Redux Overview

* Redux Overview

Redux is a predictable state container for JavaScript apps. As the application grows, it becomes difficult to keep it organized and maintain data flow. Redux solves this problem by managing application’s state with a single global object called Store. Redux fundamental principles help in maintaining consistency throughout your application, which makes debugging and testing easier.

More importantly, it gives you live code editing combined with a time-travelling debugger. It is flexible to go with any view layer such as React, Angular, Vue, etc.

* Flux

Flux is an application architecture that Facebook uses internally for building the client-side web application with React. It is not a library nor a framework. It is neither a library nor a framework. It is a kind of architecture that complements React as view and follows the concept of Unidirectional Data Flow model. It is useful when the project has dynamic data, and we need to keep the data updated in an effective manner. It reduces the runtime errors.

Flux applications have three major roles in dealing with data:

1. Dispatcher
2. Stores
3. Views (React components)

* Flux Elements

Following is a simple explanation of the **flux** concept.

* **Actions** − Actions are sent to the dispatcher to trigger the data flow.
* **Dispatcher** − This is a central hub of the app. All the data is dispatched and sent to the stores.
* **Store** − Store is the place where the application state and logic are held. Every store is maintaining a particular state and it will update when needed.
* **View** − The **view** will receive data from the store and re-render the app.
* Dispatcher

It is a central hub for the React Flux application and manages all data flow of your Flux application. It is a registry of callbacks into the stores. It has no real intelligence of its own, and simply acts as a mechanism for distributing the actions to the stores. All stores register itself and provide a callback. It is a place which handled all events that modify the store. When an action creator provides a new action to the dispatcher, all stores receive that action via the callbacks in the registry.

The dispatcher's API has five methods. These are:

|  |  |  |
| --- | --- | --- |
| SN | Methods | Descriptions |
| 1. | register() | It is used to register a store's action handler callback. |
| 2. | unregister() | It is used to unregisters a store's callback. |
| 3. | waitFor() | It is used to wait for the specified callback to run first. |
| 4. | dispatch() | It is used to dispatches an action. |
| 5. | isDispatching() | It is used to checks if the dispatcher is currently dispatching an action. |

* Stores

It primarily contains the application state and logic. It is similar to the model in a traditional MVC. It is used for maintaining a particular state within the application, updates themselves in response to an action, and emit the change event to alert the controller view.

* Views

It is also called as controller-views. It is located at the top of the chain to store the logic to generate actions and receive new data from the store. It is a React component listen to change events and receives the data from the stores and re-render the application.

* Actions

The dispatcher method allows us to trigger a dispatch to the store and include a payload of data, which we call an action. It is an action creator or helper methods that pass the data to the dispatcher.

* Flux Pros
* It is a unidirectional data flow model which is easy to understand.
* It is open source and more of a design pattern than a formal framework like MVC architecture.
* The flux application is easier to maintain.
* The flux application parts are decoupled.
* Connecting Redux and React

we will learn how to implements Redux in React application. Here, we provide a simple example to connect Redux and React.

**Step 1: Import Redux NPM packages**

npm install redux react-redux

**Step 2: Create a Reducer**

A reducer is a pure function that accepts 2 parameters: state and action. State can be anything, including objects. Action is an object with type property that specifies the type of action as a string. Let's create a countReducer in src/index.js

**src/index.js**

const countReducer = function (state = 0, action) {

switch (action.type) {

case "INCREMENT":

return state + 1;

case "DECREMENT":

return state - 1;

default:

return state;

}

};

**Step 3: Create a Redux Store**

The store holds the state as one big global object known as a state tree. The store allows us to:  
- **dispatch** actions to modify the state  
- **subscribe** to recieve notification about state changes  
- **retrieve** the entire state tree  
Let's import and create the redux store and use our reducer to initialize it:

**src/index.js**

...

import { createStore } from 'redux';

...

let store = createStore(countReducer);

...

**Step 4: Wrap the Main App Component with Provider**

Now, we will connect redux to react using the NPM library react-redux. Let's import the <Provider />, and wrap our main app component with it. Also, pass our previously created store into the <Provider />'s store prop.

