**Babel: React code is written with ES6 Javascript and as we know very few browsers are supporting it. So babel will compile our ES6 code to ES5 bundles, so that it will support in all browsers**

**Webpack: Webpack generates all of our code to bundle so that JSX code that we write with react will get converted to browser compatible ES5 code. Then it will run our bundle file with appropriate code.**

**Instll Create –ract-app node pakage using npm:**

**npm install -g create-react-app**

**create react project:**

**create-react-app rayan\_react**

**React DOM : It is responsible for rendering our webpage.**

**React scripts: Will install babel & webpack for us. And it will give webpack node server also**

**Not we will start react application :**

**Npm start**

Remove all the files of **src** folder except index.js file. And clear the content of indexjs file

React make use of elements to describe the user interface about our application

React.createElement('h1',null,'Welcome to React App!');

Firstt tag is html element, 2nd tag is prop and thir tag is content

React elements only describe how the web page is going to look but in real what actually renders our webspage is the DOM

Components: Components are the min building blocks of an application to help us to break down the UI component into small reusable pieces.By combiling all these UI components into a single Main component that’s going to form all of the UI required. And this is called compositional model.

Class way of creating component:

class List extends Component{

render(){

return (

<ol>

{tasks.map((task,index) => <li key={index}>{task}</li>)}

</ol>

)

}

}

class Title extends Component{

render(){

return <h1>Task List</h1>

}

}

class Main extends Component{

render(){

return <div>

<Title/>

<List/>

<List/>

</div>

}

}

ReactDom.render(<Main/>,document.getElementById('root'));

Component models with props:

import React, {Component} from 'react';

import ReactDom from 'react-dom';

class List extends Component{

render(){

console.log('items', this.props.taks);

return (

<ol>

{this.props.tasks.map((task,index) => <li key={index}>{task}</li>)}

</ol>

)

}

}

class Title extends Component{

render(){

return <h1>{this.props.title}</h1>

}

}

class Main extends Component{

render(){

return <div>

<Title title={'To Do List'}/>

<List tasks = {['Anil','Sunil']}/>

<List tasks = {['Nancy','Neha']}/>

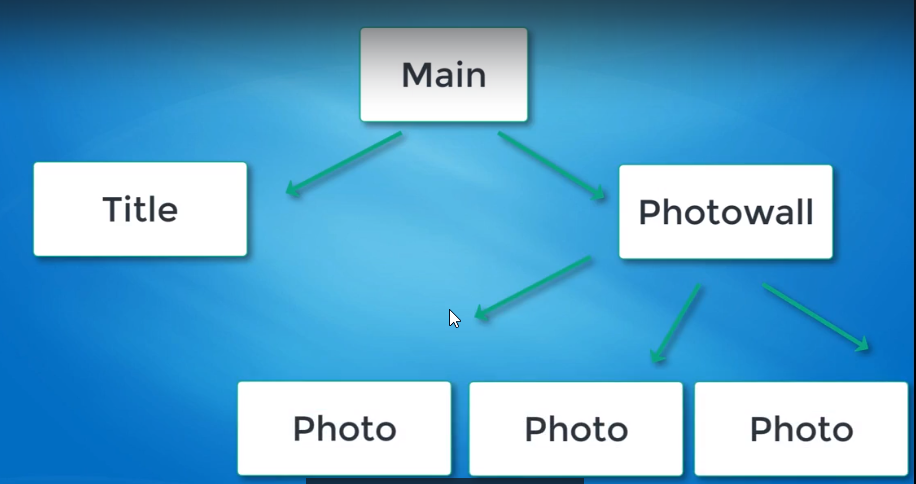
</div>

}

}

ReactDom.render(<Main/>,document.getElementById('root'));

Photowall application:

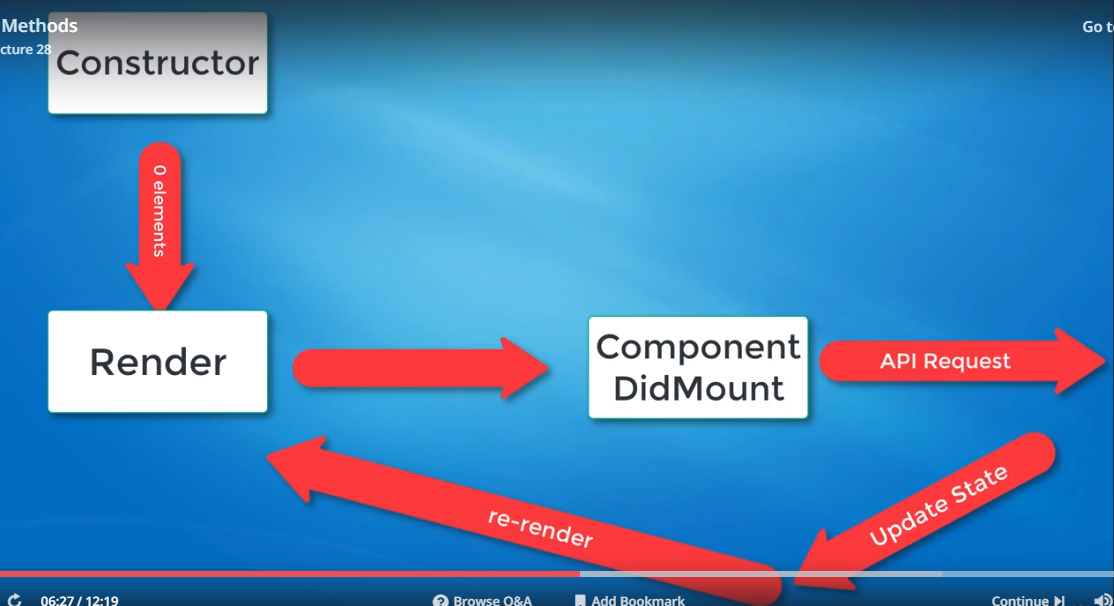


Proptype will help us to avoid debugging because it wil throw big error if we missed something or we should define prop type required in component.

Different Lifecycle methods of components:

Method 1 Constructor Method: The constructor Method gets called once before our component instance is even monted or before it is even inserted into the DOM. In constructor you don’t ever want to include any side of logic by changing the value of a variable writing to it or triggering a change in our UI. The constructor should only be use to initialize

Both render ad constructor automatically invoked unlike function.



ComponentWillMount: It will invoked before component inserted into the DOM or before render methods get invoked.

Life cycle methods get called whenever component is re rendered or we can also say that whenever we are updating the state of component

Make DB request on component did mount

Here's a list of some **common**lifecycle methods used in React.

quick definition before we start with the methods:

**side-effect**: modifying the state of something else outside of its scope. We'll consider asynchronous calls a side effect, because they will change the state in some way.

