**Part 02: React Fundamentals**

**Module 01: React Fundamentals**

**Lesson 01: Why React**

* Its compositional model
* Its declarative nature
* The way data flows through a Component

Composition occurs when *simple* functions are *combined* together to create *more complex* functions. Think of each function as a single building block that *does one thing* (DOT). When you combine these simple functions together to form a more complex function, this is **composition**.

With declarative code, we don't code up all of the steps to get us to the end result. Instead, we *declare* what we want done, and JavaScript will take care of doing it.

In React, the data flows from the parent component to a child component.

[**Lesson 02: Rendering UI with React**](file:///D:\Courses\React%20Nanodegree\Udacity%20-%20React%20nd019%20v2.0.0\Part%2002-Module%2001-Lesson%2002_Rendering%20UI%20with%20React\index.html)

We'll be looking at using React's .createElement() method in the next couple of videos. For starters, here is its signature:

React.createElement( /\* type \*/, /\* props \*/, /\* content \*/ );

* type – either a string or a React Component

This can be a string of any existing HTML element (e.g. 'p', 'span', or 'header') or you could pass a React *component*(we'll be creating components with JSX, in just a moment).

* props – either null or an object

This is an object of HTML attributes and custom data about the element.

* content – null, a string, a React Element, or a React Component

Anything that you pass here will be the content of the rendered element. This can include plain text, JavaScript code, other React elements, etc.

Since React's main focus is to streamline building our app's UI, there is only one method that is absolutely required in any React component class: render().

A great mindset to have when building React apps is to [think in components](https://facebook.github.io/react/docs/thinking-in-react.html). Components represent the modularity and reusability of React. You can think of your component classes as factories that produce instances of components. These component classes should follow the [single responsibility principle](https://en.wikipedia.org/wiki/Single_responsibility_principle) and just "do one thing". If it manages too many different tasks, it may be a good idea to decompose your component into smaller subcomponents.

Scaffolding Your React App

JSX is awesome, but it does need to be transpiled into regular JavaScript before reaching the browser. We typically use a transpiler like [Babel](https://github.com/babel/babel) to accomplish this for us. We can run Babel through a build tool, like [Webpack](https://github.com/webpack/webpack" \t "_blank) which helps bundle all of our assets (JavaScript files, CSS, images, etc.) for web projects.

To streamline these initial configurations, we can use Facebook's Create React App package to manage all the setup for us! This tool is incredibly helpful to get started in building a React app, as it sets up everything we need with *zero configuration*! Install Create React App (through the command-line with [npm](https://www.npmjs.com/get-npm" \t "_blank)), and then we can walk through what makes it so great.

*npm install -g create-react-app*

If you're seeing errors when trying to install a package globally, feel free to check out [this article](https://docs.npmjs.com/getting-started/fixing-npm-permissions) in the npm documentation. Note that to find out where global packages are installed, you can run npm list -g in your console.

Use this.props.<propertyName> - to access properties (component’s attribute)

**Lesson 03: State Management**

Three main components of React:

* Props – allow you to pass data into your components
* Functional Components – an alternative, and probably more intuitive approach to creating components
* Controlled Components – allow you to hook up the forms in your application to your component state

Creating component:

* import React, { Component } from 'react'
* class ListContacts extends Component {
* render() { return … }
* }
* export default ListContacts

If your component does not keep track of internal state (i.e., all it really has is just a render() method), you can declare the component as a Stateless Functional Component.

Earlier in this Lesson, we learned that props refer to attributes from parent components. In the end, props represent "read-only" data that are *immutable*.

A component's state, on the other hand, represents *mutable* data that ultimately affects what is rendered on the page. State is managed internally by the component itself and is meant to change over time, commonly due to user input (e.g., clicking on a button on the page).

To add state to a component all we need to do is to add state property to our class whose value is an object. This object represents the state of our component. Each key in object represents a distinct piece of state for this component. As with props we can access state property using this.state.<property>

When defining a component's initial state, avoid initializing that state with props. This is an error-prone *anti-pattern*, since state will only be initialized with props when the component is first created.

By having a component manage its own state, any time there are changes made to that state, React will know and *automatically*make the necessary updates to the page. The process of determining what has changed in the previous and new outputs is called Reconciliation.

