

The Things They Didn't Teach You in React



LogRocket

Table of Contents

3

Introduction

4

Death by a Thousand Cuts:
A Checklist for Eliminating Common
React Performance Issues

28

Five Common Practices That
You Can Stop Doing in React

43

Modern Component Reusability:
Render Props in React



Death by a Thousand Cuts: A Checklist for Eliminating Common React Performance Issues

By Ohans Emmanuel • FE Engineer

Have you ever wondered how to make your React applications faster?

Yes?

How about having a checklist for eliminating common React performance issues?

Well, you are in the right place.

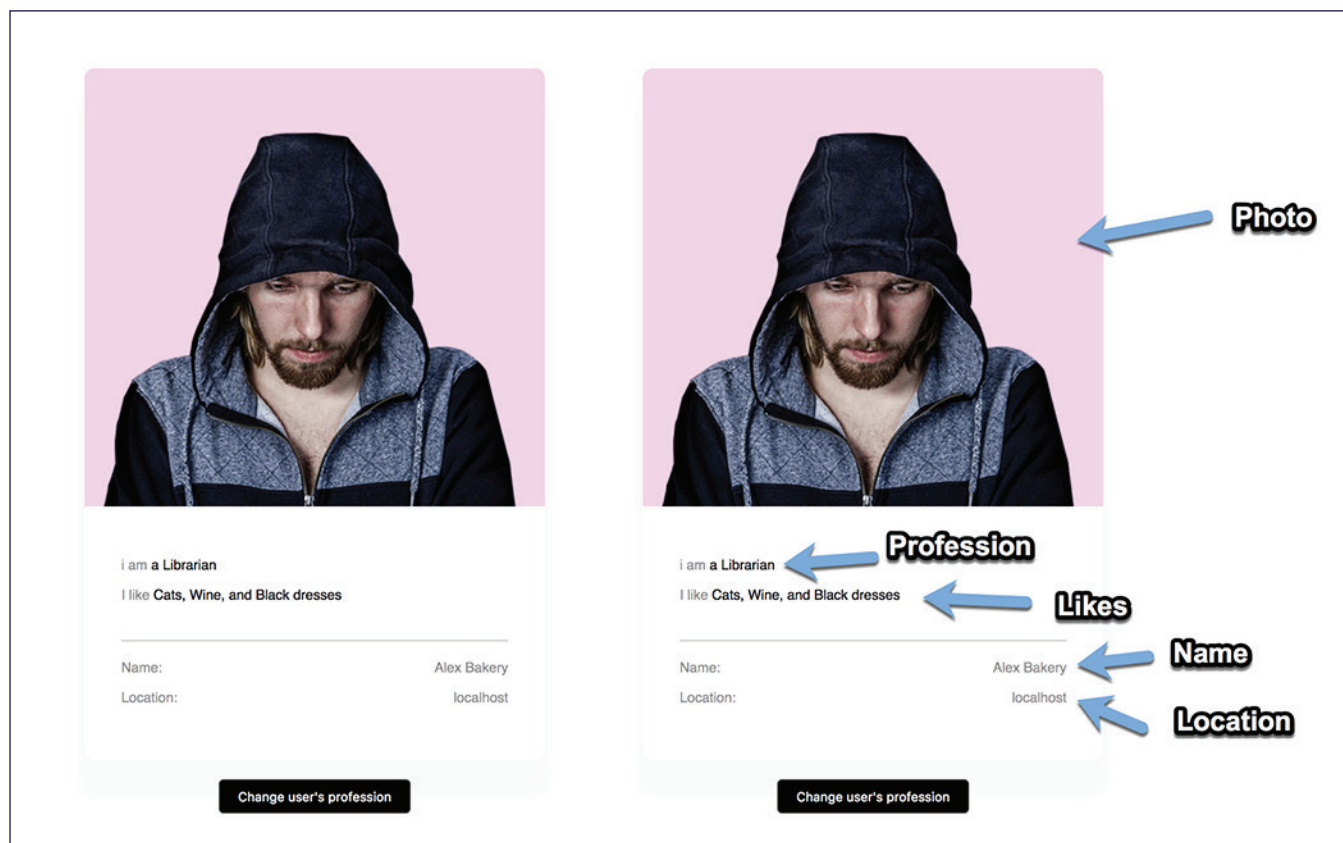
In this article, I'll walk you through a pragmatic step-by-step guide to eliminating common react performance issues.

First, I'll show you the problems, and then provide solutions to them. In doing this, you'll be able to carry over the same concepts to your real-world projects. The goal of this article is not to be a lengthy essay, rather, I'll discuss quick tips you can start using in your applications today. Let's get started!

The Sample Project

To make this article as pragmatic as possible, I'll walk you through various use cases while working on a single React application.

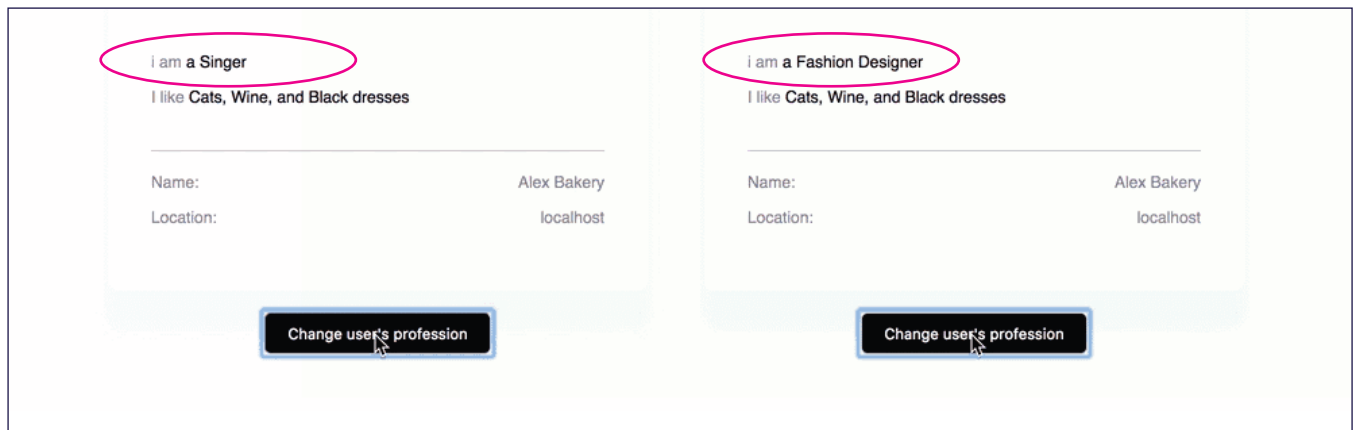
I call this app Cardie.



Here's the [Github repo](#) if you want to follow along.

Cardie is a simple application. All it does is display the profile information of a user. Commonly known as a user profile card.

It also includes the added functionality of being able to change the user's profession with a button click.



Upon clicking the button at the bottom of the app, the user's profession is changed.

While you may be quick to laugh this off as a simple application, and nowhere near a real-world app, you may be surprised to know that the knowledge gained from hunting down performance issues with this example application applies to whatever real-world app you build.

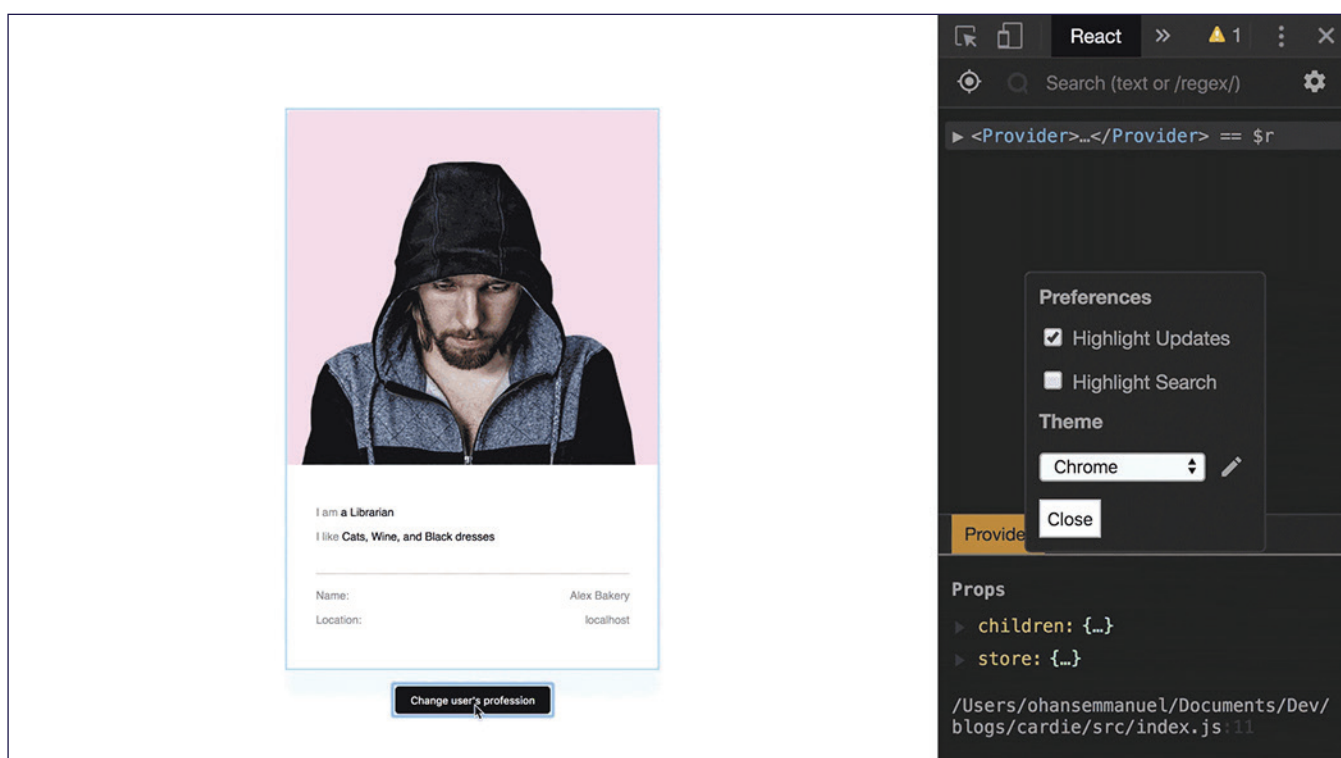
So, stay calm and enjoy the article.

Here comes the checklist!

1. Identify wasted renders

Identifying wasted renders in your react applications is the perfect start to identifying most performance issues.

There are a couple different ways to approach this, but the simplest method is to toggle on the “highlight updates” option in the React dev tools.



