Higher-Order Components

In this lesson, you will learn how to implement higher-order components in React.

What are Higher-order components?

Higher-order components (HOC) are an advanced concept in React. HOCs are an equivalent to higher-order functions. They take any input, usually a component, but also optional arguments, and return a component as output. The returned component is an enhanced version of the input component, and it can be used in your JSX.

HOCs are used for different use cases. They can prepare properties, manage state, or alter the representation of a component. One case is to use a HOC as a helper for a conditional rendering. Imagine you have a List component that renders a list of items or nothing, because the list is empty or null. The HOC could shield away that the list would render nothing when there is no list. On the other hand, the plain List component doesn't need to bother anymore about an non existent list, as it only cares about rendering the list.

Basic example of an HOC

Let's do a simple HOC that takes a component as input and returns a component. Lets place it in the *App.js* file.

```
function withFoo(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

It is a useful convention to prefix a HOC with `with`. Since you are using JavaScript ES6, you can express the HOC better with an ES6 arrow function.

```
const withEnhancement = (Component) => (props) =>
  <Component { ...props } />
```

HOCs and conditional rendering: Loading component

In our example, the input component stays the same as the output, so nothing happens. The output component should show the Loading component when the loading state is true, otherwise it should show the input component. A conditional rendering is a great use case for an HOC.

```
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
```

Based on the loading property, you can apply a conditional rendering. The function will return the Loading component or the input component. In general, it can be very efficient to spread an object like the props object in the previous example as input for a component. See the difference in the following code snippet:

```
// before you would have to destructure the props before passing them
const { firstname, lastname } = props;
<UserProfile firstname={firstname} lastname={lastname} />

// but you can use the object spread operator to pass all object properties
<UserProfile { ...props } />
```

We passed all the props including the `isLoading` property by spreading the object into the input component. The input component may not care about the `isLoading` property. You can use the ES6 rest destructuring to avoid it:

```
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />
```

It takes one property out of the object, but keeps the remaining object, which also works with multiple properties. More can be read about it in Mozilla's destructuring assignment.

Now you can use the HOC in JSX. Maybe you want to show either the "More" button or the Loading component. The Loading component is already encapsulated in the HOC, but an input component is missing. For showing either a Button component or a Loading component, the Button is the input

either a Button component or a Loading component, the Button is the input

component of the HOC. The enhanced output component is a ButtonWithLoading component.

```
const Button = ({
  onClick,
  className = '',
  children,
}) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);
```

Everything is defined now. As a last step, you have to use the `ButtonWithLoading` component, which receives the loading state as an additional property. While the HOC consumes the loading property, all other props get passed to the Button component.

```
class App extends Component {

  ...

  render() {
    ...
    return (
      <div className="page">
        ...
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopStories(searchKey, page + 1)}
          >
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
```

```
}
```

Why Snapshot tests will fail now

Note that when you run your tests again, your snapshot test for the App component fails. The diff might look like the following on the command line:

```
  <button
     className=""
     onClick={[Function]}
     type="button"
   >
     More
   </button>
   <div>
     Loading ...
   </div>
```

You can either fix the component now, when you think there is something wrong about it, or can accept the new snapshot of it. Because you introduced the Loading component in this chapter, you can accept the altered snapshot test on the command line in the interactive test.

Higher-order components are an advanced pattern in React. They have multiple purposes: improved reusability of components, greater abstraction, composability of components, and manipulations of props, state and view. I encourage you to read gentle introduction to higher-order components. It gives you another approach to learn them, shows you an elegant way to use them in a functional programming way, and solves the problem of conditional rendering with higher-order components.

```
import React, { Component } from 'react';
require('./App.css');

const DEFAULT_QUERY = 'redux';
const DEFAULT_HPP = '100';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
```

```
    results: null,
    searchKey: '',
    searchTerm: DEFAULT_QUERY,

    error: null,
    isLoading: false,
  };

  this.needsToSearchTopstories = this.needsToSearchTopstories.bind(this);
  this.setSearchTopstories = this.setSearchTopstories.bind(this);
  this.fetchSearchTopstories = this.fetchSearchTopstories.bind(this);
  this.onSearchChange = this.onSearchChange.bind(this);
  this.onSearchSubmit = this.onSearchSubmit.bind(this);
  this.onDismiss = this.onDismiss.bind(this);
}

needsToSearchTopstories(searchTerm) {
  return !this.state.results[searchTerm];
}

setSearchTopstories(result) {
  const { hits, page } = result;
  const { searchKey, results } = this.state;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];

  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  });
}

fetchSearchTopstories(searchTerm, page = 0) {
  this.setState({ isLoading: true });

  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}&${PARAM_PAGE}${page}`)
    .then(response => response.json())
    .then(result => this.setSearchTopstories(result))
    .catch(e => this.setState({ error: e }));
}

componentDidMount() {
  const { searchTerm } = this.state;
  this.setState({ searchKey: searchTerm });
  this.fetchSearchTopstories(searchTerm);
}

onSearchChange(event) {
  this.setState({ searchTerm: event.target.value });
}

onSearchSubmit(event) {
  const { searchTerm } = this.state;
```

```
    this.setState({ searchKey: searchTerm });

    if (this.needsToSearchTopstories(searchTerm)) {
      this.fetchSearchTopstories(searchTerm);
    }

    event.preventDefault();
  }

  onDismiss(id) {
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];

    const isNotId = item => item.objectID !== id;
    const updatedHits = hits.filter(isNotId);

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      }
    });
  }

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];

    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
            onSubmit={this.onSearchSubmit}
          >
            Search
          </Search>
        </div>
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
```

```
            onDismiss={this.onDismiss}
          />
        }
        <div className="interactions">
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopstories(searchKey, page + 1)}>
            More
          </ButtonWithLoading>
        </div>
      </div>
    );
  }
}

const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) =>
  <form onSubmit={onSubmit}>
    <input
      type="text"
      value={value}
      onChange={onChange}
    />
    <button type="submit">
      {children}
    </button>
  </form>

const Table = ({ list, onDismiss }) =>
  <div className="table">
    { list.map(item =>
      <div key={item.objectID} className="table-row">
        <span style={{ width: '40%' }}>
          <a href={item.url}>{item.title}</a>
        </span>
        <span style={{ width: '30%' }}>
          {item.author}
        </span>
        <span style={{ width: '10%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '10%' }}>
          {item.points}
        </span>
        <span style={{ width: '10%' }}>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
      </div>
    )}
  </div>

const Button = ({ onClick, className = '', children }) =>
  <button
```

```
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading ? <Loading /> : <Component { ...rest } />

const ButtonWithLoading = withLoading(Button);

export default App;

export {
  Button,
  Search,
  Table,
};
```

Quick quiz on Higher Order Components in React!

Q

Can Higher Order Components be used to

COMPLETED 0%

1 of 1

Exercises:

- Experiment with the HOC you have created
- Think about a use case where another HOC would make sense
    - Implement the HOC, if there is a use case

Further Reading

- Read a gentle introduction to higher-order components