**Redux: End-2-End Implementation**

npm install -g create-react-app

create-react-app crud-redux

cd src

rm App.css App.test.js logo.svg registerServiceWorker.js

cd projectname

npm install --save redux

npm install –save react-redux

Redux is a state management library that gives you access to the state anywhere in your components without the need to pass props. So it can be used with any front-end libraries like Angular and React but it works best with React. ‘react-redux’ is the official library that connects the two.

Since we deleted some of those files earlier we need to make some changes in index.js and App.js which are as follows-

Go to crud-redux/src/index.js

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React from 'react'; | |  | import ReactDOM from 'react-dom'; | |  | import './index.css'; | |  | import App from './App'; | |  |  | |  |  | |  | ReactDOM.render( | |  | <App />,document.getElementById('root')); | |

and crud-redux/src/App.js

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | class App extends Component { | |  | render() { | |  | return ( | |  | <div className="App"> | |  | <h1>Hello React!</h1> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default App; | |

With that done, let’s start the server and make sure everything is working before we start writing any code. To start the server type the following command in the terminal

npm start

Whenever I make any React application I always try to make the basic version of it and then add interactivity to it. So with that in mind let’s create some components. In crud-redux/App.js do the following-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  | import PostForm from './PostForm';//user defined component | |  | import AllPost from './AllPost'; //user defined component | |  |  | |  |  | |  | class App extends Component { | |  | render() { | |  | return ( | |  | <div className="App"> | |  | <PostForm /> | |  | <AllPost /> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default App; | |

Here I have created two components. The PostForm component will contain the form elements for creating a post and the AllPost component will contain all the posts. So let’s create the files for each of these components. Under src folder create two files called ‘PostForm.js’ and ‘AllPost.js’.

Inside PostForm.js add the following code-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | class PostForm extends Component { | |  | render() { | |  | return ( | |  | <div> | |  | <h1>Create Post</h1> | |  | <form> | |  | <input required type="text" placeholder="Enter Post Title" /><br /><br /> | |  | <textarea required rows="5" cols="28" placeholder="Enter Post" /><br /><br /> | |  | <button>Post</button> | |  | </form> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default PostForm; | |

We will add styles later so let’s get on with the other component. Inside crud-redux/src/AllPost.js add the following lines-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | class AllPost extends Component { | |  | render() { | |  | return ( | |  | <div> | |  | <h1>All Posts</h1> | |  | </div> | |  | ); | |  | } | |  | } | |  |  | |  | export default AllPost; | |

Now that we have our basic UI in place let’s get into Redux. First thing to understand about Redux is something called the **store**. It’s where the entire state of your application will live. This is the first main benefit of using Redux. Instead of having to manage the state in different components we have to only manage it in one single place called the store. The store is an object which has some methods in it that allows us to get the current state of our application, subscribe to changes or update the existing state of our application. This is great because now we don’t have to pass down data from the parent component to deeply nested child components through props. So anytime a component needs data it can ask the store and the store will provide it with the data. As simple as that. With that in mind let’s create the store. In our crud-redux/src/index.js make the following changes-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React from 'react'; | |  | import ReactDOM from 'react-dom'; | |  | import './index.css'; | |  | import App from './App'; | |  |  | |  | import { createStore } from 'redux'; | |  |  | |  | const store = createStore(); | |  |  | |  |  | |  | ReactDOM.render(<App />, document.getElementById('root')); | |

The **createStore** method will allow us to create the store but we are not done yet. This method needs a special argument and this argument goes by a special name called the ‘**reducer**’. Let’s create a separate folder called reducers. So under crud-redux/src create a folder called ‘reducers’. Inside that folder create a file called postReducer.js Add the following code for now.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | const postReducer = (state = [], action) => { | |  |  | |  | } | |  | export default postReducer; | |

Cost postReducer = (state=[], action)-> {

If(action.TYPE==”ADD”)

If (action.TYPE==”GET”)

…

}

Export default postReducer;

We will fill in the contents of that function a bit later. Now let’s understand another important concept in Redux called **actions**. Actions are nothing but plain Javascript objects with a type property. This type property describes the event that is taking place in the application. This event can be anything from incrementing a counter to adding items in an array. These actions help us track the different events that are happening in our application. The structure of an action is as follows-

