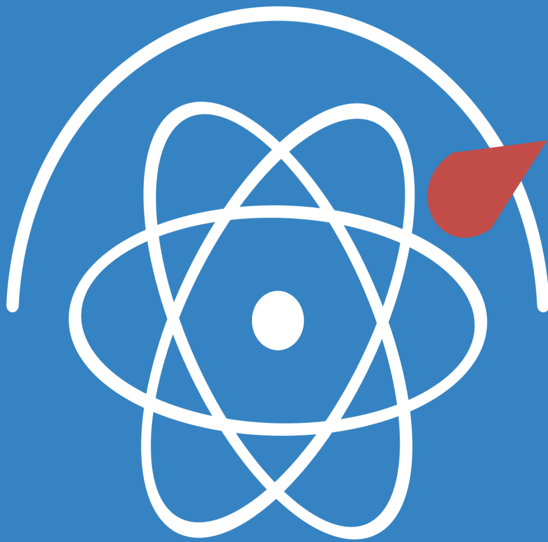
High-Performance
REACT



by

Thomas Hintz

High Performance React

Thomas Hintz

This book is for sale at
http://leanpub.com/high-performance-react

This version was published on 2021-01-21

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2020 - 2021 Thomas Hintz

Contents

Preface **1**

Introduction **2**

Foundations: Building our own React **5**

Components of React 6

Markup in JavaScript: JSX 8

Getting Ready to Render with createElement 12

Render: Putting Elements on the Screen 14

Reconciliation, or How React Diffs 16

Fibers: Splitting up Render 25

Putting it all together 26

Conclusion 28

Rendering Model **30**

React.memo 32

React.PureComponent 34

Adding support for memoization to our React . . . 34

Identifying & Diagnosing Bottlenecks **41**

Describing Performance Issues 42
Measuring 43

Reducing Renders 46

Improving DOM Merge Performance 47

Reducing Number of Components 48

Windowing 49

Performance Tools 50

JS Performance Tools 51

Code Splitting 52

Server Side Rendering 53

Concurrent Rendering 54

UX 55

JS Service Workers 56

Keys 57

Reconciliation 58

Preface

Introduction

It was the late 90's and I was just a kid visiting my Aunt and Uncle and their family in Denver. The days were packed with endless playing and goofing around. I didn't get to see my cousins much and we were having a good time. But it was the late 90's and the Internet was booming. And my cousin was in on it.

A "startup", that's what he called it. I didn't understand any of what he was saying about it. Grown-up stuff. Then he showed us the webpage for the startup and I thought that was impressive.

"How did you make that"? I asked him. I think he was a little confused at first about what I was even talking about but he quickly brought me over to the computer and showed me a screen full of text.

"You just type HTML, that's how you make the webpage." I thought this was the coolest.

"What do you type that into? What program is it? Can I do that?" He told me it was easy: just use Notepad. I wasn't going to let him go without some hook I could grab into this alien world. He told me it's really easy to learn: do an AOL search

for “HTML tutorial”.

So began my journey with web development. I AOL searched my way through as many blinking text tutorials as I could find. It wasn’t long until I was building AJAX. We had IE 5.5 and 6 and Mozilla Pheonix. And GMail came out. That changed things, now web apps were “legitimate.”

A lot of the technologies and libraries came and went over the years but one thing remained constant in large web apps: poor performance. From the very early days I was timing things with my stop watch. Sometimes things were slow and I had to understand why and how to fix them. Over the years I learned all about the browser’s DOM and its APIs and how they work. I learned how jQuery worked and backbone.js and all the rest. I made apps that didn’t lag or have jank.

I was able to do this because I understood the performance implications of the tools and libraries I was using and I learned how to measure performance. I had discovered the recipe for high-performance code.

And that is what this book is: a recipe for producing high-performance React applications. First, we learn how React works. Then we learn how to measure performance. And last we learn how to address the bottlenecks we find. Parts of any technical book will go stale as technology changes and that is no less true for this book. But what I hope you learn is not just the technical details but more importantly the method for

writing high-performance code. The API might change but the method will remain the same.

TODO note that the book references React-DOM but the algorithms should generally apply to all React implementations.

Foundations: Building our own React

Baking bread. When I first began to learn how to bake bread the recipe told me what to do. It listed some ingredients and told me how to combine them and prescribed times of rest. It gave me an oven temperature and a period of wait. It gave me mediocre bread of wildly varying quality. I tried different recipes but the result was always the same.

