ReactJS + Redux

**ReactJS**

* Javascript library that is used to create user interfaces.
* Allows you to use JSX which is basically a better version of HTML in your Javascript code.
* Has a VirtualDOM that allows individual parts (components) to be refreshed when something needs updating. Saves reloading of entire page.

**Getting Started**

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 are 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’));

You won’t need to care too much about this last bit if you are using create-react-app that sets it all up for you already.

**JSX**

React uses a syntax extension of JavaScript called JSX that allows you to write HTML directly within JavaScript. This has several benefits.

* It lets you use the full programmatic power of JavaScript within HTML, and
* helps to keep your code readable.

However, because JSX is not valid JavaScript, JSX code must be compiled into JavaScript. The transpiler Babel is a popular tool for this process.

It's worth noting that under the hood the challenges are calling ReactDOM.render(JSX, document.getElementById('root')). This function call is what places your JSX into React's own lightweight representation of the DOM. React then uses snapshots of its own DOM to optimize updating only specific parts of the actual DOM.

**Using JSX**

Pretty much like HTML with your headings, paragraphs, and such e.g. <h1>Header</h1>. Look at documentation to see if HTML5 elements can be used as Javascript. There are a few examples where some things are different, look for that in the later sections.

To use Javascript within JSX, you simply include the code you want to be treated as JavaScript within curly braces: { 'this is treated as JavaScript code' }.

One important thing to know about nested JSX is that it must return a single element.

This one parent element would wrap all of the other levels of nested elements.

For instance, several JSX elements written as siblings with no parent wrapper element will not transpile.

JSX Comments: {/\* \*/}

**React Render**

Once you have created your React component, you have to render it to your DOM.

With React, we can render this JSX directly to the HTML DOM using React's rendering API known as ReactDOM.

ReactDOM offers a simple method to render React elements to the DOM which looks like this: ReactDOM.render(componentToRender, targetNode), where the first argument is the React element or component that you want to render, and the second argument is the DOM node that you want to render the component to.

As you would expect, ReactDOM.render() must be called after the JSX element declarations, just like how you must declare variables before using them.

**HTML className**

Now that you're getting comfortable writing JSX, you may be wondering how it differs from HTML.

So far, it may seem that HTML and JSX are exactly the same.

One key difference in JSX is that you can no longer use the word class to define HTML classes. This is because class is a reserved word in JavaScript. Instead, JSX uses className.

In fact, the naming convention for all HTML attributes and event references in JSX become camelCase. For example, a click event in JSX is onClick, instead of onclick. Likewise, onchange becomes onChange. While this is a subtle difference, it is an important one to keep in mind moving forward.

**Self Closing Tags**

In JSX, the rules are a little different. Any JSX element can be written with a self-closing tag, and every element must be closed. The line-break tag, for example, must always be written as <br /> in order to be valid JSX that can be transpiled. A <div>, on the other hand, can be written as <div /> or <div></div>. The difference is that in the first syntax version there is no way to include anything in the <div />. You will see in later challenges that this syntax is useful when rendering React components.

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

**Functional Component**

Components are the core of React. Everything in React is a component and here you will learn how to create one.

There are two ways to create a React component. The first way is to use a JavaScript function. Defining a component in this way creates a *stateless functional component*. The concept of state in an application will be covered in later challenges. For now, think of a stateless component as one that can receive data and render it, but does not manage or track changes to that data. (We'll cover the second way to create a React component in the next challenge.)

To create a component with a function, you simply write a JavaScript function that returns either JSX or null. One important thing to note is that React requires your function name to begin with a capital letter. Here's an example of a stateless functional component that assigns an HTML class in JSX:

// After being transpiled, the <div> will have a CSS class of 'customClass'  
const DemoComponent = function() {  
  return (  
    <div className='customClass' />  
  );  
};

Because a JSX component represents HTML, you could put several components together to create a more complex HTML page. This is one of the key advantages of the component architecture React provides. It allows you to compose your UI from many separate, isolated components. This makes it easier to build and maintain complex user interfaces.

**Class Component**

class Kitten extends React.Component {  
  constructor(props) {  
    super(props);  
  }  
  
  render() {  
    return (  
      <h1>Hi</h1>  
    );  
  }  
}

This creates an ES6 class Kitten which extends the React.Component class. So the Kitten class now has access to many useful React features, such as local state and lifecycle hooks. Don't worry if you aren't familiar with these terms yet, they will be covered in greater detail in later challenges.

