

Tema 4

Vistas

Renderizado Condicional

Una de las características más interesantes de React es la posibilidad de trabajar con los operadores condicionales de Javascript.

Esto nos permite seleccionar qué elementos de nuestros componentes van a estar visibles a la hora de renderizarlos. Mediante los diferentes atributos de nuestros componentes vamos a ser capaces de renderizar unos elementos u otros.

Es una manera también de evitar la repetición de código en los diferentes componentes de nuestra aplicación.

Renderizado Condicional

Habitualmente, la manera más sencilla para mostrar los elementos de un componente de manera condicional es trabajar con variables.

Podemos definir con variables los diferentes elementos que vamos a mostrar en el componente y comprobando diferentes propiedades, mostrar unas u otras.

En el siguiente ejemplo vamos a ver cómo generar un componente que cambie de aspecto dependiendo de una propiedad almacenada dentro del atributo **state** del propio componente.

Renderizado Condicional

```
class LoginControl extends React.Component{
  constructor (props){
    super(props);
    this.state = {entra: false};
  }
  manejarLoginClick = () => {
    this.setState({entra: true});
  }
  manejarLogoutClick = () => {
    this.setState({entra: false});
  }
  render(){
    const entra = this.state.entra;
    let boton = null;
    if(entra){
      boton = <LogoutButton onClick={this.manejarLogoutClick} /
    >;
    }else{
      boton = <LoginButton onClick={this.manejarLoginClick} />;
    }
    console.log(boton);
    return (
      <div>
        {boton}
      </div>
    );
  }
}

function LoginButton(props) {
  return (<button onClick={props.onClick}>Login</button>);
}
function LogoutButton(props) {
  return (<button onClick={props.onClick}>Logout</button>);
}

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

Renderizado Condicional

Existen otro tipo de estructuras más sencillas para poder seleccionar qué elementos vamos a mostrar en el renderizado de nuestros componentes.

Podemos combinar el uso de cualquier expresión en Javascript que devuelva un valor booleano y el operador **&&**. Por ejemplo, un componente que muestre un mensaje en función de una propiedad:

```
function Mensajes(props){  
  const mensajesSinLeer = props.mensajesSinLeer;  
  return(  
    <div>  
      {mensajesSinLeer > 0 &&  
        <h2>Tienes {mensajesSinLeer} mensajes sin leer</h2>  
      }  
    </div>  
  );  
}  
  
ReactDOM.render(  
  <Mensajes mensajesSinLeer="1" />,  
  document.getElementById('root')  
);
```

Renderizado Condicional

La fórmula utilizada en el ejemplo anterior funciona porque en Javascript, cuando evaluamos **true && expresión**, siempre devuelve la expresión para ser renderizada.

Si evaluamos **false && expresión**, el resultado es false y por tanto, la expresión no se mostrará.

Renderizado Condicional

Otro de los métodos para generar un renderizado condicional es a través del **operador ternario** de Javascript.

condición ? true : false

Modificamos el ejemplo anterior para usar este operador.

```
function Mensajes(props){  
  const mensajesSinLeer = props.mensajesSinLeer;  
  return(  
    <div>  
      <h2>Tienes {mensajesSinLeer > 0 ? mensajesSinLeer : 'cero o  
menos' } mensajes sin leer</h2>  
    </div>  
  );  
}
```

Renderizado Condicional

Podemos incluso, utilizar dicho operador con expresiones más complejas. Podríamos adaptar el ejemplo anterior con los botones Login / Logout:

```
render(){  
  const entra = this.state.entra;  
  return (  
    <div>  
      {entra ? (  
        <LogoutButton onClick={this.manejarLogoutClick} />  
      ) : (  
        <LoginButton onClick={this.manejarLoginClick} />  
      )}  
    </div>  
  );  
}
```


Renderizado Condicional

En ocasiones queremos ocultar nuestros componentes en función de los datos registrados en nuestra aplicación.

Lo podemos realizar de manera sencilla devolviendo **null** en el método que renderiza el componente.

