Higher-Order Components

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature.

Concretely, **a higher-order component is a function that takes a component and returns a new component.**

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

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as Redux's `connect` and Relay's `createFragmentContainer`.

In this document, we'll discuss why higher-order components are useful, and how to write your own.

Use HOCs For Cross-Cutting Concerns

> **Note**
> We previously recommended mixins as a way to handle cross-cutting concerns. We've since realized that mixins create more trouble than they are worth. Read more about why we've moved away from mixins and how you can transition your existing components.

patterns aren't a straightforward fit for traditional components.

For example, say you have a `CommentList` component that subscribes to an external data source to render a list of comments:

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // Clean up listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Update component state whenever the data source changes
    this.setState({
      comments: DataSource.getComments()
    });
  }

  render() {
    return (
      <div>
        {this.state.comments.map((comment) => (
          <Comment comment={comment} key={comment.id} />
        ))}
      </div>
    );
  }
}
```

pattern:

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

`CommentList` and `BlogPost` aren't identical — they call different methods on `DataSource`, and they render different output. But much of their implementation is the same:

- On mount, add a change listener to `DataSource`.
- Inside the listener, call `setState` whenever the data source changes.
- On unmount, remove the change listener.

You can imagine that in a large app, this same pattern of subscribing to `DataSource` and calling `setState` will occur over and over again. We want an abstraction that allows us to

order components excel.

We can write a function that creates components, like `CommentList` and `BlogPost`, that subscribe to `DataSource`. The function will accept as one of its arguments a child component that receives the subscribed data as a prop. Let's call the function `withSubscription`:

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

The first parameter is the wrapped component. The second parameter retrieves the data we're interested in, given a `DataSource` and the current props.

When `CommentListWithSubscription` and `BlogPostWithSubscription` are rendered, `CommentList` and `BlogPost` will be passed a `data` prop with the most current data retrieved from `DataSource`:

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      DataSource.addChangeListener(this.handleChange);
    }
```

```
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

Note that a HOC doesn't modify the input component, nor does it use inheritance to copy its behavior. Rather, a HOC *composes* the original component by *wrapping* it in a container component. A HOC is a pure function with zero side-effects.

And that's it! The wrapped component receives all the props of the container, along with a new prop, `data`, which it uses to render its output. The HOC isn't concerned with how or why the data is used, and the wrapped component isn't concerned with where the data came from.

Because `withSubscription` is a normal function, you can add as many or as few arguments as you like. For example, you may want to make the name of the `data` prop configurable, to further isolate the HOC from the wrapped component. Or you could accept an argument that configures `shouldComponentUpdate`, or one that configures the data source. These are all possible because the HOC has full control over how the component is defined.

Like components, the contract between `withSubscription` and the wrapped component is entirely props-based. This makes it easy to swap one HOC for a different one, as long as they provide the same props to the wrapped component. This may be useful if you change data-fetching libraries, for example.

Don't Mutate the Original Component. Use Composition.

HOC:

```
function logProps(InputComponent) {
  InputComponent.prototype.componentWillReceiveProps = function(nextProps) {
    console.log('Current props: ', this.props);
    console.log('Next props: ', nextProps);
  };
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);
```

There are a few problems with this. One is that the input component cannot be reused separately from the enhanced component. More crucially, if you apply another HOC to `EnhancedComponent` that *also* mutates `componentWillReceiveProps`, the first HOC's functionality will be overridden! This HOC also won't work with function components, which do not have lifecycle methods.

Mutating HOCs are a leaky abstraction—the consumer must know how they are implemented in order to avoid conflicts with other HOCs.

Instead of mutation, HOCs should use composition, by wrapping the input component in a container component:

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentWillReceiveProps(nextProps) {
      console.log('Current props: ', this.props);
      console.log('Next props: ', nextProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

clashes. It works equally well with class and function components. And because it's a pure function, it's composable with other HOCs, or even with itself.

You may have noticed similarities between HOCs and a pattern called **container components**. Container components are part of a strategy of separating responsibility between high-level and low-level concerns. Containers manage things like subscriptions and state, and pass props to components that handle things like rendering UI. HOCs use containers as part of their implementation. You can think of HOCs as parameterized container component definitions.

Convention: Pass Unrelated Props Through to the Wrapped Component

HOCs add features to a component. They shouldn't drastically alter its contract. It's expected that the component returned from a HOC has a similar interface to the wrapped component.

HOCs should pass through props that are unrelated to its specific concern. Most HOCs contain a render method that looks something like this:

```
render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

This convention helps ensure that HOCs are as flexible and reusable as possible.

