**REDUX**

An open-source JS library for managing application state. Commonly used with React or Angular for building user interfaces. It is a state management framework that can be used with a number of different web technologies, including React.

A predictable state container for JS apps. Redux helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.

**1. Create a Redux Store**

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)

**2. Get State from the Redux Store**

const store = Redux.createStore(

(state = 5) => state

);

// change code below this line

const currentState = store.getState();

**3. Define a Redux Action**

let action = {

type: 'LOGIN'

}

An “action creator” is simply a JS function that returns an action. It creates objects that represent action events.

**4. Define an Action Creator**

const action = {

type: 'LOGIN'

}

// Define an action creator here:

function actionCreator() {

return action;

}

**5. Dispatch an Action Event**

const store = Redux.createStore(

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

);

const loginAction = () => {

return {

type: 'LOGIN'

}

};

// Dispatch the action here:

store.dispatch(loginAction());

**Reducers** in Redux are responsible for state modifications that take place in response to actions. A **reducer** takes state and action as arguments, and always returns a new state. It is the **only** role of the reducer.

**6. Handle an Action in the Store**

const defaultState = {

login: false

};

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

// change code below this line

if(action.type === 'LOGIN') {

return {

login: true

}

} else {

return defaultState

}

// change code above this line

};

const store = Redux.createStore(reducer);

const loginAction = () => {

return {

type: 'LOGIN'

}

};

**7. Use a Switch Statement to Handle Multiple Actions**

const defaultState = {

authenticated: false

};

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

// change code below this line

switch(action.type) {

case 'LOGIN':

return {

authenticated: true

}

break;

case 'LOGOUT':

return {

authenticated: false

}

break;

default:

return defaultState;

}

// change code above this line

};

const store = Redux.createStore(authReducer);

const loginUser = () => {

return {

type: 'LOGIN'

}

};

const logoutUser = () => {

return {

type: 'LOGOUT'

}

};

A common practice when working with Redux is to assign action types as read-only constants, then reference these constants wherever they are used.

**8. Use const for Action Types**

// 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

}

};

store.subscribe()

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.

**9. Register a Store Listener**

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 addCount = () => count += 1;

store.subscribe(addCount);

// 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);

**10. Combine Multiple Reducers**

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);

**11. Send Action Data to the Store**

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;

break;

// 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());

**12. Use Middleware to Handle Asynchronous Actions**

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} }

//action creator

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)

);

**13. Write a Counter with Redux**

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

//global variable:

//let state = 0; \*\*\*no need

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

**14. Never Mutate State**

 Immutable state means that you never modify state directly, instead, you return a new copy of state.

There are a few ways to accomplish this with standard JavaScript or ES6. See if you can find a way to return a new array with the item from action.todoappended to the end.

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;

}

};

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

const addToDo = (todo) => {

return {

type: ADD\_TO\_DO,

todo

}

}

const store = Redux.createStore(immutableReducer);

\*\*\***concat = returns a new array**

**15. Use the Spread Operator on Arrays**

One solution from ES6 to help enforce state immutability in Redux is the spread operator: .... producing a new array from an existing array

if you have an array myArrayand write:

let newArray = [...myArray];

newArray is now a clone of myArray

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 myArrayand 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.

 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);

**16. Remove an Item from an Array**

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);

**17. Copy an Object with Object.assign**

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.

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

This creates newObjectas a new object, which contains the properties that currently exist in obj1and obj2.

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);