Advance Topics in React JS

* What is 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.

* **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.

* When to use the context API

Context API 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.+a

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} />;

  }

}

* How to use the Context API

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

Setup a context API provider and define the data which you want to store.

Use a context API consumer whenever you need the data from the store

* Fragments

whenever you want to render something on the screen, you need to use a render method inside the component. This render method can return **single** elements or **multiple** elements. The render method will only render a single root node inside it at a time. However, if you want to return multiple elements, the render method will require a '**div**' tag and put the entire content or elements inside it. This extra node to the DOM sometimes results in the wrong formatting of your HTML output and also not loved by the many developers.

**Syntax**

<React.Fragment>

      <h2> child1 </h2>

    <p> child2 </p>

      .. ..... .... ...

</React.Fragment>

**Example**

// Rendering with fragments tag

class App extends React.Component {

    render() {

     return (

       <React.Fragment>

            <h2> Hello World! </h2>

        <p> Welcome to the JavaTpoint. </p>

         </React.Fragment>

     );

    }

}

* Why we use Fragments?

The main reason to use Fragments tag is:

1. It makes the execution of code faster as compared to the div tag.
2. It takes less memory.

* Fragments short Syntax

There is also another shorthand exists for declaring fragments for the above method. It looks like **empty** tag in which we can use of '<>' and '' instead of the '**React.Fragment**'.

**Example**

//Rendering with short syntax

class Columns extends React.Component {

  render() {

    return (

      <>

        <h2> Hello World! </h2>

        <p> Welcome to the JavaTpoint </p>

      </>

    );

  }

}

* Keyed Fragments

The shorthand syntax does not accept key attributes. You need a key for mapping a collection to an array of fragments such as to create a description list. If you need to provide keys, you have to declare the fragments with the explicit <**React.Fragment**> syntax.

**Example**

Function  = (props) {

  return (

    <Fragment>

      {props.items.data.map(item => (

        // Without the 'key', React will give a key warning

        <React.Fragment key={item.id}>

          <h2>{item.name}</h2>

          <p>{item.url}</p>

          <p>{item.description}</p>

        </React.Fragment>

      ))}

    </Fragment>

  )

}

* Higher-order components

Also known as HOC, the React Higher-Order Components is an advanced technique that takes a component and returns a new component. It is used for reusing component logic.

**Higher-Order Component Conventions:**

* HOCs should not be used inside the render method of a component.
* To have access to the higher-order components, the static methods must be copied over.
* Since ‘Refs’ does not pass through as a parameter or argument, hence HOCs does not work for refs.
* If a ref is added to an element in the HOC component, it will not be referred to as a wrapped component but will be referred to as an instance of the outermost container component.

|  |
| --- |
| const NewComponent = higherOrderComponent(WrappedComponent); |

**Syntax:**

**HOC.js:**

|  |
| --- |
| import React, {Component} from 'react';  export default function Hoc(HocComponent){  return class extends Component{  render(){  return (  <div>  <HocComponent></HocComponent>  </div>  );  }  }  } |

**App.js:**

|  |
| --- |
| import React, { Component } from 'react';  import Hoc from './HOC';  class App extends Component {  render() {  return (  <div>  <h2>Hello World!!</h2>  <p> Have a Great day.</p>  </div>  )  }  }  App = Hoc(App);  export default App; |

**Output:**  
