Composition of ReactElement

* Composition of ReactComponents

In React, we can make components more generic by accepting props, which are to React components what parameters are to functions.

Component composition is the name for passing components as props to other components, thus creating new components with other components.

const Button = ({ onClick, children }) => (

<button onClick={onClick}>{children}</button>

);

const App = () => {

const onClick = () => alert('Hey 👋');

return (

<Button onClick={onClick}>Click me!</Button>

);

};

children is nothing more than a prop to the Button component.

Instead of passing down a string to Button, we may want to add an icon to the text as well:

const App = () => {

const onClick = () => alert('Hey 👋');

return (

<Button onClick={onClick}>

<img src="/logos/logo.svg" /> Click me!

</Button>

);

};

But we're not limited to the children prop. We can create more specific props that can accept components as well:

const Button = ({ onClick, icon, children }) => (

<button onClick={onClick}>{icon}{children}</button>

);

const App = () => {

const onClick = () => alert('Hey 👋');

return (

<Button

onClick={onClick}

icon={<img src="/logos/logo.svg" />} >

Click me!

</Button>

);

};

And that is the essence of component composition: a simple yet incredibly powerful pattern that makes React components highly reusable.

When working on React projects, you'll continuously find yourself refactoring components to be more generic through component composition so that you can use them in multiple places.

How can composition help prop drilling?

Prop drilling is the act of passing props through multiple layers of components.

Here's an example where we are passing userName through multiple layers:

const App = () => {

const userName = 'Joe';

return (

<WelcomePage userName={userName} />

);

}

const WelcomePage = ({ userName }) => {

return (

<>

<WelcomeMessage userName={userName} />

{/\*\* Some other welcome page code \*/}

</>

);

}

const WelcomeMessage = ({ userName }) => {

return (

<h1>Hey, {userName}!</h1>

);

}

Here's a visualization of how the components are structured:



App passes userName to WelcomePage, and WelcomePage passes userName to WelcomeMessage.

With only a few layers, this isn't a big deal, but this can quickly get out of hand in larger applications.

The easiest solution? Component composition!

Instead of passing userName through all these layers, we can try to compose the components at a higher level, like the App component.

const App = () => {

const userName = 'Joe';

return (

<WelcomePage title={<WelcomeMessage userName={userName} />} />

);

}

const WelcomePage = ({ title }) => {

return (

<>

{title}

{/\*\* Some other welcome page code \*/}

</>

);

}

const WelcomeMessage = ({ userName }) => {

return (

<h1>Hey, {userName}!</h1>

);

}

We can now see that only App imports all the components. We successfully removed one layer and avoided prop drilling.



Our app, after removing one layer with component composition. We don't need to pass down userName twice anymore—we just pass it to WelcomeMessage right away.

Now, I don't say that either of these examples is bad code and that prop drilling should be avoided at all costs.

However, it's a useful pattern to be aware of if prop-drilling becomes an issue.

How can composition help performance?

Composition is also a great ally if you try to reduce the number of re-renders in your application.

Let's say you have a Post component that displays the scroll progress. It updates the state base on the scroll event:

const Post = () => {

const [progress, setProgress] = React.useState(0);

React.useEffect(() => {

const scrollListener = () => {

// update the progress based on the scroll position

}

window.addEventListener('scroll', scrollListener, false);

}, [])

return (

<>

<h2 className="progress">

Progress: {progress}%

</h2>

<div className="content">

<h1>Content Title</h1>

{/\*\* more content \*/}

</div>

</>

)

}

Here's the [link to the Codepen](https://codepen.io/fgerschau/pen/PoEvmMq) if you want to see the details.

This code will cause a lot of re-renders, and we can assume that the blog post content contains a lot more components—so re-renders will be expensive.

If we move the logic to a separate component and use component composition to glue them together, the number of re-renders goes from 61 (on my computer) to 1 for the content section of the Post component.

All I did was moving the state updates to PostLayout and rendering the post content as a prop.

const PostLayout = ({ children }) => {

const [progress, setProgress] = React.useState(0);

React.useEffect(() => {

const scrollListener = () => {

// update the progress based on the scroll position

}

window.addEventListener('scroll', scrollListener, false);

}, []);

return (

<>

<h2 className="progress">

Progress: {progress}%

</h2>

{children}

</>

);

};

const Post = () => {

return (

<PostLayout>

<div className="content">

<h1>Content Title</h1>

{/\*\* more content \*/}

</div>

</PostLayout>

);

};

Here's the [link to the Codepen](https://codepen.io/fgerschau/pen/QWaRgKg) for the optimized version.

Why is it that the content only renders once in this case?

The reason is that React renders props only if they change.

As a refresher on re-renders: React components re-render when their state or props change.

[Here can read more about how React re-renders components](https://felixgerschau.com/react-rerender-components/) in detail.

So looking at PostLayout again, children doesn't re-render because it's a prop that hasn't changed.

<>

<h2 className="progress">

Progress: {progress}%

</h2>

{children}

</>