```
function Mensajes(props){
  const mensajesSinLeer = props.mensajesSinLeer;
  if (mensajesSinLeer == 0){
    return null;
  }
  return(
    <div>
      <h2>Tienes {mensajesSinLeer > 0 ? mensajesSinLeer : 'cero o
menos' } mensajes sin leer</h2>
    </div>
  );
}
```

Listas

En Javascript disponemos de una serie de métodos que nos permiten el trabajo por bloques sobre arrays. La función **map**, por ejemplo, nos permite aplicar la misma acción a todos los elementos de un array.

Podemos utilizar este tipo de funciones para aplicar, sobre un grupo de elementos de nuestro componente las mismas acciones. Por ejemplo, si necesitamos generar una lista de números, podríamos aplicarle las características de la siguiente manera, a partir de un array:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);

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

Listas

Podemos reescribir el ejemplo anterior para incorporar dicha funcionalidad dentro de nuestros componentes.

```
function ListaNumeros(props) {  
  const numeros = props.numbers;  
  const elementos = numeros.map((number) =>  
    <li>{number}</li>  
  );  
  return (  
    <ul>{elementos}</ul>  
  );  
}  
  
const numbers = [1, 2, 3, 4, 5, 6];  
ReactDOM.render(  
  <ListaNumeros numbers={numbers} />,  
  document.getElementById('root')  
);
```

Listas

Si inspeccionamos la página generada, podemos observar un aviso como el siguiente

```

Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of `ListaNumeros`. See
s://fb.me/react-warning-keys for more information.
    in li (at index.js:59)
    in ListaNumeros (at index.js:68)
  
```

Nos indica que debemos asignarle una clave para cada uno de los elementos dentro de la lista, sobre todo para identificar en cualquier momento, cualquiera de los componentes generados.

Podemos agregar la clave sobre cada uno de los elementos de la lista.

```
<li key={number.toString}>{number}</li>
```

Claves

La asignación de claves ayuda a React a identificar cada uno de los elementos dentro de la estructura de componentes de nuestras aplicaciones.

Juegan un papel muy importante para ayudar a la aplicación a identificar qué elementos han cambiado, han sido añadidos o se han borrado.

Como hemos visto en el ejemplo anterior, debemos definir una clave para cada uno de los elementos de listas generadas de manera dinámica.

Claves

La mejor manera de establecer estas claves es la de asignar aquel literal que identifique de manera única el elemento con aquellos elementos que se encuentren a la misma altura en la estructura del componente. No hace falta que estas claves sean únicas a nivel global de la aplicación.

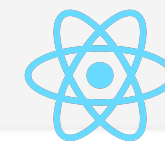
Si no disponemos de una clave clara, siempre se puede usar el índice del bucle que estemos utilizando para listar los elementos.

Claves

En los ejemplos vistos anteriormente hemos utilizado el método `map` para definir una variable situada fuera de nuestro código JSX.

JSX nos permite el uso de cualquier expresión, por lo tanto, podríamos incluir el uso de **map** en la propia definición de nuestros elementos JSX.

```
function ListaNumeros(props) {  
  const numeros = props.numbers;  
  return (  
    <ul>  
      { numeros.map((number) =>  
        <li key={number.toString()}>{number}</li>  
      )}  
    </ul>  
  );  
}
```



Formularios

Los formularios que utilizamos en la creación de nuestras páginas HTML tienen un comportamiento diferente al resto de elementos que estamos acostumbrados a utilizar.

El hecho de definir un botón de tipo *submit* implica que todos los datos incluidos dentro del formulario se van a enviar a través del método seleccionado a otra página en el momento de pulsar el botón.

Cuando trabajamos on React es recomendable *sobreescribir* esta funcionalidad para poder realizar todas estas acciones a través de Javascript. El trabajo con este tipo de formularios se realiza a través de los **Componentes Controlados**. Dentro del mismo componente vamos a mostrar y a procesar el formulario.

Formularios

Para poder procesar nuestros formularios tenemos que tener en cuenta una serie de acciones:

- Debemos controlar el envío del formulario asignando la acción **onSubmit** del mismo, sobre un método de nuestro componente.
- Tenemos que enlazar la propiedad **value** de cada uno de los campos del formulario que queremos controlar con la propiedad **state** del componente.
- Podemos controlar los cambios sobre los diferentes campos capturando el evento **onChange** de cada uno de ellos.

En el siguiente ejemplo vemos cómo capturar el valor de un campo de texto dentro de un sencillo formulario.

Formularios

