**Redux**

**Redux es un marco de gestión estatal que se puede utilizar con varias tecnologías web diferentes, incluido React.**

En Redux, hay un único objeto de estado que es responsable de todo el estado de su aplicación. Esto significa que si tuviera una aplicación React con diez componentes y cada componente tuviera su propio estado local, el estado completo de su aplicación estaría definido por un único objeto de estado alojado en Redux store. Este es el primer principio importante que se debe comprender al aprender Redux: la tienda Redux es la única fuente de verdad cuando se trata del estado de la aplicación.

Esto también significa que cada vez que cualquier parte de su aplicación quiera actualizar el estado, debe hacerlo a través de la tienda Redux. El flujo de datos unidireccional facilita el seguimiento de la gestión del estado en su aplicación.

**const reducer = (state = 5) => {**

**return state;**

**}**

**// Redux methods are available from a Redux object**

**// For example: Redux.createStore()**

**// Define the store here:**

**const store = Redux.createStore(reducer)**

**Obtenga el estado de la tienda Redux**

El objeto de la tienda Redux proporciona varios métodos que le permiten interactuar con él. Por ejemplo, puede recuperar el objeto actual stateretenido en el objeto de la tienda Redux con el getState()método.

**const store = Redux.createStore(**

**(state = 5) => state**

**);**

**// Change code below this line**

**let currentState = store.getState();**

**Definir una acción Redux**

Dado que Redux es un marco de gestión estatal, actualizar el estado es una de sus tareas principales. En Redux, todas las actualizaciones de estado se activan mediante acciones de envío. Una acción es simplemente un objeto JavaScript que contiene información sobre un evento de acción que ha ocurrido. La tienda Redux recibe estos objetos de acción y luego actualiza su estado en consecuencia. A veces, una acción de Redux también contiene algunos datos. Por ejemplo, la acción lleva un nombre de usuario después de que un usuario inicia sesión. Si bien los datos son opcionales, las acciones deben tener una typepropiedad que especifique el "tipo" de acción que ocurrió.

Piense en las acciones de Redux como mensajeros que entregan información sobre eventos que suceden en su aplicación a la tienda de Redux. Luego, la tienda realiza la tarea de actualizar el estado en función de la acción que ocurrió.

**let action={**

**type: 'LOGIN'**

**}**

**Definir un creador de acciones**

Después de crear una acción, el siguiente paso es enviar la acción a la tienda Redux para que pueda actualizar su estado. En Redux, defines creadores de acciones para lograr esto. Un creador de acciones es simplemente una función de JavaScript que devuelve una acción. En otras palabras, los creadores de acciones crean objetos que representan eventos de acción.

**const action = {**

**type: 'LOGIN'**

**}**

**// Define an action creator here:**

**function actionCreator() {**

**return action;**

**}**

**Enviar un evento de acción**

dispatchEl método es lo que se utiliza para enviar acciones a la tienda Redux. Llamar store.dispatch()y pasar el valor devuelto por un creador de acciones envía una acción de regreso a la tienda.

Recuerde que los creadores de acciones devuelven un objeto con una propiedad de tipo que especifica el tipo de acción que se ha producido. Luego, el método envía un objeto de acción a la tienda Redux. Según el ejemplo del desafío anterior, las siguientes líneas son equivalentes y ambas envían la acción de tipo LOGIN:

**store.dispatch(actionCreator());**

**store.dispatch({ type: 'LOGIN' });**

**const store = Redux.createStore(**

**(state = {login: false}) => state**

**);**

**const loginAction = () => {**

**return {**

**type: 'LOGIN'**

**}**

**};**

**// Dispatch the action here:**

**store.dispatch(loginAction());**

**Manejar una acción en la tienda**

Después de crear y enviar una acción, la tienda Redux necesita saber cómo responder a esa acción. Este es el trabajo de una reducerfunción. Los reductores en Redux son responsables de las modificaciones de estado que tienen lugar en respuesta a las acciones. A reducertoma statey actioncomo argumentos y siempre devuelve un nuevo state. Es importante ver que ésta es la única función del reductor. No tiene efectos secundarios: nunca llama a un punto final de API y nunca tiene sorpresas ocultas. El reductor es simplemente una función pura que toma estado y acción y luego devuelve un nuevo estado.

