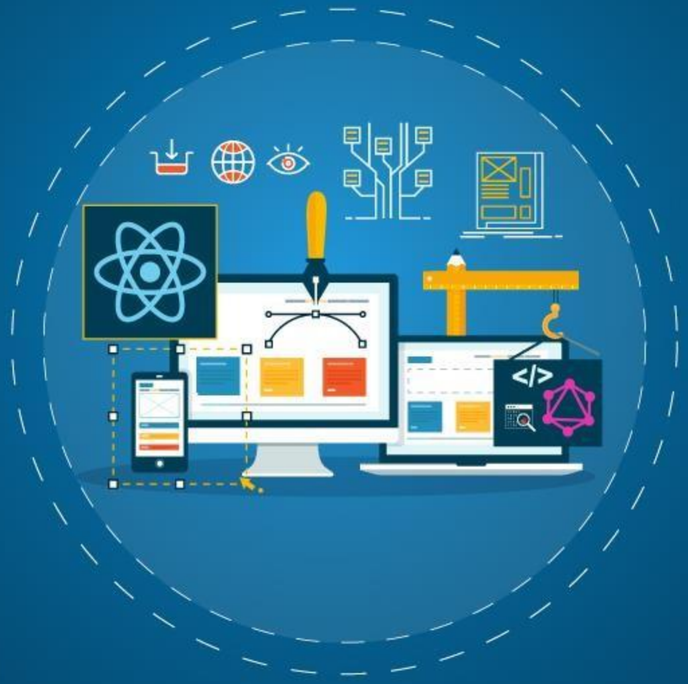
React With Redux Certification Training

COURSE OUTLINE
MODULE 05

1. Introduction to Web Development and React
2. Components and Styling the Application Layout
3. Handling Navigation with Routes
4. React State Management using Redux
5. ***Asynchronous Programming with Saga Middleware***



6. React Hooks
7. Fetching Data using GraphQL
8. React Application Testing and Deployment
9. Introduction to React Native
10. Building React Native Applications with APIs

Topics

Following are the topics covered in this module:

- Need of Async operations
- Async Workflow
- Action Creators
- How to write Action Creators?
- Handling Async Actions via Reducers
- Middleware
- Redux-Saga
- Generators in Redux-Saga
- Saga Methods()
- Major Sections of Redux-Saga
- Building a Product List application using Redux-Saga Middleware
- Redux Devtools

Objectives

After completion of this module you should be able to:

- Understand Async Workflow
- Define Action Creators in Redux applications
- Handle Async Actions using Reducers
- Understand role of Middleware in React-Redux applications
- Write sagas to Fetch data from an API
- Build an application using Redux-Saga middleware
- Debug your application using Redux DevTools

Need Of Async Operations

Flow of data in Redux is ***synchronous***, where the actions dispatched are received by the reducers which later via store, updates the final state in view

This is ***not a flexible*** approach in case of applications, that needs to ***communicate*** with ***external API*** or ***perform side effects***

Here, ***side effects*** means ***interactions*** with the world beyond your Redux application like *fetching data from a remote server, accessing local storage, recording analytics events and more*

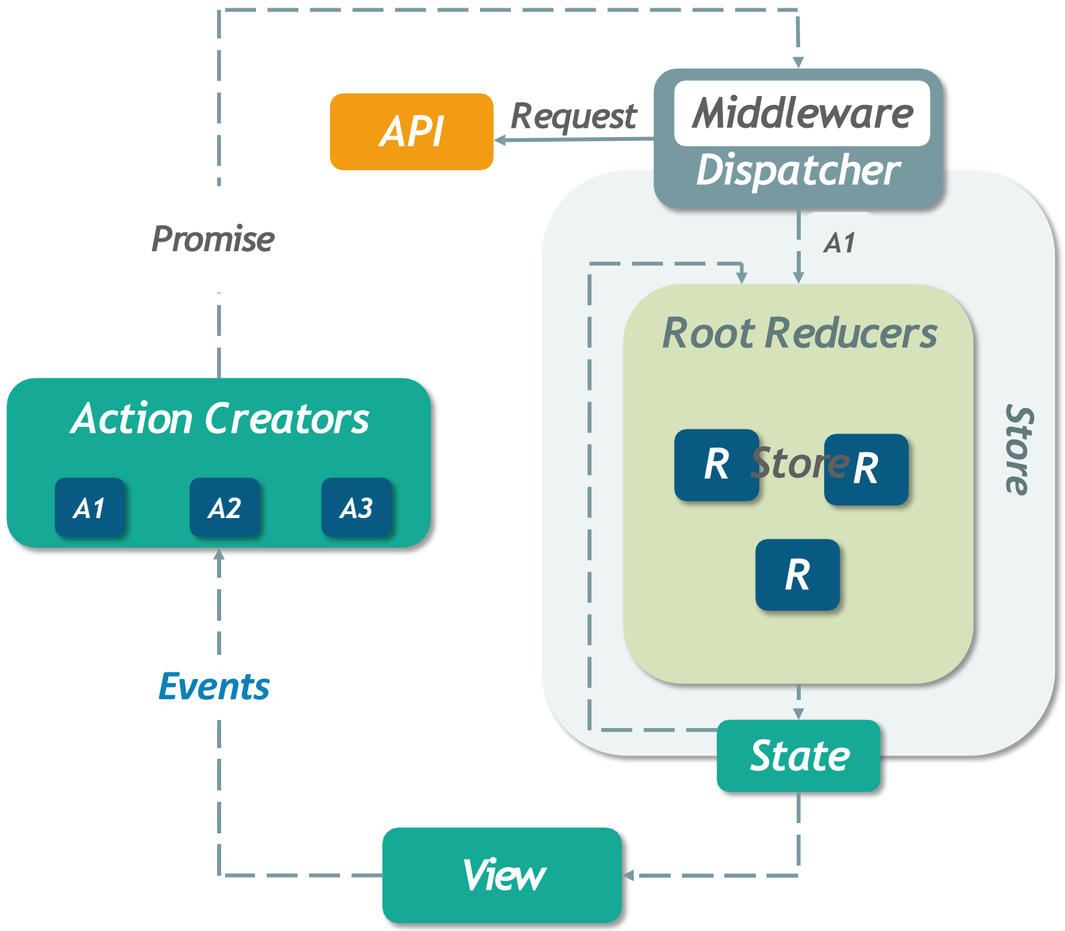
In order to implement the above events, we need to introduce ***asynchronous calls*** within the synchronous Redux data flow