You can use setState() to update component state. There are two ways to use it:

* Pass the function – first argument is previous state. The object returned from this function will be merged with the current state to from the new state of the component
* Pass in an object. This object will be merged with the current state to form the new state of the component

Use first when new state of the component depends on previous state. The end result will always be the same. The UI is just a function of your state.

PropTypes is a package that lets us define the data type we want to see right from the get-go and warn us during development if the prop that's passed to the component doesn't match what is expected.

To use PropTypes in our app, we need to install [prop-types](https://facebook.github.io/react/docs/typechecking-with-proptypes.html):

npm install --save prop-types

Alternatively, if you have been using [yarn](https://www.npmjs.com/package/yarn) to manage packages, feel free to use it as well to install:

*yarn add prop-types*

*Usage:*

Email.propTypes = {

text: // ???

};

Controlled components are components which render a form, but the source of truth for the form state lives inside of the component state rather than inside of the DOM. The reason they're called controlled components, is because React is controlling the state of the form.

value={this.state.query}

onChange={(event) => this.updateQuery(event.target.value)}

To recap how user input affects the ListContacts component's own state:

1. The user enters text into the input field.

* The onChange event listener invokes the updateQuery() function.
* updateQuery() then calls setState(), merging in the new state to update the component's internal state.
* Because its state has changed, the ListContacts component re-renders.

Let's see how we can leverage this updated state to filter our contacts. To help us with our filtering we'll need the following packages:

* [escape-string-regexp](https://www.npmjs.com/package/escape-string-regexp)
* [sort-by](https://www.npmjs.com/package/sort-by)

npm install --save escape-string-regexp sort-by

**Lesson 04: Lifecycle Events**

The render method used to be free of side effects. It shouldn’t make ajax requests to do anything that is async in nature. It should only receive props and return the description of the UI.

Lifecycle events are special methods each component can have that allow us to hook into the view when specific condition happen:

* **componentDidMount** – invoke immediately after the component is inserted into the DOM
* **componentWillUnmount** – invoke immediately before the component is removed from the DOM
* **getDerivedStateFromProps** – invoked when mounting or re-rendering the component

To use one of these, you'd just create a method in your component with the name and React will call it. It's an easy way to hook into different parts of the lifecycle of React components.

You'll sometimes see shouldComponentUpdate() in React apps as well. It returns true by default. This means that whenever a component's state (or its parent's state) is updated, the component re-renders.

The [React documentation](https://reactjs.org/docs/react-component.html#shouldcomponentupdate) provides the following guidance for using this lifecycle event:

* The default behavior is to re-render on every state change, and in the vast majority of cases you should rely on the default behavior.
* Do not rely on it to “prevent” a rendering, as this can lead to bugs.
* Consider using the built-in PureComponent instead of writing shouldComponentUpdate() by hand.
* We do not recommend doing deep equality checks or using JSON.stringify() in shouldComponentUpdate(). It is very inefficient and will harm performance.

**componentDidMount()** is invoked immediately after a component is mounted. Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request. Setting state in this method will trigger a re-rendering.

The following lifecycle events will be called in order when a component is being added to the DOM:

1. constructor()

* getDerivedStateFromProps()
* render()
* componentDidMount()

⚠️componentWillMount() has been deprecated. ⚠️

**Lesson 5: React Router**

React Router turns React projects into single-page applications. It does this by providing a number of specialized components that manage the creation of links, manage the app's URL, provide transitions when navigating between different URL locations, and so much more.

React Router is a collection of **navigational components** that compose declaratively with your application.

Install React Router

To use React Router in our app, we need to install [react-router-dom](https://www.npmjs.com/package/react-router-dom).

npm install --save react-router-dom

When you use BrowserRouter, what you're really doing is rendering a Router component and passing it a history prop. Wait, what is history? history comes from the [history](https://github.com/ReactTraining/history) library (also built by React Training). The whole purpose of this library is it abstracts away the differences in various environments and provides a minimal API that lets you manage the history stack, navigate, confirm navigation, and persist state between sessions.

So in a nutshell, when you use BrowserRouter, you're creating a history object which will listen to changes in the URL and make sure your app is made aware of those changes.

In summary, for React Router to work properly, you need to wrap your whole app in a BrowserRouter component. Also, BrowserRouter wraps the history library which makes it possible for your app to be made aware of changes in the URL.

As you've seen, Link is a straightforward way to provide declarative, accessible navigation around your application. By passing a to property to the Link component, you tell your app which path to route to.