Also notice the Kitten class has a constructor defined within it that calls super(). It uses super() to call the constructor of the parent class, in this case React.Component. The constructor is a special method used during the initialization of objects that are created with the class keyword. It is best practice to call a component's constructor with super, and pass props to both. This makes sure the component is initialized properly. For now, know that it is standard for this code to be included. Soon you will see other uses for the constructor as well as 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.

**Props (fcc explanation functional component)**

**props**. In React, you can pass props, or properties, to child components. Say you have an App component which renders a child component called Welcome that is a stateless functional component. You can pass Welcome a user property by writing:

<App>  
  <Welcome user='Mark' />  
</App>

You use **custom HTML attributes** that React provides support for to pass the property user to the component Welcome. Since Welcome is a stateless functional component, it has access to this value like so:

const Welcome = (props) => <h1>Hello, {props.user}!</h1>

It is standard to call this value props and when dealing with stateless functional components, you basically consider it as an argument to a function which returns JSX. You can access the value of the argument in the function body. With class components, you will see this is a little different.

**Pass Array as Props and Read Out as Props**

To pass an array to a JSX element, it must be treated as JavaScript and wrapped in curly braces.

<ParentComponent>  
  <ChildComponent colors={["green", "blue", "red"]} />  
</ParentComponent>

The child component then has access to the array property colors. Array methods such as join() can be used when accessing the property.

const ChildComponent = (props) => <p>{props.colors.join(', ')}</p>

This will join all colors array items into a comma separated string and produce:

<p>green, blue, red</p>

Later, we will learn about other common methods to render arrays of data in React.

**DefaultProps**

React also has an option to set default props. You can assign default props to a component as a property on the component itself and React assigns the default prop if necessary. This allows you to specify what a prop value should be if no value is explicitly provided. For example, if you declare MyComponent.defaultProps = { location: 'San Francisco' }, you have defined a location prop that's set to the string San Francisco, unless you specify otherwise. React assigns default props if props are undefined, but if you pass null as the value for a prop, it will remain null.

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

}

}

}

**Component Composition**

Now we will look at how we can compose multiple React components together. Imagine you are building an App and have created three components, a Navbar, Dashboard, and Footer.

To compose these compo props nents together, you could create an App *parent* component which renders each of these three components as *children*. To render a component as a child in a React component, you include the component name written as a custom HTML tag in the JSX. For example, in the render method you could write:

return (  
<App>  
  <Navbar />  
  <Dashboard />  
  <Footer />  
</App>  
)

When React encounters a custom HTML tag that references another component (a component name wrapped in < /> like in this example), it renders the markup for that component in the location of the tag. This should illustrate the parent/child relationship between the App component and the Navbar, Dashboard, and Footer.

Component composition is one of React's powerful features. When you work with React, it is important to start thinking about your user interface in terms of components like the App example in the last challenge. You break down your UI into its basic building blocks, and those pieces become the components. This helps to separate the code responsible for the UI from the code responsible for handling your application logic. It can greatly simplify the development and maintenance of complex projects.

Rendering ES6 style class components within other components is no different than rendering the simple components you used in the last few challenges. You can render JSX elements, stateless functional components, and ES6 class components within other components.

**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.

**FCC LIFECYCLE HOOKS**

componentDidMount()

Mostly to call an API endpoint to retrieve data. The componentDidMount() method is also the best place to attach any event listeners you need to add for specific functionality. React provides a **synthetic event** **system** which wraps the native event system present in browsers. This means that the synthetic event system behaves exactly the same regardless of the user's browser - even if the native events may behave differently between different browsers.

You've already been using some of these synthetic event handlers such as onClick(). React's synthetic event system is great to use for most interactions you'll manage on DOM elements. However, if you want to attach an event handler to the document or window objects, you have to do this directly.

componentWillReceiveProps(nextProps)

This is used to do stuff before component updates. Like a setState on other state values.

ComponentDidUpdate()

This is called after a component has rerendered.

componentWillMount()

This is when a component is about to be mounted (page is loaded).

ShouldComponentUpdate(nextProps, nextState)

For optimisation of component rerenders. Usually a component will rerender if it receives new props, even if the newProps are the same as the current props. So shouldComponentUpdate can block the component from needlessly rerendering.

componentWillUpdate()

This is called when component is actually going to update, it is different from componentWillReceiveProps in that it is also covers when a component has to perform a state change.

componentWillUnmount()

This is immediately called before component is unmounted, and is used to cancel lingering methods such as deleting network requests, ending timers, or deleting things that aren’t required anymore.

**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.

**Pass a Callback as Props**

You can also pass handler functions or any method that's defined on a React component to a child component. Just bind the handler in the parent component and pass it down as this.props.handleFunc() as props.

