React Context

CS568 – Web Application Development I

Assistant Professor Umur Inan
Computer Science Department
Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- React Context
- Before you use: Composition & IoC
- React Context API
  - createContext
  - Provider
  - contextType
  - Consumer
  - displayName
- Consuming multiple contexts
- Caveats
- React Context vs Redux

React Context

In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome. For example, passing locale preference to many components.

React Context provides a way to pass data through the component tree without having to pass props down manually at every level. In other words, Context is designed to share data that can be considered "**global**" for a tree of React components, such as the current authenticated user, theme, or preferred language.

The issue

In this example, we are passing down the avatar size from the Page component all way down to the Avatar component.

```
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... which renders ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... which renders ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

Before you use

- Do not overuse! Every time the context value changes, all consumer components will rerender. That causes performance issues.
- Use for simple data that do not change often.
- If your state is frequently updated, React Context may not be as effective or efficient as a tool like Redux.

Composition

- If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.
- Composition is for reusing the code.
- You can also use composition when some components don’t know their children ahead of time such as sidebar, dialog.
- Composition is basically passing down the component itself.

```jsx
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

```jsx
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

redundant to pass down the user and avatarSize props through many levels if in the end only the Avatar component really needs it. It's also annoying that whenever the Avatar component needs more props from the top, you have to add them at all the intermediate levels too.

```
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... which renders ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... which renders ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

One way to solve this issue without context is to pass down the Avatar component itself so that the intermediate components don't need to know about the user or avatarSize props.

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// Now, we have:
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
<PageLayout userLink={...} />
// ... which renders ...
<NavigationBar userLink={...} />
// ... which renders ...
{props.userLink}
```

React.createContext

```
const MyContext = React.createContext(defaultValue);
```

Creates a Context object. When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.

The defaultValue argument is used when a component does not have a matching Provider above it in the tree. This default value can be helpful for testing components in isolation without wrapping them.

Context.Provider

```
<MyContext.Provider value={/* some value */}>
```

Provider component allows consuming components to subscribe to context changes.

The Provider component accepts a value prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.

All consumers that are descendants of a Provider will re-render whenever the Provider’s value prop changes.

Class.contextType

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* render something based on the value */
  }
}
```

There is contextType property in class. With that, you can get the context value from the nearest the provider using this.context.

You can get only one context!

Context.Consumer

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

A React component that subscribes to context changes. Using this component lets you subscribe to a context within a function component.

Requires a function as a child. The function receives the current context value and returns a React node.

Context.displayName

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

Context object accepts a displayName string property. React DevTools uses this string to determine what to display for the context.

Consuming Multiple Contexts

```
// Code snipped of the provider
...
return (
  <ThemeContext.Provider value={theme}>
    <UserContext.Provider value={signedInUser}>
      <Layout />
    </UserContext.Provider>
  </ThemeContext.Provider>
);
...

// A component may consume multiple contexts
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

React Context vs Redux

- Redux is one of the most two popular state management tools in React.
- React Context and Redux both have the same goal, to manage global state so you don’t need to pass down props to many components.
- They both create global variables like window.myVar.
- The only difference is the way to access the global variable (state).
- If you use React Context properly and solve the re-rendering issues in child components, it will work just like Redux in big applications.
- Some developers don’t like creating multiple folders and files in Redux. Other developers don’t like writing producers and consumers in the component over and over again in React Context. So option is all up to you!