Xseed

Curso de React _ Technisys

Xseed

Clase 14 -

React Context, y Testing

---

by Diego Cáceres

Repaso

Formik

Formik nos proporciona un High Order Component llamado withFormik. Esto es una función que recibe un componente y devuelve otro componente.
En particular, este HoC se encarga de manejar el state de nuestro componente formulario, y nos proporciona las siguientes Props:

- values
- errors
- touched
- handleChange
- handleBlur
- handleSubmit
- isSubmitting

Formik

El uso de este HoC es muy simple, simplemente lo importamos de la librería, y le pasamos un objeto de configuración:

```
import React from 'react';
import { withFormik } from 'formik';

const enhancer = withFormik({
 /*... Siguiente slide ...*/
});
const MyForm = props => (
 <div>
   <input  onChange={props.handleChange}  value={props.values.email} />
   <button type="submit" onClick={props.handleSubmit}>
      Submit
   </button>
 </div>
);
export default enhancer(MyForm);
```

REPASO

```
const enhancer = withFormik({
 mapPropsToValues: props => { // Transformamos las props externas a valores
   return {
     email: props.initialEmail || ''
   };
 },
 validate: (values, props) => { // Agregamos nuestra funcion de validacion custom
   const errors = {};
   if (!values.email) {
     errors.email = 'Required';
   } else if (
     !/^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(values.email)
   ) {
     errors.email = 'Invalid email address' ;
   }
   return errors;
 },
 handleSubmit: values => { // realizamos el submit
 },
});
```

Formik

Para poder escribir menos código aun, Formik también nos proporciona dos componentes para utilizar en vez del `<form />` y el `<input />`. La ventaja, es que nos podemos ahorrar pasarle el `handleSubmit` al form, y los `event handlers` y `value` al input.

- <Form />
- <Field /> : Por defecto dibuja un input, pero podemos cambiar el elemento utilizando la prop '`component`', que recibe tanto un string, como '`select`', u otro componente React.
  Este componente recibe en la prop `field` los datos del campo: name, value, onChange, onBlur
  En la prop `form` recibe datos del formulario en sí: values, los setters, y los handlers, la bandera dirty, entre otros.

REPASO

```jsx
import React from 'react';
import { Formik, Form, Field } from 'formik';

const Basic = () => (
<div>
  <h1>My Form</h1>
  <p>This can be anywhere in your application</p>
  <Formik
    initialValues={{
      email: ''
    }}
    validate={ /* igual que el validate del objeto de opciones de withFormik */}
    onSubmit={(
      values,
      { setSubmitting, setErrors } /* setValues y otros */
    ) => { /* realizamos el submit */ }}
    render={ /* Siguiente Slide */}
   />
 </div>
)
export default Basic;
```

Los importamos junto con el componente Formik

REPASO

```
render={({
        values, errors, touched,
        handleChange, handleBlur,
        isSubmitting,
      }) => (
        <Form>
          <Field
            type="email"
            name="email"
          />
          {touched.email && errors.email && <div>{errors.email}</div>}
          <button type="submit" disabled={isSubmitting}>
            Submit
          </button>
        </Form>
      )}
```

No es necesario definir el onSubmit

No es necesario definir el onChange, onBlur, ni darle el value

Xseed

React Context

React Context

En una aplicación típica de React, los datos se pasan de arriba hacia abajo (de padres a hijos) a través de **props**, pero esto puede ser engorroso para ciertos tipos de props (por ejemplo, preferencia de configuración regional, el tema de UI) que requieren muchos componentes dentro de una aplicación.

El contexto proporciona una forma de compartir valores como estos entre los componentes sin tener que pasar explícitamente una prop a través de cada nivel del árbol de componentes.

React Context - Cuándo usarlo?

El contexto está diseñado para compartir datos que se pueden considerar "globales" para un árbol de componentes de React, como el usuario autenticado actual, el tema seleccionado o el idioma preferido.

En el siguiente ejemplo, manualmente pasamos una prop ‘theme’ para darle el estilo correspondiente al componente Button.
El componente “Toolbar” tiene que recibir una **prop** ‘extra’ para pasarsela al “ThemedButton”, y si todos los botones de nuestra aplicación necesitan esto, es una propiedad que tendríamos que pasar por todos los componentes.

Documentación

React Context - Cuándo usarlo?

```
class App extends React.Component {
 render() {
   return <Toolbar theme="dark" />;
 }
}
function Toolbar(props) {
 return (
   <div>
     <ThemedButton theme={props.theme} />
   </div>
 );
}
function ThemedButton(props) {
 return <Button theme={props.theme} />;
}
```

React Context - Cuándo usarlo?

