Applause from you and 118 others

Pratik Chakravorty Following
Software Developer with a passion for learning new things.
Apr 2 · 16 min read

Redux-A CRUD Example

What we are aiming for

This article's goal is to explain the core concepts of Redux by making a CRUD application. Most applications are CRUD applications. If you don't know what CRUD really is, it's short for Create, Read, Update and Delete.

So this blog post will show you the steps to create an app where the user can create posts, read posts , edit posts and delete posts using React and Redux so let's get started. I believe that you have Nodejs installed on your system.

First let's spin up a new project so go to a directory where you will store this project and type the following in the terminal-

```
create-react-app crud-redux
```

The above command uses the create-react-app CLI tool to generate a react boilerplate project for us so that we don't have to configure any tooling. If that command fails for you or gives out an error make sure you have create-react-app installed on your machine. If not do the following and then run the above command.

```
npm install -g create-react-app
```

The above command will install create-react-app globally on your system so that you can use it anywhere you want.

Once that command finishes it's job you will have a new directory named 'crud-redux'. Change to this new directory. Now let's clean this up a little bit as we don't need some of the stuff for this project.

Type the following

```
cd src

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

Finally go back to the root directory of the project and open it up in your favorite text-editor.

Now let's install Redux and it's React bindings. So back in the terminal type the following-

```
npm install --save redux 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

```
1   import React from 'react';
2   import ReactDOM from 'react-dom';
3   import './index.css';
4   import App from './App';
5
6
```

and crud-redux/src/App.js

```
1   import React, { Component } from 'react';
2
3   class App extends Component {
4     render() {
5       return (
6       <div className="App">
7         <h1>Hello React!</h1>
8       </div>
```

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
```

Hello React!

React is Running

If you are getting errors please make sure you have made those changes that I have shown above in index.js and App.js.

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-

```
1    import React, { Component } from 'react';
2    import PostForm from './PostForm';
3    import AllPost from './AllPost';
4
5
6    class App extends Component {
7      render() {
8        return (
9        <div className="App">
10            <PostForm />
11            <AllPost />
```

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-

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

2

3    class PostForm extends Component {

4    render() {

5    return (

6    <div>

7      <h1>Create Post</h1>

8      <form>

9       <input required type="text" placeholder="Enter Post Tit

10      <textarea required rows="5" cols="28" placeholder="Ente

11      <button>Post</button>

12      </form>
```

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

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

2

3    class AllPost extends Component {

4      render() {

5       return (

6      <div>

7        <h1>All Posts</h1>

8      </div>

9       );
```

With that we have the following output in the browser.

Create Post

Enter Post Title

Enter Post

Post

All Posts

UI for Creating Post and Displaying Posts

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-

```
1    import React from 'react';
2    import ReactDOM from 'react-dom';
3    import './index.css';
4    import App from './App';
5
6    import { createStore } from 'redux';
7
8    const store = createStore();
```

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.

```
1    const postReducer = (state = [], action) => {

2

3    }

4    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.**
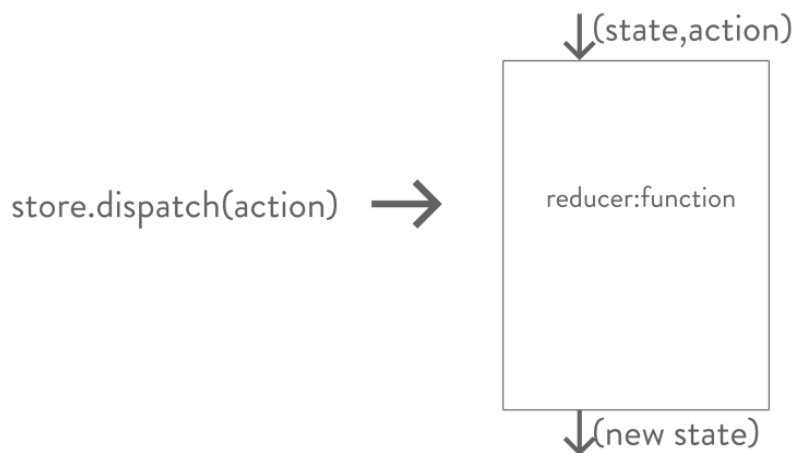
What Reducer really is