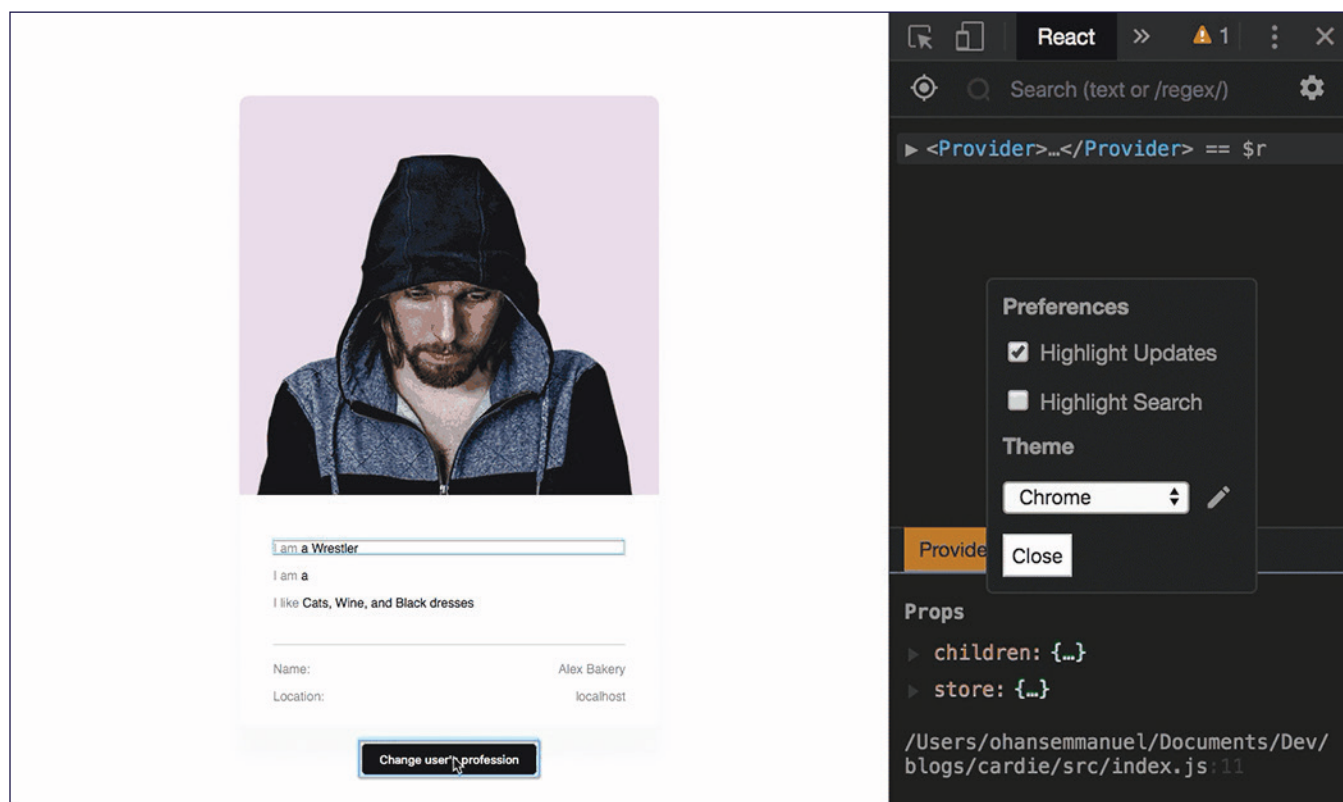
Note the green flash around the user card

While interacting with the app, updates are highlighted on the screen with a green flash.

The green flash shows the components in the app that are re-rendered by React under the hood.

With Cardie, upon changing the user's profession, it appears that the entire parent component **App** is re-rendered.

A more ideal highlighted update should look like this:



Note how the highlighted update is contained within the small updated region.

In more complex applications, the impact of a wasted render may be huge! The re-rendered component may be large enough to promote performance concerns.

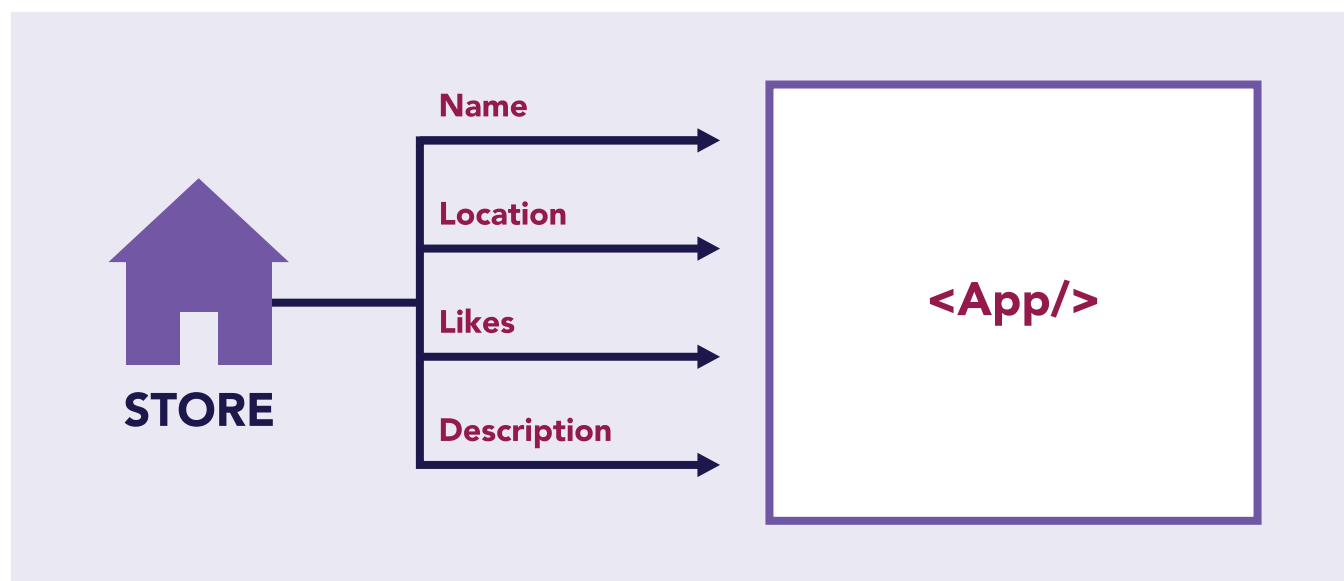
Having seen the problem, are there any solutions to this?

2. Extract frequently updated regions into isolated components

Once you've visually noted wasted renders in your application, a good place to start is to attempt to break up your component tree to support regions updated frequently.

Let me show you what I mean.

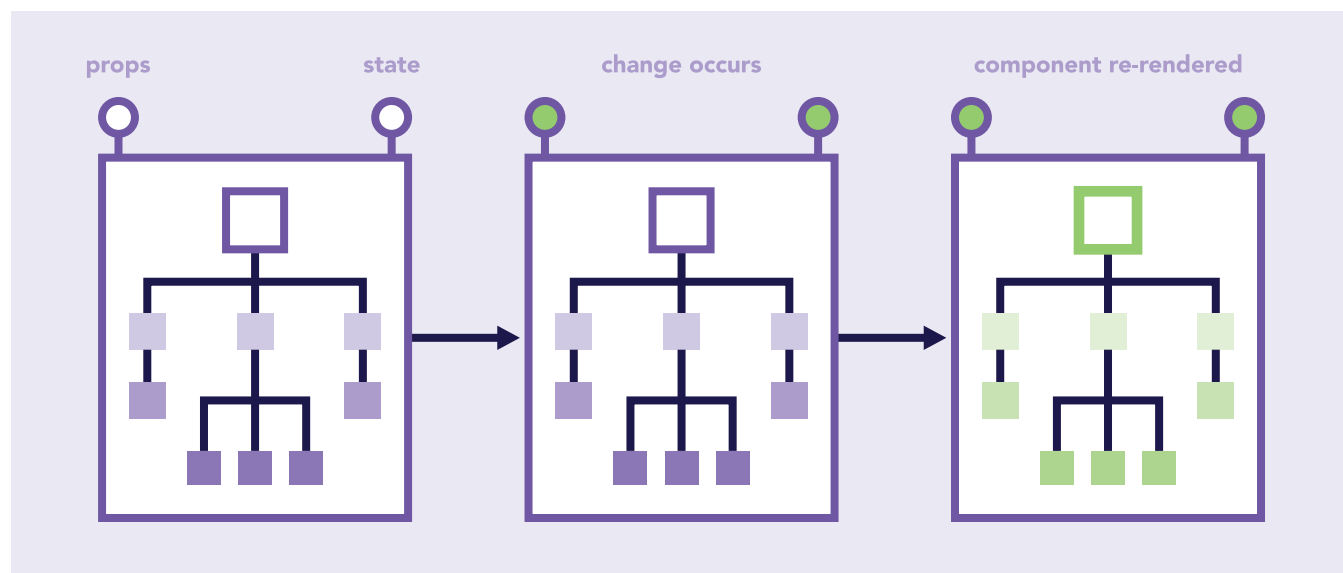
In Cardie, the `App` component is connected to the redux store via the `connect` function from `react-redux`. From the store, it receives the props: `name`, `location`, `likes` and `description`.



The description `props` define the current profession of the user.

Essentially, what's happening is that whenever the user profession is changed by clicking the button, the `description` prop is changed. This change in props then causes the `App` component to be re-rendered entirely.

