SmartCliff

CAREER MOBILITY SOLUTIONS

We are on a mission to address the digital skills gap for 10 Million+ young professionals, train and empower them to forge a career path into future tech

Redux

Introduction to Redux

Introduction to Redux

- React is one of the most popular JavaScript libraries which is used for front-end development.
- It has made our application development easier and faster by providing a component-based approach.
- As we might know, it’s not the complete framework but just the view part of the MVC (Model-View-Controller) framework.
- So, how do we keep track of the data and handle the events in the applications developed using React?
- Well, this is where Redux comes as a savior and handles the data flow of the application from the backend.
- Redux is quite an excellent State Management Framework usually used with React.js library

Redux Installation

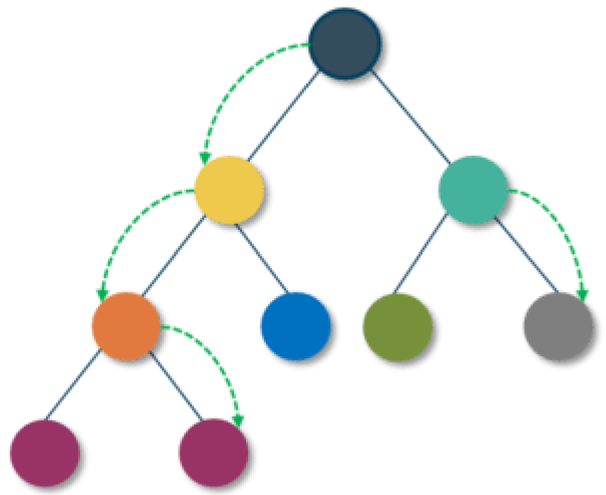
Requirements:

- React Redux requires React 16.8.3 or later version.
- Before installing Redux, **we have to install Nodejs and NPM.**
- To use React Redux with React application, to install use the below command**.**

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

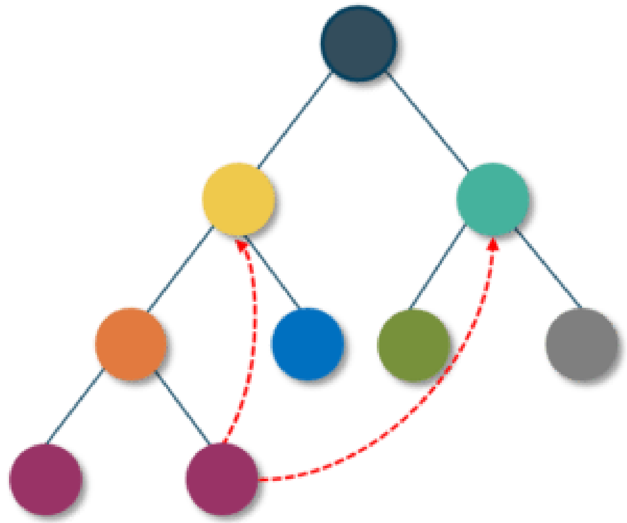
Why Redux with React?

- React follows the component-based approach, where the data flows through the components.
- In fact, the data in React always flows from parent to child components which makes it unidirectional.
- This surely keeps our data organized and helps us in controlling the application better.
- Because of this, the application's state is contained in specific stores and as a result, the rest of the components remain loosely coupled.
- This makes our application more flexible leading to increased efficiency.
- That's why communication from a parent component to a child component is convenient.



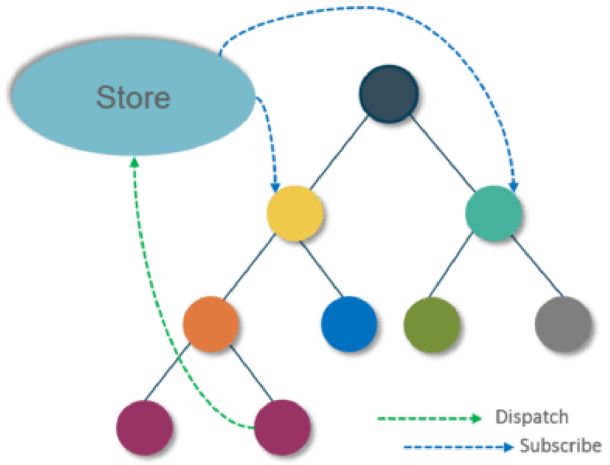
Why Redux with React?

- But what happens when we try to communicate from a non-parent component?
- A child component can never pass data back up to the parent component.
- React does not provide any way for direct component-to-component communication.
- Even though React has features to support this approach, it is considered to be a poor practice.
- So, how can two non-parent components pass data to each other?



Why Redux with React?

- This is where React fails to provide a solution and Redux comes into the picture.
- Redux provides a **“store”** as a solution to this problem. A store is a place where you can store all your **application state** together.
- Now the components can **“dispatch”** state changes to the store and not directly to the other components.
- Then the components that need the **updates** about the **state changes** can **“subscribe”** to the store.
- Thus, with **Redux,** it becomes clear where the components get their state from as well as where should they send their states to.
- Now the component initiating the change does not have to worry about the **list of components** needing the state change and can simply **dispatch** the change to the store. This is how Redux makes the data flow **easier.**



What is Redux ?

- Just like React, Redux is also a library which is used widely for front-end development.
- It is basically a tool for managing both data-state and UI-state in JavaScript applications. Redux separates the application data and business logic into its own container in order to let React manage just the view.
- Rather than a traditional library or a framework, it’s an application data-flow architecture. It is most compatible with Single Page Applications (SPAs) where the management of the states over time can get complex.
- Redux is the predictable state container for JavaScript applications.

What is Redux ?

- Redux is following the Unidirectional flow.
- Redux has a Single Store.
- Redux can not have multiple stores.
- The store is divided into various state objects.
- So, all we need is to maintain the single store, or we can say the only source of truth.

Principles of Redux

Redux follows three fundamental principles:

- **Single source of truth**
- **State is read-only**
- **Changes are made with pure functions**

- It is the state of our whole application is stored in an object within a single store.
- There is an only way to change the state is to emit an action, an object describing what happened.
- To specify how actions transform the state, we write pure reducers.

Principles of Redux : Single source of truth

- The state of the entire application is stored in an object/ state tree within a single store.
- The single state tree makes it easier to keep track of the changes over time and debug or inspect the application.
- For a faster development cycle, it helps to persist the application’s state in development.

Principles of Redux : State is read-only

- The only way to change the state is to trigger an action.
- An action is a plain JS object describing the change.
- Just like the state is the minimal representation of data, the action is the minimal representation of the change to that data.
- An action must have a type property (conventionally a String constant).
- All the changes are centralized and occur one by one in a strict order.

Principles of Redux : Changes are made with pure functions

- In order to specify how the state tree is transformed by actions, we need pure functions.
- Pure functions are those whose return values depend solely on the values of their arguments.
- Reducers are just pure functions that take the previous state and an action and return the next state.
- We can have a single reducer in our application and as it grows, we can split it off into smaller reducers.
- These smaller reducers will then manage specific parts of the state tree.

Advantages of Redux

Following are some of the major advantages of Redux:

- **Predictability of outcome** – Since there is always one source of truth, i.e. the store, there is no confusion about how to sync the current state with actions and other parts of the application.
- **Maintainability –** The code becomes easier to maintain with a predictable outcome and strict structure.
- **Server side rendering** – We just need to pass the store that is created on the server, to the client side. This is very useful for initial render and provides a better user experience as it optimizes the application performance.
- **Developer tools** – From actions to state changes, developers can track everything going on in the application in real time.

Advantages of Redux

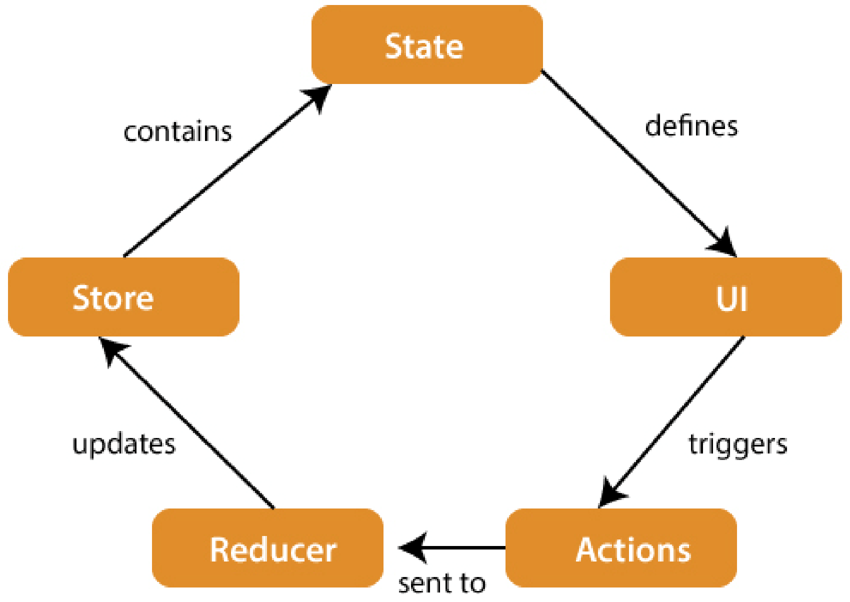
Following are some of the major advantages of Redux:

- **Community and ecosystem** – Redux has a huge community behind it which makes it even more captivating to use. A large community of talented individuals contribute to the betterment of the library and develop various applications with it.
- **Ease of testing** – Redux code are mostly functions which are small, pure and isolated. This makes the code testable and independent.
- **Organization** – Redux is very precise about how the code should be organized, this makes the code more consistent and easier when a team works with it

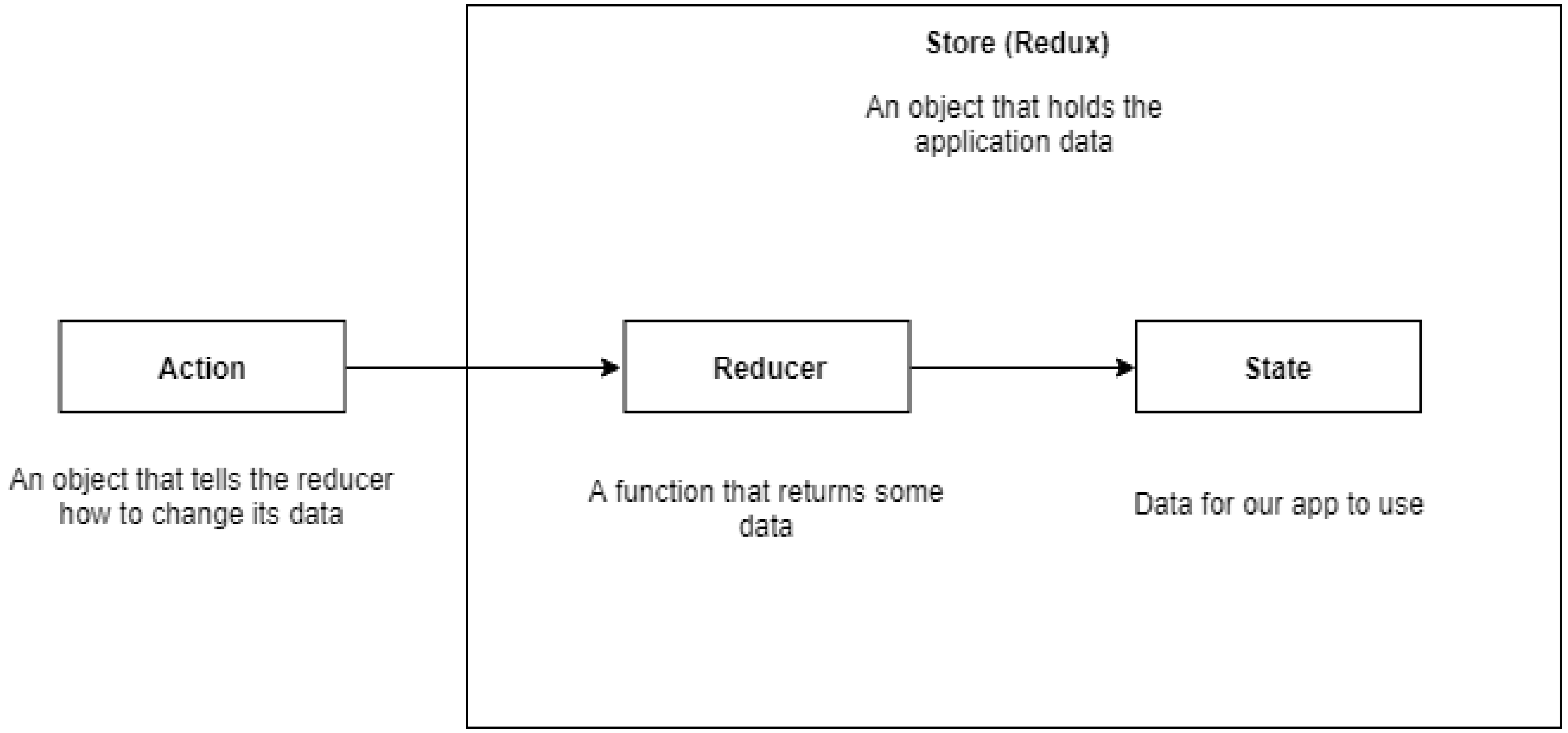
Pillars of Redux

Redux has **three** main pillars:

- **Store :** A Store is a place where the entire state of your application lists.
- **Action :** Action is sent or dispatched from the view which are payloads that can be read by Reducers.
- **Reducers :** Reducer read the payloads from the actions and then updates the store via the state accordingly.