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.

```
1    const postReducer = (state = [], action) => {
2      switch(action.type) {
3        case 'ADD_POST':
4          return state.concat([action.data]);
5        default:
6          return state;
7      }
```

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.

```
1    import React from 'react';
2    import ReactDOM from 'react-dom';
3    import './index.css';
4    import App from './App';
5    import { createStore } from 'redux';
6
7    import postReducer from './reducers/postReducer';
```

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-

```
1    import React from 'react';
2    import ReactDOM from 'react-dom';
3    import './index.css';
4    import App from './App';
5    import { createStore } from 'redux';
6    import { Provider } from 'react-redux';
7
8
9    import postReducer from './reducers/postReducer';
10   const store = createStore(postReducer);
```

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.

```
1     import React, { Component } from 'react';
2
3     class PostForm extends Component {
4       handleSubmit = (e) => {
5         e.preventDefault();
6         const title = this.getTitle.value;
7         const message =  this.getMessage.value;
8         const data = {
9           id: new Date(),
10          title,
11          message
12        }
13      }
14    render() {
15    return (
16    <div>
17      <h1>Create Post</h1>
18      <form onSubmit={this.handleSubmit}>
19       <input required type="text" ref={(input)=>this.getTitle
20        placeholder="Enter Post Title"/>
```

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-

```
1     import React, { Component } from 'react';
2
3     class PostForm extends Component {
4       handleSubmit = (e) => {
5         e.preventDefault();
6         const title = this.getTitle.value;
7         const message =  this.getMessage.value;
8         const data = {
9           id: new Date(),
10          title,
11          message
12        }
13        console.log(data)
14      }
15    render() {
16    return (
17    <div>
18      <h1>Create Post</h1>
19      <form onSubmit={this.handleSubmit}>
20       <input required type="text" ref={(input)=>this.getTitle
21         placeholder="Enter Post Title"/>
```

▼ {id: Fri Mar 23 2018 00:08:18 GMT+0000 (GMT Standard Time), title: "First Post", message: "This is the first post."} 🛈
  ▶ id: Fri Mar 23 2018 00:08:18 GMT+0000 (GMT Standard Time) {}
    message: "This is the first post."
    title: "First Post"

console.log(data) output

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-

```
1    import React, { Component } from 'react';
2    import {connect} from 'react-redux';
3    class PostForm extends Component {
4      handleSubmit = (e) => {
5        e.preventDefault();
6        const title = this.getTitle.value;
7        const message =  this.getMessage.value;
8        const data = {
9          id: new Date(),
10         title,
11         message
12       }
13
14     }
15   render() {
16   return (
17   <div>
18     <h1>Create Post</h1>
19     <form onSubmit={this.handleSubmit}>
20      <input required type="text" ref={(input)=>this.getTitle
21        placeholder="Enter Post Title"/>
```

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

```
1    import React, { Component } from 'react';
2    import {connect} from 'react-redux';
3    class PostForm extends Component {
4      handleSubmit = (e) => {
5         e.preventDefault();
6         const title = this.getTitle.value;
7         const message =  this.getMessage.value;
8         const data = {
9           id: new Date(),
10          title,
11          message
12        }
13        this.props.dispatch({
14          type:'ADD_POST',
15          data});
16        this.getTitle.value = '';
17        this.getMessage.value = '';
18      }
19    render() {
20    return (
21    <div>
22      <h1>Create Post</h1>
23      <form onSubmit={this.handleSubmit}>
```

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-

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

 2

 3      import { connect } from 'react-redux';

 4

 5      class AllPost extends Component {

 6          render() {

 7              return (

 8                  <div>

 9                      <h1>All Posts</h1>

10                  </div>

11              );

12          }

13      }
```

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

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

 2

 3      import { connect } from 'react-redux';

 4

 5      class AllPost extends Component {

 6          render() {

 7              return (

 8                  <div>

 9                      <h1>All Posts</h1>

10                      {console.log(this.props.posts)}

11                  </div>

12              );

13          }

14      }
```

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

```
▼ [{…}] ⓘ
  ▶ 0: {id: Sat Mar 24 2018 14:35:41 GMT+0000 (GMT Standard Time), title: "First Post ", message: "This is the first post."}
    length: 1
  ▶ __proto__: Array(0)
```

we have the post!

