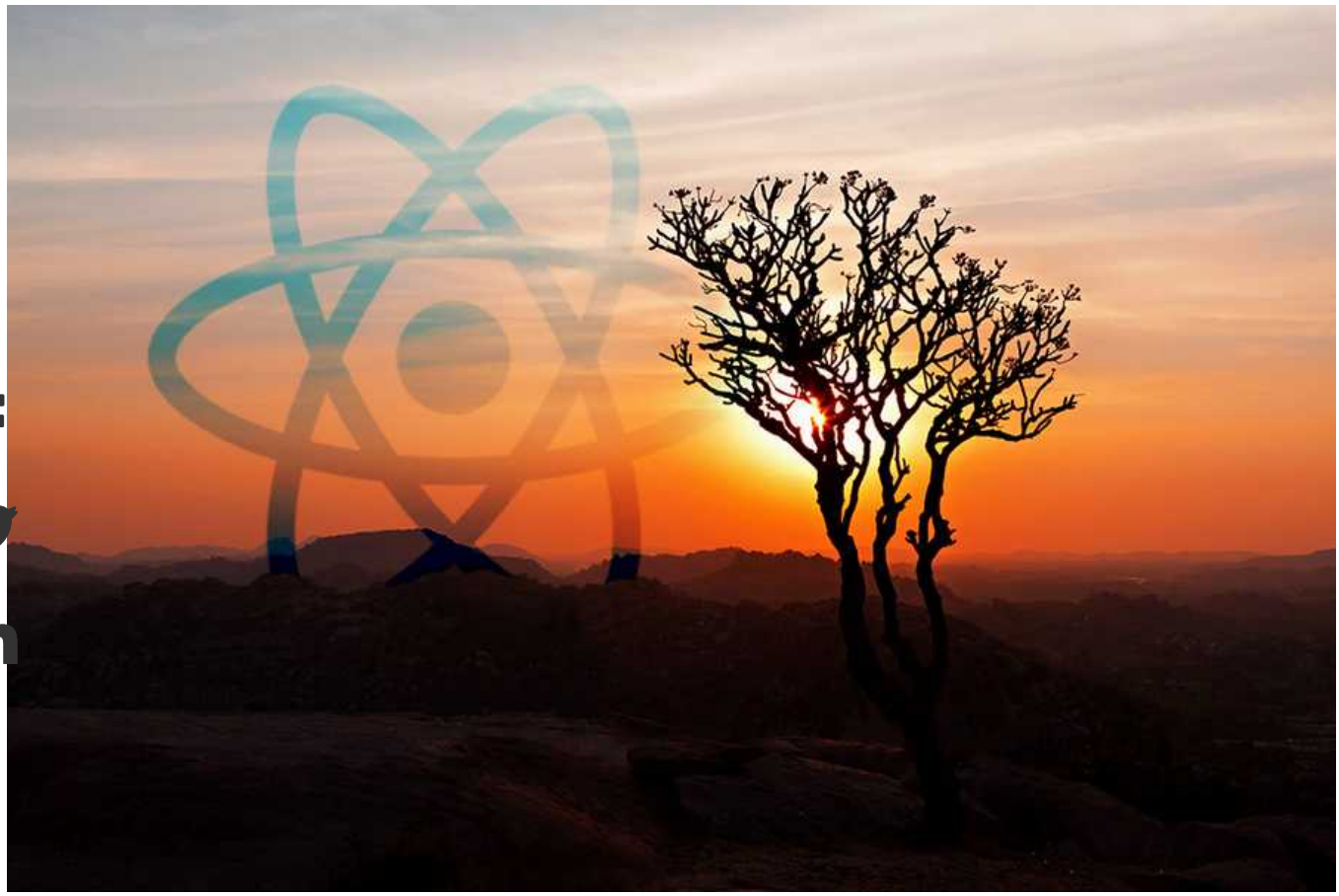
Why React Hooks over HOCs

OCTOBER 01, 2020 BY ROBIN WIERUCH - EDIT THIS POST



In a modern React world, everyone uses function components with React Hooks. However, the concept of higher-order components (HOCs) is still applicable in a modern React world, because they can be used for class components and function components. Therefore they are the perfect bridge for using reusable abstractions among legacy and modern React components.

