**Redux**

# **Create a Redux Store**

Redux is a state management framework that can be used with a number of different web technologies, including React.

In Redux, there is a single state object that's responsible for the entire state of your application. This means if you had a React app with ten components, and each component had its own local state, the entire state of your app would be defined by a single state object housed in the Redux store. This is the first important principle to understand when learning Redux: the Redux store is the single source of truth when it comes to application state.

This also means that any time any piece of your app wants to update state, it **must** do so through the Redux store. The unidirectional data flow makes it easier to track state management in your app.

*The Redux store is an object which holds and manages application state. There is a method called createStore() on the Redux object, which you use to create the Redux store. This method takes a reducer function as a required argument. The reducer function is covered in a later challenge, and is already defined for you in the code editor. It simply takes state as an argument and returns state.*

*Declare a store variable and assign it to the createStore() method, passing in the reducer as an argument.*

***Note:****The code in the editor uses ES6 default argument syntax to initialize this state to hold a value of 5.*

**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);**

# **Get State from the Redux Store**

The Redux store object provides several methods that allow you to interact with it. For example, you can retrieve the current state held in the Redux store object with the getState() method.

*The code from the previous challenge is re-written more concisely in the code editor. Use store.getState() to retrieve the state from the store, and assign this to a new variable currentState.*

**const store = Redux.createStore(**

**(state = 5) => state**

**);**

// Change code below this line

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

# **Define a Redux Action**

Since Redux is a state management framework, updating state is one of its core tasks. In Redux, all state updates are triggered by dispatching actions. An action is simply a JavaScript object that contains information about an action event that has occurred. The Redux store receives these action objects, then updates its state accordingly. Sometimes a Redux action also carries some data. For example, the action carries a username after a user logs in. While the data is optional, actions must carry a type property that specifies the 'type' of action that occurred.

Think of Redux actions as messengers that deliver information about events happening in your app to the Redux store. The store then conducts the business of updating state based on the action that occurred.

*Writing a Redux action is as simple as declaring an object with a type property. Declare an object action and give it a property type set to the string 'LOGIN'.*

// Define an action here:

**const action={**

**type:'LOGIN'**

**}**

# **Define an Action Creator**

After creating an action, the next step is sending the action to the Redux store so it can update its state. In Redux, you define action creators to accomplish this. An action creator is simply a JavaScript function that returns an action. In other words, action creators create objects that represent action events.

*Define a function named actionCreator() that returns the action object when called.*

**const action = {**

**type: 'LOGIN'**

**}**

// Define an action creator here:

**const actionCreator=()=>{**

**return action**

**}**

# **Dispatch an Action Event**

dispatch method is what you use to dispatch actions to the Redux store. Calling store.dispatch() and passing the value returned from an action creator sends an action back to the store.

Recall that action creators return an object with a type property that specifies the action that has occurred. Then the method dispatches an action object to the Redux store. Based on the previous challenge's example, the following lines are equivalent, and both dispatch the action of type LOGIN:

store.dispatch(actionCreator());

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

*The Redux store in the code editor has an initialized state that's an object containing a login property currently set to false. There's also an action creator called loginAction() which returns an action of type LOGIN. Dispatch the LOGIN action to the Redux store by calling the dispatch method, and pass in the action created by loginAction().*

**const store = Redux.createStore(**

**(state = {login: false}) => state**

**);**

**const loginAction = () => {**

**return {**

**type: 'LOGIN'**

**}**

**};**

// Dispatch the action here:

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

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

# **Handle an Action in the Store**

After an action is created and dispatched, the Redux store needs to know how to respond to that action. This is the job of a reducer function. Reducers in Redux are responsible for the state modifications that take place in response to actions. A reducer takes state and action as arguments, and it always returns a new state. It is important to see that this is the **only** role of the reducer. It has no side effects — it never calls an API endpoint and it never has any hidden surprises. The reducer is simply a pure function that takes state and action, then returns new state.

Another key principle in Redux is that state is read-only. In other words, the reducer function must **always** return a new copy of state and never modify state directly. Redux does not enforce state immutability, however, you are responsible for enforcing it in the code of your reducer functions. You'll practice this in later challenges.

