React Hooks

Hooks are the new feature introduced in the React 16.8 version. It allows you to use state and other React features without writing a class. Hooks are the functions which "hook into" React state and lifecycle features from function components. It does not work inside classes.

Hooks are backward-compatible, which means it does not contain any breaking changes. Also, it does not replace your knowledge of React concepts.

When to use a Hooks

If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class. But, now you can do it by using a Hook inside the existing function component.

Rules of Hooks

Hooks are similar to JavaScript functions, but you need to follow these two rules when using them. Hooks rule ensures that all the stateful logic in a component is visible in its source code. These rules are:

1. Only call Hooks at the top level

Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions. This rule ensures that Hooks are called in the same order each time a components renders.

2. Only call Hooks from React functions

You cannot call Hooks from regular JavaScript functions. Instead, you can call Hooks from React function components. Hooks can also be called from custom Hooks.

Pre-requisites for React Hooks

1. Node version 6 or above
2. NPM version 5.2 or above
3. Create-react-app tool for running the React App

React Hooks Installation

To use React Hooks, we need to run the following commands:

```
$ npm install react@16.8.0-alpha.1 --save
$ npm install react-dom@16.8.0-alpha.1 --save
```

The above command will install the latest React and React-DOM alpha versions which support React Hooks. Make sure the **package.json** file lists the React and React-DOM dependencies as given below.

```
"react": "^16.8.0-alpha.1",
"react-dom": "^16.8.0-alpha.1",
```

Hooks State

Hook state is the new way of declaring a state in React app. Hook uses useState() functional component for setting and retrieving state. Let us understand Hook state with the following example.

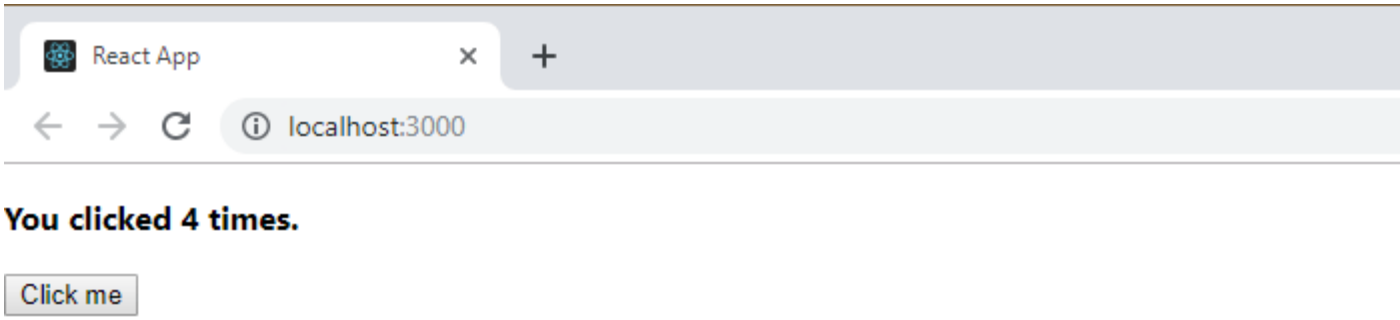
App.js

```
import React, { useState } from 'react';

function CountApp() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
export default CountApp;
```

output:



In the above example, useState is the Hook which needs to call inside a function component to add some local state to it. The useState returns a pair where the first element is the current state value/initial value, and the second one is a function which allows us to update it. After that, we will call this function from an event handler or somewhere else. The useState is similar to this.setState in class. The equivalent code without Hooks looks like as below.

App.js

```
import React, { useState } from 'react';

class CountApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p><b>You clicked {this.state.count} times</b></p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

```
21. export default CountApp;
```

Hooks Effect

The Effect Hook allows us to perform side effects (an action) in the function components. It does not use components lifecycle methods which are available in class components. In other words, Effects Hooks are equivalent to componentDidMount(), componentDidUpdate(), and componentWillUnmount() lifecycle methods.

Side effects have common features which the most web applications need to perform, such as:

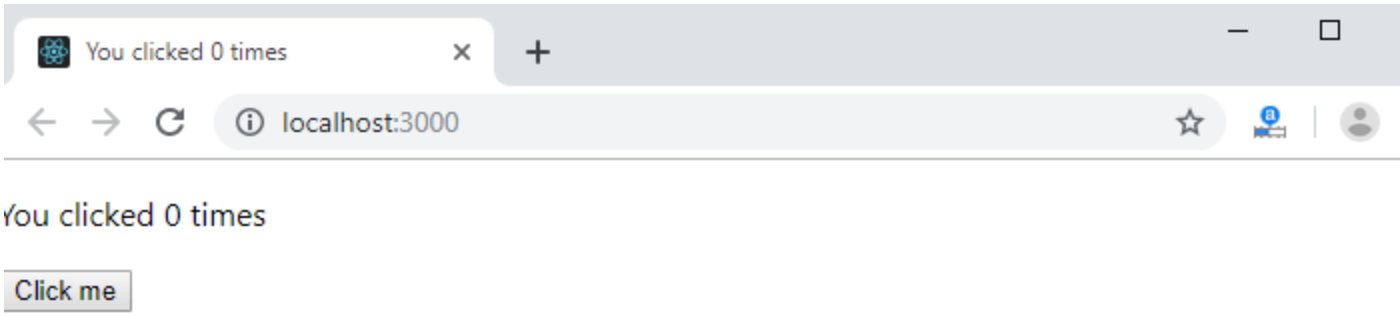
- Updating the DOM,
- Fetching and consuming data from a server API,
- Setting up a subscription, etc.

Let us understand Hook Effect with the following example.

