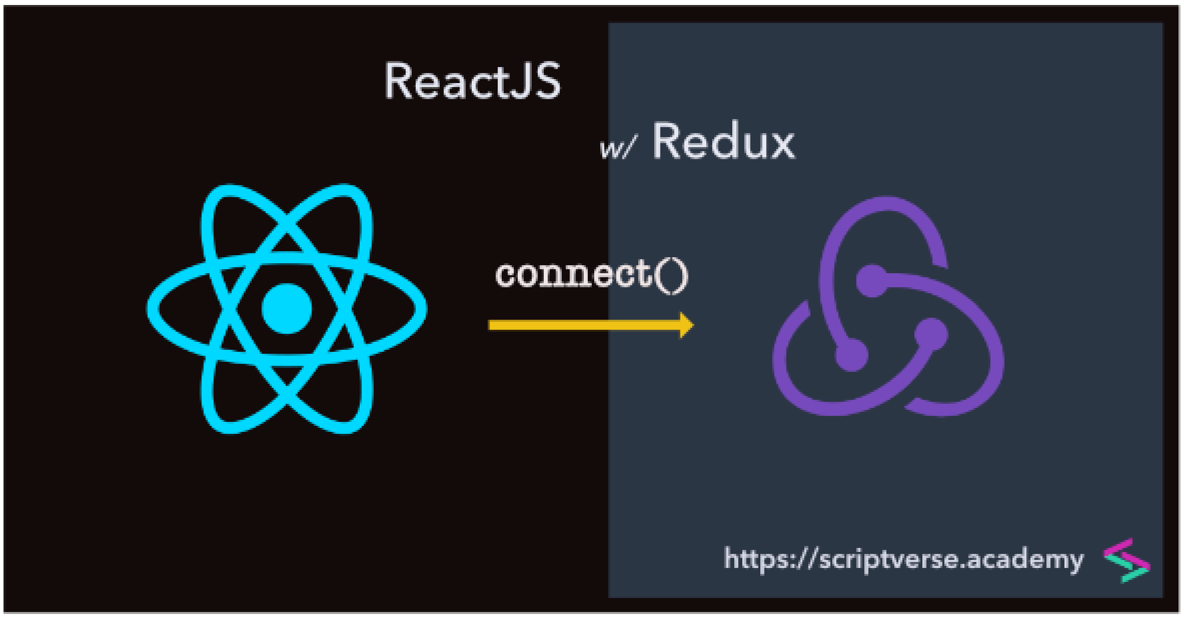
Redux w/ React/ReactJS: A Very Simple Example

Redux is a popular state management library. It is lightweight, with around just 2kB in size, and has great community support.

This tutorial is a simplifed step-by-step guide on how to set-up or use Redux with React using a simple example.



Before we start, we install the dependent packages that would be needed to build our React-Redux example code: `react-redux`, `redux` and `redux-thunk`

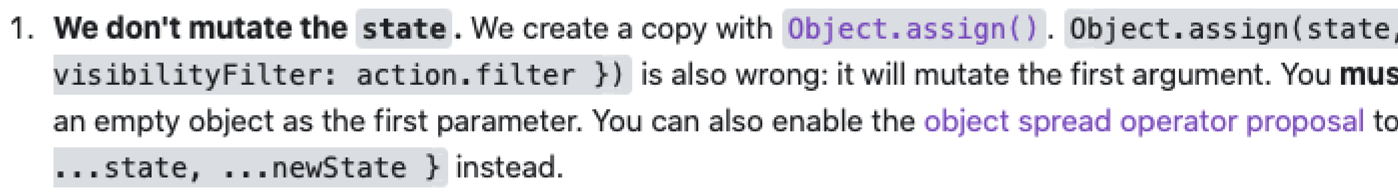
```
npm i react-redux
npm i redux
npm i redux-thunk
```

We first start with the Reducer. It is named that way because it is akin to the Array `reduce()` method, which takes in an array and returns a single condensed value. The first thing we do is setup the initial state. Here is where your application state is stored as a single object.

For our example, we introduce two state variables `name`, an (empty) string, and `books`, an (empty) array.

Just below the `INITIAL_STATE` is the reducer, which actually is a function that takes in the current state of the application and the associated action (we will come to that in the next section) as parameters and returns a new state. The body of a reducer usually consists of a `switch` statement for all cases of the action types you design.

Now we define one action type called `SET_DATA` which updates state variables (like `name` and `books` in our example here). But to make changes to an existing state, there is some rule to follow in Redux: **never mutate a state**. Instead, a copy of it with new/updated values should be created which is usually done using Object.assign(). But again, Redux gave an alternative to achieve this by using the Spread Operator. Here is a screenshot from the official site's page on Reducers:



1. **We don't mutate the `state`.** We create a copy with `Object.assign()`. `Object.assign(state, visibilityFilter: action.filter })` is also wrong: it will mutate the first argument. You **mus** an empty object as the first parameter. You can also enable the object spread operator proposal to `...state, ...newState }` instead.