```
class Formulario extends React.Component{

  constructor(props){
    super(props);
    this.state = {value: ''}
  }

  manejarCambios = (event) => {
    this.setState({value: event.target.value});
  }

  manejarEnvio = (event) => {
    console.log("Se ha enviado el valor " + this.state.value);
    event.preventDefault();
  }

  render(){
    return (
      <form onSubmit={this.manejarEnvio}>
        <label>Nombre:</label>
        <input type="text" value={this.state.value} onChange={this.manejarCambios} />
        <input type="submit" value="Enviar" />
      </form>
    );
  }
}
```

Formularios

En el ejemplo anterior hemos conseguido tener de manera global dentro de nuestro componente, el valor del campo de texto.

Además, como conocemos cuándo se están modificando cada uno de los campos, podemos utilizar los métodos que capturan dichos cambios para realizar validaciones sobre los valores o para realizar modificaciones antes del envío de los datos.

Formularios - TextArea

Existen algunos elementos que tienen comportamientos diferentes en React.

El elemento **textarea** normalmente define su contenido a partir del valor que encontremos entre sus etiquetas de apertura y cierre.

En React, tenemos que enlazar la propiedad **value** del textarea con cualquiera de las propiedades de nuestra clase.

```
<textarea value={this.state.descripcion} onChange={this.manejarTextArea} />
```

Formularios - Select

Lo mismo pasa con el elemento **select** a la hora de generar campos desplegados.

Necesitamos enlazar su valor a partir del atributo **value**:

```
<select value={this.state.seleccion} onChange={this.manejarSeleccion}>
  <option value="platano">Plátano</option>
  <option value="pera">Pera</option>
  <option value="manzana">Manzana</option>
  <option value="sandia">Sandía</option>
</select>
```

Formularios

Cuando tenemos que controlar la inserción de varios campos dentro de un mismo formulario, podemos trabajar con la propiedad **name** de cada uno de ellos para identificarlos.

El primer paso sería definir el elemento y asignarle su nombre y valor:

```
<input type="text" name="nombre" value={this.state.nombre} onChange={this.manejarCambios} />
```

Formularios

Una vez recibamos los cambios de los campos que estamos controlando, podemos guardar dentro de **state** sus valores de manera genérica.