Pillars of Redux


Store (Redux)
An object that holds the application data
Action
Reducer
State
An object that tells the reducer how to change its data
A function that returns some data
Data for our app to use

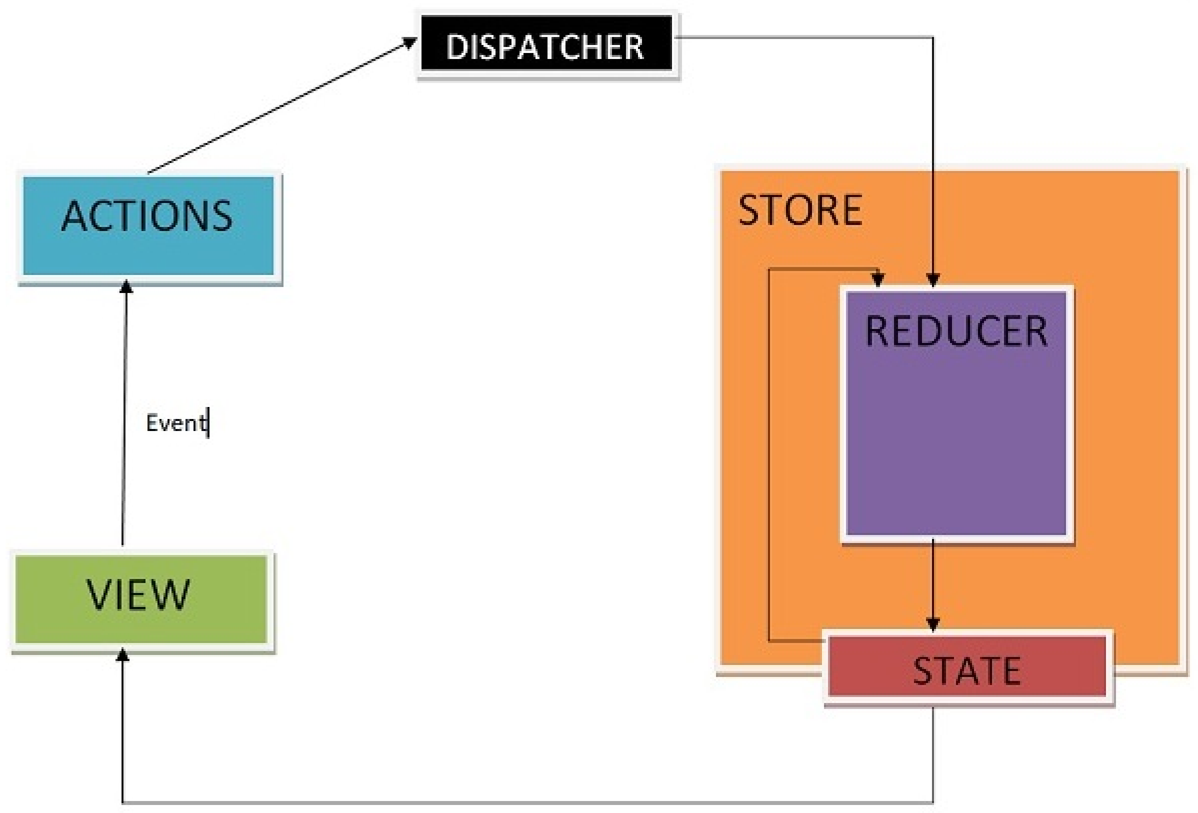
Basic Redux Data Flow

Redux - Data Flow

- Redux follows the unidirectional data flow.
- It means that your application data will follow in one-way binding data flow.
- As the application grows & becomes complex, it is hard to reproduce issues and add new features if we have no control over the state of your application.
- Redux reduces the complexity of the code, by enforcing the restriction on how and when state update can happen.
- This way, managing updated states is easy.

Redux - Data Flow

- Following diagram will help we understand Redux data flow better −



Redux - Data Flow

- An action is dispatched when a user interacts with the application.
- The root reducer function is called with the current state and the dispatched action.
- The root reducer may divide the task among smaller reducer functions, which ultimately returns a new state.
- The store notifies the view by executing their callback functions.
- The view can retrieve updated state and re-render again.

The Redux Store

Store

- A store is an object that brings them together.
- A 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);
    - It handles unregistering of listeners via the function returned by subscribe(listener)

Store

- The entire state/ object tree of an application is saved in a single store.
- As a result of this, Redux is very simple and predictable.
- We can pass middleware to the store to handle the processing of data as well as to keep a log of various actions that change the state of stores.
- To create the store, we make use of createStore from redux and pass our reducer as an argument in it as shown below :

```
import { createStore } from 'redux’
 import todoApp from './reducers’
  let store = createStore(reducer);
```

Store

- A createStore function can have three arguments. The following is the syntax −

```
createStore(reducer, [preloadedState], [enhancer])
```

- A reducer is a function that returns the next state of app.
- A preloadedState is an optional argument and is the initial state of our app.
- An enhancer is also an optional argument. It will help we enhance store with third-party capabilities.

Store

- A store has three important methods as given below −

getState

- It helps you retrieve the current state of your Redux store.
- The syntax for getState is as follows −

```
store.getState()
```

dispatch

- It allows you to dispatch an action to change a state in your application.
- The syntax for dispatch is as follows −

```
store.dispatch({type:'ITEMS_REQUEST'})
```

Store

subscribe

- It helps you register a callback that Redux store will call when an action has been dispatched. As soon as the Redux state has been updated, the view will re-render automatically.
- The syntax for dispatch is as follows −

```
store.subscribe(()=>{ console.log(store.getState());})
```

Note that subscribe function returns a function for unsubscribing the listener. To unsubscribe the listener, we can use the below code −

```
const unsubscribe = store.subscribe(()=>{console.log(store.getState());});
unsubscribe();
```

The Redux Action

Action

- An action is a plain object that represents an intention to change the state.
- They must have a property to indicate the type of action to be carried out.
    - Actions are payloads of information that send data from your application to your store.
    - Any data, whether from UI events or network callbacks, needs to eventually be dispatched as actions.
    - Actions must have a type field, indicating the type of action being performed

Action

- Actions must have a type property that indicates the type of action being performed.
- Types should typically be defined as string constants.

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

- The only way to change state content is by emitting an action.
- Actions are the plain JavaScript objects which are the main source of information used to send data (user interactions, internal events such as API calls, and form submissions) from the application to the store.
- The store receives information only from the actions.
- You have to send the actions to the store using **store.dispatch().**

Action

- Internal actions are simple JavaScript objects that have a type property (usually String constant), describing the type of action and the entire information being sent to the store .

```
{
    type: ADD_TODO,
    text
}
```

- Actions are created using action creators which are the normal functions that return actions.

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

- To call actions anywhere in the app, use dispatch()method:

```
dispatch(addTodo(text));
```

Redux Reducers

Reducers

- Actions describe the fact that something happened, but don’t specify how the application’s state changes in response.
- This is the job of reducers.
- It is based on the array reduce method, where it accepts a callback (reducer) and lets we get a single value out of multiple values, sums of integers, or an accumulation of streams of values.
- In Redux, reducers are functions (pure) that take the current state of the application and an action and then return a new state.
- Understanding how reducers work is important because they perform most of the work.

Reducers

- Reducers are pure functions that specify how the application's state changes in response to actions sent to the store.
  - Actions only describe what happened, not how the application's state changes.
  - A reducer is a function that accepts the current state and action, and returns a new state with the action performed.
  - **combineReducers()** utility can be used to combine all the reducers in the app into a single index reducer which makes maintainability much easier.

combinedReducer

- The combinedReducer is a function that takes a **hash of Reducers** and return as a single reducer.
- The resulting reducer represents an object of the **same shape as the hash.**
- The state produced by **combineReducers()** namespaces the states of each reducer under their keys as passed to **combineReducers().**

```
import { combineReducers } from 'redux’;

import Reducer1 from './Reducer1';

import Reducer2 from './Reducer2';


export default combineReducers({
   reducer1: Reducer1,
   reducer2: Reducer2
});
```

Connecting Redux to React

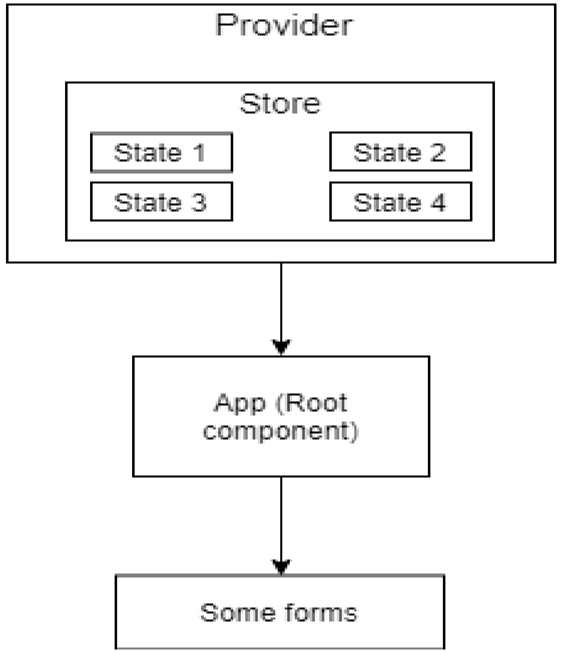
Redux - Integrate React

- Redux provides the react-redux package to bind react components with two utilities as given below −
    - Provider
    - Connect
- Provider makes the store available to rest of the application.
- Connect function helps react component to connect to the store, responding to each change occurring in the store’s state.

Provider

- Provider is React component given by **React-Redux** to makes the **Redux store** available to the rest of your app without passing it explicitly.
- You only need to use it once when you render the root component:

```
Index.js
import React, { Component } from 'react';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import reducers from './src/reducers';
import App from './components/App'

export default class App extends Component {
  render() {
    return (
      <Provider store={createStore(reducers)}>
        //Root component
         <App />
      </Provider>
    );
  }
}
```



connect()

- **connect()** is a function that is used to connect a React component to a Redux store.
- connect is do mainly two jobs here first it provides its connected component with the pieces of the data it needs from the store, and then the functions it can use to dispatch actions to the store.
- For an example I want to connect redux store with sampleComponent:

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

connect()

- The **mapStateToProps** and **mapDispatchToProps** deals with Redux store's state and dispatch, respectively.
- **state and dispatch** will be supplied to your **mapStateToProps** or **mapDispatchToProps** functions as the first argument.
- If you don't want to subscribe, pass null or undefined in place of **mapStateToProps** or **mapDispatchToProps** in connect function.

```
export default connect(null,null ) (sampleComponent)
```

mapStateToProps

- **if mapStateToProps** is specified that any time the store is updated, **mapStateToProps** will be called.
- The results of **mapStateToProps** must be a plain object, which will be merged into the component's props.
- There are maximum of **two** parameters that we can passed in to **mapStateToProps**.
- It will be called with the **store state** as the **first parameter** and the **props** passed to the **connected component** as the **second parameter**.

