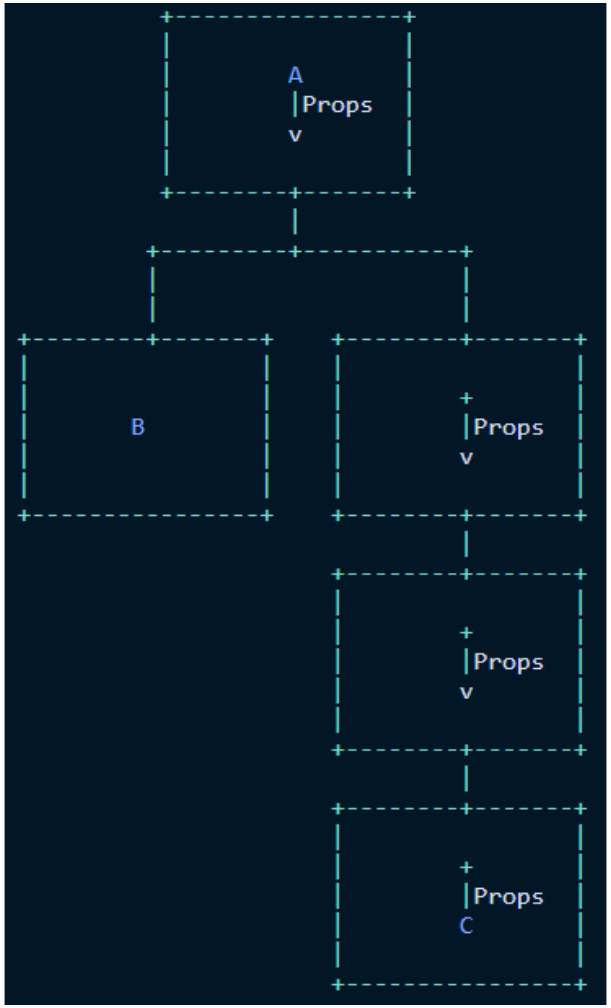
CONTEXT

Context

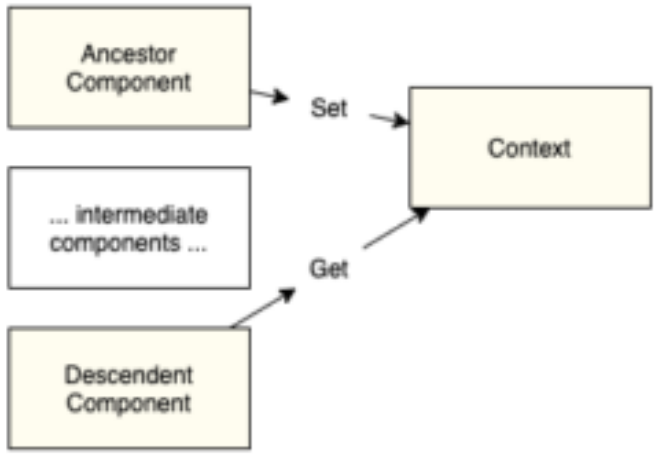
- Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language

```
class App extends React.Component {
    render() {
      return <Toolbar theme="dark" />;
    }
  }

  function Toolbar(props) {
    return (
      <div>
        <ThemedButton theme={props.theme} />
      </div>
    );
  }

  class ThemedButton extends React.Component {
    render() {
      return <Button theme={this.props.theme} />;
    }
  }
```



React Context API

- Store the state in a Context value in the common ancestor component (called the Provider Component), and access it from as many components as needed (called Consumer Components), which can be nested at any depth under this ancestor.
- This solution has the same benefits as the Props solution, but because of what could be called “hierarchical scoping”, it has the added benefit that any component can access the state in any Context that is rooted above itself in React’s hierarchy, without this state needing to be passed down to it as props.
- React.js takes care of all the magic behind the scenes to make this work.
- Primary situations where the React Context API really shines are:
  - When your state needs to be accessed or set from deeply nested components.
  - When your state needs to be accessed or set from many child components.



Three aspects to using React Contexts

- 1) Defining the Context object so we can use it.
  - If we wanted to store data about the current user of a web app, we could create a UserContext that can be used in the next two steps:

```
// Here we provide the initial value of the context
const UserContext = React.createContext({
    currentUser: null,
});
```

Note: It doesn't matter where this Context lives, as long as it can be accessed by all components that need to use it in the next two steps.

- 2) Providing a value for a Context in the hierarchy.
  - Assuming you had an AccountView component, you might provide a value like this

```
const AccountView = (props) => {
    const [currentUser, setCurrentUser] = React.useState(null);
    return (
        {/* Here we provides the actual value for its descendents */}
        <UserContext.Provider value={{ currentUser: currentUser }}>
        <AccountSummary/>
        <AccountProfile/>
        </UserContext.Provider>
    );
};
```

Three aspects to using React Contexts

- 3) Accessing the current Context value lower in the hierarchy.
  - If the AccountSummary component needed the user, we could have just passed it as a prop. But let's assume that it doesn't directly access the user data, but rather contains another component that does:

```
// Here we don't use the Context directly, but render children that do.
const AccountSummary = (props) => {
  return (
    <AccountSummaryHeader/>
    <AccountSummaryDashboard/>
    <AccountSummaryFooter/>
  );
};
```

  - All three of these child components may not want to access the current user's data. But as an example, let's just look at the AccountSummaryHeader component:

```
const AccountSummaryHeader = (props) => {
// Here we retrieve the current value of the context
   const context = React.useContext(UserContext);
    return (
       <section><h2>{context.currentUser.name}</h2> </section>
   );
};
```

Context

- Using context, we can avoid passing props through intermediate elements

```
const ThemeContext = React.createContext('light');
class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
// A component in the middle doesn't have to pass the theme d
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```