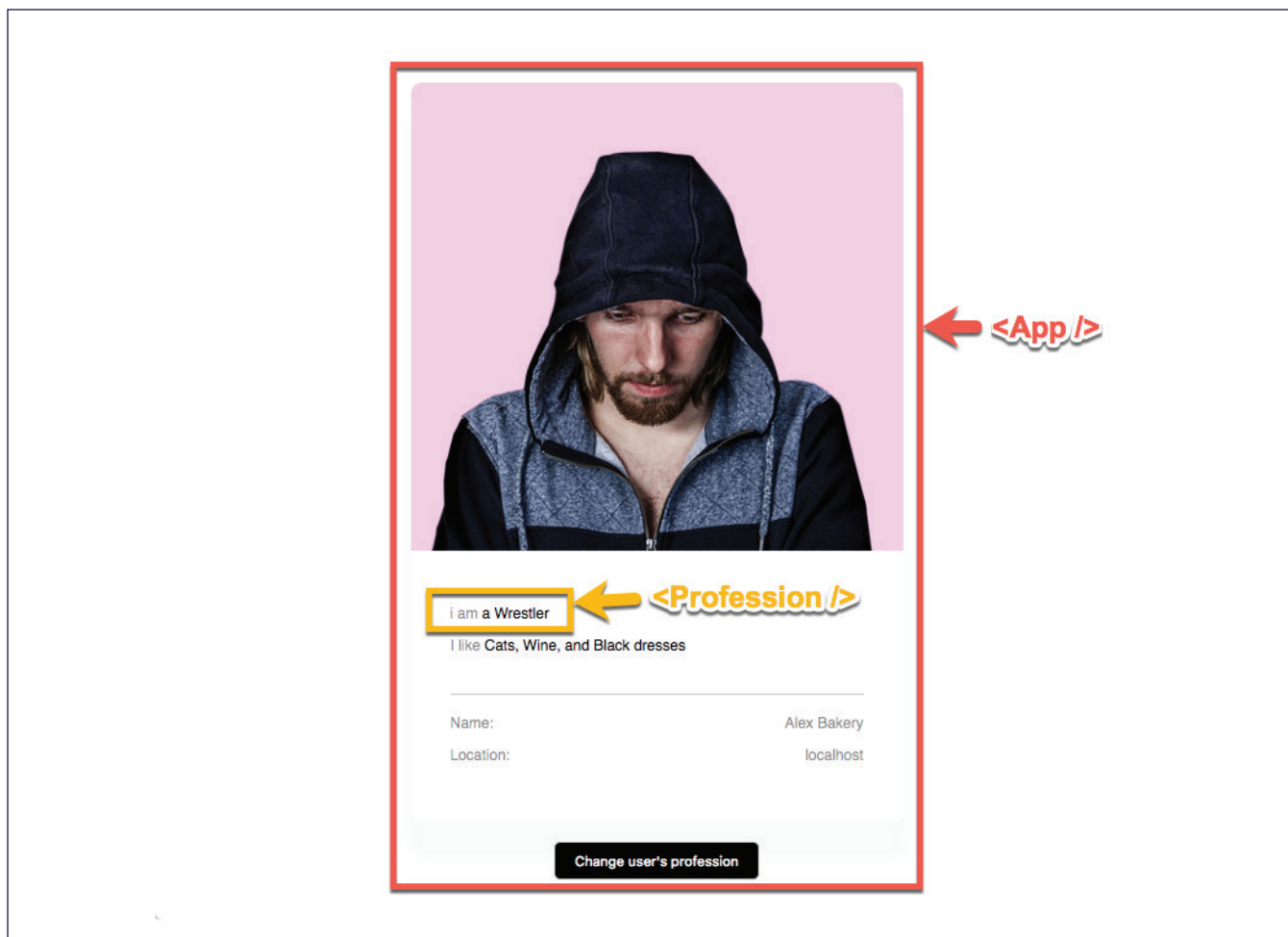
If you remember from React 101, whenever the `props` or `state` of a component changes, a re-render is triggered.



A React component renders a tree of elements. These elements are defined via props and state. If the props or state values changes, the tree of elements is re-rendered. This results in a new tree.

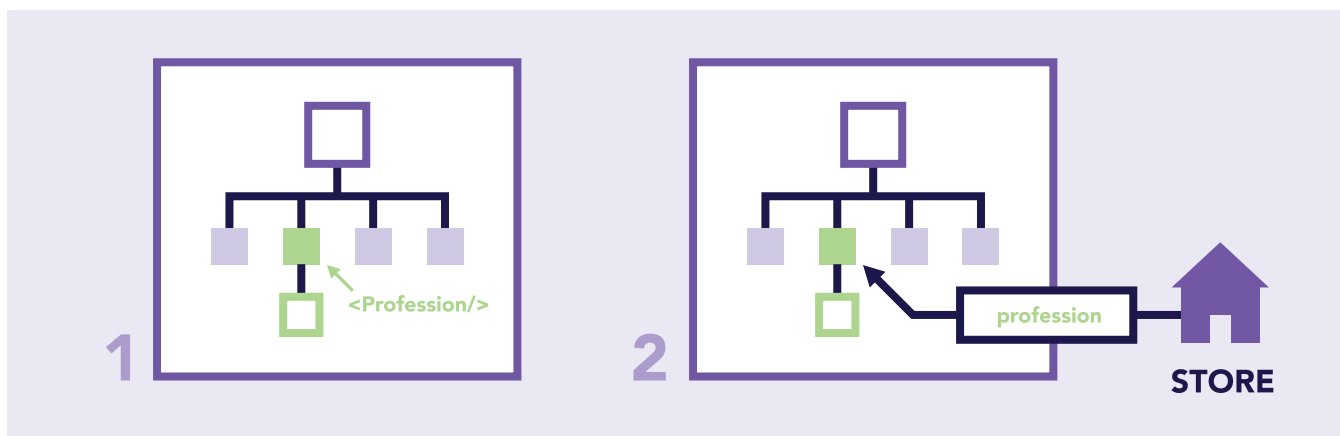
Instead of allowing the `App` component to re-render pointlessly, what if we localized the elements being updated to a specific React component?

For example, we could create a new component called `Profession` which renders its own DOM elements.



In this case, the **Profession** component will render the description of the user's profession e.g "I am a Coder".

The component tree would now look like this:

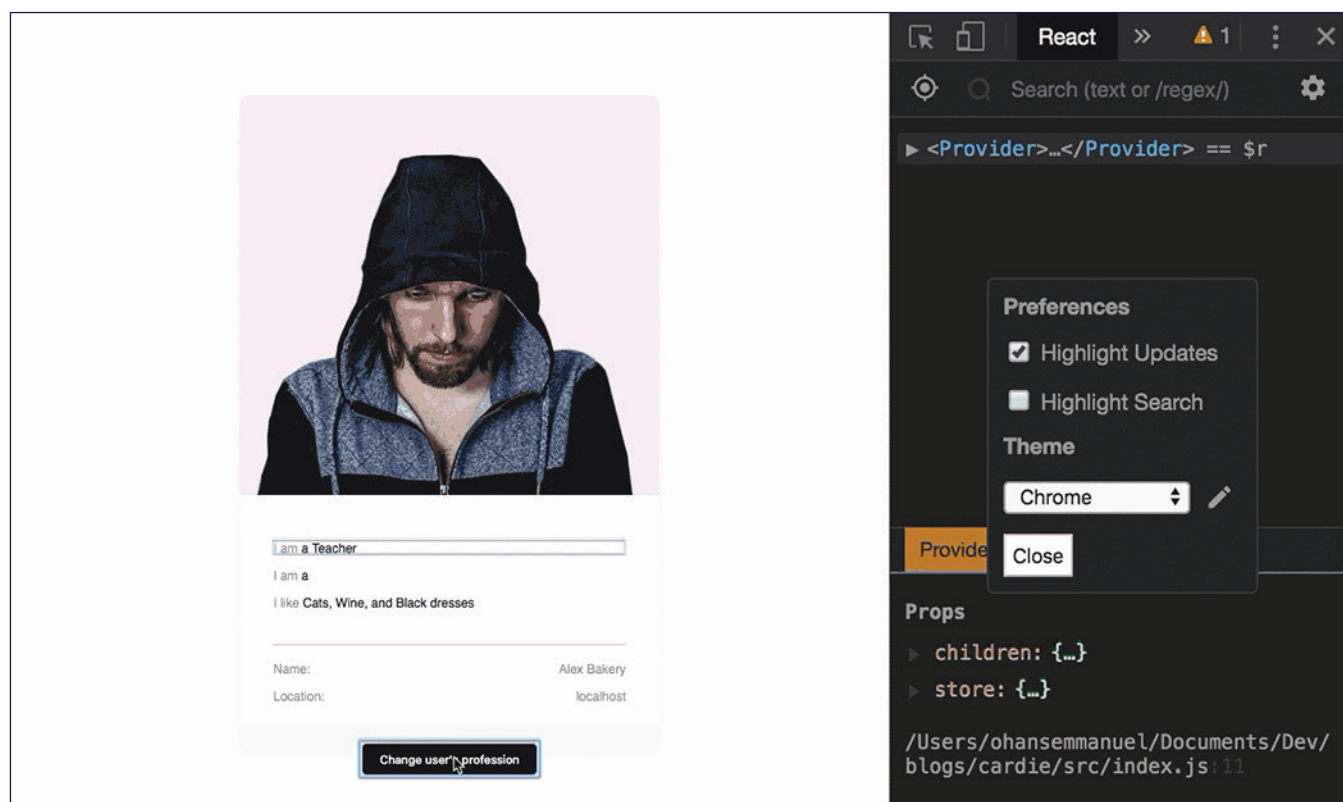


1. The `<App />` component renders the elements it renders plus the `<Profession />` component.
2. The profession will now be retrieved directly from the redux store by the `<Profession />` component.

What's also important to note here is that instead of letting `<App/>` worry about the `profession` prop, that'll now be a concern for the `<Profession/>` component.

Whether you use `redux` or not, the point here is that `App` no longer has to be re-rendered owing to changes from the `profession` prop. Instead, the `<Profession/>` component will be.

Upon completing this refactor, we have an ideal update highlighted as seen below:



Note how the highlighted updates is contained within `<Profession />`

To view the code change, please see the [isolated-component branch](#) from the repo.

3. Use pure components when appropriate

Any React resource talking about performance will very likely mention pure components. However, how do you know when to properly use [pure components](#)?

Well, it is true that you could make every component a pure component, but remember there's a reason why that isn't the case out of the box. Hence the `shouldComponentUpdate` function.

The premise of a pure component is that the component **ONLY** re-renders if the `props` are different from the previous props and state. An easy way to implement a pure component is to use `React.PureComponent` as opposed to the default `React.Component`.

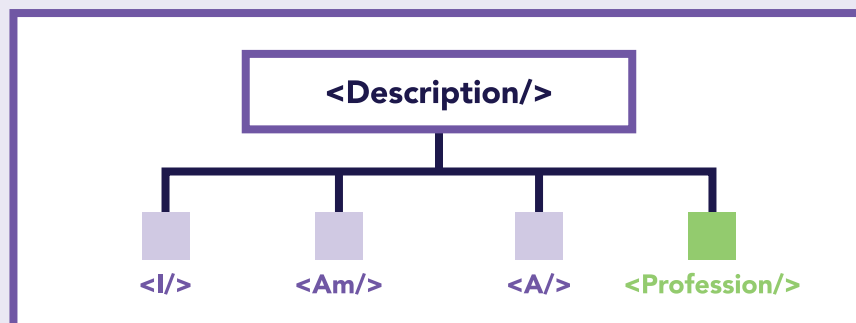
To illustrate this specific use case in Cardie, let's break down the render elements of the `Profession` component into smaller components.

```
const Description = ({ description }) => {  
  return (  
    <p>  
      <span className="faint">I am</span> a {description}  
    </p>  
  );  
}
```

Here's what we'll change it to:

```
const Description = ({ description }) => {  
  return (  
    <p>  
      <I />  
      <Am />  
      <A />  
      <Profession profession={description} />  
    </p>  
  );  
};
```

Now, the **Description** component renders 4 children components.