<Link to="/about">About</Link>

If you're experienced with routing on the web, you'll know that sometimes our links need to be a little more complex than just a string. For example, you can pass along query parameters or link to specific parts of a page. What if you wanted to pass state to the new route? To account for these scenarios, instead of passing a string to Links to prop, you can pass it an object like this,

<Link to={{

pathname: '/courses',

search: '?sort=name',

hash: '#the-hash',

state: { fromDashboard: true }

}}>

Courses

</Link>

Route component if you want to be able to pass props to a specific component that the router is going to render, you'll need to use Route’s render prop. As you saw, render puts you in charge of rendering the component which in turn allows you to pass any props to the rendered component as you'd like.

In summary, the Route component is a critical piece of building an application with React Router because it's the component which is going to decide which components are rendered based on the current URL path.

## **Serialize The Form Data**

At this point, our form will serialize the values from user input (i.e., the name and email), adding them as a query string to the URL. We can add some additional functionality by having our app serialize these form fields on its own. After all, we want the app to ultimately handle creating the contact and saving it to the state.

To accomplish this, we'll use the [form-serialize](https://www.npmjs.com/package/form-serialize) package to output this information as a regular JavaScript object for the app to use.

npm install --save form-serialize

**Part 03 :** React & Redux

**Module 01:** React & Redux

**Lesson 01: Managing State**

Remember that the main goal of Redux is to make the state management of an application more predictable. One of the key points of Redux is that all of the data is stored in a single object called the *state tree*.

If we are going to build real application with our state tree, there are three ways in which we need to interface with it:

* Getting the state
* Listening for changes
* Updating the state

Then we combine the three items above and the state tree object itself into one unit which we called *the store*.

Only an event can change the state of the store.

When an event takes place in a Redux application, we use a plain JavaScript object to keep track of what the specific event was. This object is called an **Action**.

Let's take another look at an Action:

{

type: "ADD\_PRODUCT\_TO\_CART"

}

As you can see, an Action is clearly just a plain JavaScript object. What makes this plain JavaScript object special in Redux, is that every Action must have a *type* property. The purpose of the type property is to let our app (Redux) know exactly what event just took place. This Action tells us that a product was added to the cart. That's incredibly descriptive and quite helpful, isn't it?

Now, since an Action is just a regular object, we can include extra data about the event that took place:

{

type: "ADD\_PRODUCT\_TO\_CART",

productId: 17

}

In this Action, we're including the productId field. Now we know exactly which product was added to the store!

One more note to keep in mind as you build your Action objects: it's better practice to pass as little data as possible in each action. That is, prefer passing the index or ID of a product rather than the entire product object itself.

The function that returns the new state needs to be a pure function. Pure functions are defined by three characteristics:

* 1. They always return the same result if the same arguments are passed in
  2. Depends solely on the arguments passed in to them
  3. Does not produce side effects

The new dispatch() method is pretty small, but is vital to our functioning store code. To briefly recap how the method functions:

* dispatch() is called with an Action
* the reducer that was passed to createStore() is called with the current state tree and the action…this updates the state tree
* because the state has (potentially) changed, all listener functions that have been registered with the subscribe() method are called

We've finally finished creating the createStore function! Using the image above as a guide, let's break down what we've accomplished:

* we created a function called createStore() that returns a *store* object
* createStore() must be passed a "reducer" function when invoked
* the store object has three methods on it:
  + .getState() - used to get the current state from the store
  + .subscribe() - used to provide a listener function the store will call when the state changes
  + .dispatch() - used to make changes to the store's state
* the store object's methods have access to the state of the store via closure

**Lesson 2: UI + Redux**

combineReducers, under the hood, is our first look at reducer composition. combineReducers is responsible for invoking all the other reducers, passing them the portion of their state that they care about. We're making one root reducer, by composing a bunch of other reducers together.

**Lesson 3: Middleware**

Between the dispatching of an action and the reducer running, we can introduce code called **middleware** to intercept the action before the reducer is invoked. The [Redux docs](http://redux.js.org/docs/advanced/Middleware.html" \t "_blank) describe middleware as:

…a third-party extension point between dispatching an action, and the moment it reaches the reducer.

What's great about middleware is that once it receives the action, it can carry out a number of operations, including:

* producing a side effect (e.g., logging information about the store)
* processing the action itself (e.g., making an asynchronous HTTP request)
* redirecting the action (e.g., to another piece of middleware)
* dispatching supplementary actions

…or even some combination of the above! Middleware can do any of these before passing the action along to the reducer.