Async Operations mainly make use of ***JavaScript Promises*** to manage the Async Actions

Async Workflow

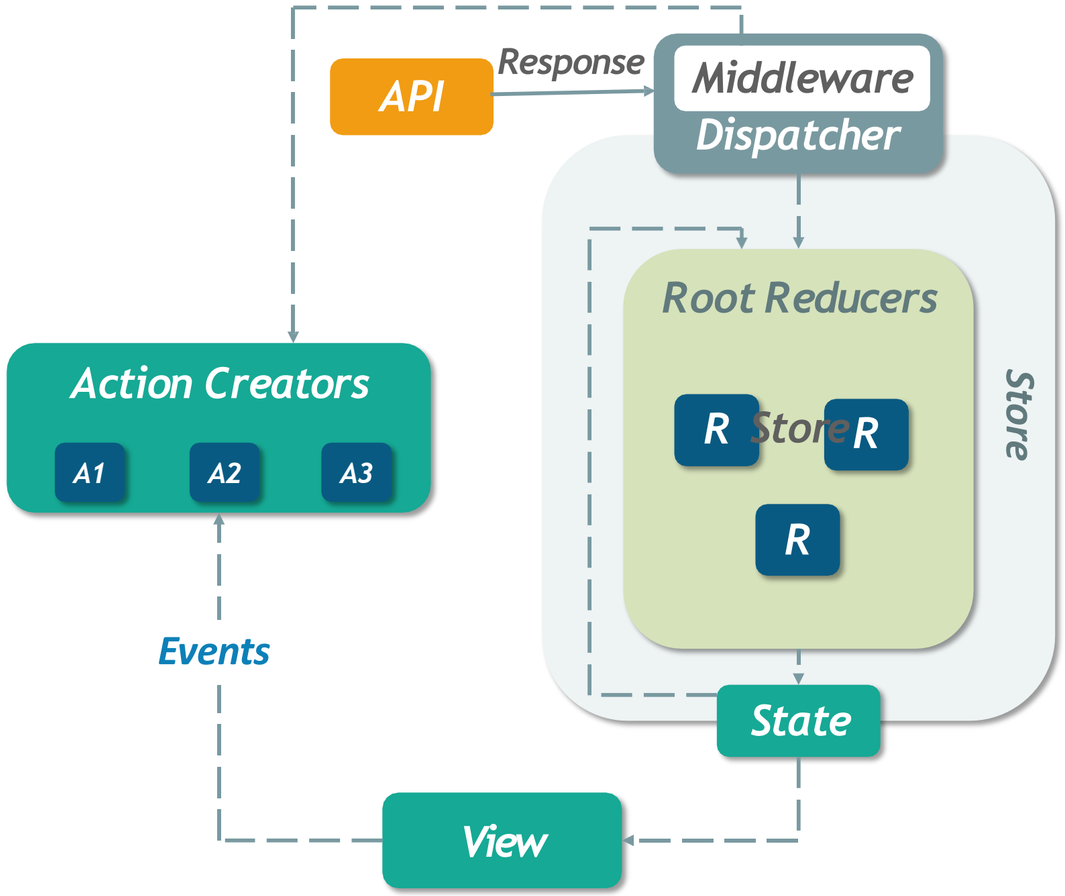
Async Workflow

- When user ***clicks*** on the button in an application, user activity is carried to the ***action creators*** via events
- Action creators dispatch a ***Promise*** which is executed by the ***middleware*** to ***fetch*** the data from an ***API***
- Meanwhile, until data from an API is received, a ***PENDING action*** is dispatched, to inform the UI to display ***spinner*** (indication of loading state)