```
manejarCambios = (event) => {  
  const name = event.target.name;  
  this.setState({[name]: event.target.value});  
}
```

De esta manera, podremos acceder a cada uno de los valores de nuestro formulario a través de **state** acompañado del nombre del campo.

Aplicación TODO

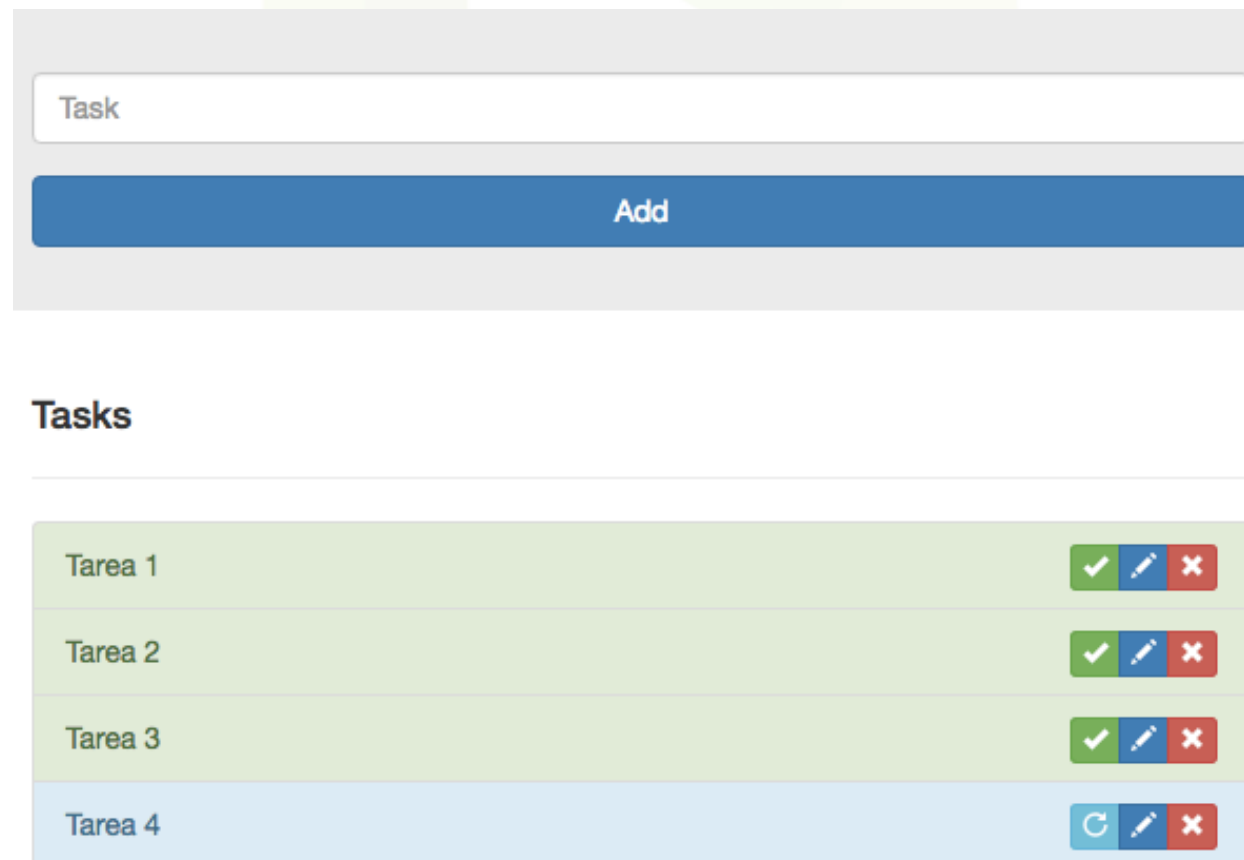
Una aplicación TODO se basa en la gestión de una lista con diferentes tareas, de manera que el usuario puede marcarlas como completadas o no.

El principal objetivo para el usuario es organizar las tareas que debe hacer y poder gestionarlas de forma eficiente.













Es un tipo de aplicación muy útil para empezar a programar en cualquier lenguaje, ya que se aprende a desarrollar funcionalidades para futuras aplicaciones más complejas.

Aplicación TODO

Vamos a desarrollar una aplicación sencilla para gestionar una lista de tareas. El objetivo de la aplicación es poder añadir tareas a una lista, marcarlas como completadas o no completadas y eliminarlas de la lista. Tendrá el siguiente aspecto:



The mockup shows a web interface for a TODO application. At the top, there is a text input field labeled 'Task' and a blue 'Add' button below it. Below the input is a section titled 'Tasks' which contains a list of four tasks. The first three tasks, 'Tarea 1', 'Tarea 2', and 'Tarea 3', are highlighted in light green and each has a green checkmark icon, a blue pencil icon, and a red 'x' icon. The fourth task, 'Tarea 4', is highlighted in light blue and has a blue circular arrow icon, a blue pencil icon, and a red 'x' icon.

Tasks	
Tarea 1	  
Tarea 2	  
Tarea 3	  
Tarea 4	  

Aplicación TODO

En nuestra aplicación, distinguiremos entre cuatro componentes principales:

- **App**: mantiene los datos y los métodos que actúan sobre esos datos.
- **ToDoAction**: Proporciona y asigna los elementos a realizar para una tarea. Este componente será consumido por el componente que generará la lista de tareas. En este componente, hay un bloque de control el cual generará diferentes tipos de botones para un objeto tarea, dependiendo del estado de la tarea.

Aplicación TODO

- **ToDoForm**: tiene el control de la forma de la aplicación. Este componente servirá para la creación de nuevas tareas y para la edición de las tareas actuales.
- **ToDoList**: creará la lista de tareas y añadirá los accionables a los objetos con la ayuda del componente *ToDoAction*. Dentro de este componente, se decidirá el nombre de la clase del objeto en base a su estado.

Aplicación TODO

Para empezar a crear nuestra aplicación de tareas, deberemos crear un nuevo proyecto react con la orden **\$ create-react-app todoapp**. Modificaremos el archivo **index.js** para que se comporte como tal.

Empezaremos creando el componente App. Este constará de distintos métodos para la gestión de las tareas de la lista. Entre ellos estarán:

- **Añadir tarea:** agregará una nueva tarea a la lista.
- **Eliminar tarea:** quitará la tarea seleccionada de la lista de tareas.
- **Editar tarea:** cambiará el texto de la tarea seleccionada.

App

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

var App = React.createClass({
  getInitialState: function() {
    return {
      tasks: [
        { text: "Tarea 1", completed: false },
        { text: "Tarea 2", completed: false },
        { text: "Tarea 3", completed: false },
        { text: "Tarea 4", completed: true }
      ]
    }
  },
  addTask: function(task) {
    if( task.text ){
      this.state.tasks.push(task);
      this.setState( this.state );
    }
  },
  saveTask: function(index, task) {
    if( task.text ){
      this.state.tasks[index] = task;
      this.state.edit = null;
      this.setState( this.state );
    }
  },
  removeTask: function(index) {
    if( index >= 0 && this.state.tasks.length > 0 ){
      this.state.tasks.splice(index, 1);
      this.setState( this.state );
    }
  },