With Redux's middleware feature, we can run code *between* the call to store.dispatch() and reducer(). The reason this works, is because Redux's version of dispatch() is a bit more sophisticated than ours was, and because we provide the middleware functions when we create the store.

const store = Redux.createStore( <reducer-function>, <middleware-functions> )

Redux's createStore() method takes the reducer function as its first argument, but then it can take a second argument of the middleware functions to run. Because we set up the Redux store with knowledge of the middleware function, it runs the middleware function between store.dispatch() and the invocation of the reducer.

we can implement middleware into a Redux app by passing it in when creating the store. More specifically, we can pass in the applyMiddleware() function as an optional argument into createStore(). Here's applyMiddleware()'s signature:

applyMiddleware(...middlewares)

Note the spread operator on the middlewares parameter. This means that we can pass in as many different middleware as we want! Middleware is called in the order in which they were provided to applyMiddleware().

We currently have the checker middleware applied to our app, but we'll soon add a new logger middleware as well. To create a Redux store that uses our checker middleware, we can do the following:

const store = Redux.createStore(rootReducer, Redux.applyMiddleware(checker))

Redux middleware leverages a concept called **higher-order functions**. A higher-order function is a function that either:

* *accepts* a function as an argument
* *returns* a function

Higher-order functions are a powerful programming technique that allow functions to be significantly more dynamic.

**Lesson 4: Redux with React**

In this lesson, we're going to move away from our application being plain HTML and convert it to being powered by React. To do that, we'll need to add a number of libraries:

* [react](https://www.npmjs.com/package/react)
* [react-dom](https://www.npmjs.com/package/react-dom)
* [babel](https://www.npmjs.com/package/babel)

Here are the packages that we'll be adding in the next video:

<script src="https://unpkg.com/react@16.3.0-alpha.1/umd/react.development.js"></script>

<script src="https://unpkg.com/react-dom@16.3.0-alpha.1/umd/react-dom.development.js"></script>

<script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>

In order to save time, we used an uncontrolled component for our input field.

## **ref**

[Refs provide a way to access DOM nodes or React elements created in the render method.](https://reactjs.org/docs/refs-and-the-dom.html#callback-refs)

## **When to Use Refs**

The [docs](https://reactjs.org/docs/refs-and-the-dom.html#callback-refs) outline a few good use cases for refs:

* Managing focus, text selection, or media playback.
* Triggering imperative animations.
* Integrating with third-party DOM libraries.

Let's take a look at a similar example:

class Color extends React.Component {

alertTextInput = e => {

e.preventDefault();

alert(this.colorElement.value);

};

render() {

return (

<div>

<input

type="text"

placeholder="Add Input"

ref={(inputElement) => this.colorElement = inputElement}

/>

<button onClick={this.alertTextInput}>Alert Input</button>

</div>

);

}

}

In the line ref={(inputElement) => this.colorElement = inputElement}, inputElement is a reference to the input DOM element. We are storing a reference to the input DOM element in the colorElement instance property of the Color class.

Please note:

[React will call the ref callback with the DOM element when the component mounts, and call it with null when it unmounts. Refs are guaranteed to be up-to-date before componentDidMount or componentDidUpdate fires.](file:///D:\Courses\React%20Nanodegree\Udacity%20-%20React%20nd019%20v2.0.0\Part%2003-Module%2001-Lesson%2004_Redux%20with%20React\(https:\reactjs.org\docs\refs-and-the-dom.html#callback-refs)

#### componentDidMount()

[componentDidMount() is invoked immediately after a component is mounted (inserted into the tree)…If you need to load data from a remote endpoint, this is a good place to instantiate the network request.](https://reactjs.org/docs/react-component.html#componentdidmount)

#### forceUpdate()

[By default, when your component’s state or props change, your component will re-render. If your render() method depends on some other data, you can tell React that the component needs re-rendering by calling forceUpdate().](https://reactjs.org/docs/react-component.html#forceupdate)

[Calling forceUpdate() will cause render() to be called on the component, skipping shouldComponentUpdate(). This will trigger the normal lifecycle methods for child components, including the shouldComponentUpdate() method of each child. React will still only update the DOM if the markup changes.](https://reactjs.org/docs/react-component.html#forceupdate)

**Lesson 05: Asynchronous Redux**

The reducer expects to receive an action object, but what if, instead of returning an object, we have our action creator return a function?