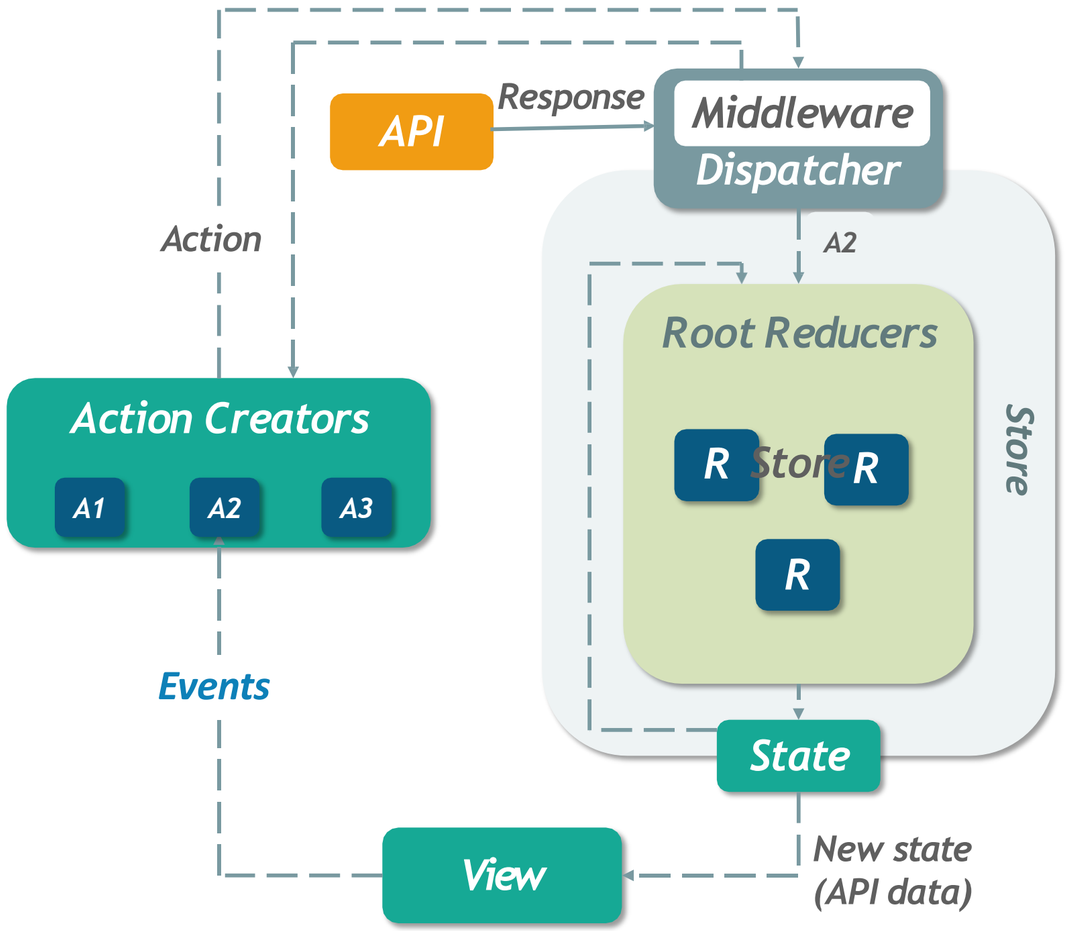
Async Workflow

- API sends ***payload*** as a response
- The moment we receive an API response, Action creator is ***triggered*** by the middleware to ***dispatch*** the ***next action***
- Hence, the ***spinner is hided*** in the view section



Async Workflow

- This action is received by the reducers
- Reducers after processing the actions pass the final state to the store
- Store later updates the view with the current state (contains API data)
- Hence, API data is displayed on the screen



In order to include asynchronous operations in Redux, we have to introduce two new constructs, an ***action creator*** and a ***middleware***

Action Creators

01

An action creator encapsulates the process of creating an ***action object***

Action creators holds the ***details*** of how an ***action is created*** and where can we implement the ***logic*** to establish ***communication*** with ***backend APIs***

02

03

bindActionCreators() method is used to ***bind*** multiple action creators with ***dispatch function***

You can install an Action Creator using: ***npm i action-creators***

04

How To Write Action Creators?

First Create a Store

```
const store = createStore(rootReducer)
```

Define an Action Creators

```
export function getUserDetailsRequest(id) {
  return {
    type: Actions.GET_USER_DETAILS_REQUEST,
    payload: id,
  };
}
```

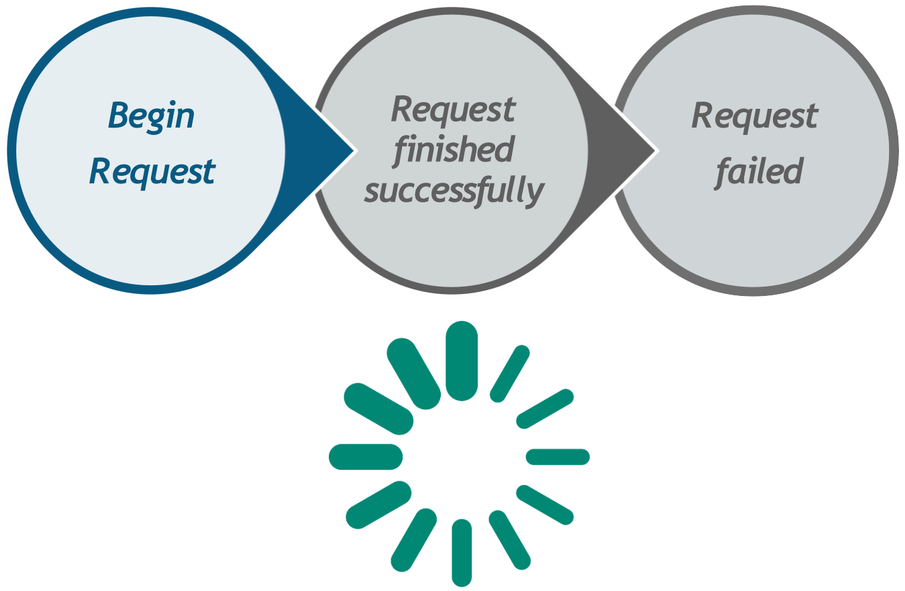
string that specify the ***type*** of an action

Bind all the action creators and dispatch them to Store

```
const bound = bindActionCreators(getUserDetailsRequest, store.dispatch)
```

How Are Async Actions Handled?

When an API is called the two important moments are, ***call time and the moment we get the response***. At each moment reducers ***update*** the state by dispatching following Async actions:



When a ***begin request action*** is dispatched, reducers handle it by toggling an ***isFetching*** flag in the state. Hence, the UI is triggered to show a spinner.

How Are Async Actions Handled?

When an API is called the two important moments are, ***call time and the moment we get the response***. At each moment reducers ***update*** the state by dispatching following Async actions:

When a ***Request finished successfully action*** is dispatched, reducers handle it this by merging new data into the state they manage and reset ***isFetching*** flag. Hence, UI ***hides the spinner*** and display the fetched data.

How Are Async Actions Handled?

When an API is called the two important moments are, ***call time and the moment we get the response***. At each moment reducers ***update*** the state by dispatching following Async actions:

When a ***request failed action*** is dispatched, reducers handle it by ***resetting isFetching*** flag. Hence, UI displays ***error message.***

Middleware

Middleware

Middleware is the software that connects network-based requests generated by a client to the back-end data the client is requesting.