**The following methods are called when a component is being added to the DOM:**

* **constructor**() : called before component is mounted. NEVER put side effect code inside of the constructor. Use ComponentDidMount for that instead. Commonly used to initialize state or bind methods.
* **componentWillMount**(): invoked immediately before mounting occurs. Called before render. Once again, DO NOT put any side effect code inside of componentWillMount, and do not make API calls in this method
* render(): never fetch data inside of render. Should only be used to return elements.
* **componentDidMount**(): Perfect place to fetch data. It gets called after render. This makes it clear that the initial state is empty at first, until we fetch it and re-trigger render. This forces us to set up our initial state properly, otherwise you're likely to get undefined states.

**The following methods are called when a component is re-rendered to the DOM**

* **componentDidUpdate()**: called when the state of a component changes. Perfect place to update UI or make network calls based on previous state before update, and current state

Router:

Brower router it will keep track of url changes

Change in url on being clicked on kind of hyperlink

Router contain a given path whatever url were directed by the link component. We are going to find proper route which has a given path associated with that link and that path will have its own UI to display. As we are navigating through these various links these urls with the combination of bowser outer always keep track of all the url changes, the URL in the changes will always be in Sync

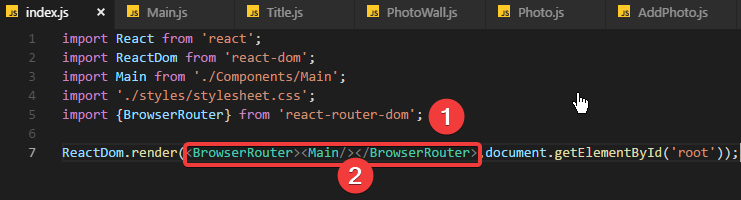
.

Step 1:

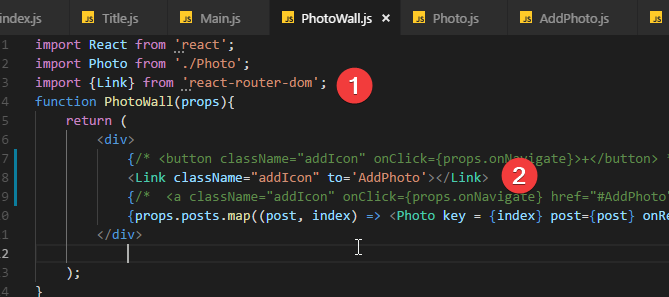
Install router npm command:

npm install [react-router-dom@4.2.2](mailto:react-router-dom@4.2.2)

Step 2



Import {Link} in photowall.js



Browser Router: will keep track of the browser changes and urls

Linked Component: It is incharge of changing URLS . It invokes the changes in URL upon being clicked on.

Route: For every piece of UI we are going to assign route if that route path matches the URL that we are currently on It will render that piece of UI

Route:

Step 3: In Main.js :

import {Route} from 'react-router-dom';

In render function change text like this :

render(){

return(

<div>

{/\* Metho 1 of Routing \*/}

<Route exact path = "/" render={() => (

<div className="photoGrid">

<Title title={'Photowall'}/>

<PhotoWall posts = {this.state.posts} onRemovePhoto = {this.removePhoto} onNavigate = {this.navigate} />

</div>

)} />

{/\* Method 2 or Routing \*/}

<Route path = "/AddPhoto" component={AddPhoto}/>

</div>

)

}

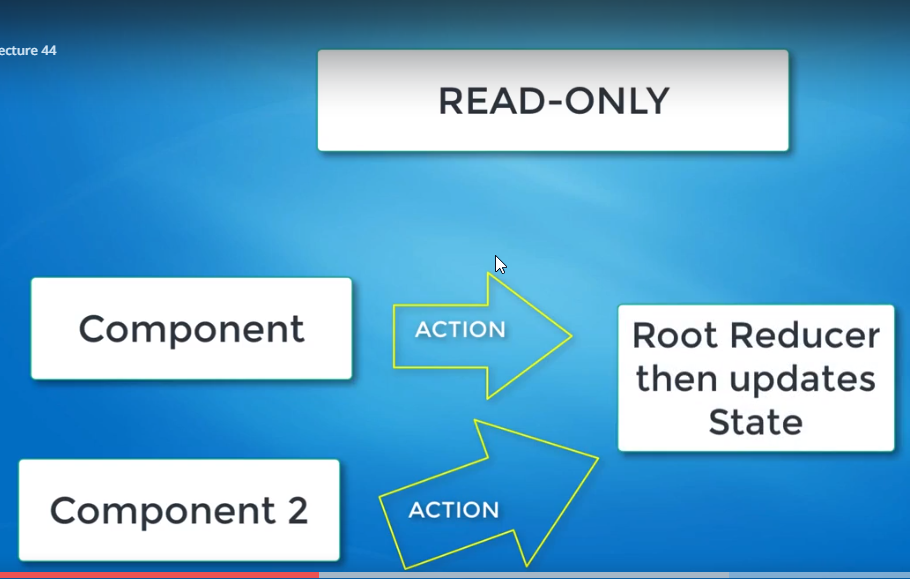
Sort the gallery list with recent is first:

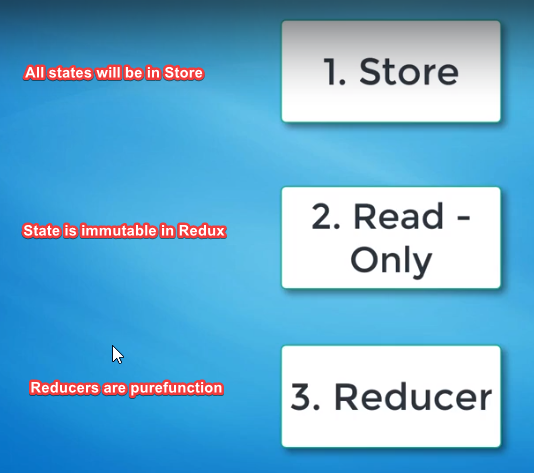
Why Redux:

