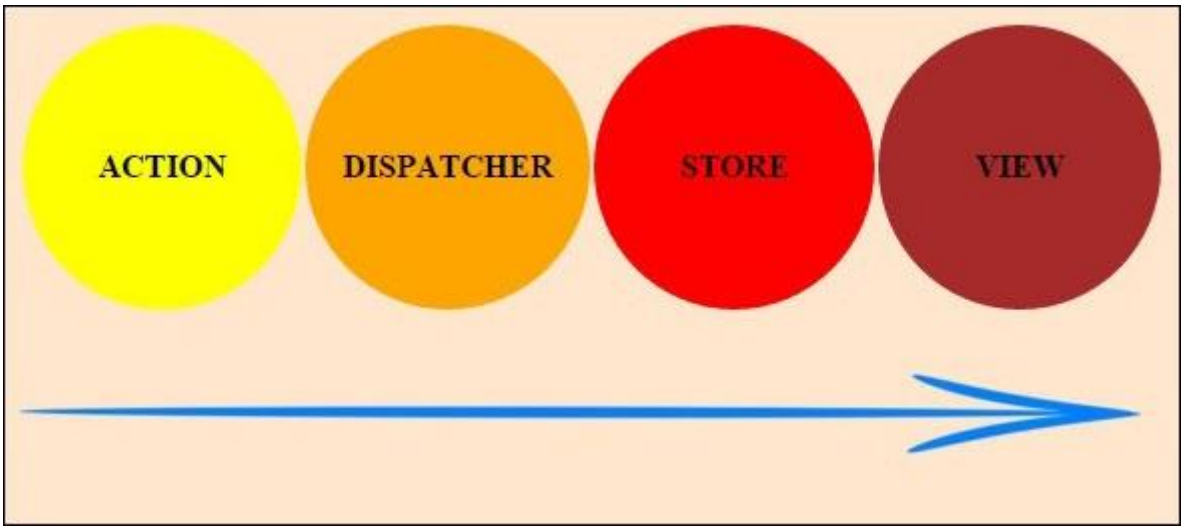
Flux es un concepto de programación, donde los datos son **unidireccionales** . Estos datos ingresan a la aplicación y fluyen a través de ella en una dirección hasta que se muestran en la pantalla.

Elementos de flujo

A continuación hay una explicación simple del concepto de **flujo** . En el próximo capítulo, aprenderemos cómo implementar esto en la aplicación.

- **Acciones** : las acciones se envían al despachador para activar el flujo de datos.
- **Despachador** : este es un centro central de la aplicación. Todos los datos se envían y se envían a las tiendas.
- **Store** - Store es el lugar donde se guardan el estado y la lógica de la aplicación. Cada tienda mantiene un estado particular y se actualizará cuando sea necesario.
- **Ver** : la **vista** recibirá datos de la tienda y volverá a representar la aplicación.

El flujo de datos se representa en la siguiente imagen.



Flux Pros

- El flujo de datos direccional único es fácil de entender.
- La aplicación es más fácil de mantener.
- Las partes de la aplicación están desacopladas.

ReactJS - Usando Flux

En este capítulo, aprenderemos cómo implementar el patrón de flujo en las aplicaciones React. Utilizaremos el marco de **Redux** . El objetivo de este capítulo es presentar el ejemplo más simple de cada pieza necesaria para conectar **Redux** y **React** .

Paso 1 - Instalar Redux

Instalaremos Redux a través de la ventana del **símbolo del sistema** .

```
C:\Users\username\Desktop\reactApp>npm install --save react-
redux
```

Paso 2: crea archivos y carpetas

En este paso, crearemos carpetas y archivos para nuestras **acciones** , **reductores** y **componentes** . Una vez que hayamos terminado, así es como se verá la estructura de carpetas.

```
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir actions
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir components
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir reducers
C:\Users\Tutorialspoint\Desktop\reactApp>type nul >
actions/actions.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul >
reducers/reducers.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul >
components/AddTodo.js
```

```
C:\Users\Tutorialspoint\Desktop\reactApp>type nul >
components/Todo.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul >
components/TodoList.js
```

Paso 3 - Acciones

Las acciones son objetos de JavaScript que utilizan la propiedad de **tipo** para informar sobre los datos que deben enviarse a la tienda. Estamos definiendo la acción **ADD_TODO** que se usará para agregar un nuevo elemento a nuestra lista. La función **addTodo** es un creador de acciones que devuelve nuestra acción y establece una **identificación** para cada elemento creado.

actions / actions.js

```
export const ADD_TODO = 'ADD_TODO'

let nextTodoId = 0;

export function addTodo(text) {
   return {
      type: ADD_TODO,
      id: nextTodoId++,
      text
   };
}
```

Paso 4 - Reductores

Si bien las acciones solo desencadenan cambios en la aplicación, los **reductores** especifican esos cambios. Estamos utilizando la instrucción **switch** para buscar una acción **ADD_TODO** . El reductor es una función que toma dos parámetros (**estado** y **acción**) para calcular y devolver un estado actualizado.

