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.

```
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} />;
  }
}
```

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.

```
// Create a context for the current theme which is "light" as the default.
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    /* Use a ContextProvider to pass the current theme, which allows every component to read it, no matter how c

    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

// Now, it is not required to pass the theme down explicitly for every component.
function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

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

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. Context.Consumer
4. 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

```
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

```
<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.

Context.Consumer

It is the React component which subscribes to the context changes. It allows us to subscribe to the context within the function component. It requires the function as a component. A consumer is used to request data through the provider and manipulate the central data store when the provider allows it.

Syntax

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

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

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.

```
$ npx create-react-app mycontextapi
```

Step2 Install bootstrap CSS framework using the following command.

```
$ npm install react-bootstrap bootstrap --save
```

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

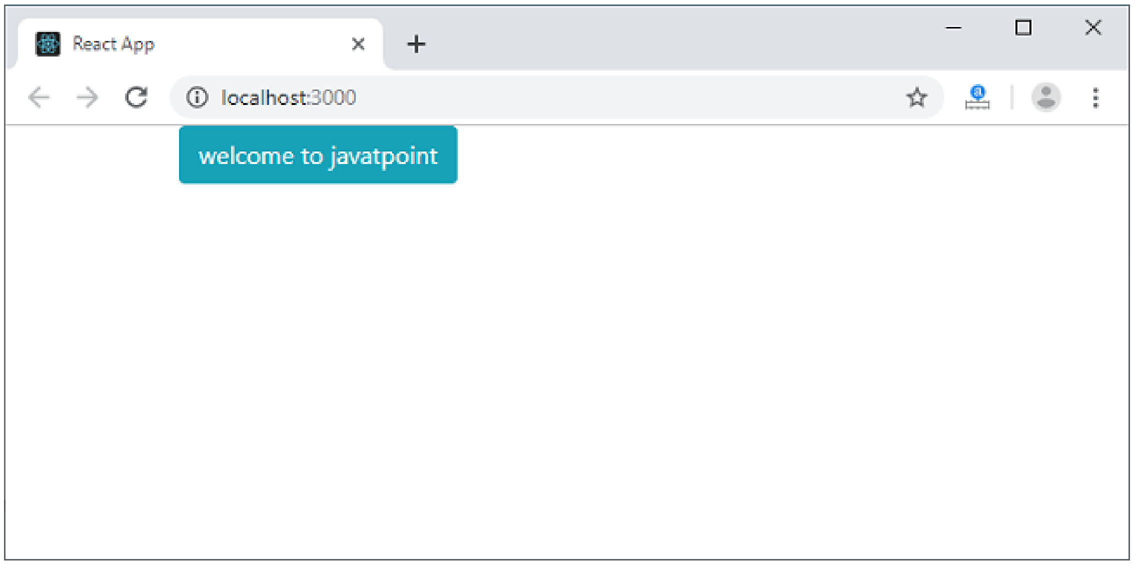
```
import React, { Component } from 'react';
import 'bootstrap/dist/css/bootstrap.min.css';
```

```
const BtnColorContext = React.createContext('btn btn-darkyellow');

class App extends Component {
  render() {
    return (
      <BtnColorContext.Provider value="btn btn-info">
        <Button />
      </BtnColorContext.Provider>
    );
  }
}

function Button(props) {
  return (
  <div className="container">
    <ThemedButton />
  </div>
  );
}

class ThemedButton extends Component {

  static contextType = BtnColorContext;
  render() {
    return <button className={this.context} >
      welcome to javatpoint
    </button>;
  }
}
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.



← Prev Next →

Youtube For Videos Join Our Youtube Channel: Join Now

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share

Learn Latest Tutorials

Splunk tutorial
Splunk

SPSS tutorial
SPSS

Swagger tutorial
Swagger

T-SQL tutorial
Transact-SQL

Tumblr tutorial
Tumblr

React tutorial
ReactJS

Regex tutorial
Regex

Reinforcement learning tutorial
Reinforcement Learning

R Programming tutorial
R Programming

RxJS tutorial
RxJS

React Native tutorial
React Native

Python Design Patterns
Python Design Patterns

Python Pillow tutorial
Python Pillow

Python Turtle tutorial
Python Turtle

Keras tutorial
Keras

Preparation

Aptitude
Aptitude

Logical Reasoning
Reasoning

Verbal Ability
Verbal Ability

Interview Questions
Interview Questions

Company Interview Questions
Company Questions

Trending Technologies

Artificial Intelligence

Artificial Intelligence

AWS Tutorial

AWS

Selenium tutorial

Selenium

Cloud Computing

Cloud Computing

Hadoop tutorial

Hadoop

ReactJS Tutorial

ReactJS

Data Science Tutorial

Data Science

Angular 7 Tutorial

Angular 7

Blockchain Tutorial

Blockchain

Git Tutorial

Git

Machine Learning Tutorial

Machine Learning

DevOps Tutorial

DevOps

B.Tech / MCA

DBMS tutorial

DBMS

Data Structures tutorial

Data Structures

DAA tutorial

DAA

Operating System

Operating System

Computer Network tutorial

Computer Network

Compiler Design tutorial

Compiler Design

Computer Organization and Architecture

Computer Organization

Discrete Mathematics Tutorial

Discrete Mathematics

Ethical Hacking

Ethical Hacking

Computer Graphics Tutorial

Computer Graphics

Software Engineering

html tutorial

Web Technology

Cyber Security tutorial

Automata Tutorial

C Language tutorial

Software Engineering

Cyber Security

Automata

C Programming

C++ tutorial

C++

Java tutorial

Java

.Net Framework tutorial

.Net

Python tutorial

Python

List of Programs

Programs

Control Systems tutorial

Control System

Data Mining Tutorial

Data Mining

Data Warehouse Tutorial

Data Warehouse