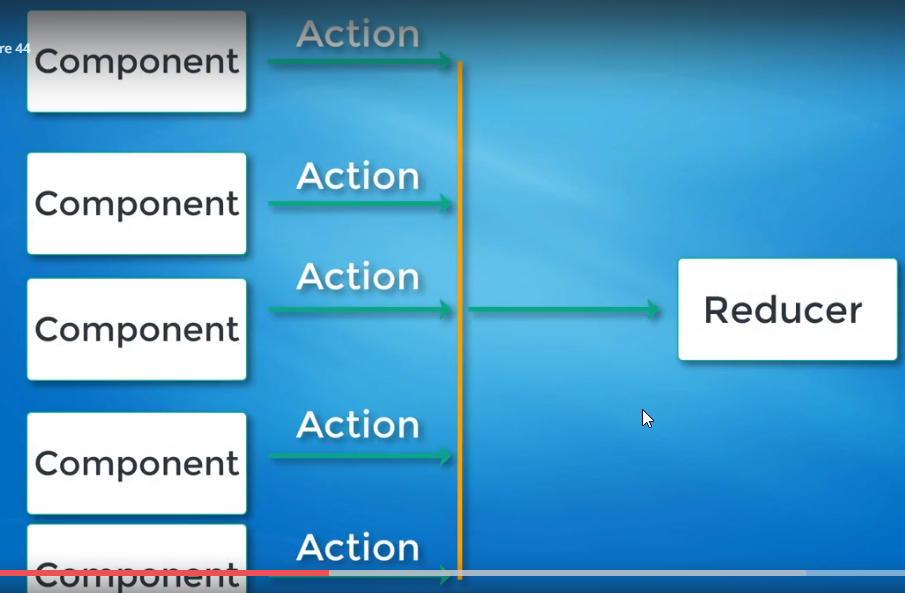
* 1. Predictability
  2. Complexity = bugs

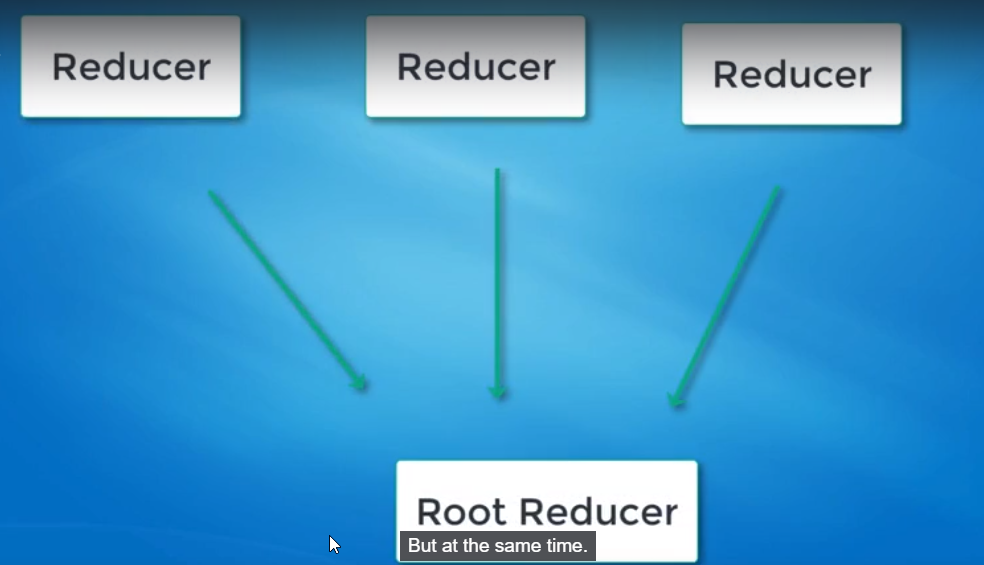
**State in Redux is read only and in redux components can’t change state directly except all it can do it can emit an action to describe how it wants to update the state .**

**All state changes are happened in reducers**

****

****

****

****

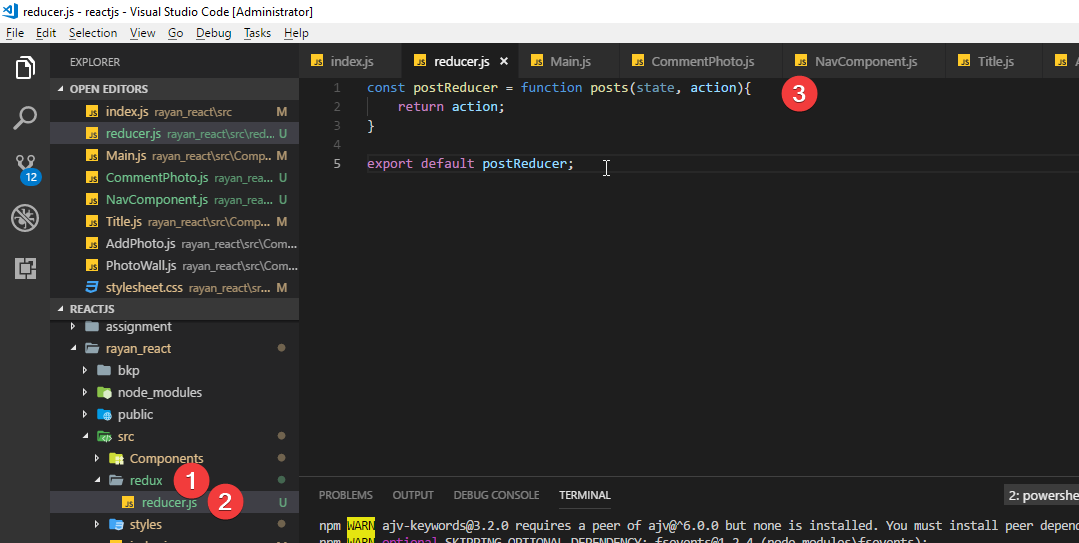
**Redux installation:**

**Step 1 : install Redux via npm:**

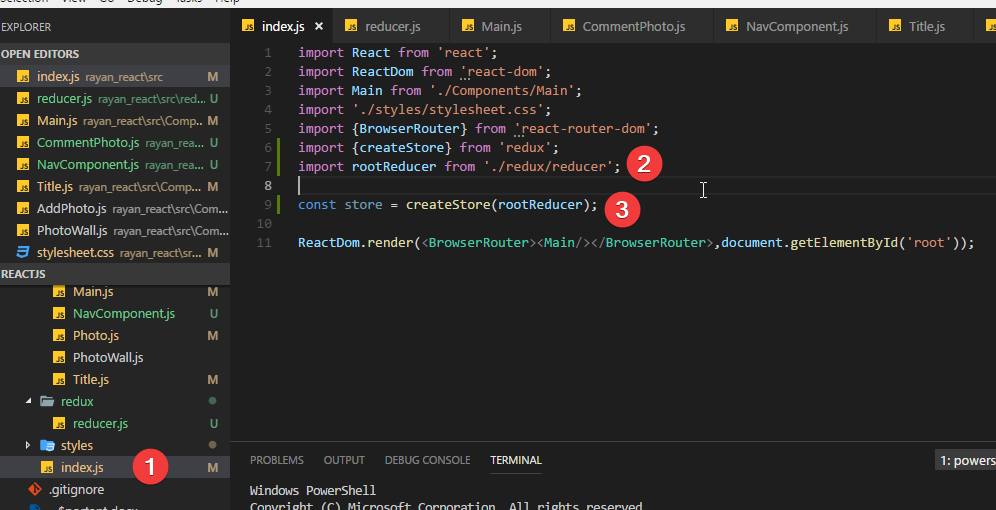
**npm install** [**redux@3.7.2**](mailto:redux@3.7.2)