1. In Redux, Middleware is a function that let us "tap into" what's happening inside Redux when we ***dispatch*** an ***action***
2. It sits between an ***action being dispatched*** and the ***store processing the action***
3. Middleware can inspect *actions and state, modify actions, dispatch other actions, stop actions from reaching the reducer*
4. To use middleware in Redux, we use the ***applyMiddleware() method*** from ***redux*** library
5. ***Redux Middleware use cases***: *logging, crash reporting, talking to an asynchronous API, routing, and more*

Redux-Saga

What Is Redux-Saga?

Redux-Saga middleware allows you to express complex application logic as functions called sagas

- It can be ***started, paused and cancelled*** from the main application using normal ***redux actions***
- It has ***access*** to the full redux application ***state*** and it can ***dispatch redux actions*** as well
- Sagas are implemented through special functions called ***generator***
- Redux-Saga is preferred for ***better control*** on actions being dispatched and ***better management*** of complex applications
- You can ***install*** Redux saga using ***npm install redux-saga***

Generator Function

Generators In Redux-Saga

A **generator** is a function that produces a sequence of results instead of a single value (executes multiple values in iterations).
Syntax of Generator function: ***function* name(param) {statements}***

Working of Generators:

- Here, First we ***invoke the function*** and ***store in a variable***
- The invocation of function returns an ***iterable object*** back to us
- Make a call to ***'next()'*** on the object ***to move to first yield point*** in function
- This call gives us an object with properties : ***value*** and ***done***
- We can continue iterating through this until we ***reach the end point***

Source: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*

Example Of Generator Function

```
function* generator() {
  yield 'react'
  return 'redux'
}

const gen = generator()

console.log(gen.next())
console.log(gen.next())
console.log(gen.next())

);
```

Generator function

yield expression is used to pause the function and give out a value

Method applied to get the next iterations

Output

```
{ value: 'react', done: false }
{ value: 'redux', done: true }
{ value: undefined, done: true }
```

Done indicates the execution of function, if we have still one more iteration the it returns false. If we have come to end of iterations then it returns true

Saga Methods()

The most commonly used Saga Methods are:

takeLatest()

If you dispatch the action before the previous API call finishes, takeLatest() will stop that call and return only the ***latest one.***

takeEvery()

Allows ***execution*** of multiple instances of sagas at the same time.

put(action)

Creates an ***Effect*** that instructs the middleware to schedule the dispatching of an action to the store.

Saga Methods()

The most commonly used Saga Methods are:

call(fn, ...args):

Creates an effect that instructs the middleware to call the function ***fn*** with ***args*** as arguments.

delay(ms, [val])

Returns an effect to ***block*** execution for ***ms milliseconds*** and return ***val value***.

all(effects):

Creates an effect that instructs the middleware to ***run multiple Effects*** in parallel and ***wait*** for all of them to ***complete.***

select()

gets data from the redux store.

How To Write Sagas?

Major Sections Of Redux Saga

Redux Sagas include following sections:

Watcher Saga listens to dispatched actions and triggers the Worker Saga.

Syntax of Watcher Saga

```
export default function* watchAction()
{
    yield takeLatest('TYPE_TO_LISTEN', workerSaga)
}
```

Action Type

Major Sections Of Redux Saga

Redux Sagas include following sections:

Worker Saga on receiving inputs from Watcher Saga gets the data from the API.

Syntax of Worker Saga

```
function* fetchData(action) {
  try {
     const data = yield fetch(API_URL)
     .then(response => response.data(), );

     yield put({type: "FETCH_SUCCEEDED", payload: data});
  }
catch (error) {
     yield put({type: "FETCH_FAILED", message: error});
  }}
```

Major Sections Of Redux Saga

Redux Sagas include following sections:

Watcher Saga → *Worker Saga* → ***Root Saga***

All the sagas should be written and registered with a ***root saga***, where sagas are processed one by one

Syntax of Root Saga

```
export default function* rootSaga() {
  yield all([
    WatcherSaga1(),
    WatcherSaga2(),
  ])
}
```

Integration Of redux-saga And Store

Like your reducers, all the sagas should be written and registered with a root saga

```
import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

import {rootSaga} from './sagas'

const sagaMiddleware = createSagaMiddleware()

const store = createStore(reducer, applyMiddleware(sagaMiddleware))

sagaMiddleware.run(rootSaga)

export default class App extends Component {
  render() {
    return (
      <Provider store={store}>
        <App/>
      </Provider>
    )
  }
```

Creates the Saga Middleware

Connects your middleware to store and reducers

Inform the middleware to start using sagas


Now lets build an application with all the topics learnt so far.

Demo: Building A Product List Application Using Redux-Saga Middleware

Demo: Installation Of Required Packages

Create the react application using the command ***create-react-app <application name>.***

```
PS C:\Users\[redacted]\Desktop\React\ReactJSDemo\Code\reactsaga> create-react-app productlist
```

Navigate to the application folder.

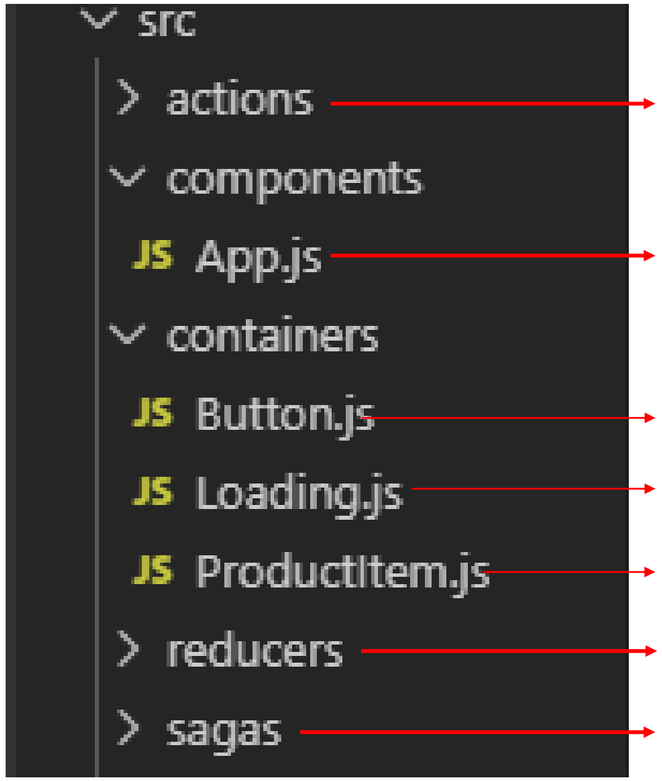
```
PS C:\Users\[redacted]\Desktop\React\ReactJSDemo\Code\reactsaga> cd productlist
```