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-

```
1    import React, { Component } from 'react';
2
3    import { connect } from 'react-redux';
4
5    import Post from './Post';
6
7    class AllPost extends Component {
8        render() {
9            return (
10                <div>
11                    <h1>All Posts</h1>
12                    {this.props.posts.map((post) => <Post key=
13                </div>
14            );
15        }
```

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-

```
1    import React, { Component } from 'react';
2
3    class Post extends Component {
4      render() {
5      return (
6        <div>
7            <h2>{this.props.post.title}</h2>
8            <p>{this.props.post.message}</p>
9        </div>
```

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-

Create Post

Enter Post Title

Enter Post

Post

All Posts

First Post

Learn Redux.

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.

Before diving into the update and the delete part of this application let's recap.

The entire state of our application lives inside an object called the store. In order to update the state we need to dispatch an action. Actions are nothing but Javascript objects with a type property which describes the event taking place. Events can be anything from updating counters to adding posts like we have seen above. Once the action has been dispatched, it is received by the reducer. The reducer takes in the current state of the application and the dispatched action and produces the next state of the application based on action.type.

For our React application to use the Redux store, we use the Provider component provided by the react-redux library and put it as the root of

all the components.

In order to access our Redux store within our React components we use the special connect() function. This function gives us access to dispatch and when we pass in mapStateToProps it gives us access to the state. mapStateToProps is a function that takes in the state of our application as a parameter and returns an object with keys of that object becoming the props of the component so that whenever we use this.props.key_name we get back the state we need.

With that out of the way, let's go back to the Post.js file and add in some buttons for Delete and Edit.

```
1    import React, { Component } from 'react';
2
3    class Post extends Component {
4      render() {
5      return (
6        <div>
7          <h2>{this.props.post.title}</h2>
8          <p>{this.props.post.message}</p>
9          <button>Edit</button>
10         <button>Delete</button>
```

Create Post

Enter Post Title

Enter Post

Post

All Posts

First Post

Learn Redux

Edit | Delete

Edit and Delete Buttons

Let's tackle the delete functionality first as it is easier. What we want to do is that whenever the user clicks the delete button it should remove the post. Now to do that we need to identify which post the user is deleting and we can do that with the post.id property that we included when we were adding the post earlier in PostForm component. So we need the following things, first we need an onClick handler so that when the user clicks the delete button we can do something. Then what we need is to dispatch an action of type say 'DELETE_POST'. We know pretty well how to get that going and that is by using connect.

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

2

3    import {connect} from 'react-redux';

4

5    class Post extends Component {
6      render() {
7      return (
8        <div>
9          <h2>{this.props.post.title}</h2>
10          <p>{this.props.post.message}</p>
11          <button>Edit</button>
12          <button
13          onClick={()=>this.props.dispatch({type:'DELETE_POST'
```

Here inside the onClick handler we have an arrow function that is invoked when the user clicks the delete button. Once they do, we dispatch an action of type 'DELETE_POST' and we also pass in the id of the given post.

To make this work we need to add this event in our reducers so let's go back to our reducers under crud-redux/src/reducers/postReducer.js and add in the following-