Otro principio clave en Redux es que statees de solo lectura. En otras palabras, la reducerfunción siempre debe devolver una nueva copia statey nunca modificar el estado directamente. Redux no impone la inmutabilidad del estado; sin embargo, usted es responsable de imponerla en el código de sus funciones reductoras. Practicarás esto en desafíos posteriores.

**const defaultState = {**

**login: false**

**};**

**const reducer = (state = defaultState, action) => {**

**// Change code below this line**

**if (action.type === "LOGIN") {**

**return {**

**login: true**

**};**

**} else {**

**return state;**

**}**

**// Change code above this line**

**};**

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

**const loginAction = () => {**

**return {**

**type: 'LOGIN'**

**}**

**};**

**Utilice una declaración de cambio para manejar múltiples acciones**

Puede indicarle a la tienda Redux cómo manejar múltiples tipos de acciones. Supongamos que está administrando la autenticación de usuarios en su tienda Redux. Desea tener una representación estatal de cuándo los usuarios inician sesión y cuándo cierran sesión. Representa esto con un objeto de estado único con la propiedad authenticated. También necesita creadores de acciones que creen acciones correspondientes al inicio y cierre de sesión del usuario, junto con los propios objetos de acción.

**const defaultState = {**

**authenticated: false**

**};**

**const authReducer = (state = defaultState, action) => {**

**// Change code below this line**

**switch (action.type) {**

**case "LOGIN":**

**return {**

**authenticated: true**

**};**

**case "LOGOUT":**

**return {**

**authenticated: false**

**};**

**default:**

**return defaultState;**

**}**

**// Change code above this line**

**};**

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

**const loginUser = () => {**

**return {**

**type: 'LOGIN'**

**}**

**};**

**const logoutUser = () => {**

**return {**

**type: 'LOGOUT'**

**}**

**};**

**Utilice const para tipos de acción**

Una práctica común cuando se trabaja con Redux es asignar tipos de acción como constantes de solo lectura y luego hacer referencia a estas constantes dondequiera que se utilicen. Puedes refactorizar el código con el que estás trabajando para escribir los tipos de acciones como constdeclaraciones.

**// change code below this line**

**const LOGIN = 'LOGIN';**

**const LOGOUT = 'LOGOUT';**

**// change code above this line**

**const defaultState = {**

**authenticated: false**

**};**

**const authReducer = (state = defaultState, action) => {**

**switch (action.type) {**

**case LOGIN:**

**return {**

**authenticated: true**

**}**

**case LOGOUT:**

**return {**

**authenticated: false**

**}**

**default:**

**return state;**

**}**

**};**

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

**const loginUser = () => {**

**return {**

**type: LOGIN**

**}**

**};**

**const logoutUser = () => {**

**return {**

**type: LOGOUT**

**}**

**};**

**Registrar un oyente de tienda**

Otro método al que tiene acceso en el storeobjeto Redux es store.subscribe(). Esto le permite suscribir funciones de escucha a la tienda, a las que se llama cada vez que se envía una acción contra la tienda. Un uso sencillo de este método es suscribir una función a su tienda que simplemente registra un mensaje cada vez que se recibe una acción y se actualiza la tienda.

**const ADD = 'ADD';**

**const reducer = (state = 0, action) => {**

**switch(action.type) {**

**case ADD:**

**return state + 1;**

**default:**

**return state;**

**}**

**};**

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

**// Global count variable:**

**let count = 0;**

**// Change code below this line**

**const add = () => count++;**

**store.subscribe(add);**

**// Change code above this line**

**store.dispatch({type: ADD});**

**console.log(count);**

**store.dispatch({type: ADD});**

**console.log(count);**

**store.dispatch({type: ADD});**

**console.log(count);**

**Combinar múltiples reductores**

Cuando el estado de su aplicación comienza a volverse más complejo, puede resultar tentador dividir el estado en varias partes. En su lugar, recuerde el primer principio de Redux: todo el estado de la aplicación se mantiene en un único objeto de estado en la tienda. Por lo tanto, Redux proporciona una composición reductora como solución para un modelo de estado complejo. Usted define múltiples reductores para manejar diferentes partes del estado de su aplicación y luego compone estos reductores juntos en un reductor raíz. Luego, el reductor raíz se pasa al createStore()método Redux.

