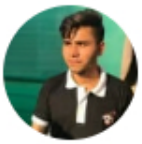
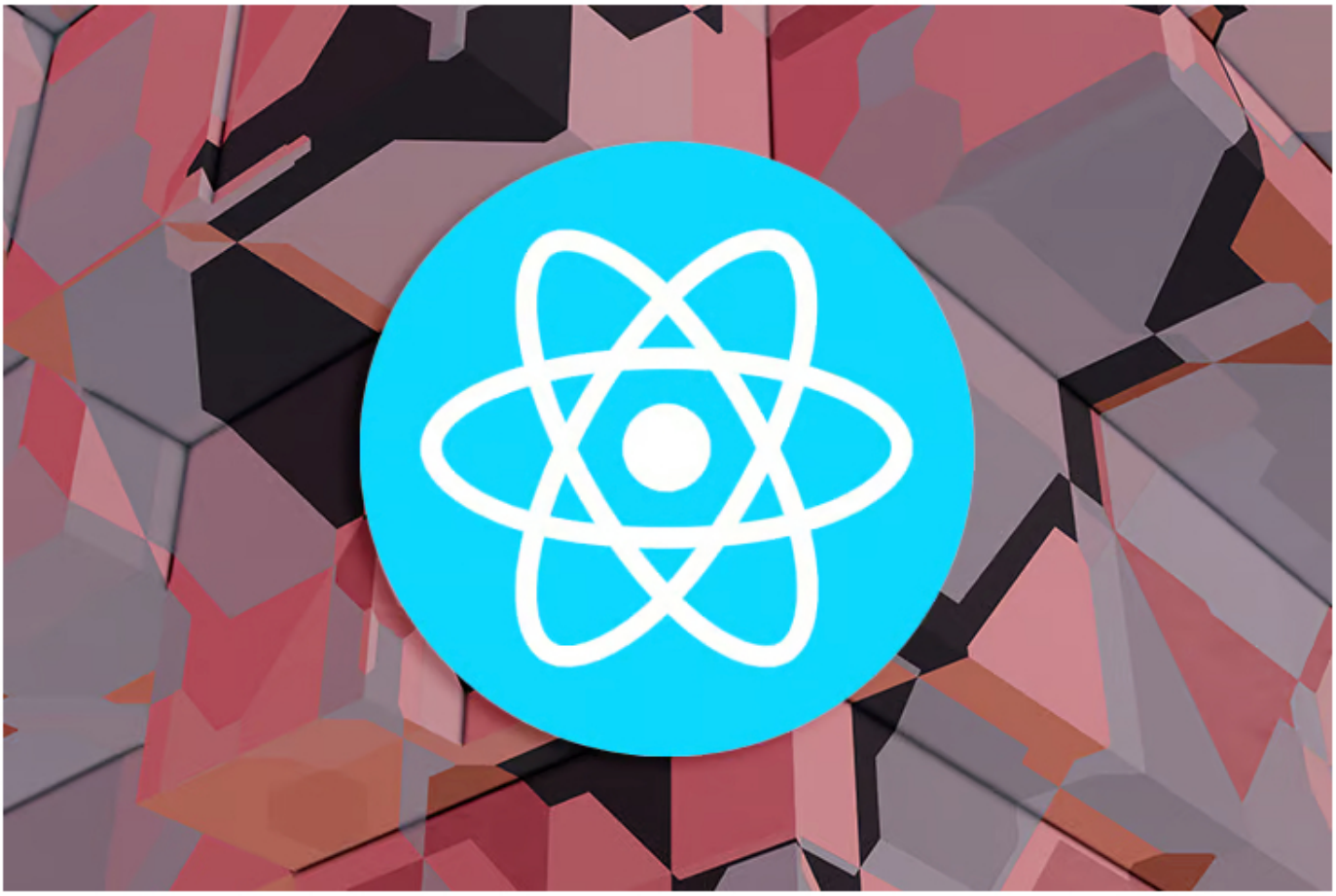


Hussain Arif Follow

Hussain is a CS student in Pakistan whose biggest interest is learning and teaching programming to make the world a better place.

Understanding React higher-order components

July 20, 2022 · 7 min read



In this article, we'll cover the basics of React's HOC concept, including introducing you to higher-order components, teaching you the syntax, and showing you how you can apply HOCs. We'll also go over some common problems you might encounter with higher-order components.