```
1.  import React, { useState, useEffect } from 'react';
2.
3.  function CounterExample() {
4.    const [count, setCount] = useState(0);
5.
6.    // Similar to componentDidMount and componentDidUpdate:
7.    useEffect(() => {
8.      // Update the document title using the browser API
9.      document.title = `You clicked ${count} times`;
10.   });
11.
12.   return (
13.     <div>
14.       <p>You clicked {count} times</p>
15.       <button onClick={() => setCount(count + 1)}>
16.         Click me
17.       </button>
18.     </div>
19.   );
20. }
21. export default CounterExample;
```

The above code is based on the previous example with a new feature which we set the document title to a custom message, including the number of clicks.

Output:



In React component, there are two types of side effects:

1. Effects Without Cleanup
2. Effects With Cleanup

Effects without Cleanup

It is used in useEffect which does not block the browser from updating the screen. It makes the app more responsive. The most common example of effects which don't require a cleanup are manual DOM mutations, Network requests, Logging, etc.

Effects with Cleanup

Some effects require cleanup after DOM updation. For example, if we want to set up a subscription to some external data source, it is important to clean up memory so that we don't introduce a memory leak. React performs the cleanup of memory when the component unmounts. However, as we know that, effects run for every render method and not just once. Therefore, React also cleans up effects from the previous render before running the effects next time.

Custom Hooks

A custom Hook is a JavaScript function. The name of custom Hook starts with "use" which can call other Hooks. A custom Hook is just like a regular function, and the word "use" in the beginning tells that this function follows the rules of Hooks. Building custom Hooks allows you to extract component logic into reusable functions.

Let us understand how custom Hooks works in the following example.

