ReactJS + Redux

**ReactJS**

It’s a Javascript library that is used to create user interfaces. It’s good because it allows you to render parts of a website from the server which makes it faster on your own computer because you don’t need to do that much on your own computer. The basic building blocks of stuff in React are components.

You’ll need to call the ReactJS headers and add this line to initiate your fundamental class. Here it is element. The header files are:

import React from 'react';

import ReactDOM from 'react-dom';

If you writing in a different file, you’ll need to import that class using:

import Game from './Game';

You’ll need to export everything from that other file using this command:

export default Game;

To actually render everything, you’ll need to call this to your fundamental component.

ReactDOM.render(<Game />, document.getElementById('root'));

**JSX**

ReactJS is typicaly used with JSX, a syntax extension to Javascript which allows HTML to be embedded into your Javascript code. You need to use {} braces for Babel to use Javascript expressions into JSX HTML code, this allows you to use Javascript to change different components elsewhere.

If you have an empty tag, <img></img>, you write instead as <img /> which is also how you write components.

You can embed user input in JSX without a malignant user putting malware into your code because JSX converts everything into Strings before it used.

Elements in JSX

These are the objects that can contain headings, methods to run these objects, and values.

Creating an element is as simple as:

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

Or using the createElement function:

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

);

This is what it actually looks like:

// Note: this structure is simplified

const element = {

type: 'h1',

props: {

className: 'greeting',

children: 'Hello, world'

}

};

As I said before: elements are rendered onto the DOM using the render function within the reactDOM node.

const element = <h1>Hello, world</h1>;

ReactDOM.render(

element,

document.getElementById('root')

);

Elements are immutable, they can’t be changed at all after they’ve been made, so the only way to change an element is by creating a new element everytime and then calling the render function in reactDOM.

But most of the time, you only call the reactDOM.render() function only once in your app, and the way you do this is by using stateful components.

The really cool thing about React is that it only renders the changes you’ve made, so even if you call the whole element that contain heaps of children, React will look at your current element and compare it to what it was, and then only make the changes that you made.

Functional Components and Props

Are things that accept at least one property (**props**) and return an element. This element is defined by the user, using a props. Props is an object and is usually defined and sent to the functional component as another element. Functional components are named as such because they are actually functions, and differ from class components.

*Always start component names with a capital letter.*

*For example, <div /> represents a DOM tag, but <Welcome /> represents a component and requires Welcome to be in scope.*

Components are usually created to represent an entity like a button, screen, or form. Their output is often important, returning the element behind the component itself, and so components are often called from other components. E.g. A component Board calls multiple Button components, which render these multiple buttons on the board.

*Components must return a single root element. This is why we added a <div> to contain all the <Welcome /> elements.*

Components that have a lot of nesting can be written so it has components within the component of it extracted out of it. Kind of like taking cumbersome function, and making parts of the function into small functions.

*We recommend naming props from the component's own point of view rather than the context in which it is being used.*

The big rule behind React is that props must **NEVER change** inside a component (otherwise called ‘pure’). But of course websites are dynamic, so the way around this is by using states.

State and Lifecycle

Make a constructor function within your class component.

Class ClassName extends Component{

Constructor(props){ // props if you want to pass values into component from another component

Super(props);

this.state = {

stateName:stateValue

}

}

}

**Lifecycle Hook Methods:**

Methods of a class component that change based off component event.

componentDidMount() {

}

componentWillUnmount() {

}

The componentDidMount() hook runs after the component output has been rendered to the DOM.

While this.props is set up by React itself and this.state has a special meaning, you are free to add additional fields to the class manually if you need to store something that is not used for the visual output.

**If you don't use something in render(), it shouldn't be in the state.**

**Using State Correctly: 3 Things**

1. Do not modify State directly, use setState();
2. State may be set asynchronously, do not rely on their values to set values for other states.

To fix it, use a second form of setState() that accepts a function rather than an object. That function will receive the previous state as the first argument, and the props at the time the update is applied as the second argument:

// Correct

this.setState((prevState, props) => ({

counter: prevState.counter + props.increment

}));

1. State updates are merged, so you can do multiple state updates within one setState.

Data flows down.

So parent or child do not know if have states or not but states can be passed down to children as props.

