React Higher-Order Components

It is also known as HOC. In React, HOC is an advanced technique for reusing component logic. It is a function that takes a component and returns a new component. According to the official website, it is not the feature(part) in React API, but a pattern that emerges from React compositional nature. They are similar to JavaScript functions used for adding additional functionalities to the existing component.

A higher order component function accepts another function as an argument. The **map** function is the best example to understand this. The main goal of this is to decompose the component logic into simpler and smaller functions that can be reused as you need.

Syntax

```
const NewComponent = higherOrderComponent(WrappedComponent);
```

We know that component transforms props into UI, and a higher-order component converts a component another component and allows to add additional data or functionality into this. **Hocs** are common in **third-party** libraries. The examples of HOCs are **Redux's connect** and **Relay's createFragmentContainer**.

Now, we can understand the **working of HOCs** from the below example.

```
//Function Creation
function add (a, b) {
  return a + b
}
function higherOrder(a, addReference) {
  return addReference(a, 20)
}
//Function call
higherOrder(30, add) // 50
```

In the above example, we have created two functions **add()** and **higherOrder()**. Now, we provide the add() function as an **argument** to the higherOrder() function. For invoking, rename it **addReference** in the higherOrder() function, and then **invoke it**.

Here, the function you are passing is called a callback function, and the function where you are passing the callback function is called a **higher-order(HOCs)** function.

Example

Create a new file with the name HOC.js. In this file, we have made one function HOC. It accepts one **argument** as a component. Here, that component is **App**.

HOC.js

```
import React, {Component} from 'react';

export default function Hoc(HocComponent){
    return class extends Component{
        render(){
            return (
                <div>
                    <HocComponent></HocComponent>
                </div>

            );
        }
    }
}
```

Now, include **HOC.js** file into the **App.js** file. In this file, we need to **call** the HOC function.

```
App = Hoc(App);
```

The App component wrapped inside another React component so that we can modify it. Thus, it becomes the primary application of the Higher-Order Components.

App.js

```
import React, { Component } from 'react';
import Hoc from './HOC';

class App extends Component {
  render() {
    return (
      <div>
        <h2>HOC Example</h2>
```

```
9.          JavaTpoint provides best CS tutorials.
10.       </div>
11.     )
12.   }
13. }
14. App = Hoc(App);
15. export default App;
```

Output

When we execute the above file, it will give the output as below screen.

Higher-Order Component Conventions

- Do not use HOCs inside the render method of a component.
- The static methods must be copied over to have access to them. You can do this using hoist-non-react-statics package to automatically copy all non-React static methods.
- HOCs does not work for refs as 'Refs' does not pass through as a parameter or argument. If you add a ref to an element in the HOC component, the ref refers to an instance of the outermost container component, not the wrapped component.

React Code Splitting

The React app bundled their files using tools like **Webpack** or **Browserfy**. Bundling is a process which takes multiple files and merges them into a single file, which is called a **bundle**. The bundle is responsible for loading an entire app at once on the webpage. We can understand it from the below example.

App.js

```
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

math.js

```
export function add(a, b) {
  return a + b;
}
```

Bundle file as like below:

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

As our app grows, our bundle will grow too, especially when we are using large third-party libraries. If the bundle size gets large, it takes a long time to load on a webpage. For avoiding the large bundling, it?s good to start ?splitting? your bundle.

React 16.6.0, released in **October 2018**, and introduced a way of performing code splitting. Code-Splitting is a feature supported by Webpack and Browserify, which can create multiple bundles that can be dynamically loaded at runtime.

Code splitting uses **React.lazy** and **Suspense** tool/library, which helps you to load a dependency lazily and only load it when needed by the user.

The code splitting improves:

- The performance of the app
- The impact on memory

- The downloaded Kilobytes (or Megabytes) size

React.lazy

The best way for code splitting into the app is through the dynamic **import()** syntax. The React.lazy function allows us to render a dynamic import as a regular component.

Before

```
import ExampleComponent from './ExampleComponent';

function MyComponent() {
  return (
    <div>
      <ExampleComponent />
    </div>
  );
}
```

After

```
const ExampleComponent = React.lazy(() => import('./ExampleComponent'));

function MyComponent() {
  return (
    <div>
      <ExampleComponent />
    </div>
  );
}
```

The above code snippet automatically loads the bundle which contains the ExampleComponent when the ExampleComponent gets rendered.

Suspense

If the module which contains the ExampleComponent is not yet loaded by the function component(MyComponent), then we need to show some **fallback** content while we are waiting for it to load. We can do this using the suspense component. In other words, the suspense component is responsible for handling the output when the lazy component is fetched and rendered.

```
const ExampleComponent = React.lazy(() => import('./ ExampleComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <ExampleComponent />
      </Suspense>
    </div>
  );
}
```

The **fallback** prop accepts the React elements which you want to render while waiting for the component to load. We can combine multiple lazy components with a single Suspense component. It can be seen in the below example.

```
const ExampleComponent = React.lazy(() => import('./ ExampleComponent'));
const ExamComponent = React.lazy(() => import('./ ExamComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <ExampleComponent />
          <ExamComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

> *Note:* ***React.lazy and Suspense components are not yet available for server-side rendering. For code-splitting in a server-rendered app, it is recommended to use Loadable Components.***

Error boundaries

If any module fails to load, for example, due to network failure, we will get an error. We can handle these errors with Error Boundaries. Once we have created the Error Boundary, we can use it anywhere above our lazy components to display an error state.

```
import MyErrorBoundary from './MyErrorBoundary';
const ExampleComponent = React.lazy(() => import('./ ExampleComponent'));
const ExamComponent = React.lazy(() => import('./ ExamComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <ExampleComponent />
          <ExamComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);
```

Route-based code splitting

It is very tricky to decide where we introduce code splitting in the app. For this, we have to make sure that we choose the place which will split the bundles evenly without disrupting the user experience.

The route is the best place to start the code splitting. Route based code splitting is essential during the page transitions on the web, which takes some amount of time to load. Here is an example of how to setup route-based code splitting into the app using React Router with React.lazy.

```
import { Switch, BrowserRouter as Router, Route} from 'react-router-dom';
import React, { Suspense, lazy } from 'react';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));
```

```
6.  const Contact = lazy(() => import('./routes/Contact'));
7.
8.  const App = () => (
9.    <Router>
10.     <Suspense fallback={<div>Loading...</div>}>
11.       <Switch>
12.         <Route exact path="/" component={Home}/>
13.         <Route path="/about" component={About}/>
14.         <Route path="/contact" component={Contact}/>
15.       </Switch>
16.     </Suspense>
17.   </Router>
18. );
```

Named Export

Currently, React.lazy supports default exports only. If any module you want to import using named exports, you need to create an intermediate module that re-exports it as the default. We can understand it from the below example.

ExampleComponents.js

```
1. export const MyFirstComponent = /* ... */;
2. export const MySecondComponent = /* ... */;
```

MyFirstComponent.js

```
1. export { MyFirstComponent as default } from "./ExampleComponents.js";
```

MyApp.js

```
1. import React, { lazy } from 'react';
2. const MyFirstComponent = lazy(() => import("./MyFirstComponent.js"));
```

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 deep it is. Here, we are passing the "dark" theme as the current value.*/

    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 {
```

```
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. 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

```
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.

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.