```
1. import React, { useState, useEffect } from 'react';
2.
3. const useDocumentTitle = title => {
4.   useEffect(() => {
5.     document.title = title;
6.   }, [title])
7. }
8.
9. function CustomCounter() {
10.  const [count, setCount] = useState(0);
11.  const incrementCount = () => setCount(count + 1);
12.  useDocumentTitle(`You clicked ${count} times`);
13.  // useEffect(() => {
14.  //   document.title = `You clicked ${count} times`
15.  // });
16.
17.  return (
18.    <div>
19.      <p>You clicked {count} times</p>
20.      <button onClick={incrementCount}>Click me</button>
21.    </div>
22.  )
23. }
24. export default CustomCounter;
```

In the above snippet, useDocumentTitle is a custom Hook which takes an argument as a string of text which is a title. Inside this Hook, we call useEffect Hook and set the title as long as the title has changed. The second argument will perform that check and update the title only when its local state is different than what we are passing in.

> Note: A custom Hook is a convention which naturally follows from the design of Hooks, instead of a React feature.

Built-in Hooks

Here, we describe the APIs for the built-in Hooks in React. The built-in Hooks can be divided into two parts, which are given below.

Basic Hooks

- useState
- useEffect
- useContext

Additional Hooks

- useReducer
- useCallback
- useMemo
- useRef
- useImperativeHandle
- useLayoutEffect
- useDebugValue

React Flux Concept

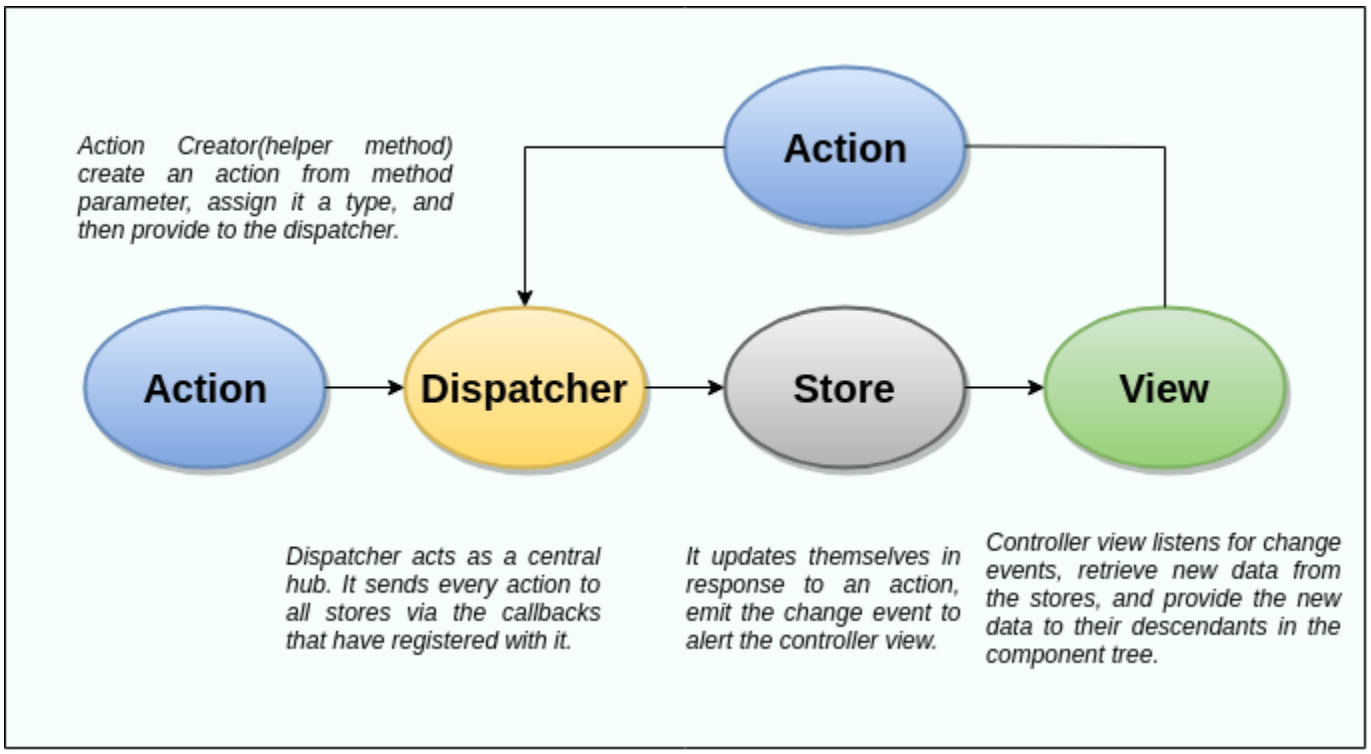
Flux is an application architecture that Facebook uses internally for building the client-side web application with React. It is not a library nor a framework. It is neither a library nor a framework. It is a kind of architecture that complements React as view and follows the concept of Unidirectional 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)

Here, you should not be confused with the Model-View-Controller (MVC) model. Although, Controllers exists in both, but Flux controller-views (views) found at the top of the hierarchy. It retrieves data from the stores and then passes this data down to their children. Additionally, action creators - dispatcher helper methods used to describe all changes that are possible in the application. It can be useful as a fourth part of the Flux update cycle.

Structure and Data Flow



In Flux application, data flows in a single direction(unidirectional). This data flow is central to the flux pattern. The dispatcher, stores, and views are independent nodes with inputs and outputs. The actions are simple objects that contain new data and type property. Now, let us look at the various components of flux architecture one by one.