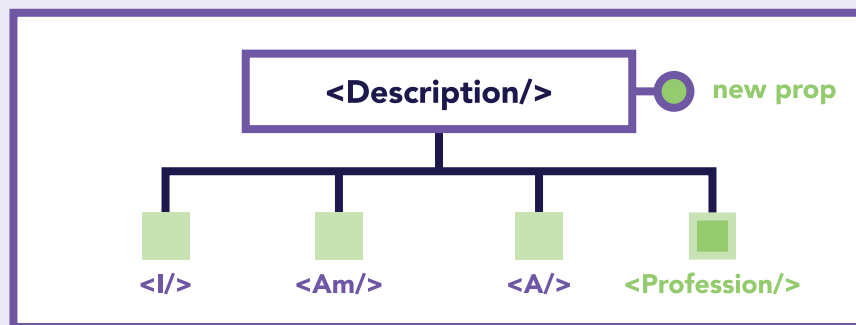


Note that the **Description** component takes in a **profession** prop. However, it passes on this prop to the **Profession** component. Technically, none of the other 3 components care about this **profession** prop.

The contents of these new components are super simple. For example, the **<I/>** component just returns a **span** element with an "I" text: **I**

If the application runs, the result is just the same. The app works.

What's interesting is that upon a change to the `description` prop, every child component of `Profession` is also re-rendered.



I added a few logs in the `render` methods of each child component—and as you can see they were indeed all re-rendered.

```
Console
top
Filter
Navigated to http://localhost:3000/
render called in <I/> I.js:5
render called in <Am/> Am.js:5
render called in <A/> A.js:5
render called Profession.js:5 in <Profession/>
```

You may also view the highlighted updates using the react dev tools.

This behavior is expected. Whenever a component has either **props** or **state** changed, the tree of elements it renders is recomputed.

This is synonymous to a re-render.

In this particular example, you'll agree with me that it really makes no sense for **<I/>**, **<Am/>** and **<A/>** child components to be re-rendered. Yes, the props in the parent element, **<Description/>** changed, but if this were a sufficiently large application, this behaviour may pose some performance threats.

What if we made these child components pure components?

Consider the **<I/>** component:

```
import React, {Component, PureComponent} from "react"

//before
class I extends Component {
  render() {
    return <span className="faint">I </span>;
  }
}

//after
class I extends PureComponent {
  render() {
    return <span className="faint">I </span>;
  }
}
```


By implication, React is informed under the hood so that if the prop values for these child components aren't changed, there's no need to re-render them.

Yes, do not re-render them even when the parent element has a change in its props!

Upon inspecting the highlighted updates after the refactor, you can see that the child components are no longer re-rendered. Just the **Profession** component whose **prop** actually changes is re-rendered.

In a larger application, you may find immense performance optimizations by just making certain components pure components.

To view the code change, please see the [pure-component branch](#) from the repo.

4. Avoid passing new objects as props

Remember again that whenever the **props** for a component changes, a re-render happens. What if the props for your component didn't change but React thinks it did change?


Well, there'll also be a re-render!

But isn't that weird?

This seemingly weird behavior happens because of how Javascript works & how React handles its comparison between old and new prop values.

Let me show you an example.

Here's the content for the **Description** component:



```
const Description = ({ description }) => {  
  return (  
    <p>  
      <I />  
      <Am />  
      <A />  
      <Profession profession={description} />  
    </p>  
  );  
};
```

Now, we will refactor the **I** component to take in a certain **i** prop. This will be an object of the form:

```
const i = {  
  value: "i"  
};
```

Whatever value property is present in **i** will be set as the value in the **I** component.

```
class I extends PureComponent {  
  render() {  
    return <span className="faint">{this.props.i.value} </span>;  
  }  
}
```

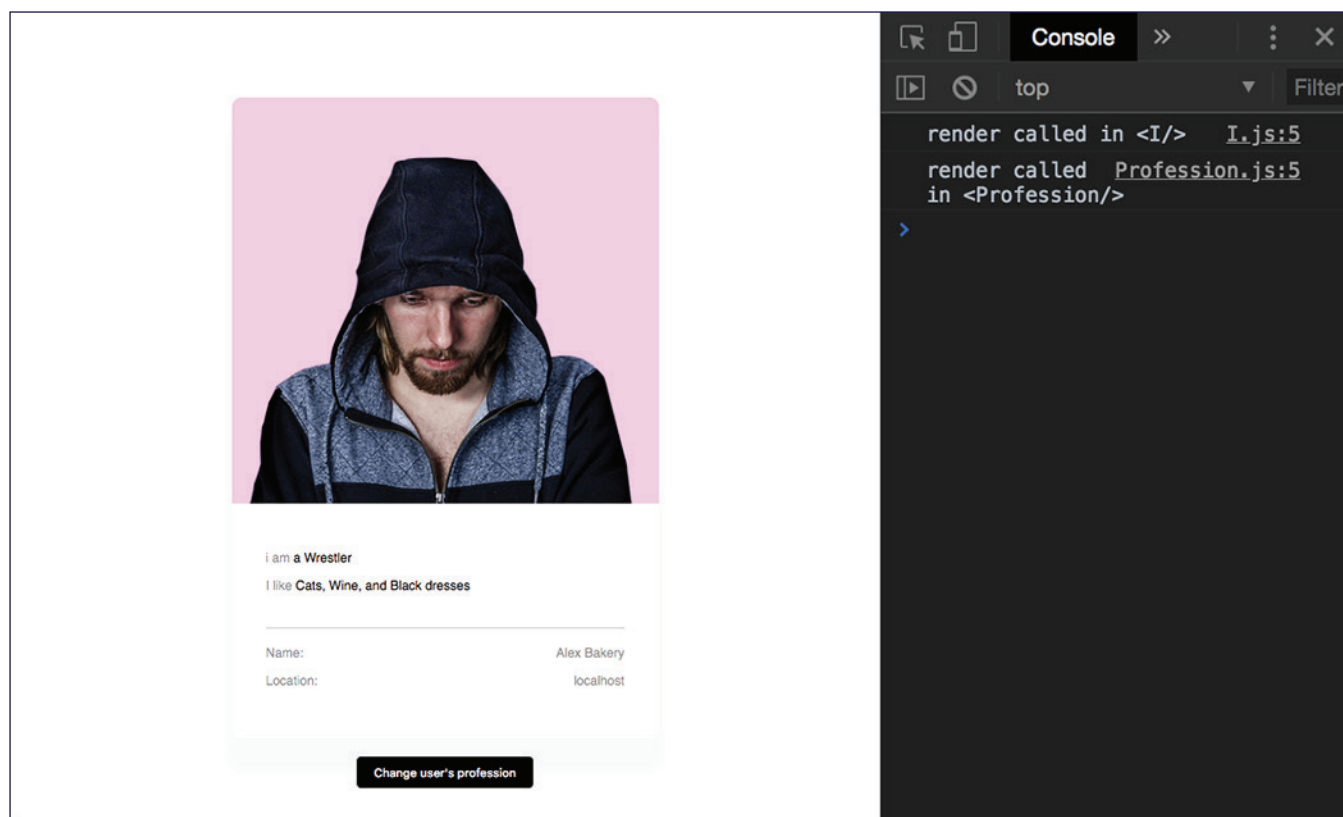
In the **Description** component, the **i** prop is created and passed in as shown below:

```
class Description extends Component {  
  render() {  
    const i = {  
      value: "i"  
    };  
    return (  
      <p>  
        <I i={i} />  
        <Am />  
        <A />  
        <Profession profession={this.props.description} />  
      </p>  
    );  
  }  
}
```

Bear with me while I explain this.

This is perfectly correct code, and it works fine—but there is one problem.

Even though **I** is a pure component, now it re-renders whenever the profession of the user is changed!

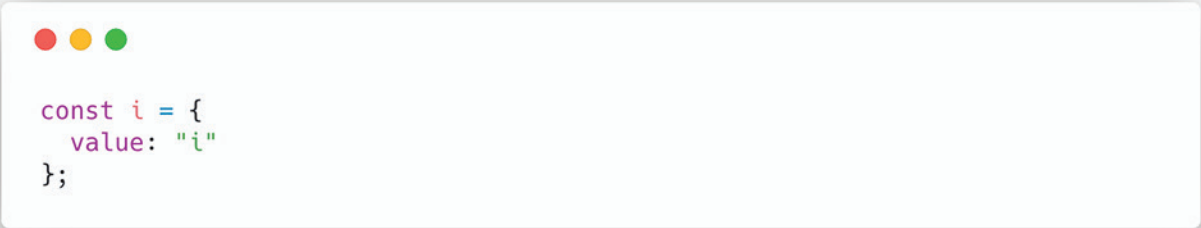


On Clicking the button, the logs show that both `<I/>` and `<Profession/>` are re-rendered. Remember there's been no actual props change in `<I/>`. Why the re-render?

But why?

As soon as the **Description** component receives the new props, the **render** function is called to create its element tree.

Upon invoking the render function it creates a new `i` constant:



```
const i = {  
  value: "i"  
};
```

When React gets to evaluate the line, `<I i={i} />`, it sees the props `i` as a different prop, a new object—therefore the re-render.

If you remember from React 101, React does a shallow comparison between the previous and next props. Scalar values such as strings and numbers are compared by value. Objects are compared by reference.

By implication, even though the constant `i` has the same value across re-renders, the reference is not the same. The position in memory isn't.

It's a newly created object with every single render call.

For this reason, the `prop` value passed to `<I/>` is regarded as "new", consequently a re-render. In bigger applications, this can lead to a wasted render, and potential performance pitfalls.

Avoid this.

This applies to every **prop** including event handlers.
If you can avoid it, you shouldn't do this:

```
...  
render() {  
  <div onClick={() => {//do something here}}  
}  
...
```

You're creating a new function object every time within render.
Better do this:

```
...  
handleClick:() = {  
}  
render() {  
  <div onClick={this.handleClick}  
}  
...
```

Got that?