Para permitirnos combinar varios reductores, Redux proporciona el combineReducers()método. Este método acepta un objeto como argumento en el que se definen propiedades que asocian claves a funciones reductoras específicas. Redux utilizará el nombre que le dé a las claves como nombre del estado asociado.

Normalmente, es una buena práctica crear un reductor para cada estado de la aplicación cuando son distintos o únicos de alguna manera. Por ejemplo, en una aplicación para tomar notas con autenticación de usuario, un reductor podría manejar la autenticación mientras que otro maneja el texto y las notas que envía el usuario. Para tal aplicación, podríamos escribir el combineReducers()método así:

**const rootReducer = Redux.combineReducers({**

**auth: authenticationReducer,**

**notes: notesReducer**

**});**

Ahora, la clave notescontendrá todo el estado asociado con nuestras notas y manejado por nuestro notesReducer. Así es como se pueden componer múltiples reductores para gestionar estados de aplicaciones más complejos. En este ejemplo, el estado mantenido en la tienda Redux sería un único objeto que contiene propiedades authy notes.

**const INCREMENT = 'INCREMENT';**

**const DECREMENT = 'DECREMENT';**

**const counterReducer = (state = 0, action) => {**

**switch(action.type) {**

**case INCREMENT:**

**return state + 1;**

**case DECREMENT:**

**return state - 1;**

**default:**

**return state;**

**}**

**};**

**const LOGIN = 'LOGIN';**

**const LOGOUT = 'LOGOUT';**

**const authReducer = (state = {authenticated: false}, action) => {**

**switch(action.type) {**

**case LOGIN:**

**return {**

**authenticated: true**

**}**

**case LOGOUT:**

**return {**

**authenticated: false**

**}**

**default:**

**return state;**

**}**

**};**

**const rootReducer = Redux.combineReducers({**

**count: counterReducer,**

**auth: authReducer**

**});**

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

**Enviar datos de acción a la tienda**

A estas alturas ya has aprendido cómo enviar acciones a la tienda Redux, pero hasta ahora estas acciones no contienen ninguna información más que un archivo type. También puedes enviar datos específicos junto con tus acciones. De hecho, esto es muy común porque las acciones generalmente se originan a partir de alguna interacción del usuario y tienden a llevar consigo algunos datos. La tienda Redux a menudo necesita conocer estos datos.

**const ADD\_NOTE = 'ADD\_NOTE';**

**const notesReducer = (state = 'Initial State', action) => {**

**switch(action.type) {**

**// Change code below this line**

**case ADD\_NOTE:**

**return action.text;**

**// Change code above this line**

**default:**

**return state;**

**}**

**};**

**const addNoteText = (note) => {**

**// Change code below this line**

**return{**

**type: ADD\_NOTE,**

**text: note**

**};**

**// Change code above this line**

**};**

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

**console.log(store.getState());**

**store.dispatch(addNoteText('Hello!'));**

**console.log(store.getState());**

**Utilice middleware para manejar acciones asincrónicas**

Hasta ahora estos desafíos han evitado discutir acciones asincrónicas, pero son una parte inevitable del desarrollo web. En algún momento necesitarás llamar a puntos finales asincrónicos en tu aplicación Redux, entonces, ¿cómo manejas este tipo de solicitudes? Redux proporciona middleware diseñado específicamente para este propósito, llamado middleware Redux Thunk. Aquí hay una breve descripción de cómo usar esto con Redux.

Para incluir middleware Redux Thunk, lo pasa como argumento a Redux.applyMiddleware(). Luego, esta declaración se proporciona como un segundo parámetro opcional de la createStore()función. Eche un vistazo al código en la parte inferior del editor para ver esto. Luego, para crear una acción asincrónica, devuelve una función en el creador de la acción que toma dispatchcomo argumento. Dentro de esta función, puede enviar acciones y realizar solicitudes asincrónicas.

En este ejemplo, se simula una solicitud asincrónica con una setTimeout()llamada. Es común enviar una acción antes de iniciar cualquier comportamiento asincrónico para que el estado de su aplicación sepa que se están solicitando algunos datos (este estado podría mostrar un ícono de carga, por ejemplo). Luego, una vez que recibe los datos, envía otra acción que lleva los datos como una carga útil junto con la información de que la acción se completó.