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.

Advantage of Flux

- 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.

React Flux Vs. MVC

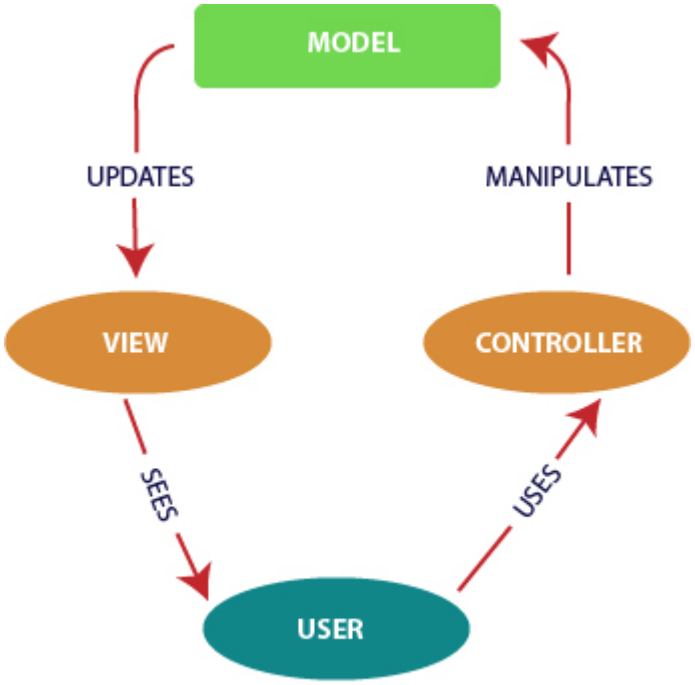
MVC

MVC stands for Model View Controller. It is an architectural pattern used for developing the user interface. It divides the application into three different logical components: the Model, the View, and the Controller. It is first introduced in 1976 in the Smalltalk programming language. In MVC, each component is built to handle specific development aspect of an application. It is one of the most used web development frameworks to create scalable projects.

MVC Architecture

The MVC architecture contains the three components. These are:

- **Model:** It is responsible for maintaining the behavior and data of an application.
- **View:** It is used to display the model in the user interface.
- **Controller:** It acts as an interface between the Model and the View components. It takes user input, manipulates the data(model) and causes the view to update.



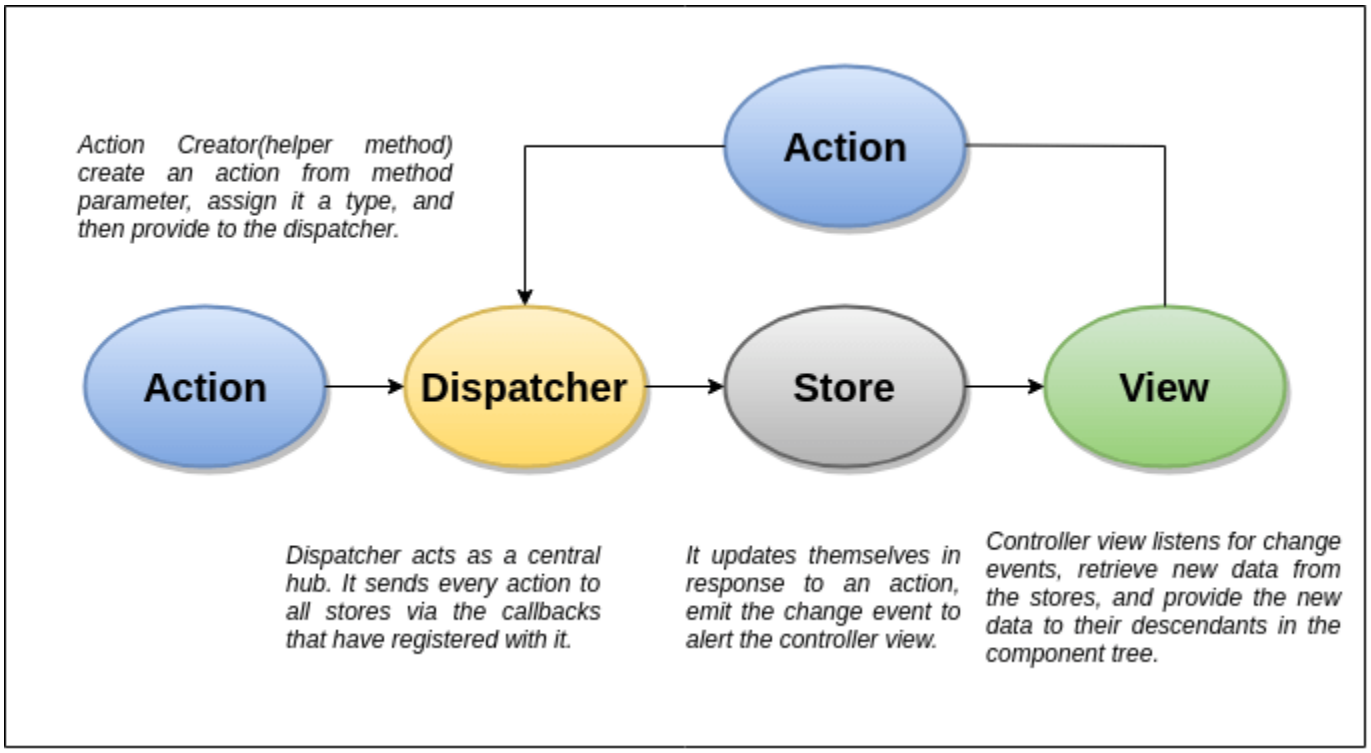
MVC Architecture

Flux

According to the official site, Flux is the application architecture that Facebook uses for building client-side web applications. It is an alternative to MVC architecture and other software design patterns for managing how data flows in the react application. It is the backbone of all React application. It is not a library nor a framework. It complements React as view and follows the concept of Unidirectional Data Flow model.

Flux Architecture has three major roles in dealing with data:

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



MVC Vs. Flux

SN	MVC	FLUX
1.	It was introduced in 1976.	It was introduced just a few years ago.
2.	It supports Bi-directional data Flow model.	It supports Uni-directional data flow mode

3.	In this, data binding is the key.	In this, events or actions are the keys.
4.	It is synchronous.	It is asynchronous.
5.	Here, controllers handle everything(logic).	Here, stores handle all logic.
6.	It is hard to debug.	It is easy to debug because it has com Dispatcher.
7.	It is difficult to understand as the project size increases.	It is easy to understand.
8.	Its maintainability is difficult as the project scope goes huge.	Its maintainability is easy and reduces runt
9.	Testing of application is difficult.	Testing of application is easy.
10.	Scalability is complex.	It can be easily scalable.

React 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.