Install redux-saga packages using the command ***npm i redux-saga.***

```
PS C:\Users\[redacted]\Desktop\React\ReactJSDemo\Code\reactsaga\productlist> npm i redux-saga
```

Demo: Folder Structure

Create the folder structure as mentioned below:



- actions → Carries user activities
- App.js → Binds all the three files of containers folder
- Button.js → Initial view of the application
- Loading.js → Code to display Spinner
- ProductItem.js → Code to display products list
- reducers → Manages states and returns final state to view
- sagas → Contains the bunch of sagas to manage the actions

Demo: Index.js File

Open Index.js file and add the paths of respective folders.

```
import React from 'react';
import createSagaMiddleware from 'redux-saga';
import { render } from 'react-dom';
import { createStore, applyMiddleware } from 'redux';
import { Provider } from 'react-redux';
import { logger } from 'redux-logger';
import reducer from './reducers';
import App from './components/App';
import rootSaga from './sagas';
```

- To create Redux middleware and connect saga to it
- Connects middleware and store
- Connects store to view section
- Inspects panel triggered actions and state of Redux store in console panel
- Common source to connect to all sagas

Demo: Integration Of Store

In the ***index.js*** file, integrate store, connect ***reducer, middleware and logger*** to the store.

```
const sagaMiddleware = createSagaMiddleware();

const store = createStore(
    reducer,
    applyMiddleware(sagaMiddleware, logger),
);
sagaMiddleware.run(rootSaga);
render(
    <Provider store={store}>
      <App />
    </Provider>,
document.getElementById('root'));
```

Demo: App.js

In the ***component*** folder, create ***App.js*** file and include the 3 major sections of application: ***Button, Spinner and Products list***

educers | JS ProductItem.js | JS Loading.js | JS App.js .../components

src > components > JS App.js > ...

```
1   import React from 'react';
2   import Button from '../containers/Button';
3   import ProductItem from '../containers/ProductItem'
4   import Loading from '../containers/Loading'
5   let App = () => (
6     <div>
7       <Button />
8       <Loading />
9       <br/>
10      <ProductItem />
11    </div>
12  );
13  export default App;
```

Initiates API call

Displays Spinner while getting data from API

After receiving data from API displays Products List

Demo: Button.js

In the containers folder, create Button.js file, write the code to make an API call when user press the button.

oductItem.js | JS Loading.js | JS App.js .../components | JS Button.js ×

src > containers > JS Button.js > ...

```
1   import React from 'react';
2   import { connect } from 'react-redux';
3   import { getProduct } from '../actions';
4   let Button=({getProduct})=>(
5       <div>
6       <br/>
7       <center>
8       <button onClick={getProduct}
9       className="btn btn-success">Press to see Products</button>
10      </center>
11      </div>
12
13  )
14  cons
15      getProduct: getProduct,
16  };
17  Button = connect(null,mapDispatchToProps)(Button);
18  export default Button;
```

```
(property) getProduct: () => {
    type: string;
}
```

Displays button on Screen, which initiates later actions

On click of button triggers getProduct action

Button component is connected to React-Redux using connect() function

Demo: Loading.js

In the containers folder, create Loading.js file to display spinner during async call to the API.

src > containers > JS Loading.js > ...

```
1   import React from 'react';
2   import { connect } from 'react-redux'
3   import img from '../loading.gif'
4   let Loading = ({ loading }) => (
5   loading ?
6   <div style={{ textAlign: 'center' }}>
7      <img src={img} alt='loading' />
8      <h1>LOADING</h1>
9   </div> :
10  null
11  );
12  const mapStateToProps = (state) => ({loading: state.loading})
13  Loading = connect(mapStateToProps,null)(Loading)
14  export default Loading;
```

Location of Spinner GIF

Displays Spinner while API Call

Maps the current state to props loading

Connects Loading component to react-redux using connect() function

Demo: Actions

Create an ***index.js*** file in ***actions*** folder and define the actions that are supposed to be triggered when user clicks the button.

ing.js | JS App.js .../components | JS Button.js | JS index.js .../actions

src > actions > JS index.js > getProduct

```
1  export const getProduct = () => ({
2      type: 'GET_PRODUCT',
3  });
```

Action Creator

Action type (used to call this action)

Demo: Reducer

Create an ***index.js*** file in ***reducer*** folder, to ***consume the actions*** and accordingly ***update the state*** to display elements in browser.