I am still an advocate for higher-order components these days because their composable nature of enhancing components fascinates me. However, there are problems with HOCs which shouldn't be denied and which are entirely solved by React Hooks. This is why I want to point out these problems, so that developers can make an informed decision whether they want to use an HOC over an Hook for certain scenarios or whether they just want to go all-in with React Hooks after all.

HOCS VS HOOKS: PROP CONFUSION

Let's take the following higher-order component (HOC) which is used for a conditional rendering. If there is an error, it renders an error message. If there is no error, it renders the *given component*:

```
import * as React from 'react';

const withError = (Component) => (props) => {
  if (props.error) {
    return <div>Something went wrong ...</div>;
  }

  return <Component {...props} />;
};

export default withError;
```

Note how the HOC passes all the props to the given component if there is no error. Everything should be working fine this way, however, there *may* be too many props passed to the next component which isn't necessarily concerned about all of them.

For example, it could be that the next component doesn't care at all about the error, thus it would be a better practice to remove the error with a rest operator from the props before forwarding the props to the next component:

```
import * as React from 'react';

const withError = (Component) => ({ error, ...rest }) => {
  if (error) {
    return <div>Something went wrong ...</div>;
  }

  return <Component {...rest} />;
};

export default withError;
```

This version should work as well, at least if the given component doesn't need the error prop. However, these both versions of a HOC already show the surfacing problem of prop confusion when using HOCs. Most often props are just passed through HOCs by using the spread operator and only partly used in the HOC itself. Often it isn't clear from the start whether the given component needs all the props provided to the HOC (first version) or is just fine with only a part of the props (second version).

That's the first caveat of using a HOC; which gets quickly unpredictable when using multiple HOCs which are composed onto each other, because then one has not only to consider what props are needed for the given component, but also what props are needed for the other HOCs in the composition. For example, let's say we have another HOC for rendering a conditional loading indicator:

```
import * as React from 'react';

const withLoading = (Component) => ({ isLoading, ...rest }) => {
  if (isLoading) {
    return <div>Loading ...</div>;
  }

  return <Component {...rest} />;
};

export default withLoading;
```

Both HOCs, `withError` and `withLoading` are composed on a component now. Once this component is used, it may look like the following:

```
const DataTableWithFeedback = compose(
  withError,
  withLoading,
)(DataTable);

const App = () => {
  ...

  return (
    <DataTableWithFeedback
      columns={columns}
      data={data}
      error={error}
      isLoading={isLoading}
    />
  );
};
```

Without knowing the implementation details of the HOCs, would you know which props are consumed by the HOCs and which are dedicated the the underlying component? It's not clear which props are really passed through to the actual DataTable component and which props are consumed by HOCs on the way.

Let's take this example one step further, by introducing another HOC for data fetching where we don't show the implementation details:

```
const DataTableWithFeedback = compose(
  withFetch,
  withError,
  withLoading,
)(DataTable);

const App = () => {
  ...

  const url = 'https://api.mydomain/mydata';

  return (
    <DataTableWithFeedback
      url={url}
      columns={columns}
    />
  );
};
```

Suddenly we don't need `data`, `isLoading`, and `error` anymore, because all this information is generated in the new `withFetch` HOC by using the `url`. What's interesting though is that `isLoading` and `error`, while generated inside the `withFetch` HOC, will already be consumed on the way by `withLoading` and `withError`. On the other hand, the generated (here fetched) `data` from `withFetch` will be passed as prop to the underlying DataTable component.

```
App     withFetch   withError   withLoading   DataTable

        data->      data->      data->        data
url->   error->     error
        isLoading-> isLoading-> isLoading
```

In addition to all of this hidden magic, see how order matters too: `withFetch` needs to be the outer HOC while `withLoading` and `withError` follow without any particular order here which gives lots of room for bugs.