In the same vein, we may refactor the prop sent to `<I />` as shown below:

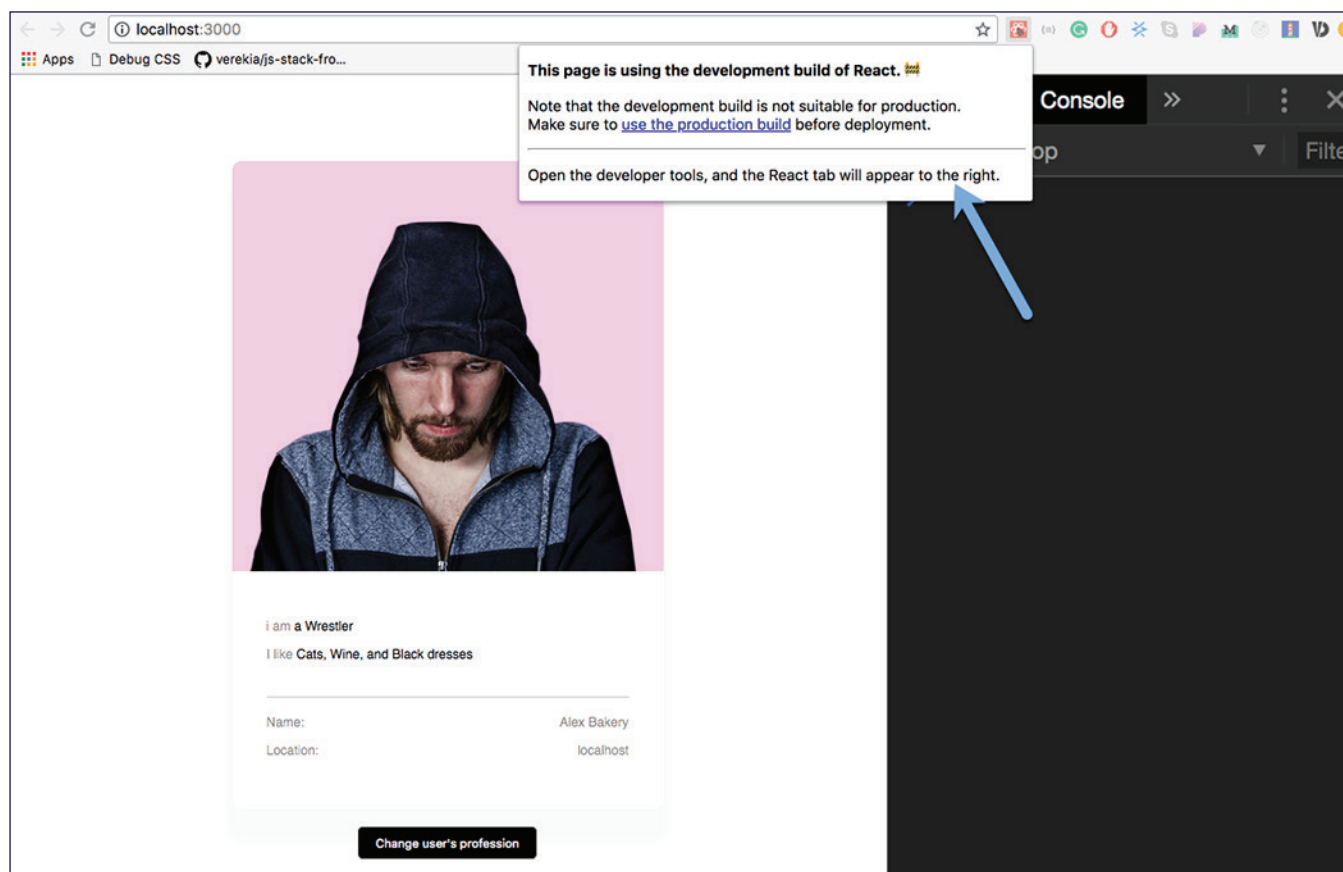
```
class Description extends Component {  
  i = {  
    value: "i"  
  };  
  render() {  
    return (  
      <p>  
        <I i={this.i} />  
        <Am />  
        <A />  
        <Profession profession={this.props.description} />  
      </p>  
    );  
  }  
}
```

Now, the reference is always the same, `this.i`. A new object isn't created at render time.

To view the code change, please see the [new-objects branch](#) from the repo.

5. Use the production build

When deploying to production, always use the production build. This is a very simple, but great practice.



The "development build" warning you get with the react devtools in development.

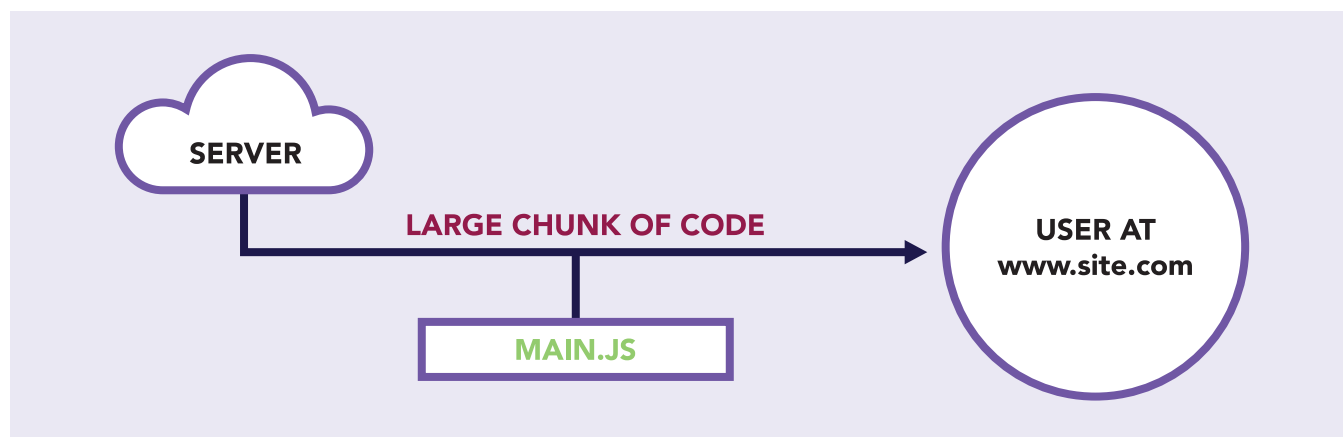
If you have bootstrapped your application with `create-react-app`, to run the production build, use the command: `npm run build`.

This will yield bundle optimized files for production.

6. Employ code splitting

When you bundle your application, you likely have the entire application bundled in one large chunk.

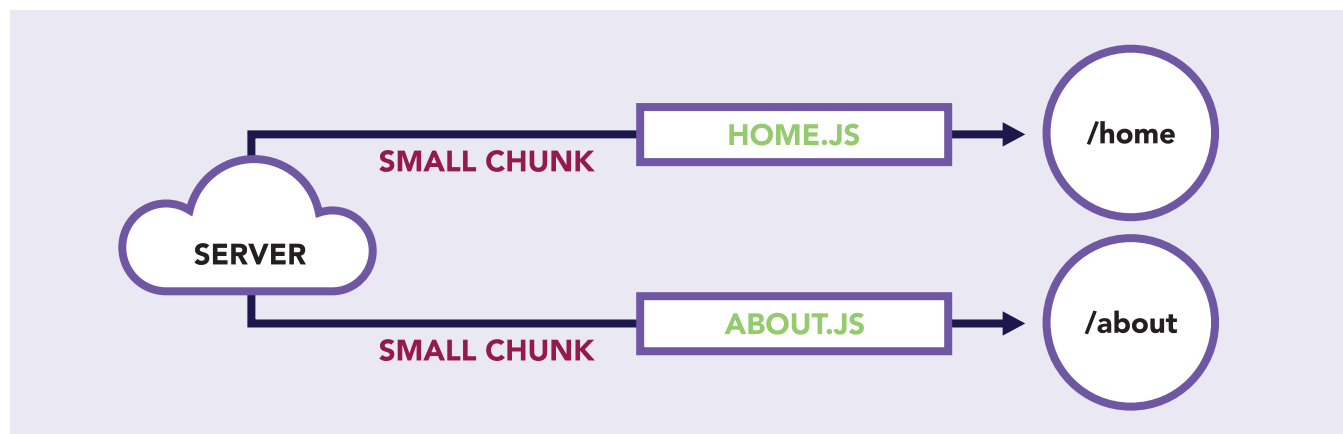
The problem with this is that as your app grows, so does the bundle.



Once the user visits the site, they are sent a large chunk of code for the entire app.

Code splitting advocates that instead of sending this large chunk of code to the user at once, you may dynamically send chunks to the user when they need it.

A common example is with route based code splitting. In this method, the code is split into chunks based on the routes in the application.



The /home route gets a small chunk of code, so does the /about route.

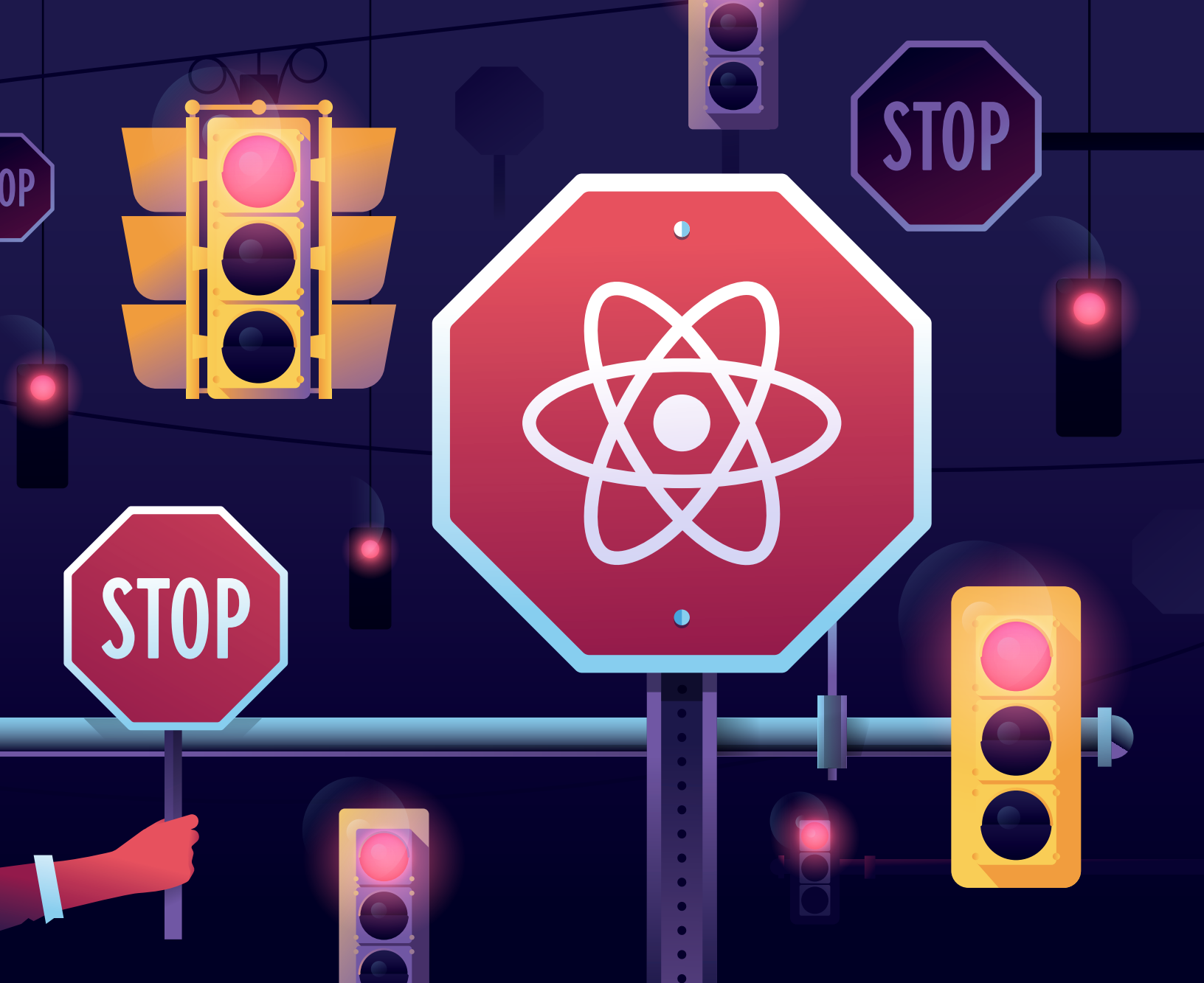
Another approach is component based code splitting. In this method, if a component is currently not displayed to the user, it's code may be delayed from being sent to the user.

Whichever method you stick to, it is important to understand the trade-offs and not degrade the user experience of your application. Code splitting is great, and it can improve your application's performance.

I have taken a conceptual approach to explain code splitting. If you want more technical grounds, please have a look at the official [React docs](#). They do a decent job at explaining the concept technically.

Now you've got a decent checklist for tracking and fixing common performance issues in react apps. Go build some fast apps!





Five Common Practices That You Can Stop Doing in React

By Manjunath M • Founder of Storylens.com

At this point, it's tough to argue that React is one of the most loved libraries on the planet.

There is a tremendous amount of interest in React and new developers are swayed into the platform because of its UI-first approach. And while both the library and the entire React ecosystem have matured over the years, there are certain instances where you find yourself asking “what’s the right way to do this, exactly?”

And that’s a fair question to ask—there isn’t always a firm “right” way of doing things. In fact, as you likely already know, sometimes best practices aren’t so great. Some of them can compromise performance, readability and make things unproductive in the long run.

In this article, I’ll describe 5 generally accepted development practices that you can actually avoid when using React. Naturally, I’ll explain why I consider the practice avoidable and suggest alternative approaches that let you accomplish the same thing.

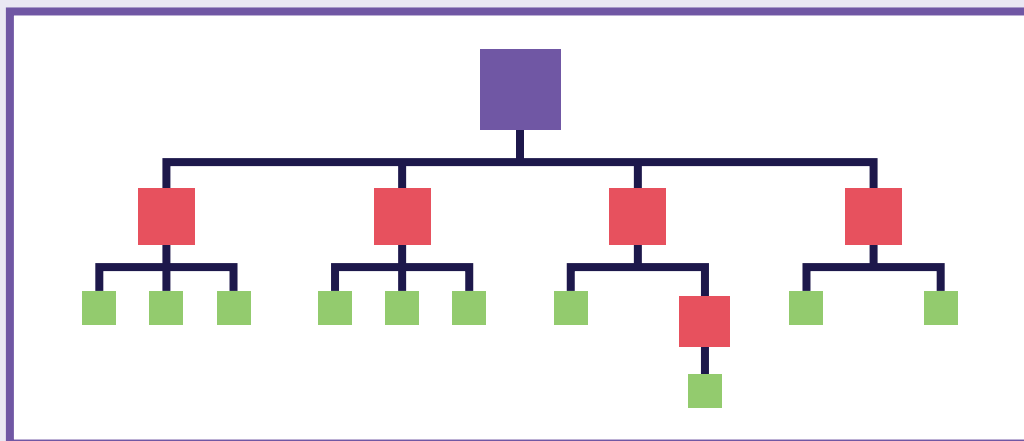
Optimizing React Right from the Start

The developers at React have put a lot of effort into making React fast and new optimizations are added to the mix after each new update. In my opinion, you shouldn't spend time optimizing stuff until you see actual performance hits.

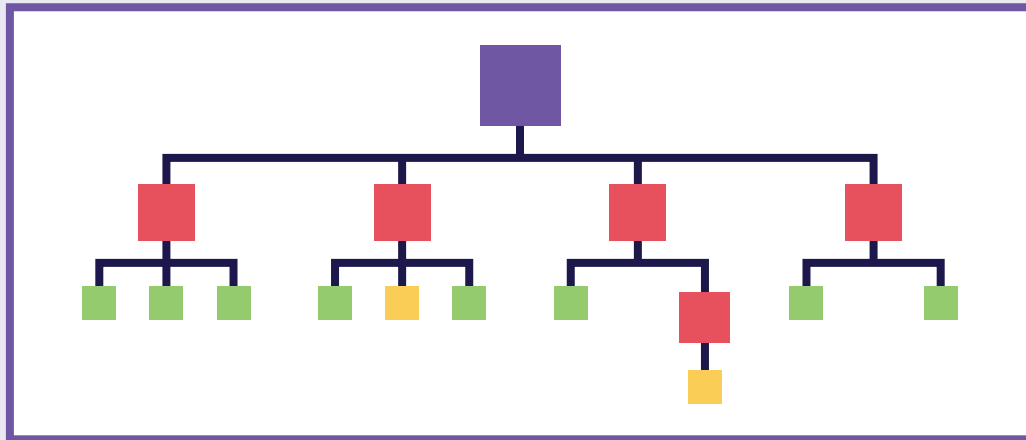
Why?

It's easier to scale React compared to other front-end platforms because you don't have to rewrite entire modules to make things faster. The usual culprit that causes performance issues is the reconciliation process that React uses to update the virtual DOM.

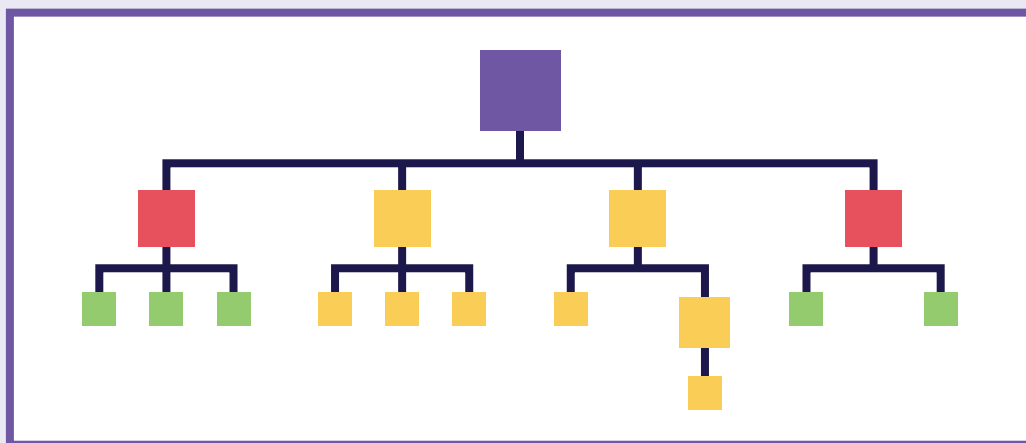
Let's have a look at how React handles things under the hood. On each `render()`, React generates a tree that's composed of UI elements—the leaf nodes being the actual DOM elements. When the `state` or `props` get updated, React needs to generate a new tree with minimal number changes and keep things predictable. Imagine you have a tree that looks like this:



Imagine that the application receives new data and the following nodes need to be updated:




React usually ends up re-rendering the entire subtree instead of rendering only the relevant nodes like this:



When the state changes at top-order components, all the components below it get re-rendered. That's the default behavior and it's okay for a small-sized application. As the application grows, you should consider measuring the actual performance using Chrome Profiling Tools. The tool will give you precise details about the time wasted on unwanted renders. If the numbers are significant, you can then optimize the rendering time by adding a `shouldComponentUpdate` hook into your component.

The hook gets triggered before the re-rendering process starts and by default, it returns `true`:



```
shouldComponentUpdate(nextProps, nextState) {  
  return true;  
}
```

When it returns `true`, React's diff algorithm takes over and re-renders the entire subtree. You can avoid that by adding comparison logic into `shouldComponentUpdate` and updating the logic only when the relevant `props` have changed.



```
shouldComponentUpdate(nextProps, nextState) {  
  if (this.props.color !== nextProps.color) {  
    return true;  
  }  
  if (this.state.count !== nextState.count) {  
    return true;  
  }  
  return false;  
}
```

The component won't update if any other **props** / **state** has changed except **color** / **count**.

Apart from this, there are certain non-React optimization tricks that developers usually miss, but they have an impact on the application's performance.

I've listed some of the avoidable habits and the solutions below:

Unoptimized images

If you're building on dynamic images, you need to consider your options while dealing with images. Images with huge file sizes can give the user an impression that the application is slow. Compress the images before you push them into the server or use a [dynamic image manipulation](#) solution instead. I personally like Cloudinary to optimize react images because it has its own react library, but you could also use Amazon S3 or Firebase instead.

Uncompressed build files

Gzipping build files (bundle.js) can reduce the file size by a good amount. You will need to make modifications to the webserver configuration. Webpack has a gzip compression plugin known as [compression-webpack-plugin](#). You can use this technique to generate bundle.js.gz during build time.

Server-side rendering for SEO

Although Single Page applications are awesome, there are two issues that are still attributed back to them.

1. When the application loads initially, there is no cache of JavaScript in the browser. If the application is big, the time taken to initially load the application will also be huge.
2. Since the application is rendered in the client side, the web crawlers that search engines use won't be able to index the JavaScript generated content. The search engines will see your application to be blank and then rank you poorly.

That's where the server-side rendering technique comes in handy. In SSR, the JavaScript content is rendered from the server initially. After the initial render, the client-side script takes over and it works like a normal SPA. The complexity and the cost involved in setting up the traditional SSR is higher because you need to use a Node/Express server.

There's good news if you're in it for the SEO benefit, Google indexes and crawls the JavaScript content without any trouble. Google actually started to crawl JavaScript material back in 2016 and the algorithm works flawlessly right now.

Here's an excerpt from the [Webmaster blog back](#) in October 2015:

Today, as long as you're not blocking Googlebot from crawling your JavaScript or CSS files, we are generally able to [render and understand your web pages like modern browsers](#). To reflect this improvement, we recently [updated our technical Webmaster Guidelines](#) to recommend against disallowing Googlebot from crawling your site's CSS or JS files.

If you're only using server-side rendering because you're worried about your Google Page Rank, then you don't need to use SSR. It used to be a thing in the past, but not anymore.

However, if you're doing it to improve the initial render speed, then you should try an easier implementation of SSR using a library like Next.js. Next saves you time that you would otherwise spend on setting up the Node/Express server.

Inline styles & CSS imports

While working with React, I've personally tried different styling ideas to find new ways to introduce styles into React components. The traditional CSS-in-CSS approach that has been around for decades works with React components. All your stylesheets would go into a stylesheets directory and you can then import the required CSS into your component.

However, when you're working with components, stylesheets don't make sense anymore. While React encourages you to think of your application in terms of components, stylesheets force you to think of it at the document level.

