Reusable Components

This lesson explains how we can implement component hierarchies in React to achieve reusability.

Reusable and composable components empower you to come up with capable component hierarchies, the foundation of React's view layer. The last sections mentioned reusability, and now we can see how reusing the Table and Search components works in our case. Even the App component is reusable, as it can be instantiated elsewhere as well.

Let's define one more reusable component, a Button component which we'll eventually reuse often:

```
class Button extends Component {
  render() {
    const {
      onClick,
      className,
      children,
    } = this.props;

    return (
      <button
        onClick={onClick}
        className={className}
        type="button"
      >
        {children}
      </button>
    );
  }
}
```

It might seem redundant to declare components like this, but it's not. We use a `Button` component instead of a `button` element, which spares only the `type="button"`. It might not seem like a huge win, but these measures are about long-term, however. Imagine you have several buttons in your application, and you want to change an attribute, style, or behavior for just one. Without the component, you'd have to change (refactor) each one. The Button component ensures that the operation has a single source of truth, or one Button to refactor all the others at once.

Since you already have a button element, you can use the Button component instead. It omits the type attribute, because the Button component specifies it.

```
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        { list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <Button onClick={() => onDismiss(item.objectID)}>
                Dismiss
              </Button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

The Button component expects a `className` property in the `props`. The `className` attribute is another React derivate for the HTML attribute class. We didn't pass any `className` when the Button was used, though. It should be more explicit in our Button component that the `className` is optional, so we'll assign a default value in the object destructuring.

```
class Button extends Component {
  render() {
    const {
      onClick,
      className = '',
      children,
    } = this.props;

    ...
  }
}
```

Now, whenever there is no `className` property specified in the Button component, the value will be an empty string instead of `undefined`.

Further Reading:

- Read about how to pass props in React