In conclusion, all these props coming in and out from HOCs travel somehow through a blackbox which we need to examine with a closer look to really understand which props are produced on the way, which props are consumed on the way, and which props get passed through. Without looking into the HOCs, we don't know much about what happens between these layers.

Finally, in comparison, let's see how React Hooks solve this issue with one -- easy to understand from a usage perspective -- code snippet:

```
const App = () => {
  const url = 'https://api.mydomain/mydata';
  const { data, isLoading, error } = useFetch(url);
```

```
  if (error) {
    return <div>Something went wrong ...</div>;
  }

  if (isLoading) {
    return <div>Loading ...</div>;
  }

  return (
    <DataTable
      columns={columns}
      data={data}
    />
  );
};
```

When using React Hooks, everything is laid out for us: We see all the props (here `url`) that are going into our "blackbox" (here `useFetch`) and all the props that are coming out from it (here `data`, `isLoading`, `error`). Even though we don't know the implementation details of `useFetch`, we clearly see which input goes in and which output comes out. And even though `useFetch` can be treated as a blackbox like `withFetch` and the other HOCs, we see the whole API contract with this React Hook in just one plain line of code.

This wasn't as clear with HOCs before, because we didn't clearly see which props were needed (input) and which props were produced (output). In addition, there are not other HTML layers in between, because we just use the conditional rendering in the parent (or in the child) component.

HOCS VS HOOKS: NAME CONFLICTS/COLLISION

If you give a component a prop with the same name two times, the latter will override the former:

```
<Headline text="Hello World" text="Hello React" />
```

When using a plain component like in the previous example, this issue gets quite obvious and we are less likely to override props accidentally (and only on purpose if we need to). However, with HOCs this gets messy again when two HOCs pass props with the same name.

The easiest illustration for this problem is by composing two identical HOCs on top of a component:

```
const UserWithData = compose(
  withFetch,
  withFetch,
```

```
  withError,
  withLoading,
)(User);

const App = () => {
  ...

  const userId = '1';

  return (
    <UserWithData
      url={`https://api.mydomain/user/${userId}`}
      url={`https://api.mydomain/user/${userId}/profile`}
    />
  );
};
```

This is a very common scenario; often components need to fetch from multiple API endpoints.

As we have learned before, the `withFetch` HOC expects an `url` prop for the data fetching. Now we want to use this HOC two times and thus we are not able anymore fulfil both HOCs contract. In contrast, both HOCs will just operate on the latter URL which will lead to a problem. A solution (and yes, there is more than one solution) to this problem would be changing our `withFetch` HOC to something more powerful in order to perform not a single but multiple requests:

```
const UserWithData = compose(
  withFetch,
  withError,
  withLoading,
)(User);

const App = () => {
  ...

  const userId = '1';

  return (
    <UserWithData
      urls={[
        `https://api.mydomain/user/${userId}`,
        `https://api.mydomain/user/${userId}/profile`,
      ]}
    />
  );
};
```

This solution seems plausible, but let's let this sink in for a moment: The `withFetch` HOC, previously just concerned about one data fetching -- which based on this one data fetching sets states for

`isLoading` and `error` -- suddenly becomes a monster of complexity. There are many questions to answer here:

- Does the loading indicator still show up even though one of the requests finished earlier?
- Does the whole component render as an error if only one request fails?
- What happens if one request depends on another request?
- ...

Despite of this making the HOC already a super complex (yet powerful) HOC -- where my personal gut would tell me it's too powerful -- we introduced another problem internally. Not only did we have the problem of passing a duplicated prop (here `url`, which we solved with `urls`) to the HOC, but also the HOC will output a duplicate prop (here `data`) and pass it to the underlying component.

That's why, in this case the User component has to receive a merged `data` props -- the information from both data fetches -- or has to receive an array of data -- whereas the first entry is set accordingly to the first URL and the second entry accordingly to the second URL. In addition, when both requests don't fulfil in parallel, one data entry can be empty while the other one is already there ...

