Front End Development Libraries

# Redux

## Introduction

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.

# Format:

## Syntax:

const reducer = (state = 5) => {

return state;

const store = Redux.createStore(reducer);

**OR**

const store = Redux.createStore(

(state = 5) => state

);

const currentState = store.getState();

# Topics Covered:

## Create a Redux Store

### Example:

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

### Explanation:

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

### Example:

the getState() method.

const store = Redux.createStore(

(state = 5) => state

);

// Change code below this line

const currentState = store.getState();

## Define a Redux Action

### Explanation:

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.

### Example:

// Define an action here:

let action = {

type: 'LOGIN'

}

## Define an Action Creator

### Explanation

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.

### Example

const action = {

  type: 'LOGIN'

}

// Define an action creator here:

const actionCreator = () => {

  return action;

  };

## Dispatch an Action Event

### Explanation:

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' });

### Example:

const store = Redux.createStore(

  (state = {login: false}) => state

);

const loginAction = () => {

  return {

    type: 'LOGIN'

  }

};

// Dispatch the action here:

store.dispatch(loginAction());

## Handle an Action in the Store

### Explanation:

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.

### Example:

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

### Explanation:

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.

### Example:

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{

        authenticated: false

        };

    };

  // 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

### Explanation:

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.

### Example:

const LOGIN = 'LOGIN';

const LOGOUT = 'LOGOUT';

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

### Explanation:

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.

### Example:

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

store.subscribe(()=>{count++});

// 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

## Explanation:

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.

### Example:

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);

## Send Action Data to the Store

### Explanation:

By now you've learned how to dispatch actions to the Redux store, but so far these actions have not contained any information other than a type. You can also send specific data along with your actions. In fact, this is very common because actions usually originate from some user interaction and tend to carry some data with them. The Redux store often needs to know about this data.

### Example:

const ADD\_NOTE = 'ADD\_NOTE';

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

  switch(action.type) {

    // Change code below this line

    case ADD\_NOTE:

      return state = 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());

## Use Middleware to Handle Asynchronous Actions

### Explanation:

So far these challenges have avoided discussing asynchronous actions, but they are an unavoidable part of web development. At some point you'll need to call asynchronous endpoints in your Redux app, so how do you handle these types of requests? Redux provides middleware designed specifically for this purpose, called Redux Thunk middleware. Here's a brief description how to use this with Redux.

To include Redux Thunk middleware, you pass it as an argument to Redux.applyMiddleware(). This statement is then provided as a second optional parameter to the createStore() function. Take a look at the code at the bottom of the editor to see this. Then, to create an asynchronous action, you return a function in the action creator that takes dispatch as an argument. Within this function, you can dispatch actions and perform asynchronous requests.

In this example, an asynchronous request is simulated with a setTimeout() call. It's common to dispatch an action before initiating any asynchronous behavior so that your application state knows that some data is being requested (this state could display a loading icon, for instance). Then, once you receive the data, you dispatch another action which carries the data as a payload along with information that the action is completed.

Remember that you're passing dispatch as a parameter to this special action creator. This is what you'll use to dispatch your actions, you simply pass the action directly to dispatch and the middleware takes care of the rest.

### Example:

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)

);

## Write a Counter with Redux

### Explanation:

Now you've learned all the core principles of Redux! You've seen how to create actions and action creators, create a Redux store, dispatch your actions against the store, and design state updates with pure reducers. You've even seen how to manage complex state with reducer composition and handle asynchronous actions. These examples are simplistic, but these concepts are the core principles of Redux. If you understand them well, you're ready to start building your own Redux app. The next challenges cover some of the details regarding state immutability, but first, here's a review of everything you've learned so far.