Various other approaches are being practiced to merge the CSS and the JS code into a single file. The Inline Style is probably the most popular among them.

```
import React from 'react';

const divStyle = {
  margin: '40px',
  border: '5px solid pink'
};
const pStyle = {
  fontSize: '15px',
  textAlign: 'center'
};

const TextBox = () => (
  <div style={divStyle}>
    <p style={pStyle}>Yeah!</p>
  </div>
);

export default TextBox;
```

You don't have to import CSS anymore, but you're sacrificing readability and maintainability. Apart from that, Inline Styles don't support media queries, pseudo classes and pseudo elements and style fallbacks. Sure, there are hacks that let you do some of them, but it's just not that convenient.

That's where CSS-in-JSS comes in handy and Inline Styles are not exactly CSS-in-JSS. The code below demonstrates the concept using styled-components.



```
import styled from 'styled-components';

const Text = styled.div`
  color: white,
  background: black
`

<Text>This is CSS-in-JS</Text>
```

What the browser sees is something like this:



```
<style>
.hash234dd2 {
  background-color: black;
  color: white;
}
</style>

<p class="hash234dd3">This is CSS-in-JS</p>
```

A new `<style>` tag is added to the top of the DOM and unlike inline styles, actual CSS styles are generated here. So, anything that works in CSS works in styled components too. Furthermore, this technique enhances CSS, improves readability and fits into the component architecture. With the `styled-components` lib, you also get SASS support that's been bundled into the lib.

Nested ternary operator

Ternary operators are popular in React. It's my go-to operator for creating conditional statements and it works great inside the `render()` method. For instance, they help you to render elements inline and in the example below, I've used it to display the login status.

```
render() {  
  const isLoggedIn = this.state.isLoggedIn;  
  return (  
    <div>  
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.  
    </div>  
  );  
}
```

However, when you nest the ternary operators over and over, they can become ugly and hard to read.

```
int median(int a, int b, int c) {  
  return (a < b) ? (b < c) ? b : (a < c) ? c : a : (a < c) ? a : (b < c) ? c : b;  
}
```

As you can see, the shorthand notations are more compact, but they make the code appear messy. Now imagine if you had a dozen or more nested ternaries in your structure. And it happens a lot often than you think. Once you start with the conditional operators, it's easy to keep on nesting it and finally, you reach a point where you decide that you need a better technique to handle conditional rendering.

But the good thing is that you have many alternatives that you can choose from. You can use a babel plugin like JSX Control Statements that extends JSX to include components for conditional statements and loops.

```

<If condition={ test }>
  <span>Truth</span>
</If>

// after transformation
{ test ? <span>Truth</span> : null }
```

There's another popular technique called *iify* (IIFE—Immediately-invoked function expressions). It's an anonymous function that is invoked immediately after they are defined.

```

(function() {
  // Do something
})()
```

We've wrapped the function inside a pair of parentheses to make the anonymous function a function expression. This pattern is popular in JavaScript for many reasons. But in React, we can place all the `if` / `else` statements inside the function and return whatever that we want to render.

Here is an example that demonstrates how we're going to use IFFE in React.

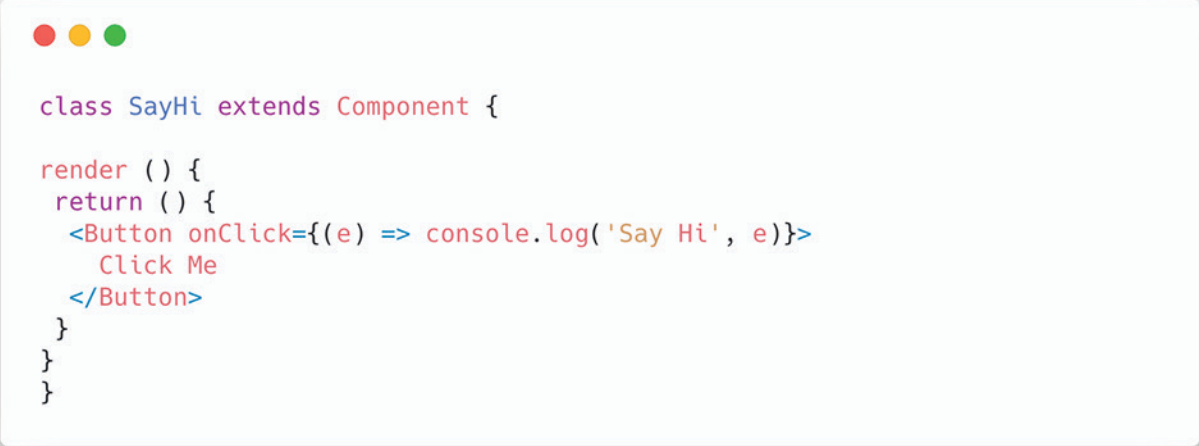


```
{
  (() => {
    if (this.props.status === 'PENDING') {
      return (<div className="loading" />);
    }
    else {
      return (<div className="container" />);
    }
  })()
}
```

IFFE's can have an impact on performance, but it won't be anything significant in most cases. There are more methods to run conditional statements in React and we've covered that in [8 methods for conditional rendering in React](#).

Closures in React

Closures are inner functions that have access to the outer function's variables and parameters. Closures are everywhere in JavaScript and you've been probably using it even if you've not realized yet.



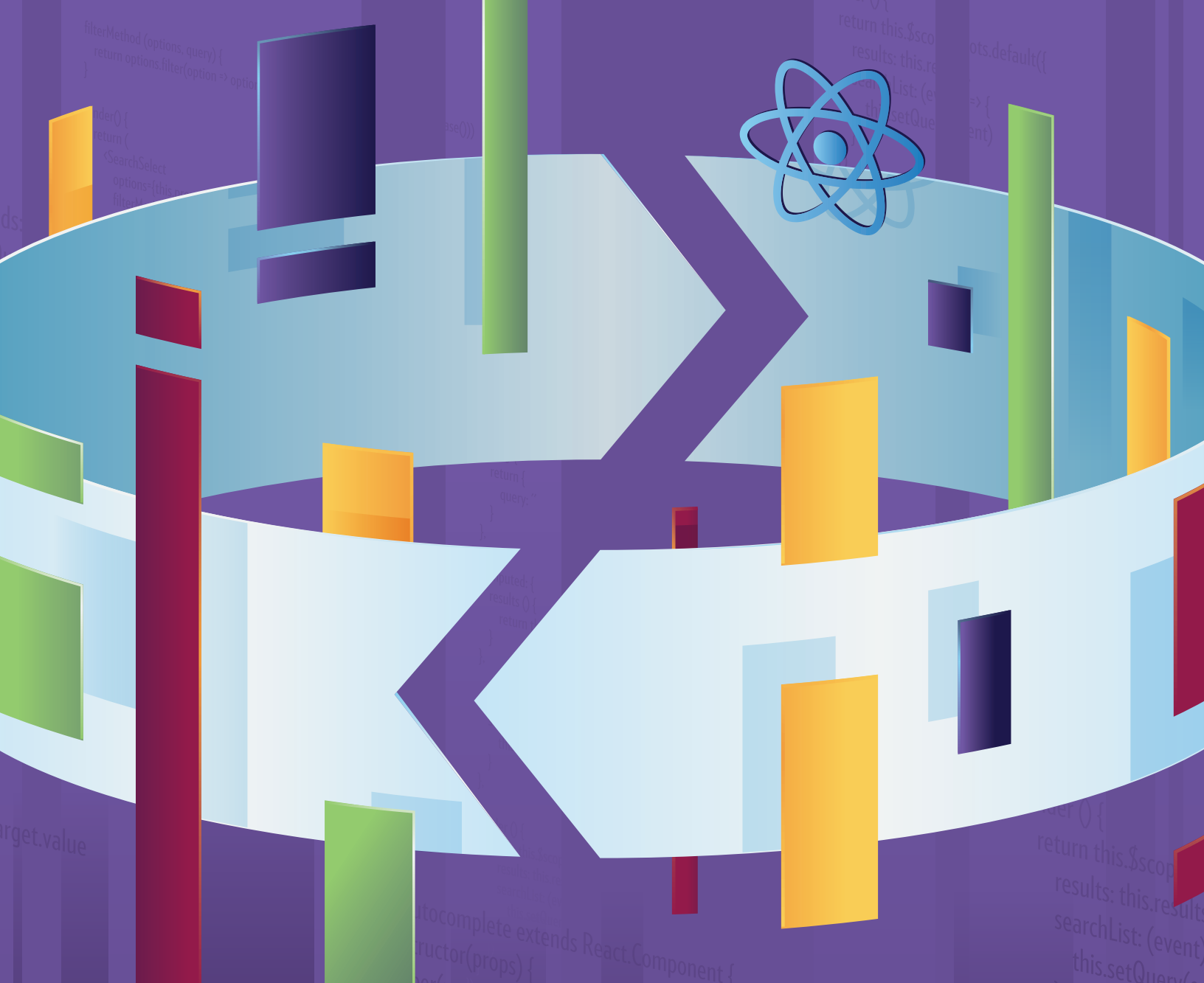
```
class SayHi extends Component {  
  render () {  
    return () {  
      <Button onClick={(e) => console.log('Say Hi', e)}>  
        Click Me  
      </Button>  
    }  
  }  
}
```

But when you are using closures inside the `render()` method, it's actually bad. Every time the `SayHi` component is rendered, a new anonymous function is created and passed to `Button` component. Although the props haven't changed, `<Button />` will be forced to re-render. As previously mentioned, wasted renders can have a direct impact on performance.

Instead, replace the closures with a class method. Class methods are more readable and easy to debug.

When a platform grows, new patterns emerge each day. Some patterns help you improve your overall workflow whereas a few others have significant side effects. When the side effects impact your application's performance or compromise readability, it's probably a better idea to look for alternatives. In this post, I've covered some of the practices in React that you can avoid because of their shortcomings.





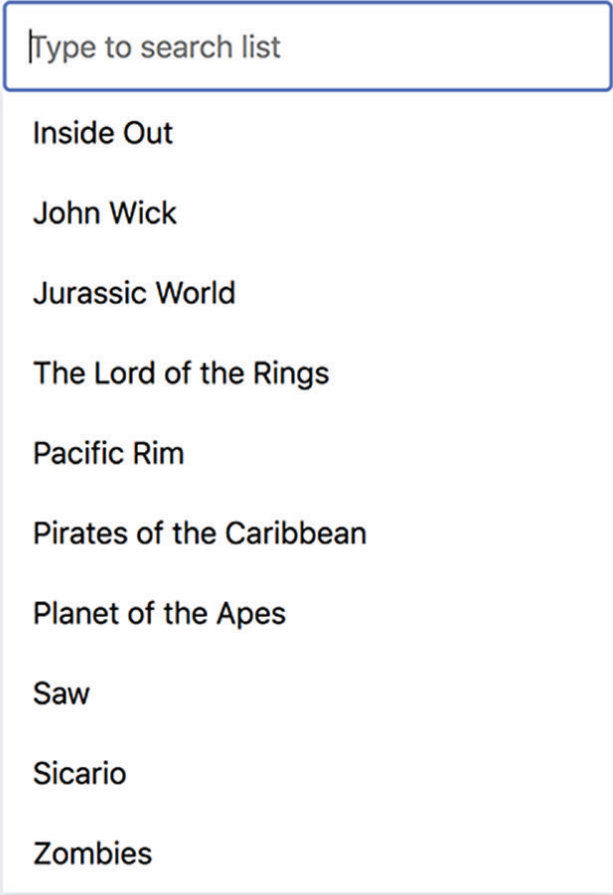
Modern Component Reusability: Render Props in React

By Jonathan Harrell • UI/UX Designer & Front-End Developer

One of the issues all front-end developers face is how to make UI components reusable.