Here are the steps that we'll take:

Introduction

What are HOCs and when should you use them?

Let's say that the user wants a component that increments a counter variable on every `onClick` event:

```
function ClickCounter() {
  const [count, setCount] = useState(0); //default value of this state wil
  return (
    <div>
      {/*When clicked, increment the value of 'count'*/}
      <button onClick={() => setCount((count) => count + 1)}>Increment</bu
      <p>Clicked: {count}</p> {/*Render the value of count*/}
    </div>
  );
}
export default ClickCounter;
```

Our code works! However, consider this situation: what if the client wants another component that contains the same functionality, but it triggers on an `onMouseOver` event?

To make this possible, we would have to write the following code:

```
function HoverCounter(props) {
  const [count, setCount] = useState(0);
  return (
    <div>
      {/*If the user hovers over this button, then increment 'count'*/}
      <button onMouseOver={() => setCount((count) => count + 1)}>
        Increment
      </button>
      <p>
        Clicked: {count}
      </p>
    </div>
  );
}
export default HoverCounter;
```

Even though our code samples are valid, there is a major problem: both of the files possess similar code logic.

Consequently, this breaks the DRY concept. So how do we fix this issue?

This is where HOCs come in. Here, higher-order components allow developers to reuse code logic in their project. As a result, this means less repetition and more optimized, readable code.

Now that we've covered its advantage, let's get started using HOCs!

Syntax

According to React's documentation, a typical React HOC has the following definition:

A higher-order component is a function that takes in a component and returns a new component.

Using code, we can rewrite the above statement like so:

```
const newComponent = higherFunction(WrappedComponent);
```

In this line:

- `newComponent` — will be the enhanced component
- `higherFunction` — as the name suggests, this function will enhance `WrappedComponent`
- `WrappedComponent` — The component whose functionality we want to extend. In other words, this will be the component that we want to enhance

In the next segment of the article, we will see React's HOC concept in action.

Using HOCs

Initializing our repository

Before writing some code, we have to first create a blank React project. To do so, start by writing the following code:

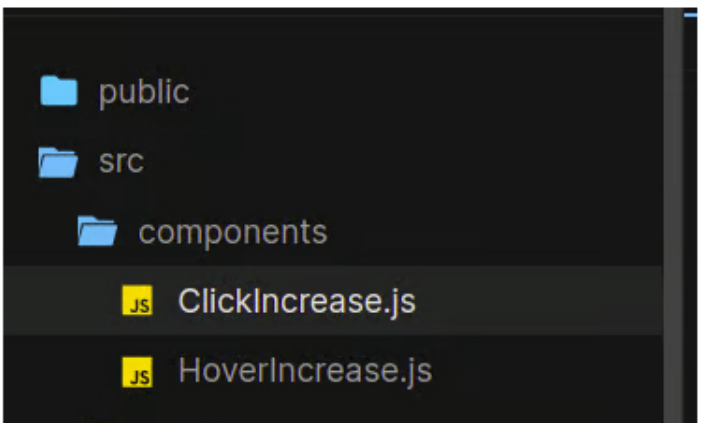
```
npx create-react-app hoc-tutorial
cd hoc-tutorial #navigate to the project folder.
cd src #go to codebase
mkdir components #will hold all our custom components
```

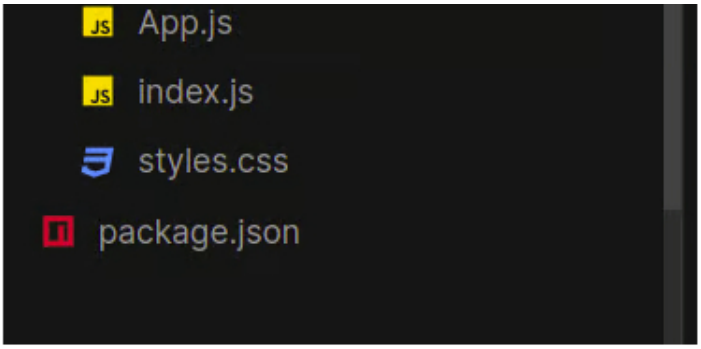
For this article, we will build two custom components to demonstrate HOC usage:

- `ClickIncrease.js` — This component will render a button and a piece of text. When the user clicks on this button (an `onClick` event), the `fontSize` property of the text will increase
- `HoverIncrease.js` — Will be similar to that of `ClickIncrease`. However, unlike the former, this component will listen to `onMouseOver` events

In your project, navigate to the `components` folder. Here, create these two new files.

When that's done, your file structure should look like so:





Now that we have laid out the groundwork for the project, it's time to build our custom components.

Coding our components

In `ClickIncrease.js`, start by writing the following code:

```
//file name: components/ClickIncrease.js
function ClickIncrease() {
  const [fontSize, setFontSize] = useState(10); //set initial value of Hoo
  return (
    <div>
      {/*When clicked, increment the value of fontSize*/}
      <button onClick={() => setFontSize((size) => size + 1)}>
        Increase with click
      </button>
      {/*Set the font size of this text to the fontSize variable.*/}
      {/*Furthermore, display its value as well.*/}
      <p style={{ fontSize }}>Size of font in onClick function: {fontSize}
    </div>
  );
}
export default ClickIncrease;
```

Next, in your `HoverIncrease` component, paste these lines of code:

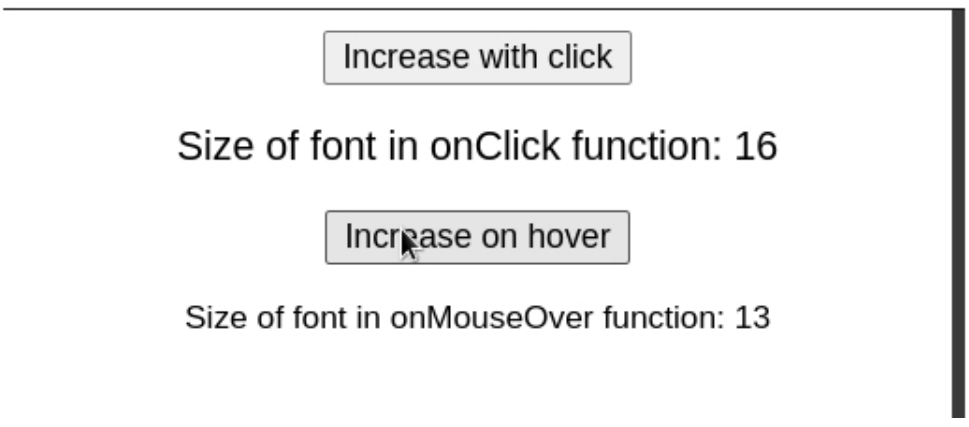
```
function HoverIncrease(props) {
  const [fontSize, setFontSize] = useState(10);
  return (
    <div>
```

```
      {/*This time, instead of listening to clicks,*/}
      {/*Listen to hover events instead*/}
      <button onMouseOver={() => setFontSize((size) => size + 1)}>
        Increase on hover
      </button>
      <p style={{ fontSize }}>
        Size of font in onMouseOver function: {fontSize}
      </p>
    </div>
  );
}
export default HoverIncrease;
```

Finally, render these functions to the GUI like so:

```
//import both components
import ClickIncrease from "./components/ClickIncrease";
import HoverIncrease from "./components/HoverIncrease";
export default function App() {
  return (
    <div className="App">
      {/*Render both of these components to the UI */}
      <ClickIncrease />
      <HoverIncrease />
    </div>
  );
}
```

Let's test it out! This will be the outcome of the code:



Creating and using our HOC function

Within the `components` folder, create a file called `withCounter.js`. Here, start by writing the following code:

```
import React from "react";
const UpdatedComponent = (OriginalComponent) => {
  function NewComponent(props) {
  //render OriginalComponent and pass on its props.
    return <OriginalComponent />;
  }
  return NewComponent;
};
export default UpdatedComponent;
```

Let's deconstruct this code piece by piece:

- In the start, we created a function called `UpdatedComponent` that takes in an argument called `OriginalComponent`. In this case, the `OriginalComponent` will be the React element which will be wrapped
- Later on, we told React to render `OriginalComponent` to the UI. We will implement enhancement functionality later in this article

When that's done, it's now time to use the `UpdatedComponent` function in our app.