```
1    const postReducer = (state = [], action) => {
2      switch(action.type) {
3        case 'ADD_POST':
4          return state.concat([action.data]);
5        case 'DELETE_POST':
6          return state.filter((post)=>post.id !== action.id);
7        default:
8          return state;
```

Here we use Array.prototype.filter to remove the post with the id that matches action.id.

Now that is in place, go back to the app and try adding some data and then click the delete button. If the post goes away then great you have successfully implemented the **D** of this CRUD application. The only thing that is left is the update operation.

The last and the final CRUD operation is the Update operation and this one is a bit different. All the other operations we have seen are mostly one-click operations. You click on post, the post gets added,click on delete the post gets removed. This is not the case with Update. Because to update an existing post, first the user clicks on edit and then we provide them with a way in which they can change the post title and the message. Finally when they submit the changes, we perform the necessary updates and show it in the browser. So from this we know that the update operation is a two-step operation.

One approach in doing this is to use a boolean in our data object. This boolean will be false initially when the posts are added but when the user clicks on edit, we change it's value to true. If the value of this boolean is true then instead of rendering a regular Post component, we render a special EditComponent which will have a form with title and message fields. Once the user has made the necessary changes and hits update we go back to rendering the Post component but with the updated value. So let's do this-

Inside crud-redux/src/PostForm.js make the following changes-

```
 1   import React, { Component } from 'react';
 2   import {connect} from 'react-redux';
 3   class PostForm extends Component {
 4     handleSubmit = (e) => {
 5       e.preventDefault();
 6       const title = this.getTitle.value;
 7       const message =  this.getMessage.value;
 8       const data = {
 9         id: new Date(),
10         title,
11         message,
12         editing:false
13       }
14       this.props.dispatch({
15         type:'ADD_POST',
16         data});
17       this.getTitle.value = '';
18       this.getMessage.value = '';
19     }
20   render() {
21   return (
22   <div>
23     <h1>Create Post</h1>
24     <form onSubmit={this.handleSubmit}>
```

Here we have added an extra property called 'editing' and have set its value as false. Next create a file called EditComponent.js inside the src folder. Once that is done head over to crud-redux/src/AllPost.js and make the following changes-

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

 2

 3    import { connect } from 'react-redux';

 4

 5    import Post from './Post';

 6

 7    import EditComponent from './EditComponent';

 8

 9    class AllPost extends Component {

10        render() {

11            return (

12                <div>

13                    <h1>All Posts</h1>

14                    {this.props.posts.map((post) => (

15                        <div key={post.id}>

16                            {post.editing ? <EditComponent pos

17                                <Post key={post.id} post={post

18                        </div>

19                    ))}

20                </div>
```

All that this code is doing is that it is checking the value of editing in
each of the posts and if it is true then instead of rendering the Post
component it is rendering the EditComponent and passing it the post as
a prop.

Before going in and adding the Form in EditComponent, we need to
make one more change inside Post.js so go inside crud-
redux/src/Post.js and make the following change-

```
1    import React, { Component } from 'react';
2
3    import {connect} from 'react-redux';
4
5    class Post extends Component {
6      render() {
7      return (
8        <div>
9          <h2>{this.props.post.title}</h2>
10          <p>{this.props.post.message}</p>
11          <button
12           onClick={()=>this.props.dispatch({type:'EDIT_POST',
13           Edit</button>
14          <button
```

All we are doing is that when the user clicks the edit button we are
dispatching an action of type 'EDIT_POST' and also passing the id of
the post.

Since we have dispatched a new event, we need to make some changes
in postReducer.js so head over to this file and make the following
changes-

```
1    const postReducer = (state = [], action) => {
2      switch(action.type) {
3        case 'ADD_POST':
4          return state.concat([action.data]);
5        case 'DELETE_POST':
6          return state.filter((post)=>post.id !== action.id);
7        case 'EDIT_POST':
8          return state.map((post)=>post.id === action.id ? {..
9        default:
```

Here we are using Array.prototype.map to loop over each item and then
check the id of the post with the id that was passed in the action. If
there is a match then return a new object but change the value of
editing to true if it was false or vice-versa. If there is no match then just
return the object as it is.