**Step2: in index.js import redux**

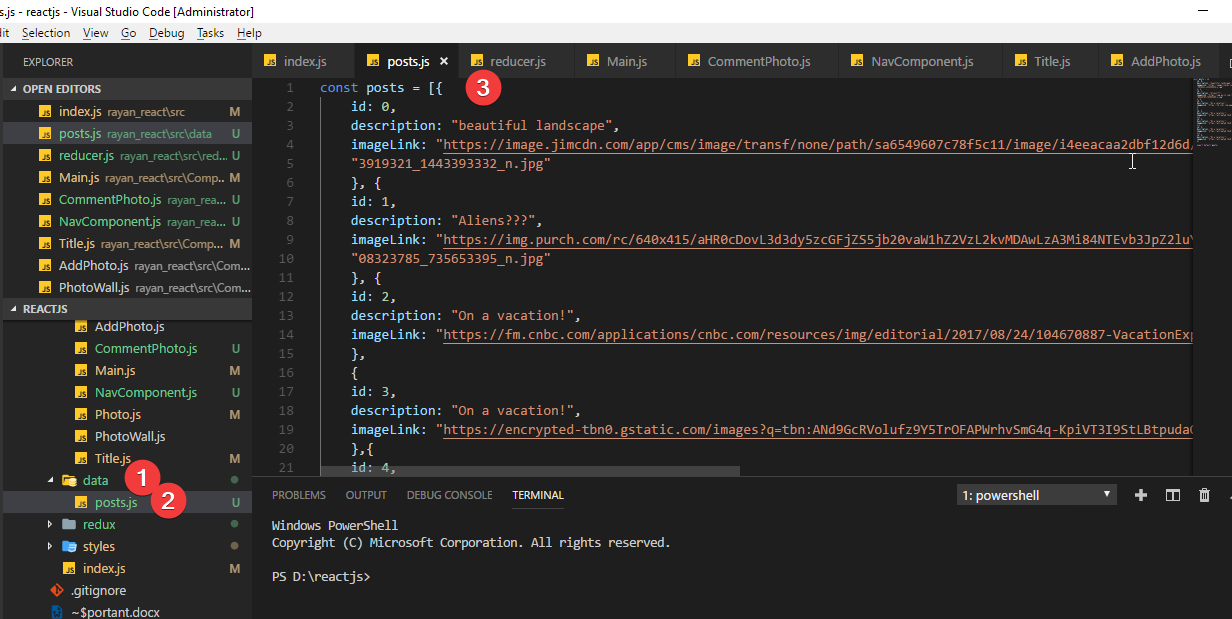
import {createStore} from 'redux';

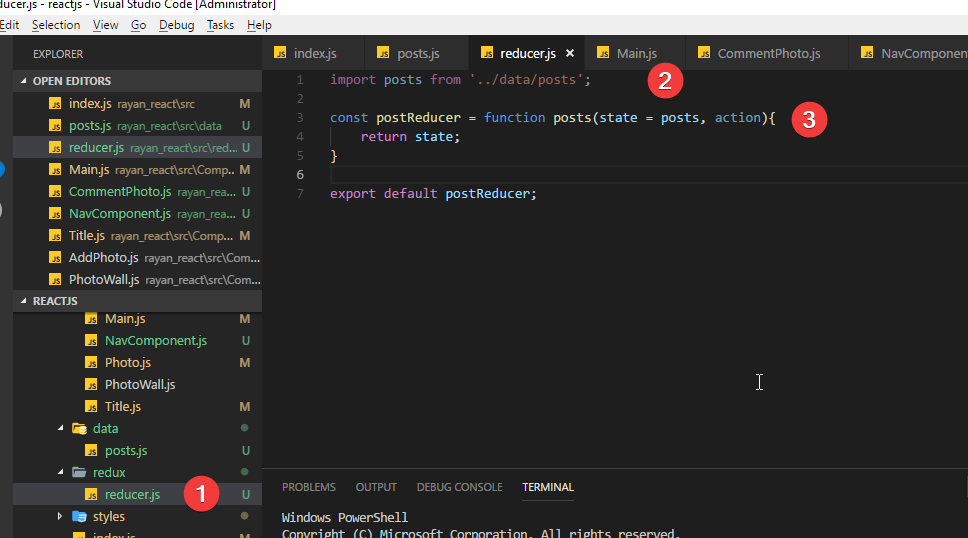
****

**Declare rootReducer and pass it through createStore:**

****

**Create Reducer data source file :**

****

**Assign Posts data to post reducer state:**

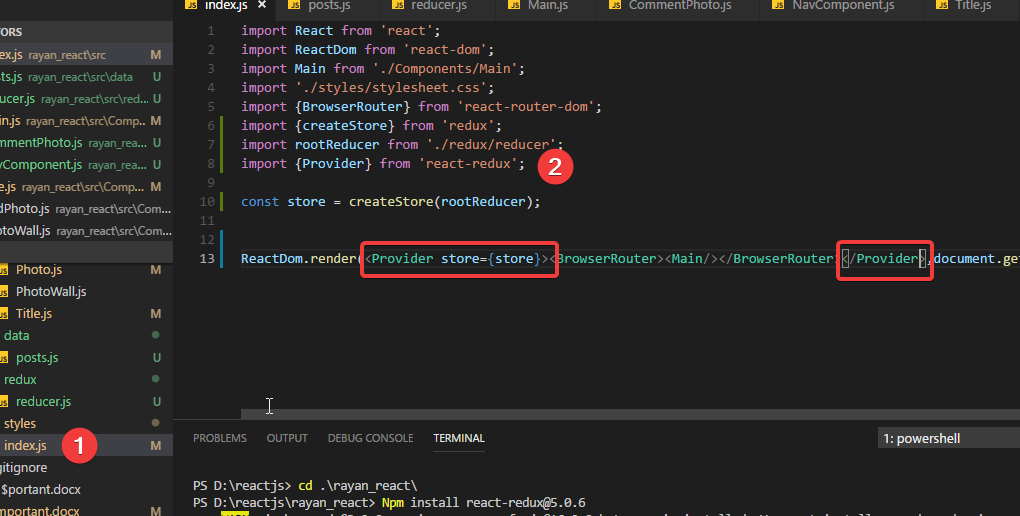
**Provider: it will give ability to component to receive or access the store directly. We can its main purpose is to provide our store to its child components**