To do so, first go to the `HoverIncrease.js` file and write the following lines:

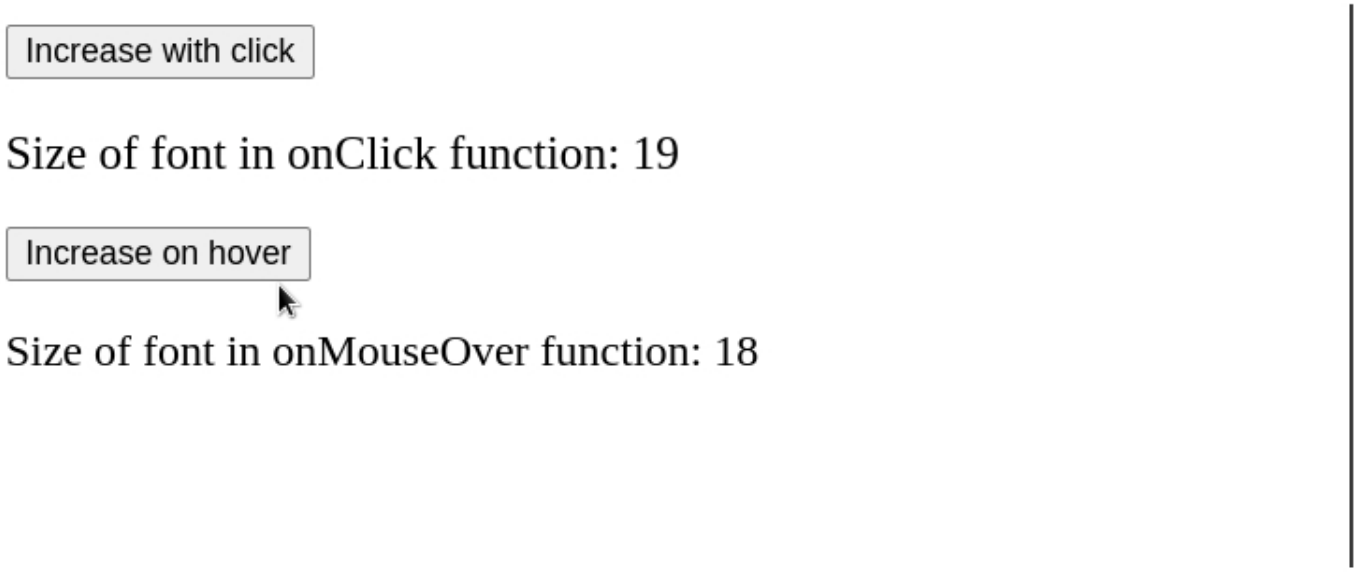
```
import withCounter from "./withCounter.js" //import the withCounter functi
//..further code ..
function HoverIncrease() {
//..further code
}
//replace your 'export' statement with:
```

```
export default withCounter(HoverIncrease);
//We have now converted HoverIncrease to an HOC function.
```

Next, do the same process with the `ClickIncrease` module:

```
//file name: components/ClickIncrease.js
import withCounter from "./withCounter";
function ClickIncrease() {
//...further code
}
export default withCounter(ClickIncrease);
//ClickIncrease is now a wrapped component of the withCounter method.
```

Let's test it out! This will be the result in the code:



Notice that our result is unchanged. This is because we haven't made changes to our HOC yet. In the next subsection, you will learn how to share props between our components.

Sharing props

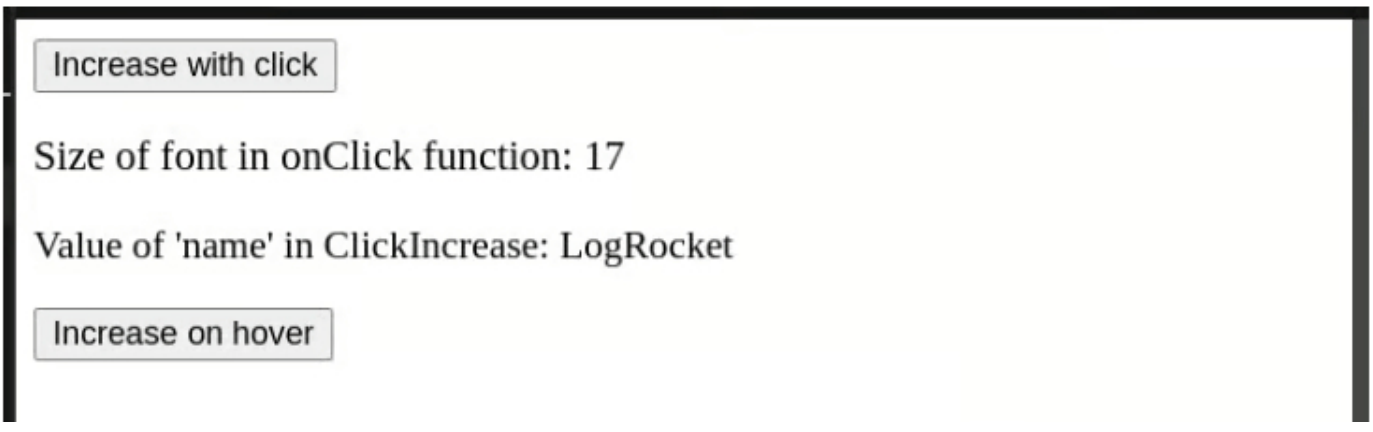
Via HOCs, React allows users to share props within the project's wrapped components.