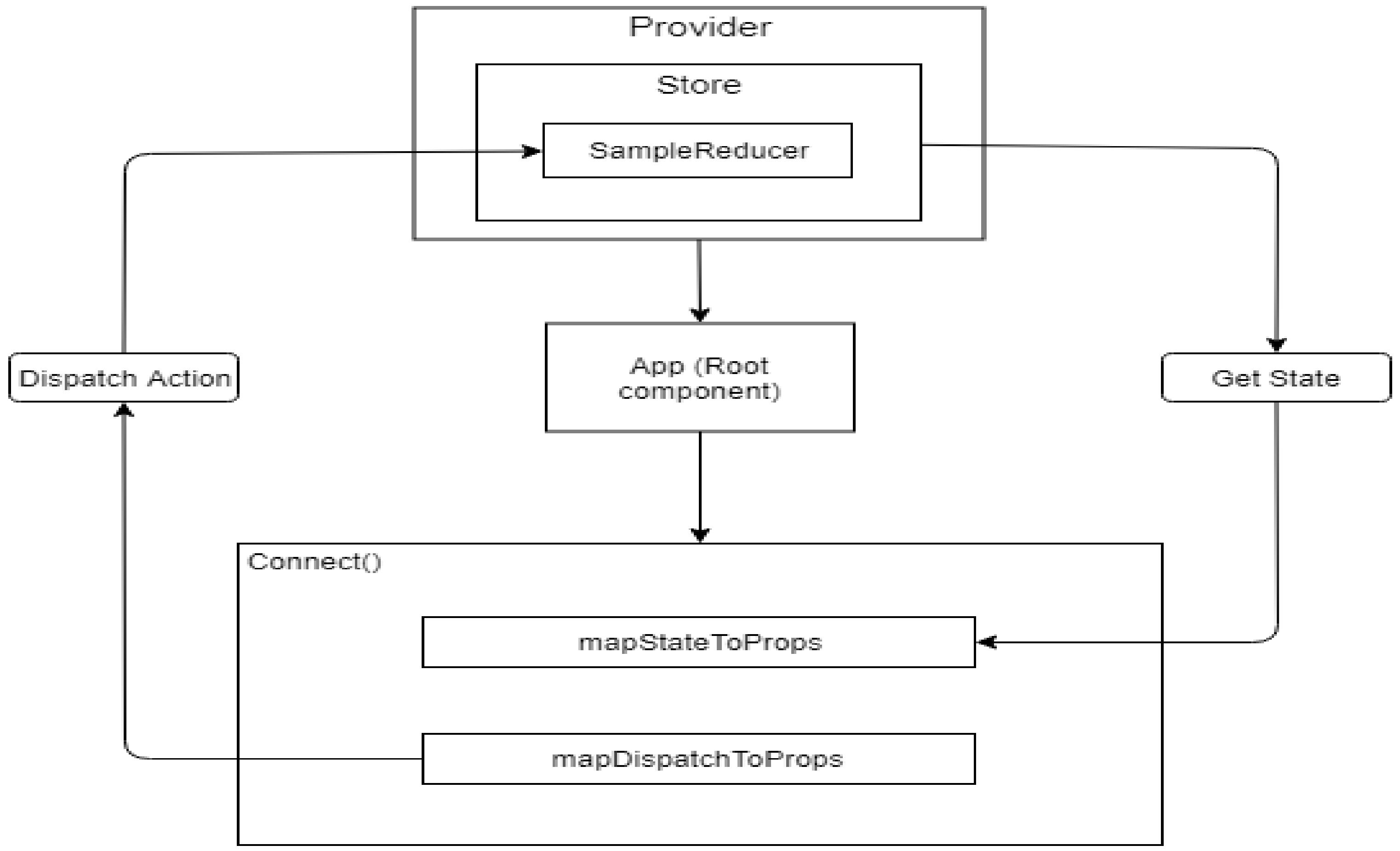
```
   const mapStateToProps = (state, ownProps) => {

    console.log(state); // Redux store state

    console.log(ownProps); // ownProps

}
```

mapDispatchToProps

- **mapDispatchToProps** is used for dispatching actions to the **store. dispatch** is a function of the **Redux store.**
- **store.dispatch** to dispatch an action

```
// return action object
const mapDispatchToProps = () => {
return {
        type: 'add_character',
        payload: 'a'
       }
}
```

Redux Flow Diagram



Redux-Devtools

Redux-Devtools

- Redux-Devtools provide us debugging platform for Redux apps.
- It allows us to perform time-travel debugging and live editing.
- Some of the features in official documentation are as follows −
    - It lets you inspect every state and action payload.
    - It lets you go back in time by “cancelling” actions.
    - If you change the reducer code, each “staged” action will be re-evaluated.
    - If the reducers throw, we can identify the error and also during which action this happened.
    - With **persistState() store enhancer**, you can persist debug sessions across page reloads**.**

Redux-Devtools

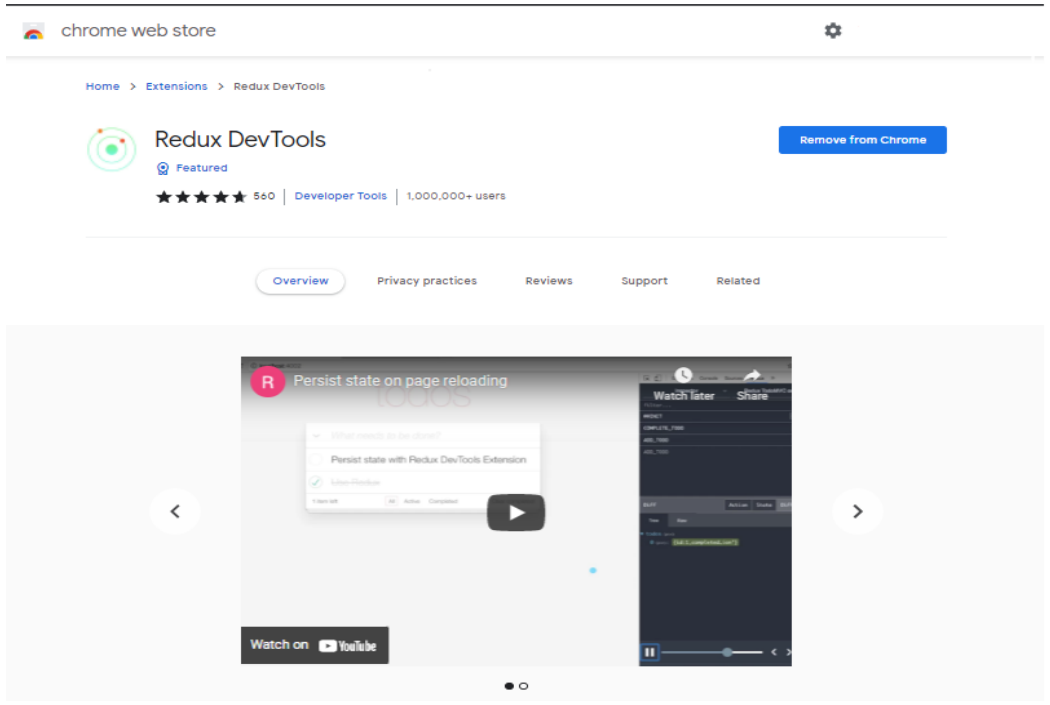
There are **two** variants of **Redux dev-tools** as given below −

- **Redux DevTools** − It can be installed as a package and integrated into your application as given below − https://github.com/reduxjs/redux-devtools/blob/master/docs/Walkthrough.md#manual-integration
- **Redux DevTools Extension** − A browser extension that implements the same developer tools for Redux is as follows − https://github.com/zalmoxisus/redux-devtools-extension

Redux-Devtools

- Download and install **Redux DevTools** from the browser.

- Add the extension to the browser as shown below.



Installation and Setup of DevTools dependencies

- To install the dependencies of **Redux DevTools**, all we need to do is to seek some commands into our system.
- The development dependencies can be served with the below commands.

```
npm install --save-dev-devtools-extension
```

- After this step, we should be importing the compose functionality from Redux DevTools using

```
import { composeWithDevTools } from 'redux-devtools-extension';
```

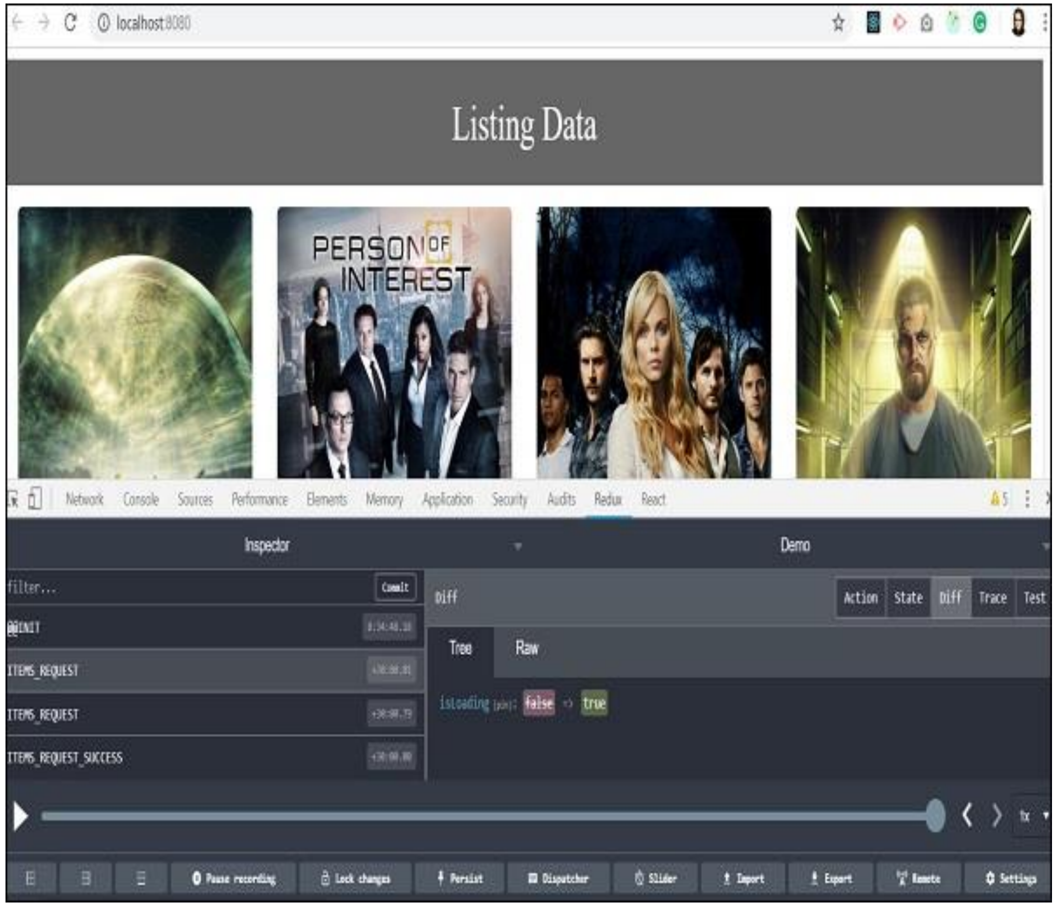
Installation and Setup of DevTools dependencies

- The next step is to create a store from using this compose method as shown below

```
import { createStore, applyMiddleware } from "redux";
import { composeWithDevTools } from "redux-devtools-extension";
import thunk from "redux-thunk";
import rootReducer from "./reducers";

const initialState = {};
const middleware = [thunk];

const store = createStore(
  rootReducer,
  initialState,
  composeWithDevTools(applyMiddleware(...middleware))
);
export default store;
```

Redux-Devtools

- Now let us check how we can skip actions and go back in time with the help of Redux dev tool.
- Following screenshots explain about the actions we have dispatched earlier to get the listing of items.
- Here we can see the actions dispatched in the inspector tab.
- On the right, you can see the Demo tab which shows you the difference in the state tree.