Recuerda que estás pasando dispatchcomo parámetro a este creador de acciones especiales. Esto es lo que usará para enviar sus acciones; simplemente pasa la acción directamente al envío y el middleware se encarga del resto.

**const REQUESTING\_DATA = "REQUESTING\_DATA";**

**const RECEIVED\_DATA = "RECEIVED\_DATA";**

**const requestingData = () => {**

**return { type: REQUESTING\_DATA };**

**};**

**const receivedData = data => {**

**return { type: RECEIVED\_DATA, users: data.users };**

**};**

**const handleAsync = () => {**

**return function(dispatch) {**

**// dispatch request action here**

**dispatch(requestingData());**

**setTimeout(function() {**

**let data = {**

**users: ["Jeff", "William", "Alice"]**

**};**

**// dispatch received data action here**

**dispatch(receivedData(data));**

**}, 2500);**

**};**

**};**

**const defaultState = {**

**fetching: false,**

**users: []**

**};**

**const asyncDataReducer = (state = defaultState, action) => {**

**switch (action.type) {**

**case REQUESTING\_DATA:**

**return {**

**fetching: true,**

**users: []**

**};**

**case RECEIVED\_DATA:**

**return {**

**fetching: false,**

**users: action.users**

**};**

**default:**

**return state;**

**}**

**};**

**const store = Redux.createStore(**

**asyncDataReducer,**

**Redux.applyMiddleware(ReduxThunk.default)**

**);**

**Escribir un contador con Redux**

¡Ahora ha aprendido todos los principios básicos de Redux! Ha visto cómo crear acciones y creadores de acciones, crear una tienda Redux, enviar sus acciones a la tienda y diseñar actualizaciones de estado con reductores puros. Incluso has visto cómo gestionar estados complejos con composición reductora y manejar acciones asincrónicas. Estos ejemplos son simplistas, pero estos conceptos son los principios básicos de Redux. Si los comprende bien, estará listo para comenzar a crear su propia aplicación Redux. Los próximos desafíos cubren algunos de los detalles relacionados con statela inmutabilidad, pero primero, aquí hay una revisión de todo lo que has aprendido hasta ahora.

**const INCREMENT = "INCREMENT"; // define a constant for increment action types**

**const DECREMENT = "DECREMENT"; // define a constant for decrement action types**

**// define the counter reducer which will increment or decrement the state based on the action it receives**

**const counterReducer = (state = 0, action) => {**

**switch (action.type) {**

**case INCREMENT:**

**return state + 1;**

**case DECREMENT:**

**return state - 1;**

**default:**

**return state;**

**}**

**};**

**// define an action creator for incrementing**

**const incAction = () => {**

**return {**

**type: INCREMENT**

**};**

**};**

**// define an action creator for decrementing**

**const decAction = () => {**

**return {**

**type: DECREMENT**

**};**

**};**

**// define the Redux store here, passing in your reducers**

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

**Estado nunca mutar**

Estos desafíos finales describen varios métodos para hacer cumplir el principio clave de inmutabilidad del estado en Redux. Estado inmutable significa que nunca modifica el estado directamente, sino que devuelve una nueva copia del estado.

Si tomara una instantánea del estado de una aplicación Redux a lo largo del tiempo, vería algo como state 1,,,, etc. , donde cada estado puede ser similar al anterior, pero cada state 2uno es un dato distinto. Esta inmutabilidad, de hecho, es lo que proporciona funciones como la depuración de viajes en el tiempo de las que quizás haya oído hablar.state 3state 4...

Redux no aplica activamente la inmutabilidad del estado en su tienda o reductores, esa responsabilidad recae en el programador. Afortunadamente, JavaScript (especialmente ES6) proporciona varias herramientas útiles que puede utilizar para imponer la inmutabilidad de su estado, ya sea string, number, arrayo object. Tenga en cuenta que las cadenas y los números son valores primitivos y son inmutables por naturaleza. En otras palabras, 3 es siempre 3. No puedes cambiar el valor del número 3. Sin embargo, un arrayo es mutable. objectEn la práctica, su estado probablemente consistirá en un arrayo object, ya que son estructuras de datos útiles para representar muchos tipos de información.