La primera función se usará para crear un nuevo elemento, mientras que la segunda empujará ese elemento a la lista. Hacia el final, estamos utilizando la función de ayuda **combineReducers** donde podemos agregar cualquier nuevo reductor que podamos usar en el futuro.

reductores / reductores.js

```
import { combineReducers } from 'redux'
import { ADD_TODO } from '../actions/actions'

function todo(state, action) {
   switch (action.type) {
      case ADD_TODO:
         return {
            id: action.id,
            text: action.text,
         }
      default:
         return state
   }
}
function todos(state = [], action) {
   switch (action.type) {
      case ADD_TODO:
         return [
            ...state,
            todo(undefined, action)
         ]
      default:
         return state
   }
}
const todoApp = combineReducers({
   todos
})
export default todoApp
```

Paso 5 - Almacenar

La tienda es un lugar que contiene el estado de la aplicación. Es muy fácil crear una tienda una vez que tenga reductores. Estamos pasando la propiedad de la tienda al elemento **proveedor** , que envuelve nuestro componente de ruta.

main.js

```
import React from 'react'

import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
```

```
import App from './App.jsx'
import todoApp from './reducers/reducers'

let store = createStore(todoApp)
let rootElement = document.getElementById('app')

render(
   <Provider store = {store}>
      <App />
   </Provider>,

   rootElement
)
```

Paso 6 - Componente raíz

El componente de la **aplicación** es el componente raíz de la aplicación. Solo el componente raíz debe tener en cuenta un redux. La parte importante a tener en cuenta es la función de **conexión** que se utiliza para conectar nuestra **aplicación de** componente raíz a la **tienda** .

Esta función toma la función **select** como argumento. La función de selección toma el estado de la tienda y devuelve los accesorios (**visiblesTodos**) que podemos usar en nuestros componentes.

App.jsx

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { addTodo } from './actions/actions'

import AddTodo from './components/AddTodo.js'
import TodoList from './components/TodoList.js'

class App extends Component {
   render() {
      const { dispatch, visibleTodos } = this.props

      return (
         <div>
            <AddTodo onAddClick = {text
=>dispatch(addTodo(text))} />
            <TodoList todos = {visibleTodos}/>
         </div>
      )
   }
}
function select(state) {
   return {
      visibleTodos: state.todos
   }
```

```
}
export default connect(select)(App);
```

Paso 7 - Otros componentes

Estos componentes no deben ser conscientes de redux.

componentes / AddTodo.js

```
import React, { Component, PropTypes } from 'react'

export default class AddTodo extends Component {
   render() {
      return (
         <div>
            <input type = 'text' ref = 'input' />

            <button onClick = {(e) => this.handleClick(e)}>
               Add
            </button>
         </div>
      )
   }
   handleClick(e) {
      const node = this.refs.input
      const text = node.value.trim()
      this.props.onAddClick(text)
      node.value = ''
   }
}
```

componentes / Todo.js

```
import React, { Component, PropTypes } from 'react'

export default class Todo extends Component {
   render() {
      return (
         <li>
            {this.props.text}
         </li>
      )
   }
}
```

componentes / TodoList.js

```
import React, { Component, PropTypes } from 'react'
import Todo from './Todo.js'
```

```
export default class TodoList extends Component {
   render() {
      return (
         <ul>
            {this.props.todos.map(todo =>
               <Todo
                  key = {todo.id}
                  {...todo}
               />
            )}
         </ul>
      )
   }
}
```

Cuando iniciemos la aplicación, podremos agregar elementos a nuestra lista.

ReactJS - Animaciones

En este capítulo, aprenderemos cómo animar elementos usando React.

Paso 1 - Instalar el grupo React CSS Transitions

Este es el complemento React utilizado para crear transiciones y animaciones CSS básicas. Lo instalaremos desde la ventana del **símbolo del sistema** :

```
C:\Users\username\Desktop\reactApp>npm install react-addons-
css-transition-group
```

Paso 2: agrega un archivo CSS

Creemos un nuevo archivo style.css.

```
C:\Users\Tutorialspoint\Desktop\reactApp>type nul >
css/style.css
```

Para poder usarlo en la aplicación, necesitamos vincularlo al elemento head en index.html.