{  
 type: 'EVENT\_NAME'

}

An action can have any number of properties but it must have a type property. So an action can include data like so

{  
 type:'ADD\_ITEM',  
 name: 'Redux'

}

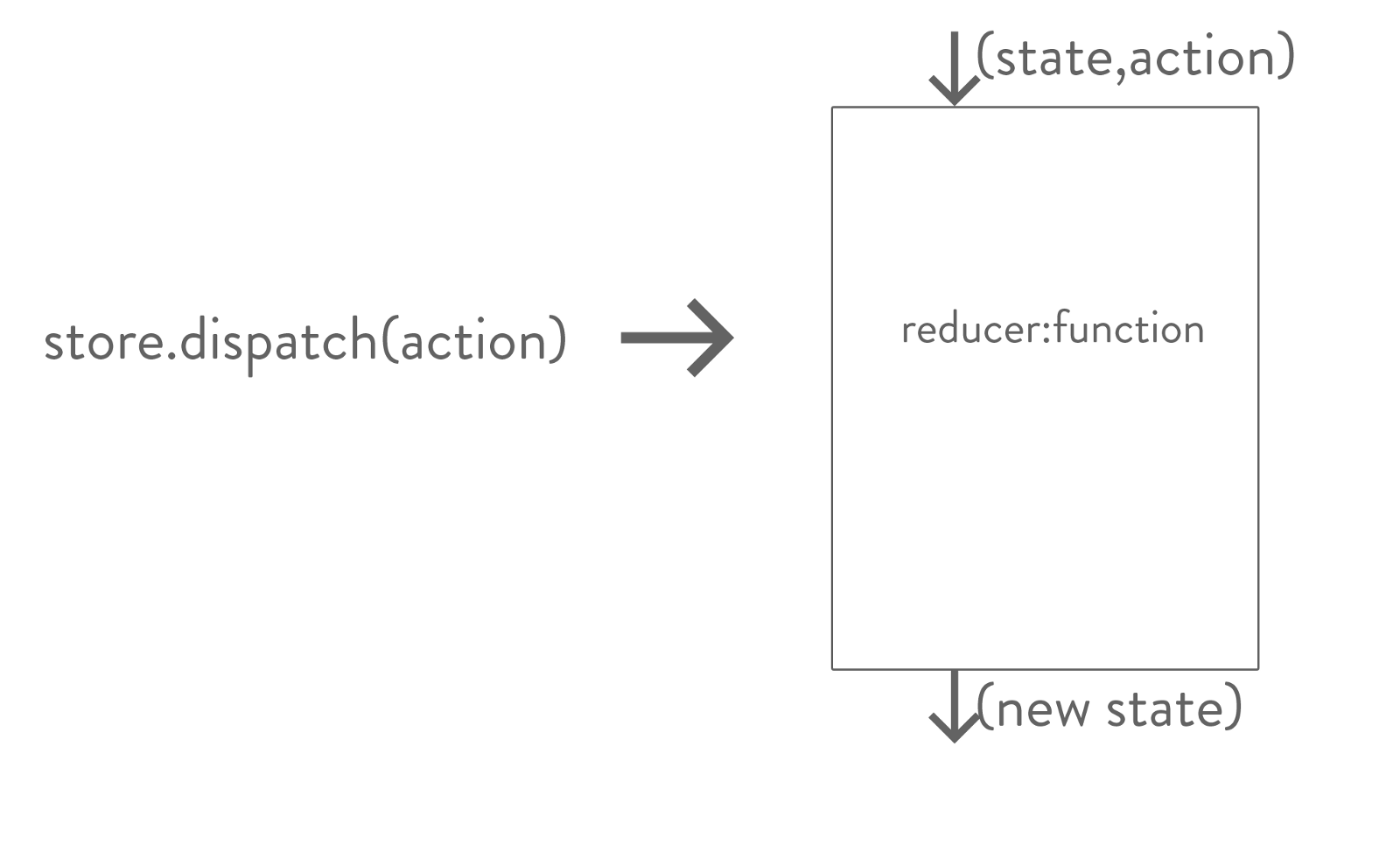
In this example the event name is ‘ADD\_ITEM’ and the data is the name property with a value of ‘Redux’. Now another important term that is used alongside actions is called dispatch. When we say ‘dispatch an action’ we simply mean call the dispatch method which is inside the store object with an action. Still with me?

