React Context

Context allows passing data through the component tree without passing props down manually at every level.

In React application, we passed data in a top-down approach via props. Sometimes it is inconvenient for certain types of props that are required by many components in the React application. Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

How to use Context

There are two main steps to use the React context into the React application:

1. Setup a context provider and define the data which you want to store.
2. Use a context consumer whenever you need the data from the store

When to use Context

Context is used to share data which can be considered "global" for React components tree and use that data where needed, such as the current authenticated user, theme, etc. For example, in the below code snippet, we manually thread through a "theme" prop to style the Button component.

1. **class** App **extends** React.Component {
2. render() {
3. **return** <Toolbar theme="dark" />;
4. }
5. }
7. function Toolbar(props) {
8. **return** (
9. <div>
10. <ThemedButton theme={props.theme} />
11. </div>
12. );
13. }
15. **class** ThemedButton **extends** React.Component {
16. render() {
17. **return** <Button theme={**this**.props.theme} />;
18. }
19. }

In the above code, the Toolbar function component takes an extra "theme" prop and pass it to the ThemedButton. It can become inconvenient if every single button in the app needs to know the theme because it would be required to pass through all components. But using context, we can avoid passing props for every component through intermediate elements.

We can understand it from the below example. Here, context passes a value into the component tree without explicitly threading it through every single component.

1. // Create a context for the current theme which is "light" as the default.
2. **const** ThemeContext = React.createContext('light');
4. **class** App **extends** React.Component {
5. render() {
6. /\* Use a ContextProvider to pass the current theme, which allows every component to read it, no matter how deep it is. Here, we are passing the "dark" theme as the current value.\*/
8. **return** (
9. <ThemeContext.Provider value="dark">
10. <Toolbar />
11. </ThemeContext.Provider>
12. );
13. }
14. }
16. // Now, it is not required to pass the theme down explicitly for every component.
17. function Toolbar(props) {
18. **return** (
19. <div>
20. <ThemedButton />
21. </div>
22. );
23. }
24. // consumer
25. **class** ThemedButton **extends** React.Component {
26. **static** contextType = ThemeContext;
27. render() {
28. **return** <Button theme={**this**.context} />;
29. }
30. }

React Context API

The React Context API is a component structure, which allows us to share data across all levels of the application. The main aim of Context API is to solve the problem of prop drilling (also called "Threading"). The Context API in React are given below.

1. React.createContext
2. Context.provider
3. Class.contextType

React.createContext

It creates a context object. When React renders a component which subscribes to this context object, then it will read the current context value from the matching provider in the component tree.

**Syntax**

1. **const** MyContext = React.createContext(defaultValue);

When a component does not have a matching Provider in the component tree, it returns the defaultValue argument. It is very helpful for testing components isolation (separately) without wrapping them.

Context.Provider

Every Context object has a Provider React component which allows consuming components to subscribe to context changes. It acts as a delivery service. When a consumer component asks for something, it finds it in the context and provides it to where it is needed.

**Syntax**

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

It accepts the value prop and passes to consuming components which are descendants of this Provider. We can connect one Provider with many consumers. Context Providers can be nested to override values deeper within the component tree. All consumers that are descendants of a Provider always re-render whenever the Provider's value prop is changed. The changes are determined by comparing the old and new values using the same algorithm as **Object.is** algorithm.

Class.contextType

The contextType property on a class used to assign a Context object which is created by React.createContext(). It allows you to consume the closest current value of that Context type using this.context. We can reference this in any of the component life-cycle methods, including the render function.

**Note:** We can only subscribe to a single context using this API. If we want to use the experimental public class field's syntax, we can use a static class field to initialize the contextType.

**React Context API Example**

**Step1** Create a new React app using the following command.

1. $ npx create-react-app mycontextapi

**Step2** Install bootstrap CSS framework using the following command.

1. $ npm install react-bootstrap bootstrap --save

**Step3** Add the following code snippet in the src/APP.js file.

1. **import** React, { Component } from 'react';
2. **import** 'bootstrap/dist/css/bootstrap.min.css';
4. **const** BtnColorContext = React.createContext('btn btn-darkyellow');
6. **class** App **extends** Component {
7. render() {
8. **return** (
9. <BtnColorContext.Provider value="btn btn-info">
10. <Button />
11. </BtnColorContext.Provider>
12. );
13. }
14. }
16. function Button(props) {
17. **return** (
18. <div className="container">
19. <ThemedButton />
20. </div>
21. );
22. }
24. **class** ThemedButton **extends** Component {
26. **static** contextType = BtnColorContext;
27. render() {
28. **return** <button className={**this**.context} >
29. welcome to javatpoint
30. </button>;
31. }
32. }
33. export **default** App;

In the above code snippet, we have created the context using React.createContext(), which returns the Context object. After that, we have created the wrapper component which returns the Provider component, and then add all the elements as children from which we want to access the context.

**output:**

When we run the React app, we will get the following screen.

