**Reaccionar y Redux**

**Comenzando con React Redux**

Esta serie de desafíos presenta cómo usar Redux con React. Primero, aquí hay una revisión de algunos de los principios clave de cada tecnología. React es una biblioteca de vistas a la que usted le proporciona datos y luego representa la vista de una manera eficiente y predecible. Redux es un marco de gestión de estado que puede utilizar para simplificar la gestión del estado de su aplicación. Normalmente, en una aplicación React Redux, crea una única tienda Redux que administra el estado de toda su aplicación. Sus componentes de React se suscriben solo a los datos de la tienda que son relevantes para su función. Luego, envía acciones directamente desde los componentes de React, que luego activan las actualizaciones de la tienda.

Aunque los componentes de React pueden administrar su propio estado localmente, cuando tienes una aplicación compleja, generalmente es mejor mantener el estado de la aplicación en una única ubicación con Redux. Hay excepciones cuando los componentes individuales pueden tener un estado local específico solo para ellos. Finalmente, debido a que Redux no está diseñado para funcionar con React de fábrica, debe usar el react-reduxpaquete. Proporciona una forma de pasar Redux statey dispatchsus componentes de React como props.

Durante los próximos desafíos, primero creará un componente React simple que le permitirá ingresar nuevos mensajes de texto. Estos se agregan a una matriz que se muestra en la vista. Esta debería ser una buena revisión de lo que aprendió en las lecciones de React. A continuación, creará una tienda Redux y acciones que administren el estado de la matriz de mensajes. Finalmente, usará react-reduxpara conectar la tienda Redux con su componente, extrayendo así el estado local en la tienda Redux.

**class DisplayMessages extends React.Component {**

**// change code below this line**

**constructor(props){**

**super(props);**

**this.state={**

**input:'',**

**messages:[]**

**}**

**}**

**// change code above this line**

**render() {**

**return <div />**

**}**

**};**

**Administrar el estado localmente primero**

**Aquí terminará de crear el DisplayMessagescomponente.**

**class DisplayMessages extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: '',**

**messages: []**

**}**

**}**

**// add handleChange() and submitMessage() methods here**

**handleChange(event){**

**this.setState({**

**input: event.target.value,**

**messages: this.state.messages**

**})**

**}**

**submitMessage(){**

**this.setState({**

**input: '',**

**messages: [...this.state.messages, this.state.input]**

**})**

**}**

**render() {**

**return (**

**<div>**

**<h2>Type in a new Message:</h2>**

**{ /\* render an input, button, and ul here \*/ }**

**<input onChange={this.handleChange.bind(this)} value={this.state.input}/>**

**<button onClick={this.submitMessage.bind(this)}>Submit</button>**

**<ul>**

**{this.state.messages.map((x, i)=>{**

**return <li key={i}>{x}</li>**

**})}**

**</ul>**

**{ /\* change code above this line \*/ }**

**</div>**

**);**

**}**

**};**

**Extraer lógica de estado a Redux**

Ahora que terminó el componente React, necesita mover la lógica que está realizando localmente statea Redux. Este es el primer paso para conectar la sencilla aplicación React a Redux. La única funcionalidad que tiene su aplicación es agregar nuevos mensajes del usuario a una lista desordenada. El ejemplo es simple para demostrar cómo funcionan juntos React y Redux.

**// define ADD, addMessage(), messageReducer(), and store here:**

**const ADD = "ADD";**

**const addMessage = message => {**

**return {**

**type: ADD,**

**message**

**};**

**};**

**// Use ES6 default paramter to give the 'previousState' parameter an initial value.**

**const messageReducer = (previousState = [], action) => {**

**// Use switch statement to lay out the reducer logic in response to different action type**

**switch (action.type) {**

**case ADD:**

**// Use ES6 spread operator to return a new array where the new message is added to previousState**

**return [...previousState, action.message];**

**break;**

**default:**

**// A default case to fall back on in case if the update to Redux store is not for this specific state.**

**return previousState;**

**}**

**};**

**const store = Redux.createStore(messageReducer);**

**Utilice el proveedor para conectar Redux y reaccionar**

En el último desafío, creaste una tienda Redux para manejar la matriz de mensajes y creaste una acción para agregar nuevos mensajes. El siguiente paso es proporcionar acceso a React a la tienda Redux y las acciones que necesita para enviar actualizaciones. React Redux proporciona su react-reduxpaquete para ayudar a realizar estas tareas.