When action 'GET_PRODUCT' is dispatched, property of state *loading* becomes equal to **true** and the ***spinner*** is appears on the screen.

src > reducers > JS index.js > [⊗] reducer

```
1   const reducer = (state = {}, action) => {
2       switch (action.type) {
3         case 'GET_PRODUCT':
4             return { ...state, loading: true };
5
6         default:
7             return state;
8       }
9     };
10    export default reducer;
```

State after processing the action

Demo: sagas

Create an index.js file in sagas folder, define the required sagas to get the data from API using fetch method.

src > sagas > JS index.js > actionWatcher

```
1   import { put, takeLatest, all } from 'redux-saga/effects';
2   function* fetchProduct() {
3       const json = yield fetch('https://ngapi4.herokuapp.com/api/getProdu
4           .then(response => response.json(), );
5       yield put({ type: "PRODUCT_RECEIVED", json: json, });
6   }
7   function* actionWatcher() {
8       yield takeLatest('GET_PRODUCT', fetchProduct)
9   }
10  export default function* rootSaga() {
11      yield all([
12      actionWatcher(),
13      ]);
14  }
```

- Worker saga
- API call to receive data
- Next action to be dispatched after receiving data from API
- Watcher Saga listens to action and triggers the worker saga
- Collects all the sagas to be processed
- Makes call to the defined sagas one by one

Demo: reducers

In the reducers section write a pure function to consume the ***action PRODUCT_RECIEVED.***

Once response from the API call is received, Redux *state* will have property product which contains json data of products.

src > reducers > JS index.js > [⟐] reducer

```
1   onst reducer = (state = {}, action) => {
2     switch (action.type) {
3       case 'GET_PRODUCT':
4           return { ...state, loading: true };
5       case 'PRODUCT_RECEIVED':
6           return { ...state, product: action.json, loading: false }
7       default:
8           return state;
9     }
10   };
11   export default reducer;
```

Indicates no other action to be processed after this action

Final State

Demo: ProductItem.js

In the Container folder, create a ProductItem.js file, this component displays the received Products list in browser.

```
2   import { connect } from 'react-redux'
3
4   const articleStyle = {
5   width: '50%',
6   margin: '0 auto',
7   color: 'olive',
8   border: '4px solid RebeccaPurple ',
9   }
10  let ProductItem = ({ article }) => (
11
12  article ?
13  article.map((item) => {
14      return (
15          <article key={item.id} style={articleStyle}>
16              <div className="media">
17              <div className="media-left">
18                  <img src={item.imageUrl} className="media-object" />
19              </div>
20              <div className="media-body">
21                  <h4 className="media-heading">{item.productName}</h4>
22                  <p>{item.description}</p>
23              </div>
24          </div>
25
26          </article>
27      )
28  })
29   :
30  null
31  );
32
33  const mapStateToProps = (state) => (
34  {
35     article: state.product
36  })
37
38  ProductItem = connect(mapStateToProps,null)(ProductItem)
```

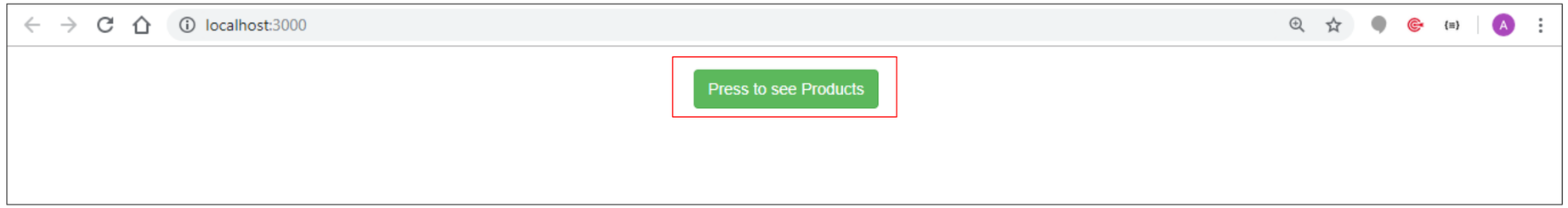
Data binding, to map the props articles with JSON data to display products list

Maps the state containing the productlist data to props articles

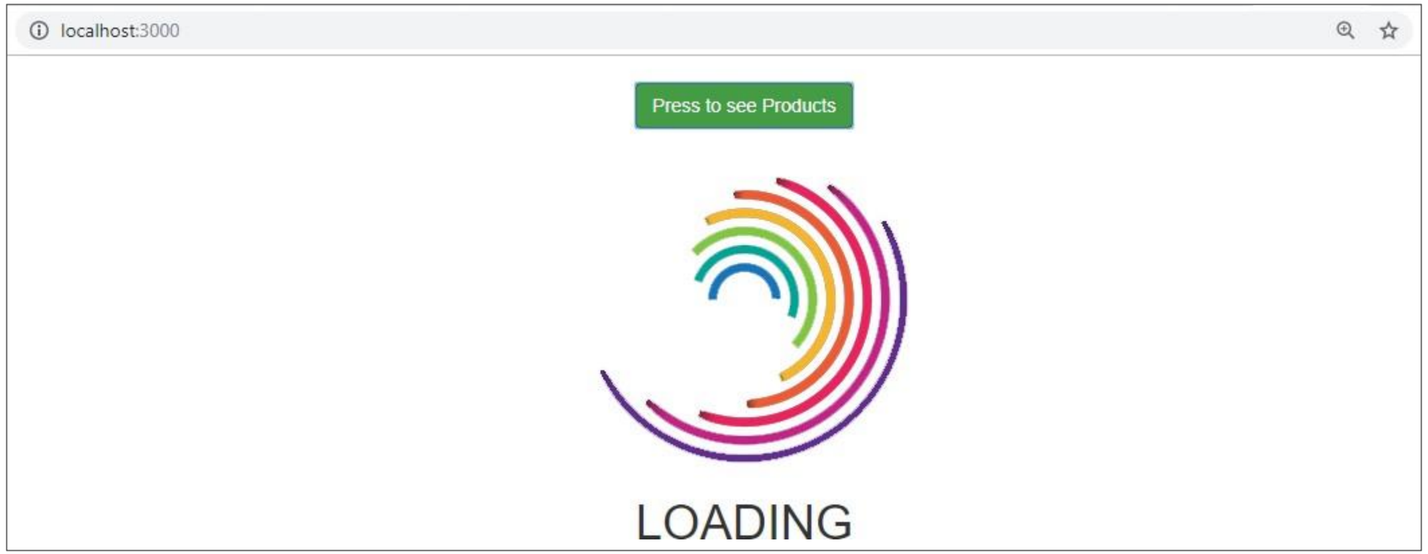
Connects ProductItem component to react-redux using connect() function