The code editor has the previous example as well as the start of a reducer function for you. Fill in the body of the reducer function so that if it receives an action of type 'LOGIN' it returns a state object with login set to true. Otherwise, it returns the current state. Note that the current state and the dispatched action are passed to the reducer, so you can access the action's type directly with action.type.

**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'**

**}**

**};**

# **Use a Switch Statement to Handle Multiple Actions**

You can tell the Redux store how to handle multiple action types. Say you are managing user authentication in your Redux store. You want to have a state representation for when users are logged in and when they are logged out. You represent this with a single state object with the property authenticated. You also need action creators that create actions corresponding to user login and user logout, along with the action objects themselves.

*The code editor has a store, actions, and action creators set up for you. Fill in the reducer function to handle multiple authentication actions. Use a JavaScript switch statement in the reducer to respond to different action events. This is a standard pattern in writing Redux reducers. The switch statement should switch over action.type and return the appropriate authentication state.*

**Note:** *At this point, don't worry about state immutability, since it is small and simple in this example. For each action, you can return a new object — for example, {authenticated: true}. Also, don't forget to write a default case in your switch statement that returns the current state. This is important because once your app has multiple reducers, they are all run any time an action dispatch is made, even when the action isn't related to that reducer. In such a case, you want to make sure that you return the current state.*

**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'**

**}**

**};**

# **Use const for Action Types**

A common practice when working with Redux is to assign action types as read-only constants, then reference these constants wherever they are used. You can refactor the code you're working with to write the action types as const declarations.

*Declare LOGIN and LOGOUT as const values and assign them to the strings 'LOGIN' and 'LOGOUT', respectively. Then, edit the authReducer() and the action creators to reference these constants instead of string values.*

**Note:***It's generally a convention to write constants in all uppercase, and this is standard practice in Redux as well.*

// 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**

**}**

**};**

# **Register a Store Listener**

Another method you have access to on the Redux store object is store.subscribe(). This allows you to subscribe listener functions to the store, which are called whenever an action is dispatched against the store. One simple use for this method is to subscribe a function to your store that simply logs a message every time an action is received and the store is updated.

Write a callback function that increments the global variable count every time the store receives an action, and pass this function in to the store.subscribe() method. You'll see that store.dispatch() is called three times in a row, each time directly passing in an action object. Watch the console output between the action dispatches to see the updates take place.

**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 addOne = () => (count += 1);**

**function sayHi() {**

**console.log("Hi!");**

**addOne();**

**}**

**store.subscribe(sayHi)**

// 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);**

# **Combine Multiple Reducers**

When the state of your app begins to grow more complex, it may be tempting to divide state into multiple pieces. Instead, remember the first principle of Redux: all app state is held in a single state object in the store. Therefore, Redux provides reducer composition as a solution for a complex state model. You define multiple reducers to handle different pieces of your application's state, then compose these reducers together into one root reducer. The root reducer is then passed into the Redux createStore() method.

In order to let us combine multiple reducers together, Redux provides the combineReducers() method. This method accepts an object as an argument in which you define properties which associate keys to specific reducer functions. The name you give to the keys will be used by Redux as the name for the associated piece of state.

Typically, it is a good practice to create a reducer for each piece of application state when they are distinct or unique in some way. For example, in a note-taking app with user authentication, one reducer could handle authentication while another handles the text and notes that the user is submitting. For such an application, we might write the combineReducers() method like this:

const rootReducer = Redux.combineReducers({

auth: authenticationReducer,

notes: notesReducer

});

Now, the key notes will contain all of the state associated with our notes and handled by our notesReducer. This is how multiple reducers can be composed to manage more complex application state. In this example, the state held in the Redux store would then be a single object containing auth and notes properties.

*There are counterReducer() and authReducer() functions provided in the code editor, along with a Redux store. Finish writing the rootReducer() function using the Redux.combineReducers() method. Assign counterReducer to a key called count and authReducer to a key called auth.*

**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**

**});** // Define the root reducer here

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