Utilizando el contexto, podemos evitar pasar las **props** por los elementos intermedios.

```
// Creamos un contexto para el tema actual (light como valor por defecto)
const ThemeContext = React.createContext('light');

class App extends React.Component {
 render() {
   // Usamos un Provider para enviar el tema actual al arbol de elementos.
   // No importa en que altura, cualquier componente puede leer el valor.
   return (
     <ThemeContext.Provider value="dark">
       <Toolbar />
     </ThemeContext.Provider>
   );
 }
}
```

React Context - Cuándo usarlo?

```
// El componente en el medio no tiene por que recibir nada
function Toolbar(props) {
 return (
   <div> <ThemedButton /> </div>
 );
}
function ThemedButton(props) {
 // Usamos un Consumer para leer el valor del contexto.
 // React buscara el Provider mas cercano hacia arriba y usar su valor.
 return (
   <ThemeContext.Consumer>
     {theme => <Button {...props} theme={theme} />}
   </ThemeContext.Consumer>
 );
}
```

React Context - Api

React.createContext

Crea un objeto con dos propiedades, Provider y Consumer. Cuando React dibuje un el Consumer de un contexto, leerá el valor del Provider del mismo contexto más cercano hacia arriba en el árbol de componentes.

El valor por defecto solo es utilizado si no encuentra ningún Provider en la jerarquía hacia arriba.

```
const { Provider, Consumer } = React.createContext(defaultValue);
```

React Context - Api

Provider

Es un componente de React que permite que Consumers debajo se puedan subscribir a los cambios en el contexto. Acepta una propiedad value que es la que le dará a los Consumers.
Un Provider puede tener varios Consumers conectados a el, y tambien puede ser anidado debajo de otro Provider para sobreescribir un valor más abajo en el árbol de componentes.

```
<Provider value={/* some value */}>
```

React Context - Api

Consumer

Es un componente de React que se subscribe a los cambios del contexto. Requiere una función como hijo. Esta función es la que recibe el valor del contexto y retorna un nodo React.

Todos los Consumers descendientes de un Provider se re dibujaran cuando el Provider cambie su valor. La propagación de este cambio no depende del método de ciclo de vida shouldComponentUpdate, osea que no se detiene aunque un componente superior no se actualice.

```
<Consumer>
 {value => /* render something based on the context value */}
</Consumer>
```

React Context - Ejemplo 1

theme-context.js

```
export const themes = {
 light: {
   foreground: '#000000',
   background: '#eeeeee',
 },
 dark: {
   foreground: '#ffffff',
   background: '#222222',
 },
};

export const ThemeContext = React.createContext(
 themes.dark // default value
);
```

React Context - Ejemplo 1

themed-button.js

```
import {ThemeContext} from './theme-context';

function ThemedButton(props) {
 return (
   <ThemeContext.Consumer>
     {theme => (
       <button
         {...props}
         style={{backgroundColor: theme.background}}
       />
     )}
   </ThemeContext.Consumer>
 );
}
export default ThemedButton;
```

React Context - Ejemplo 1

app.js

```
export default ThemedButton;

import {ThemeContext, themes} from './theme-context';
import ThemedButton from './themed-button';

// Un componente intermedio, que usa ThemedButton
function Toolbar(props) {
 return (
   <ThemedButton onClick={props.changeTheme}>
     Change Theme
   </ThemedButton>
 );
}
```

```
class App extends React.Component {
 constructor(props) { super(props);
   this.state = { theme: themes.light };
 }
 toggleTheme = () => {
   this.setState(state => ({
     theme: state.theme === themes.dark ? themes.light : themes.dark
   }));
 };
 render() {
   return (
     <div>
       <ThemeContext.Provider  value={this.state.theme}>
         <Toolbar changeTheme={this.toggleTheme} />
       </ThemeContext.Provider>
       <ThemedButton> Another Button </ThemedButton>
     </div>
   );
 }
}
```

React Context - Ejemplo 2

También se puede acceder a varios providers diferentes:

```
const ThemeContext = React.createContext('light');
const UserContext = React.createContext({ name: 'Guest' });
class App extends React.Component {
 render() {
   const {signedInUser, theme} = this.props;
   return (
     <ThemeContext.Provider value={theme}>
       <UserContext.Provider value={signedInUser}>
         <Layout />
       </UserContext.Provider>
     </ThemeContext.Provider>
   );
 }
}
```

```
function Layout() {
 return <div>
     <Sidebar />
     <Content />
   </div>
}
function Content() {
 return (
   <ThemeContext.Consumer>
     {theme => (
       <UserContext.Consumer>
         {user => (
           <ProfilePage user={user} theme={theme} />
         )}
       </UserContext.Consumer>
     )}
   </ThemeContext.Consumer>
 );
}
```

React Context - Ejemplo 3

El acceso a los valores desde el contexto en los métodos del ciclo de vida es un caso de uso relativamente común.