Demo: Product List Application (Output)

Execute your application using ***npm start,*** application displays the component as shown below:

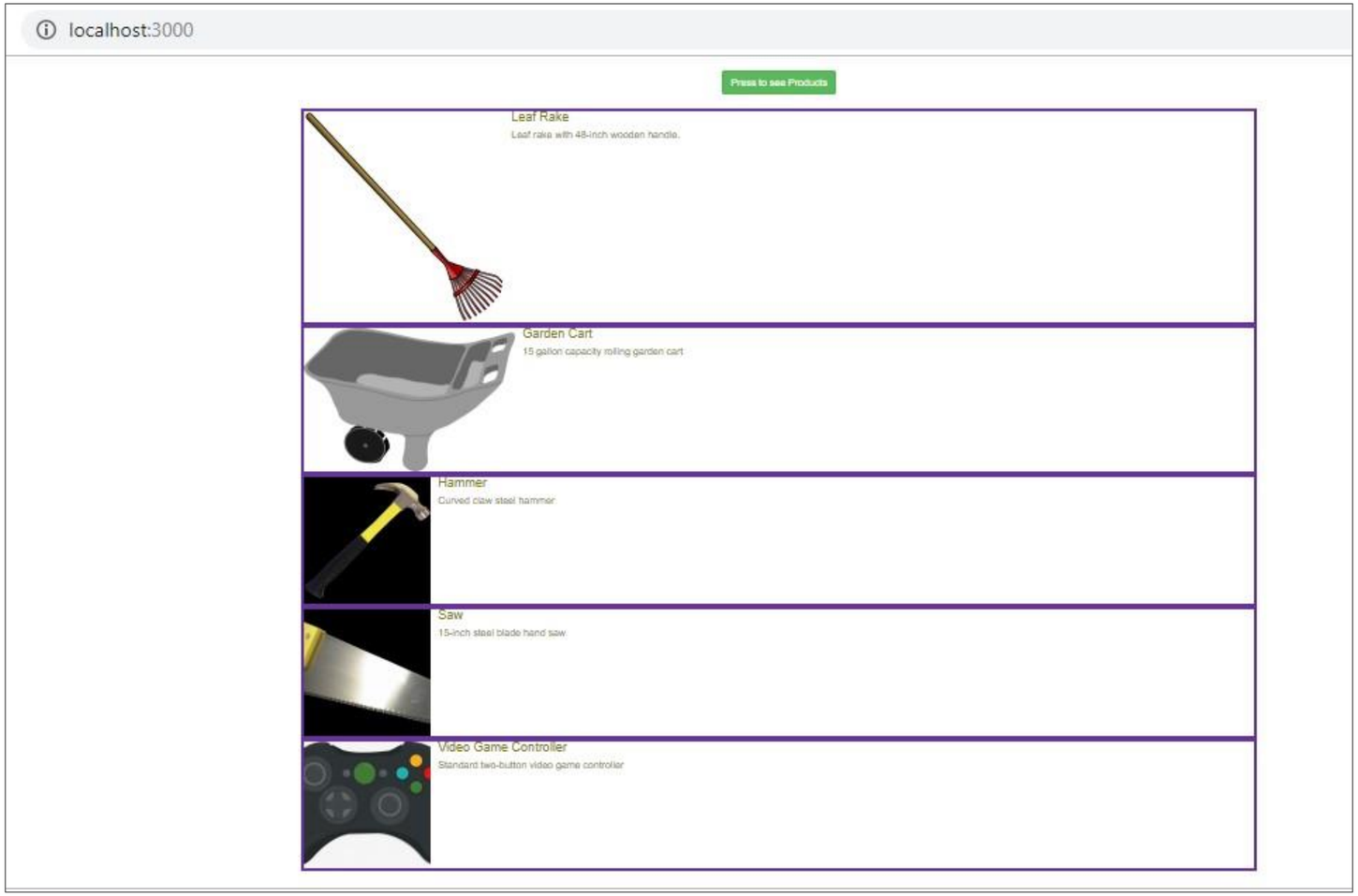


On pressing the button ***Spinner*** is displayed and simultaneously API is called.



Demo: Output

After receiving the data from API, Products List is displayed.



Redux DevTools

Redux-DevTools

Redux-Devtools provide us debugging platform for Redux apps. It allows us to perform time-travel debugging and live editing. It allows you to inspect every state and action payload.

There are two variants of Redux-DevTools: ***Redux DevTools and Redux DevTools Extension***

- ***Redux DevTools*** – It can be installed as a package and integrated into your application
- Install using the command :***npm install --save redux-devtools-extension***
- Add it to your application code as mentioned below