Understanding: that's what I was missing. The bread I make is now consistently good. The recipes I use are simpler and only give ratios and general recommendations for rests and waits. So why does the bread turn out better?

Before baking is finished bread is a living organism. The way it grows and develops and flavors depend on what you feed it and how you feed it and massage it, care for it. If you have it grow and ferment at a higher temperature and more yeast it overdevelops producing too much alcohol. If you give it too much time, acidity will take over the flavor. The recipes I used initially were missing a critical ingredient: the rising temperature.

But unlike a lot of ingredients: temperature is hard to control for the home cook. So the recipe can't just tell you exactly what temperature to grow the bread at. My initial recipes just silently made assumptions for the temperature, which rarely turn out to be true. This means that the only way to consistently make good bread is to have an understanding of how bread develops so that you can adjust the other ingredients to complement the temperature. Now the bread can tell me what to do.

While React isn't technically a living organism that can tell us what to do, it is, in its whole, a complex, abstract entity. We could learn basic recipes for how to write high-performance React code but they wouldn't apply in all cases, and as React and things under it change our recipes would fall out-of-date. So like the bread, to produce consistently good results we need to understand how React does what it does.

Components of React

The primary elements that make up any React program are its components. A `component` in React maintains local state and "renders" output to eventually be included in the browser's DOM. A tree of components is then created whenever a component outputs other components.

So, conceptually, React's core algorithm is very simple: it starts by walking a tree of components and building up a tree

of their output. Then it compares that tree to the tree currently in the browser's DOM to find any differences between them. When it finds differences it updates the browser's DOM to match its internal tree.

But what does that actually look like? If your app is janky does that explanation point you towards what is wrong? No. It might make you wonder if maybe it is too expensive to re-render the tree or if maybe the diffing React does is slow but you won't really know. When I was initially testing out different bread recipes I had guesses at why it wasn't working but I didn't really figure it out until I had a deeper understanding of how making bread worked. It's time we build up our understanding of how React works so that we can start to answer our questions with solid answers.

React is centered around the `render` method. The `render` method is what walks our trees, diffs them with the browser's DOM tree, and updates the DOM as needed. But before we can look at the `render` method we have to understand its input. The input comes from `createElement`. While `createElement` itself is unlikely to be a bottleneck it's good to understand how it works so that we can have a complete picture of the entire process. The more black-boxes we have in our mental model the harder it will be for us to diagnose performance problems.

Markup in JavaScript: JSX

`createElement`, however, takes as input something that is probably not familiar to us since we usually work in JSX, which is the last element of the chain in this puzzle and the first step in solving it. While not strictly a part of React, it is almost universally used with it. And if we understand it then `createElement` will be less of a mystery since we will be able to connect all the dots.

JSX is not valid HTML or JavaScript but its own language compiled by a compiler, like Babel. The output of that compilation is valid JavaScript that represents the original markup.

Before JSX or similar compilers, the normal way of injecting HTML into the DOM was via directly utilizing the browser's DOM APIs or by setting `innerHTML`. This was very cumbersome. The code's structure did not match the structure of the HTML that it output which made it hard to quickly understand what the output of a piece of code would be. So naturally programmers have been endlessly searching for better ways to mix HTML with JavaScript.

And this brings us to JSX. It is nothing new; nothing complicated. Forms of it have been made and used long before React adopted it. Now let's see if we can discover JSX for ourselves.

To start with, we need to create a data-structure – let's call it JavaScript Markup (JSM) – that both represents a DOM tree

and can also be used to insert one into the browser's DOM. And to do that we need to understand what a tree of DOM nodes is constructed of. What parts do you see here?

```
<div class="header">
  <h1>Hello</h1>
  <input type="submit" disabled />
</div>
```

I see three parts: the name of the tag, the tag's properties, and its children.

Name:	'div', 'h1', 'input'
Props:	'class', 'type', 'disabled'
Children:	<h1>, <input>, Hello

Now how could we recreate that in JavaScript?

In JavaScript, we store lists of things in arrays, and key/value properties in objects. Luckily for us, JavaScript even gives us literal syntax for both so we can easily make a compact DOM tree with our own notation.