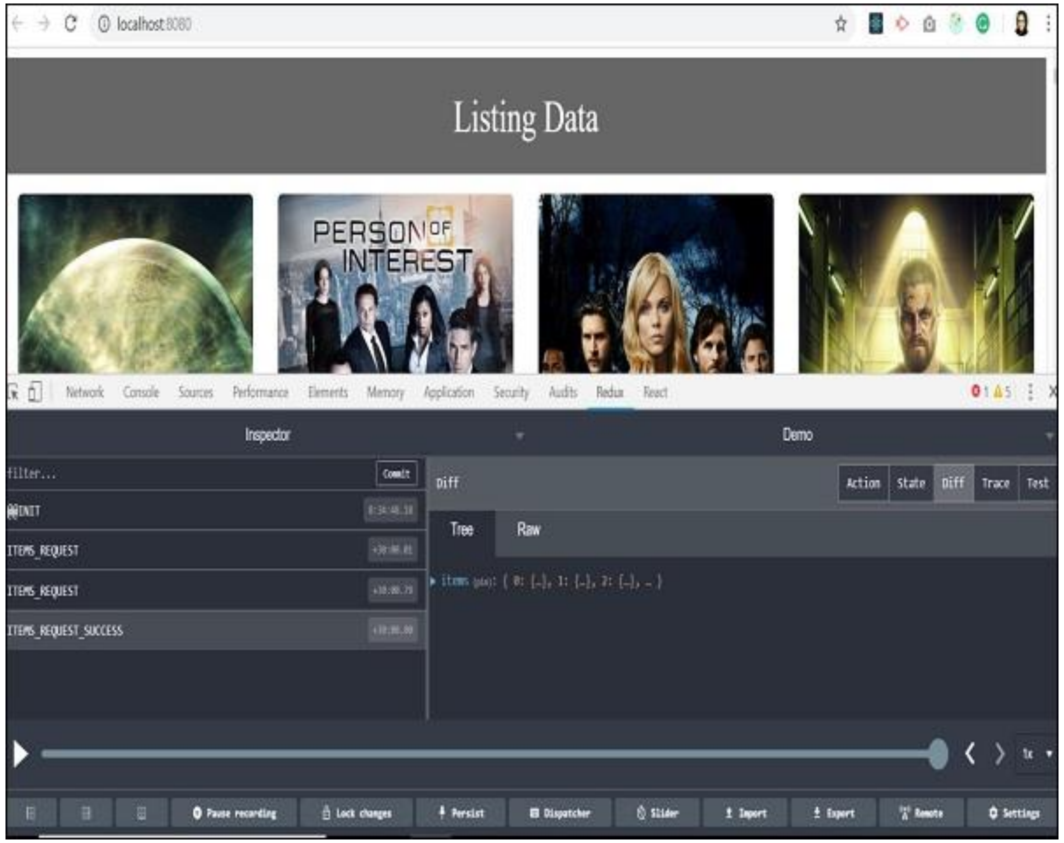


```
@@INIT
ITEMS_REQUEST
ITEMS_REQUEST
ITEMS_REQUEST_SUCCESS
```

```
isLoading (pin): false => true
```

Redux-Devtools

- You will get familiar with this tool when you start using it.
- You can dispatch an action without writing the actual code just from this Redux plugin tool.
- A Dispatcher option in the last row will help you with this.
- Let us check the last action where items are fetched successfully.

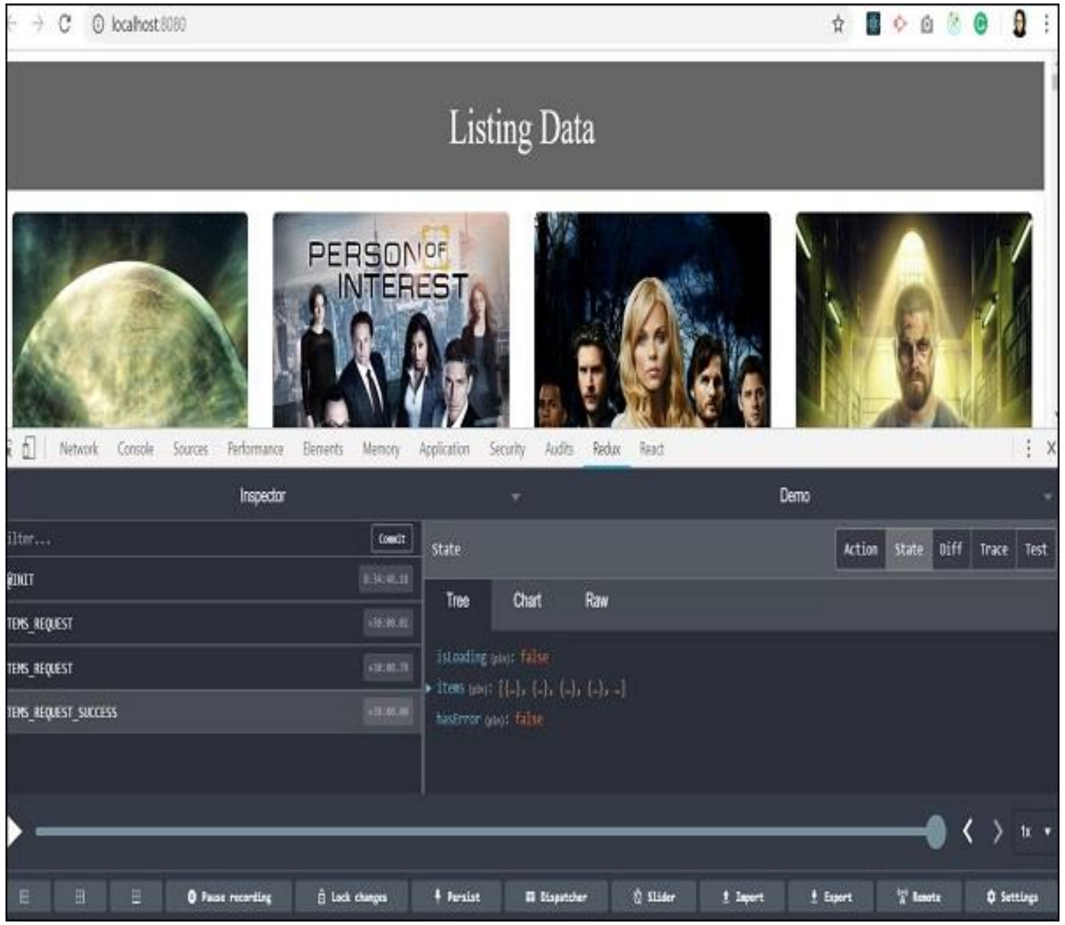


```
@@INIT
ITEMS_REQUEST
ITEMS_REQUEST
ITEMS_REQUEST_SUCCESS
```

```
▸ items (pin): { 0: {…}, 1: {…}, 2: {…}, … }
```

Redux-Devtools

- We received an array of objects as a response from the server.
- All the data is available to display listing on our page.
- You can also track the store's state at the same time by clicking on the state tab on the upper right side.



```
@INIT
TEMS_REQUEST
TEMS_REQUEST
TEMS_REQUEST_SUCCESS
```

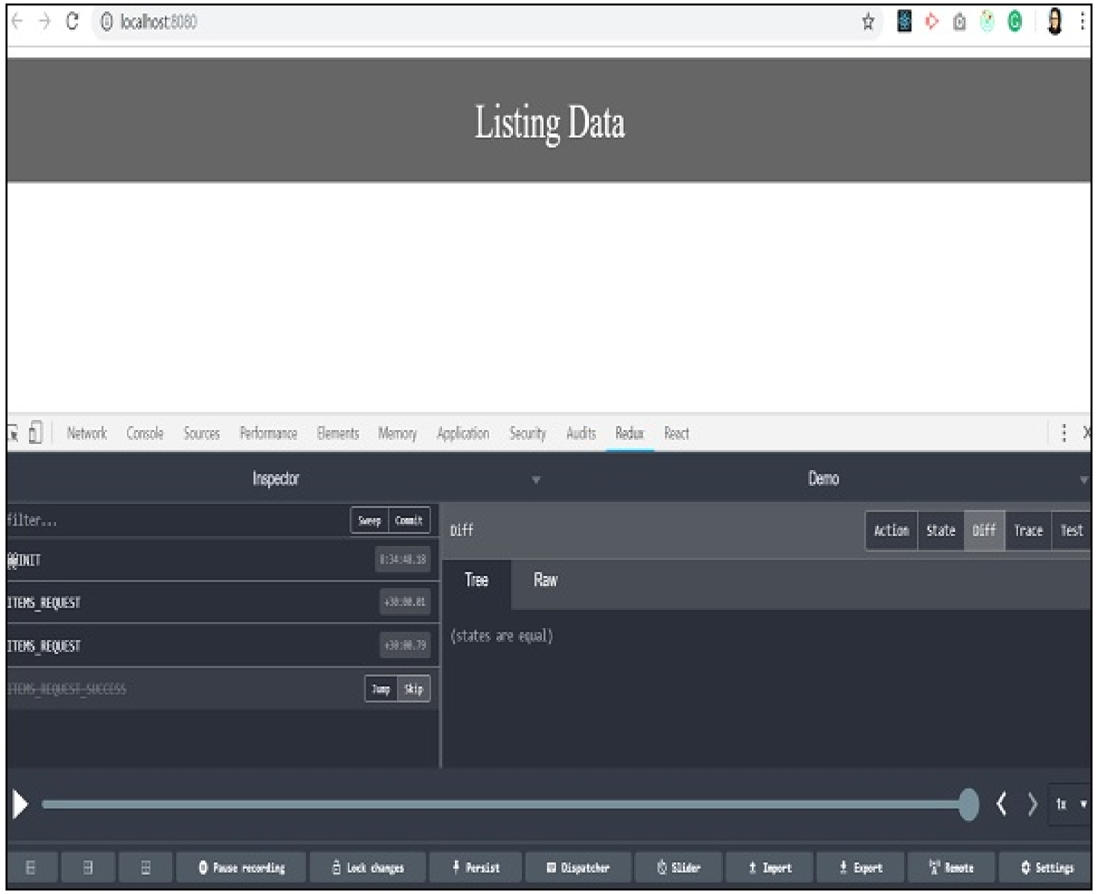
```
isLoading (pin): false
▸ items (pin): [{…}, {…}, {…}, {…}, …]
hasError (pin): false
```

Redux-Devtools

- Let us now check how to skip one action and go back in time to analyze the state of our app.
- As you click on any action type, two options: ‘Jump’ and ‘Skip’ will appear.
- By clicking on the skip button on a certain action type, you can skip particular action.
- It acts as if the action never happened.
- When you click on jump button on certain action type, it will take you to the state when that action occurred and skip all the remaining actions in sequence.
- This way you will be able to retain the state when a particular action happened.
- This feature is useful in debugging and finding errors in the application.

Redux-Devtools

- We skipped the last action, and all the listing data from background got vanished.
- It takes back to the time when data of the items has not arrived, and our app has no data to render on the page.
- It actually makes coding easy and debugging easier.



```
@@INIT
ITEMS_REQUEST
ITEMS_REQUEST
ITEMS_REQUEST_SUCCESS
```

(states are equal)

Redux Thunk

Redux Thunk

- By default, Redux’s actions are dispatched synchronously, which is a problem for any non-trivial app that needs to communicate with an external API or perform side effects.
- Redux also allows for middleware that sits between an action being dispatched and the action reaching the reducers.
- Thunk is a programming concept where a function is used to delay the evaluation/calculation of an operation.
- Redux Thunk is a middleware that lets you call action creators that return a function instead of an action object.
- This allows for delayed actions, including working with promises.
- That function receives the store’s dispatch method, which is then used to dispatch regular synchronous actions inside the function’s body once the asynchronous operations have been completed.