**src/index.js**

...

import { Provider } from 'react-redux';

...

const App = () => (

<Provider store={store}>

<h1>Helloworld React & Redux!</h1>

</Provider>

);

ReactDOM.render(<App />, document.getElementById('root'));

**Step 5: Create and Connect a Container Component**

In the world of React & Redux, Container (smart) Components are responsible for pulling state from the Redux store, transforming it, and passing it down to Presentational (dumb) Components. Let's convert our <h1> tag into a Container Component.

**src/index.js**

...

import { Provider, connect } from 'react-redux';

...

const Component = () => <h1>Helloworld React & Redux!</h1>;

const Container = connect()(Component);

const App = () => (

<Provider store={store}>

<Container />

</Provider>

);

...

**Step 6: Select and Transform State from Redux Store**

Let's use our Container component to select the state and optionally, transform it.

**src/index.js**

...

const mapStateToProps = state => {

return {

count: state

};

};

const Container = connect(mapStateToProps)(Component);

...

**Step 7: Use the State in the Presentational Component**

The count prop is now being passed to our Component. Let's declare it as our parameter, and add it to the JSX. The store responsibility of the Presentational Component is to convert props into JSX with little or no logic.

**src/index.js**

const Component = ({count}) => <h1>Helloworld React & Redux! {count}</h1>;

**Step 8: Add Buttons to our Presentational Component**

Now, we're going to add two buttons in our Presentational Component that increment and decrement the count.

**src/index.js**

const Component = ({count, handleIncrementClick, handleDecrementClick}) => (

<div>

<h1>Helloworld React & Redux! {count}</h1>

<button onClick={handleDecrementClick}>Decrement</button>

<button onClick={handleIncrementClick}>Increment</button>

</div>

);

**Step 9: Pass Callback that Dispatch Actions to Store**

It's time to map our store dispatch to callback functions. Here's the change:

**src/index.js**

const mapDispatchToProps = dispatch => {

return {

handleIncrementClick: () => dispatch({ type: 'INCREMENT' }),

handleDecrementClick: () => dispatch({type: 'DECREMENT'})

}

};

const Container = connect(mapStateToProps, mapDispatchToProps)(Component);

**Step 10: (optional): Refactor the Code**

Let's move the similar code into separate files to keep the project tidy and maintained. Let's create a separate file for the the Container component, Presentational component, store initialization, and reducer. Also, let's put all the counter code into a single directory, because as the project grows, we will create other components with their own reducers, containers, and presentational components. The final directory structure should look like this:

src

├── configure-store.js

├── counter

│   ├── component.js

│   ├── container.js

│   └── reducer.js

├── index.html

└── index.js

And here's the code in each file:

**src/counter/component.js**

import React from 'react';

export const Component = ({ count, handleIncrementClick, handleDecrementClick }) => (

<div>

<h1>Helloworld React & Redux! {count}</h1>

<button onClick={handleDecrementClick}>Decrement</button>

<button onClick={handleIncrementClick}>Increment</button>

</div>

);

**src/counter/container.js**

import { connect } from 'react-redux';

import { Component } from './component';

const mapStateToPros = state => {

return {

count: state

};

};

const mapDispatchToProps = dispatch => {

return {

handleIncrementClick: () => dispatch({ type: 'INCREMENT' }),

handleDecrementClick: () => dispatch({ type: 'DECREMENT' })

}

};

export const Container = connect(mapStateToProps, mapDispatchToProps)(Component);

**src/counter/reducer.js**

export const countReducer = function (state = 0, action) {

switch (action.type) {

case "INCREMENT":

return state + 1;

case "DECREMENT":

return state - 1;

default:

return state;

}

};

**src/configure-store.js**

import { createStore } from 'redux';

import { countReducer } from './counter/reducer';

export const store = createStore(countReducer);

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import { store } from './configure-store';

import { Container } from './counter/container';

const App = () => (

<Provider store={store}>

<Container />

</Provider>

);

ReactDOM.render(<App />, document.getElementById('root'));

* What is Redux?