Let’s look at the store. The store that we created using the **createStore**method is an object which has some methods in it. One of those methods is called dispatch. This dispatch method accepts an object as it’s argument and this object is what we call as ‘action’.



what dispatch really is

With that out of the way, let’s finally go back to that function that we wrote earlier inside postReducer.js. You see whenever we dispatch an action, this action with it’s type property is received by something called the reducer. Now what the heck is the reducer? Well it’s nothing but a function that takes the **current state**and an **action**that was dispatched as it’s parameters and returns the **new state.**



What Reducer really is

So next time when you see the term reducer thrown around remember that it’s just a function that gives you new state for your components.

Now the question is how does the reducer go about producing the new state for the application. Well that is pretty simple, it first checks which **type**of action was dispatched and based on it returns the new state. Under crud-redux/src/reducers/postReducer.js add the following lines of code.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | const postReducer = (state = [], action) => { | |  | switch(action.type) { | |  | case 'ADD\_POST': | |  | return state.concat([action.data]); | |  | default: | |  | return state; | |  | } | |  | } | |  | export default postReducer; | |

Now what is happening here is that we are using a ‘switch statement’ and we are switching based on the value of **action.type.**If the value is ‘ADD\_POST’ we are returning a new array containing action.data. Basically whenever the ‘ADD\_POST’ event happens we want to push some data into the state array.Now what is action.data? Well it’s nothing but an object with our individual post title and the post message. One thing to note here is that the reducer function expects a default value for the state. Here we are using ES6 default-parameter syntax to add that. The default value for the state here is an empty array. One other thing to note is that a reducer must always have the default clause inside the switch statement. In the default clause we simply return the state. This is done so that in case none of the action.type value matches any of the cases we simply return the state.

Now that we have some code inside postReducer.js let’s import it in our index.js file and pass it to the store as an argument.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React from 'react'; | |  | import ReactDOM from 'react-dom'; | |  | import './index.css'; | |  | import App from './App'; | |  | import { createStore } from 'redux'; | |  |  | |  | import postReducer from './reducers/postReducer'; | |  |  | |  | const store = createStore(postReducer); | |  | ReactDOM.render(<App />,document.getElementById('root')); | |

Now that we are done with the reducer. Let’s pass this store to our components. To do that let’s use the Provider component from the ‘react-redux’ library. Change crud-redux/src/index.js as follows-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React from 'react'; | |  | import ReactDOM from 'react-dom'; | |  | import './index.css'; | |  | import App from './App'; | |  | import { createStore } from 'redux'; | |  | import { Provider } from 'react-redux'; | |  |  | |  |  | |  | import postReducer from './reducers/postReducer'; | |  | const store = createStore(postReducer); | |  | ReactDOM.render( | |  | <Provider store={store}> | |  | <App /> | |  | </Provider>, document.getElementById('root')); | |

The Provider component uses something called React Context which allows you to pass the store object to any components that needs to access it without the need to pass props. Here we are wrapping the App component which is our parent component with the Provider component so that all the child components in our app can get access to the store. The Provider component takes the store as a prop.

Let’s head back to our PostForm component and connect it to our store so that we can dispatch actions.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | class PostForm extends Component { | |  | handleSubmit = (e) => { | |  | e.preventDefault(); | |  | const title = this.getTitle.value; | |  | const message = this.getMessage.value; | |  | const data = { | |  | id: new Date(), | |  | title, | |  | Message | |  | } | |  | } | |  | render() { | |  | return ( | |  | <div> | |  | <h1>Create Post</h1> | |  | <form onSubmit={this.handleSubmit}> | |  | <input required type="text" ref={(input)=>this.getTitle = input} | |  | placeholder="Enter Post Title"/> | |  | <br /><br /> | |  | <textarea required rows="5" ref={(input)=>this.getMessage = input} cols="28" | |  | placeholder="Enter Post" /> | |  | <br /><br /> | |  | <button>Post</button> | |  | </form> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default PostForm; | |

So in here the form element now accepts an onSubmit event. Whenever this event takes place the handleSubmit function will execute. The handleSubmit function takes one argument which is the event. Calling e.preventDefault() will prevent the page from refreshing. Next we grab the value of the title and the message from the inputs using refs and then put them inside an object called data. We also have an id property whose value is set to whatever new Date() returns. We will use this id property to perform update and delete operations.