We could use some middleware to check if the returned action is either a function or an object. If the action is an object, then things will work as normal - it will call the reducer passing it the action. However, if the action is a function, it can invoke the function and pass it whatever information it needs (e.g. a reference to the dispatch() method). This function could do anything it needs to do, like making asynchronous network requests, and can then dispatch a *different* action (that returns a regular object) when its finished.

An action creator that returns a function might look something like this:

function asyncActionCreator (id) {

return (dispatch) => {

return API.fetchUser(id)

.then((user) => {

dispatch(addUser(user));

});

};

}

We'll be adding the [redux-thunk library](https://github.com/gaearon/redux-thunk" \t "_blank) in the following video, so you'll need this:

<script src="https://unpkg.com/redux-thunk@2.2.0/dist/redux-thunk.min.js"></script>

Remember middleware executes in the order it is listed in the applyMiddleware() function.

## **Benefits of Thunks**

Out of the box, the Redux store can only support the synchronous flow of data. Middleware like **thunk** helps support asynchronicityin a Redux application. You can think of thunk as a wrapper for the store’s dispatch() method; rather than returning action objects, we can use thunk action creators to dispatch functions (or even or Promises).

Without thunks, synchronous dispatches are the default. We could still make API calls from React components (e.g., using the componentDidMount() lifecycle method to make these requests) -- but using thunk middleware gives us a cleaner separation of concerns. Components don't need to handle what happens after an asynchronous call, since API logic is moved away from components to action creators. This also lends itself to greater predictability, since action creators will become the source of every change in state. With thunks, we can dispatch an action only when the server request is resolved!

 I encourage to read up on both of the other (popular) options.

* [Redux Promise](https://github.com/redux-utilities/redux-promise) - FSA-compliant promise middleware for Redux.
* [Redux Saga](https://github.com/redux-saga/redux-saga) - An alternative side effect model for Redux apps

**Lesson 6: React-Redux**

## **Context.Provider**

The Provider component is used in the upper level of the component tree; that is, the component from which the data to be passed is held. In our case, this was the App component. We passed the name data as the value of Provider's value prop:

class App extends React.Component {

render() {

const name = 'Tyler';

return (

<Context.Provider value={name}>

<Parent />

</Context.Provider>

);

}

}

Note that the Provider component simply wraps around the entire component to be rendered!

## **Context.Consumer**

On the receiving end (i.e., a component "under" the Provider in the component hierarchy), we use the Consumer component. In our example, we passed Consumer a function as a child. This function accepts a value and returns some JSX:

function Grandchild ({ name }) {

return (

<Context.Consumer>

{(name) => (

<div>

<h1>Grandchild</h1>

<h3>Name: {name}</h3>

</div>

)}

</Context.Consumer>

);

}

As a result, we were able to render the Grandchild component with the correct name data without ever having to pass that data down the entire component thread! That's a lot less code than the previous way we had to do it. So React's Context API provides a terse, approachable way to easily communicate information from one component to another.

Connected component note:

* A connected component is connected to the Redux store and is responsible for getting data from the store.
* A presentational component should not access the store. It should receive any information it needs as props and then just render a UI.

# **The react-redux Bindings**

Let's take a moment to recap the changes we've made to our app in this Lesson, because we've updated quite a bit!

Previously, we leveraged the standard redux library to build our app. This allowed us to create a Redux store with the createStore() function, giving us an API to listen (subscribe()), get updates (getState()), and make updates (dispatch()) to state. We then created our own Provider component to efficiently pass the store to components that needed it, as well as our own connect() function so that our components can access "slices" of state as props.

We can build a fully-functional React and Redux app without Provider or connect(), but since they greatly simplify how React components interact with the Redux store, the creators of redux have included them in the react-redux package!

## **Provider**

With react-redux, rather than creating and using our own Provider which looks like this:

const Context = React.createContext()

class Provider extends React.Component {

render () {

return (

<Context.Provider value={this.props.store}>

{this.props.children}

</Context.Provider>

);

}

}

ReactDOM.render(

<Provider store={store}>

<ConnectedApp />

</Provider>,

document.getElementById('app')

);

…we can simply use the Provider component defined by the react-redux package! This allows us to wrap our entire app with Provider, effectively passing the store to even the most deeply nested components.

ReactDOM.render(

<ReactRedux.Provider store={store}>

<ConnectedApp />

</ReactRedux.Provider>,

document.getElementById('app')

);

## **connect()**

Similarly, we can also leverage react-redux's connect() function right out of the box. connect() is a higher-order function that takes in two arguments (as well as a few [optional arguments](https://github.com/reactjs/react-redux/blob/master/docs/api.md#connectmapstatetoprops-mapdispatchtoprops-mergeprops-options)) and returns a function. Check out its signature below:

const buildConnectedComponent = connect(mapStateToProps, mapDispatchToProps);

What's vital to understand is that buildConnectedComponent is a function. buildConnectedComponent will take a regular (presentational) React component and return a new, "connected" component.

const ConnectedComponent = buildConnectedComponent(MyComponent);

ConnectedComponent renders MyComponent, passing it the props as defined by mapStateToProps and mapDispatchToPros.

We can avoid having the intermediary buildConnectedComponent variable and just call the functions back-to-back:

const ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)(MyComponent)

Notice the double set of parentheses!

**Lesson 7: Real World Redux**

## **A Guide for the Planning Stages of Your Project**

1. Identify What Each View Should Look Like

* Break Each View Into a Hierarchy of Components
* Determine What Events Happen in the App
* Determine What Data Lives in the Store

## **Use** [**NINJAMOCK**](https://ninjamock.com/)[**INVISION**](https://www.invisionapp.com/)[**MOCKUP.IO**](https://mockup.io/about/)

# **Reducers**

A [Reducer](https://redux.js.org/basics/reducers) describes how an application's state changes. You’ll often see the [Object Spread Operator](https://redux.js.org/recipes/using-object-spread-operator) (...) used inside of a reducer because a reducer **must return a new object** instead of mutating the old state. If you need a refresher on the spread operator, check out [this ES6 lesson](https://classroom.udacity.com/nanodegrees/nd019/parts/290ec447-6555-41bf-ac39-457220a09aae/modules/9c5b7af0-0943-4d6e-b672-520440885aba/lessons/42383e89-ac6a-491a-b7d0-198851287bbe/concepts/398d36e6-3393-4c50-b870-44a4dffb0ac4).

If you want to know why Redux requires immutability, check out the [Immutable Data Section of the docs:](https://redux.js.org/faq/immutable-data#why-is-immutability-required).

Reducers have the following signature:

(previousState, action) => newState

Redux applications have a single store. We have to pass the Root Reducer to our createStore() function in order for the store to know what pieces of state it should have. The point of creating a store is to allow components to be able to access it without having to pass the data down through multiple components.

The Provider component (which comes from the react-redux package) makes it possible for all components to access the store via the connect() function.

All middleware follows this currying pattern:

const logger = (store) => (next) => (action) => {

// ...

}

Use [the Babel Repl](http://babeljs.io/repl/#?babili=false&browsers=&build=&builtIns=false&code_lz=Q&debug=false&forceAllTransforms=false&shippedProposals=false&circleciRepo=&evaluate=true&fileSize=false&lineWrap=false&presets=latest%2Creact%2Cstage-2&prettier=false&targets=&version=6.26.0&envVersion=) if you want to see this code in ES5.

The variable logger is assigned to a function that takes the store as its argument. That function returns another function, which is passed next (which is the next middleware in line or the dispatch function). That other function return another function which is passed an action. Once inside that third function, we have access to store, next, and action.

It’s important to note that the value of the next parameter will be determined by the applyMiddleware function. Why? All middleware will be called in the order it is listed in that function. In our case, the next will be dispatch because logger is the last middleware listed in that function.

Here’s our middleware wiring:

export default applyMiddleware(

thunk,

logger

);

Each thing returned by an action creator - be it an action or a function - will go through our thunk middleware. This is the source code for the thunk middleware:

function createThunkMiddleware(extraArgument) {

return ({ dispatch, getState }) => next => action => {

if (typeof action === 'function') {

return action(dispatch, getState, extraArgument);

}

return next(action);

};

}

const thunk = createThunkMiddleware();

thunk.withExtraArgument = createThunkMiddleware;

export default thunk;

If the thunk middleware sees an *action*, that action will be sent to the next middleware in line - the logger middleware. If it sees a *function*, the thunk middleware will call that function. That function can contain side effects - such as API calls - and dispatch actions, simple Javascript objects. These dispatched actions will again go to all of the middleware. The thunk middleware will see that it’s a simple action and pass the action on to the next middleware, the logger.

