React Context and Other Considerations

Context

Sometimes there are situations where you need to quickly pass along data using props but this can sometimes result in awkward coding known as **prop drilling**.

Recall: props are only passed to a child component via the parent component. Since props are only passed in a parent → child direction, if a descendant component (a couple levels down) needs the prop data, you have to pass the prop along through every component in the tree until it reaches the component requiring the prop data. This is what is called prop drilling and it is not a very elegant way of passing along data.

You can use something called a Context to bypass this behaviour. Note that the more you use Context, the more difficult it is to make the component reusable because of dependencies so use it sparingly. Usually, you only need to bypass for global types of data (e.g. authenticated user data).

Let's try using a Context for country location (e.g. CA, US). This can be used for a shop site so that prices can be displayed using "CAD" or "USD" based on the selected country. **We're not going to worry about conversion rates for this example** though you can add this later by incorporating an API.

The code to choose currency is based on the following tutorial:
https://react.dev/reference/react/useContext

By default, Context is stateless, so if you want to be able to change the context value, you'll need to use a state. By using a state, changes in the context value will trigger component re-rendering.

/src/context/Context.js: (because we need an easy way to import the context)

```
import {createContext} from "react";

export const CurrencyContext = createContext();
```

- Create a Context named "CurrencyContext" using createContext(). There is no need to set a value since we'll set this value later.
- We're also exporting the "CurrencyContext" so it's available to any file which imports it.
- This is set up in a file so that other files can easily import the Context.

/src/App.jsx:

```
import Header from "./components/Header";
import ProductList from "./components/ProductList";
import Footer from "./components/Footer";
import {CurrencyContext} from "./context/CurrencyContext";

export default function App() {
  return (
    <CurrencyContext.Provider value="CAD">
      <div className="page">
        <Header />
```

```
        <main id="main">
          <ProductList />
        </main>
        <Footer />
      </div>
    </CurrencyContext.Provider>
  );
}
```

- As ***App.js*** is the main code, we're loading the Context and passing a default value to children components using `<`*[context]*`.Provider>` (this provides the context values to other components).
- Header and Footer just display a site name and copyright text, respectively.

/src/components/ProductList.jsx:

```
import Product from "./Product";

export default function ProductList() {
  let products = [
    {
      id: "1",
      name: "Product 1",
      price: "2.00"
    },
    {
      id: "2",
      name: "Product 2",
      price: "5.50"
    }
  ];
  return (
    <>
      <h1>Products</h1>
      {
        products.map((p) => (
          <Product key={p.id} name={p.name} price={p.price} />
        ))
      }
    </>
  );
}
```

- For simplicity's sake, we're creating a default list of products right inside the ProductList component. (Expanding on this, you could pull data from an API.)
- The ProductList component only lists Product components, passing the product values as props.

/src/components/Product.jsx:

```
import {useContext} from "react";
import {CurrencyContext} from "../context/CurrencyContext";

export default function Product(props) {
  const currency = useContext(CurrencyContext);
  return (
    <div>
      <h2>{props.name}</h2>
      <div>${props.price} {currency}</div>
    </div>
  )
}
```

- To use the `CurrencyContext` value (passed down from the provider in *App.jsx*), you need to import the Context (`CurrencyContext`) and the `useContext()` hook function.
- All this does is display the product data. The price is displayed with the currency code printed at the end. (For simplicity's sake we're not dealing with currency conversions though you can expand this to use an exchange rate API though you'd need to then store a default currency per product or assume a default base currency.)

Viewing the app in a browser, you should now see the currency code being displayed. Now that we have the basics working, we can add more to allow for user-changeable values via a CurrencyPicker component. By default, a Context is **stateless**. This means that, by default, a Context is something where you set the value once and you're done. If you want your Context to have a changeable value you can save and remember these changes by using a state variable.

We'll need to set up the state variable in *App.jsx* because to make this available to the whole app, you need to set things up in the root/main component.

Modify ***App.jsx*** as follows:

```
import {useState} from "react";
import Header from "./components/Header";
import ProductList from "./components/ProductList";
import Footer from "./components/Footer";
import {CurrencyContext} from "./context/CurrencyContext";

export default function App() {
  const [currency, setCurrency] = useState("CAD");

  return (
    <CurrencyContext.Provider value={{currency, setCurrency}}>
      <div className="page">
        <Header />
        <main id="main">
          <ProductList />
        </main>
```

```
        <Footer />
      </div>
    </CurrencyContext.Provider>
  );
}
```

- *App.jsx* has been modified to use a state. The state stores the currency code.
- The Context Provider value has been modified. You can pass more than just a value. You can also pass objects and functions. Notice the double curly brackets. The outer set of curly brackets indicates a script, while the inner set indicates a JS object containing a variable and a function (i.e. the state variable and its setter function).

