React And Separation Of Concerns

In this lesson, we will learn how React and its ecosystem have a separation of concerns despite markup, styles, and logic living in the same JavaScript land.

WE'LL COVER THE FOLLOWING

Years ago when Facebook announced their JSX syntax we had a wave of comments how this was against some of the well established good practices. The main point of most people was that it violates the separation of concerns. They said that React and its JSX are mixing HTML, CSS and JavaScript which were suppose to be separated.

In this article we will see how React and its ecosystem has quite good separation of concerns. We will prove that markup, styles and logic may live in the same JavaScript land and still be separated.

Styling

React components render to DOM elements. Nothing stops us to use the good old `class` attribute and attach a CSS class to the produced HTML element. The only difference is that the attribute is called `className` instead of `class`. The rest still works which means that if we want we may put our styles into external `.css` files. Following this approach we are not breaking the separation of concerns principle and still build a React app using JSX.

```
// assets/css/styles.css
.pageTitle {
  color: pink;
}
```

```
}

// assets/js/app.js
function PageTitle({ text }) {
  return <h1 className='pageTitle'>{ text }</h1>;
}
```

The “problem” became a problem when developers started talking about “CSS in JavaScript”. Back in 2014 this looked weird and wrong. However, the next couple of years showed that this is not that bad. Let’s take the following example:

```
function UserCard({ name, avatar }) {
  const cardStyles = {
    padding: '1em',
    boxShadow: '0px 0px 45px 0px #000'
  };
  const avatarStyles = {
    float: 'left',
    display: 'block',
    marginRight: '1em'
  };
  return (
    <div style={cardStyles}>
      <img src={avatar} width="50" style={avatarStyles} />
      <p>{name}</p>
    </div>
  );
}
```

The pain point is where we are mixing CSS and markup or we may say mixing styling and structure. To solve the issue let’s keep `UserCard` still responsible for the structure, but extract the styling out into dedicated components `Card` and `Avatar`:

```
function Card({ children }) {
  const styles = {
    padding: '1em',
    boxShadow: '0px 0px 45px 0px #000',
    maxWidth: '200px'
  };
  return <div style={styles}>{children}</div>;
}
function Avatar({ url }) {
  const styles = {
    float: 'left',
    display: 'block',
    marginRight: '1em'
  };
  return <img src={url} width="50" style={styles} />;
}
```

Then `UserCard` component becomes simpler and has no styling concerns:

```
function UserCard({ name, avatar }) {
  return (
    <Card>
      <Avatar url={avatar} />
      <p>{name}</p>
    </Card>
  );
}
```

So, as we can see, it is all a matter of composition. React even makes our applications more compact because everything is defined as a reusable component and lives in the same context - JavaScript.

There are bunch of libraries that help writing maintainable CSS in JavaScript (and to be more specific in React ecosystem) like Glamorous and styled-components. The result of such libraries is usually ready for use component that encapsulates the styling and renders a specific HTML tag.

Logic

Very often we write logic inside our React components which is more then clicking a button and showing a message. The snippet below demonstrates a component that fetches data from a fake API and renders users on the screen.

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      loading: false,
      users: null,
      error: null
    };
  }
  componentDidMount() {
    this.setState({ loading: true }, () => {
      fetch('https://jsonplaceholder.typicode.com/users')
        .then(response => response.json())
        .then(users => this.setState({ users, loading: false }))
        .catch(error => this.setState({ error, loading: false }));
    });
  }
  render() {
    const { loading, users, error } = this.state;

    if (isRequestInProgress) return <p>Loading</p>;
    if (error) return <p>Ops, sorry. No data loaded.</p>;
    if (users) return users.map(({ name }) => <p>{name}</p>);
```

```
    return null;
  }
}
```

Quite a lot of things are happening. When first rendered the component shows nothing - `null`. Then we get the life-cycle `componentDidMount` method fired where we set the `loading` flag to `true` and fire the API request. While the request is in flight we display a paragraph containing the text `"Loading"`. In the end, if everything is ok we turn `loading` to false and render list of user names. In case of error we display `"Ops, sorry. No data loaded"`.

Now you can say that the `App` component is kind of violating the separation of concerns. It contains data fetching and data representation. There are couple of ways to solve this problem one of which is FaCC (Function as Child Components). Let's write a `FetchUsers` component that will take care for the API request.

```
class FetchUsers extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      loading: false,
      users: null,
      error: null
    };
  }
  componentDidMount() {
    this.setState({ loading: true }, () => {
      fetch('https://jsonplaceholder.typicode.com/users')
        .then(response => response.json())
        .then(users => this.setState({ users, loading: false }))
        .catch(error => this.setState({ error, loading: false }));
    });
  }
  render() {
    const { loading, users, error } = this.state;

    return this.props.children({ loading, users, error });
  }
}
```

The very first thing that we notice is that the constructor and `componentDidMount` method are just copy-pasted from the `App` component. The difference is that we render nothing (no data representation) but call the `children` as a function passing the status/result of the request. Having `FetchUsers` we may transform our `App` into a stateless component:

```
function App() {
```

```
  return (
    <FetchUsers>
      {({ loading, users, error }) => {
        if (loading) return <p>Loading</p>;
        if (error) return <p>Ops, sorry. No data loaded.</p>;
        if (users) return users.map(({ name }) => <p>{name}</p>);
        return null;
      }}
    </FetchUsers>
  );
}
```

At this point, our markup is separated from the logic. We still operate with the same data and as a bonus, we have this reusable `FetchUsers` component that may be dropped anywhere.

Markup

JSX syntax is following the XML/HTML semantics and as such comes with a huge benefit - composability. You can say that React is one level up over the HTML because it allows us to group complex markup into a single component. For example we have a `<header>` with some `<h1>`, `<nav>` and `<p>` tags inside. We may easily create a `<Header>` component and put all those bits inside. We still keep them together but now they are easy to move around. Perhaps, there are places where we write some logic directly into the markup like so:

```
function CallToActionButton({ service, token }) {
  return <button onClick={ () => service.request(token) } />;
}

<CallToAction server={ service } token={ token } />
```

In such cases, it is recommended to use composition and remove any app logic concerns out of the presentation.

```
function CallToActionButton({ onButtonClicked }) {
  return <button onClick={ onButtonClicked } />;
}

<CallToAction server={ () => service.request(token) } />
```

It is just a matter of spotting those bits and changing them on time.

Conclusion

React is not against the separation of concerns. It is all about design and composition. There are patterns that help us to compose and logically separate our apps. We can still write well-organized programs with clearly defined responsibilities.