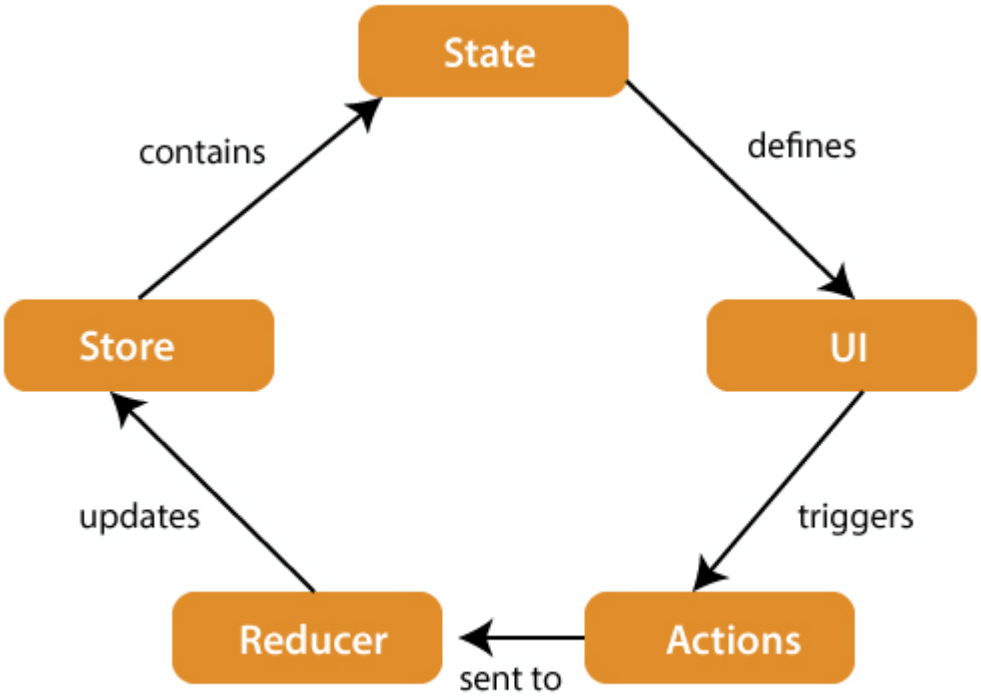
- Redux does not have Dispatcher concept.
- Redux has an only Store whereas Flux has many Stores.
- The Action objects will be received and handled directly by Store.

Why use React Redux?

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.

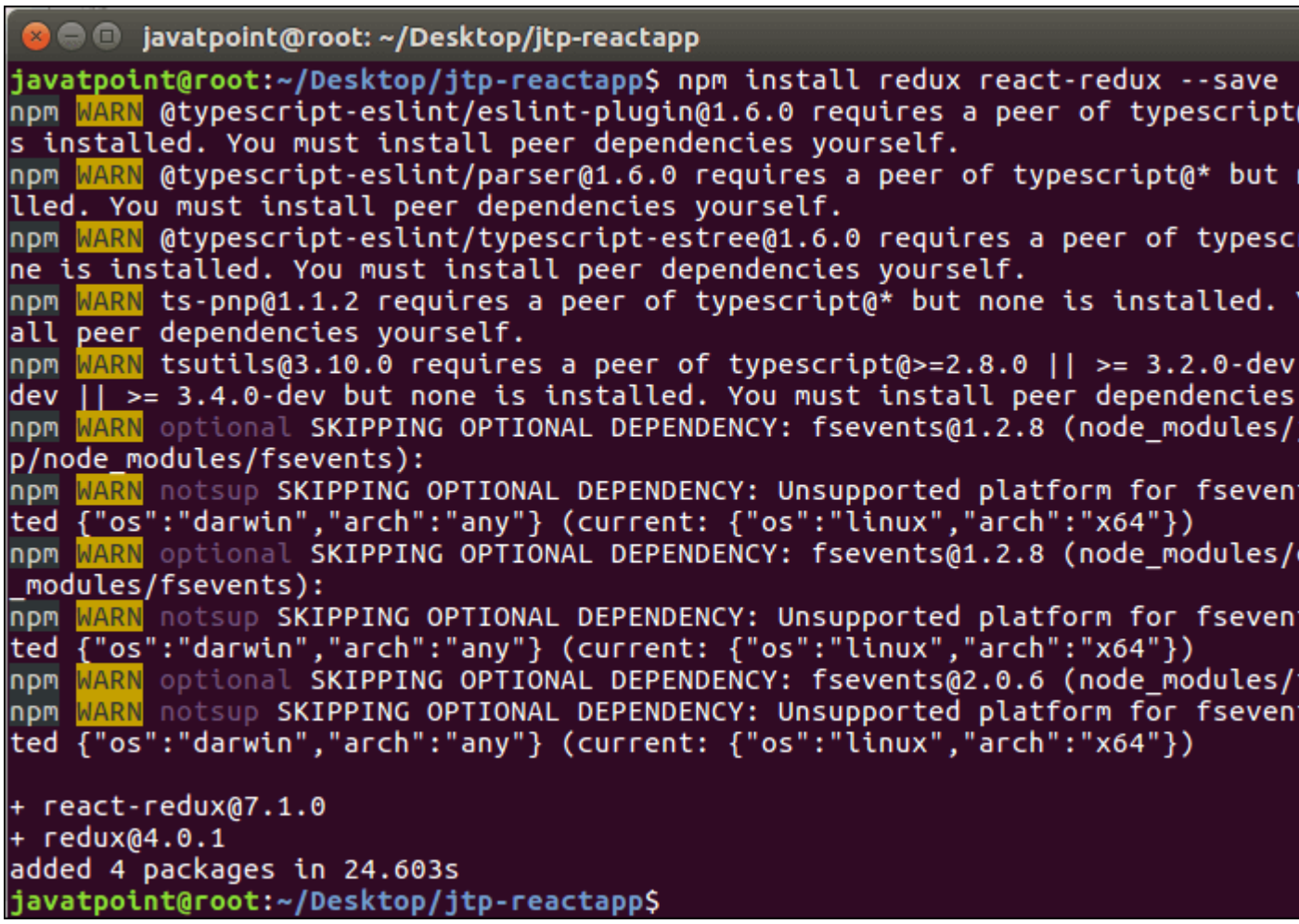
REDUCER: Reducer read the payloads from the actions and then updates the store via the state accordingly. It is a pure function to return a new state from the initial state.

Redux Installation

Requirements: React Redux requires React 16.8.3 or later version.

To use React Redux with React application, you need to install the below command.

1. $ npm install redux react-redux --save


javatpoint@root: ~/Desktop/jtp-reactapp
javatpoint@root:~/Desktop/jtp-reactapp$ npm install redux react-redux --save
npm WARN @typescript-eslint/eslint-plugin@1.6.0 requires a peer of typescript(
s installed. You must install peer dependencies yourself.
npm WARN @typescript-eslint/parser@1.6.0 requires a peer of typescript@* but
lled. You must install peer dependencies yourself.
npm WARN @typescript-eslint/typescript-estree@1.6.0 requires a peer of typesc
ne is installed. You must install peer dependencies yourself.
npm WARN ts-pnp@1.1.2 requires a peer of typescript@* but none is installed.
all peer dependencies yourself.
npm WARN tsutils@3.10.0 requires a peer of typescript@>=2.8.0 || >= 3.2.0-dev
dev || >= 3.4.0-dev but none is installed. You must install peer dependencies
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.8 (node_modules/
p/node_modules/fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fseven
ted {"os":"darwin","arch":"any"} (current: {"os":"linux","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.8 (node_modules/
_modules/fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fseven
ted {"os":"darwin","arch":"any"} (current: {"os":"linux","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.0.6 (node_modules/
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fseven
ted {"os":"darwin","arch":"any"} (current: {"os":"linux","arch":"x64"})
+ react-redux@7.1.0
+ redux@4.0.1
added 4 packages in 24.603s
javatpoint@root:~/Desktop/jtp-reactapp$

React Redux Example

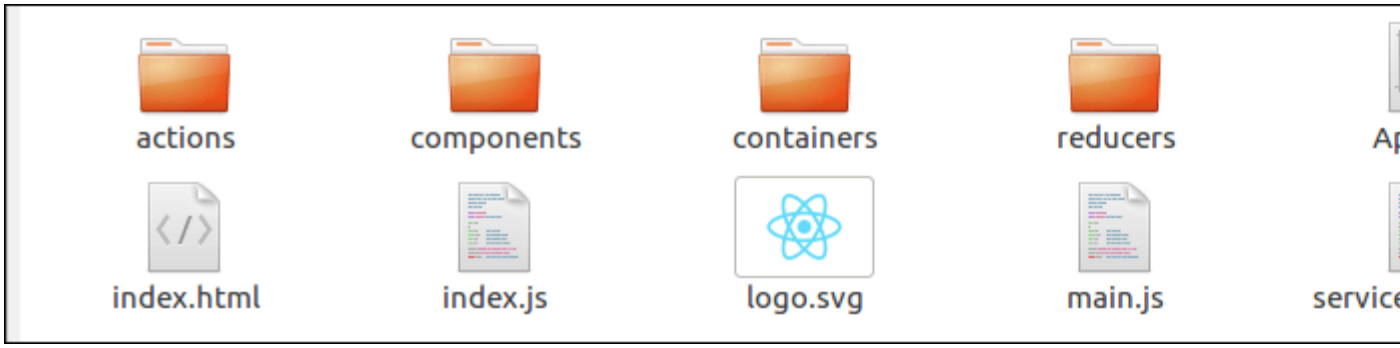
In this section, we will learn how to implements Redux in React application. Here, we provide a simple example to connect Redux and React.

Step-1 Create a new react project using **create-react-app** command. I choose the project name: "**reactproject**." Now, install **Redux** and **React-Redux**.