Installation and Setup

- Redux Thunk can be installed by running following command:

```
npm install redux-thunk--save
```

- Because it is a Redux tool, you will also need to have Redux set up.
- Once installed, it is enabled using **applyMiddleware():**

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

const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);
```

How to Use Redux Thunk

- Once **Redux Thunk** has been installed and included in your project with **applyMiddleware(thunk),** you can start dispatching actions asynchronously.
- For example, here's a **simple increment counter**:

```
const INCREMENT_COUNTER = 'INCREMENT_COUNTER';
function increment() {
  return {
    type: INCREMENT_COUNTER
  };
}

function incrementAsync() {
  return dispatch => {
    setTimeout(() => {
      // You can invoke sync or async actions with `dispatch`
      dispatch(increment());
    }, 1000);
  };
}
```

How to Use Redux Thunk

- And here's how set up success and failure actions after polling an API:

```
const GET_CURRENT_USER = 'GET_CURRENT_USER';
const GET_CURRENT_USER_SUCCESS =
'GET_CURRENT_USER_SUCCESS';
const GET_CURRENT_USER_FAILURE = 'GET_CURRENT_USER_FAILURE';

const getUser = () => {
  return (dispatch) => {     //nameless functions
    // Initial action dispatched
    dispatch({ type: GET_CURRENT_USER });
    // Return promise with success and failure actions
    return axios.get('/api/auth/user').then(
      user => dispatch({ type: GET_CURRENT_USER_SUCCESS, user }),
      err => dispatch({ type: GET_CURRENT_USER_FAILURE, err })
    );
  };
};
```

Redux and Async Data

Redux and Async Data

- React Redux apps are common on the web today due to their versatility, potential to scale, and innumerable features.
- Using actions and reducers complements your app's architecture and allows you to keep your code clean while implementing complex features.
- However, when you need to use asynchronous operations, such as dispatching actions after receiving the response from a server, actions and reducers alone aren't sufficient.
- To interact with asynchronous data in our React Redux app, we use a Redux library called redux-thunk.

React Redux Application Example

React Redux Application Example: Introduction

- For the demonstration purpose, we are going to develop a basic application with two main components involved:

  1. **Counter Component :** It will have a counter and three buttons viz. Increment button, Decrement button and a Reset button which will control the counter value.
  2. **UsersData Component** : It will request a list of some user's data from a dummy API endpoint (JSON placeholder) and will show on the page when the page is opened in the browser.

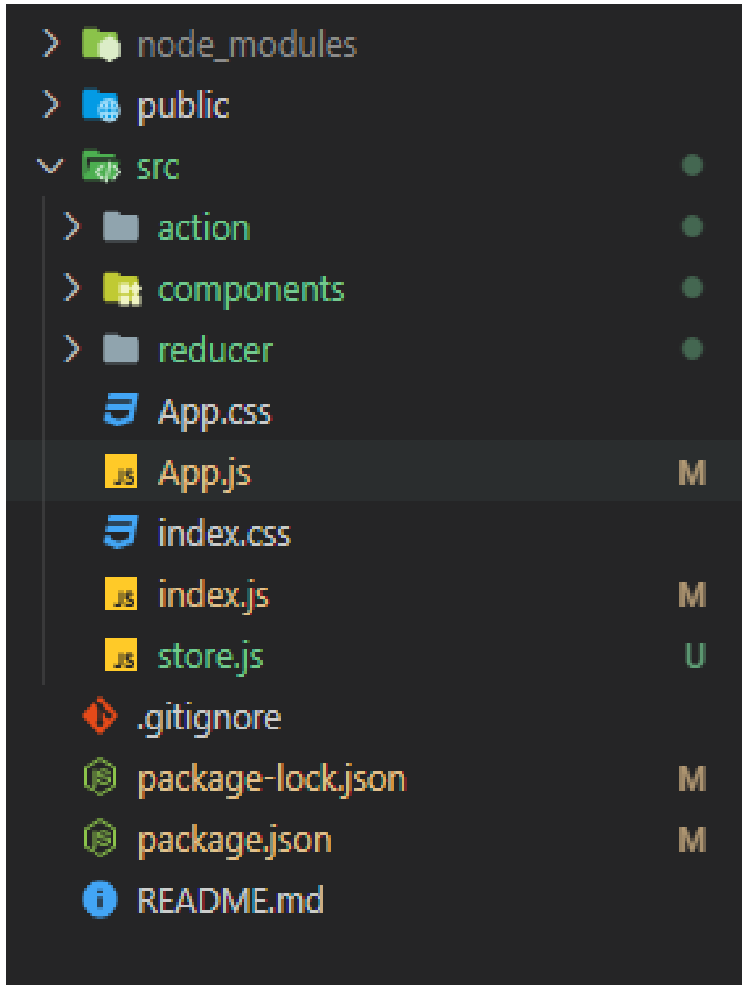
React Redux Application Example: Installation

- In order to create this project we need to install certain dependencies in our react app.
- Following are the commands we need to run in our terminal to create the redux based react project and run it.

```
1. npx create-react-app sample_app
2. cd sample_app
3. npm i axios redux react-redux redux-devtools-extension redux-thunk
4. npm start
```

React Redux Application Example: Folder Structure

- In redux-based react apps, we have **two** important folders viz. **action and reducer** and also a **store.js** file in our **src folder**.
- Action folder contains all the action files of the various components involved in our app and also a **types.js** file where all the action types are defined and exported.
- Reducer folder contains all the **reducer files** of the various app components and an **index.js** file where all the reducers are combined.
- **store.js** is where our **redux store** is created.



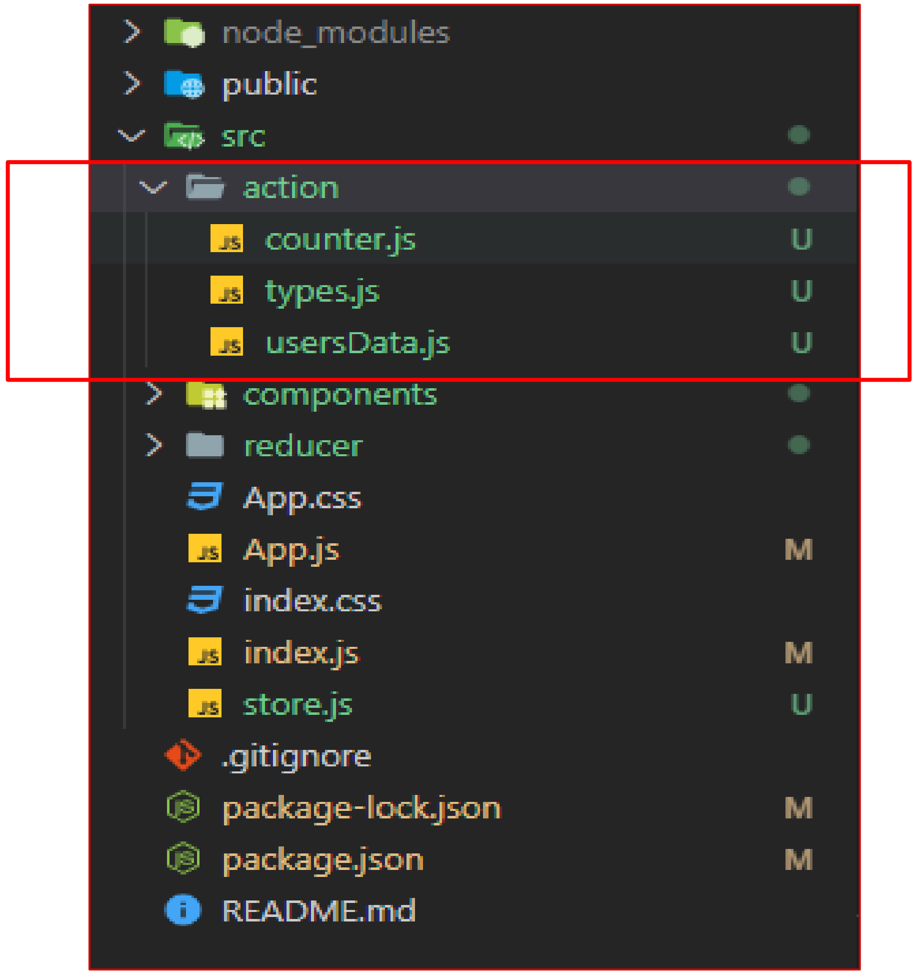
```
> node_modules
> public
v src                     ●
  > action                ●
  > components            ●
  > reducer               ●
    App.css
    App.js                M
    index.css
    index.js              M
    store.js              U
  .gitignore
  package-lock.json       M
  package.json            M
  README.md
```

React Redux Application Example: Action Folder

In our app's action folder, we two types of **.js** files which are :

1. **types.js :** This is where all the action types are defined in a string variable and exported so that there is no typo error while calling actions or defining their reducers.
2. **action files :** These are the .js files of various app components where the actions are actually defined.

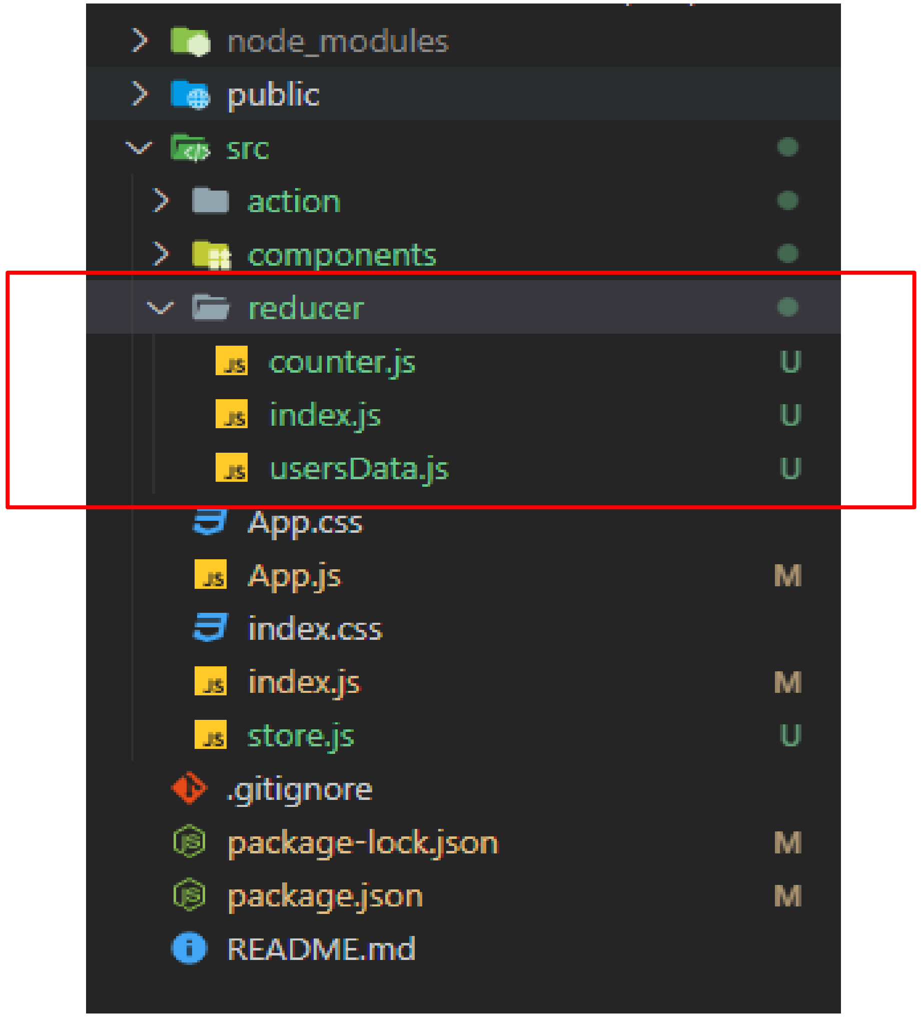
- When the action is called based on action types, the data is sent to its reducer as a payload.
- In case of server/ API requests, this is where we make those server/ API requests and send the response received from those endpoints as payload to its reducer.



React Redux Application Example: Reducer Folder

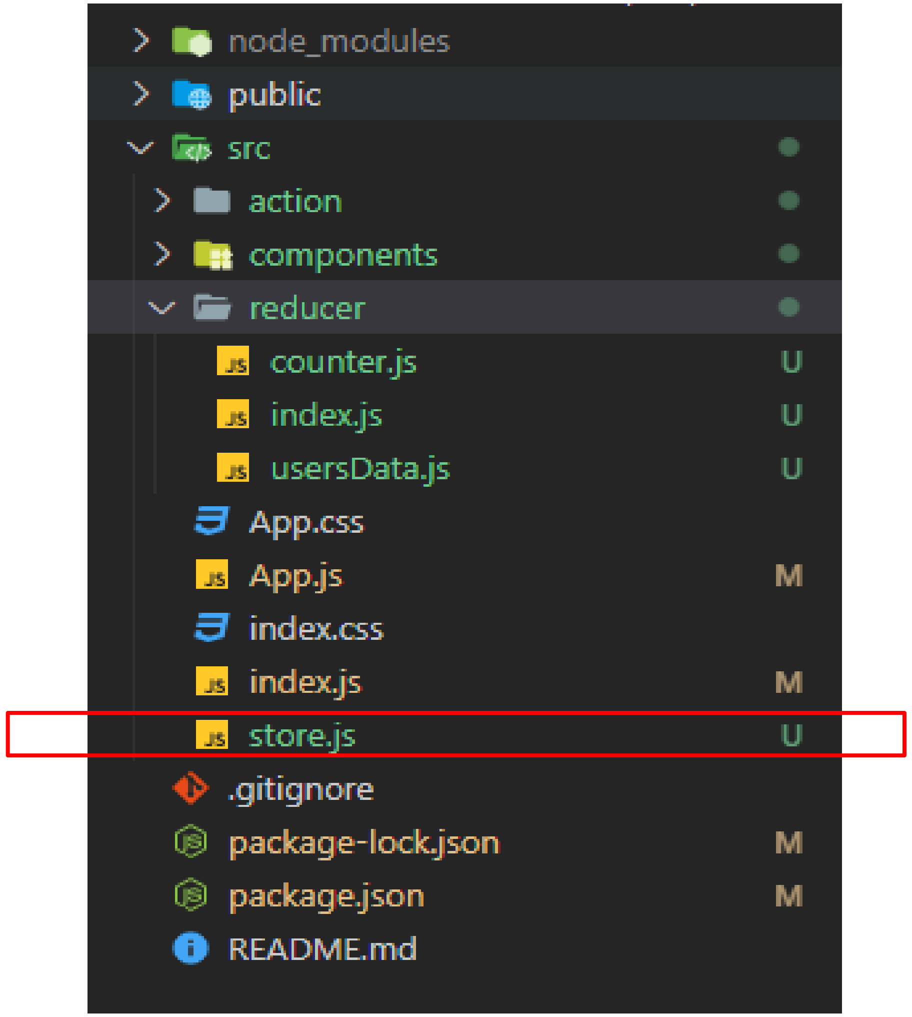
- In our app’s reducer folder, we again have **two** types of **.js** files:

1. **index.js :** This is where all the reducers files related to our actions are combined in a single entity and is then exported.
2. **reducer files :** These are the .js files where each reducer is defined, states are initialized and states are updated based on various action types invoked.



React Redux Application Example: store.js

- Redux provides us a single store for our app.
- This store contains all the app states controlled by redux.
- This **store.js** file is kept in **src** folder of our react app as shown here.



React Redux Application Example: store.js setup

- In our sample app, the first thing we create is our store in store.js file using createStore.
- We pass our root reducer i.e. reducer/index.js file, an empty initialState object and Redux DevTools(so that it can access our app states) with needed middlewares such as redux-thunk as shown here.
- Then we export this store so that we can access it in our app files.