****

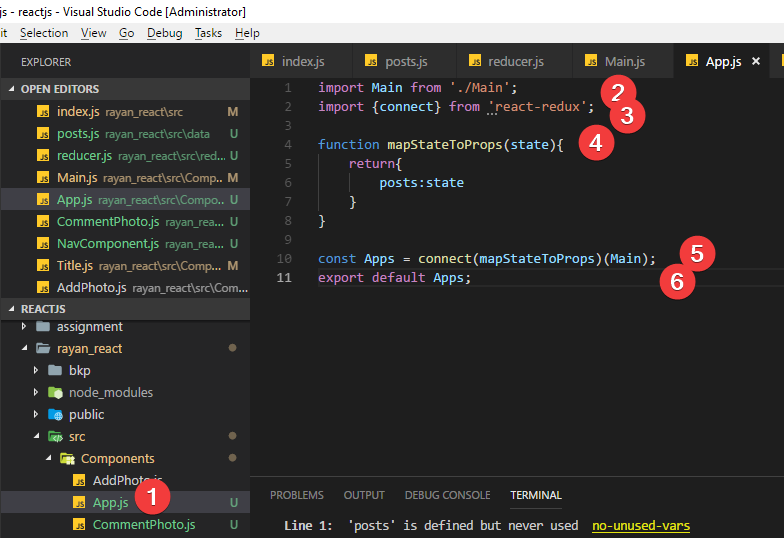
**We should install react-redux dependency to access Providers**

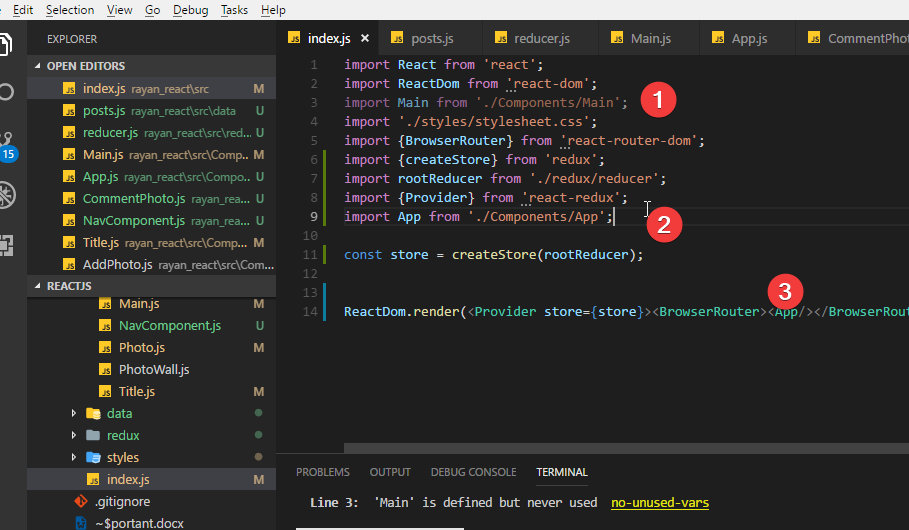
**Npm install** [**react-redux@5.0.6**](mailto:react-redux@5.0.6)

**Connected Component:**

****

**Connect our redux store to Main component:**

****

****

**And remove unwanted code from Main.js file:**

import React, {Component} from 'react';

import Title from './Title';

import PhotoWall from './PhotoWall';

import AddPhoto from './AddPhoto';

import {Route} from 'react-router-dom';

class Main extends Component{

constructor(){

super()

}

render(){

//console.log(this.state.posts);

return(

<div>

<Route exact path="/" render={()=>(

<div>

<Title title={'My Gallery'}/>

{/\* <PhotoWall posts = {this.state.posts} onRemovePhoto = {this.removePhoto} onNavigate = {this.navigate} /> \*/}

</div>

)} />

{/\* <Route path="/AddPhoto" render={({history})=>(

<div>

<Title title={'Add Photo'}/>

<AddPhoto onAddphoto={(addedPost) => {

console.log(addedPost);

this.addPhoto(addedPost)

history.push('/')

}}

/>

</div>

)}/> \*/}

</div>

)

}

}

export default Main;