This is what I'm thinking:

JSM - JavaScript Markup

```
['div', { 'className': 'header' },
  [['h1', {}, ['Hello']],
   ['input', { 'type': 'submit', 'disabled': 'disabled' }, []]
  ]
]
```

As you can see, we have a clear mapping from our notation, JSM, to the original HTML. Our tree is made up of three element arrays. The first item in the array is the tag, the second is an object containing the tag's properties, and the third is an array of its children; which are all made up of the same three element arrays.

The truth is though, if you stare at it long enough, although the mapping is clear, how much fun would it be to read and write that on a consistent basis? I can assure you, it is rather not fun. But it has the advantage of being easy to insert into the DOM. All you need to do is write a simple recursive function that ingests our data structure and updates the DOM accordingly. We will get back to that.

So now we have a way to represent a tree of nodes and we (theoretically) have a way to get those nodes into the DOM. But if we are being honest with ourselves, while functional, it isn't a pretty notation nor easy to work with.

And this is where our object of study enters the scene. JSX is just a notation that a compiler takes as input and outputs in its place a tree of nodes nearly identical to the notation we came

up with! And if you look back to our notation you can see that you can easily embed arbitrary JavaScript expressions wherever you want in a node. As you may have realized, that's exactly what the JSX compiler does when it sees curly braces!

There are three main differences between JSM and the real output of the JSX compiler: it uses objects instead of arrays, it inserts calls to React.createElement on children, and spreads the children instead of containing them in an array. Here is what real JSX compiler output looks like:

```
React.createElement(
  'div',
  { className: 'header' },
  React.createElement('h1', {}, 'Hello'),
  React.createElement(
    'input',
    { type: 'submit', 'disabled': 'disabled' })
);
```

As you can see, it is very similar to our JSM data-structure and for the purposes of this book we will use JSM, as it's a bit easier to work with. A JSX compiler also does some validation and escapes input to prevent cross-site scripting attacks. In practice though, it would behave the same in our areas of study and we will keep things simple by leaving those aspects of the JSX compiler out.

So now that we've worked through JSX we're ready to tackle `createElement`, the next item on our way to building our

own React.

Getting Ready to Render with createElement

React's `render` expects to consume a tree of element objects in a specific, uniform format. `createElement` is the method by which we achieve that objective. `createElement` will take as input JSM and output a tree of objects compatible with `render`.

React expects nodes defined as JavaScript objects that look like this:

```
{
    type: NODE_TYPE,
    props: {
        propA: VALUE,
        propB: VALUE,
        ...
        children: STRING | ARRAY
    }
}
```

That is: an object with two properties: `type` and `props`. The `props` property contains all the properties of the node. The node's `children` are also considered part of its properties. The full version of React's `createElement` includes more properties but they are not relevant to our study here.

```
function createElement(node) {
    // if array (our representation of an element)
    if (Array.isArray(node)) {
        const [ tag, props, children ] = node;
        return {
            type: tag,
            props: {
                ...props,
                children: children.map(createElement)
            }
        };
    }

    // primitives like text or number
    return {
        type: 'TEXT',
        props: {
            nodeValue: node,
            children: []
        }
    };
}
```

Our `createElement` has two main parts: complex elements and primitive elements. The first part tests whether `node` is a complex node (specified by an array) and then generates an `element` object based on the input node. It recursively calls `createElement` to generate an array of children elements. If the node is not complex then we generate an element of type 'TEXT' which we use for all primitives like strings and numbers. We call the output of `createElement` a tree of `elements` (surprise).

That's it. Now we have everything we need to actually begin the process of rendering our tree to the DOM!

Render: Putting Elements on the Screen

There are now only two major puzzles remaining in our quest for our own React. The next piece is: `render`. How do we go from our JSM tree of nodes, to actually displaying something on screen? To do that we will explore the `render` method.

The signature for our `render` method should be familiar to you:

```
function render(element, container)
```

This is the same signature as that of React itself. We begin by just focusing on the initial render. In pseudocode it looks like this:

```
function render(element, container) {
    const domElement = createDOMElement(element);
    setProps(element, domElement);
    renderChildren(element, domElement);
    container.appendChild(domElement);
```

Our DOM element is created first. Then we set the properties, render children elements, and finally append the whole tree to the container.

Now that we have an idea of what to build we will work on expanding the pseudocode until we have our own fully functional `render` method using the same general algorithm

React uses. In our first pass we will focus on the initial render and ignore reconciliation.