```
javatpoint@root:~/Desktop$ npx create-react-app reactproject
javatpoint@root:~/Desktop/reactproject$ npm install redux react-redux --save
```

Step-2 Create Files and Folders

In this step, we need to create folders and files for actions, reducers, components, and containers. After creating folders and files, our project looks like as below image.



Step-3 Actions

It uses '**type**' property to inform about data that should be sent to the **Store**. In this folder, we will create two files: **index.js** and **index.spec.js**. Here, we have created an **action creator** that returns our action and sets an **id** for every created item.

Index.js

```
let nextTodoId = 0
export const addTodo = text => ({
  type: 'ADD_TODO',
  id: nextTodoId++,
  text
})

export const setVisibilityFilter = filter => ({
  type: 'SET_VISIBILITY_FILTER',
```

```
  filter
})

export const toggleTodo = id => ({
  type: 'TOGGLE_TODO',
  id
})

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

Index.spec.js

```
import * as actions from './index'

describe('todo actions', () => {
  it('addTodo should create ADD_TODO action', () => {
    expect(actions.addTodo('Use Redux')).toEqual({
      type: 'ADD_TODO',
      id: 0,
      text: 'Use Redux'
    })
  })

  it('setVisibilityFilter should create SET_VISIBILITY_FILTER action', () => {
    expect(actions.setVisibilityFilter('active')).toEqual({
      type: 'SET_VISIBILITY_FILTER',
      filter: 'active'
    })
  })

  it('toggleTodo should create TOGGLE_TODO action', () => {
    expect(actions.toggleTodo(1)).toEqual({
      type: 'TOGGLE_TODO',
```

```
      id: 1
    })
  })
})
```

Step-4 Reducers

As we know, Actions only trigger changes in the app, and the Reducers specify those changes. The Reducer is a function which takes two parameters 'Action' and 'State' to calculate and return an updated State. It read the payloads from the 'Actions' and then updates the 'Store' via the State accordingly.

In the given files, each Reducer 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. When the app becomes larger, we can split the Reducers into separate files and keep them completely independent and managing different data domains.

Here, we are using 'combineReducers' helper function to add any new Reducers we might use in the future.

index.js

```
import { combineReducers } from 'redux'
import todos from './todos'
import visibilityFilter from './visibilityFilter'

export default combineReducers({
  todos,
  visibilityFilter
})
```

Todos.js

```
const todos = (state = [], action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          id: action.id,
          text: action.text,
```

```
9.            completed: false
10.         }
11.       ]
12.     case 'TOGGLE_TODO':
13.       return state.map(todo =>
14.         (todo.id === action.id)
15.           ? {...todo, completed: !todo.completed}
16.           : todo
17.       )
18.     default:
19.       return state
20.   }
21. }
22. export default todos
```