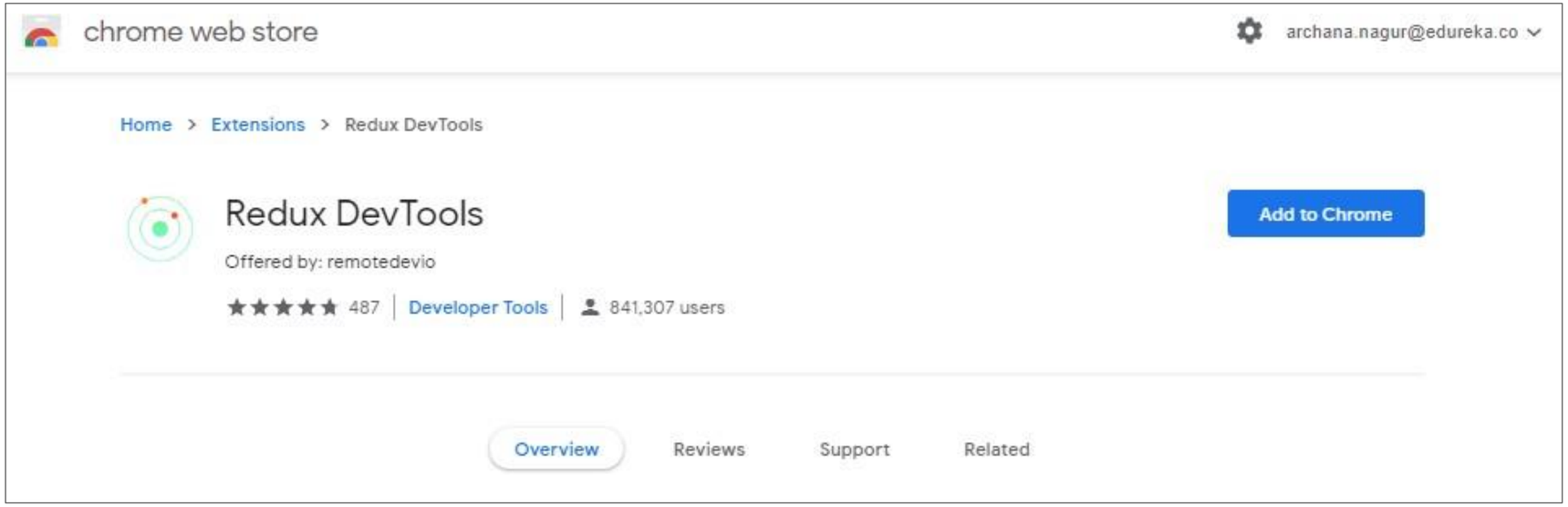
```
import { createStore, applyMiddleware } from 'redux';
import { composeWithDevTools } from 'redux-devtools-
extension';

const store = createStore(reducer, composeWithDevTools(
  applyMiddleware(...middleware),
  // other store enhancers if any
));
```

Redux DevTools Extension

Redux DevTools Extension is a browser extension that implements the developer tools for Redux

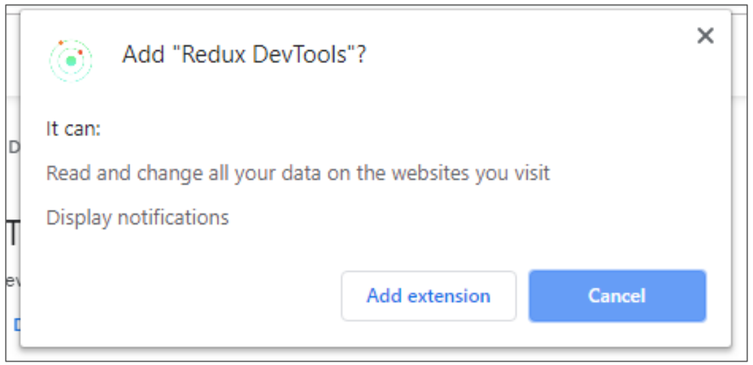
Make use of the link mentioned below, and click on Add to chrome



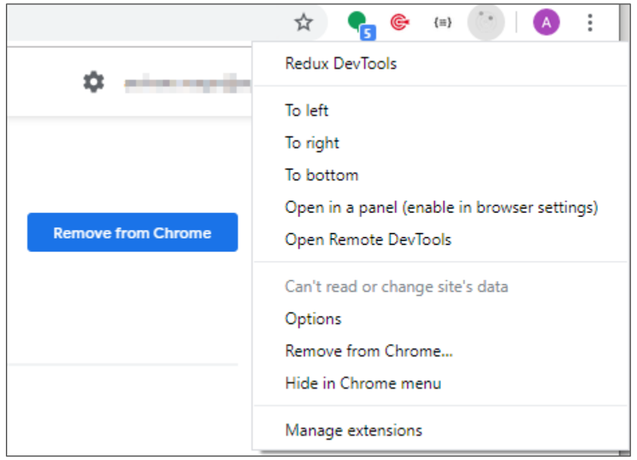
Source: https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en

Redux-Devtools: Redux DevTools Extension

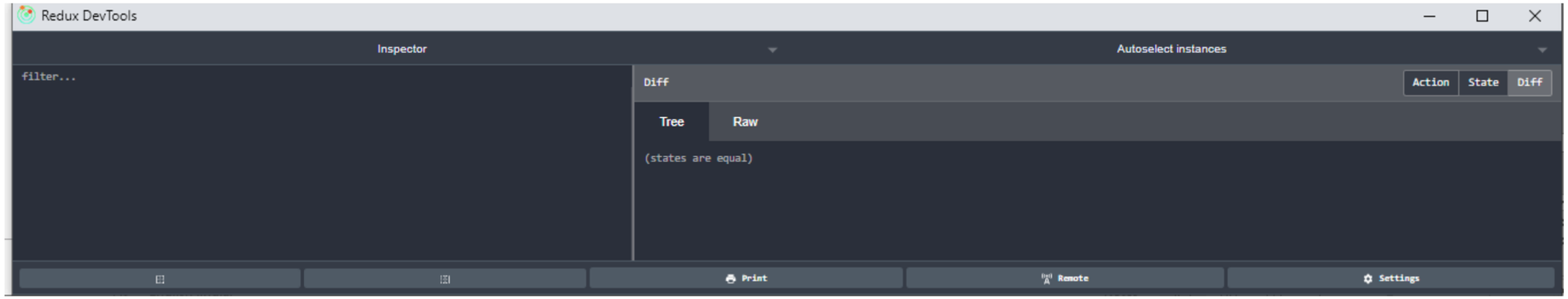
Click on Add extention



Run your application in browser and click on Devtools extension symbol



Debug your application and check the state transformation and props



Questions


Comments
Ratings
Suggestions
FEEDBACK
Survey
Likes
Ideas