> Reconciliation is basically React's "diffing" algorithm. We will be exploring it after we work out the initial render.

```
function render(element, container) {
    const { type, props } = element;

    // create the DOM element
    const domElement = type === 'TEXT' ?
      document.createTextNode(props.nodeValue) :
      document.createElement(type);

    // set its properties
    Object.keys(props)
      .filter((key) => key !== 'children')
      .forEach((key) => domElement[key] = props[key]);

    // render its children
    props.children.forEach((child) => render(child, domElement));

    // add our tree to the DOM!
    container.appendChild(domElement);
}
```

The `render` method starts by creating the DOM element. Then we need to set its properties. To do this we first need to filter out the `children` property and then we simply loop over the keys, setting each property directly. Following that, we render each of the children by looping over them and recursively calling `render` on each child with the

`container` set to the current DOM element (which is each child's parent).

Now we can go all the way from our JSX-like notation to a rendered tree in the browser's DOM! But so far we can only add things to our tree. To be able to remove and modify the tree we need one more part: reconciliation.

Reconciliation, or How React Diffs

A tale of two trees. These are the two trees that people most often talk about when talking about React's "secret sauce": the virtual DOM and the browser's DOM tree. This idea is what originally set React apart. React's reconciliation is what allows you to program declaratively. Reconciliation is what makes it so we no longer have to manually update and modify the DOM whenever our own internal state changes. In a lot of ways, it is what makes React, React.

Conceptually, the way this works is that React generates a new element tree for every render and compares the newly generated tree to the tree generated on the previous render. Where it finds differences between the trees it knows to mutate the DOM state. This is the "tree diffing" algorithm.

Unfortunately, those researching tree diffing in Computer Science have not yet produced a generic algorithm with

sufficient performance for use in something like React; as the current best algorithm still runs in $O(n^3)$.

Since an $O(n^3)$ algorithm isn't going to cut it in the real-world, the creators of React instead use a set of heuristics to determine what parts of the tree have changed. Understanding how the React tree diffing algorithm works in general and the heuristics currently in use can help immensely in detecting and fixing React performance bottlenecks. And beyond that it can help one's understanding of some of React's quirks and usage. Even though this algorithm is internal to React and can be changed anytime its details have leaked out in some ways and are overall unlikely to change in major ways without larger changes to React itself.

According to the React documentation their diffing algorithm is $O(n)$ and based on two major components:

- Elements of differing types will yield different trees
- You can hint at tree changes with the `key` prop.

In this section we will focus on the first part: differing types. In a later chapter we will discuss and implement the `key` prop.

The approach we will take here is to integrate the heuristics that React uses into our `render` method. Our implementation will be very similar to how React itself does it and we will discuss React's actual implementation later when we talk about Fibers.

Before we get into the code changes that implement the heuristics it is important to remember that React *only* looks at an element's type, existence, and key. It does not do any other diffing. It does not diff props. It does not diff sub-trees of modified parents.

While keeping that in mind, here is an overview of the algorithm we will be implementing in the `render` method. `element` is the element from the current tree and `prevElement` is the corresponding element in the tree from the previous render.

```
if (!element && prevElement)
  // delete dom element
else if (element && !prevElement)
  // add new dom element, render children
else if (element.type === prevElement.type)
  // update dom element, render children
else if (element.type !== prevElement.type)
  // replace dom element, render children
```

Notice that in every case, except deletion, we still call `render` on the element's children. And while it's possible that the children will have their associated DOM elements reused, their `render` methods will still be invoked.

Now, to get started with our render method we must make some modifications to our previous render method. First, we need to be able to store and retrieve the previous render tree. Then we need to add code to compare parts of the tree to decide if we can re-use DOM elements from the previous

render tree. And last, we need to return a tree of elements that can be used in the next render as a comparison and to reference the DOM elements that we create. These new element objects will have the same structure as our current elements but we will add two new properties: `domElement` and `parent`. `domElement` is the DOM element associated with our synthetic element and `parent` is a reference to the parent DOM element.

Here we begin by adding a global object that will store our last render tree, keyed by the `container`. `container` refers to the browser's DOM element that will be the parent for all of the React derived DOM elements. This parent DOM element can only be used to render one tree of elements at a time so it works well to use as a key for `renderTrees`.

```
const renderTrees = {};
function render(element, container) {
    const tree =
      render_internal(element, container, renderTrees[container]);
    // render complete, store the updated tree
    renderTrees[container] = tree;
}
```

As you can see, the change we made is to move the core of our algorithm into a new function called `render_internal` and pass in the result of our last render to `render_internal`.

