Xseed

Curso de React _ Technisys

Xseed

Clase 13 -

Formik

by Diego Cáceres

Repaso

ES6 Generators

Generators

Sin embargo, con **Generators** tenemos un nuevo tipo de funciones, que pueden ser pausados en el medio de su ejecución, y continuadas **luego**, permitiendo que otro código se ejecute en estas pausas. El contexto de la función se mantiene entre las diferentes pausas.

Las funciones **Generators** son Cooperativas, lo que significa que ellas deciden cuándo permitir una interrupción y de esta forma cooperar con el resto del programa. Dentro del cuerpo de la función, podemos utilizar la nueva palabra clave '**yield**' para pausar la función desde dentro. Nada puede pausar un Generator desde fuera, pausa cada vez que encuentra un **yield**.

Sin embargo, una vez pausada, no puede continuar su ejecución sola, sino que necesita que un control externo le indique que debe continuar.

Generators - Como ejecutar un function*

Cuando ejecutamos un Generator, nos devuelve un “Generator Iterator”, un iterador para poder controlar la ejecución de ese Generator, invocando el método next() sobre el. Al llamar a next(), ejecutará la función hasta el primer o siguiente yield.
El next() nos devuelve un objeto con propiedades value y done, que indica si el Generator se terminó de ejecutar:

```
function *foo() {
      yield 1;
      yield 2;
      yield 3;
      yield 4;
      yield 5;
}
```

```
let it = foo();
var message = it.next();
console.log(message); // { value:1, done:false }

console.log( it.next() ); // { value:2, done:false }
console.log( it.next() ); // { value:3, done:false }
console.log( it.next() ); // { value:4, done:false }
console.log( it.next() ); // { value:5, done:false }
// Tecnicamente aun no termino, podriamos pasar un valor a yield 5
console.log( it.next() ); // { value:undefined, done:true }
```

Xseed

REPASO

Generators - Como ejecutar un function*

Cuando retornamos algo desde un Generator, el valor vendrá en value luego de la última interacción:

```
function* foo() {
 let x = yield 'Give me a value for x'
 let y = yield 'Give me a value for y'
 let z = yield 'Give me a value for z'
 return (x + y + z)
}
```

```
let generatingFoo = foo()
//Tenemos el objeto iterable
console.log( generatingFoo.next() )
//{ value: 'Give me a value for x', done: false }
console.log( generatingFoo.next(8) )
//{ value: 'Give me a value for y', done: false }
console.log( generatingFoo.next(4) )
//{ value: 'Give me a value for z', done: false }
console.log( generatingFoo.next(8) )
//{ value: 20, done: true }
```

Xseed

REPASO

Redux Saga

Redux Saga

El Setup es bastante sencillo:

```
import { createStore, applyMiddleware } from 'redux';
import createSagaMiddleware from 'redux-saga'
import reducer from './reducers';
import rootSaga from './sagas';

//Creamos el middleware
const sagaMiddleware = createSagaMiddleware()
//Se lo damos al store
const store = createStore(
 reducer,
 applyMiddleware(sagaMiddleware)
)

// Iniciamos la Saga
sagaMiddleware.run(rootSaga)
```

Redux Saga

En el archivo principal de Sagas, vamos a dividir las Sagas en dos categorías, **Workers** y **Watchers**

- Una saga **Watcher** se encarga de observar todas las acciones emitidas al Store de Redux y si el type coincide con la acción que le corresponde, se la asigna a un Worker Saga. Normalmente desde el archivo de sagas, vamos a exportar una combinación de todos los watchers.

- La **Worker** saga es la que se encarga de procesar esa acción, y realizar todas las demás acciones que sean necesarias.

Redux Saga

```
//En sagas.js
import { types, receiveProducts } from './actions'
import * as api from './api'

//WATCHER SAGA
export function* watchGetProducts() {
     yield takeEvery(types.GET_ALL_PRODUCTS,getAllProducts)
}

//WORKER SAGA
export function* getAllProducts() {
     const products = yield call(api.getProducts)
     yield put(receiveProducts(products))
}
```

takeEvery es un helper y **call** es un Effect Creator de redux-saga

Redux Saga - Effect Creators

El effect **take** es similar al **takeEvery**, pero en vez de disparar siempre una función cuando llega una acción que coincida, nos permite a nosotros decidir qué hacer luego de que llegue la acción, e incluso no escuchar más por esa acción.
En este ejemplo, la aplicación esperaría a que el usuario agregue sus primeros 3 ToDos antes de felicitarlo:

```
import { take, put } from 'redux-saga/effects'

function* watchFirstThreeTodosCreation() {
 for (let i = 0; i < 3; i++) {
   const action = yield take('TODO_CREATED')
 }
 yield put({type: 'SHOW_CONGRATULATION'})
}
```

