استفاده از Flux در ReactJS

در این بخش می خواهیم به چگونگی پیاده سازی الگوی فلاکس در برنامه های ری اکت بپردازیم. برای انجام این کار از فریمورک Redux استفاده می کنیم. هدف این فصل ارائه ی ساده ترین مثال از تمامی بخش های مورد نیاز جهت اتصال Redux و ری اکت است.

مرحله 1 – نصب Redux

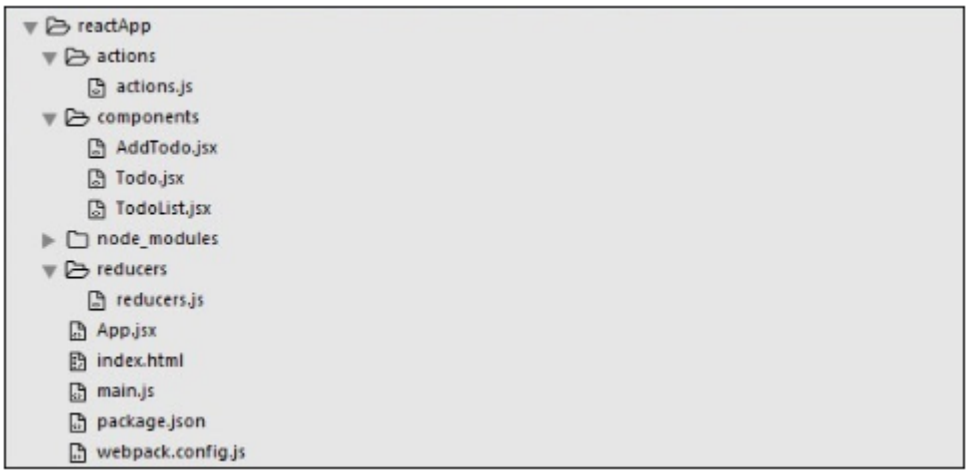
Redux را از طریق پنجره ی cmd نصب می کنیم.

```
C:\Users\username\Desktop\reactApp>npm install --save react-redux
```

مرحله 2 – ایجاد فایل ها و فولدرها

در این بخش می خواهیم پوشه ها و فولدرهای مربوط به action ها، reducer ها و component های خود را ایجاد کنیم. بعد از انجام این کار ساختار پوشه ها به شکل زیر درخواهد آمد.

```
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir actions
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir components
C:\Users\Tutorialspoint\Desktop\reactApp>mkdir reducers
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > actions/actions.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > reducers/reducers.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/AddTodo.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/Todo.js
C:\Users\Tutorialspoint\Desktop\reactApp>type nul > components/TodoList.js
```



مرحله 3 – Action ها

اکشن ها اشیاء جاوا اسکریپتی هستند که جهت اطلاع دادن به داده برای ارسال شدن به store از ویژگی type استفاده می کنند. ما می خواهیم عمل ADD_TODO را تعریف کنیم. این عمل در اضافه کردن آیتم جدید به لیست کاربرد دارد. تابع addTodo ایجاد کننده ی عملی است که عمل ما را برگشت می دهد و برای هر یک از آیتم های ایجاد شده شناسه ای را تنظیم می کند.

actions/actions.js

```
export const ADD_TODO = 'ADD_TODO'


let nextTodoId = 0;


export function addTodo(text) {
  return {
    type: ADD_TODO,
    id: nextTodoId++,
    text
  };
```

```
}
```

مرحله 4 – Reducer ها

در حالی که عمل ها تنها تغییرات را در برنامه فعال می کنند، reducer ها این تغییرات را مشخص می کنند. ما برای جستجوی عمل ADD_TODO از دستور switch استفاده می کنیم. reducer تابعی است که جهت محاسبه و برگشت یک حالت به روز شده دو پارامتر (state و action) را می گیرد.

تابع اول در ایجاد آیتم جدید و تابع دوم در ارسال این آیتم به لیست کاربرد دارد. در همین راستا ما از تابع کمکی combineReducers استفاده می کنیم. در این تابع می توانیم هر reducer جدیدی را برای روز مبادا اضافه کنیم.

reducers/reducers.js

```
import { combineReducers } from 'redux'
import { ADD_TODO } from '../actions/actions'


function todo(state, action) {
  switch (action.type) {
    case ADD_TODO:
      return {
        id: action.id,
        text: action.text,
      }
    default:
      return state
  }
}
```

```
function todos(state = [], action) {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        todo(undefined, action)
      ]
    default:
      return state
  }
}
const todoApp = combineReducers({
  todos
})
export default todoApp
```

مرحله 5 ـ Store

Store مکانی است که در آن حالت برنامه نگهداری می شود. بعد از در اختیار داشتن reducer ها، ایجاد یک Store کار آسانی است. ما ویژگی Store را به عنصر provider عبور می دهیم که این عنصر جزء route ما را پوشش می دهد.

main.js

```
import React from 'react'


import { render } from 'react-dom'
```

```
import { createStore } from 'redux'

import { Provider } from 'react-redux'


import App from './App.jsx'

import todoApp from './reducers/reducers'


let store = createStore(todoApp)

let rootElement = document.getElementById('app')


render(

  <Provider store = {store}>

    <App />

  </Provider>,


  rootElement

)
```

مرحله 6 – جزء اصلی

جزء App، جزء اصلی برنامه است. تنها جزء اصلی باید نسبت به Redux آگاه باشد. بخش مهمی که باید به آن توجه کرد، تابع connect است. این تابع جهت اتصال جزء اصلی App به store کاربرد دارد.

این تابع، تابع select را به عنوان یک آرگومان می گیرد. تابع select حالت را از store گرفته و ویژگی هایی (visibleTodos) را برگشت می دهد که از آن ها می توانیم در اجزای خود استفاده کنیم.

App.jsx

```
import React, { Component } from 'react'
```

```
import { connect } from 'react-redux'

import { addTodo } from './actions/actions'


import AddTodo from './components/AddTodo.js'

import TodoList from './components/TodoList.js'


class App extends Component {

  render() {

    const { dispatch, visibleTodos } = this.props


    return (

      <div>

        <AddTodo onAddClick = {text =>dispatch(addTodo(text))} />

        <TodoList todos = {visibleTodos}/>

      </div>

    )

  }

}

function select(state) {

  return {

    visibleTodos: state.todos

  }

}

export default connect(select)(App);
```

مرحله 7 – اجزای دیگر

این اجزا نباید نسبت به Redux آگاه باشند.

components/AddTodo.js

```
import React, { Component, PropTypes } from 'react'


export default class AddTodo extends Component {
  render() {
    return (
      <div>
        <input type = 'text' ref = 'input' />


        <button onClick = {(e) => this.handleClick(e)}>
          Add
        </button>
      </div>
    )
  }
  handleClick(e) {
    const node = this.refs.input
    const text = node.value.trim()
    this.props.onAddClick(text)
    node.value = ''
  }
}
```

components/Todo.js

```
import React, { Component, PropTypes } from 'react'
```

```
export default class Todo extends Component {
  render() {
    return (
      <li>
        {this.props.text}
      </li>
    )
  }
}
```

components/TodoList.js

```
import React, { Component, PropTypes } from 'react'
import Todo from './Todo.js'

export default class TodoList extends Component {
  render() {
    return (
      <ul>
        {this.props.todos.map(todo =>
          <Todo
            key = {todo.id}
            {...todo}
          />
        )}
      </ul>
    )
```

```
  }

}
```

بعد از اجرای برنامه می توانیم آیتم هایی را به لیست خود اضافه کنیم.