**Handling Events**

onClick = {doSomethingFunction}

Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly.

function ActionLink() {

function handleClick(e) {

e.preventDefault();

console.log('The link was clicked.');

}

return (

<a href="#" onClick={handleClick}>

Click me

</a>

);

}

Here, e is a synthetic event.

If you want to use *this* in your event function: add this to constructor class

// This binding is necessary to make `this` work in the callback

this.handleClick = this.handleClick.bind(this);

Can use ES6 functions if you don’t want to use bind().

// This syntax ensures `this` is bound within handleClick.

// Warning: this is \*experimental\* syntax.

handleClick = () => {

console.log('this is:', this);

}

Or this:

// This syntax ensures `this` is bound within handleClick

return (

<button onClick={(e) => this.handleClick(e)}>

But if you pass this callback down to lower components, it may trigger double renderings.

**Conditional Rendering**

You can use if statements to render different things.

if (isLoggedIn) {

return <UserGreeting />;

}

return <GuestGreeting />;

}

You can do this within an expression in JSX by doing something like this: (equivalent to an if statement)

{unreadMessages.length > 0 &&

<h2>

You have {unreadMessages.length} unread messages.

</h2>

}

Because anything false that is && will equal false, and will not be rendered.

If else inline can easily be done using the ternary operator.

The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.

If you want to hide a component, return null.

Lists and Keys

**Basic Lists**

Best to do this inside a component using a map() function. You’ll need a key as well, because React needs to be able to reference each item in your array to know which one has been changed. Most often you would use IDs from your data as keys:

function NumberList(props) {

const numbers = props.numbers;

const listItems = numbers.map((number) =>

<li key={number.id}>{number}</li>

);

return (

<ul>{listItems}</ul>

);

}

const numbers = [1, 2, 3, 4, 5];

ReactDOM.render(

<NumberList numbers={numbers} />,

document.getElementById('root')

);

Keys serve as a hint to React but they don't get passed to your components. If you need the same value in your component, pass it explicitly as a prop with a different name.

Forms

**Controlled Components**

Forms typically have own internal state. You want to be able to have a submission function that has access to your data. So you want to combine internal state with your component states.

In your form element:

<form onSubmit={this.handleSubmit}>

<label>

Name:

<input type="text" value={this.state.value} onChange={this.handleChange} />

</label>

<input type="submit" value="Submit" />

</form>

With the handleChange function using the target value of the event. (What is event? What is the target value?)

handleChange(event) {

this.setState({value: event.target.value});

}

**TextArea**

In React, a <textarea> uses a value attribute instead. And everything else is the same.

**Select Tag**

In HTML, <select> creates a drop-down list. For example, this HTML creates a drop-down list of flavors.

Put this into your form, and you’ll be able to record your values.

<select value={this.state.value} onChange={this.handleChange}>

<option value="grapefruit">Grapefruit</option>

<option value="lime">Lime</option>

<option value="coconut">Coconut</option>

<option value="mango">Mango</option>

</select>

Lifting State Up

If you want to change state of parent in props, you can make event function ‘controlled’. E.g. pass the event function down as well.

There should be a single "source of truth" for any data that changes in a React application.

If something can be derived from either props or state, it probably shouldn't be in the state.

Composition and Inheritance

Sometimes, you don’t know what’s going to be inside a component, based on your parent component, so you can pass in your props like this.

function FancyBorder(props) {

return (

<div className={'FancyBorder FancyBorder-' + props.color}>

{props.children}

</div>

);

}

Where props.children fills in everything you need into FancyBorder.

function WelcomeDialog() {

return (

<FancyBorder color="blue">

<h1 className="Dialog-title">

Welcome

</h1>

<p className="Dialog-message">

Thank you for visiting our spacecraft!

</p>

</FancyBorder>

);

}

If you want more specialisation, you can call it another name other than children.

function SplitPane(props) {

return (

<div className="SplitPane">

<div className="SplitPane-left">

{props.left}

</div>

<div className="SplitPane-right">

{props.right}

</div>

</div>

);

}

function App() {

return (

<SplitPane

left={

<Contacts />

}

right={

<Chat />

} />

);

}

Specialisation

You can make a component do different things by having a component within another component.

**Redux**

**Actions**

export const ADD\_TODO = "ADD\_TODO"; // action is explicitly declared using string literal constants, not necessary but very helpful to do this

// action creators

// they simply are functions that return an action

export function addTodo(text){

return{

type: ADD\_TODO,

text

} // what does text do when it is like this? New ES6 syntax?

