Redux

As the requirements for JavaScript single-page applications have become increasingly complicated, **our code must manage more state than ever before**. This state can include server responses and cached data, as well as locally created data that has not yet been persisted to the server. UI state is also increasing in complexity, as we need to manage active routes, selected tabs, spinners, pagination controls, and so on.

Managing this ever-changing state is hard. At some point, you no longer understand what happens in your app as you have **lost control over the when, why, and how of its state.**

Redux is the best solution for managing the state in a global level.

Three principles:

- **Single source of truth**

The state of your whole application is stored in an object tree within a single store.

- **State is read-only**

The only way to change the state is to emit an action, an object describing what happened.

This ensures that neither the views nor the network callbacks will ever write directly to the state.

- **Changes are made with pure functions**

To specify how the state tree is transformed by actions, you write pure reducers.

Reducers are just pure functions that take the previous state and an action, and return the next state. Remember to return new state objects, instead of mutating the previous state.

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()`

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

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.

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.

The `dispatch()` function can be accessed directly from the store as `store.dispatch()`, but more likely you'll access it using a helper like react-redux's `connect()`

Reducers:

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

In Redux, all the application state is stored as a single object. You'll often find that you need to store some data, as well as some UI state, in the state tree. This is fine, but try to keep the data separate from the UI state.

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

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

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()

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.

// initial state.js

```
Const initialState= {
 currentBuddyInfo: {
   currentBuddy: {}
```

```
 },
 profileInfo: {
   userId: null,
   firstName: null,
   lastName: null,
   profileUrl: null,
   onlineStatus: 'Available',
   emailId: localStorage.getItem('emailId') !== null ? localStorage.getItem('emailId') : null,
   password: localStorage.getItem('password') !== null ?
localStorage.getItem('password') : null
 }
}

// reducer

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
         }
       ]
     })
   default:
     return state
 }
}
```

Note that:

1. **We don't mutate the** `state`**.** We create a copy with `Object.assign()`. `Object.assign(state, { visibilityFilter: action.filter })` is also wrong: it will mutate the first argument. You **must** supply an empty object as the first parameter. You can also enable the object spread operator proposal to write `{ ...state, ...newState }` instead.
2. **We return the previous** `state` **in the** `default` **case.** It's important to return the previous `state` for any unknown action.

Just like before, we never write directly to `state` or its fields, and instead we return new objects. The new `todos` is equal to the old `todos` concatenated with a single new item at the end. The fresh todo was constructed using the data from the action.

Finally, the implementation of the `TOGGLE_TODO` handler shouldn't come as a complete surprise:

```
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
   })
 })
```

Because we want to update a specific item in the array without resorting to mutations, we have to create a new array with the same items except the item at the index.

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.

This is already looking good! When the app is larger, we can split the reducers into separate files and keep them completely independent and managing different data domains.

Finally, 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
```

And like other reducers, combineReducers() does not create a new object if all of the reducers provided to it do not change state.

Store:

In the previous sections, we defined the actions that represent the facts about “what happened” and the reducers that update the state according to those actions.

The **Store** is the object that brings them 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 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)`

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.

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

1. You call `store.dispatch(action)`.

2. 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)
```

Note that a reducer is a pure function. It only *computes* the next state. It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an action is dispatched.

3. The root reducer may combine the output of multiple reducers into a single state tree.

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.

Usuage with Redux:

From the very beginning, we need to stress that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

Installing React Redux

React bindings are not included in Redux by default. You need to install them explicitly:

```
npm install --save react-redux
```

Presentational and Container Components

React bindings for Redux embrace the idea of **separating presentational and container components**. If you're not familiar with these terms, read about them first, and then come back. They are important, so we'll wait!

	Presentational component	Container component
Purpose	How things look	How things work (data fetching, state update)
Aware of Redux	No	Yes
To read data	Read data from props	Subscribe to Redux state
To change data	Invoke callbacks from props	Dispatch redux actions
Are written	By hand	Usually generated by React Redux

Most of the components we'll write will be presentational, but we'll need to generate a few container components to connect them to the Redux store.

```
import {connect} from 'react-redux'
import { SignIn } from '../components/signIn/SignIn'
import * as apis from '../middleware/api'

const mapStateToProps = (store, ownProps) => {
 return {
   isAuth: (localStorage.getItem('emailId') && localStorage.getItem('password')) ? false :
true
 }
}
```

```
const mapDispatchToProps = (dispatch, ownProps) => {
 return {
   handleSubmit: (e) => {
     e.preventDefault()
     let loginDetails = {
       access_id: e.target.accessId.value,
       password: e.target.password.value
     }
     dispatch(apis.loginUser(loginDetails))
   }
 }
}

const SignInContainer = connect(
 mapStateToProps,
 mapDispatchToProps
)(SignIn)

export default SignInContainer
```

Passing the Store

All container components need access to the Redux store so they can subscribe to it.