```
<!DOCTYPE html>
<html lang = "en">
   <head>
      <link rel = "stylesheet" type = "text/css" href =
"./style.css">
      <meta charset = "UTF-8">
```

```
      <title>React App</title>
   </head>
   <body>
      <div id = "app"></div>
      <script src = 'index_bundle.js'></script>
   </body>
</html>
```

Paso 3: aparece la animación

Crearemos un componente básico React. El elemento **ReactCSSTransitionGroup** se usará como una envoltura del componente que queremos animar. Utilizará **transitionAppear** y **transitionAppearTimeout** , mientras **transitionEnter** y **transitionLeave** son falsos.

App.jsx

```
import React from 'react';
var ReactCSSTransitionGroup = require('react-addons-css-
transition-group');

class App extends React.Component {
   render() {
      return (
         <div>
            <ReactCSSTransitionGroup transitionName =
"example"
               transitionAppear = {true}
transitionAppearTimeout = {500}
               transitionEnter = {false} transitionLeave =
{false}>

               <h1>My Element...</h1>
            </ReactCSSTransitionGroup>
         </div>
      );
   }
}
export default App;
```

main.js

```
import React from 'react'
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

La animación CSS es muy simple.

css / style.css

```
.example-appear {
   opacity: 0.04;
}
.example-appear.example-appear-active {
   opacity: 2;
   transition: opacity 50s ease-in;
}
```

Una vez que iniciamos la aplicación, el elemento se desvanecerá.

Paso 4: ingresar y salir de animaciones

Las animaciones de entrada y salida se pueden usar cuando deseamos agregar o eliminar elementos de la lista.

App.jsx

```
import React from 'react';
var ReactCSSTransitionGroup = require('react-addons-css-
transition-group');

class App extends React.Component {
   constructor(props) {
      super(props);

      this.state = {
         items: ['Item 1...', 'Item 2...', 'Item 3...', 'Item
4...']
      }
      this.handleAdd = this.handleAdd.bind(this);
   };
   handleAdd() {
      var newItems = this.state.items.concat([prompt('Create
New Item')]);
      this.setState({items: newItems});
   }
   handleRemove(i) {
      var newItems = this.state.items.slice();
      newItems.splice(i, 1);
```

```
      this.setState({items: newItems});
   }
   render() {
      var items = this.state.items.map(function(item, i) {
         return (
            <div key = {item} onClick =
{this.handleRemove.bind(this, i)}>
               {item}
            </div>
         );
      }.bind(this));

      return (
         <div>
            <button onClick = {this.handleAdd}>Add
Item</button>

            <ReactCSSTransitionGroup transitionName =
"example"
               transitionEnterTimeout = {500}
transitionLeaveTimeout = {500}>
               {items}
            </ReactCSSTransitionGroup>
         </div>
      );
   }
}
export default App;
```

main.js

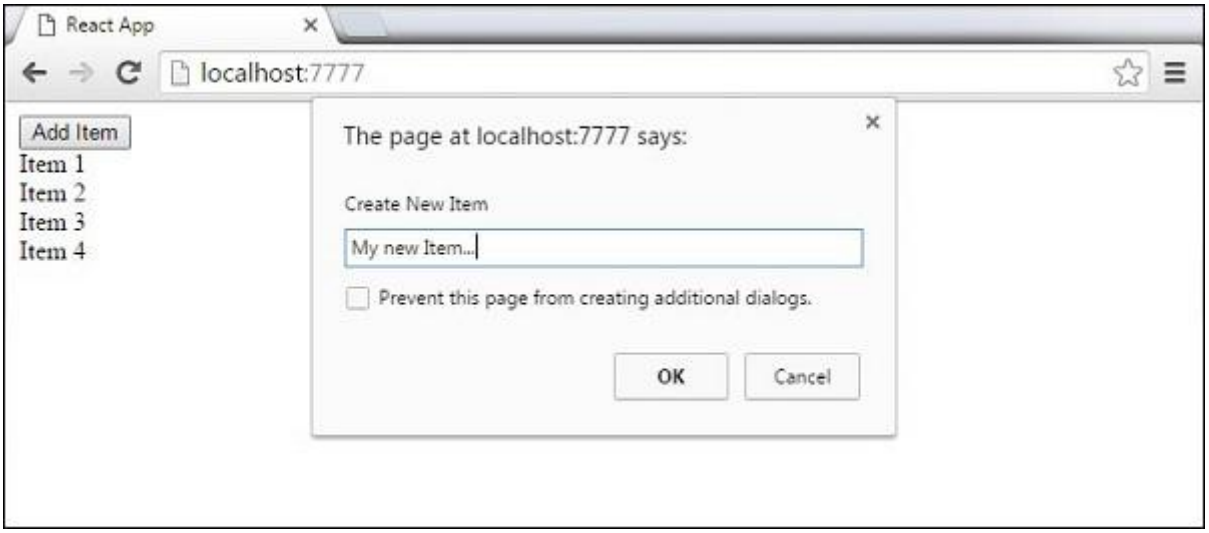
```
import React from 'react'
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

css / style.css