}

**Reducers**

// reducers actually list out what happens to the state in store when the actions are called.

The state is immutable, so the only way to modify the state is through Object.assign() functions. You use the action.type to switch between what action you want to choose.

function todoApp(state = initialState,action){ // ES6 voodoo magic that if state is undefined set to initial State

switch(action.type){ // this is a property of action, which returns an object containing type, index, text and all of the properties you set.

case SET\_VISIBILITY\_FILTER:

return Object.assign({}, state,{

visibilityFilter:action.filter

}) // this merges the empty object with state and the new action, which changes the visibilityFilter. The empty object first arg is the target

case ADD\_TODO:

return Object.assign({}, state, {

todos:[

...state.todos, // spread operator as state.todos will come as its own array

{

text:action.text,

completed:false

} // the todo items is an object that contains two properties, text and completed. The Object.assign has merged the new item with the existing

]

})

case TOGGLE\_TODO:

return Object.assign({}, state,{

todos:state.todos.map((todo,index)=>{ // map creates a new array and returns it, so there is no mutation

if (index === action.index){

return Object.assign({},todo,{ // this is the todo item object, not the list, which gets assigned into its own todo item

completed:!todo.completed

})

}

return todo // don't forget to return the todo item if it hasn't been changed

})

})

default:

return state // very important to return state if action.type is unknown

}

}

**Store**

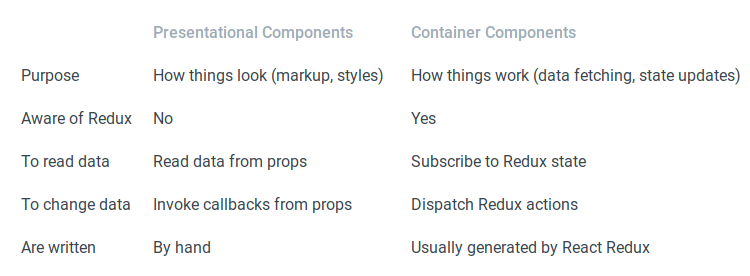
The store itself holds the application state, can be accessed using a getState() function, updated through dispatch(action) functions and can have listeners registered to it via subscriber(listener) functions.

**Usage with React**

Separate your components between **Presentational** and **Container** components.

Presentational components just present what their props gives to them. Concerns with how things look. May container both presentation and container components inside.

Container components are connected to the Redux store are concerned with how things work. These provide data and behaviour to presentational (or container) components. Typically, wrap around your Presentational components.

****

**Designing Presentational Components**

You then design how your app will look.

TodoList is a list showing visible todos.

* + todos: Array is an array of todo items with { id, text, completed } shape.
  + onTodoClick(id: number) is a callback to invoke when a todo is clicked.
* Todo is a single todo item.
  + text: string is the text to show.
  + completed: boolean is whether the todo should appear crossed out.
  + onClick() is a callback to invoke when the todo is clicked.
* Link is a link with a callback.
  + onClick() is a callback to invoke when the link is clicked.
* Footer is where we let the user change currently visible todos.
* App is the root component that renders everything else.

**Designing Container Components**

VisibleTodoList filters the todos according to the current visibility filter and renders a TodoList.

* FilterLink gets the current visibility filter and renders a Link.
  + filter: string is the visibility filter it represents.

**Designing Other Components** (the ones where you can’t tell where it should be presentational or container)

AddTodo is an input field with an “Add” button

**Implementing Container Components**

Rather than using store.subscribe to read part of Redux tree and supply props to component, use React-Redux’s **connect()** function which will help it avoid multiple unnecessary rerenders. This means that you don’t need to write your own performance optimisations.

To use connect(), need to use function called **mapStateToProps** which tells how to convert your Redux store state into props you want to pass into presentational component that you are wrapping.

# **Passing the Store**

All container components need access to the Redux store so they can subscribe to it. One option would be to pass it as a prop to every container component. However it gets tedious, as you have to wire store even through presentational components just because they happen to render a container deep in the component tree.

The option we recommend is to use a special React Redux component called [<Provider>](https://github.com/reduxjs/react-redux/blob/master/docs/api.md" \l "provider-store) to [magically](https://facebook.github.io/react/docs/context.html) make the store available to all container components in the application without passing it explicitly. You only need to use it once when you render the root component: