React – Props & HOCS

Children Props

Composition vs Inheritance

- React has a powerful composition model, and we recommend using composition instead of inheritance to reuse code between components.

Composition vs Inheritance

- Some components don't know their children ahead of time. This is especially common for components like Sidebar or Dialog that represent generic "boxes".
- Such components may use the special `children` prop to pass children elements directly into their output

Composition vs Inheritance

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

Composition vs Inheritance

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

Composition vs Inheritance

- Anything inside the <FancyBorder> JSX tag gets passed into the FancyBorder component as a children prop. Since FancyBorder renders {props.children} inside a , the passed elements appear in the final output.

Composition vs Inheritance

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

Composition vs Inheritance

- React elements like <Contacts /> and <Chat /> are just objects, so you can pass them as props like any other data. This approach may remind you of “slots” in other libraries but there are no limitations on what you can pass as props in React.

Composition vs Inheritance

- Props and composition give you all the flexibility you need to customize a component's look and behavior in an explicit and safe way. Remember that components may accept arbitrary props, including primitive values, React elements, or functions.

Rest Props

Rest Props

```
const PrivateRoute = ({component: Component, ...rest}) => {
  return (
    <Route {...rest} render={props => (
      isLogin()
        ? <Component {...props} />
        : <Redirect to="/signin" />
    )} />
  );

}
```

Rest Props

- We destructure the props object of PrivateRoute component:
- We find a property name component and rename it to Component (capitalize).
- The rest of the properties (if any), we store them to a variable name rest (hence the name). You can name it whatever you want e.g. ...whatever

Rest Props

- Later on, in the return statement, we spread the remaining properties to React RouterRoute component.

Rest Props

When using the `PrivateRoute`:

```
<PrivateRoute component={Dashboard} path="/dashboard" exact />
```

`…rest` would be

```
{path: "/dashboard", exact: true}
```

Rest Props

- https://gist.github.com/at0g/6b6df72af556ff46e227

HOCS

What is a Higher Order Component?

- 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);
```

How it works?

- https://v5.reactrouter.com/core/api/withRouter
- https://reactjs.org/docs/higher-order-components.html

How it works?

- **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}
    />
  );
}
```

How it works?

- **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:

Further reading

Further reading

- https://reactjs.org/docs/higher-order-components.html
- https://www.codingame.com/playgrounds/8595/reactjs-higher-order-components-tutorial
- https://levelup.gitconnected.com/real-world-examples-of-higher-order-components-hoc-for-react-871f0d8b39d8
- https://www.smashingmagazine.com/2020/06/higher-order-components-react/

- https://blog.jakoblind.no/simple-explanation-of-higher-order-components-hoc/ (! – as simple as it can get)

Assignment

Assignment

https://levelup.gitconnected.com/real-world-examples-of-higher-order-components-hoc-for-react-871f0d8b39d8

1. Create a view/edit toggle component with HOC (you can name it withToggle)
2. Create an expand/collapsed component with HOC (you can name it withToggle)