Composition in React

When writing React components we may need to write some components that will not know the exact content they need to render ahead of time. This could be for layout components such as a sidebar or popup that wll keep the same behaviour and re-use different content.

All popups will share the same behaviour and layout but we may want to customise the content of the popup each time.

React gives us the ability to write `composed` components that accept dynamic content as `children`.

Components children

When rendering components we can use self closing tags or a closing tag, as we would in html. When using a closing tag we can pass any number of children to our components that they will have access to.

```
<Popup>
  <h2>Accept cookies?</h2>
  <button>Sure</button>
  <button>No way</button>
</Popup>

<Popup>
  <h2>Thanks for ordering</h2>
  <p>Your item is on it's way</p>
</Popup>
```

React will pass any child elements of our component on a prop called `children`. `props.children` is an array (if the component has more than one child) of elements. As we can render arrays in React we can use this prop inside the return statement of our component. Consider the following example that renders it's children wrapped in a div with specific styling.

```
const FancyBorder = (props) => {
  return <div className="fancy-border">{props.children}</div>;
};
```

This is not just limited to styling or layout as our `composed components` are regular components that can have state, side-effects or any other functionality that components normally would have. Consider the `Expandable` component below that adds a button to toggle showing or hiding its content.

```
const Expandable = ({ children }) => {
  const [isOpen, setIsOpen] = useState(false);

  const toggleOpen = () => setIsOpen((currOpen) => !currOpen);

  return (
    <div>
      <button onClick={toggleOpen}>{isOpen ? 'Close' : 'Open'}</button>
      {isOpen && children}
    </div>
  );
};
```

Custom context providers

When using Reacts context api we need to provide a value to be passed to that context. Extracting the logic of holding some state and passing it as a value is a great use case for composition.

Consider this Theme context example that keeps track of whether the app is in light or dark mode.

```
// src/contexts/Theme.js
import { createContext } from 'react';

export const ThemeContext = createContext();
```

```
// src/App.js
import { useState } from 'react';
import { ThemeContext } from './contexts/Theme';

const App = () => {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <div>{/* Rest of the app */}</div>
    </ThemeContext.Provider>
  );
};

export default App;
```

At the moment the state of `theme` is held in the `App` component. The `App` component may have to deal with other state or contexts so we can create a new component, `ThemeProvider` that will extract the theme logic using composition.

```
export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
```

This component will keep the state for our context and render it's children wrapped in a `Context.Provider` making the theme value available in all of those children.

This component can then be used to wrap our entire App, typically in `index.js`.

```
// src/contexts/Theme.js
import { createContext } from 'react';

export const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme((currTheme) => (currTheme === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, setTheme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

// src/App.js
const App = () => {
  return <div>{/* Rest of the app */}</div>;
};

// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
import { ThemeProvider } from './contexts/Theme';
import './index.css';

ReactDOM.render(
  <ThemeProvider>
    <App />
  </ThemeProvider>,
  document.getElementById('root')
);
```

This keeps our App independent of the context logic completely and nicely separates our concerns.