React Redux proporciona una pequeña API con dos características clave: Providery connect. Otro desafío cubre connect. Es Providerun componente contenedor de React Redux que envuelve su aplicación React. Este contenedor le permite acceder a Redux storey dispatcha las funciones de todo su árbol de componentes. Providerrequiere dos accesorios, la tienda Redux y los componentes secundarios de su aplicación. La definición Providerde un componente de aplicación podría verse así:

**<Provider store={store}>**

**<App/>**

**</Provider>**

**// Redux:**

**const ADD = 'ADD';**

**const addMessage = (message) => {**

**return {**

**type: ADD,**

**message**

**}**

**};**

**const messageReducer = (state = [], action) => {**

**switch (action.type) {**

**case ADD:**

**return [**

**...state,**

**action.message**

**];**

**default:**

**return state;**

**}**

**};**

**const store = Redux.createStore(messageReducer);**

**// React:**

**class DisplayMessages extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: '',**

**messages: []**

**}**

**this.handleChange = this.handleChange.bind(this);**

**this.submitMessage = this.submitMessage.bind(this);**

**}**

**handleChange(event) {**

**this.setState({**

**input: event.target.value**

**});**

**}**

**submitMessage() {**

**this.setState((state) => {**

**const currentMessage = state.input;**

**return {**

**input: '',**

**messages: state.messages.concat(currentMessage)**

**};**

**});**

**}**

**render() {**

**return (**

**<div>**

**<h2>Type in a new Message:</h2>**

**<input**

**value={this.state.input}**

**onChange={this.handleChange}/><br/>**

**<button onClick={this.submitMessage}>Submit</button>**

**<ul>**

**{this.state.messages.map( (message, idx) => {**

**return (**

**<li key={idx}>{message}</li>**

**)**

**})**

**}**

**</ul>**

**</div>**

**);**

**}**

**};**

**const Provider = ReactRedux.Provider;**

**class AppWrapper extends React.Component {**

**// Render the Provider below this line**

**render() {**

**return (**

**<Provider store={store}>**

**<DisplayMessages />**

**</Provider>**

**);**

**}**

**// Change code above this line**

**};**

**Asignar estado a accesorios**

El Providercomponente le permite proporcionar statey dispatcha sus componentes de React, pero debe especificar exactamente qué estado y acciones desea. De esta manera, te aseguras de que cada componente solo tenga acceso al estado que necesita. Esto se logra creando dos funciones: mapStateToProps()y mapDispatchToProps().

En estas funciones, usted declara a qué partes del estado desea tener acceso y qué creadores de acciones necesita poder enviar. Una vez que estas funciones estén implementadas, verás cómo usar el connectmétodo React Redux para conectarlas a tus componentes en otro desafío.

Nota: Detrás de escena, React Redux usa el store.subscribe()método para implementar mapStateToProps().

**const state = [];**

**// Change code below this line**

**const mapStateToProps = (state)=>{**

**return {**

**messages: state**

**}**

**}**

**Envío de mapas a accesorios**

La mapDispatchToProps()función se utiliza para proporcionar creadores de acciones específicas a sus componentes de React para que puedan enviar acciones contra la tienda Redux. Es similar en estructura a la mapStateToProps()función que escribiste en el último desafío. Devuelve un objeto que asigna acciones de envío a nombres de propiedades, que se convierten en componentes props. Sin embargo, en lugar de devolver una parte de state, cada propiedad devuelve una función que llama dispatchcon un creador de acciones y cualquier dato de acción relevante. Tienes acceso a esto dispatchporque se pasa a mapDispatchToProps()como parámetro cuando defines la función, tal como lo pasaste statea mapStateToProps(). Detrás de escena, React Redux está utilizando Redux store.dispatch()para realizar estos despachos con mapDispatchToProps(). Esto es similar a cómo se usa.store.subscribe()para componentes que están asignados a state.

