React Component Composition Explained

react

Tweet Copy link

Component composition is a powerful pattern to make your components more reusable.

If you are already familiar with React, you're probably already using it (maybe without knowing its name).

This article will explain what component composition means and **how to use it**. After that, we'll see how we can use component composition to **improve performance** and **avoid prop drilling**.

What is component composition in React?

In React, we can make components more generic by accepting **props**, which are to

Scroll up

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 }) => (
```

Scroll up

```
  <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} />
  );
}
```

Scroll up

```
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 c

Scroll up

components at a higher level, like the `App` component.

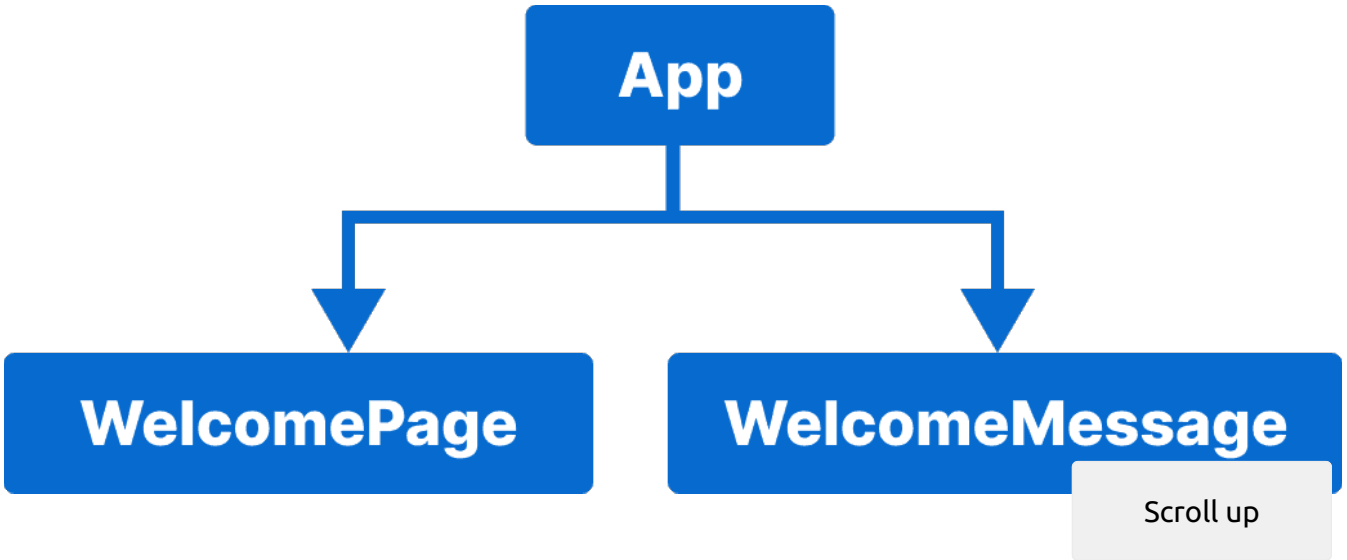
```
const App = () => {
  const userName = 'Joe';

  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>
    </>
  )
```

Scroll up

```
}
```

Here's the link to the Codepen 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 */}
```

Scroll up

```
      </div>
    </PostLayout>
  );
};
```

Here's the link to the Codepen 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** 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}
</>
```

Only the highlighted sections re-render.

Conclusion

As you can see, re-organizing your components can be incredibly powerful to make your code more organized, avoid prop drilling, and even improve performance.

Here are some other articles you may find helpful:

Scroll up

- **When does React re-render components?**
  If the current article got you interested in performance, this one goes much more into detail about how re-renders in React work under the hood.
- **How to use React.memo() to improve performance**
  React memo is another way of improving performance. While it certainly has its place, I would strongly advise trying using better component composition to reduce re-renders instead.
- **JavaScript's Memory Management Explained**
  When working on React applications for the browser, it's good to understand what happens behind the scenes. In this article, I explain how JavaScript manages memory for us.

Did you find this article helpful? You can subscribe to my email list in the footer to get updates on content about React and JavaScript 👇

🌎 Global 💰 $50-300K

MineCryptoGG

Frontend Developer (Next/React)

🇪🇺 EU

CodeSandbox

Senior Frontend Engineer (React)

🌎 Global

ResearchRabbit

Founding Frontend Engineer (React/Redux)

🇺🇸 US 💰 $125-175K 0.5-1.5%

More positions