Todos.spec.js

```
1.  import todos from './todos'
2.
3.  describe('todos reducer', () => {
4.    it('should handle initial state', () => {
5.      expect(
6.        todos(undefined, {})
7.      ).toEqual([])
8.    })
9.
10.   it('should handle ADD_TODO', () => {
11.     expect(
12.       todos([], {
13.         type: 'ADD_TODO',
14.         text: 'Run the tests',
15.         id: 0
16.       })
17.     ).toEqual([
18.       {
19.         text: 'Run the tests',
20.         completed: false,
```

```
    id: 0
  }
])

expect(
  todos([
    {
      text: 'Run the tests',
      completed: false,
      id: 0
    }
  ], {
    type: 'ADD_TODO',
    text: 'Use Redux',
    id: 1
  })
).toEqual([
  {
    text: 'Run the tests',
    completed: false,
    id: 0
  }, {
    text: 'Use Redux',
    completed: false,
    id: 1
  }
])

expect(
  todos([
    {
      text: 'Run the tests',
      completed: false,
      id: 0
    }, {
      text: 'Use Redux',
      completed: false,
```

```
      id: 1
    }
  ], {
    type: 'ADD_TODO',
    text: 'Fix the tests',
    id: 2
  })
).toEqual([
  {
    text: 'Run the tests',
    completed: false,
    id: 0
  }, {
    text: 'Use Redux',
    completed: false,
    id: 1
  }, {
    text: 'Fix the tests',
    completed: false,
    id: 2
  }
])
})

it('should handle TOGGLE_TODO', () => {
  expect(
    todos([
      {
        text: 'Run the tests',
        completed: false,
        id: 1
      }, {
        text: 'Use Redux',
        completed: false,
        id: 0
      }
    ], {
```

```
95.         type: 'TOGGLE_TODO',
96.         id: 1
97.       })
98.     ).toEqual([
99.       {
100.                 text: 'Run the tests',
101.                 completed: true,
102.                 id: 1
103.               }, {
104.                 text: 'Use Redux',
105.                 completed: false,
106.                 id: 0
107.               }
108.             ])
109.           })
110.         })
```

VisibilityFilter.js

```
1.  import { VisibilityFilters } from '../actions'
2.
3.  const visibilityFilter = (state = VisibilityFilters.SHOW_ALL, action) => {
4.    switch (action.type) {
5.      case 'SET_VISIBILITY_FILTER':
6.        return action.filter
7.      default:
8.        return state
9.    }
10. }
11. export default visibilityFilter
```

Step-5 Components

It is a Presentational Component, which concerned with how things look such as markup, styles. It receives data and invokes callbacks exclusively via props. It does not know where the data comes from or how to change it. It only renders what is given to them.

App.js

It is the root component which renders everything in the UI.

```
import React from 'react'
import Footer from './Footer'
import AddTodo from '../containers/AddTodo'
import VisibleTodoList from '../containers/VisibleTodoList'

const App = () => (
  <div>
    <AddTodo />
    <VisibleTodoList />
    <Footer />
  </div>
)
export default App
```

Footer.js

It tells where the user changes currently visible **todos**.

```
import React from 'react'
import FilterLink from '../containers/FilterLink'
import { VisibilityFilters } from '../actions'

const Footer = () => (
  <p>
    Show: <FilterLink filter={VisibilityFilters.SHOW_ALL}>All</FilterLink>
    {', '}
    <FilterLink filter={VisibilityFilters.SHOW_ACTIVE}>Active</FilterLink>
    {', '}
    <FilterLink filter={VisibilityFilters.SHOW_COMPLETED}>Completed</FilterLink>
  </p>
)
export default Footer
```

Link.js

It is a link with a callback.

```
import React from 'react'
import PropTypes from 'prop-types'

const Link = ({ active, children, onClick }) => {
  if (active) {
    return <span>{children}</span>
  }

  return (
    <a
      href=""
      onClick={e => {
        e.preventDefault()
        onClick()
      }}
    >
      {children}
    </a>
  )
}

Link.propTypes = {
  active: PropTypes.bool.isRequired,
  children: PropTypes.node.isRequired,
  onClick: PropTypes.func.isRequired
}

export default Link
```

Todo.js

It represents a single todo item which shows **text**.

```
import React from 'react'
import PropTypes from 'prop-types'

const Todo = ({ onClick, completed, text }) => (
  <li
```

```
      onClick={onClick}
      style={{
        textDecoration: completed ? 'line-through' : 'none'
      }}
    >
      {text}
    </li>
)

Todo.propTypes = {
  onClick: PropTypes.func.isRequired,
  completed: PropTypes.bool.isRequired,
  text: PropTypes.string.isRequired
}

export default Todo
```

TodoList.js

It is a list to show visible todos{ id, text, completed }.

```
import React from 'react'
import PropTypes from 'prop-types'
import Todo from './Todo'

const TodoList = ({ todos, onTodoClick }) => (
  <ul>
    {todos.map((todo, index) => (
      <Todo key={index} {...todo} onClick={() => onTodoClick(index)} />
    ))}
  </ul>
)

TodoList.propTypes = {
  todos: PropTypes.arrayOf(
    PropTypes.shape({
      id: PropTypes.number.isRequired,
      completed: PropTypes.bool.isRequired,
```

```
    text: PropTypes.string.isRequired
  }).isRequired
).isRequired,
onTodoClick: PropTypes.func.isRequired
}
export default TodoList
```

Step-6 Containers

It is a Container Component which concerned with how things work such as data fetching, updates State. It provides data and behavior to presentational components or other container components. It uses Redux State to read data and dispatch Redux Action for updating data.