Once inside the logger:

const logger = store => next => action => {

console.group(action.type);

console.log("The action:", action);

const returnValue = next(action);

console.log("The new state:", store.getState());

console.groupEnd();

return returnValue;

};

Using the connect() function upgrades a component to a container. Containers can read state from the store and dispatch actions.

In the Planning Stage, we also determined that the Dashboard Component will be a container since it will need access to the `tweets` part of the store in order to display the list of tweets.

To make a container, we need to make use the `connect()` function. Remember that the signature of the connect function looks like this:

connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])

These details about `mapStateToProps` and `mapDispatchToProps` are crucial:

> mapStateToProps - If this argument is specified, the new component will subscribe to Redux store updates. This means that any time the store is updated, mapStateToProps will be called. The results of mapStateToProps must be a plain object, which will be merged into the component’s props. If you don't want to subscribe to store updates, pass null or undefined in place of mapStateToProps.

> mapDispatchToProps - If an object is passed, each function inside it is assumed to be a Redux action creator. An object with the same function names, but with every action creator wrapped into a dispatch call so they may be invoked directly, will be merged into the component’s props.

> If a function is passed, it will be given dispatch as the first parameter. It’s up to you to return an object that somehow uses dispatch to bind action creators in your own way. (Tip: you may use the [bindActionCreators()](https://redux.js.org/api-reference/bindactioncreators) helper from Redux.)

Do you remember the Component Hierarchy we made in Step 2 of the Planning Stage? We said that the Tweet Component will be inside of the Dashboard Component. If the Dashboard Component knows the ID of the tweet that needs to be displayed, it can just pass that ID to the Tweet Component, which will render the tweet.

Remember that the signature of the `mapStateToProps` function is:

mapStateToProps(state, [ownProps])

* state is the state inside the store
* ownProps are the properties that have been passed to this component from a parent component

Since we only care about the tweets part of the store, we can use destructuring to pass the tweets part of the state in the store as the parameter to the mapStateToProps() function.

**C:\Users\AVGasanov\Desktop\untitled-diagram-21.png**

So this is what the Dashboard Component's mapStateToProps() function looks like:

function mapStateToProps( {tweets} ){

return { tweetIds: Object.keys(tweets) };

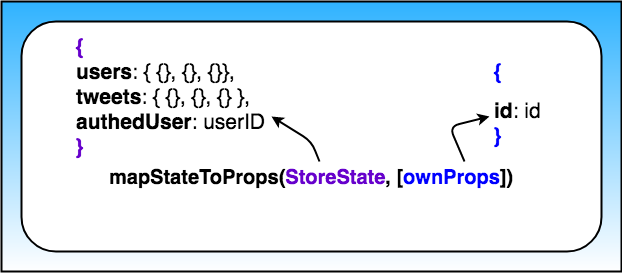
}

The important things to note are that:

* **tweets** is the slice of the state that this component cares about
* **tweetIds** will show up as a property on this container

Notice how we're passing an id prop along to the Tweet component:

* <Tweet id={id} />
* Because we're doing this, the mapStateToProps function's second argument (ownProps) will be an object that has an id property with this value.



So as of right now, this is what the mapStateToProps function looks like:

function mapStateToProps ({authedUser, users, tweets}, { id }) {

const tweet = tweets[id];

return {

authedUser,

tweet: formatTweet(tweet, users[tweet.author], authedUser)

};

}

The important thing to notice here is that mapStateToProps accepts two arguments:

* the state of the store
* the props passed to the Tweet component

We're destructuring both arguments. From the store, we're extracting:

* the authedUser data
* the users data
* the tweets data

Then we're getting the id from the props passed to the Tweets Component. We need both of these pieces of data (coming from the store's state and coming from the component) so that we can determine which Tweet should be displayed by Tweet Component.

So this is what the final state of the Tweet Component's mapStateToProps function looks like:

function mapStateToProps ({authedUser, users, tweets}, { id }) {

const tweet = tweets[id];

const parentTweet = tweet ? tweets[tweet.replyingTo] : null;

return {

authedUser,

tweet: tweet

? formatTweet(tweet, users[tweet.author], authedUser, parentTweet)

: null

};

}