```

```
editTask: function(index) {
  if( index >= 0 && this.state.tasks.length > 0 ){
    var task = this.state.tasks[index];
    task.index = index
    this.setState({edit:task});
  } },
  taskAction: function(index){
    this.state.tasks[index].completed = !
    this.state.tasks[index].completed;
    this.setState( this.state );
  },
  render: function() {
    return (
      <div>
        <ToDoForm
          add={this.addTask}
          save={this.saveTask}
          edit={this.state.edit} />
        <ToDoList
          tasks={this.state.tasks}
          remove={this.removeTask}
          edit={this.editTask}
          taskAction={this.taskAction} />
      </div> );
    }
  });

```

ToDoForm

```
var ToDoForm = React.createClass({  
  ...  
})
```

El siguiente componente será el *ToDoForm*. Este componente renderizará principalmente el botón para añadir una nueva tarea o guardar una tarea editada.

Contará con un método para manejarlo, el cual creará una nueva variable tarea en caso de estar agregando. Para el modo de edición, cogerá el texto actual de la tarea en cuestión y su estado (completada o no), y los cambiará en función de los parámetros que le pasemos.

ToDoForm

```
var ToDoForm = React.createClass({
  componentWillReceiveProps(newProps){
    if(newProps.edit){
      this.refs.task.value = newProps.edit.text;
    }
  },
  handleAddSave: function(){
    if(this.props.edit){
      var task = {
        text: this.refs.task.value,
        completed: this.props.edit.completed
      };
      this.props.save( this.props.edit.index,
        task );
    } else {
      var task = {text: this.refs.task.value,
        completed: false};
      this.props.add( task );
    }
    this.refs.task.value = '';
  },
```

```
render: function() {
  return (
    <div className="jumbotron padded">
      <form>
        <div className="row">
          <div className="col-md-9">
            <div className="form-group">
              <input type="text"
className="form-control" placeholder="Task"
ref="task" />
            </div>
          </div>
          <div className="col-md-3">
            <button type="button"
onClick={this.handleAddSave} className="btn btn-
primary btn-block">{this.props.edit ? 'Save' :
'Add'}</button>
          </div>
        </div>
      </form>
    </div>
  )
}
```

ToDoAction

ToDoAction se encarga de manejar el estado de las tareas. Para ello, tiene diversos métodos:

- **handleStatus**: este método permite cambiar el estado de una tarea.

```
handleStatus: function(){  
  this.props.setStatus(this.props.index);  
}
```

- **handleEdit**: marca la tarea como “editándose”.

```
handleEdit: function(){  
  this.props.edit(this.props.index);  
}
```

- **handleRemove**: elimina la tarea seleccionada de la lista, tanto si está completa como si no.

```
handleRemove: function(){  
  this.props.remove(this.props.index);  
},
```


ToDoAction

Este componente renderizará los botones de marcar como completado o viceversa, editar y eliminar. Según el botón que se pulse, la función llamará a uno de los métodos descritos anteriormente.