En lugar de agregar contexto a cada método del ciclo de vida, solo tenemos que pasarlo como un **props**, y luego leerlo como cualquier otra **prop**.

React Context - Ejemplo 3

```
class Button extends React.Component {
 componentDidMount() {
   // el valor de ThemeContext esta en this.props.theme
 }
 componentDidUpdate(prevProps, prevState) {
 }
 render() {
   return <button className={props.theme ? 'dark' : 'light'}>
       {props.children}
     </button>
 }
}
export default props => (
 <ThemeContext.Consumer>
   {theme => <Button {...props} theme={theme} />}
 </ThemeContext.Consumer>
);
```

React Context - Ejemplo 4

Algunos datos del contexto se consumen desde varios componentes (por ejemplo, theme o idioma). Puede ser tedioso envolver cada componente en un Consumer. Podemos usar un HoC para solucionar esto.

```
const ThemeContext = React.createContext('light');

function ThemedButton(props) {
 return (
   <ThemeContext.Consumer>
     {theme => <button className={theme} {...props} />}
   </ThemeContext.Consumer>
 );
}
```

React Context - Ejemplo 4

```
const ThemeContext = React.createContext('light');


export function withTheme(Component) {
 // retornamos otro componente
 return function ThemedComponent(props) {
   // que envolvemos con el dato de contexto
   return (
     <ThemeContext.Consumer>
       {theme => <Component {...props} theme={theme} />}
     </ThemeContext.Consumer>
   );
 };
}
```

React Context - Ejemplo 4

Y ahora exportamos cualquier componente que necesite acceso al theme en el HoC que creamos llamado withTheme:

```
function Button({theme, ...rest}) {
 return <button className={theme} {...rest} />;
}

const ThemedButton = withTheme(Button);
```

React Context - Advertencia

Debido a cómo el contexto determina los cambios para re dibujar, hay algunos errores que podrían desencadenar renders involuntarios en los consumidores cuando los padres de un proveedor se de dibujan. Por ejemplo, el siguiente código re dibujara a todos los consumidores cada vez que el Proveedor se actualice porque siempre se crea un nuevo objeto para el valor:

```
class App extends React.Component {
 render() {
   return (
     <Provider value={{something: 'something'}}>
       <Toolbar />
     </Provider>
   );
 }
}
```

React Context - Advertencia

Para solucionar esto, subimos el valor al state del componente.

```
class App extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
     value: {something: 'something'},
   };
 }
 render() {
   return (
     <Provider value={this.state.value}>
       <Toolbar />
     </Provider>
   );
 }
}
```

Testing

Testing

Hay varias alternativas en librerías de Testing para React. La que recomiendan y utilizan dentro de Facebook se llama Jest.

Para utilizarlo, es necesario instalar una dependencia de desarrollo:

```
npm install --save-dev react-test-renderer
```

Si nuestro proyecto no fue creado con ‘create-react-app’ se necesitan algunas dependencias más:

```
npm install --save-dev jest babel-jest babel-preset-env babel-preset-react
```

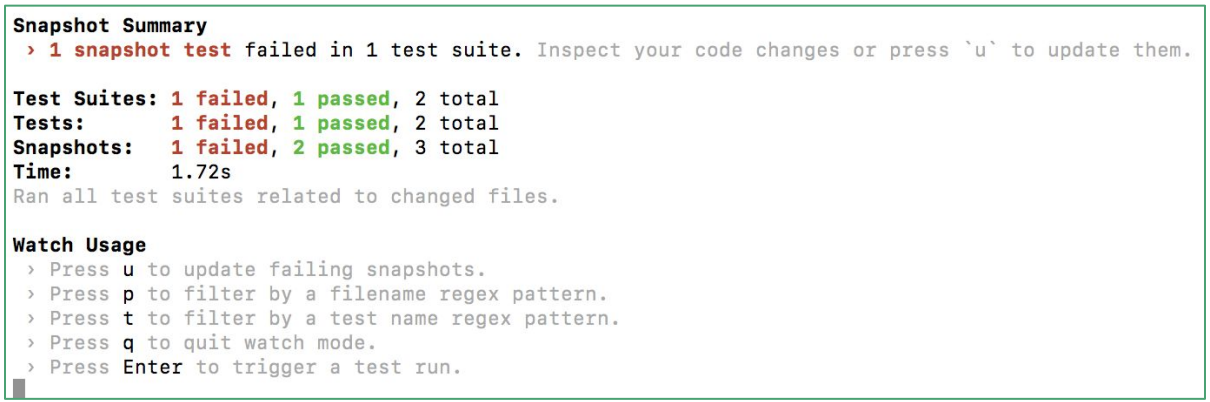
Y crear el archivo .babelrc

```
// .babelrc
{
 "presets": ["env", "react"]
}
```