Por ejemplo, tienes un loginUser()creador de acciones que toma usernameuna carga útil de acción. El objeto devuelto mapDispatchToProps()por este creador de acciones se vería así:

**{**

**submitLoginUser: function(username) {**

**dispatch(loginUser(username));**

**}**

**}**

**const addMessage = (message) => {**

**return {**

**type: 'ADD',**

**message: message**

**}**

**};**

**// change code below this line**

**const mapDispatchToProps = (dispatch) => {**

**return {**

**submitNewMessage: (message)=>{**

**dispatch(addMessage(message))**

**}**

**}**

**}**

**Conecte Redux para reaccionar**

Ahora que ha escrito las funciones mapStateToProps()y mapDispatchToProps(), puede usarlas para asignar statey dispatcha propsuno de sus componentes de React. El connectmétodo de React Redux puede realizar esta tarea. Este método toma dos argumentos opcionales mapStateToProps()y mapDispatchToProps(). Son opcionales porque es posible que tenga un componente al que solo necesita acceso statepero no necesita realizar ninguna acción, o viceversa.

Para usar este método, pase las funciones como argumentos e inmediatamente llame al resultado con su componente. Esta sintaxis es un poco inusual y se parece a:

**connect(mapStateToProps, mapDispatchToProps)(MyComponent)**

Nota: Si desea omitir uno de los argumentos del connectmétodo, páselo nullen su lugar.

**const addMessage = (message) => {**

**return {**

**type: 'ADD',**

**message: message**

**}**

**};**

**const mapStateToProps = (state) => {**

**return {**

**messages: state**

**}**

**};**

**const mapDispatchToProps = (dispatch) => {**

**return {**

**submitNewMessage: (message) => {**

**dispatch(addMessage(message));**

**}**

**}**

**};**

**class Presentational extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**return <h3>This is a Presentational Component</h3>**

**}**

**};**

**const connect = ReactRedux.connect;**

**// change code below this line**

**const ConnectedComponent = connect(mapStateToProps, mapDispatchToProps) (Presentational)**

**Conecte Redux a la aplicación Mensajes**

Ahora que comprende cómo utilizar connectpara conectar React a Redux, puede aplicar lo que ha aprendido a su componente React que maneja mensajes.

En la última lección, el componente que conectó a Redux recibió el nombre Presentationaly esto no fue arbitrario. Este término generalmente se refiere a componentes de React que no están conectados directamente a Redux. Simplemente son responsables de la presentación de la interfaz de usuario y lo hacen en función de los accesorios que reciben. Por el contrario, los componentes del contenedor están conectados a Redux. Por lo general, son responsables de enviar acciones a la tienda y, a menudo, pasan el estado de la tienda a componentes secundarios como accesorios.

**// Redux:**

**const ADD = 'ADD';**

**const addMessage = (message) => {**

**return {**

**type: ADD,**

**message: message**

**}**

**};**

**const messageReducer = (state = [], action) => {**

**switch (action.type) {**

**case ADD:**

**return [**

**...state,**

**action.message**

**];**

**default:**

**return state;**

**}**

**};**

**const store = Redux.createStore(messageReducer);**

**// React:**

**class Presentational extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**input: '',**

**messages: []**

**}**

**this.handleChange = this.handleChange.bind(this);**

**this.submitMessage = this.submitMessage.bind(this);**

**}**

**handleChange(event) {**

**this.setState({**

**input: event.target.value**

**});**

**}**

**submitMessage() {**

**const currentMessage = this.state.input;**

**this.setState({**

**input: '',**

**messages: this.state.messages.concat(currentMessage)**

**});**

**}**

**render() {**

**return (**

**<div>**

**<h2>Type in a new Message:</h2>**

**<input**

**value={this.state.input}**

**onChange={this.handleChange}/><br/>**

**<button onClick={this.submitMessage}>Submit</button>**

**<ul>**

**{this.state.messages.map( (message, idx) => {**

**return (**

**<li key={idx}>{message}</li>**

**)**

**})**

**}**

**</ul>**

**</div>**

**);**

**}**

**};**

**// React-Redux:**

**const mapStateToProps = (state) => {**

**return { messages: state }**

**};**

**const mapDispatchToProps = (dispatch) => {**

**return {**

**submitNewMessage: (newMessage) => {**

**dispatch(addMessage(newMessage))**

**}**

**}**

**};**

**const Provider = ReactRedux.Provider;**

**const connect = ReactRedux.connect;**

**// define the Container component here:**

**const Container = connect(mapStateToProps,mapDispatchToProps)(Presentational)**

**class AppWrapper extends React.Component {**

**constructor(props) {**

**super(props);**

**}**

**render() {**

**// complete the return statement:**

**return (**

**<Provider store={store}>**

**<Container />**

**</Provider>**

**);**

**}**

**};**

**Extraer el estado local en Redux**

¡Ya casi terminas! Recuerde que escribió todo el código de Redux para que Redux pudiera controlar la administración del estado de su aplicación de mensajes React. Ahora que Redux está conectado, debe extraer la administración del estado del Presentationalcomponente y colocarla en Redux. Actualmente, tiene Redux conectado, pero está manejando el estado localmente dentro del Presentationalcomponente.