Finally, let's head over to EditComponent.js and add in the following-

```
1    import React, { Component } from 'react';
2    import { connect } from 'react-redux';
3
4
5    class EditComponent extends Component {
6    handleEdit = (e) => {
7      e.preventDefault();
8      const newTitle = this.getTitle.value;
9      const newMessage = this.getMessage.value;
10     const data = {
11       newTitle,
12       newMessage
13     }
14     this.props.dispatch({ type: 'UPDATE', id: this.props.pos
15   }
16   render() {
17   return (
18   <div>
19     <form onSubmit={this.handleEdit}>
20       <input required type="text" ref={(input) => this.getTi
```

Here we are creating another Form which has an onSubmit handler. When the form is submitted, this.handleEdit function is invoked. This function takes in the event as a parameter. e.preventDefault() stops the page from refreshing. Then we are grabbing the data from the inputs using refs and putting it inside an object.Finally we are dispatching a new action with type property as 'UPDATE' . We are also passing in the id of the post that needs to be updated along with the updated data. Don't forget to use the connect function when dispatching actions.

Since we have added in a new event in our files, we need to make some changes in the reducer so head back to the reducer and make the following changes-

```
1    const postReducer = (state = [], action) => {
2      switch(action.type) {
3        case 'ADD_POST':
4          return state.concat([action.data]);
5        case 'DELETE_POST':
6          return state.filter((post)=>post.id !== action.id);
7        case 'EDIT_POST':
8          return state.map((post)=>post.id === action.id ? {..
9        case 'UPDATE':
10         return state.map((post)=>{
11           if(post.id === action.id) {
12             return {
13                 ...post,
14                 title:action.data.newTitle,
15                 message:action.data.newMessage,
16                 editing: !post.editing
```

In here all we are doing is using Array.prototype.map and looping over each posts and the post whose id matches the one with the id that was passed in the action we are returning a new object but with the updated values for title and message.Finally we are setting editing to false.

With that we are done, head over to the app, add some posts and hit the edit button.

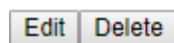## All Posts

First Post

Learn Redux

Update

And the Update Form appears!

When we hit the edit button the post changes to a form with the title and the message fields populated with the current values. Let's make a change and hit update.

Update Works!

Great, so our CRUD functionality is complete. Now let's add in some styles so that it looks good. I have written all the CSS for this application inside index.css just to keep things simple. Here is all the CSS code you need-

```css
1    body {
2    margin: 0;
3    padding: 0;
4    font-family: 'Work Sans',sans-serif;
5    background:#f4f4ef;
6    }
7
8
9    .center {
10   text-align: center;
11   }
12   a {
13   color:#636363;
14   text-decoration: none;
15   transition:all 0.5s ease-in;
16   }
17
18
19   a:hover {
20   text-decoration: underline;
21   }
22
23
24   /* Navigation */
25
26
27   .navbar {
28   background:#fff;
29   text-align:center;
30   padding:1.5rem 0;
31   width:100%;
32   display:flex;
33   justify-content: center;
34   align-items: center;
35   box-shadow:0 5px 15px 0 rgba(46, 61, 73, 0.12);
36   }
37
38
39   .navbar h2 {
40   margin:0;
41   font-weight:200;
```

```css
42    font-size:30px;
43    letter-spacing: 4px;
44    font-family:'Work Sans',sans-serif;
45    }
46
47
48    /* Post Styles */
49    .post-container {
50    background:#fff;
51    padding:50px;
52    width:50%;
53    margin:40px auto;
54    box-shadow:0 5px 15px 0 rgba(46, 61, 73, 0.12);
55    font-family:'Work Sans',sans-serif;
56    }
57
58
59    .post_heading {
60    text-align:center;
61    font-weight:400;
62    font-size:40px;
63    color:#636363;
64    }
65
66
67    .form {
68    display:flex;
69    flex-direction: column;
70    justify-content: center;
71    align-items: center;
72    }
73
74
75    .form input {
76    height:44px;
77    padding-left:15px;
78    padding-right:15px;
79    border: 1px solid #dbe2e8;
80    font-size:14px;
81    border-radius:2px;
82    color:#636363;
83    box-shadow: 0 2px 2px 0 rgba(46, 60, 73, 0.05);
```

```
 84        outline:none;
 85        width:80%;
 86        }
 87
 88
 89        .form textarea {
 90        width:80%;
 91        max-width:80%;
 92        padding-left:15px;
 93        padding-right:15px;
 94        padding-top:15px;
 95        border: 1px solid #dbe2e8;
 96        font-size:14px;
 97        border-radius:2px;
 98        color:#636363;
 99        font-family: 'Work Sans,',sans-serif;
100        box-shadow: 0 2px 2px 0 rgba(46, 60, 73, 0.05);
101        outline:none;
102        }
103
104
105        /* All Posts */
106
107
108        .all_post_heading {
109        text-align:center;
110        font-weight:400;
111        font-size:40px;
112        color:#636363;
113        word-wrap: break-word;
114        }
115
116
117        .post {
```