**const ADD\_TO\_DO = "ADD\_TO\_DO";**

**// A list of strings representing tasks to do:**

**const todos = [**

**"Go to the store",**

**"Clean the house",**

**"Cook dinner",**

**"Learn to code"**

**];**

**const immutableReducer = (state = todos, action) => {**

**switch (action.type) {**

**case ADD\_TO\_DO:**

**// don't mutate state here or the tests will fail**

**return state.concat(action.todo);**

**// or return [...state, action.todo]**

**default:**

**return state;**

**}**

**};**

**// an example todo argument would be 'Learn React',**

**const addToDo = todo => {**

**return {**

**type: ADD\_TO\_DO,**

**todo**

**};**

**};**

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

**Utilice el operador de extensión en matrices**

Una solución de ES6 para ayudar a imponer la inmutabilidad del estado en Redux es el operador de extensión: .... El operador de extensión tiene una variedad de aplicaciones, una de las cuales se adapta bien al desafío anterior de producir una nueva matriz a partir de una matriz existente. Esta es una sintaxis relativamente nueva, pero de uso común. Por ejemplo, si tienes una matriz myArrayy escribes:

**let newArray = [...myArray];**

newArrayahora es un clon de myArray. Ambas matrices todavía existen por separado en la memoria. Si realiza una mutación como newArray.push(5), myArrayno cambia. ...Distribuye efectivamente los valores en myArrayuna nueva matriz. Para clonar una matriz pero agregar valores adicionales en la nueva matriz, puede escribir [...myArray, 'new value']. Esto devolvería una nueva matriz compuesta por los valores myArrayy la cadena new valuecomo último valor. La sintaxis extendida se puede usar varias veces en una composición de matriz como esta, pero es importante tener en cuenta que solo hace una copia superficial de la matriz. Es decir, solo proporciona operaciones de matriz inmutables para matrices unidimensionales.

**const immutableReducer = (state = ["Do not mutate state!"], action) => {**

**switch (action.type) {**

**case "ADD\_TO\_DO":**

**// don't mutate state here or the tests will fail**

**let arr = [...state, action.todo];**

**return arr;**

**default:**

**return state;**

**}**

**};**

**const addToDo = todo => {**

**return {**

**type: "ADD\_TO\_DO",**

**todo**

**};**

**};**

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

**Eliminar un elemento de una matriz**

Es hora de practicar la eliminación de elementos de una matriz. El operador de extensión también se puede utilizar aquí. Otros métodos útiles de JavaScript incluyen slice()y concat().

**const immutableReducer = (state = [0,1,2,3,4,5], action) => {**

**switch(action.type) {**

**case 'REMOVE\_ITEM':**

**// Don't mutate state here or the tests will fail**

**return [**

**...state.slice(0, action.index),**

**...state.slice(action.index + 1, state.length)**

**]**

**default:**

**return state;**

**}**

**};**

**const removeItem = (index) => {**

**return {**

**type: 'REMOVE\_ITEM',**

**index**

**}**

**}**

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

**Copiar un objeto con Object.assign**

Los últimos desafíos funcionaron con matrices, pero también hay formas de ayudar a imponer la inmutabilidad del estado cuando el estado es un object. Una herramienta útil para manipular objetos es la Object.assign()utilidad. Object.assign()toma un objeto de destino y objetos de origen y asigna propiedades de los objetos de origen al objeto de destino. Cualquier propiedad coincidente se sobrescribe con las propiedades de los objetos de origen. Este comportamiento se usa comúnmente para hacer copias superficiales de objetos pasando un objeto vacío como primer argumento seguido del objeto(s) que desea copiar. He aquí un ejemplo:

**const newObject = Object.assign({}, obj1, obj2);**

Esto se crea newObjectcomo un nuevo object, que contiene las propiedades que existen actualmente en obj1y obj2.

**const defaultState = {**

**user: "CamperBot",**

**status: "offline",**

**friends: "732,982",**

**community: "freeCodeCamp"**

**};**

**const immutableReducer = (state = defaultState, action) => {**

**switch (action.type) {**

**case "ONLINE":**

**// to enforce state immutability, return a new state object using Object.assign() method**

**return Object.assign({}, state, { status: "online" });**

**default:**

**return state;**

**}**

**};**

**const wakeUp = () => {**

**return {**

**type: "ONLINE"**

**};**

**};**

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