Convention: Maximizing Composability

Not all HOCs look the same. Sometimes they accept only a single argument, the wrapped component:

```
const NavbarWithRouter = withRouter(Navbar);
```

Usually, HOCs accept additional arguments. In this example from Relay, a config object is used to specify a component's data dependencies:

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

The most common signature for HOCs looks like this:

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

What?! If you break it apart, it's easier to see what's going on.

```
// connect is a function that returns another function
const enhance = connect(commentListSelector, commentListActions);
// The returned function is a HOC, which returns a component that is connected
// to the Redux store
const ConnectedComment = enhance(CommentList);
```

In other words, `connect` is a higher-order function that returns a higher-order component!

This form may seem confusing or unnecessary, but it has a useful property. Single-argument HOCs like the one returned by the `connect` function have the signature `Component => Component`. Functions whose output type is the same as its input type are really easy to compose together.

```
// ... you can use a function composition utility
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))
const enhance = compose(
  // These are both single-argument HOCs
  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)
```

(This same property also allows `connect` and other enhancer-style HOCs to be used as decorators, an experimental JavaScript proposal.)

The `compose` utility function is provided by many third-party libraries including lodash (as `lodash.flowRight`), Redux, and Ramda.

Convention: Wrap the Display Name for Easy Debugging

The container components created by HOCs show up in the React Developer Tools like any other component. To ease debugging, choose a display name that communicates that it's the result of a HOC.

The most common technique is to wrap the display name of the wrapped component. So if your higher-order component is named `withSubscription`, and the wrapped component's display name is `CommentList`, use the display name `WithSubscription(CommentList)`:

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
`WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

Caveats

Higher-order components come with a few caveats that aren't immediately obvious if you're new to React.

Don't Use HOCs Inside the render Method

React's diffing algorithm (called reconciliation) uses component identity to determine whether it should update the existing subtree or throw it away and mount a new one. If the component returned from `render` is identical (`===`) to the component from the previous render, React recursively updates the subtree by diffing it with the new one. If they're not equal, the previous subtree is unmounted completely.

Normally, you shouldn't need to think about this. But it matters for HOCs because it means you can't apply a HOC to a component within the render method of a component:

```
render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}
```

The problem here isn't just about performance — remounting a component causes the state of that component and all of its children to be lost.

Instead, apply HOCs outside the component definition so that the resulting component is created only once. Then, its identity will be consistent across renders. This is usually what you want, anyway.

In those rare cases where you need to apply a HOC dynamically, you can also do it inside a component's lifecycle methods or its constructor.

Sometimes it's useful to define a static method on a React component. For example, Relay containers expose a static method `getFragment` to facilitate the composition of GraphQL fragments.

When you apply a HOC to a component, though, the original component is wrapped with a container component. That means the new component does not have any of the static methods of the original component.

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

To solve this, you could copy the methods onto the container before returning it:

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

However, this requires you to know exactly which methods need to be copied. You can use hoist-non-react-statics to automatically copy all non-React static methods:

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

```
// Instead of...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

Refs Aren't Passed Through

While the convention for higher-order components is to pass through all props to the wrapped component, this does not work for refs. That's because `ref` is not really a prop — like `key`, it's handled specially by React. If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component.

The solution for this problem is to use the `React.forwardRef` API (introduced with React 16.3). Learn more about it in the forwarding refs section.

Edit this page

Contributing
FAQ

Facebook
Twitter

COMMUNITY

Community Resources
Tools

MORE

Tutorial
Blog
Acknowledgements
React Native