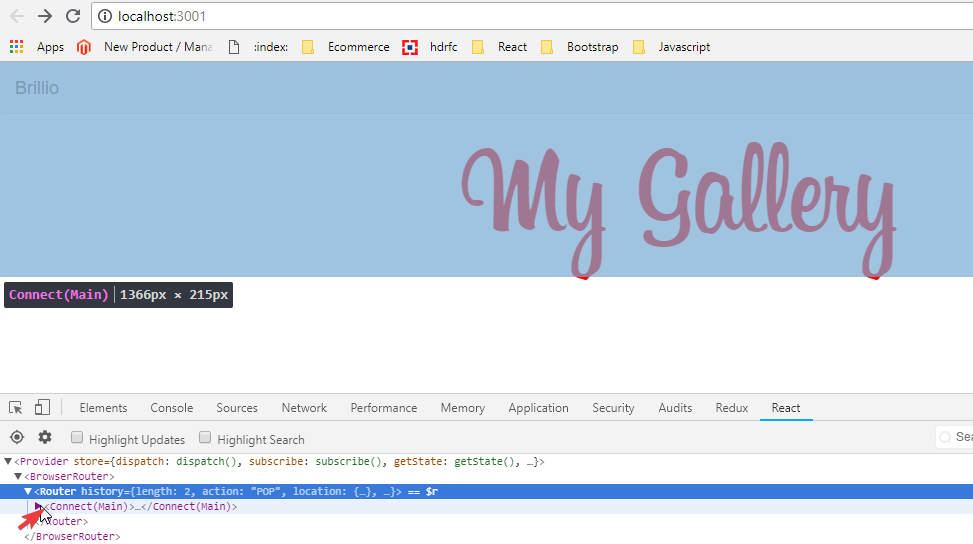
// <div>

// <Title title={'Comment Photo'}/>

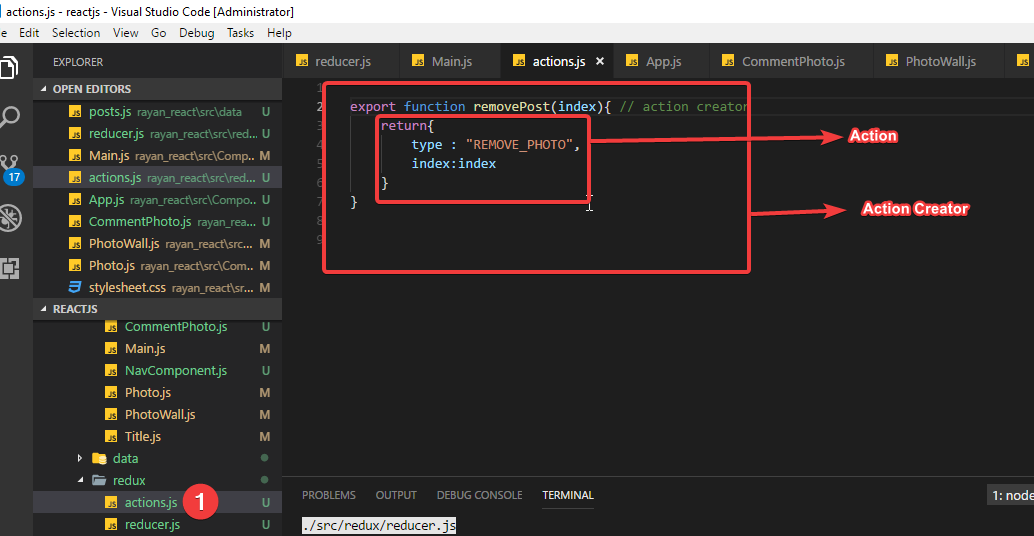
// <CommentPhoto />

// </div>

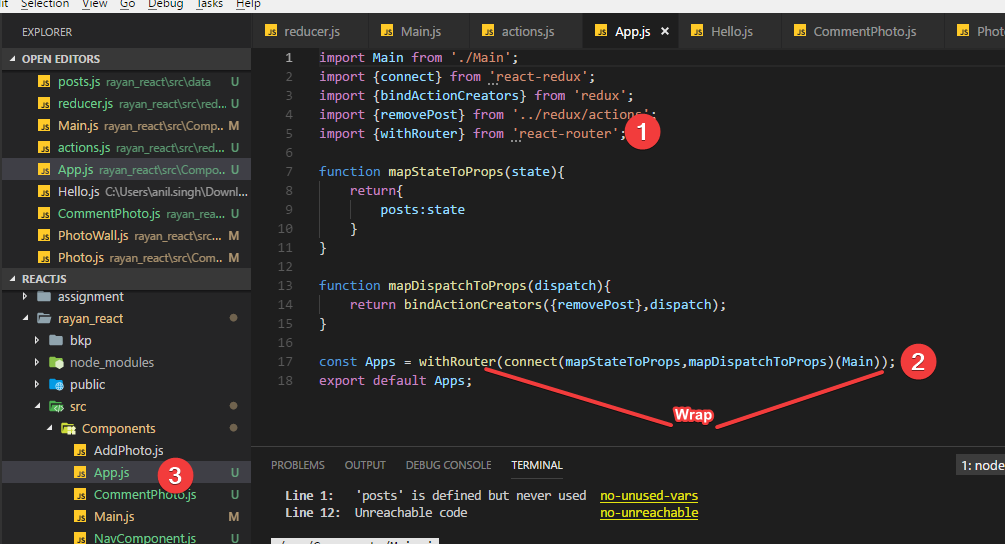
**After successful connection we will see following in console:**