The option we recommend is to use a special React Redux component called <Provider> to magically make the store available to all container components in the application without passing it explicitly. You only need to use it once when you render the root component:

```
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import { store } from './store/store'
```

```
import AppContainer from './containers/AppContainer'
import 'babel-polyfill'

ReactDOM.render(
 <Provider store={store}>
   <AppContainer />
 </Provider>
 , document.getElementById('root')
)
```

Async actions:

When you call an asynchronous API, there are two crucial moments in time: the moment you start the call, and the moment when you receive an answer (or a timeout).

Each of these two moments usually require a change in the application state; to do that, you need to dispatch normal actions that will be processed by reducers synchronously.

Usually, for any API request you'll want to dispatch at least three different kinds of actions:

- **An action informing the reducers that the request began.**

  The reducers may handle this action by toggling an `isFetching` flag in the state. This way the UI knows it's time to show a spinner.

- **An action informing the reducers that the request finished successfully.**

  The reducers may handle this action by merging the new data into the state they manage and resetting `isFetching`. The UI would hide the spinner, and display the fetched data.

- **An action informing the reducers that the request failed.**

The reducers may handle this action by resetting `isFetching`. Additionally, some reducers may want to store the error message so the UI can display it.

Async Action Creators

The standard way to do it with Redux is to use the Redux Thunk middleware. It comes in a separate package called `redux-thunk`.

One important thing you need to know: by using this specific middleware, an action creator can return a function instead of an action object. This way, the action creator becomes a thunk.

When an action creator returns a function, that function will get executed by the Redux Thunk middleware. This function doesn't need to be pure; it is thus allowed to have side effects, including executing asynchronous API calls. The function can also dispatch actions—like those synchronous actions we defined earlier.

```
function loginUser (loginRequestModel) {
 return function (dispatch) {
   dispatch(actions.loginUserRequest(loginRequestModel))
   return fetch(config.loginUserEndpoint, {
     method: 'post',
     headers: {
       'Content-Type': 'application/json'
     },
     body: JSON.stringify(loginRequestModel)
   }).then(response =>
     response.json().then(json => {
       if (!response.ok) {
         return Promise.reject(json)
       }
       console.log(json)
       if (json.hasOwnProperty('error')) {
         dispatch(actions.loginUserResponseError(json))
       } else {
```

```
        if (json.status_code === '2002') {
          // Save to session & local storage
          sessionStorage.setItem('handshakeToken', json.handshake_token)
          sessionStorage.setItem('firstLogin', 'true')
          localStorage.setItem('emailId', loginRequestModel.access_id)
          localStorage.setItem('password', loginRequestModel.password)
          dispatch(actions.loginUserResponseSuccess(json))
          if ((loginUser.handshakeToken !== null) &&
              (typeof store.getState().loginUser.handshakeToken !== 'undefined')) {
            console.log('Initialising handshake with the Chat Server')

dispatch(initialiseHandshakeWithChatServer(store.getState().loginUser.handshakeToke
n))
          }
        }
      }
    }))
 }
}
```

Note on fetch

We use fetch API in the examples. It is a new API for making network requests that replaces XMLHttpRequest for most common needs. Because most browsers don't yet support it natively, we suggest that you use cross-fetch library:

```
// Do this in every file where you use `fetch`
import fetch from 'cross-fetch'
```

Thunk middleware isn't the only way to orchestrate asynchronous actions in Redux:

- You can use redux-promise or redux-promise-middleware to dispatch Promises instead of functions.
- You can use the redux-saga middleware to build more complex asynchronous actions.

Async flow:

Without middleware, Redux store only supports synchronous data flow. This is what you get by default with createStore().

You may enhance createStore() with applyMiddleware(). It is not required, but it lets you express asynchronous actions in a convenient way.

Asynchronous middleware like redux-thunk or redux-promise wraps the store's dispatch() method and allows you to dispatch something other than actions, for example, functions or Promises.

Middleware:

In these frameworks, middleware is some code you can put between the framework receiving a request, and the framework generating a response.

Redux middleware solves different problems. **It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer. People use Redux middleware for logging, crash reporting, talking to an asynchronous API, routing, and more.**

To log the state for each action dispatch,

```
import thunkMiddleware from 'redux-thunk'
import { createLogger } from 'redux-logger'
import { createStore, applyMiddleware } from 'redux'
import rootReducer from '../reducers/index'

const loggerMiddleWare = createLogger()

export const store = createStore(
 rootReducer,
 applyMiddleware(
   thunkMiddleware,
   loggerMiddleWare
 )
)
```