Redux Introduction

Springboard

Download Demo Code

Goals

- Describe what Redux is and how it can be useful
- Define key redux terminology
  - ***store***
  - ***reducer***
  - ***action***
  - ***dispatch***
- Define a pure function, and see examples of pure and impure functions
- Include Redux in an application

Redux

- A library for **state management**
- Very useful for managing larger applications with quite a bit of state
- Helps solves the issue of prop-drilling
- Commonly used with React, but doesn't need to be!
  - We'll first look at Redux as a standalone library

The case for central state management

- Passing down props over and over is challenging
- Passing data back up over and over is challenging
- It's common in large applications to have shared state

How it works

- In Redux, the centralized place where state is stored is called a **store**
- Let's include the Redux CDN so we can start using it!

```
<script src="https://unpkg.com/redux"></script>
```

Let's make a store

```
const store = Redux.createStore();
```

```
> Redux.createStore()
✖ ▶Uncaught Error: Expected the reducer to be a function.                    redux.js:523
    at Object.createStore (redux.js:523)
    at <anonymous>:1:7
```

Welp...that didn't work

- You can not create a store without specifying how to define initial state
- The store also needs to know what changes to make to the state
- We solve this problem by passing a function to the store.
- That function is called a **reducer**

Reducers

What is a reducer?

- Reducers are functions
- They accept two objects: a **state** and an **action**
- They use the action to produce and return a new state object
- They should be **pure** functions, i.e. they should not mutate their inputs (more on this later)

Our first reducer

```
const INITIAL_STATE = { count: 0 };

function countReducer(state=INITIAL_STATE, action) {
  // no way to update state yet,
  // let's just return it.
  return state;
}
```

Getting state

```
const store = Redux.createStore(countReducer);

store.getState(); // { count: 0 }
```

Pure Functions

An essential note on reducers

- Reducers **must be pure functions**
- We need to make sure that we do not mutate state
- You won't see difference now, but things won't work when we add React
- You won't even get any nice error messages 😅

Methods that mutate

- ***push* / *pop***
- ***shift* / *unshift***
- ***splice***
- modifying keys in an object/array

Methods / patterns that do not mutate

- ***map***
- ***filter***
- spread / ***Object.assign***
- ***concat***
- ***slice***

Pure vs. Impure: Example 1

```
// impure: adds val to an array
function addToArrImpure(arr, val) {
  arr.push(val);
  return arr;
}

let nums = [1, 2, 3];
addToArrImpure(nums, 4);
nums; // [1, 2, 3, 4] <-- impure!
```

```
// pure: adds val to an array
function addToArrPure(arr, val) {
  return [...arr, val];
}

let nums = [1, 2, 3];
addToArrPure(nums, 4);
nums; // [1, 2, 3] <-- pure!
```

Pure vs. Impure: Example 2

```
// impure: adds key-val pair to an object
function addToObjImpure(obj, key, val) {
  obj[key] = val;
  return obj;
}

let dog = { name: "Whiskey" };
addToObjImpure(dog, "favFood", "popcorn");
dog;
// {
//   name: "Whiskey",
//   favFood: "popcorn"
// } <-- impure!
```

```
// pure: adds key-val pair to an object
function addToObjPure(obj, key, val) {
  return { ...obj, [key]: val };
}

let dog = { name: "Whiskey" };
addToObjPure(dog, "favFood", "popcorn");
dog; // { name: "Whiskey" } <-- pure!
```

Pure vs. Impure: Example 3

```
// impure: doubles values in an array
function doubleImpure(nums) {
  nums.forEach((num, i) => nums[i] *= 2);
  return nums;
}

let nums = [1, 2, 3];
doubleImpure(nums);
nums; // [2, 4, 6] <-- impure!
```

```
// pure: doubles values in an array
function doublePure(nums) {
  return nums.map(num => 2 * num);
}

let nums = [1, 2, 3];
doublePure(nums);
nums; // [1, 2, 3] <-- pure!
```

Actions

- Reducers are not called directly.
- Instead, we fire off an **action**, which is intercepted and processed by a reducer
- Redux actions are simple instructions that tell reducer(s) how to adjust state
- They are objects and contain a **type** key
  - **type** is, by convention, a string in UPPER_SNAKE_CASE.
- The way we "fire off an action" is by running the **dispatch** function on the store

Actions are **objects with a key of type**

```
store.dispatch({ type: "LOG_STATE" });
```

Modifying Our Reducer

```
const INITIAL_STATE = { count: 0 };

function countReducer(state=INITIAL_STATE, action) {
  if (action.type === "LOG_STATE") {
    // doesn't actually update state,
    // but let's make sure the action is processed
    console.log("COMO MERE IS THE STATE", state);
    return state;
  }

  // if we can't match the action type,
  // just return the state
  return state;
}

const store = Redux.createStore(countReducer);

store.dispatch({ type: "LOG_STATE" });
// will console.log

store.dispatch({ type: "WILL_NOT_FIND_THIS" });
// won't console.log
```

Common Dispatch Errors

```
store.dispatch();
// error! dispatch wants an object
```

```
> store.dispatch()
✖ ▶Uncaught Error: Actions must be plain objects. Use custom middleware for async actions.   redux.js:232
    at Object.dispatch (redux.js:232)
    at <anonymous>:1:7
```

```
store.dispatch({});
// error! dispatch wants an object with a type key
```

```
> store.dispatch({})
✖ ▶Uncaught Error: Actions may not have an undefined "type" property. Have you misspelled a constant?   redux.js:236
    at Object.dispatch (redux.js:236)
    at <anonymous>:1:7
```

Actions can have additional keys

They're just objects!

```
store.dispatch({
  type: "SOME_ACTION",
  payload: "some value",
});
```

- Often the data in the action is referred to as a "payload", hence the key name
- You can reference the payload in the reducer via `action.payload`

Building a Counter with Redux!

Our HTML

demo/index.html

```
<body>
  <h1>0</h1>
  <button id="increment"> + </button>
  <button id="decrement"> - </button>

  <script src="https://unpkg.com/redux"></script>
  <script src="reduxSetup.js"></script>
  <script src="counter.js"></script>
</body>
```

Our Redux Setup

demo/reduxSetup.js

```
const INITIAL_STATE = { count: 0 };

function rootReducer(state = INITIAL_STATE, action) {
  switch (action.type) {
    case "INCREMENT":
      return { ...state, count: state.count + 1 };

    case "DECREMENT":
      return { ...state, count: state.count - 1 };


    default:
      return state;
  }
}

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

It's common to see ***switch*** statements in reducers, where you switch on the action type

demo/counter.js

```
window.onload = function() {
  const counterElement = document.querySelector("h1");

  document.querySelector("#increment")
      .addEventListener("click", function () {
        store.dispatch({ type: "INCREMENT" });
        const currentCount = store.getState().count;
        counterElement.innerText = currentCount;
  });

  document.querySelector("#decrement")
      .addEventListener("click", function () {
        store.dispatch({ type: "DECREMENT" });
        const currentCount = store.getState().count;
        counterElement.innerText = currentCount;
  });
}
```

Recap

Redux data flow

Data lifecycle

The data lifecycle in any Redux app follows these 4 steps:

- You call ***store.dispatch(action)***.
- Redux store calls the reducer function you gave it.
- Root reducer may combine output of multiple reducers into single state tree.
- Redux store saves the complete state tree returned by the root reducer.

Looking Ahead

Coming Up

- Integrating React with Redux
- Combining reducers
- Async with Redux