Now that we have stored our last render tree we can go ahead and update our render method with the heuristics for

reusing the DOM elements. We name it `render_internal` because it is what controls the rendering but takes an additional argument now: the `prevElement`. `prevElement` is a reference to the corresponding `element` from the previous render and contains a reference to its associated DOM element and parent DOM element. If it's the first render or if we are rendering a new node or branch of the tree then `prevElement` will be `undefined`. If, however, `element` is `undefined` and `prevElement` is defined then we know we need to delete a node that previously existed.

```
function render_internal(element, container, prevElement) {
    let domElement, children;
    if (!element && prevElement) {
        removeDOMElement(prevElement);
        return;
    } else if (element && !prevElement) {
        domElement = createDOMElement(element);
    } else if (element.type === prevElement.type) {
        domElement = prevElement.domElement;
    } else { // types don't match
        removeDOMElement(prevElement);
        domElement = createDOMElement(element);
    }
    setDOMProps(element, domElement, prevElement);
    children = renderChildren(element, domElement, prevElement);

    if (!prevElement || domElement !== prevElement.domElement) {
        container.appendChild(domElement);
    }

    return {
        domElement: domElement,
        parent: container,
```

```
        type: element.type,
        props: {
            ...element.props,
            children: children
        }
    };
}
```

The only time we shouldn't set DOM properties on our element and render its children is when we are deleting an existing DOM element. We use this observation to group the calls for `setDOMProps` and `renderChildren`. Choosing when to append a new DOM element to the container is also part of the heuristics. If we can reuse an existing DOM element then we do, but if the element type has changed or if there was no corresponding existing DOM element then and only then do we append a new DOM element. This ensures the actual DOM tree isn't being replaced every time we render, only the elements that change are replaced.

In the real React, when a new DOM element is appended to the DOM tree, React would invoke `componentDidMount` or schedule `useEffect`.

Next up we'll go through all the auxiliary methods that complete the implementation.

Removing a DOM element is straightforward; we just `removeChild` on the parent element. Before removing the element, React would invoke `componentWillUnmount` and schedule the cleanup function for `useEffect`.

```
function removeDOMElement(prevElement) {
    prevElement.parent.removeChild(prevElement.domElement);
}
```

In creating a new DOM element we just need to branch if we are creating a text element since the browser API differs slightly. We also populate the text element's value as the API requires the first argument to be specified even though later on when we set props we will set it again. This is where React would invoke `componentWillMount` or schedule `useEffect`.

```
function createDOMElement(element) {
    return element.type === 'TEXT' ?
      document.createTextNode(element.props.nodeValue) :
      document.createElement(element.type);
}
```

To set the props on an element, we first clear all the existing props and then loop through the current props, setting them accordingly. Of course, we filter out the `children` prop since we use that elsewhere and it isn't intended to be set directly.

```
function setDOMProps(element, domElement, prevElement) {
    if (prevElement) {
        Object.keys(prevElement.props)
            .filter((key) => key !== 'children')
            .forEach((key) => {
                domElement[key] = ''; // clear prop
            });
    }
    Object.keys(element.props)
      .filter((key) => key !== 'children')
      .forEach((key) => {
          domElement[key] = element.props[key];
    });
}
```

React is more intelligent about only updating or removing props that need to be updated or removed.

This algorithm for setting props does not correctly handle events, which must be treated specially. For this exercise that detail is not important and we leave it out for simplicity.

For rendering children we use two loops. The first loop removes any elements that are no longer being used. This would happen when the number of children is decreased. The second loop starts at the first child and then iterates through all of the children of the parent element, calling `render_internal` on each child. When `render_internal` is

called the corresponding previous element in that position is passed to `render_internal`, or `undefined` if there is no corresponding element, like when the list of children has grown.

```
function renderChildren(element, domElement, prevElement = { prop\
s: { children: [] }}) {
    const elementLen = element.props.children.length;
    const prevElementLen = prevElement.props.children.length;
    // remove now unused elements
    for (let i = elementLen; i < prevElementLen - elementLen; i++\
) {
        removeDOMElement(element.props.children[i]);
    }
    // render existing and new elements
    return element.props.children.map((child, i) => {
        const prevChild = i < prevElementLen ? prevElement.props.\
children[i] : undefined;
        return render_internal(child, domElement, prevChild);
    });
}
```