****

**Action & action creator:**

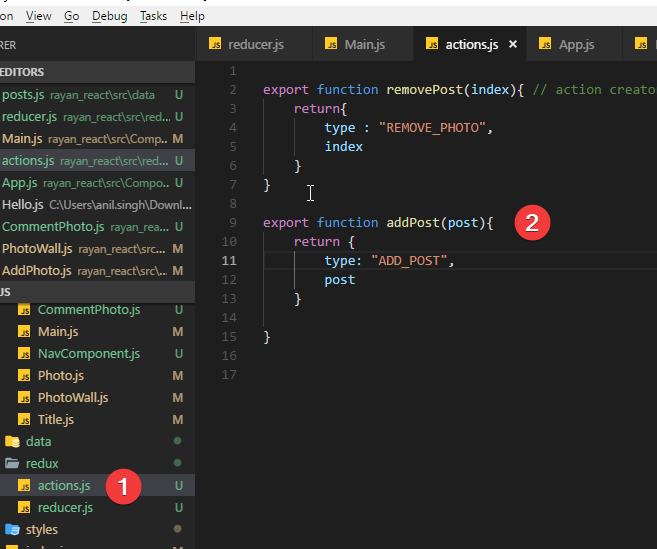
****

**After adding action to reducers our routing url not woking properly for that we should do following activities:**

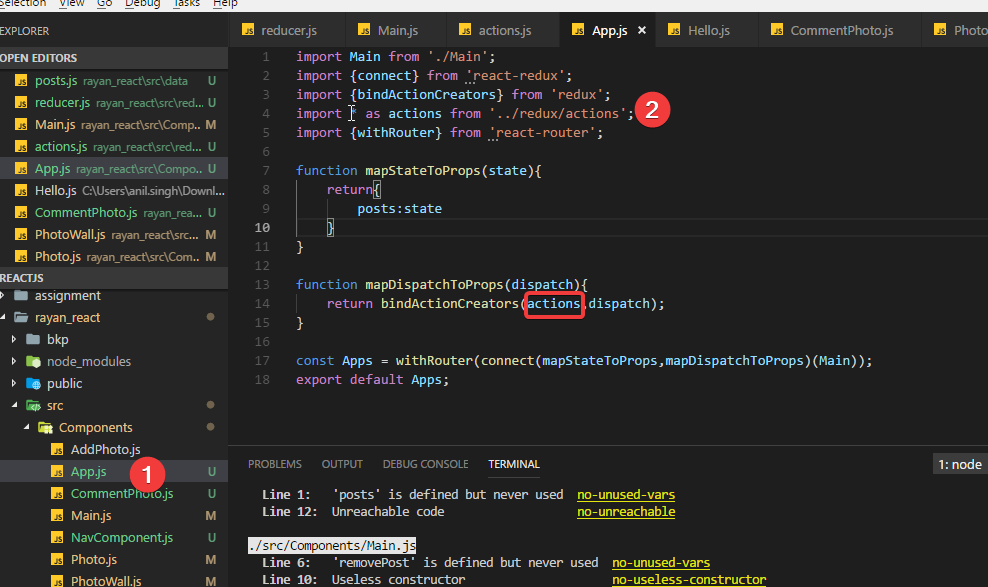
****

**Add Posts:**

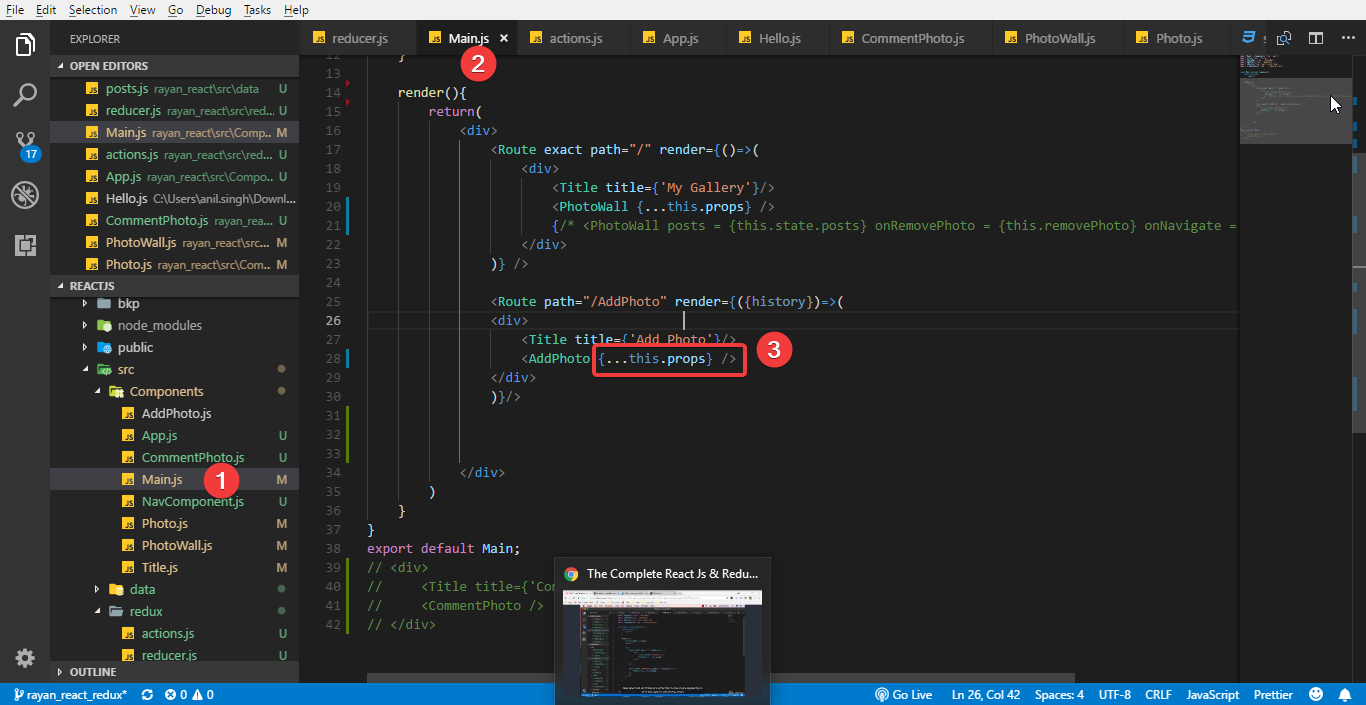
**Add action in Action.js:**

****

**Import all actions into App.js:**

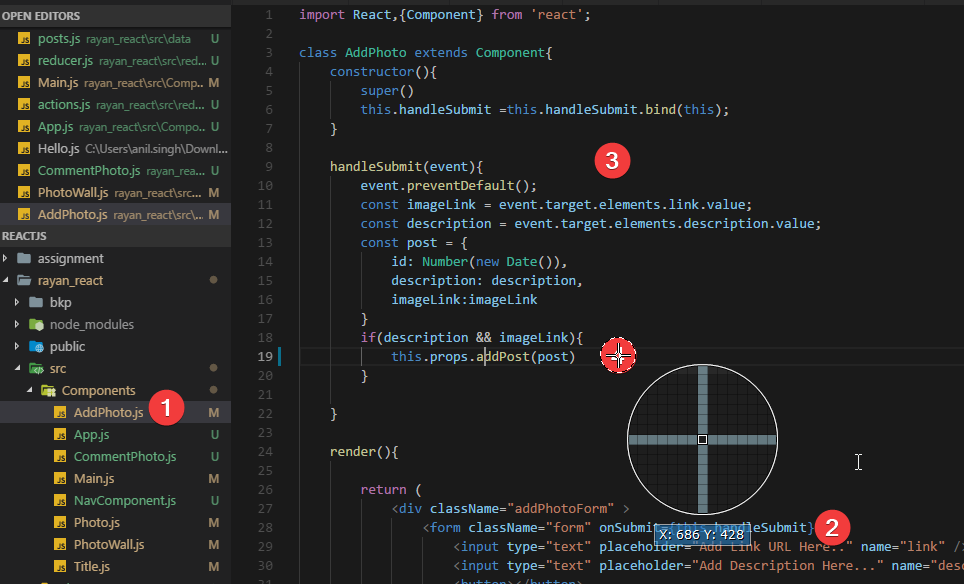
****

**Pass props value from Main.js to Addphotocomponent:**

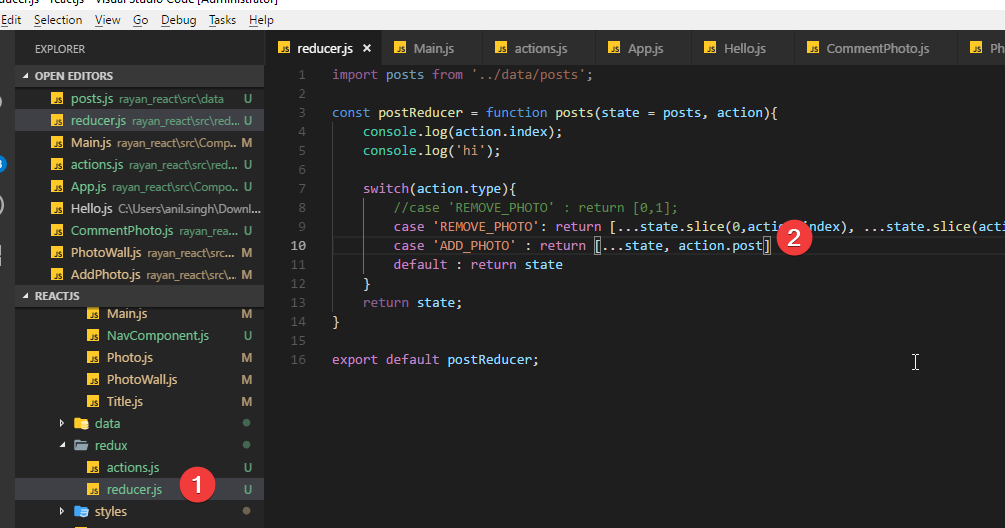
****

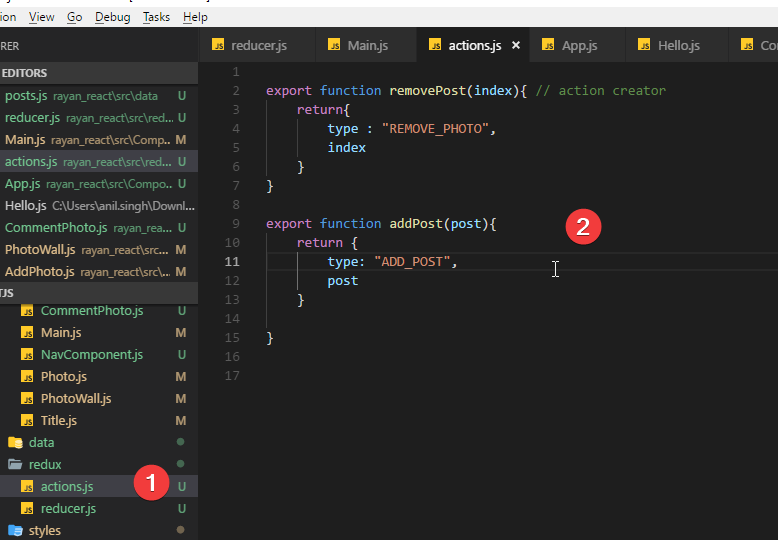