As a first step, create a `name` prop in `withCounter.js` like so:

```
//file name: components/withCounter.js
const UpdatedComponent = (OriginalComponent) => {
  function NewComponent(props) {
    //Here, add a 'name' prop and set its value of 'LogRocket'.
    return <OriginalComponent name="LogRocket" />;
  }
//..further code..
```

That's it! To read this data prop, all we have to do is to make the following changes to its child components:

```
    <div>
      {/* Further code..*/}
      {/*Now render the value of the 'name' prop */ }
      <p> Value of 'name' in HoverIncrease: {props.name}</p>
    </div>
  );
}
//Now In components/ClickIncrease.js
function ClickIncrease(props) {
  //accept incoming props
  return (
    <div>
      {/*Further code..*/}
      <p>Value of 'name' in ClickIncrease: {props.name}</p>
    </div>
  );
}
```



> Size of font in onMouseOver function: 24
>
> Value of 'name' in HoverIncrease: LogRocket

That was easy! As you can see, React's HOC design allows developers to share data between components with relative ease.

In upcoming sections, you will now learn how to share states via HOC functions.

Sharing state variables with Hooks

Just like props, we can even share Hooks:

```
//In components/withCounter.js
const UpdatedComponent = (OriginalComponent) => {
  function NewComponent(props) {
    const [counter, setCounter] = useState(10); //create a Hook
    return (
      <OriginalComponent
        counter={counter} //export our counter Hook
```

```
        //now create an 'incrementSize' function
        incrementCounter={() => setCounter((counter) => counter + 1)}
      />
    );
  }
//further code..
```

Here's an explanation of the code:

- First, we created a Hook variable called `counter` and set its initial value to `10`
- Other than that, we also coded an `incrementCounter` function. When invoked, this method will increment the value of `counter`
- Finally, export the `incrementSize` method and the `size` Hook as props. As a result, this allows the wrapped components of `UpdatedComponent` to get access to these Hooks

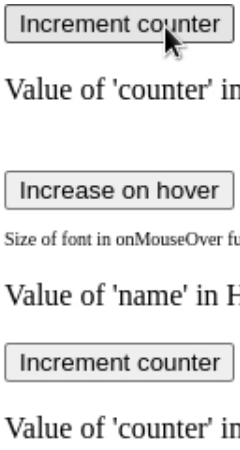
As the last step, we now have to use the `counter` Hook.
To do so, write these lines of code in the `HoverIncrease` and `ClickIncrease` module:

```
//make the following file changes to components/HoverIncrease.js and Click
//extract the counter Hook and incrementCounter function from our HOC:
const { counter, incrementCounter } = props;
return (
  <div>
    {/*Use the incrementCounter method to increment the 'counter' state..*
    <button onClick={() => incrementCounter()}>Increment counter</button>
    {/*Render the value of our 'counter' variable:*/}
    <p> Value of 'counter' in HoverIncrease/ClickIncrease: {counter}</p>
  </div>
);
```

Increase with click

Size of font in onClick function: 12

Value of 'name' in ClickIncrease: LogRocket



Here, one important thing to notice is that the value of the `counter` state is not shared between our child components. If you want to share states between various React components, please use React's Context API, which allows you to effortlessly share states and Hooks throughout your app.

Passing parameters

Even though our code works, consider the following situation: what if we want to increment the value of `counter` with a custom value? Via HOCs, we can even tell React to pass specific data to certain child components. This is made possible with parameters.

