The key idea is that you want your components to do one thing. Either present something visual (presentational component) OR fetch data (container component).

* Presentational components are usually stateless, contains DOM markup and optionally callback functions.
* Container components don’t have any DOM elements or styles, but they do Ajax calls, calls Redux Actions, or fetch data from Redux store.

The reason to split it up this way is the separation of concerns. It is easier to test and reuse components.

Start without it

You should **not** think about presentational/container components in the early stages of your application development process. At the start, you should only focus on writing regular React components and get things to work.

If you start planning for a presentational/container architecture before you have written any code, there is a big chance that one of the following will happen:

* You get confused and get stuck
* You will write unnecessarily complicated code because you will write presentation/container components that are not really needed.

When to use the pattern in a Redux app

When your application grows, you will reach a point where you will benefit from introducing the pattern. But when do you know when you have reached that point?

Let’s look at an example.

You can use the container/presentational pattern when using both regular Ajax calls inside your components, or when you are using Redux to fetch data. In this example, we are looking at using Redux to fetch data.

A new Redux app

When you start coding your Redux app, you usually start with a wrapper component that receives the data from the Redux store. Something like this:

const BlogPost = ({}) => (

<div>

<h2>First version of my redux webpage!</h2>

</div>

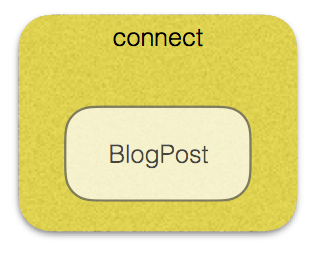
)

export default connect(mapStateToProps, mapDispatchToProps)(BlogPost)

connect is a Higher Order Component (HOC) that returns a new component. Learn more [connect](https://blog.jakoblind.no/learn-react-redux-by-coding-the-connect-function-yourself/) and [HOC](https://blog.jakoblind.no/simple-explanation-of-higher-order-components-hoc/) in my previous articles.

*This simple code actually uses the container/presentational pattern.* The PlogPost component is the presentational component and the component returned from connect is the container component.

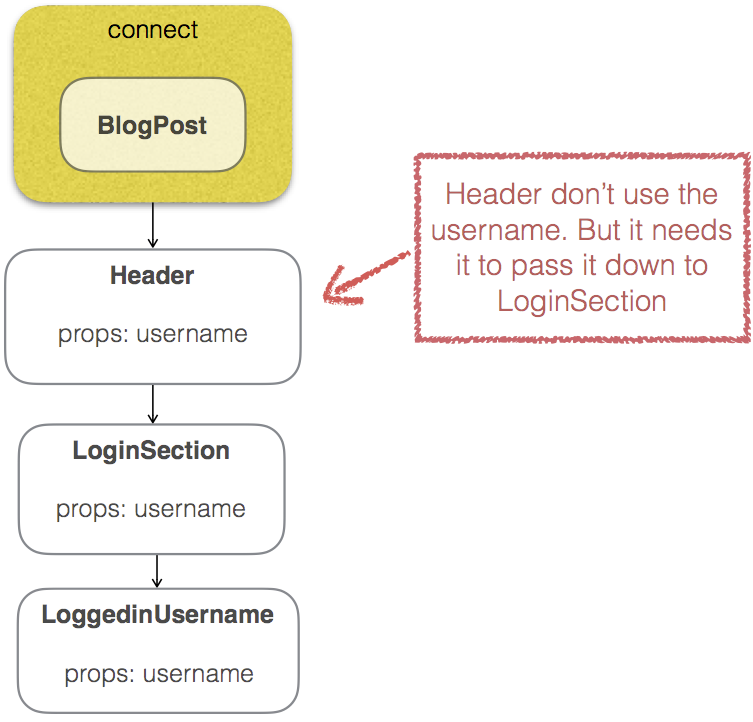
We can visualize it as a component graph like this:

[](https://d4jyn86spyrm8.cloudfront.net/graph0.png)

That’s a minimalistic graph. Let’s add some more stuff to it!

Add a feature (and some technical debt)

Now let’s say we implement a header where a user name should be visible. Complete code example here:

Our component graph now looks something like this: [](https://d4jyn86spyrm8.cloudfront.net/graphone.png)

We now have 3 nested components in our Page component. For LoggedinUsername to be able to access the username, we need to pass it down as props, through both Header and LoginSection.

That means that Header must know about the username when it shouldn’t need to. What if we decide to move LoginSection to the sidebar instead? Or if we want to delete it completely? It would be easy to forget to delete the username props passed into Header. Also every time we want to display more data in LoginSection (email for example) we need to update Header.