**Summary:**

**We submit the form that emits the action and that action goes to the reducer**

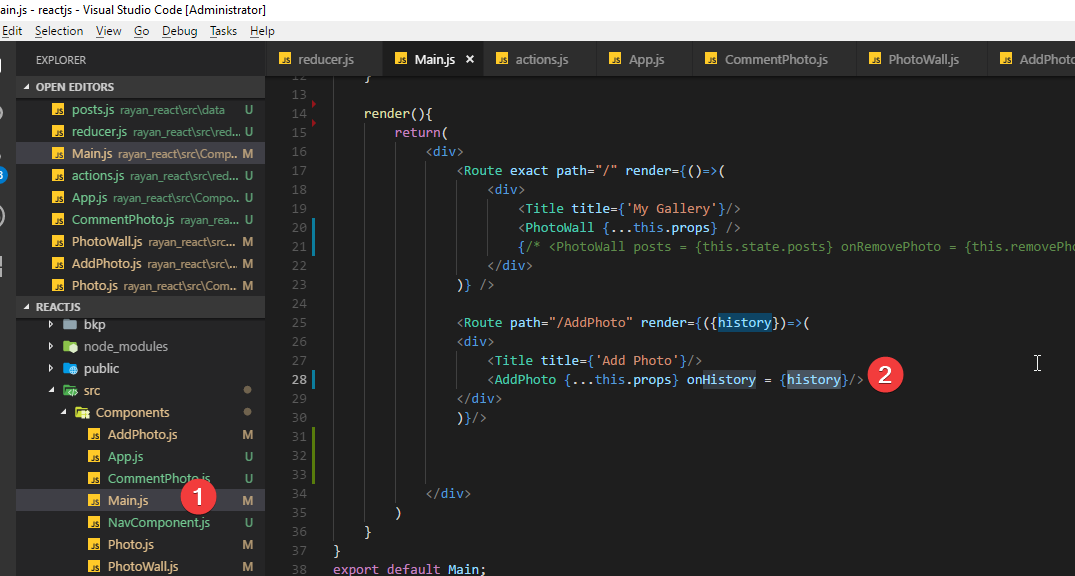
****

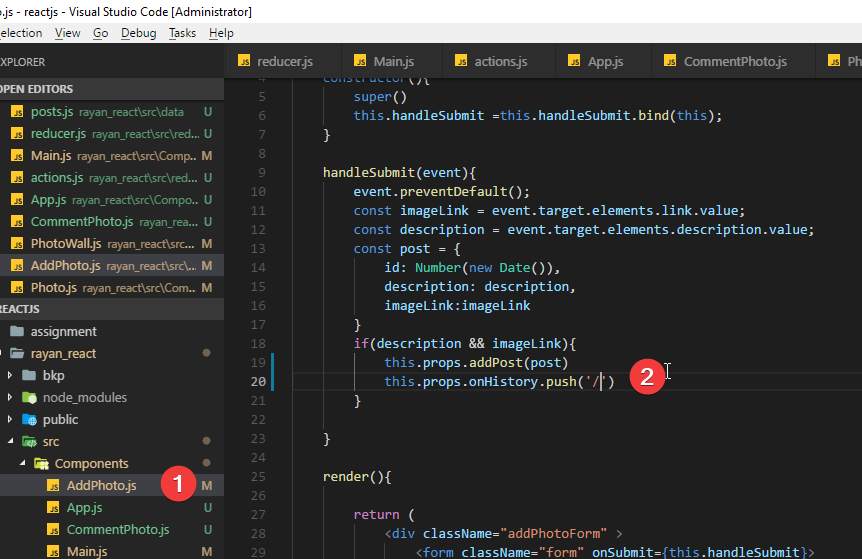
**So it will return the current array of posts and it will also contains the state that was just submitted. And passed into our action as Payload**

****

****

**Pass history to addpost component via main.js:**

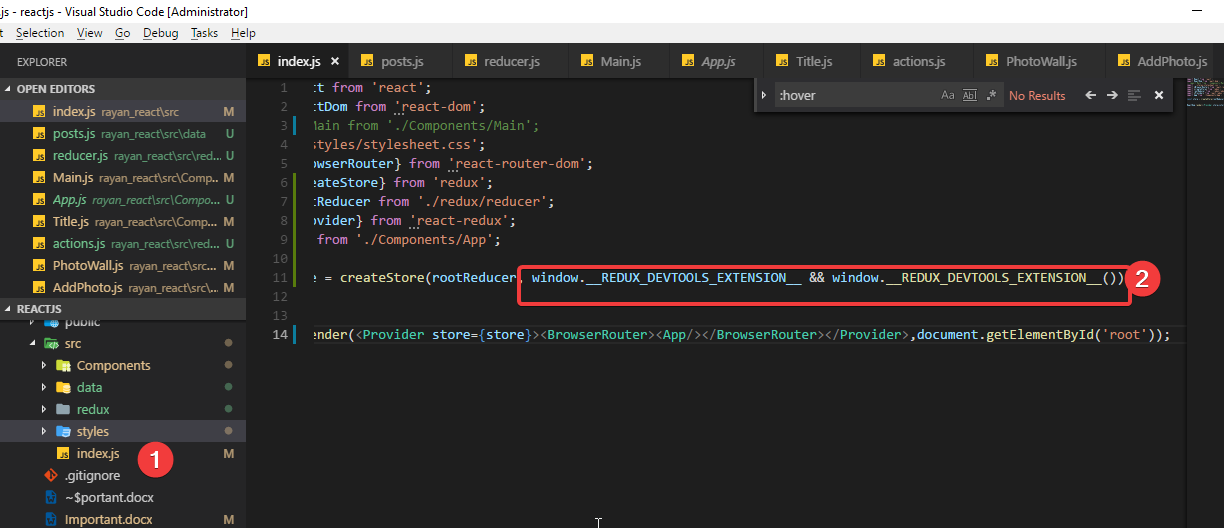
****

****

**Redux dev tools:**

**Configure following command to store:**

window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()

****