Testing - Jest

Jest se basa en snapshots. Es decir, captura en un archivo de texto plano como se renderiza el componente la primera vez que se ejecuta el test, y luego siempre compara contra el snapshot anterior para ver si hay errores. Estos snapshots deben ser commiteados en conjunto con el código fuente de la aplicación.

En el caso que la diferencia se de debido a un cambio esperado, se debe actualizar el snapshot corriendo 'jest - u' para sobreescribir el snapshot anterior (o seguir las instrucciones en consola, por ejemplo solo presionando 'u' cuando el test aún está corriendo).



Testing - Ejemplo

```
//Button.js
import React from 'react';

const Button = (props) => {
 return (
   <button
     className='button'
     onClick={props.onPress}
   >
     {props.children}
   </button>
 );
}

export default Button;
```

Testing - Ejemplo

Jest buscará los archivos '.test'. Utilizamos el renderer para dibujar nuestro componente, y en este caso, simplemente nos fijamos que coincida con el Snapshot previo (o si es el primero, lo crea).

```
// Button.test.js
import React from 'react';
import Button from './Button';
import renderer from 'react-test-renderer';

test('Button render', () => {
 const component = renderer.create(
   <Button onPress={() => console.log('Button clicked')}>Click me</Button>,
 );
 let tree = component.toJSON();
 expect(tree).toMatchSnapshot();
});
```

Testing - Ejemplo

El snapshot generado se ve asi:

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`Button render 1`] = `
<button
  className="button"
  onClick={[Function]}
>
  Click me
</button>
`;
```

Testing - Enzyme

Cuando queremos testear la manipulación de nuestros componentes, tenemos dos opciones. Podemos utilizar React Test Utils que nos da unos cuantos helpers para manipular el DOM del elemento simulado, o podemos utilizar Enzyme, una librería de airbnb para este mismo propósito. Esta librería suele estar más completa, y es más popular.

Para instalarlo, debemos instalar la librería y un adapter para la versión de React que estemos utilizando:

```
npm install --save-dev enzyme enzyme-adapter-react-16
```

Y al menos una vez, debemos configurarlo:

```
import Enzyme from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

Enzyme.configure({ adapter: new Adapter() });
```

Testing - Ejemplo

```
export default class CheckboxWithLabel extends React.Component {
 constructor(props) {
   super(props);
   this.state = {isChecked: false};
 }
 onChange = () => {
   this.setState({isChecked: !this.state.isChecked});
 }
 render() {
   return <label>
       <input type="checkbox" checked={this.state.isChecked} onChange={this.onChange} />
       {this.state.isChecked ? this.props.labelOn : this.props.labelOff}
     </label>
 }
}
```

Testing - Ejemplo

```
import React from 'react';
import {shallow} from 'enzyme';
import CheckboxWithLabel from './CheckboxWithLabel';

test('CheckboxWithLabel changes the text after click', () => {
 // Render a checkbox with label in the document
 const checkbox = shallow(<CheckboxWithLabel labelOn="On" labelOff="Off" />);

 expect(checkbox.text()).toEqual('Off');

 checkbox.find('input').simulate('change');