```
import { createStore, applyMiddleware } from 'redux'
import { composeWithDevTools } from 'redux-devtools-
extension'
import thunk from 'redux-thunk'
import rootReducer from './reducer'

const initialState = {}
const middleware = [thunk]

const store = createStore(
  rootReducer,
  initialState,

composeWithDevTools(applyMiddleware(...middleware)
)
)

export default store
```

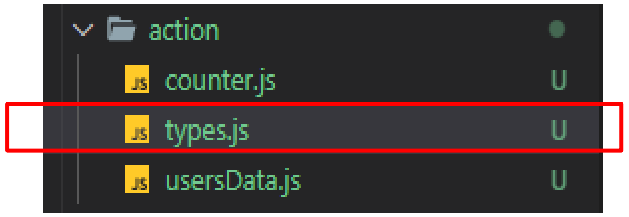
React Redux Application Example: store.js setup

- Now that we have created our store, the next step is to provide this store to our app.
- The **<Provider>** component makes the Redux store available to any nested components that need to access the Redux store.
- Since any React component in a **React Redux app** can be connected to the store, most applications will render a **<Provider>** at the top level, with the entire app's component tree inside of it.

```
import './App.css'
import { Provider } from 'react-redux'
import store from './store'

function App() {
  return (
    <Provider store={store}>
      <iv className="App">
        Hello World
      </div>
    </Provider>
  )
}

export default App
```

React Redux Application Example: type.js

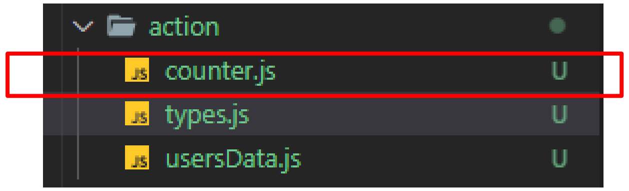
- All the action types are defined here as string variable and exported from here so that we can access them from our action and reducer files to avoid any possible **typo error** as these are the string variables and difficult to debug if an error occurs.
- It is kept inside our action folder as types.js.
- In our sample app, we have in total **five** different action types for **two** different components as shown here



```
// all types of action names exported

// action types for loading users for UsersData Component
export const LOAD_USERS_DETAILS_SUCCESS =
'LOAD_USERS_DETAILS_SUCCESS'
export const LOAD_USERS_DETAILS_FAILURE =
'LOAD_USERS_DETAILS_FAILURE'

// action types for controlling the Counter Component
export const INCREMENT_COUNTER =
'INCREMENT_COUNTER'
export const DECREMENT_COUNTER =
'DECREMENT_COUNTER'
export const RESET_COUNTER = 'RESET_COUNTER'
```

React Redux Application Example: counterAction

- This is the action file for our counter component and has **three** different actions exported from here to control our counter viz. **incrementCounter, decrementCounter and resetCounter**.
- All these actions involve a **dispatch** function which needs as on object argument involving action type and payload.
- **dispatch** is a function of the **Redux store**. You call dispatch to dispatch an action.
- This is the only way to trigger a state change.

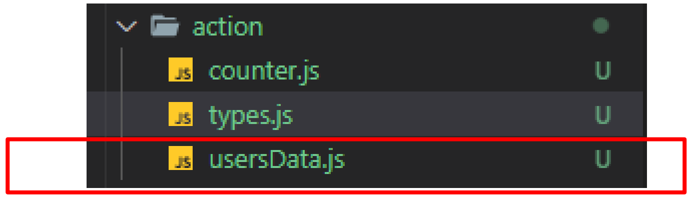


```
import { INCREMENT_COUNTER, DECREMENT_COUNTER,
RESET_COUNTER } from './types'

// action for showing loader
export const incrementCounter = () => (dispatch) => {
  dispatch({
    type: INCREMENT_COUNTER,
  })
}
export const decrementCounter = () => (dispatch) => {
  dispatch({
    type: DECREMENT_COUNTER,
  })
}
export const resetCounter = () => (dispatch) => {
  dispatch({
    type: RESET_COUNTER,
  })
}
```

React Redux Application Example: userData Action

- This is the action file for our **userData** component and has one action exported from here called **fetchUsersData.**
- Actions are the place where we make **server / API** requests.
- This **fetchUsesrData** action is here making an API request to the JSON placeholder.
- Based on the response, we have two different actions dispatched from here.
- These dispatch then bring the reducer into work by passing an action type and the response data as payload.

React Redux Application Example: userData Action

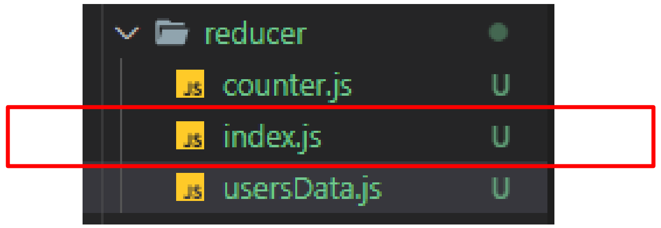


```
import axios from 'axios'
import { LOAD_USERS_DETAILS_SUCCESS,
LOAD_USERS_DETAILS_FAILURE } from './types'

// action for fetching users data from JSON placeholder endpoint
export const fetchUsersData = () => async (dispatch) => {
  try {
    const res = await
axios.get(`https://jsonplaceholder.typicode.com/users`)
    dispatch({
      type: LOAD_USERS_DETAILS_SUCCESS,
      payload: res.data,
    })
  } catch (err) {
    const errors = err
    dispatch({
      type: LOAD_USERS_DETAILS_FAILURE,
      payload: err,
    })
  }
}
```

React Redux Application Example: reducer/index.js

- As your app grows more complex, you'll want to split your reducing function into separate functions, each managing independent parts of the state.
- The **combineReducers** helper function turns an object whose values are different reducing functions into a single reducing function you can pass to **createStore.**
- The resulting reducer calls every **child reducer**, and gathers their results into a single state object.
- The state produced by **combineReducers()** namespaces the states of each reducer under their keys as passed to **combineReducers()**
- In our sample app, we have **two** reducers viz.
- **Counter and usersData** which we have combined using **combineReducers** function and exported as shown here

React Redux Application Example: reducer/index.js



```
import { combineReducers } from 'redux'

import counter from './counter'
import usersData from './usersData'

// combining all the reducers
export default combineReducers({
  counter,
  usersData,
})
```

Note:

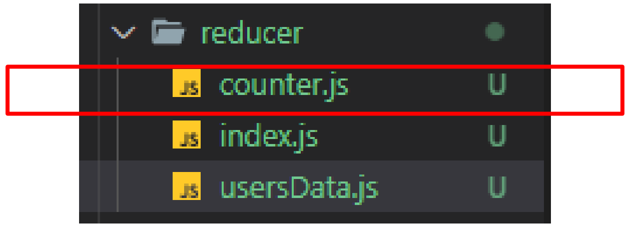
index.js file in any folder **'xyz'** can be imported as:

- import xyz from 'xyz/index'
- import xyz from 'xyz/index.js'
- import xyz from 'xyz'

React Redux Application Example: counter reducer

- In our sample app, the counter component will make use of **counter action** and **counter reducer**.
- This is where we have initialized our counter state with integer value **0.**
- This counter reducer function has **two** arguments that are **state object** and **action object.**
- This function will perform state changes based on our **three** action types of counter actions through switch case statement as shown here.
- If none of the action types **match,** it will return the same state by **default case.**
- The **updated state** is then returned from this function. This **reducer function** is exported from here so that it can be combined our reducer’s **index.js** file.

React Redux Application Example: counter reducer

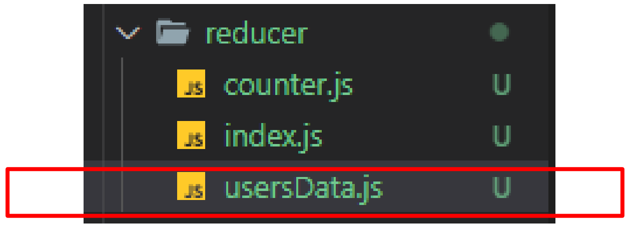


