Redux

Redux is a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger. You can use Redux together with React, or with any other view library. It is tiny (2kB, including dependencies).

Redux provides a solid, stable and mature solution to managing state in your React application. Through a handful of small, useful patterns, Redux can transform your application from a total mess of confusing and scattered state, into a delightfully organized, easy to understand modern JavaScript powerhouse.

- Actions
- Reducers
- Store
- Data Flow

Actions

Actions are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using store.dispatch(). Actions are plain JavaScript objects. Actions must have a type property that indicates the type of action being performed. Types should typically be defined as string constants. Once your app is large enough, you may want to move them into a separate module.

```
const ADD_TODO = 'ADD_TODO'

{
 type: ADD_TODO,
 text: 'Build my first Redux app'
}
```

```
import { ADD_TODO, REMOVE_TODO } from '../actionTypes'
```

Other than type, the structure of an action object is really up to you.

Action Creators

Action creators are exactly that—functions that create actions. It's easy to conflate the terms “action” and “action creator”, so do your best to use the proper term.

In Redux, action creators simply return an action:

```
function addTodo(text) {
        return {
                type: ADD_TODO,
                text
         }
}
```

This makes them portable and easy to test.

to actually initiate a dispatch, pass the result to the dispatch() function:

dispatch(addTodo(text))

dispatch(completeTodo(index))

Alternatively, you can create a bound action creator that automatically dispatches:

const boundAddTodo = text => dispatch(addTodo(text))

const boundCompleteTodo = index => dispatch(completeTodo(index))

Now you'll be able to call them directly:

boundAddTodo(text)

boundCompleteTodo(index)

Sample code : action.js

```
/*
 * action types
 */

export const ADD_TODO = 'ADD_TODO'
export const TOGGLE_TODO = 'TOGGLE_TODO'
export const SET_VISIBILITY_FILTER = 'SET_VISIBILITY_FILTER'

/*
 * other constants
 */

export const VisibilityFilters = {
  SHOW_ALL: 'SHOW_ALL',
  SHOW_COMPLETED: 'SHOW_COMPLETED',
  SHOW_ACTIVE: 'SHOW_ACTIVE'
}

/*
 * action creators
 */

export function addTodo(text) {
  return { type: ADD_TODO, text }
}

export function toggleTodo(index) {
  return { type: TOGGLE_TODO, index }
}

export function setVisibilityFilter(filter) {
  return { type: SET_VISIBILITY_FILTER, filter }
}
```

Reducers

Reducers specify how the application's state changes in response to actions sent to the store. Remember that actions only describe the fact that something happened, but don't describe how the application's state changes.

Designing the State Shape

In Redux, all the application state is stored as a single object. It's a good idea to think of its shape before writing any code.

```
{
 visibilityFilter: 'SHOW_ALL',
 todos: [
  {
   text: 'Consider using Redux',
   completed: true,
  },
  {
   text: 'Keep all state in a single tree',
   completed: false
  }
 ]
}
```

Handling Actions

Now that we've decided what our state object looks like, we're ready to write a reducer for it. The reducer is a pure function that takes the previous state and an action, and returns the next state.

(previousState, action) => newState

It's called a reducer because it's the type of function you would pass to Array.prototype.reduce(reducer, ?initialValue). It's very important that the reducer stays pure. Things you should never do inside a reducer:

- *Mutate its arguments;*
- *Perform side effects like API calls and routing transitions;*
- *Call non-pure functions, e.g. Date.now() or Math.random().*

Note: What's that a "pure" function?

A pure function doesn't depend on and doesn't modify the states of variables out of its scope. Concretely, that means a pure function always returns the same result given same parameters. Its execution doesn't depend on the state of the system.

For now, just remember that the reducer must be pure. Given the same arguments, it should calculate the next state and return it. No surprises. No side effects. No API calls. No mutations. Just a calculation.

```
import { VisibilityFilters } from './actions'

const initialState = {
 visibilityFilter: VisibilityFilters.SHOW_ALL,
 todos: []
}
```

```
function todoApp(state, action) {
 if (typeof state === 'undefined') {
  return initialState
 }

 // For now, don't handle any actions
 // and just return the state given to us.
 return state
}
```

One neat trick is to use the ES6 default arguments syntax to write this in a more compact way:

```
function todoApp(state = initialState, action) {
 // For now, don't handle any actions
 // and just return the state given to us.
 return state
}
```

Now let's handle `SET_VISIBILITY_FILTER`. All it needs to do is to change `visibilityFilter` on the state. Easy:

```
function todoApp(state = initialState, action) {
 switch (action.type) {
  case SET_VISIBILITY_FILTER:
   return Object.assign({}, state, {
    visibilityFilter: action.filter
   })
  default:
   return state
 }
}
```

Handling More Actions

We have two more actions to handle! Just like we did with SET_VISIBILITY_FILTER, we'll import the ADD_TODO and TOGGLE_TODO actions and then extend our reducer to handle ADD_TODO.

```
function todoApp(state = initialState, action) {
 switch (action.type) {
  case SET_VISIBILITY_FILTER:
   return Object.assign({}, state, {
    visibilityFilter: action.filter
   })
  case ADD_TODO:
   return Object.assign({}, state, {
    todos: [
     ...state.todos,
     {
      text: action.text,
      completed: false
     }
    ]
   })
  case TOGGLE_TODO:
   return Object.assign({}, state, {
    todos: state.todos.map((todo, index) => {
     if (index === action.index) {
      return Object.assign({}, todo, {
```

```
          completed: !todo.completed
        })
      }
      return todo
    })
  })
 default:
  return state
 }
}
```