Redux is an open-source JavaScript library used to manage application state. React uses Redux for building the user interface. It was first introduced by **Dan Abramov** and **Andrew Clark** in **2015**.

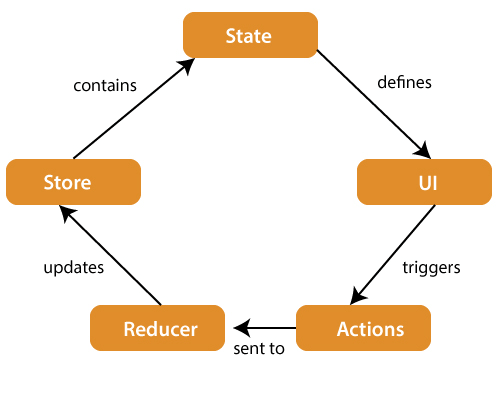
React Redux is the official React binding for Redux. It allows React components to read data from a Redux Store, and dispatch **Actions** to the **Store** to update data. Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model. React Redux is conceptually simple. It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

* Redux does not have Dispatcher concept.
* Redux has an only Store whereas Flux has many Stores.
* The Action objects will be received and handled directly by Store.
* Why use Redux with React. js?

The main reason to use React Redux are:

* React Redux is the official **UI bindings** for react Application. It is kept up-to-date with any API changes to ensure that your React components behave as expected.
* It encourages good 'React' architecture.
* It implements many performance optimizations internally, which allows to components re-render only when it actually needs.
* Redux Architecture?



The components of Redux architecture are explained below.

**STORE:** A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

**ACTION:** Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

**REDUCER:** Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

* Redux Principles

Predictability of Redux is determined by three most important principles as given below −

**Single Source of Truth**

The state of your whole application is stored in an object tree within a single store. As whole application state is stored in a single tree, it makes debugging easy, and development faster.

**State is Read-only**

The only way to change the state is to emit an action, an object describing what happened. This means nobody can directly change the state of your application.

**Changes are made with pure functions**

To specify how the state tree is transformed by actions, you write pure reducers. A reducer is a central place where state modification takes place. Reducer is a function which takes state and action as arguments, and returns a newly updated state.

* Redux Components

Redux has 4 components are given below:

**STORE:** A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

**ACTION:** Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

**REDUCER:** Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

**Middleware**: Middleware is the suggested way to extend Redux with custom functionality. Middlewares are used to dispatch async functions. We configure Middleware’s while creating a store.

const store = createStore(reducers, initialState, middleware);

* Store

A Store is a place where the entire state of your application lists. It manages the status of the application and has a dispatch(action) function. It is like a brain responsible for all moving parts in Redux.

* Action

 Action is sent or dispatched from the view which are payloads that can be read by Reducers. It is a pure object created to store the information of the user's event. It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.

* Middleware

Middleware is the suggested way to extend Redux with custom functionality. Middlewares are used to dispatch async functions. We configure Middleware’s while creating a store.

const store = createStore(reducers, initialState, middleware);

* What is a Reducer?

Reducers are a pure function in Redux. Pure functions are predictable. Reducers are the only way to change states in Redux. It is the only place where you can write logic and calculations. Reducer function will accept the previous state of app and action being dispatched, calculate the next state and returns the new object.

The following few things should never be performed inside the reducer −

* Mutation of functions arguments
* API calls & routing logic
* Calling non-pure function e.g. Math.random()

The following is the syntax of a reducer −

(state,action) => newState

Let us continue the example of showing the list of product items on a web page, discussed in the action creators module. Let us see below how to write its reducer.

const initialState = {

isLoading: false,

items: []

};

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

switch (action.type) {

case 'ITEMS\_REQUEST':

return Object.assign({}, state, {

isLoading: action.payload.isLoading

})

case ‘ITEMS\_REQUEST\_SUCCESS':

return Object.assign({}, state, {

items: state.items.concat(action.items),

isLoading: action.isLoading

})

default:

return state;

}

}

export default reducer;