```
import {
  INCREMENT_COUNTER,
  DECREMENT_COUNTER,
  RESET_COUNTER,
} from '../action/types'
// initial states of counter
const initialState = { count: 0 }
// reducer for controlling the counter value
export default function (state = initialState, action) {
  const { type } = action
  let newCount
  switch (type) {
    case INCREMENT_COUNTER:
      newCount = state.count + 1
      return { ...state, count: newCount }
    case DECREMENT_COUNTER:
      newCount = state.count - 1
      return { ...state, count: newCount }
    case RESET_COUNTER:
      return { ...state, count: 0 }
    default:
      return state
  }
}
```

React Redux Application Example: userData reducer

- This is the usesrData reducer which handles the state of usersData component.
- We have initialized a users state as empty array.
- When usersData actions are called, this reducer will update users state based on the action type and the payload provided in its action argument.
- The payload is the data we received from our API endpoint requested in the action file.
- That’s how we update data from server / API to our app states.
- The updated state will then be returned by this reducer.
- This reducer is also exported so that it can be combined in the **reducer/index.js** file just like other reducers.

React Redux Application Example: userData reducer

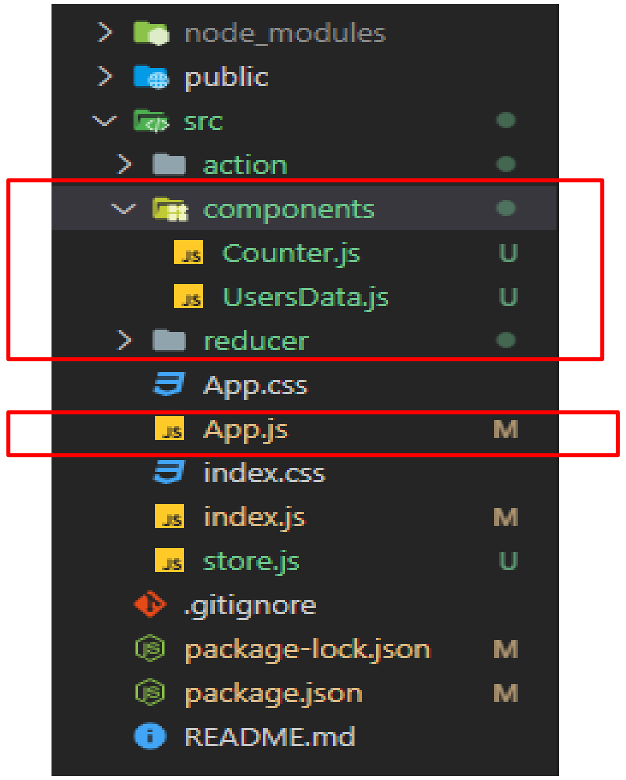


```
import {
  LOAD_USERS_DETAILS_SUCCESS,
  LOAD_USERS_DETAILS_FAILURE,
} from '../action/types'

// initial states of userData
const initialState = { users: [] }

// reducer for updating the userData
export default function (state = initialState, action) {
  const { type, payload } = action

  switch (type) {
    case LOAD_USERS_DETAILS_SUCCESS:
      return { ...state, users: payload }

    case LOAD_USERS_DETAILS_FAILURE:
      return { ...state, users: [] }

    default:
      return state
  }
}
```

React Redux Application Example: Components

- Our sample consists of two components which we have created inside the components folder which are :

1. Counter.js
2. UsersData.js



```
> node_modules
> public
v src
  > action
  v components
      Counter.js        U
      UsersData.js      U
  > reducer
    App.css
    App.js              M
    index.css
    index.js            M
    store.js            U
  .gitignore
  package-lock.json     M
  package.json          M
  README.md
```

React Redux Application Example: Components

- These components are then imported and are rendered inside our **App.js** as shown here.

```
import './App.css'
import { Provider } from 'react-redux'
import store from './store'
import Counter from './components/Counter'
import UsersData from './components/UsersData'

function App() {
  return (
    <Provider store={store}>
      <div className="App">
        <Counter />
        <br /> <hr /> <br />
        <UsersData />
      </div>
    </Provider>
  )
}

export default App
```

React Redux Application Example: Counter Component

- In our sample app, counter component contains display of **count value** and **three** buttons which control the value by **incrementing, decrementing or resetting** it.
- This count value is actually a state which we created in our **counter reducer** and in order to access this state and render it in our browser, we need to connect our component with the involved actions and our **state as props** to this component through connect function.
- As the first argument passed in to connect, **mapStateToProps** is used for selecting the part of the data from the store that the connected component needs.
- It is called every time the **store state changes**.
- It receives the entire store state, and should return an object of data this component needs.

React Redux Application Example: Counter Component

- In our counter component, we extracted counter state from our store and provided it to our counter component as props along with our three action functions imported from our counter action.
- These actions are being dispatched through **store.dispatch()** function on button clicks to control our counter.
- When these actions are dispatched here from our component, the actions then instruct our counter reducer to update the counter state based on the type of action invoked.
- Once this state is updated, our component is **re-rendered** in our browser thereby updating the counter value and making this counter component functional.

React Redux Application Example : Counter Component

```
import React from 'react'
import { connect } from 'react-redux'
import {
  incrementCounter, decrementCounter,resetCounter,
} from '../action/counter'
import store from '../store'
const Counter = ({ count }) => {
  return (
    <>
      <h1>Counter Component</h1>
      <h3>Count : {count}</h3>
      <br />
      <button
              onClick ={() => store.dispatch(incrementCounter())}
         > + </button>
      <button
              onClick ={() => store.dispatch(decrementCounter())}
         > - </button>
      <button
              onClick ={() => store.dispatch(resetCounter())}
         > Reset </button>
    </>
  )
}
```

React Redux Application Example : Counter Component

- Complete Counter component code as discussed is shown here.

```
// mapping states to props
// to pass it to the component
const mapStateToProps = (state) => {
  return {
    count: state.counter.count,
  }
}

// connecting mapStateToProps
// and action to the component
export default connect(mapStateToProps, {
  incrementCounter,
  decrementCounter,
  resetCounter,
})(Counter)
```

React Redux Application Example: User Data Component

- In our sample app, **UsersData** component shows some users data on the browser by requesting those data from an **API endpoint**.
- This users data is actually a state which we created in our **usersData** reducer and in order to access this state and render it in our browser, we need to connect our component with the involved action i.e. **fetchUsersData()** and our users state as props to this component through connect function.
- This users state is actually an empty array which will be populated with the users data we receive from our API which we requested in our **usersData** action.

React Redux Application Example: User Data Component

- In order to invoke the **fetchUsersData** action, we called **store.dispatch()** function with **fetchUsersData()** as its argument in our **useEffect** hooks (specifically, useEffect with empty array as second argument) of this component as the code written inside this **useEffect's** first argument is executed when the component is mounted in our app.
- Hence, the **API request** is made exactly here on **component mounting** and then this **usersData** action instructs **usersData** reducer to update the users state of this component.
- Once this users state is updated after the **API sends a response**, our **UsersData** component is re-rendered and shows the users data in our browser thereby making our **UsersData component functional.**

React Redux Application Example : User Data Component

```
 import React, { useEffect } from 'react'
import { connect } from 'react-redux'
import { fetchUsersData } from '../action/usersData'
import store from '../store'
const UsersData = ({ users }) => {
  useEffect(() => { store.dispatch(fetchUsersData())}, [])
  return (
    <div style={{ textAlign: 'center' }}>
      <h1>Users Data Component</h1> <br />
      <table style={{ display: 'inline-block' }}>
        <tr> <th>Name</th> <th>Email</th> <th>Website</th> </tr>
        {users.map((user) => {
          return (
            <tr>
              <td>{user.name}</td>
              <td>{user.email}</td>
              <td>{user.website}</td>
            </tr>
          )
        })}
      </table>
    </div>
  )
}
```

React Redux Application Example : User Data Component

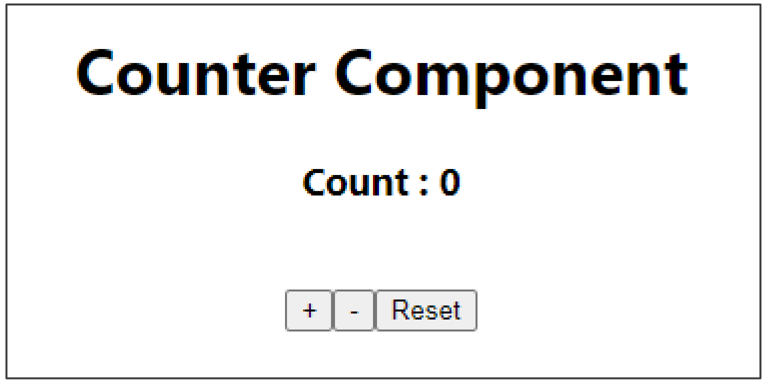
- Complete UsersData component code as discussed is shown here.

```
// mapping states to props to pass it to the component
const mapStateToProps = (state) => {
  return {
    users: state.usersData.users,
  }
}