Splitting Reducers

```
const { SHOW_ALL } = VisibilityFilters
```

```
function visibilityFilter(state = SHOW_ALL, action) {
 switch (action.type) {
  case SET_VISIBILITY_FILTER:
   return action.filter
  default:
   return state
 }
}
```

```
function todos(state = [], action) {
 switch (action.type) {
  case ADD_TODO:
   return [
    ...state,
    {
     text: action.text,
     completed: false
    }
   ]
  case TOGGLE_TODO:
   return state.map((todo, index) => {
    if (index === action.index) {
     return Object.assign({}, todo, {
      completed: !todo.completed
     })
    }
    return todo
   })
  default:
   return state
 }
}
```

```
function todoApp(state = {}, action) {
 return {
  visibilityFilter: visibilityFilter(state.visibilityFilter, action),
  todos: todos(state.todos, action)
 }
}
```

Note that each of these reducers is managing its own part of the global state. The state parameter is different for every reducer, and corresponds to the part of the state it manages.

Redux provides a utility called combineReducers() that does the same boilerplate logic that the todoApp above currently does. With its help, we can rewrite todoApp like this:

```
import { combineReducers } from 'redux'

const todoApp = combineReducers({
  visibilityFilter,
  todos
})

export default todoApp
```

Note that this is equivalent to:

```
export default function todoApp(state = {}, action) {
  return {
    visibilityFilter: visibilityFilter(state.visibilityFilter, action),
    todos: todos(state.todos, action)
  }
}
```

Store

The Store is the object that brings actions and reducers together. The store has the following responsibilities:

- Holds application state;
- Allows access to state via getState();
- Allows state to be updated via dispatch(action);
- Registers listeners via subscribe(listener);
- Handles unregistering of listeners via the function returned by subscribe(listener).

It's important to note that you'll only have a single store in a Redux application. When you want to split your data handling logic, you'll use reducer composition instead of many stores.

It's easy to create a store if you have a reducer. In the previous section, we used combineReducers() to combine several reducers into one. We will now import it, and pass it to createStore().

```
import { createStore } from 'redux'
import todoApp from './reducers'
const store = createStore(todoApp)
```

You may optionally specify the initial state as the second argument to createStore(). This is useful for hydrating the state of the client to match the state of a Redux application running on the server.

```
const store = createStore(todoApp, window.STATE_FROM_SERVER)
```

```
import { createStore } from 'redux'
```

```
import todoApp from './reducers'

const store = createStore(todoApp)
```

Dispatching Actions

Now that we have created a store, let's verify our program works! Even without any UI, we can already test the update logic.

```
import {
 addTodo,
 toggleTodo,
 setVisibilityFilter,
 VisibilityFilters
} from './actions'

// Log the initial state
console.log(store.getState())

// Every time the state changes, log it
// Note that subscribe() returns a function for unregistering the listener
const unsubscribe = store.subscribe(() =>
 console.log(store.getState())
)

// Dispatch some actions
store.dispatch(addTodo('Learn about actions'))
store.dispatch(addTodo('Learn about reducers'))
store.dispatch(addTodo('Learn about store'))
store.dispatch(toggleTodo(0))
store.dispatch(toggleTodo(1))
store.dispatch(setVisibilityFilter(VisibilityFilters.SHOW_COMPLETED))

// Stop listening to state updates
unsubscribe()
```

Data Flow

Redux architecture revolves around a strict unidirectional data flow.

This means that all data in an application follows the same lifecycle pattern, making the logic of your app more predictable and easier to understand. It also encourages data normalization, so that you don't end up with multiple, independent copies of the same data that are unaware of one another.

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

1) You call store.dispatch(action).
   An action is a plain object describing what happened. For example:

```
{ type: 'LIKE_ARTICLE', articleId: 42 }
{ type: 'FETCH_USER_SUCCESS', response: { id: 3, name: 'Mary' } }
{ type: 'ADD_TODO', text: 'Read the Redux docs.' }
```

2) The Redux store calls the reducer function you gave it.

The store will pass two arguments to the reducer: the current state tree and the action. For example, in the todo app, the root reducer might receive something like this:

```
// The current application state (list of todos and chosen filter)
let previousState = {
  visibleTodoFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Read the docs.',
      complete: false
    }
  ]
}

// The action being performed (adding a todo)
let action = {
  type: 'ADD_TODO',
  text: 'Understand the flow.'
}

// Your reducer returns the next application state
let nextState = todoApp(previousState, action)
```

3) The root reducer may combine the output of multiple reducers into a single state tree. Here's how combineReducers() works. Let's say you have two reducers, one for a list of todos, and another for the currently selected filter setting:

```
  function todos(state = [], action) {
  // Somehow calculate it...
  return nextState
}

function visibleTodoFilter(state = 'SHOW_ALL', action) {
  // Somehow calculate it...
  return nextState
}

let todoApp = combineReducers({
  todos,
  visibleTodoFilter
})
```

4) The Redux store saves the complete state tree returned by the root reducer.
This new tree is now the next state of your app! Every listener registered with store.subscribe(listener) will now be invoked; listeners may call store.getState() to get the current state. Now, the UI can be updated to reflect the new state. If you use bindings like React Redux, this is the point at which component.setState(newState) is called.