So here is our `reducer.js`.

```
// reducer.js
const INITIAL_STATE = {
        name: '',
        books: []
}

export default (state = INITIAL_STATE, action={}) => {
        switch(action.type) {
                case "SET_DATA":
                        return {
                                ...state,
                                ...action.content
                        };
                default:
                        return state;
        }
};
```

And now we come to `action.js` where actions are defined. The only way the state in Redux store can be changed is by emitting an action. Actions are basically plain JavaScript objects with the `type` key. An action is sent to the store using the dispatch() method.

Here we define an action type `SET_DATA`.

```
// action.js
const setData = (content) => {
        return {
                type: "SET_DATA",
                content
        }
}

const appendData = (obj) => {
        return (dispatch) => {
                dispatch(setData(obj));
        }
}

export {
        appendData
}
```

Inside `store.js`, the reducer along with the `applyMiddleware()` function are passed as parameters to the `createStore()` method which creates a store to keep together the state tree of your application.

```
// store.js
```

```
import { createStore, applyMiddleware } from "redux";
import thunk from 'redux-thunk';
import reducer from "./reducer";

const store = createStore(
  reducer,
  applyMiddleware(thunk)
);

export default store;
```

Now comes the component, the "view" part of Redux. We create one called `<Books/>` which first updates the state inside the Redux store upon its load and renders these updated values from the store. This is achieved via the `connect()` function which is imported from `react-redux`. The `connect()` function connects the wrapped component to the Redux store. It takes two optional arguments: `mapStateToProps` and `mapDispatchToProps`.

The `mapStateToProps` argument specification implies that the component would be subscribed to the Redux store updates. The `mapDispatchToProps` argument dispatches actions and it can either be an object full of action creators (created in your `action.js`) or a function composed of functions using the `dispatch()` method.

For the sake of simplicity, we assign some hard-coded values inside `componentDidMount()`. In a real world React-Redux application, those values would either have been passed down from parent components as props or retrieved via axios or fetch(). The only important thing to note here is how the Redux state variables are updated. One is a string and the other is an array.

The `appendData()` method we defined in `action.js` is imported and used for the purpose.

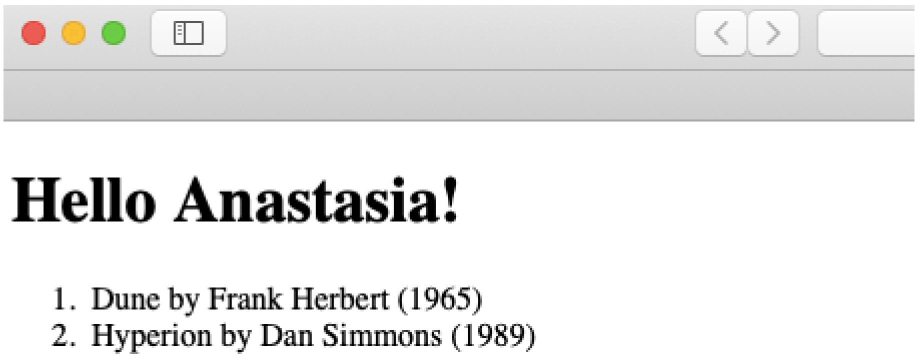
```
// Books.js
import React from "react";
import { connect } from "react-redux";
import { appendData } from "./action";

class Library extends React.Component {
  componentDidMount() {
        let name = 'Anastasia';
        let arr = [];

        arr.push({
                book_id: 1,
                title: 'Dune',
                author: 'Frank Herbert',
                year: 1965
        });

        arr.push({
                book_id: 2,
                title: 'Hyperion',
                author: 'Dan Simmons',
                year: 1989
        });

        this.props.appendData({
                name: name,
                books: [...this.props.books, ...arr]
        });
  }

  render() {
        const { name, books } = this.props;

    let booksList = books.length > 0
        && books.map((item, i) => {
      return (
        <li key={i} value={item.book_id}>
        {item.title} by {item.author} ({item.year})
        </li>
      )
    }, this);

          return (
            <div>
                <h1>Hello {name}!</h1>
                <ol>
                        { booksList }
                </ol>
            </div>
          );
        }
}

const mapDispatchToProps = {
        appendData
}

const mapStateToProps = state => ({
  name: state.name,
  books: state.books
});

export default connect(mapStateToProps, mapDispatchToProps)(Library);
```

And finally, we have the `index.js` where the passed component is wrapped over by the `<Provider/>` component which passes the `store` as props.

```
// index.js
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import Books from "./Books";
import store from "./store";

const App = () => {
  return (
    <div>
      <Books/>
    </div>
  );
}

const rootElement = document.getElementById("root");

ReactDOM.render(
  <Provider store={store}>
```

```
    <App/>
  </Provider>,
  rootElement
);
```

And below we have the rendered view of the component.



Hello Anastasia!

1. Dune by Frank Herbert (1965)
2. Hyperion by Dan Simmons (1989)