In the child component, just call it as onChange = {this.props.handleFunc)

**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. Note that keys only need to be unique between sibling elements, they don't need to be globally unique in your application.

**Filtering Lists**

Another method related to map is filter, which filters the contents of an array based on a condition, then returns a new array. For example, if you have an array of users that all have a property online which can be set to true or false, you can filter only those users that are online by writing:

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});

}

Need to add **event.preventDefault()** into handleSubmit to stop it defaulting and reloading webpage.

**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.

**ReactDOM.render()**

ReactDOM.render(componentToRender, targetNode). The first argument is the React component that you want to render. The second argument is the DOM node that you want to render that component within.

React components are passed into ReactDOM.render() a little differently than JSX elements. For JSX elements, you pass in the name of the element that you want to render. However, for React components, you need to use the same syntax as if you were rendering a nested component, for example ReactDOM.render(<ComponentToRender />, targetNode). You use this syntax for both ES6 class components and functional components. TargetNote is found using document.getElementById(“id\_name”);

**Inline Styles**

Different from HTML, you need to put all of your styles into a Javascript object which when converted into React needs to be read between curley brackets {}.

<div style={{color: "yellow", fontSize: 16}}>Mellow Yellow</div>

When you give only numbers, it is assumed to be in pxs. But if you want to add units then you need to make it a string.

**React Render Method Javascript**

You can also write JavaScript directly in your rendermethods, before the returnstatement, **without** inserting it inside of curly braces. This is because it is not yet within the JSX code. When you want to use a variable later in the JSX code inside the returnstatement, you place the variable name inside curly braces.

**&& for More Concise Conditional**

Used for JSX.

Imagine that you are tracking several conditions in a component and you want different elements to render depending on each of these conditions. If you write a lot of else ifstatements to return slightly different UIs, you may repeat code which leaves room for error. Instead, you can use the &&logical operator to perform conditional logic in a more concise way. This is possible because you want to check if a condition is true, and if it is, return some markup. Here's an example:

{condition && <p>markup</p>}

If the conditionis true, the markup will be returned. If the condition is false, the operation will immediately return falseafter evaluating the conditionand return nothing. You can include these statements directly in your JSX and string multiple conditions together by writing &&after each one. This allows you to handle more complex conditional logic in your render()method without repeating a lot of code.

**Ternary Expression for Conditional Rendering**

there's one last way to use built-in JavaScript conditionals to render what you want: the **ternary operator**. The ternary operator is often utilized as a shortcut for if/elsestatements in JavaScript. They're not quite as robust as traditional if/elsestatements, but they are very popular among React developers. One reason for this is because of how JSX is compiled, if/elsestatements can't be inserted directly into JSX code. You might have noticed this a couple challenges ago — when an if/elsestatement was required, it was always outside the returnstatement. Ternary expressions can be an excellent alternative if you want to implement conditional logic within your JSX. Recall that a ternary operator has three parts, but you can combine several ternary expressions together. Here's the basic syntax:

condition ? expressionIfTrue : expressionIfFalse

// Combine this with conditional rendering and list and keys

**Rendering to Server: renderToString()**

So far, you have been rendering React components on the client. Normally, this is what you will always do. However, there are some use cases where it makes sense to render a React component on the server. Since React is a JavaScript view library and you can run JavaScript on the server with Node, this is possible. In fact, React provides a renderToString() method you can use for this purpose.

There are two key reasons why rendering on the server may be used in a real world app. First, without doing this, your React apps would consist of a relatively empty HTML file and a large bundle of JavaScript when it's initially loaded to the browser. This may not be ideal for search engines that are trying to index the content of your pages so people can find you. If you render the initial HTML markup on the server and send this to the client, the initial page load contains all of the page's markup which can be crawled by search engines. Second, this creates a faster initial page load experience because the rendered HTML is smaller than the JavaScript code of the entire app. React will still be able to recognize your app and manage it after the initial load.

**Redux**

**Actions**

In Redux, all state updates are triggered by dispatching actions. An action is simply a JavaScript object that contains information about an action event that has occurred. The Redux store receives these action objects, then updates its state accordingly. Sometimes a Redux action also carries some data. For example, the action carries a username after a user logs in. While the data is optional, actions must carry a type property that specifies the 'type' of action that occurred.