How do we craft components in such a way that satisfies the narrow use case that is clear to us now, while also making them reusable enough to work in a variety of circumstances?

Let's say we are building an autocomplete component:



Type to search list

- Inside Out
- John Wick
- Jurassic World
- The Lord of the Rings
- Pacific Rim
- Pirates of the Caribbean
- Planet of the Apes
- Saw
- Sicario
- Zombies

Take a look at the initial React component code:

```
class Autocomplete extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      results: props.options
    }
  }

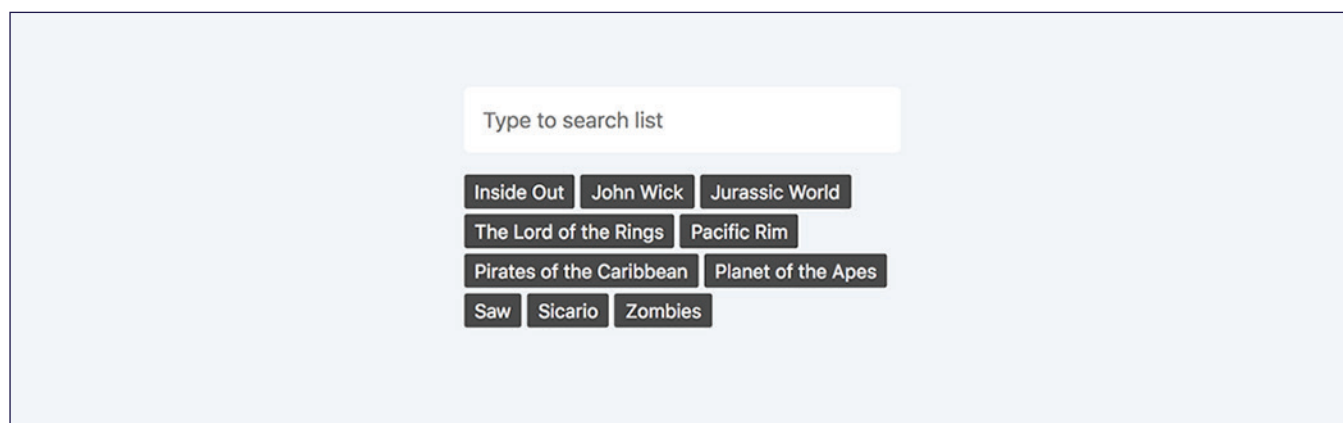
  searchList(event) {
    const results = this.props.options
      .filter(option => option.toLowerCase().includes(event.target.value.toLowerCase()))
    this.setState({ results })
  }

  render () {
    return (
      <div className="autocomplete">
        <input
          type="text"
          placeholder="Type to search list"
          onChange={searchList}
        />
        <div className="autocomplete-dropdown">
          <ul className="autocomplete-search-results">
            {this.state.results.map(option => (
              <li class="autocomplete-search-result">{option}</li>
            ))}
          </ul>
        </div>
      </div>
    )
  }
}
```

In this component, we have some logic that controls the core search behavior, but we also specify how the input and search results will be rendered. In this instance, we render a `div` that serves as a dropdown container and an unordered list containing a list item for each result within it.

Think about how you would reuse this component. Sure, you could use this very same component if you want to reproduce exactly the same behavioral and visual result. But what if you want to reuse the same behavior, but visualize the component slightly differently? What if you want to reuse the core search behavior but add a few modifications for a slightly different use case?

Imagine that instead of a dropdown containing the search results, you want a tag-like list of search results that always display:



At their core, the functionality of these two components is very similar: type into an input to filter a list.

This is a perfect use case for some relatively new tools that modern JavaScript frameworks now provide. These are *render props*. They provide a way to separate the **behavior** of a component from its **presentation**.

Render Props in React

First, let's look at how we would restructure our autocomplete component using render props in React. We will now have two components—one for our Autocomplete component and one for a core SearchSelect component.

Let's look at the SearchSelect component first:

```
class SearchSelect extends React.Component {  
  constructor(props) {  
    super(props)  
    this.state = {  
      results: props.options  
    }  
  }  
  
  searchList(event) {  
    const results = this.props.filterMethod(this.props.options, event.target.value)  
    this.setState({ results })  
  }  
  
  render() {  
    return this.props.render({  
      results: this.state.results,  
      searchList: (event) => this.searchList(event)  
    })  
  }  
}
```

This is a *renderless* component (one that doesn't render any markup of its own). Rather, it returns the result of a special prop called a *render prop*. This render prop accepts an object, into which you can pass any data that you would like the parent component to have access to.

Think of this as the opposite of normal props. Usually, props are passed down from parent to child. In the case of render props, these are returned from the child so the parent can access them.

Our `SearchSelect` component is handling the lowest level functionality—filtering a list of options based on a query string. It is then using the special render prop to render an element. In the parent, we pass a function to the render prop of the `SearchSelect` component. This function returns a React element, which we can hydrate with state and behavior from the `SearchSelect` component itself. Basically, this means we are able to access data from the child component in the parent.

```
import SearchSelect from './search-select'

class Autocomplete extends React.Component {
  constructor(props) {
    super(props)
  }

  filterMethod (options, query) {
    return options.filter(option => option.toLowerCase().includes(query.toLowerCase()))
  }

  render() {
    return (
      <SearchSelect
        options={this.props.options}
        filterMethod={this.filterMethod}
        render={({results, searchList}) => (
          <div>
            <input
              type="text"
              placeholder="Type to search list"
              onChange={searchList}
            />
            <ul>
              {results.map(option => (
                <li>{option}</li>
              ))}
            </ul>
          </div>
        )}
      />
    )
  }
}
```

All this means that we can write whatever markup we want, as long as we properly hydrate it with the data and behavior exposed by the `SearchSelect` component.

Also, note how we are passing the method for filtering our list in as a prop. This will allow us to change the way our list of options is filtered, while still using the `SearchSelect` component.

Let's look at how we would implement our tag-like list component. We use the same `SearchSelect` core component and just change the markup rendered by the `render` prop:

```
import SearchSelect from './search-select'

class TagListSearch extends React.Component {
  constructor(props) {
    super(props)
  }

  filterMethod (options, query) {
    return options.filter(option => option.toLowerCase().includes(query.toLowerCase()))
  }

  render() {
    return (
      <SearchSelect
        options={this.props.options}
        filterMethod={this.filterMethod}
        render={({results, searchList}) => (
          <div className="tag-list-search">
            <input
              type="text"
              placeholder="Type to search list"
              onChange={searchList}
            />
            <ul className="tag-list">
              {results.map(result => (
                <li className="tag" key={result}>{result}</li>
              ))}
            </ul>
          </div>
        )}
      />
    )
  }
}
```


Other uses for render props

Creating reusable interface components isn't the only use for render props and scoped slots. Here are some other ideas for how you can use them to encapsulate reusable behavior in a component that can then be exposed to its parent.

Data provider components: You can use render props/scoped slots to create a component that handles asynchronously fetching data and exposing that data to its parent. This allows you to hide the logic for hitting an endpoint, getting the result and handling possible errors, as well as displaying a loading state to users while the data fetch is in progress. Here's what the base component could look like:

```
class FetchData extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      loading: false,
      results: [],
      error: false
    }
  }

  componentDidMount() {
    this.fetchData(this.props.url)
  }

  fetchData(url) {
    this.setState({ loading: true })

    fetch(url)
      .then(data => data.json())
      .then(json => {
        this.setState({ loading: false, results: json })
      })
      .catch(error => {
        this.setState({ loading: false, error: true })
      })
  }

  render() {
    return this.props.render({
      loading: this.state.loading,
      results: this.state.results,
      error: this.state.error
    })
  }
}
```

It accepts a URL as a prop and handles the actual fetching logic. Then, we use it in a parent component:

```
class App extends React.Component {
  constructor(props) {
    super(props)
  }

  render() {
    return (
      <div className="wrapper">
        <FetchData
          url="https://jsonplaceholder.typicode.com/todos"
          render={({loading, results, error}) => (
            <div>
              {loading && (
                <p>Loading...</p>
              )}
              {results.length > 0 && (
                <div className="results">
                  {results.map(result => (
                    <p key={result.id}>{result.title}</p>
                  ))}
                </div>
              )}
              {error && (
                <p>There was a problem loading.</p>
              )}
            </div>
          )}
        </FetchData>
      </div>
    )
  }
}
```

Observers (resize, intersection, etc.): You can also use render props/scoped slots to create a component that acts as a wrapper around resize or intersection observers. This component can simply expose the current size or intersection point of an element to a parent component. You can then perform whatever logic you need based on that data in the parent, preserving a nice separation of concerns. Here is a base component that observes its own size and exposes its height and width to its parent:

```
class ObserveDimensions extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      width: null,
      height: null
    }
    this.elementToObserve = React.createRef()
  }

  componentDidMount(nextProps) {
    const erd = elementResizeDetectorMaker({ strategy: 'scroll' })

    erd.listenTo(this.elementToObserve.current, element => {
      this.setState({
        width: element.offsetWidth,
        height: element.offsetHeight
      })
    })
  }

  render() {
    return (
      <div className="observed-element" ref={this.elementToObserve}>
        {this.props.render({
          width: this.state.width,
          height: this.state.height
        })}
      </div>
    )
  }
}
```

We are using the [Element Resize Detector](#) library to listen to changes in our element size, and a React ref to get a reference to the actual DOM node.

We can then use this component quite easily in our app:

```
class App extends React.Component {  
  constructor(props) {  
    super(props)  
  }  
  
  render() {  
    return (  
      <div className="wrapper">  
        <ObserveDimensions  
          render={({width, height}) => (  
            <div>  
              Width: {width}px  
              Height: {height}px  
            </div>  
          )}  
        </div>  
      )  
    )  
  }  
}
```

The key to successfully creating reusable components using both render props and scoped slots is being able to correctly separate **behavior** from **presentation**. Each time you create a new UI component, think “What is the core behavior of this component? Can I use this anywhere else?”

Having a core set of renderless components that use render props can help you cut down on code duplication in your app and think more carefully about your core interface behaviors.