### Example:

const INCREMENT = 'INCREMENT'; // Define a constant for increment action types

const DECREMENT = 'DECREMENT'; // Define a constant for decrement action types

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

  switch(action.type){

    case INCREMENT:

    return state + 1;

    case DECREMENT:

    return state - 1;

    default:

    return state;

  }

}; // Define the counter reducer which will increment or decrement the state based on the action it receives

const incAction = () =>{

  return {type: INCREMENT};

} // Define an action creator for incrementing

const decAction = () =>{

  return {type: DECREMENT};

} // Define an action creator for decrementing

const store = Redux.createStore(counterReducer); // Define the Redux store here, passing in your reducers

## Never Mutate State

### Explanation:

These final challenges describe several methods of enforcing the key principle of state immutability in Redux. Immutable state means that you never modify state directly, instead, you return a new copy of state.

If you took a snapshot of the state of a Redux app over time, you would see something like state 1, state 2, state 3,state 4, ... and so on where each state may be similar to the last, but each is a distinct piece of data. This immutability, in fact, is what provides such features as time-travel debugging that you may have heard about.

Redux does not actively enforce state immutability in its store or reducers, that responsibility falls on the programmer. Fortunately, JavaScript (especially ES6) provides several useful tools you can use to enforce the immutability of your state, whether it is a string, number, array, or object. Note that strings and numbers are primitive values and are immutable by nature. In other words, 3 is always 3. You cannot change the value of the number 3. An array or object, however, is mutable. In practice, your state will probably consist of an array or object, as these are useful data structures for representing many types of information.

### Example:

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);

    default:

      return state;

  }

};

const addToDo = (todo) => {

  return {

    type: ADD\_TO\_DO,

    todo

  }

}

const store = Redux.createStore(immutableReducer);

## Use the Spread Operator on Arrays

### Explanation:

One solution from ES6 to help enforce state immutability in Redux is the spread operator: .... The spread operator has a variety of applications, one of which is well-suited to the previous challenge of producing a new array from an existing array. This is relatively new, but commonly used syntax. For example, if you have an array myArray and write:

let newArray = [...myArray];

newArray is now a clone of myArray. Both arrays still exist separately in memory. If you perform a mutation like newArray.push(5), myArray doesn't change. The ... effectively spreads out the values in myArray into a new array. To clone an array but add additional values in the new array, you could write [...myArray, 'new value']. This would return a new array composed of the values in myArray and the string new value as the last value. The spread syntax can be used multiple times in array composition like this, but it's important to note that it only makes a shallow copy of the array. That is to say, it only provides immutable array operations for one-dimensional arrays.

### Example:

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

      return [...state, action.todo];

    default:

      return state;

  }

};

const addToDo = (todo) => {

  return {

    type: 'ADD\_TO\_DO',

    todo

  }

}

const store = Redux.createStore(immutableReducer);

## Remove an Item from an Array

### Explanation:

Time to practice removing items from an array. The spread operator can be used here as well. Other useful JavaScript methods include slice() and concat().

### Example:

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)];

    default:

      return state;

  }

};

const removeItem = (index) => {

  return {

    type: 'REMOVE\_ITEM',

    index

  }

}

const store = Redux.createStore(immutableReducer);

## Copy an Object with Object.assign

### Explanation:

The last several challenges worked with arrays, but there are ways to help enforce state immutability when state is an object, too. A useful tool for handling objects is the Object.assign() utility. Object.assign() takes a target object and source objects and maps properties from the source objects to the target object. Any matching properties are overwritten by properties in the source objects. This behavior is commonly used to make shallow copies of objects by passing an empty object as the first argument followed by the object(s) you want to copy. Here's an example:

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

This creates newObject as a new object, which contains the properties that currently exist in obj1 and obj2.

### Example:

const defaultState = {

  user: 'CamperBot',

  status: 'offline',

  friends: '732,982',

  community: 'freeCodeCamp'

};

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

  switch(action.type) {

    case 'ONLINE':

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

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

    default:

      return state;

  }

};

const wakeUp = () => {

  return {

    type: 'ONLINE'

  }

};

const store = Redux.createStore(immutableReducer);