Recordaremos que para asignarle a un botón su manejador de eventos, se hace de la siguiente manera:

```
<button type="button" onClick={this.handleRemove}>  
  ...  
</button>
```

ToDoAction

```
var TaskAction = React.createClass({
  handleStatus: function(){
    this.props.setStatus(this.props.index);
  },
  handleEdit: function(){
    this.props.edit(this.props.index);
  },
  handleRemove: function(){
    this.props.remove(this.props.index);
  },
  getButton: function(){
    if( !this.props.completed ){
      return (
        <button type="button"
          onClick={this.handleStatus}
          className="btn btn-xs btn-success">
            <i className="glyphicon glyphicon-ok"></i>
          </button>
        )
      } else {
        return (
          <button type="button"
            onClick={this.handleStatus}
            className="btn btn-xs btn-info">
              <i className="glyphicon glyphicon-repeat"></i>
            </button>
          )
        }
      }
    },
  },
});
```

```
render: function () {
  return (
    <div className="btn-group btn-group-xs pull-right"
      role="group">
      {this.getButton()}
      <button type="button"
        onClick={this.handleEdit}
        className="btn btn-xs btn-primary" >
        <i className="glyphicon glyphicon-pencil"></i>
      </button>
      <button type="button"
        onClick={this.handleRemove}
        className="btn btn-xs btn-danger" >
        <i className="glyphicon glyphicon-remove"></i>
      </button>
    </div>
  )
}
```

ToDoList

Por último, el componente *ToDoList* simplemente muestra la lista de tareas. Organiza las tareas en una tabla con sus respectivos botones de acción y cambia el color según su estado.

Si la tarea está completada la marcará de color azul. En caso contrario y por defecto, las dejará en verde.

```
var cssClass = 'list-group-item list-group-item-';  
if( task.completed ){  
  cssClass += 'info';  
} else {  
  cssClass += 'success';  
}
```

ToDoList

```
var ToDoList = React.createClass({
  render: function() {
    return (
      <div className="row padded">
        <div className="col-lg-12">
          <h4>Tasks</h4>
          <hr/>
          <ul className="list-group">
            {
              this.props.tasks.map(function(task, index){
                var cssClass = 'list-group-item list-group-item-';
                if( task.completed ){
                  cssClass += 'info';
                } else {
                  cssClass += 'success';
                }
                return (
                  <li key={index} className = {cssClass}>
                    <TaskAction
                      index={index}
                      completed={task.completed}
                      setStatus={this.props.taskAction}
                      edit={this.props.edit}
                      remove={this.props.remove}/>
                    {task.text}
                  </li>
                )
              }, this)
            }
          </ul>
        </div>
      </div>
    )
  });
```

Aplicación TODO

Para poder visualizarlo correctamente, habrá que modificar el archivo `index.html` de manera que cargue los estilos correspondientes:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
  <link href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
rel="stylesheet" type="text/css" />
</head>
<body>
<script src="https://fb.me/react-0.14.7.js"></script>
<script src="https://fb.me/react-dom-0.14.7.js"></script>
  <div id="container"></div>
</body>
</html>
```

Este código únicamente está cargando una plantilla de CSS definida llamada Bootstrap. Lo añadiremos para que la aplicación resulte más agradable.

Aplicación TODO

No olvidemos de modificar el método de renderizar de manera que cargue nuestros componentes:

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

Aplicación TODO

Ya podemos ejecutar la aplicación React y ver cómo funciona la aplicación de tareas. Esta es la aplicación básica para gestionar tareas, la cual cumple con sus funciones principales: añadir, editar y borrar.

En los próximos temas, veremos más formas de realizar este tipo de aplicaciones, mediante el uso de herramientas como **Redux**.