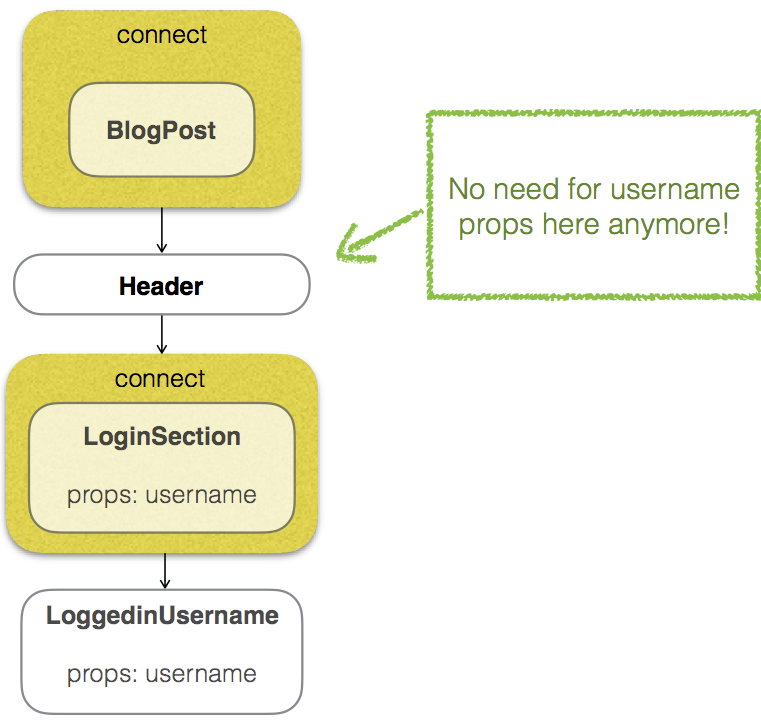
This is technical debt that we can resolve with the container/presentational pattern!

Introducing the container/presentational pattern!

Remember that I told you we already are using the container/presentational pattern with BlogPost and the connect function?

It is possible to connect Redux to any React component in your application.

So the way to solve our issue is to connect it to the component that needs the specific type of data. In our case, we connect directly to LoginSection

[](https://d4jyn86spyrm8.cloudfront.net/graphtwo.png)

LoginSection is our presentational component and the component returned from connect is our container component. Now we no longer need to pass username to Header

# Presentational and Container components

#### Problem

Data and logic together.

class Clock extends React.Component {

constructor(props) {

super(props);

this.state = {time: this.props.time};

this.\_update = this.\_updateTime.bind(this);

}

render() {

var time = this.\_formatTime(this.state.time);

return (

<h1>{ time.hours } : { time.minutes } : { time.seconds }</h1>

);

}

componentDidMount() {

this.\_interval = setInterval(this.\_update, 1000);

}

componentWillUnmount() {

clearInterval(this.\_interval);

}

\_formatTime(time) {

var [ hours, minutes, seconds ] = [

time.getHours(),

time.getMinutes(),

time.getSeconds()

].map(num => num < 10 ? '0' + num : num);

return {hours, minutes, seconds};

}

\_updateTime() {

this.setState({time: new Date(this.state.time.getTime() + 1000)});

}

}

ReactDOM.render(<Clock time={ new Date() }/>, ...);

#### Solution

Let's split the component into two parts - container and presentation.

#### Container Component

Containers know about data, it's shape and where it comes from. They know details about how the things work or the so called business logic. They receive information and format it so it is easy to use by the presentational component. Very often we use higher-order components to create containers. Their render method contains only the presentational component.

// Clock/index.js

import Clock from './Clock.jsx'; // <-- that's the presentational component

export default class ClockContainer extends React.Component {

constructor(props) {

super(props);

this.state = {time: props.time};

this.\_update = this.\_updateTime.bind(this);

}

render() {

return <Clock { ...this.\_extract(this.state.time) }/>;

}

componentDidMount() {

this.\_interval = setInterval(this.\_update, 1000);

}

componentWillUnmount() {

clearInterval(this.\_interval);

}

\_extract(time) {

return {

hours: time.getHours(),

minutes: time.getMinutes(),

seconds: time.getSeconds()

};

}

\_updateTime() {

this.setState({time: new Date(this.state.time.getTime() + 1000)});

}

};

#### Presentational component

Presentational components are concerned with how the things look. They have the additional markup needed for making the page pretty. Such components are not bound to anything and have no dependencies. Very often implemented as a stateless functional components they don't have internal state.

// Clock/Clock.jsx

export default function Clock(props) {

var [ hours, minutes, seconds ] = [

props.hours,

props.minutes,

props.seconds

].map(num => num < 10 ? '0' + num : num);

return <h1>{ hours } : { minutes } : { seconds }</h1>;

};

The nice things about containers is that they encapsulate logic and may inject data into different renderers. Very often a file that exports a container is not sending out a class directly but a function. For example, instead of using

import Clock from './Clock.jsx';

export default class ClockContainer extends React.Component {

render() {

return <Clock />;

}

}

We may export a function that accepts the presentational component:

export default function (Component) {

return class Container extends React.Component {

render() {

return <Component />;

}

}

}

Using this technique our container is really flexible in rendering its result. It will be really helpful if we want to switch from digital to analog clock representation.