React Context

CS568 – Web Application Development I

Computer Science Department

Maharishi International University

Maharishi International University - Fairfield, Iowa

Content

- React Context
  - createContext
  - Provider
  - contextType
  - Consumer
- Composition
- LocalStorage

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 language preference down to many nested children 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.

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.

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.

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>
  );
}
```

Composition in React

- If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.
- Use composition more and context less.
- Composition is done by taking advantage of the children prop.
- It is powerful pattern that makes the code more reusable and flexible.
- Composition helps with props drilling and implementing master/details page. Master is static and details change. Master code is reused in child/details pages.

Before you use React Context

- Do not overuse! Every time the context value changes, all consumer components will rerender. That could cause 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.

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.

Caveats

Because context uses reference identity to determine when to re-render, there are some gotchas that could trigger **unintentional renders in consumers** when a provider's parent re-renders.

For example, the code below will re-render all consumers every time the Provider re-renders because a new object is always created for value.

```
class App extends React.Component {
  render() {
    return (
      <MyContext.Provider value={{something: 'something'}}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}
```

Caveats

To get around this, lift the value into the parent's state.

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},
    };
  }

  render() {
    return (
      <MyContext.Provider value={this.state.value}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}
```

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!

LocalStorage

If a page is refreshed, you will loose all global state in React Context or Redux. Instead, you can use LocalStorage as a global state holder that won’t loose data when page is refreshed.

The localStorage read-only property of the window interface allows you to access a Storage object for the Document's origin; the stored data is saved across browser sessions.

```
localStorage.setItem("myCat", "Tom");
const cat = localStorage.getItem("myCat");
localStorage.removeItem('myCat');
```