With all the CSS code, here are the final changes for each of the
components-

App.js

```
1    import React, { Component } from 'react';
2    import PostForm from './PostForm';
3    import AllPost from './AllPost';
4
5
6    class App extends Component {
7    render() {
8    return (
9    <div className="App">
10     <div className="navbar">
11       <h2 className="center ">Post It</h2>
12       </div>
13       <PostForm />
```

PostForm.js

```
1    import React, { Component } from 'react';
2    import { connect } from 'react-redux';
3    class PostForm extends Component {
4    handleSubmit = (e) => {
5    e.preventDefault();
6     const title = this.getTitle.value;
7     const message = this.getMessage.value;
8     const data = {
9      id: new Date(),
10      title,
11      message,
12      editing: false
13      }
14     this.props.dispatch({
15     type: 'ADD_POST',
16     data
17     })
18     this.getTitle.value = '';
19     this.getMessage.value = '';
20     }
21     render() {
22     return (
23     <div className="post-container">
24       <h1 className="post-heading">Create Post</h1>
```

Post.js

```
1    import React, { Component } from 'react';
2    import { connect } from 'react-redux';
3    class Post extends Component {
4    render() {
5    return (
6    <div className="post">
7      <h2 className="post_title">{this.props.post.title}</h2>
8      <p className="post_message">{this.props.post.message}</p
9      <div className="control-buttons">
10       <button className="edit"
11       onClick={() => this.props.dispatch({ type: 'EDIT_POST'
12       }
13       >Edit</button>
14       <button className="delete"
15       onClick={() => this.props.dispatch({ type: 'DELETE POS
```

AllPost.js

```
1    import React, { Component } from 'react';
2    import { connect } from 'react-redux';
3    import Post from './Post';
4    import EditComponent from './EditComponent';
5    class AllPost extends Component {
6    render() {
7    return (
8    <div>
9      <h1 className="post_heading">All Posts</h1>
10     {this.props.posts.map((post) => (
11     <div key={post.id}>
12       {post.editing ? <EditComponent post={post} key={post.i
13       key={post.id} />}
14     </div>
15   ))}
16   </div>
17   );
```

