



UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

**Centro de
e-Learning**

Desarrollo en React JS

Centro de e-Learning SCEU UTN - BA.

Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148

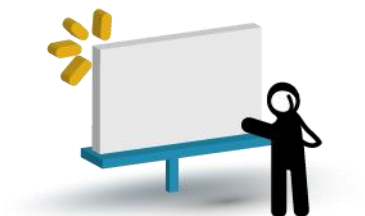
www.sceu.frba.utn.edu.ar/e-learning



Módulo III

Eventos, Firebase

Unidad 4: Administración del estado en react js



Presentación:

En esta unidad aprenderemos como administrarel estado de nuestra aplicación utilizando React Context. De esta forma podremos desarrollar aplicaciones más poderosas.



Objetivos:

Que los participantes*:

- Aprendan que es React Context y su motivación
- Apliquen React Context a sus proyectos
- Investiguen otras alternativas



Bloques temáticos*:

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



•

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.



Tomen nota*

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

¿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 es 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 array 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/>



Lo que vimos:

En esta unidad entendimos la necesidad de un administrador de estado en ciertos casos y como resolverla utilizando React Context.