Okay. I don't want to go any further fixing this here. There are solutions to this, but as I mentioned earlier, it would lead to making the `withFetch` HOC more complicated than it should be and the situation of how to use the merged `data` or `data` array in the underlying component not much better from a developer's experience perspective.

Let's see how React Hooks solve this for us with one -- easy to understand from a usage perspective -- code snippet again:

```
const App = () => {
  const userId = '1';

  const {
    data: userData,
    isLoading: userIsLoading,
    error: userError
  } = useFetch(`https://api.mydomain/user/${userId}`);

  const {
    data: userProfileData,
    isLoading: userProfileIsLoading,
    error: userProfileError
  } = useFetch(`https://api.mydomain/user/${userId}/profile`);

  if (userError || userProfileError) {
    return <div>Something went wrong ...</div>;
  }

  if (userIsLoading) {
    return <div>User is loading ...</div>;
```

```
  }

  const userProfile = userProfileIsLoading
    ? <div>User profile is loading ...</div>
    : <UserProfile userProfile={userProfileData} />;

  return (
    <User
      user={userData}>
      userProfile={userProfile}
    />
  );
};
```

Do you see the flexibility we gain here? We only return early with an loading indicator if the user is still loading, however, if the user is already there and only the user profile is pending, we are only partially rendering a loading indicator where the data is missing (here also due to the power of component composition). We could do the same for the error, however, because we gained all this power over how to deal with the outcome of the requests, we can render the same error message for both errors. If we later decide we want to deal with both errors differently, we can do this in this one component and not in our abstraction (whether it's HOC or Hook).

After all, and that's why we come to this conclusion in the first place, we avoided the naming collision by renaming the variables which comes as output from the React Hooks within the object destructuring. When using HOCs, we need to be aware of HOCs maybe using the same names for props internally. It's often obvious when using the same HOC twice, but what happens if you are using two different HOCs which -- just by accident -- use the same naming for a prop? They would override each others data and leave you baffled why your receiving component doesn't get the correct props.

HOCS VS HOOKS: DEPENDENCIES

HOCs are powerful, perhaps too powerful? HOCs *can* receive arguments two ways: When they receive props from the parent component (as we have seen before) and when they enhance a component. Let's elaborate the latter by example.

Take our `withLoading` and `withError` HOCs from before but this time more powerful:

```
const withLoading = ({ loadingText }) => (Component) => ({ isLoading, ...res
  if (isLoading) {
    return <div>{loadingText ? loadingText : 'Loading ...'}</div>;
  }

  return <Component {...rest} />;
```

```
};

const withError = ({ errorText }) => (Component) => ({ error, ...rest }) =>
  if (error) {
    return <div>{errorText ? errorText : 'Something went wrong ...'}</div>;
  }

  return <Component {...rest} />;
};
```

With these extra arguments -- here passed through a higher-order function surrounding the HOC -- we gain additional power to provide arguments when creating the enhanced component with our HOCs:

```
const DataTableWithFeedback = compose(
  withError({ errorText: 'The data did not load' }),
  withLoading({ loadingText: 'The data is loading ...' }),
)(DataTable);

const App = () => {
  ...

  return (
    <DataTableWithFeedback
      columns={columns}
      data={data}
      error={error}
      isLoading={isLoading}
    />
  );
};
```

This contributes an (1) positive and (2) negative effect to the Prop Confusion problem from before, because now we have (2) more than one place from where the HOC receives props (which doesn't make things easier to understand), but then again (1) we can avoid the implicit prop passing from the parent component (where we don't know whether this prop is consumed by the HOC or the underlying component) and try to pass props from the very beginning when enhancing the component instead.