EditComponent.js

```
1    import React, { Component } from 'react';
2    import { connect } from 'react-redux';
3
4
5    class EditComponent extends Component {
6    handleEdit = (e) => {
7      e.preventDefault();
8      const newTitle = this.getTitle.value;
9      const newMessage = this.getMessage.value;
10     const data = {
11       newTitle,
12       newMessage
13     }
14     this.props.dispatch({ type: 'UPDATE', id: this.props.pos
15     }
16     render() {
17     return (
18     <div key={this.props.post.id} className="post">
19       <form className="form" onSubmit={this.handleEdit}>
20         <input required type="text" ref={(input) => this.getTi
```
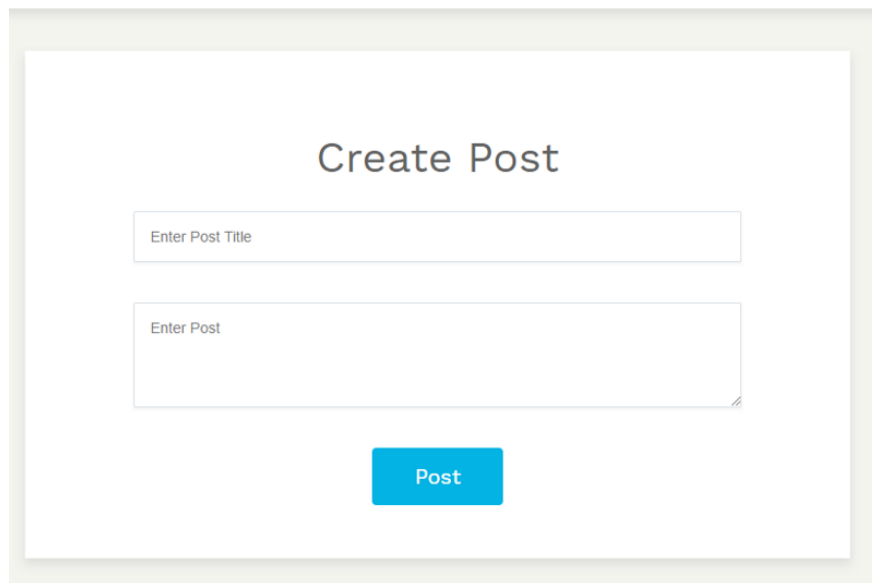
Finally the postReducer.js

```
1    const postReducer = (state = [], action) => {
2    switch (action.type) {
3    case 'ADD_POST':
4    return state.concat([action.data])
5    case 'DELETE_POST':
6    return state.filter((post) => post.id !== action.id)
7    case 'EDIT_POST':
8    return state.map((post) => post.id === action.id ? { ...po
9    case 'UPDATE':
10   return state.map((post) => {
11   if (post.id === action.id) {
12   return {
13   ...post,
14   title: action.data.newTitle,
15   message: action.data.newMessage,
16   editing: !post.editing
```
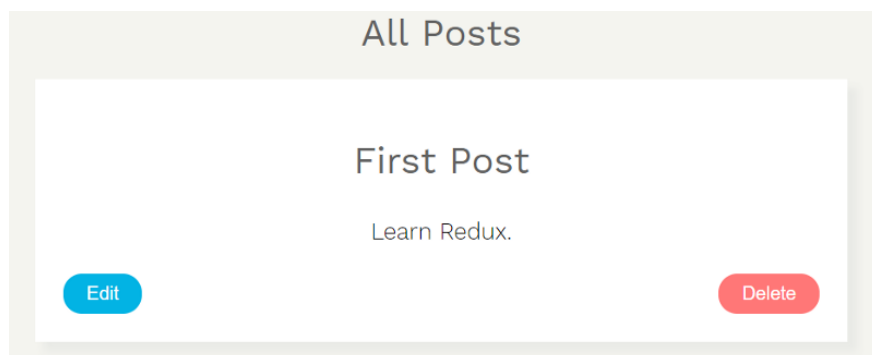
Post It

After Adding some CSS

After Adding some CSS

Here I have applied basic styles so that the app looks decent. You can modify it according to your needs. So if you have followed with me in this long blog post till the end then please congratulate yourself as now you know the core fundamentals of Redux. There are more advanced concepts that I did not go through like action-creators,middleware,combining multiple reducers and handling external API requests just to keep things simple. As always since this is my first blog post there can be some errors or mistakes so please don't hesitate to point that out. If you like to challenge yourself then please extend this application even further. Try adding a comments feature or an authentication system where only authorized users can view the posts and a user can modify only his/her own posts,add support for

upvotes and downvotes for each posts maybe add in some animations when you create a post or delete a post.There are so many things that you can do to improve this app.

Thanks for reading.

codeburst.io

✉ Subscribe to *CodeBurst's* once-weekly **Email Blast**, 🐦 Follow *CodeBurst* on **Twitter**, view 🗺**The 2018 Web Developer Roadmap**, and 🕸 **Learn Full Stack Web Development**.