AddTodo.js

It contains the input field with an ADD (submit) button.

```
import React from 'react'
import { connect } from 'react-redux'
import { addTodo } from '../actions'

const AddTodo = ({ dispatch }) => {
  let input

  return (
    <div>
      <form onSubmit={e => {
        e.preventDefault()
        if (!input.value.trim()) {
          return
        }
        dispatch(addTodo(input.value))
        input.value = ''
      }}>
        <input ref={node => input = node} />
        <button type="submit">
          Add Todo
        </button>
```

```
      </form>
    </div>
  )
}
export default connect()(AddTodo)
```

FilterLink.js

It represents the current visibility filter and renders a link.

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => ({
  active: ownProps.filter === state.visibilityFilter
})

const mapDispatchToProps = (dispatch, ownProps) => ({
  onClick: () => dispatch(setVisibilityFilter(ownProps.filter))
})

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)
```

VisibleTodoList.js

It filters the todos and renders a TodoList.

```
import { connect } from 'react-redux'
import { toggleTodo } from '../actions'
import TodoList from '../components/TodoList'
import { VisibilityFilters } from '../actions'

const getVisibleTodos = (todos, filter) => {
  switch (filter) {
    case VisibilityFilters.SHOW_ALL:
```

```
      return todos
    case VisibilityFilters.SHOW_COMPLETED:
      return todos.filter(t => t.completed)
    case VisibilityFilters.SHOW_ACTIVE:
      return todos.filter(t => !t.completed)
    default:
      throw new Error('Unknown filter: ' + filter)
  }
}

const mapStateToProps = state => ({
  todos: getVisibleTodos(state.todos, state.visibilityFilter)
})

const mapDispatchToProps = dispatch => ({
  toggleTodo: id => dispatch(toggleTodo(id))
})

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)
```

Step-7 Store

All container components need access to the Redux Store to subscribe to it. For this, we need to pass it(store) as a prop to every container component. However, it gets tedious. So we recommend using special React Redux component called which make the store available to all container components without passing it explicitly. It used once when you render the root component.

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './components/App'
import rootReducer from './reducers'
```

```
7.
8.  const store = createStore(rootReducer)
9.
10. render(
11.   <Provider store={store}>
12.     <App />
13.   </Provider>,
14.   document.getElementById('root')
15. )
```

Output

When we execute the application, it gives the output as below screen.

Now, we will be able to add items in the list.

The detailed explanation of React-Redux example can be shown here: https://redux.js.org/basics/usage-with-react.

React Portals

The **React 16.0** version introduced React portals in **September 2017**. A React portal provides a way to render an element outside of its component hierarchy, i.e., in a separate component.

Before React 16.0 version, it is very tricky to render the child component outside of its parent component hierarchy. If we do this, it breaks the convention where a component needs to render as a new element and follow a **parent-child** hierarchy. In React, the parent component always wants to go where its child component goes. That's why React portal concept comes in.

Syntax

```
ReactDOM.createPortal(child, container)
```

Here, the first argument (child) is the component, which can be an element, string, or fragment, and the second argument (container) is a DOM element.

Example before React v16

Generally, when you want to return an element from a component's render method, it is mounted as a new div into the DOM and render the children of the closest parent component.

```
render() {
// React mounts a new div into the DOM and renders the children into it
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

Example using portal

But, sometimes we want to insert a child component into a different location in the DOM. It means React does not want to create a new div. We can do this by creating React portal.

```
render() {
 return ReactDOM.createPortal(
```

```
3.      this.props.children,
4.      myNode,
5.    );
6.  }
```

Features

- It uses React version 16 and its official API for creating portals.
- It has a fallback for React version 15.
- It transports its children component into a new React portal which is appended by default to document.body.
- It can also target user specified DOM element.
- It supports server-side rendering
- It supports returning arrays (no wrapper div's needed)
- It uses <Portal /> and <PortalWithState /> so there is no compromise between flexibility and convenience.
- It doesn't produce any DOM mess.
- It has no dependencies, minimalistic.

When to use?

The common use-cases of React portal include:

- Modals
- Tooltips
- Floating menus
- Widgets

Installation

We can install React portal using the following command.

```
1.  $ npm install react-portal --save
```

Explanation of React Portal

Create a new React project using the following command.

```
$ npx create-react-app reactapp
```

Open the App.js file and insert the following code snippet.

App.js

```
import React, {Component} from 'react';
import './App.css'
import PortalDemo from './PortalDemo.js';

class App extends Component {
    render () {
        return (
            <div className='App'>
         <PortalDemo />
    </div>
        );
    }
}
export default App;
```

The next step is to create a **portal** component and import it in the App.js file.

PortalDemo.js

```
import React from 'react'
import ReactDOM from 'react-dom'

function PortalDemo(){
    return ReactDOM.createPortal(
      <h1>Portals Demo</h1>,
      document.getElementById('portal-root')
    )
}
export default PortalDemo
```

Now, open the Index.html file and add a element to access the child component outside the root node.

Index.html

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.    <head>
4.      <meta charset="utf-8" />
5.      <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
6.      <meta name="viewport" content="width=device-width, initial-scale=1" />
7.      <meta name="theme-color" content="#000000" />
8.      <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
9.      <title>React App</title>
10.   </head>
11.   <body>
12.     <noscript>It is required to enable JavaScript to run this app.</noscript>
13.     <div id="root"></div>
14.     <div id="portal-root"></div>
15.   </body>
16. </html>
```

Output:

When we execute the React app, we will get the following screen.

Now, open the **Inspect** (ctrl + shift + I). In this window, select the **Elements** section and then click on the component. Here, we can see that

each tag is under the "portal-root" DOM node, not the "root" DOM node. Hence, we can see that how React Portal provides the ability to break out of root DOM tree.