 expect(checkbox.text()).toEqual('On');
});
```

Ejercicios

Ejercicios

Descargar el zip `EjerciciosClase14`. Cada carpeta dentro corresponde a un ejercicio y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

- Si usan yarn:
  - `$ yarn install`
  - `$ yarn start`
- Si usan npm:
  - `$ npm install`
  - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en http://localhost:3000/

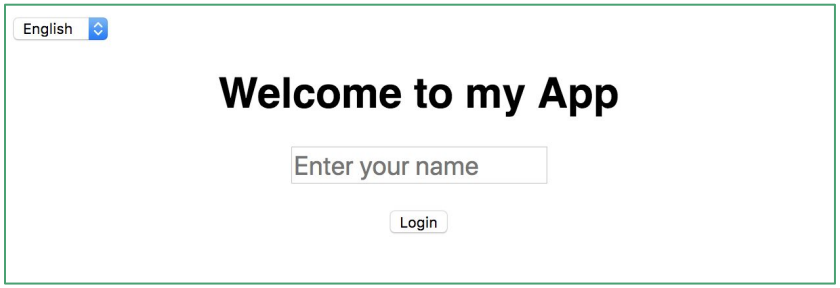
Ejercicio 1

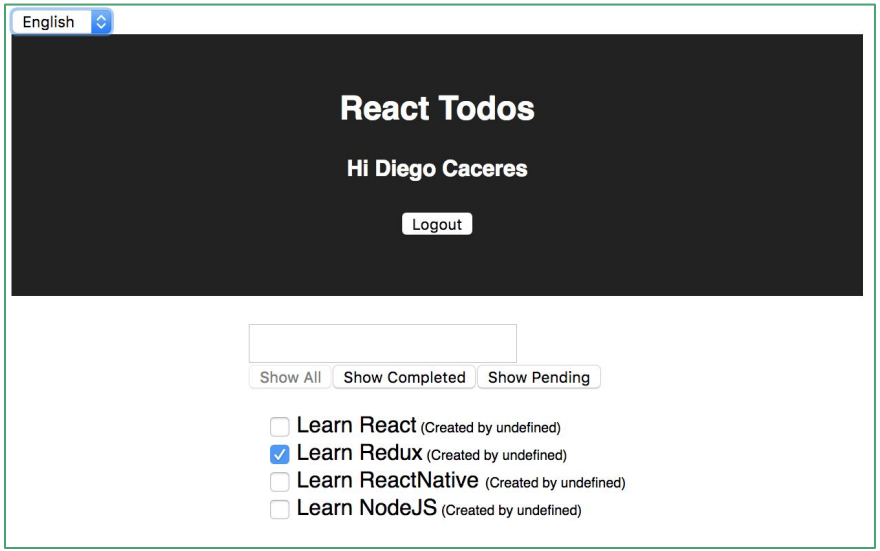
Esta aplicación es la solución de uno de los ejercicios de ToDos.

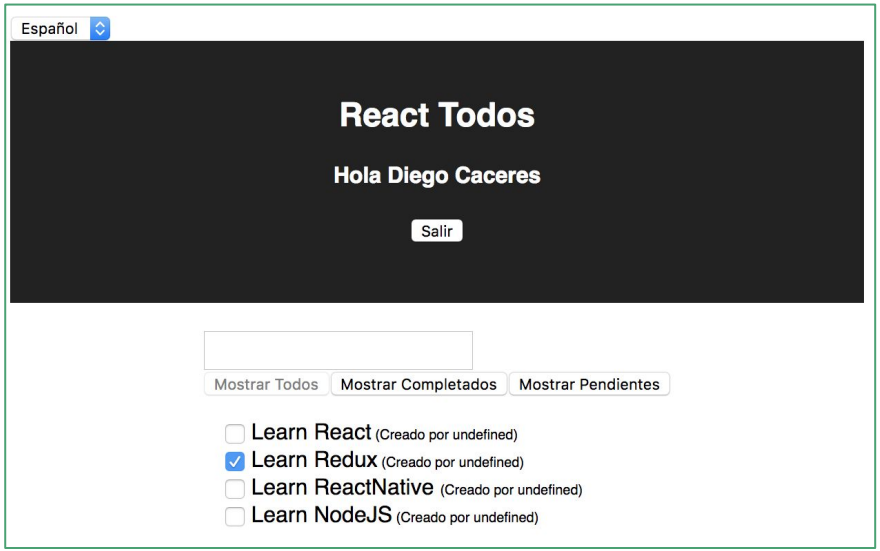
- Se debe crear un LanguageContext para que guarde el valor del lenguaje seleccionado (en un nuevo archivo LanguageContext.js).
- Completar el componente LanguageChanger para que dibuje un Provider que envuelva todos sus hijos.
- Agregar en los componentes necesarios Consumers para leer el valor del language seleccionado y muestre las traducciones correspondientes:
  - Login
  - App
  - TodoList
  - TodoItem

Xseed

Ejercicio 1







Ejercicio 2

Solucionar el ejercicio anterior, pero esta vez creando un HoC llamado `withLanguage` que provea el lenguaje desde el Contexto.

- Exportar desde `LanguageContext.js` el HoC `withLanguage` y utilizarlo desde `App`, `TodoList`, y `TodoItem` para que reciban el language desde props.

- En el caso del componente `Login`, no vamos a usar el HoC, ya que estaríamos envolviendo el `Provider` con el `Consumer`.