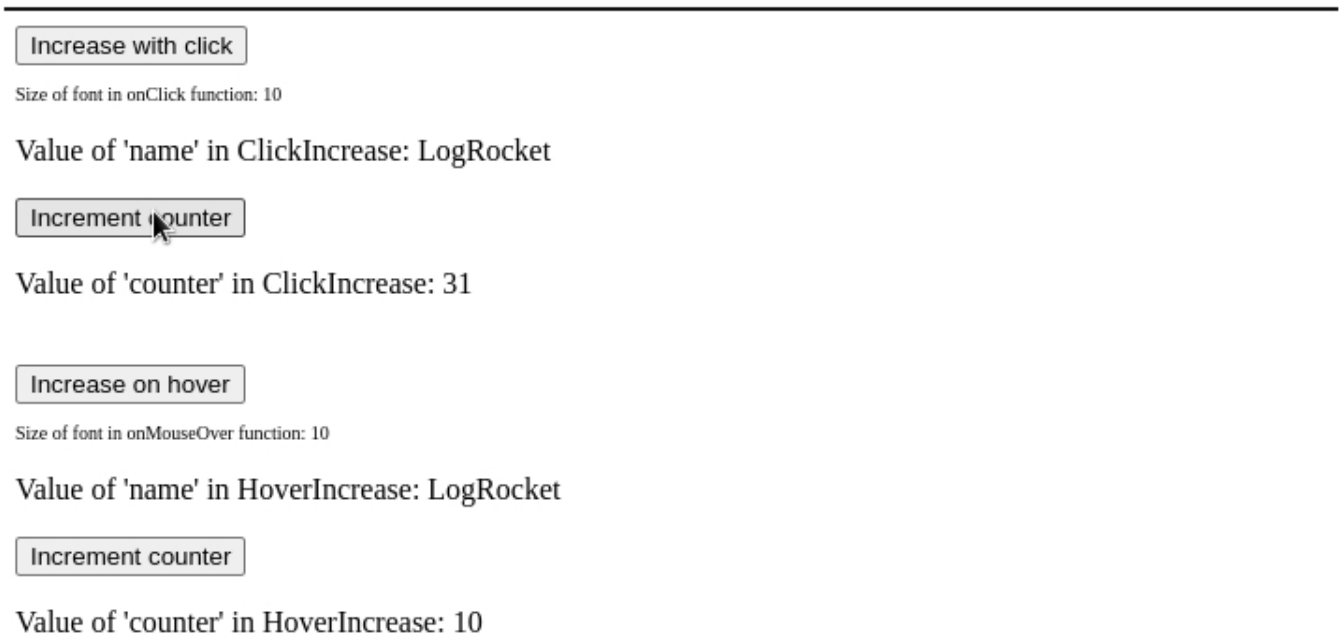
To enable support for parameters, write the following code in `components/withCounter.js`:

```
//This function will now accept an 'increaseCount' parameter.
const UpdatedComponent = (OriginalComponent, increaseCount) => {
  function NewComponent(props) {
    return (
      <OriginalComponent
        //this time, increment the 'size' variable by 'increaseCount'
        incrementCounter={() => setCounter((size) => size + increaseCount)}
      />
    );
//further code..
```

In this piece of code, we informed React that our function will now take in an additional parameter called `increaseCount`.

All that's left for us is to use this parameter in our wrapped components. To do so, add this line of code in `HoverIncrease.js` and `ClickIncrease.js`:

```
//In HoverIncrease, change the 'export' statement:
export default withCounter(HoverIncrease, 10); //value of increaseCount is
//this will increment the 'counter' Hook by 10.
//In ClickIncrease:
export default withCounter(ClickIncrease, 3); //value of increaseCount is
//will increment the 'counter' state by 3 steps.
```



In the end, the `withCounter.js` file should look like so:

```
import React from "react";
import { useState } from "react";
const UpdatedComponent = (OriginalComponent, increaseCount) => {
  function NewComponent(props) {
    const [counter, setCounter] = useState(10);
    return (
```

```
      <OriginalComponent
        name="LogRocket"
        counter={counter}
        incrementCounter={() => setCounter((size) => size + increaseCount)
      />
    );
  }
  return NewComponent;
};
export default UpdatedComponent;
```