**// Redux:**

**const ADD = 'ADD';**

**const addMessage = (message) => {**

**return {**

**type: ADD,**

**message: message**

**}**

**};**

**const messageReducer = (state = [], action) => {**

**switch (action.type) {**

**case ADD:**

**return [**

**...state,**

**action.message**

**];**

**default:**

**return state;**

**}**

**};**

**const store = Redux.createStore(messageReducer);**

**// React:**

**const Provider = ReactRedux.Provider;**

**const connect = ReactRedux.connect;**

**// Change code below this line**

**class Presentational extends React.Component {**

**constructor(props) {**

**super(props);**

**// Remove property 'messages' from Presentational's local state**

**this.state = {**

**input: ''**

**}**

**this.handleChange = this.handleChange.bind(this);**

**this.submitMessage = this.submitMessage.bind(this);**

**}**

**handleChange(event) {**

**this.setState({**

**input: event.target.value**

**});**

**}**

**submitMessage() {**

**// Call 'submitNewMessage', which has been mapped to Presentational's props, with a new message;**

**// meanwhile, remove the 'messages' property from the object returned by this.setState().**

**this.props.submitNewMessage(this.state.input);**

**this.setState({**

**input: ''**

**});**

**}**

**render() {**

**return (**

**<div>**

**<h2>Type in a new Message:</h2>**

**<input**

**value={this.state.input}**

**onChange={this.handleChange}/><br/>**

**<button onClick={this.submitMessage}>Submit</button>**

**<ul>**

**{/\* The messages state is mapped to Presentational's props; therefore, when rendering,**

**you should access the messages state through props, instead of Presentational's**

**local state. \*/}**

**{this.props.messages.map( (message, idx) => {**

**return (**

**<li key={idx}>{message}</li>**

**)**

**})**

**}**

**</ul>**

**</div>**

**);**

**}**

**};**

**// Change code above this line**

**const mapStateToProps = (state) => {**

**return {messages: state}**

**};**

**const mapDispatchToProps = (dispatch) => {**

**return {**

**submitNewMessage: (message) => {**

**dispatch(addMessage(message))**

**}**

**}**

**};**

**const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational);**

**class AppWrapper extends React.Component {**

**render() {**

**return (**

**<Provider store={store}>**

**<Container/>**

**</Provider>**

**);**

**}**

**};**

**Avanzando desde aquí**

¡Felicidades! Terminaste las lecciones sobre React y Redux. Hay un último elemento que vale la pena señalar antes de continuar. Normalmente, no escribirás aplicaciones React en un editor de código como este. Este desafío le da una idea de cómo se ve la sintaxis si trabaja con un sistema de archivos en su propia máquina. El código debería verse similar, excepto por el uso de importdeclaraciones (éstas incorporan todas las dependencias que se le han proporcionado en los desafíos).

Finalmente, escribir código de React y Redux generalmente requiere cierta configuración. Esto puede complicarse rápidamente. Si está interesado en experimentar en su propia máquina, la aplicación Create React viene configurada y lista para funcionar.

Alternativamente, puede habilitar Babel como preprocesador de JavaScript en CodePen, agregar React y ReactDOM como recursos externos de JavaScript y trabajar allí también.

**/\***

**import React from 'react'**

**import ReactDOM from 'react-dom'**

**import { Provider, connect } from 'react-redux'**

**import { createStore, combineReducers, applyMiddleware } from 'redux'**

**import thunk from 'redux-thunk'**

**import rootReducer from './redux/reducers'**

**import App from './components/App'**

**const store = createStore(**

**rootReducer,**

**applyMiddleware(thunk)**

**);**

**ReactDOM.render(**

**<Provider store={store}>**

**<App/>**

**</Provider>,**

**document.getElementById('root')**

**);**

**\*/**

**// Only change code below this line**

**console.log('Now I know React and Redux!')**