However, in the end, these arguments (here the objects with `errorText` and `loadingText`) passed when enhancing the component are static. We are not able to interpolate them with any props from the parent component here, because we are creating the composed component outside of any component. For instance, in the data fetching example we wouldn't be able to introduce a flexible user ID:

```
const UserWithData = compose(
```

```
  withFetch('https://api.mydomain/user/1'),
  withFetch('https://api.mydomain/user/1/profile'),
)(User);

const App = () => {
  ...

  return (
    <UserWithData
      columns={columns}
    />
  );
};
```

Even though there are ways to overcome this, it doesn't make this whole props passing any more easier to understand:

```
const UserWithData = compose(
  withFetch(props => `https://api.mydomain/user/${props.userId}`),
  withFetch(props => `https://api.mydomain/user/${props.userId}/profile`),
)(User);

const App = () => {
  ...

  const userId = '1';

  return (
    <UserWithData
      userId={userId}
      columns={columns}
    />
  );
};
```

Making this scenario even more complex by adding another challenge: What happens if the second request depends on the first request? For instance, the first request returns an user by ID and the second request returns a user's profile based on the `profileId` which we only get with the first request:

```
const UserProfileWithData = compose(
  withFetch(props => `https://api.mydomain/users/${props.userId}`),
  withFetch(props => `https://api.mydomain/profile/${props.profileId}`),
)(UserProfile);

const App = () => {
  ...

  const userId = '1';
```

```
  return (
    <UserProfileWithData
      columns={columns}
      userId={userId}
    />
  );
};
```

We introduced two HOCs which are tightly coupled here. In another solution, we may have created one powerful HOC to solve this for us. However, this shows us that it's difficult to create HOCs which depend on each other.

In contrast, let's see how this mess is solved by React Hooks again:

```
const App = () => {
  const userId = '1';

  const {
    data: userData,
    isLoading: userIsLoading,
    error: userError
  } = useFetch(`https://api.mydomain/user/${userId}`);

  const profileId = userData?.profileId;

  const {
    data: userProfileData,
    isLoading: userProfileIsLoading,
    error: userProfileError
  } = useFetch(`https://api.mydomain/user/${profileId}/profile`);

  if (userError || userProfileError) {
    return <div>Something went wrong ...</div>;
  }

  if (userIsLoading || userProfileIsLoading) {
    return <div>Is loading ...</div>;
  }

  return (
    <User
      user={userData}>
      userProfile={userProfileData}
    />
  );
};
```

Because React Hooks can be used directly in a function component, they can build up onto each other and it's straightforward to pass data from one hook to another hook if they depend on each other. There is also no real blackbox again, because we can clearly see which information needs to be passed

to these custom hooks and which information comes out from them. When using React Hooks that depend on each other, the dependencies are more explicit compared to using HOCs.

. . .

In the aftermath, I am still a big fan of HOCs for shielding away complexity from components (e.g. conditional rendering, protected routes). But as these last scenarios have shown, they are not always the best solution. Hence my recommendation would be using React Hooks instead.

Show Comments

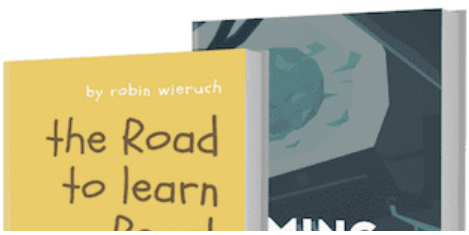
KEEP READING ABOUT REACT›

REACT HIGHER ORDER COMPONENTS

Another fitting headline for the article could be: learn Higher Order Components with Conditional Rendering in React. Higher order components, or known under the abbreviation HOCs, are often a...

REACT HOOKS MIGRATION

React Hooks were introduced to React to make state and side-effects available in React Function Components. Before it was only possible to have these in React Class Components; but since React's way...



THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers**.

GET THE BOOK >

Get it on Amazon.

TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✔ Join 50.000+ Developers
- ✔ Learn Web Development with JavaScript
- ✔ Tips and Tricks
- ✔ Access Tutorials, eBooks and Courses
- ✔ Personal Development as a Software Engineer

Your email address SUBSCRIBE