Moreover, `HoverIncrease.js` should look like so:

```
  const [fontSize, setFontSize] = useState(10);
  const { counter, incrementCounter } = props;
  return (
    <div>
      <button onMouseOver={() => setFontSize((size) => size + 1)}>
        Increase on hover
      </button>
      <p style={{ fontSize }}>
        Size of font in onMouseOver function: {fontSize}
      </p>
      <p> Value of 'name' in HoverIncrease: {props.name}</p>
      <button onClick={() => incrementCounter()}>Increment counter</button
      <p> Value of 'counter' in HoverIncrease: {counter}</p>
    </div>
  );
}
export default withCounter(HoverIncrease, 10);
```

And finally, your `ClickIncrease` component should have the following code:

```
import { useEffect, useState } from "react";
import withCounter from "./withCounter";
function ClickIncrease(props) {
  const { counter, incrementCounter } = props;
  const [fontSize, setFontSize] = useState(10);
```

```
  return (
    <div>
      <button onClick={() => setFontSize((size) => size + 1)}>
        Increase with click
      </button>
      <p style={{ fontSize }}>Size of font in onClick function: {fontSize}
      <p>Value of 'name' in ClickIncrease: {props.name}</p>
      <button onClick={() => incrementCounter()}>Increment counter</button
      <p> Value of 'counter' in ClickIncrease: {counter}</p>
    </div>
  );
}
```

Common problems

Passing down props to specific components

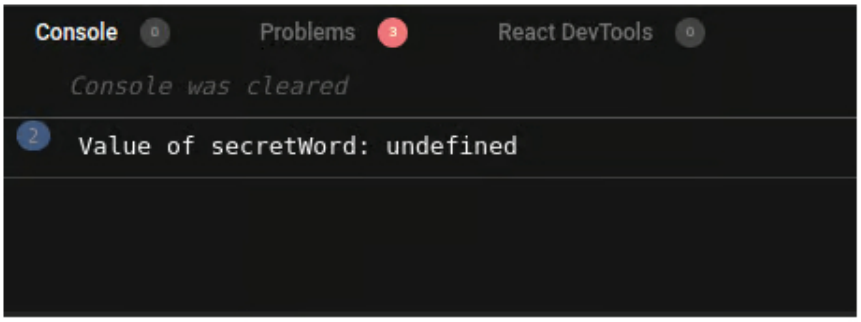
One important thing to note is that the process of passing down props to an HOC's child component is different than that of a non-HOC component.

For example, look at the following code:

```
function App() {
  return (
    <div>
      {/*Pass in a 'secretWord' prop*/}
      <HoverIncrease secretWord={"pineapple"} />
    </div>
  );
}
function HoverIncrease(props) {
  //read prop value:
  console.log("Value of secretWord: " + props.secretWord);
  //further code..
}
```

In theory, we should get the message `Value of secretWord: pineapple` in the

console. However, that's not the case here:



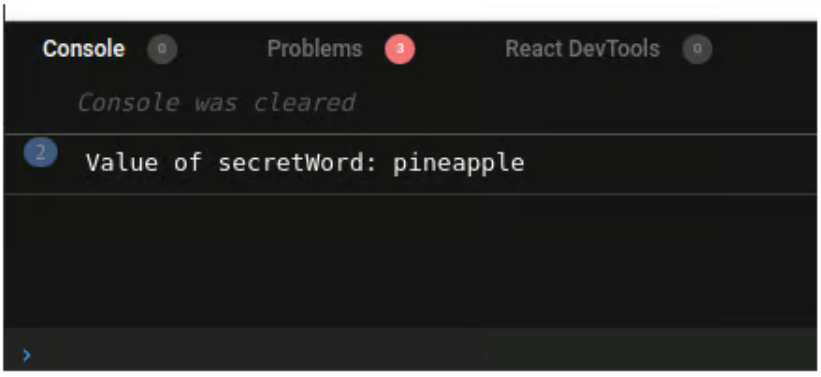
So what's happening here?

In this case, the `secretWord` prop is actually being passed to the `withCounter` function and not to the `HoverIncrease` component.

To solve this issue, we have to make a simple change to `withCounter.js`:

```
const UpdatedComponent = (OriginalComponent, increaseCount) => {
  function NewComponent(props) {
    return (
      <OriginalComponent
        //Pass down all incoming props to the HOC's children:
        {...props}
      />
    );
  }
  return NewComponent;
};
```

This minor fix solves our problem:



And we're done!

Conclusion

In this article, you learned the fundamentals of React's HOC concept. If you encountered any difficulties in this article, my suggestion to you is to deconstruct and play with the code samples above. This will help you understand the concept better.

Thank you so much for reading! Happy coding!