/src/components/CurrencyPicker.jsx: a simple set of buttons to allow a user to select country (and therefore set currency)

```
import {useContext} from "react";
import { CurrencyContext } from "../context/CurrencyContext";
export default function CurrencyPicker() {
  const {setCurrency} = useContext(CurrencyContext);
  return (
    <div>
      <button onClick={() => setCurrency("CAD")}>CA</button>
      <button onClick={() => setCurrency("USD")}>US</button>
    </div>
  );
}
```

- In this case, we're using curly brackets to retrieve the `setCurrency()` function by name from the Context (recall that we passed an object as the Context value, consisting of `currency` and `setCurrency()`).
- On click, the buttons are just set to set the currency. Note that because we are passing a parameter to setCurrency(), we need to use an arrow function with empty parentheses. Once this is done, we can import the CurrencyPicker component to display in the Header component (or wherever you want).

/src/components/Header.jsx

```
import CurrencyPicker from "./CurrencyPicker";
export default function Header() {
  return (
    <header id="header">
      <h2 id="site-name">
        <a href="/">Test Shop</a>
      </h2>
      <CurrencyPicker />
    </header>
  );
}
```

- Notice that we don't need to import the context here because the Header component doesn't explicitly require it.

We need to now modify *Product.jsx* to reflect the change to the Provider value structure.

/src/components/Product.jsx

```
import {useContext} from "react";
import {CurrencyContext} from "../context/CurrencyContext";

export default function Product(props) {
  const {currency} = useContext(CurrencyContext);
  return (
    <div>
      <h2>{props.name}</h2>
      <div>${props.price} {currency}</div>
    </div>
  )
}
```

Other React considerations

Accessibility

No matter which website we're working on, accessibility should always be considered and a React web app is no exception. If you recall: one of the most important ways we can ensure our website code is accessible is to use **proper semantic HTML**. This means using the most appropriate element to describe our content and **NOT** hacking an element by adding scripts to make it act like a different element (the biggest offender here is usually making a DIV act like a button when just using BUTTON would have been more appropriate).

Proper semantic HTML is first and foremost. Although you can use WAI-ARIA to enhance the accessibility of your page, it is considered bad practice to rely entirely upon WAI-ARIA to do the heavy lifting. Additionally, ARIA properties should be used appropriately. As the W3C states in their documentation for WAI-ARIA authoring practices: "No ARIA is better than bad ARIA." This is because improper use of ARIA—just like improper semantic HTML—can badly affect how your content is interpreted and understood.

Fragments

You may recall that when we do a `return()` for a component, we must always return a single root element. This can be problematic when it comes to complex components where you may want to break down functionality.

For example, if you have a component which generates a list, you may have a parent component which generates a `<ul>` while a child component generates the multiple `<li>` elements. Let's say the child component (ListItems) returns something like:

```
<li>List item 1</li>
<li>List item 2</li>
<li>List item 3</li>
```

In the `return()` for the ListItems component, you may think to use a `<div>` just because you need a root element. This would be a mistake because you would end up with the following (not good):

```
<ul>
  <div>
    <li>List item 1</li>
    <li>List item 2</li>
    <li>List item 3</li>
  </div>
</ul>
```

The above is not good because it's invalid HTML (invalid syntax in **red**). Wherever there is invalid HTML or improper semantic HTML, there are accessibility issues because your content will not or may not be interpreted correctly. This is especially problematic for screen reader users but can sometimes affect keyboard users as well.

To deal with this type of situation where you need a placeholder element as a root element, you can use something called a Fragment. This identifies that your component contains a fragment of markup. The fragment will not result in extra markup upon render.

For example in your component's return, to use a fragment:

```
return(
  <>
    <li>List item 1</li>
    <li>List item 2</li>
    <li>List item 3</li>
  </>
);
```

The <> is short for <React.Fragment>. Either is fine to use if you need a fragment.

Folder Structure

As you build out your code more and more, you may start wondering about an approach to better organizing your files. This is a good read about this topic: https://www.robinwieruch.de/react-folder-structure/