Let’s put in some values in the title and the post fields and log it to the console. This is to make sure that the data is being captured. Add a console.log() in between like in the following-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | class PostForm extends Component { | |  | handleSubmit = (e) => { | |  | e.preventDefault(); | |  | const title = this.getTitle.value; | |  | const message = this.getMessage.value; | |  | const data = { | |  | id: new Date(), | |  | title, | |  | Message | |  | } | |  | console.log(data) | |  | } | |  | render() { | |  | return ( | |  | <div> | |  | <h1>Create Post</h1> | |  | <form onSubmit={this.handleSubmit}> | |  | <input required type="text" ref={(input)=>this.getTitle = input} | |  | placeholder="Enter Post Title"/> | |  | <br /><br /> | |  | <textarea required rows="5" ref={(input)=>this.getMessage = input} cols="28" | |  | placeholder="Enter Post" /> | |  | <br /><br /> | |  | <button>Post</button> | |  | </form> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default PostForm; | |

It seems like our data is being captured properly. Great all is left now is to dispatch an action. To do that we will make use of the **connect()** function provided by the react-redux library. Now this is where things might get a bit tricky but I will try my best to explain it. We know that our state lives inside this object called the store and this store has it’s own set of methods for getting the current state of our application, updating the state of our application and subscribing for changes. We have already discussed one of these methods called dispatch. We need dispatch whenever we want to pass some action to the reducer to tell some sort of event has happened and then the reducer can decide what to do with the state. But to do that we need access to dispatch. Won’t it be great if we somehow got access to the dispatch method as a prop. That is what connect() allows you to do. connect() returns a function which takes in your current component as an argument and returns a new component with dispatch method as it’s prop. The main idea to remember is that connect will ultimately return a new component which has the dispatch method as a prop.The basic syntax for writing connect in your React components is as follows-

export default connect()(component-name)

So let’s use that and add it in our PostForm.js. So after that our component will look like so-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  | import {connect} from 'react-redux'; | |  | class PostForm extends Component { | |  | handleSubmit = (e) => { | |  | e.preventDefault(); | |  | const title = this.getTitle.value; | |  | const message = this.getMessage.value; | |  | const data = { | |  | id: new Date(), | |  | title, | |  | Message | |  | } | |  |  | |  | } | |  | render() { | |  | return ( | |  | <div> | |  | <h1>Create Post</h1> | |  | <form onSubmit={this.handleSubmit}> | |  | <input required type="text" ref={(input)=>this.getTitle = input} | |  | placeholder="Enter Post Title"/> | |  | <br /><br /> | |  | <textarea required rows="5" ref={(input)=>this.getMessage = input} cols="28" | |  | placeholder="Enter Post" /> | |  | <br /><br /> | |  | <button>Post</button> | |  | </form> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default connect()(PostForm); | |

With that in place we can easily access dispatch in our components so let’s use it.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  | import {connect} from 'react-redux'; | |  | class PostForm extends Component { | |  | handleSubmit = (e) => { | |  | e.preventDefault(); | |  | const title = this.getTitle.value; | |  | const message = this.getMessage.value; | |  | const data = { | |  | id: new Date(), | |  | title, | |  | Message | |  | } | |  | this.props.dispatch({ | |  | type:'ADD\_POST', | |  | data}); | |  | this.getTitle.value = ''; | |  | this.getMessage.value = ''; | |  | } | |  | render() { | |  | return ( | |  | <div> | |  | <h1>Create Post</h1> | |  | <form onSubmit={this.handleSubmit}> | |  | <input required type="text" ref={(input)=>this.getTitle = input} | |  | placeholder="Enter Post Title"/> | |  | <br /><br /> | |  | <textarea required rows="5" ref={(input)=>this.getMessage = input} cols="28" | |  | placeholder="Enter Post" /> | |  | <br /><br /> | |  | <button>Post</button> | |  | </form> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default connect()(PostForm); | |

Remember that connect() gives you access to dispatch as a prop. Here once we have captured the data from the form we dispatch the action using this.props.dispatch() passing in the data object with a type of ‘ADD\_POST’.

