Context

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

In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

When to Use 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. For example, in the code below we manually thread through a “theme” prop in order to style the Button component:

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

function Toolbar(props) {
  // The Toolbar component must take an extra "theme" prop
  // and pass it to the ThemedButton. This can become painful
  // if every single button in the app needs to know the theme
  // because it would have to be passed through all components.
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}

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

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

```
// Context lets us pass a value deep into the component tree
// without explicitly threading it through every component.
// Create a context for the current theme (with "light" as the default).
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
```

```
// A component in the middle doesn't have to
// pass the theme down explicitly anymore.
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // Assign a contextType to read the current theme context.
  // React will find the closest theme Provider above and use its value.
  // In this example, the current theme is "dark".
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

Before You Use Context

Context is primarily used when some data needs to be accessible by *many* components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.

For example, consider a `Page` component that passes a `user` and `avatarSize` prop several levels down so that deeply nested `Link` and `Avatar` components can read it:

```
<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>
```

It might feel 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.

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}
```

With this change, only the top-most Page component needs to know about the `Link` and `Avatar` components' use of `user` and `avatarSize`.

This *inversion of control* can make your code cleaner in many cases by reducing the amount of props you need to pass through your application and giving more control to the root components. However, this isn't the right choice in every case: moving more complexity higher in the tree makes those higher-level components more complicated and forces the lower-level components to be more flexible than you may want.

You're not limited to a single child for a component. You may pass multiple children, or even have multiple separate "slots" for children, as documented here:

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

This pattern is sufficient for many cases when you need to decouple a child from its immediate parents. You can take it even further with render props if the child needs to communicate with the parent before rendering.

However, sometimes the same data needs to be accessible by many components in the tree, and at different nesting levels. Context lets you “broadcast” such data, and changes to it, to all components below. Common examples where using context might be simpler than the alternatives include managing the current locale, theme, or a data cache.

API

`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 **only** used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation with wrapping them. Note: passing `undefined` as a Provider value does not cause consumin components to use `defaultValue`.

`Context.Provider`

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

Every Context object comes with a Provider React component that 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. The propagation from Provider to its descendant consumers (including `.contextType` and `useContext`) is not subject to the `shouldComponentUpdate` method, so the consumer is updated even when an ancestor component skips an update.

Changes are determined by comparing the new and old values using the same algorithm as `Object.is`.

> **Note**
>
> The way changes are determined can cause some issues when passing objects as `value`: see Caveats.

`Class.contextType`

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
```

```
      /* ... */
    }
    render() {
      let value = this.context;
      /* render something based on the value of MyContext */
    }
  }
  MyClass.contextType = MyContext;
```

The `contextType` property on a class can be assigned a Context object created by `React.createContext()`. This lets you consume the nearest current value of that Context type using `this.context`. You can reference this in any of the lifecycle methods including the render function.

> **Note:**
> You can only subscribe to a single context using this API. If you need to read more than one see Consuming Multiple Contexts.
>
> If you are using the experimental public class fields syntax, you can use a **static** class field to initialize your `contextType`.

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

`Context.Consumer`

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

A React component that subscribes to context changes. This lets you subscribe to a con within a function component.

Requires a function as a child. The function receives the current context value and returns a React node. The `value` argument passed to the function will be equal to the `value` prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the `value` argument will be equal to the `defaultValue` that was passed to `createContext()`.

> **Note**
>
> For more information about the 'function as a child' pattern, see render props.

`Context.displayName`

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

For example, the following component will appear as MyDisplayName in the DevTools:

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

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

Examples

Dynamic Context

A more complex example with dynamic values for the theme:

theme-context.js

```
export const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222',
  },
};

export const ThemeContext = React.createContext(
  themes.dark // default value
);
```

themed-button.js

```
import {ThemeContext} from './theme-context';

class ThemedButton extends React.Component {
  render() {
    let props = this.props;
    let theme = this.context;
    return (
      <button
        {...props}
        style={{backgroundColor: theme.background}}
      />
    );
  }
}
ThemedButton.contextType = ThemeContext;

export default ThemedButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemedButton from './themed-button';

// An intermediate component that uses the ThemedButton
function Toolbar(props) {
  return (
    <ThemedButton onClick={props.changeTheme}>
      Change Theme
    </ThemedButton>
  );
}

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: themes.light,
    };

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };
  }

  render() {
    // The ThemedButton button inside the ThemeProvider
    // uses the theme from state while the one outside uses
    // the default dark theme
    return (
      <Page>
        <ThemeContext.Provider value={this.state.theme}>
          <Toolbar changeTheme={this.toggleTheme} />
        </ThemeContext.Provider>
        <Section>
          <ThemedButton />
        </Section>
      </Page>
    );
  }
}

ReactDOM.render(<App />, document.root);
```

Updating Context from a Nested Component

It is often necessary to update the context from a component that is nested somewhere deeply in the component tree. In this case you can pass a function down through the context to allow consumers to update the context:

theme-context.js

```
// Make sure the shape of the default value passed to
// createContext matches the shape that the consumers expect!
```

```
export const ThemeContext = React.createContext({
  theme: themes.dark,
  toggleTheme: () => {},
});
```

theme-toggler-button.js

```
import {ThemeContext} from './theme-context';

function ThemeTogglerButton() {
  // The Theme Toggler Button receives not only the theme
  // but also a toggleTheme function from the context
  return (
    <ThemeContext.Consumer>
      {({theme, toggleTheme}) => (
        <button
          onClick={toggleTheme}
          style={{backgroundColor: theme.background}}>
          Toggle Theme
        </button>
      )}
    </ThemeContext.Consumer>
  );
}

export default ThemeTogglerButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemeTogglerButton from './theme-toggler-button';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };

    // State also contains the updater function so it will
    // be passed down into the context provider
    this.state = {
      theme: themes.light,
      toggleTheme: this.toggleTheme,
    };
  }

  render() {
    // The entire state is passed to the provider
    return (
      <ThemeContext.Provider value={this.state}>
        <Content />
      </ThemeContext.Provider>
    );
  }
}

function Content() {
  return (
    <div>
      <ThemeTogglerButton />
    </div>
  );
}

ReactDOM.render(<App />, document.root);
```

Consuming Multiple Contexts

To keep context re-rendering fast, React needs to make each context consumer a separate node in the tree.

```
// Theme context, default to light theme
const ThemeContext = React.createContext('light');

// Signed-in user context
const UserContext = React.createContext({
  name: 'Guest',
});

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    // App component that provides initial context values
    return (
      <ThemeContext.Provider value={theme}>
```

```
        <UserContext.Provider value={signedInUser}>
          <Layout />
        </UserContext.Provider>
      </ThemeContext.Provider>
    );
  }
}

function Layout() {
  return (
    <div>
      <Sidebar />
      <Content />
    </div>
  );
}

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

If two or more context values are often used together, you might want to consider creating your own render prop component that provides both.

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>
    );
  }
}
```

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

Legacy API

> **Note**
>
> React previously shipped with an experimental context API. The old API will be supported in all 16.x releases, but applications using it should migrate to the new version. The legacy API will be removed in a future major React version. Read the legacy context docs here.

Is this page useful? 👍 👎 Edit this page