Think of Redux actions as messengers that deliver information about events happening in your app to the Redux store. The store then conducts the business of updating state based on the action that occurred.

e.g.

const action = {

type: 'LOGIN'

}

A common practice when working with Redux is to assign action types as read-only constants, then reference these constants wherever they are used. You can refactor the code you're working with to write the action types as const declarations.

e.g. const LOGIN = “LOGIN”;

so action will be:

const action = {

type:LOGIN

}

**Action Creators**

After creating an action, the next step is sending the action to the Redux store so it can update its state. In Redux, you define action creators to accomplish this. An action creator is simply a JavaScript function that returns an action. In other words, action creators create objects that represent action events.

function actionCreator(){

return action;

}

**All Together**

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?

}

**Dispatch Method**

dispatch method is what you use to dispatch actions to the Redux store. Calling store.dispatch() and passing the value returned from an action creator sends an action back to the store.

Recall that action creators return an object with a type property that specifies the action that has occurred. Then the method dispatches an action object to the Redux store. Based on the previous challenge's example, the following lines are equivalent, and both dispatch the action of type LOGIN:

store.dispatch(actionCreator());  
store.dispatch({ type: 'LOGIN' });

**Reducers**

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

You can tell the Redux store how to handle multiple action types. Say you are managing user authentication in your Redux store. You want to have a state representation for when users are logged in and when they are logged out. You represent this with a single state object with the property authenticated. You also need action creators that create actions corresponding to user login and user logout, along with the action objects themselves.

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

}

}

**… Spread Operator on Arrays**

The Spread Operator (is a Javascript operator) that splits up an array into parts separated by commas. It is great for copying arrays over into a new array.

**Slice() and Concat()**

Used to enforce immutability in array operations.

**Object.assign()**

A useful tool for handling objects is the Object.assign() utility. Object.assign() takes a target object and source objects and maps properties from the source objects to the target object. Any matching properties are overwritten by properties in the source objects. This behavior is commonly used to make shallow copies of objects by passing an empty object as the first argument followed by the object(s) you want to copy. Here's an example:

const newObject = Object.assign({}, obj1, obj2);

This creates newObject as a new object, which contains the properties that currently exist in obj1 and obj2.

**CombineReducers()**

When the state of your app begins to grow more complex, it may be tempting to divide state into multiple pieces. Instead, remember the first principle of Redux: all app state is held in a single state object in the store. Therefore, Redux provides reducer composition as a solution for a complex state model. You define multiple reducers to handle different pieces of your application's state, then compose these reducers together into one root reducer. The root reducer is then passed into the Redux createStore() method.

In order to let us combine multiple reducers together, Redux provides the combineReducers() method. This method accepts an object as an argument in which you define properties which associate keys to specific reducer functions. The name you give to the keys will be used by Redux as the name for the associated piece of state.

Typically, it is a good practice to create a reducer for each piece of application state when they are distinct or unique in some way. For example, in a note-taking app with user authentication, one reducer could handle authentication while another handles the text and notes that the user is submitting. For such an application, we might write the combineReducers() method like this:

const rootReducer = Redux.combineReducers({  
  auth: authenticationReducer,  
  notes: notesReducer  
});

Now, the key notes will contain all of the state associated with our notes and handled by our notesReducer. This is how multiple reducers can be composed to manage more complex application state. In this example, the state held in the Redux store would then be a single object containing auth and notes properties.

**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.

**const store = createStore(todoApp, window.STATE\_FROM\_SERVER)**

Where the second argument is a way to get all the state from the server.

**Get State from Redux Store**: getState()

store.getState() allows you to grab the entire state of your redux container.

**Store Subscribe Listener:** store.subscribe(callbackfunction());

This is a function that is called whenever an action is dispatched again store. You can pass a callback function into this function to do something.

**Thunk Middleware**

At some point you'll need to call asynchronous endpoints in your Redux app, so how do you handle these types of requests? Redux provides middleware designed specifically for this purpose, called Redux Thunk middleware. Here's a brief description how to use this with Redux.

To include Redux Thunk middleware, you pass it as an argument to Redux.applyMiddleware(). This statement is then provided as a second optional parameter to the createStore() function. Take a look at the code at the bottom of the editor to see this. Then, to create an asynchronous action, you return a function in the action creator that takes dispatch as an argument. Within this function, you can dispatch actions and perform asynchronous requests.

const store = Redux.createStore(

asyncDataReducer,

Redux.applyMiddleware(ReduxThunk.default)

);

<https://www.npmjs.com/package/redux-thunk>

**Usage with React**

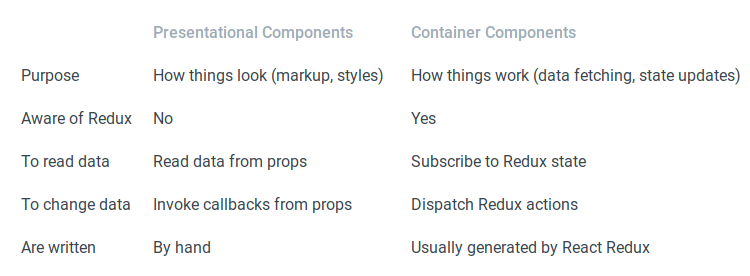
Typically, in a React Redux app, you create a single Redux store that manages the state of your entire app. Your React components subscribe to only the pieces of data in the store that are relevant to their role. Then, you dispatch actions directly from React components, which then trigger store updates.

