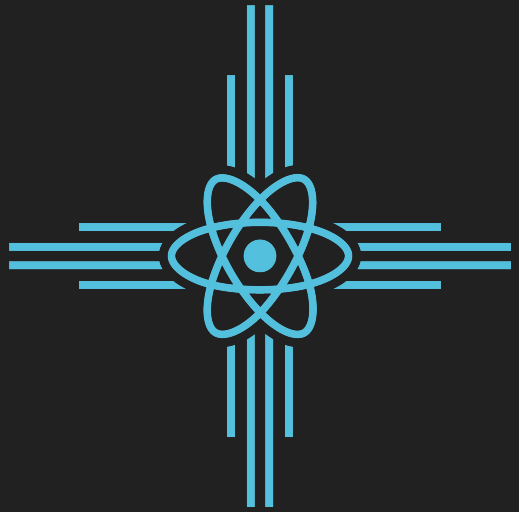
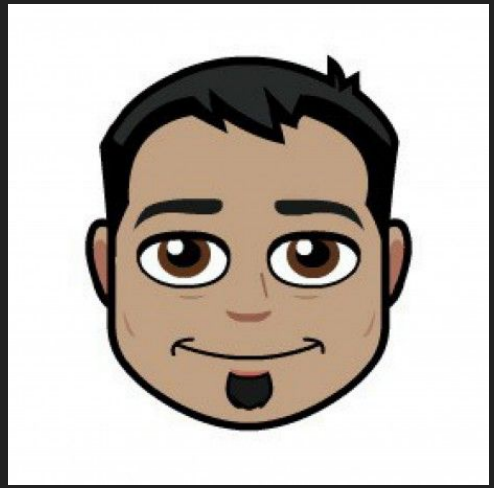


505 React

Meetup 4

About Me

- Enrique Delgado
- Works at Salesforce.com
- Based in Austin, TX
- Learning to play the Ukulele. Inspired by ‘The Moon Song’ as seen in the movie “Her”.

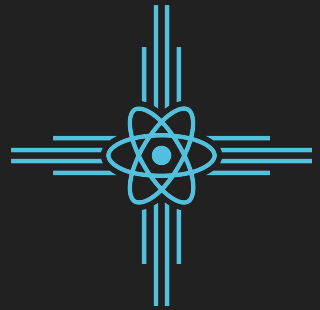


React Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

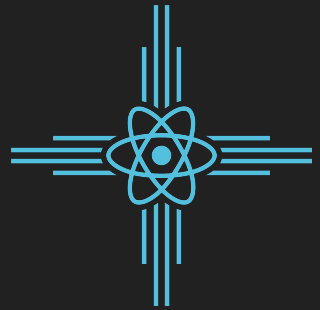
Context vs Redux

- Context is for information app-wide “global” information such as locale settings, current logged in user info, etc.
  - Context is primarily used when some data needs to be accessible by *many* components at different nesting levels.
- Not meant to be a big data store.
- Redux is great for larger data store needs such as API content caching and application state.



Pros and Cons

- Pros:
  - Avoid Prop Drilling (more on this next).
  - Simpler than Redux: No actions, reducers, mapping state to props etc.
- Cons:
  - Newer API, requires React 16.
  - Can make components less reusable as a component may expect a context provider not present in another app.



Prop Drilling

AKA “threading”. This is the main problem that the Context API is trying to solve.

- **`<Switch />`** itself does not need the **`on`** and **`onToggle`** values to function.

```
class Toggle extends React.Component {
  state = {on: false}
  toggle = () => this.setState(
    ({on}) => ({on: !on})
  )
  render() {
    return (
      <Switch
        on={this.state.on}
        onToggle={this.toggle}
      />
    )
  }
}
function Switch({on, onToggle}) {
  return (
    <div>
      <SwitchMessage on={on} />
      <SwitchButton onToggle={onToggle} />
    </div>
  )
}
function SwitchMessage({on}) {
  return (
    <div>
      The button is {on ? 'on' : 'off'}
    </div>
  )
}
function SwitchButton({onToggle}) {
  return (
    <button onClick={onToggle}>
      Toggle
    </button>
  )
}

```

Source: https://blog.kentcdodds.com/prop-drilling-bb62e02cb691

Context API Concepts

Provider

- Class or functional components:

```
class MyProvider extends React.Component {
  render() {
    return(
      <ThemeContext.Provider value={{age: 10}}>
        { this.props.children }
      </ThemeContext.Provider>
    );
  }
}

const MyProvider = () => (
  <ThemeContext.Provider value={{age: 10}}>
    { this.props.children }
  </ThemeContext.Provider>
)
```

Consumer

- Class components:

```
class MyComponent extends React.Component {
  static contextType = MyContext;
  render() {
    return(
      <p>Hello {this.context.username}</p>
    );
  }
}
```

- Functional components:

```
const MyComponent = () => (
  <MyContext.Consumer>
    {context => (
      <p>Hello {context.username}</p>
    )}
  </MyContext.Consumer>
)
```

Things to Consider

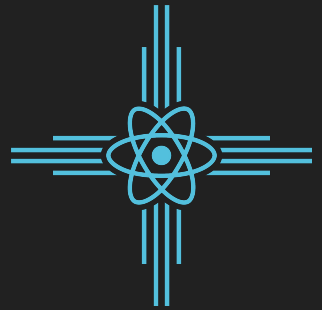
- Just as one can prematurely reach for Redux, one can do that with Context API
  - Prop drilling may be avoided by the use of component composition.
  - Make components more flexible by yielding `props.children`.
  - Use render props (props that are functions, which in turn return a react component).

Hands-on Exercise

- https://github.com/edelgado/react-505-context

Resources

- Context API docs: https://reactjs.org/docs/context.html
- Wes Bos tutorial: https://wesbos.com/react-context/
- Prop Drilling: https://blog.kentcdodds.com/prop-drilling-bb62e02cb691
- Next.js: https://nextjs.org/
- CSS in JS: https://github.com/zeit/styled-jsx



Thanks!