```
.example-enter {
   opacity: 0.04;
}
.example-enter.example-enter-active {
   opacity: 5;
   transition: opacity 50s ease-in;
}
.example-leave {
   opacity: 1;
}
.example-leave.example-leave-active {
   opacity: 0.04;
   transition: opacity 50s ease-in;
```

```
}
```

Cuando iniciamos la aplicación y hacemos clic en el botón **Agregar elemento** , aparecerá el mensaje.



Una vez que ingresamos el nombre y presionamos OK, el nuevo elemento se desvanecerá.

Ahora podemos eliminar algunos de los elementos (**Elemento 3 ...**) haciendo clic en él. Este artículo desaparecerá de la lista.

ReactJS - Componentes de orden superior

Los componentes de orden superior son funciones de JavaScript que se utilizan para agregar funcionalidades adicionales al componente existente. Estas funciones son **puras** , lo que significa que están recibiendo datos y devolviendo valores de acuerdo con esos datos. Si los datos cambian, las funciones de orden superior se vuelven a ejecutar con una entrada de

datos diferente. Si queremos actualizar nuestro componente de retorno, no tenemos que cambiar el HOC. Todo lo que necesitamos hacer es cambiar los datos que está utilizando nuestra función.

El Componente de orden superior (HOC) se ajusta al componente "normal" y proporciona datos adicionales. En realidad, es una función que toma un componente y devuelve otro componente que envuelve el original.

Echemos un vistazo a un ejemplo simple para comprender fácilmente cómo funciona este concepto. El **MyHOC** es una función de orden superior que se utiliza sólo para pasar datos a **MyComponent** . Esta función toma **MyComponent** , lo mejora con **newData** y devuelve el componente mejorado que se mostrará en la pantalla.

```
import React from 'react';

var newData = {
   data: 'Data from HOC...',
}

var MyHOC = ComposedComponent => class extends
React.Component {
   componentDidMount() {
      this.setState({
         data: newData.data
      });
   }
   render() {
      return <ComposedComponent {...this.props}
{...this.state} />;
   }
};
class MyComponent extends React.Component {
   render() {
      return (
         <div>
            <h1>{this.props.data}</h1>
         </div>
      )
   }
}

export default MyHOC(MyComponent);
```

Si ejecutamos la aplicación, veremos que los datos se pasan a **MyComponent** .

Nota : los componentes de orden superior se pueden usar para diferentes funcionalidades. Estas funciones puras son la esencia de la programación funcional. Una vez que esté acostumbrado, notará cómo su aplicación se está volviendo más fácil de mantener o actualizar.

ReactJS - Mejores prácticas

En este capítulo, enumeraremos las mejores prácticas, métodos y técnicas de React que nos ayudarán a mantener la coherencia durante el desarrollo de la aplicación.

- **Estado** : el estado debe evitarse tanto como sea posible. Es una buena práctica centralizar el estado y pasarlo al árbol de componentes como accesorios. Siempre que tengamos un grupo de componentes que necesiten los mismos datos, debemos establecer un elemento contenedor alrededor de ellos que contendrá el estado. El patrón de flujo es una buena manera de manejar el estado en las aplicaciones React.
- **PropTypes** : los PropTypes siempre deben definirse. Esto ayudará a rastrear todos los accesorios en la aplicación y también será útil para cualquier desarrollador que trabaje en el mismo proyecto.
- **Renderizado** : la mayor parte de la lógica de la aplicación debe moverse dentro del método de renderizado. Deberíamos tratar de minimizar la lógica en los métodos del ciclo de vida de los componentes y mover esa lógica en el método de representación. Cuanto menos estado y accesorios utilizamos, más limpio será el código. Siempre debemos hacer que el estado sea lo más simple posible. Si necesitamos calcular algo a partir del estado o los accesorios, podemos hacerlo dentro del método de renderizado.
- **Composición** : el equipo de React sugiere utilizar un principio de responsabilidad única. Esto significa que un componente solo debe ser responsable de una funcionalidad. Si algunos de los componentes tienen más de una funcionalidad, deberíamos refactorizar y crear un nuevo componente para cada funcionalidad.
- **Componentes de orden superior (HOC)** : las versiones anteriores de React ofrecían mixins para manejar funcionalidades reutilizables. Dado que los mixins ahora están en desuso, una de las soluciones es usar HOC.