Presentational and Container Components

Splitting React components into presentations and containers to manage state.

WE'LL COVER THE FOLLOWING

Every beginning is difficult. React is no exception and as beginners, we also have lots of questions. Where am I supposed to put my data, how to communicate changes or how to manage state? The answers to these questions are very often a matter of context and sometimes just practice and experience with the library. However, there is a pattern which is used widely and helps to organize React based applications - splitting the component into presentation and container.

Let's start with a simple example that illustrates the problem and then split the component into a container and presentation. We will use a `Clock` component. It accepts a Date object as a prop and displays the time in real time.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = { time: this.props.time };
    this._update = this._updateTime.bind(this);
  }
  render() {
    const 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() }/>, ...);
```

In the constructor of the component, we initialize the component's state which in our case is just the current `time` value. By using `setInterval` we update the state every second and the component is re-rendered. To make it look like a real clock we use two helper methods - `_formatTime` and `_updateTime`. The former extracts hours, minutes and seconds and makes sure that they are following the two-digit format. `_updateTime` mutates the current `time` object by one second.

The problems

There are a couple of things happening in our component. It looks like it has too many responsibilities.

- It mutates the state by itself. Changing the time inside the component may not be a good idea because then only `Clock` knows the current value. If there is another part of the system that depends on this data it will be difficult to share it.
- `_formatTime` is actually doing two things - it extracts the needed information from the date object and makes sure that the values are always presented by two digits. That's fine but it will be nice if the

extracting is not part of the function because then it is bound to the type

of the time object (coming as a prop), i.e., knows specifics about the shape of the data and at the same time deals with the visualization of it.

Extracting the container

Containers know about data, its 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 that it is easy to be used by the presentational component. Very often we use higher-order components to create containers because they provide a buffer space where we can insert custom logic.

Here's what our `ClockContainer` looks like:

```
// 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)
    });
  }
};
```

It still accepts `time` (a date object), does the `setInterval` loop and knows details about the data (`getHours`, `getMinutes` and `getSeconds`). In the end, it renders the presentational component and passes three numbers for hours, minutes and seconds. There is nothing about how the things look like. Only *business logic*.

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 stateless functional components they don't have internal state.

In our case the presentational component contains only the two-digits check and returns the `<h1>` tag:

```
// 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>;
};
```

Benefits

Splitting the components in containers and presentation increases the reusability of the components. Our `Clock` function/component may exist in an application that doesn't change the time or it's not working with JavaScript's Date object. That's because it is pretty *dummy*. No details about the data are required.

The containers encapsulate logic and we may use them together with different renderers because they don't leak information about the visual part. The approach that we took above is a good example of how the container doesn't care about how the things look like. We may easily switch from digital to an analog clock and the only one change will be to replace the `<Clock>` component in the `render` method.

Even the testing becomes easier because the components have fewer responsibilities. Containers are not concerned with UI. Presentational components are pure renderers and it is enough to run expectations on the resulted markup.

resulted markup.

Final thoughts

The concept of container and presentation is not new at all but it fits really nicely with React. It makes our applications better structured, easy to manage and scale. Given below is the complete working example of how containers and presentations achieve this:

```
import React from 'react';
import PropTypes from 'prop-types';

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>;
};
Clock.propTypes =
  {
  hours: PropTypes.number.isRequired,
  minutes: PropTypes.number.isRequired,
  seconds: PropTypes.number.isRequired
};
```

Quick Quiz on Presentational and Container Components!

Q

```
export default function Calendar(props) {
  var [ date, month, year ] = [
    props.date,
    props.month,
    props.year
  ]

  return <h1>{ day } : { month } : { year }</h1>;
};
```

The code above is an example of which of the following?

COMPLETED 0%

1 of 1