// connecting mapStateToProps and action to the component
export default connect(mapStateToProps, {
  fetchUsersData,
})(UsersData)
```

React Redux Application Example : Counter Output

Initially rendered output

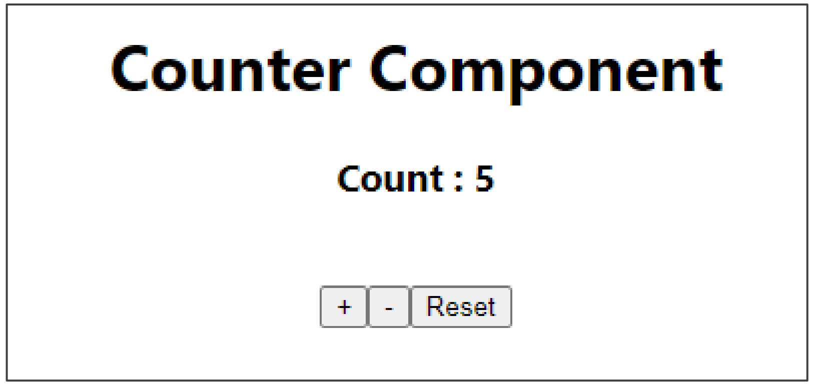
.



Counter Component

Count : 0

Value changed after some button clicks

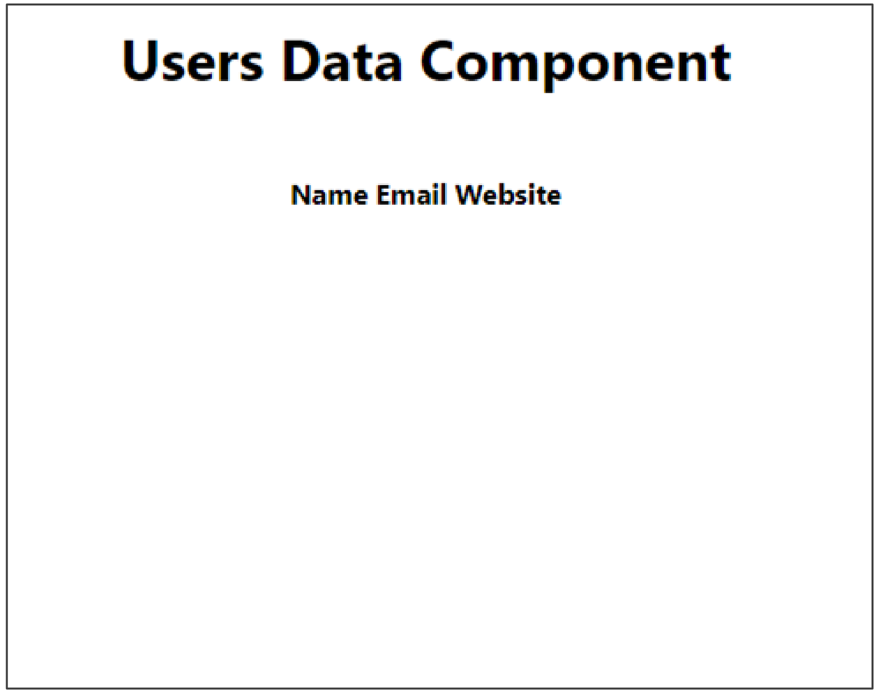


Counter Component

Count : 5

React Redux Application Example : User Data Output

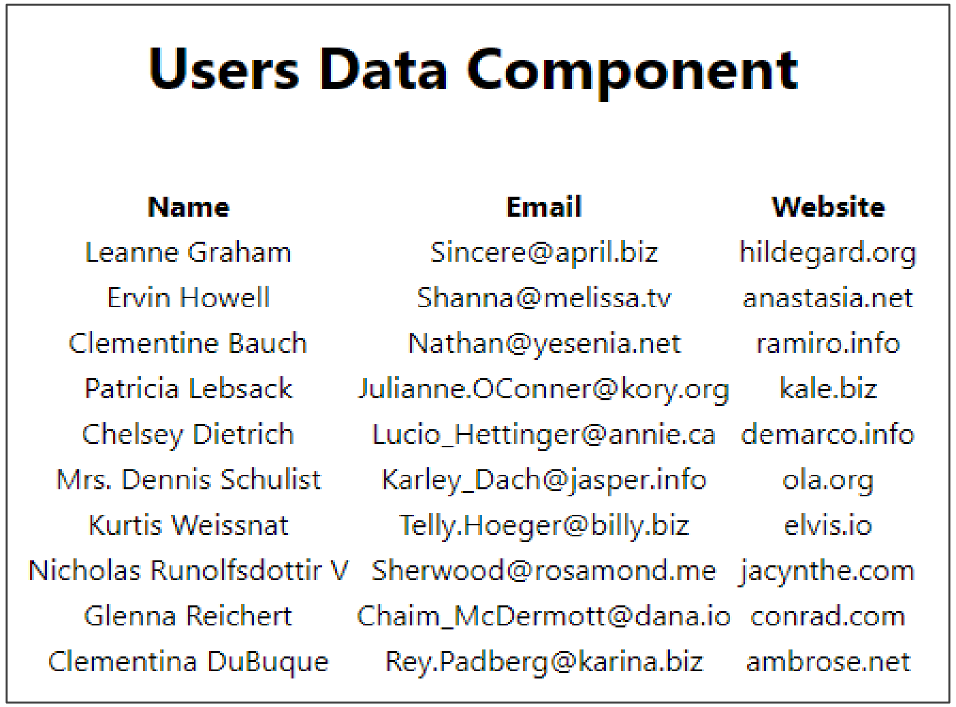
Initially rendered output



Users Data Component

Name Email Website

Re- rendered output after API response

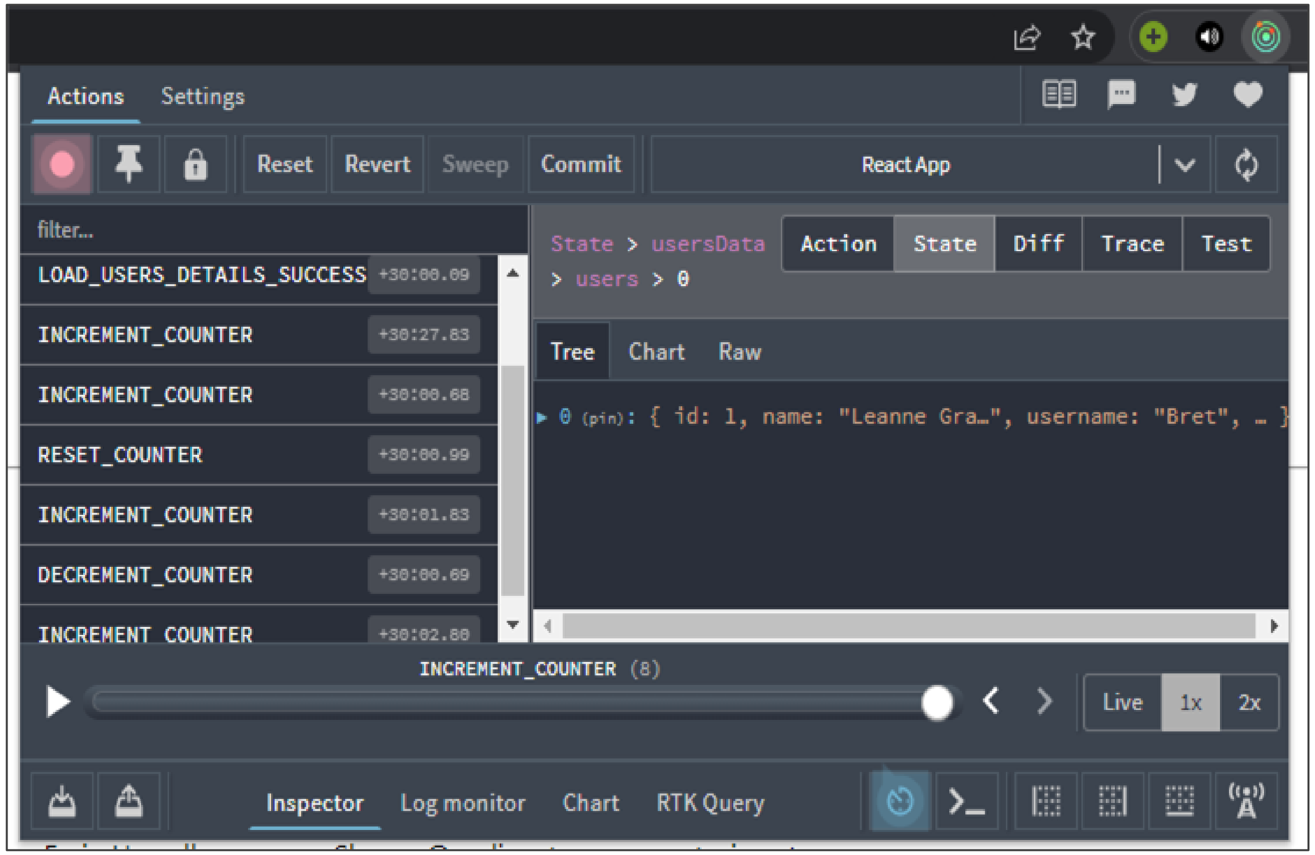


Users Data Component

Name	Email	Website
Leanne Graham	Sincere@april.biz	hildegard.org
Ervin Howell	Shanna@melissa.tv	anastasia.net
Clementine Bauch	Nathan@yesenia.net	ramiro.info
Patricia Lebsack	Julianne.OConner@kory.org	kale.biz
Chelsey Dietrich	Lucio_Hettinger@annie.ca	demarco.info
Mrs. Dennis Schulist	Karley_Dach@jasper.info	ola.org
Kurtis Weissnat	Telly.Hoeger@billy.biz	elvis.io
Nicholas Runolfsdottir V	Sherwood@rosamond.me	jacynthe.com
Glenna Reichert	Chaim_McDermott@dana.io	conrad.com
Clementina DuBuque	Rey.Padberg@karina.biz	ambrose.net

React Redux Application Example: Redux Dev Tools

We can see various actions being called in our apps and state changes through the Redux Devtools as shown here.

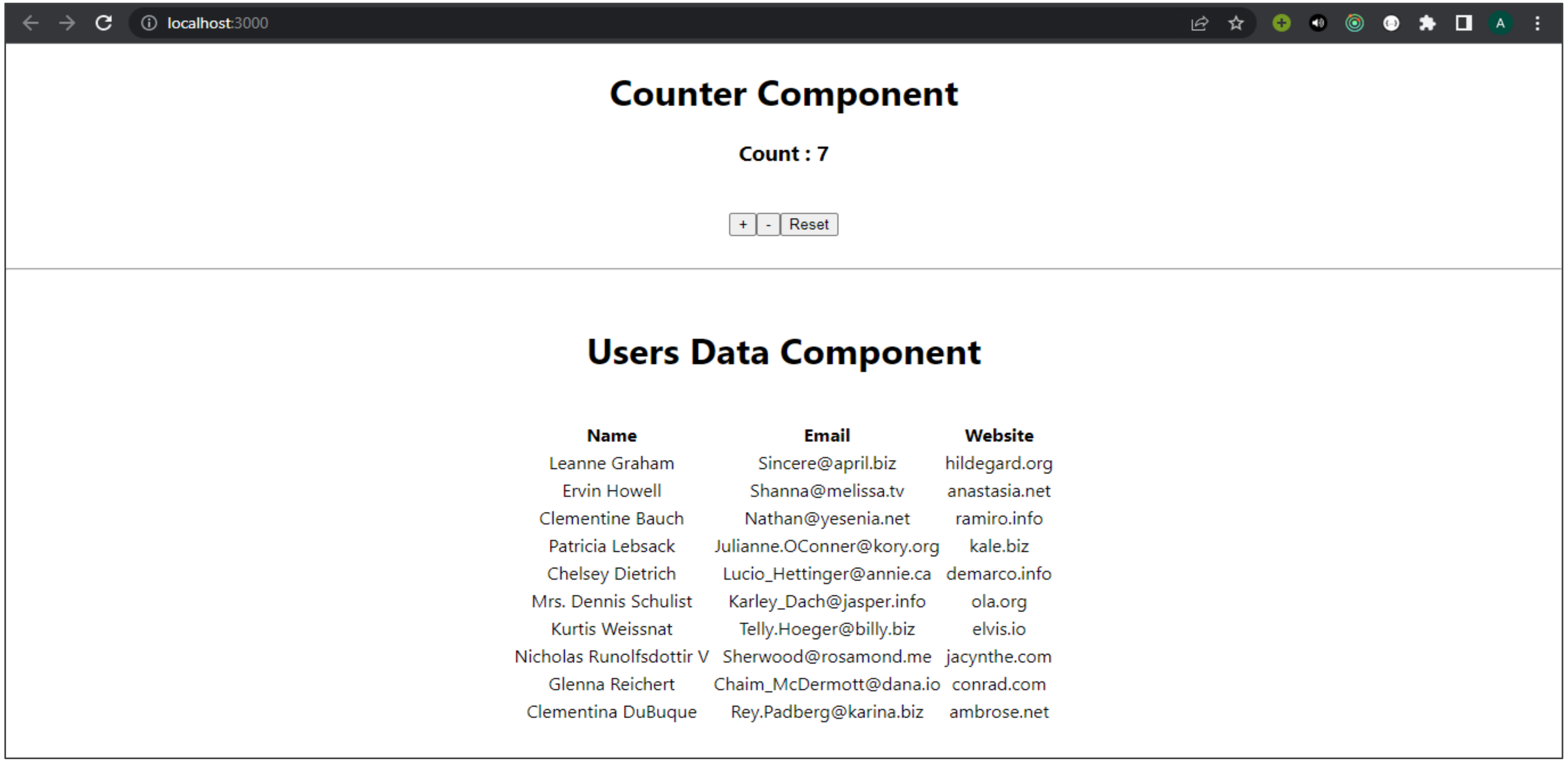


```
LOAD_USERS_DETAILS_SUCCESS  +30:00.09
INCREMENT_COUNTER           +30:27.83
INCREMENT_COUNTER           +30:00.68
RESET_COUNTER               +30:00.99
INCREMENT_COUNTER           +30:01.83
DECREMENT_COUNTER           +30:00.69
INCREMENT_COUNTER           +30:02.80
```

```
▶ 0 (pin): { id: 1, name: "Leanne Gra…", username: "Bret", … }
```

INCREMENT_COUNTER (8)

React Redux Application Example : Complete App Output



Counter Component

Count : 7

Users Data Component

Name	Email	Website
Leanne Graham	Sincere@april.biz	hildegard.org
Ervin Howell	Shanna@melissa.tv	anastasia.net
Clementine Bauch	Nathan@yesenia.net	ramiro.info
Patricia Lebsack	Julianne.OConner@kory.org	kale.biz
Chelsey Dietrich	Lucio_Hettinger@annie.ca	demarco.info
Mrs. Dennis Schulist	Karley_Dach@jasper.info	ola.org
Kurtis Weissnat	Telly.Hoeger@billy.biz	elvis.io
Nicholas Runolfsdottir V	Sherwood@rosamond.me	jacynthe.com
Glenna Reichert	Chaim_McDermott@dana.io	conrad.com
Clementina DuBuque	Rey.Padberg@karina.biz	ambrose.net

THANK YOU

SmartCliff
Career Mobility Solutions