It's very important to understand the algorithm used here because this is essentially what happens in React when incorrect keys are used, like using a list index for a key. And this is why keys are so critical to high performance (and correct) React code. For example, in our algorithm here, if you removed an item from the front of the list you may cause every element in the list to be created anew in the DOM if the types no longer match up. Later on, in the chapter on keys, we will update this algorithm to incorporate keys. It's actually only a minor difference in determining which `child` gets paired

with which `prevChild`. Otherwise this is effectively the same algorithm React uses when rendering lists of children.

Example of `renderChildren` *2nd loop when the 1st element has been removed. In this case the trees for all of the children will be torn down and rebuilt.*

i	child Type	prevChild Type
0	span	div
1	input	span
2	-	input

There are a few things to note here. First, it is important to pay attention to when React will be removing a DOM element from the tree and adding a new one as this is when the related lifecycle events or hooks are invoked. And invoking those lifecycle methods or hooks, and the whole process of tearing down and building up a component is expensive. So again, if you use a bad key, like the algorithm here simulates, you'll be hitting a major performance bottleneck since React will not only be replacing DOM elements in the browser but also tearing down and rebuilding the trees of child components.

Fibers: Splitting up Render

The actual React implementation used to look very similar to what we've built so far, but with React 16 this has changed dramatically with the introduction of Fibers. Fibers are a

name that React gives to discrete units of work during the render process. And the React reconciliation algorithm was changed to be based on small units of work instead of one large, potentially long-running call to `render`. This means that React is now able to process just part of the render phase, pause to let the browser take care of other things, and resume again. This is the underlying change the enables the experimental Concurrent Mode as well as running most hooks without blocking the render.

But even with such a large change, the underlying algorithms for deciding how and when to render components is the same. And when not running in Concurrent Mode the effect is still the same as React does the render phase in one block still. So using a simplified interpretation that doesn't include all the complexities of breaking up the process in to chunks enables us to see more clearly how the process as a whole works. At this point bottlenecks are much more likely to occur from the underlying algorithms and not from the Fiber specific details. In the chapter on Concurrent Mode we will learn more about Fibers.

Putting it all together

Throughout the rest of the book we will be building on and using our React implementation so it would be helpful to see

it all put together and working. At this point the only thing left to do is to create some components and use them!

```
const SayNow = ({ dateTime }) => {
    return ['h1', {}, [`It is: ${dateTime}`]];
};

const App = () => {
    return ['div', { 'className': 'header' },
             [SayNow({ dateTime: new Date() }),
              ['input', { 'type': 'submit', 'disabled': 'disabled'\
 }, []]
             ]
           ];
}

render(createElement(App()), document.getElementById('root'));
```

We are creating two components, that output JSM, as we defined it earlier. We create one component prop for the `SayNow` component: `dateTime`. It gets passed from the `App` component. The `SayNow` component prints out the `DateTime` passed in to it. You might notice that we are passing props the same way one does in the real React, and it just works!

The next step is to call render multiple times.

```
setInterval(() =>
  render(createElement(App()), document.getElementById('root')),
  1000);
```

If you run the code above you will see the DateTime display being updated every second. And if you watch in your dev

tools or if you profile the run you will see that the only part of the DOM that gets updated or replaced is the part that changes (aside from the DOM props). We now have a working version of our own React.

This implementation is designed for teaching purposes and has some known issues and bugs, like always updating the DOM props, along with other things. Fundamentally, it functions the same as React but if you wanted to use it in a more production setting it would take a lot more development.

Conclusion

Of course our version of React elides over many details that React must contend with, like starting a re-render from where state changes and event handlers. For understanding how to build high-performance React applications, however, the most important piece to understand is how and when React renders components, which is what we have learned in creating our own mini version of React.

At this point you should have an understanding of how React works. In the rest of the book we are going to be refining this model and looking at practical applications of it so that we are

prepared to build high performance React applications and diagnose any bottlenecks.

Rendering Model

Now that we have a firm understanding of the underpinnings of React we can begin to look at potential bottlenecks and their solutions. We'll start with a little quiz about how React chooses when to render a component.