Xseed

Formik

Forms en React

Como vimos anteriormente, React propone la siguiente solución en su documentación para manejar un formulario, y para 'lidiar' el flujo de datos unidireccional que impone React en sí:

```
handleChange = (event) => {
 this.setState({ value: event.target.value })
}

//...

<input
 onChange={this.handleChange}
 value={this.state.value}
/>
```

Forms en React

Cuando tenemos más de un campo, podemos usar el truco de Computed Properties para usar el name del input:

```
handleChange = (event) => {
 const field = event.target.name;
 this.setState({
   [field]: event.target.value
 })
}
//...
<input name="thing"
 onChange={this.handleChange} value={this.state.thing} />

<input name="other"
 onChange={this.handleChange} value={this.state.other} />
```

Forms en React

React no nos proporciona nada más para solucionar el resto de los desafíos que tienen los formularios generalmente, como:

- Validaciones y mensajes de error.
- Trackeo de cambios (Saber que campos fueron modificados o no).
- Manejar la sumisión del formulario.
- Mantener todo esto organizado, para que si tenemos muchos formularios en nuestra aplicación, todos se resuelvan de la misma forma.

Formik

Formik es una librería muy liviana que nos proporciona una forma sencilla de trabajar con formularios, y con los puntos mencionados anteriormente. Además, soluciona esto sin mover el State del formulario a Redux.

Existe una librería llamada Redux-Form muy popular, pero manejar el State de cada formulario en Redux no es la mejor solución.

- En cada keystroke, estaríamos llamando a todos los reducers de Redux, lo cual trae asociado un pequeño costo en performance, que a medida que la aplicación crece, es cada vez mayor.
- Los datos en sí del formulario no pertenecen conceptualmente al State de toda la aplicación. Una vez completado si, seguramente queramos guardarlo en el State que toda la aplicación puede acceder, pero mientras se completa no.

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

Internamente, Formik sigue utilizando la misma idea de guardar los distintos valores del formulario utilizando una ‘key’, pero además, utiliza la misma key para los errores y para trackear que campos fueron visitados (touched).

Esta es una versión reducida del handleChange interno de Formik:

```
handleChange = (event) => {
 const field = event.target.name;
 this.setState(prevState => ({
   values: {
     ...prevState.values,
     [field]: event.target.value
   }
 })
}
```

Formik

Además de las propiedades que vimos anteriormente, Formik también nos proporciona helpers para tener control total sobre los campos. Esto permite que la integración con otras librerías de terceros, por ejemplo para campos de tipo Select, sea muy sencilla.

- handleChange
- handleSubmit
- handleBlur

- setFieldValue
- setFieldTouched
- setFieldError
- setSubmitting

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);
```

```
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

Veamos un ejemplo básico en funcionamiento: Ejemplo basico

Formik se integra muy fácilmente con Yup (link), que es una librería de validación de objetos. La forma de escribir las validaciones es similar a Prop Types. Si utilizamos esta librería, en vez de darle la opción validate, utilizamos validationSchema.
Ejemplo básico con Yup

```
validationSchema: Yup.object().shape({
 email: Yup.string()
   .email('Invalid email address')
   .required('Email is required!'),
}),
```

Formik

Aparte del High Order Component llamado withFormik, también nos proporciona un Componente <Formik /> que podemos utilizar de una forma similar, con una **‘render’** prop.

En esta prop podemos definir el componente inline, o usar la prop **‘component’** y pasar un componente de un archivo separado, es igual.