Although React components can manage their own state locally, when you have a complex app, it's generally better to keep the app state in a single location with Redux. There are exceptions when individual components may have local state specific only to them. Finally, because Redux is not designed to work with React out of the box, you need to use the react-redux package. It provides a way for you to pass Redux state and dispatch to your React components as props.

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 RedReact Redux provides a small API with two key features: Provider and connect. Another challenge covers connect. The Provider is a wrapper component from React Redux that wraps your React app. This wrapper then allows you to access the Redux store and dispatch functions throughout your component tree. Provider takes two props, the Redux store and the child components of your app. Defining the Provider for an App component might look like this:ux store state into props you want to pass into presentational component that you are wrapping.

function mapStateToProps(state){

return {

messages:state

}

}

**MapDispatchToProps**

The mapDispatchToProps() function is used to provide specific action creators to your React components so they can dispatch actions against the Redux store. It's similar in structure to the mapStateToProps() function you wrote in the last challenge. It returns an object that maps dispatch actions to property names, which become component props. However, instead of returning a piece of state, each property returns a function that calls dispatch with an action creator and any relevant action data. You have access to this dispatch because it's passed in to mapDispatchToProps() as a parameter when you define the function, just like you passed state to mapStateToProps(). Behind the scenes, React Redux is using Redux's store.dispatch() to conduct these dispatches with mapDispatchToProps(). This is similar to how it uses store.subscribe() for components that are mapped to state.

For example, you have a loginUser() action creator that takes a username as an action payload. The object returned from mapDispatchToProps() for this action creator would look something like:

function mapDispatchToProps(dispatch){

return {

submitNewMessage:function(message){

dispatch(addMessage(message));

}

}

}

**Connect Function**

Now that you've written both the mapStateToProps() and the mapDispatchToProps() functions, you can use them to map state and dispatch to the props of one of your React components. The connect method from React Redux can handle this task. This method takes two optional arguments, mapStateToProps() and mapDispatchToProps(). They are optional because you may have a component that only needs access to state but doesn't need to dispatch any actions, or vice versa.

To use this method, pass in the functions as arguments, and immediately call the result with your component. This syntax is a little unusual and looks like:

const Container = connect(mapStateToProps, mapDispatchToProps)(MyComponent);

Note: If you want to omit one of the arguments to the connect method, you pass null in its place.

# 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 "_blank) 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:

Whole App FCC:

// Redux:

const ADD = 'ADD';

const addMessage = (message) => {

return {

type: ADD,

message: message

}

};

const messageReducer = (state = [], action) => {

switch (action.type) {

case ADD:

return [

...state,

action.message

];

default:

return state;

}

};

const store = Redux.createStore(messageReducer);

// React:

const Provider = ReactRedux.Provider;

const connect = ReactRedux.connect;

// Change code below this line

class Presentational extends React.Component {

constructor(props) {

super(props);

this.state = {

input: '',

}

this.handleChange = this.handleChange.bind(this);

this.submitMessage = this.submitMessage.bind(this);

}

handleChange(event) {

this.setState({

input: event.target.value

});

}

submitMessage() {

this.props.submitNewMessage(this.state.input);

this.setState({

input: '',

// messages: this.state.messages.concat(this.state.input)

});

}

render() {

return (

<div>

<h2>Type in a new Message:</h2>

<input

value={this.state.input}

onChange={this.handleChange}/><br/>

<button onClick={this.submitMessage}>Submit</button>

<ul>

{this.props.messages.map( (message, idx) => {

return (

<li key={idx}>{message}</li>

)

})

}

</ul>

</div>

);

}

};

// Change code above this line

const mapStateToProps = (state) => {

return {messages: state}

};

const mapDispatchToProps = (dispatch) => {

return {

submitNewMessage: (message) => {

dispatch(addMessage(message))

}

}

};

const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational);

class AppWrapper extends React.Component {

render() {

return (

<Provider store={store}>

<Container/>

</Provider>

);

}

};