UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Programador Web Inicial

Front End Developer

Administración del estado en React JS

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

Bloques temáticos:

- ¿Por qué utilizar un administrador de estado?
- React Context
- Objeto Context
- Provider
- Consumer
- Actualización del estado mediante Context
- Conclusión

¿Por qué utilizar un administrador de estado?

En una aplicación típica de React, los datos se pasan de arriba hacia abajo (de padre a hijo) a través de *props*, pero esto puede ser complicado para ciertos tipos de *props* (por ejemplo, localización, el tema de la interfaz) que son necesarios para muchos componentes dentro de una aplicación.

Además, a medida que nuestra aplicación vaya creciendo tendremos complejas cadenas de propiedades siendo pasadas de componente a componente.

La idea de utilizar un administrador de estado es solucionar este tipo de problemas, no hay una única solución ya que tenemos distintas librerías (react context, redux, mobx, etc.) para lograrlo y dependiendo la alternativa que elijamos cada una tendrá algunos conceptos particulares y el enfoque puede variar, pero algo que todas van a mantener el principio de única fuente de verdad (SSOT – Single Source Of Truth).

React Context

Context provee una forma de pasar datos a través del árbol de componentes sin tener que pasar *props* manualmente en cada nivel.

Entonces vamos a utilizar context para compartir datos que pueden considerarse “globales” sin la necesidad de pasar *props* a través de elementos intermedios.

Debemos tener en cuenta aplicarlo con moderación porque hace que la reutilización de los componentes sea más difícil.

Consiste en 3 bloques que veremos en detalle:

- Objeto Context
- Provider
- Consumer

Objeto Context

El objeto Context al fin y al cabo es un objeto donde podremos almacenar cualquier tipo de datos, los cuales serán compartidos a otros componentes.

Se puede crear de la siguiente forma para luego proveerlo y consumirlo desde otros componentes.

```
import React from "react";
const myContext = React.createContext({
```

```
    /* Valor inicial*/
});
export default myContext;
```

Provider

Una vez creado el contexto podemos proveer a todos los componentes que necesitaran interactuar con él, es decir, que deberíamos proveerlo en un componente que envuelva todos los niveles inferiores que eventualmente necesiten acceder al contexto.

Un componente proveedor puede estar conectado a muchos consumidores.

```
import React, { Component } from "react";
import myContext from "./Context"
class App extends Component {
    /* Inicializamos un estado*/
    state = {
        usuarios: []
    };
    componentDidMount() {
        /*Guardamos un array de usuarios en el estado*/
        this.setState({ usuarios: ["user1", "user2", "user3"] });
    }
    render() {
        return (
            <myContext.Provider value={{
                state: this.state.usuarios
            }}>
                {/* Componentes de niveles inferiores */}
            </myContext.Provider>);
    }
}
export default App;
```

En el código de ejemplo, se puede ver la creación de un componente `App`, la inicialización de un estado para ese componente y la modificación del estado en el método `componentDidMount` del ciclo de vida del componente.

Dentro del renderizado de nuestro componente vamos a envolver todo con el componente proveedor, en este caso `myContext.Provider`. La propiedad `value` definirá los valores que enviaremos a los componentes hijos, y si el valor cambia, también cambiará en los mismos.

Consumer

El consumidor será un componente que se suscriba a los cambios en el contexto. De esta forma si el contexto cambia, el componente será renderizado nuevamente.

Es importante saber que para poder consumir el contexto no es necesario que el proveedor sea el padre directo.

Podemos realizarlo de 2 formas:

- Utilizando el componente `myContext.Consumer`
  Con este componente vamos a envolver nuestro componente consumidor para poder utilizar el contexto provisto.

```
import React, { Component } from "react";
import myContext from "./Context";
class ChildComponent extends Component {
    render() {
        return (
            <myContext.Consumer>
                {context => (
                    <div>
                        {/* Renderiza algo basado en el contexto */}
                        {context.usuarios}
                    </div>
                )}
            </myContext.Consumer>
        );
    }
}
export default ChildComponent;
```

  Este componente necesita como hijo una función, la cual recibe el valor del contexto actual y devuelve un nodo de React. Cuando nos referimos al valor del contexto actual será igual al `props value` del componente proveedor.

- Utilizando static `contextType`

También podremos acceder al contexto asignando una propiedad estática en nuestra clase. A diferencia del anterior, que puede utilizarse en componentes funcionales o basados en clase, este solo puede utilizarse en componentes basados en clase.

```
import React, { Component } from "react";
import myContext from "./Context";
class ChildComponent extends Component {
    static contextType = myContext;
    render() {
        return (
            <div>
                {/* Renderiza algo basado en el contexto */}
                {this.context.usuarios}
            </div>);
    }
}

export default ChildComponent;
```

La ventaja de utilizar esta forma es que podremos utilizar el contexto en cualquier lugar de nuestro componente, por ejemplo, en el método `componentDidMount`.

Actualización del estado mediante Context

De esta forma tenemos un contexto y lo proveemos a componentes hijos que lo muestran. Pero, ¿qué pasa si queremos modificar esa información? ¿Cómo actualizamos el contexto?

Recordemos que en el Provider teníamos un `props value`, además de enviar nuestro estado, podríamos enviar métodos para modificarlo.

De esta forma nuestro Consume no solo recibiría el estado, sino también los métodos para modificarlo.

```
import React, { Component } from "react";
import myContext from "./Context";
class App extends Component {
    /*Inicializamos un estado*/
    state = {
```

```
        usuarios: []
    };
    componentDidMount() {
        /* Guardamos un arroy de usuarios en el estado */
        this.setState({ usuarios: ["user1", "user2", "user3"] });
    }
    agregarUsuario = () => {
        this.setState({ usuarios: this.state.usuarios.concat(["user4"])
});
    };
    render() {
        return (
            <myContext.Provider
                value={{
                    state: this.state.usuarios,
                    agregarUsuario: this.agregarUsuario
                }}
            > {/* Componentes de niveles inferiores */}
            </myContext.Provider>);
    }
}
export default App;
```

En el ejemplo, creamos un método para agregar un usuario a nuestro estado. Y luego lo enviamos como referencia dentro de `props value` de nuestro Provider. De esta forma el Consumer podrá acceder al método.

```
import React, { Component } from "react";
import myContext from "./Context".
class ChildComponent extends Component {
    static contextType = myContext;
    render() {
        return (
            <div>
                {/* Renderiza algo basado en el contexto */}
                {this.context.usuarios}
                {/* Renderiza un boton para agregar un usuario */}
                <button
onClick={this.context.agregarUsuario.bind(this)}>
```

```
                    Agregar Usuario
                </button>
            </div >);
    }
}
export default ChildComponent;
```

De esta forma podríamos, por ejemplo, crear un botón para que al hacer click, agregue un usuario a nuestro estado.

Conclusión

Con estos conceptos somos capaces de construir una aplicación la cual trabaje con una administración global del estado. Dándonos la ventaja de poder centralizar la información de nuestra aplicación y poder consumirla desde cualquier otro componente.

Bibliografía utilizada y sugerida

https://es.reactjs.org/docs/context.html

https://www.academind.com/learn/react/redux-vs-context-api/