- <Formik render={props => <form>...</form>}>
- <Formik component={InnerForm}>

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

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

```
render={({
       values, errors, touched,
       handleChange, handleBlur, handleSubmit,
       isSubmitting,
     }) => (
       <form onSubmit={handleSubmit}>
         <input
           type="email" name="email"
           onChange={handleChange}
           onBlur={handleBlur}
           value={values.email}
         />
         {touched.email && errors.email && <div>{errors.email}</div>}
         <button type="submit" disabled={isSubmitting}>
           Submit
         </button>
       </form>
     )}
```

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.

```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

```
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

Formik

Dos ejemplos más,

- Usando Custom Inputs: link
- Utilizando una librería third-party, como react-select: link

Referencias:

- Charla del creador de Formik: youtube
- Slides de la charla: link

Ejercicios

Ejercicios

Descargar el zip `EjerciciosClase13`. 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`

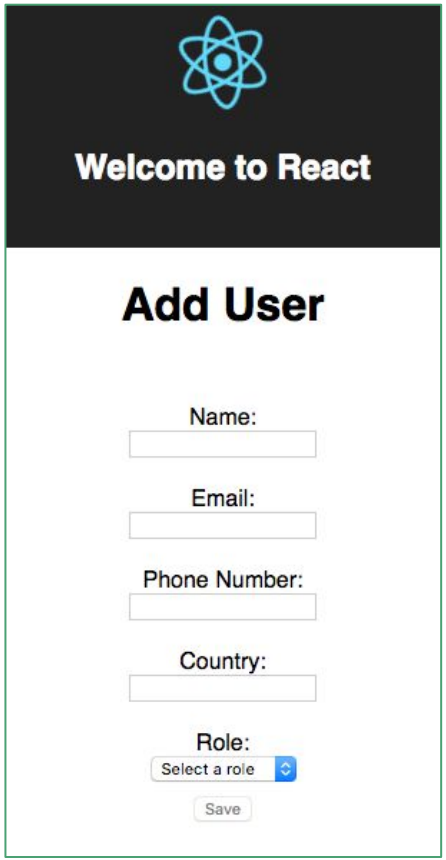
Se debería abrir automáticamente el explorador con el ejercicio corriendo en http://localhost:3000/

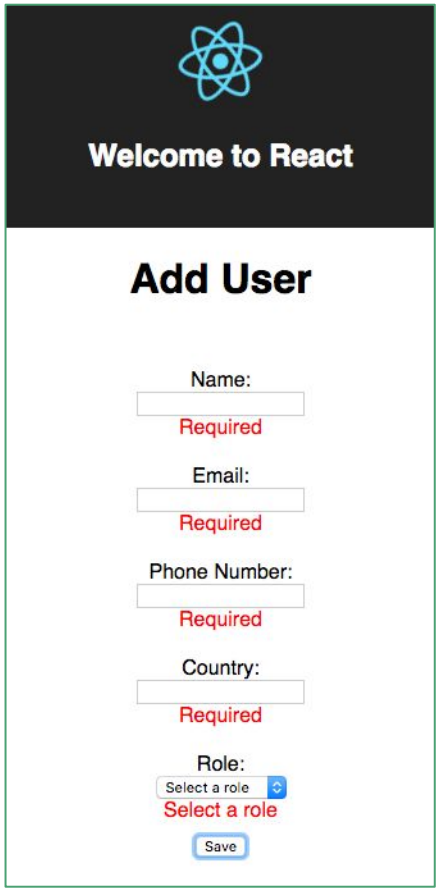
Ejercicio 1

- Completar el componente UserForm utilizando el HoC withFormik. Cada input, label y error, se puede poner dentro de un div con la clase de css ‘`input-field`’
- Agregar validaciones para que ningún campo pueda ser vacío, y también validar el formato del email, y que se haya seleccionado un rol.
- Validar que el largo del país sea mayor a 3.
- Mostrar los errores de cada campo, se puede utilizar la clase de css ‘`text-danger`’.
- El botón debe estar deshabilitado si ningun fue modificado (utilizar la bandera recibida por props: `dirty`), y mientras se esté realizando el submit.
- Agregar en la aplicación que se guarde el usuario en el state de App para mostrar en otro UserForm debajo, el usuario ingresado.

Ejercicio 1

Xseed





Ejercicio 2

Este ejercicio tiene un formulario implementado, que contiene una componente custom llamado '`TextInput`'.

- Hay que completar el componente '`FormikForm`' y '`TextInputFormik`'. Esta vez utilizar el componente `<Formik />` en vez del HoC, `<Field />` y `<Form />`.
- Utilizar el `validationSchema` de `Yup` (ya está instalada) para controlar que el título sea más largo que 3 caracteres, y que si el precio es mayor a 60 muestre un error.
- En el `onSubmit` simplemente por un timeout para mostrar un alert con el objeto ingresado luego de 1 segundo.
- Deshabilitar el botón `Add Game` si hay errores, esta submiteando, o si no se ingresó nada.
- Agregar un botón Reset que limpie el formulario.