Great, now we have added some data in our state but we can’t see any of those changes reflected in our application so let’s fix that. Before doing that let’s understand one more important thing about connect(). This special function provided by the react-redux library gives you access to dispatch whenever you call it wrapping the component-name as an argument to the function that is returned. We have seen this syntax which is as follows-

export default connect()(component-name)

Additionally, connect can do more. It can give you access to your state which is living inside your store object. To get access to your state, we need to use a special function called **mapStateToProps**. This function does exactly what it is named, map the state from the store object to the props object in your components. Let’s define this mapStateToProps function-

const mapStateToProps = (state) => {

return {

posts: state

}

}

The argument to mapStateToProps is our application state. To understand this better, imagine whatever you pass inside the mapStateToProps argument is your state. Next question to ask is what is that state is it an array or an object or something else? Well that will depend on what you have defined it in your reducer. Since we have only one reducer which is the postReducer, we know that the state is an array.

Next we return an object with a key posts and the value is the state itself. The key that we use in mapStateToProps will be available to us as props inside the component.

With that in place let’s add this function as an argument to our connect. So inside crud-redux/src/AllPost.js make the following changes-

|  |
| --- |
| The argument to mapStateToProps is our application state. To understand this better, imagine whatever you pass inside the mapStateToProps argument is your state. Next question to ask is what is that state is it an array or an object or something else? Well that will depend on what you have defined it in your reducer. Since we have only one reducer which is the postReducer, we know that the state is an array.  Next we return an object with a key posts and the value is the state itself. The key that we use in mapStateToProps will be available to us as props inside the component.  With that in place let’s add this function as an argument to our connect. So inside crud-redux/src/AllPost.js make the following changes- |

Now to check what we have here, we can log this.props.posts like so

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | import { connect } from 'react-redux'; | |  |  | |  | class AllPost extends Component { | |  | render() { | |  | return ( | |  | <div> | |  | <h1>All Posts</h1> | |  | {console.log(this.props.posts)} | |  | </div> | |  | ); | |  | } | |  | } | |  |  | |  | const mapStateToProps = (state) => { | |  | return { | |  | posts: state | |  | } | |  | } | |  | export default connect(mapStateToProps)(AllPost); | |

To test this out enter some values in the title and the message fields and check your console.

Great so we have the post. All is left is to display it in the browser. To do that let’s create another component called Post. So under crud-redux/src create a new file and call it ‘Post.js’. Now head back to AllPost.js and make the following changes-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | import { connect } from 'react-redux'; | |  |  | |  | import Post from './Post'; | |  |  | |  | class AllPost extends Component { | |  | render() { | |  | return ( | |  | <div> | |  | <h1>All Posts</h1> | |  | {this.props.posts.map((post) => <Post key={post.id} post={post} />)} | |  | </div> | |  | ); | |  | } | |  | } | |  |  | |  | const mapStateToProps = (state) => { | |  | return { | |  | posts: state | |  | } | |  | } | |  | export default connect(mapStateToProps)(AllPost); | |

We have imported the Post component inside AllPost and used the Array.prototype.map function to loop over each of the individual posts inside this.props.posts and pass it down to the Post component with the key as post.id and the post itself. Inside crud-redux/src/Post.js add in the following-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | import React, { Component } from 'react'; | |  |  | |  | class Post extends Component { | |  | render() { | |  | return ( | |  | <div> | |  | <h2>{this.props.post.title}</h2> | |  | <p>{this.props.post.message}</p> | |  | </div> | |  | ); | |  | } | |  | } | |  | export default Post; | |

With that in place, enter some values in the title and the message fields and see if it is being displayed under All Posts like so-



We finally have the Posts.

If you have got this far, great you are finally done with the **C**and the **R**part of this CRUD application as now we can create posts and can read them as well.