TODO insert img-tree of components

In figure 1, if state changes in component A but nothing changes in B will React ask B to re-render?

Yes. Absolutely. Always, unless `shouldComponentUpdate` returns false, which is not even an option with functional components and is discouraged for class based components. So if we have a large tree of components and we change state high in the tree React will be constantly re-rendering large parts of the tree. (This is common because app state often has to live up high in the tree because props can only be passed down.) This is clearly very inefficient so why does React do it?

If you remember back to when we implemented the render algorithm you'll recall that React does nothing to see if a component actually needs to re-render, it only tests whether DOM elements need to be replaced or removed. Instead React

always renders all children. React is effectively off-loading the descision to re-render to the components themselves because a general solution has poor performance.

Originally React had `shouldComponentUpdate` to solve this issue but the developers of React found that for users implementing it correctly was difficult and error prone. Programmers would add new props to a component but forget to update `shouldComponentUpdate` with the new props causing the component to not update when it should which led to strange and hard to diagnose bugs. So if we shouldn't use `shouldComponentUpdate` what tools are we left with?

And it's a great question because unneeded renders can be a massive bottleneck, especially on large lists of components. In fact, there is no other way to control renders; React will always render.

But there is still hope. While we can't control if our component will render, what if instead of just always re-running all of our render code on each render, we instead kept a copy of the result of the render and next time React asks us to re-render we just return the result we saved? Now that, with two modifications, is exactly what we will do.

TODO Note: this stops full tree from re-rendering

Obviously we can't just render once and then forever return that result because our state and props might change. So we

also need to track the state and props and only return our cached result if they haven't changed.

As you may have already noticed this is a common solution in Computer Science for such problems: memoization. What we want is to memoize our components.

TODO Note: explain memoization

This is indeed such a common bottleneck and solution that React provides an API to facilitate it.

We will learn about this API by first looking at the signatures of the React API itself, then we will extend our React implementation from chapter one to support the same API. Then we will discuss its usage and analyze when and how to use it.

React.memo

The first API React provides that we will look at is `React.memo`. `React.memo` is a higher-order component (HOC) that wraps your functional component. It handles partially memoizing your component based on its props (not state). If your component contains `useState` or `useContext` hooks it will re-render when the state or context changes.

It is important to note that while React named their function "memo" it is more like a partial memoization compared to the

usual definition of memoization. Normally in memoization when a function is given the same inputs as a previous invocation it will just return a stored result, however, with React's `memo` only the *last* invocation is memoized. So if you have a prop that alternates between two different values React's `memo` will always re-render the component whereas with traditional memoization the component would only ever get rendered twice in total.

Here is the signature for `React.memo`:

```
function (Component, areEqual?) { ... }
```

It takes two arguments, one required and one optional. The required argument is the component you want to memoize. The second and optional argument is a function that allows you to tell React when your component will produce the same output.

If the second argument is not specified then React performs a *shallow* comparison between props it has received in the past and the current props. If the current props match props that have been passed to your component before, React will use the output stored from that previous render instead of rendering your component again. If you want more control over the prop comparison, like if you wanted to deeply compare some props, you would pass in your own `areEqual?`. However, it's generally recommended to program in a more pure style

instead of using `areEqual?` because it can suffer from the same problem that `shouldComponentUpdate` did.

React.PureComponent

`React.PureComponent` is very similar to `React.memo`, but for class based components. Like `React.memo`, `React.PureComponent` memoizes the component based on a shallow comparison of its props and state.

Here is the signature for `React.PureComponent`:

```
class Pancake extends React.PureComponent {
    ...
}
```

Adding support for memoization to our React

Implementing full-blown memoization would be outside the scope of this book but since React only memoizes the last render it is quite easy for us to add `memo` support.

The most interesting part of the `memo` implementation is the default `areEqual` implementation. This is the implementation components will use if they don't provide their own. To see if `memo` can return a previous render or not it compares the props to see if they are the same use the following `defaultAreEqual` function. This what that looks like:

```
function defaultAreEqual(oldProps, newProps) {
    if (typeof oldProps !== 'object' || typeof newProps !== 'obje\
ct') {
        return false;
    }

    const oldKeys = Object.keys(oldProps);
    const newKeys = Object.keys(newProps);

    if (oldKeys.length !== newKeys.length) {
        return false;
    }

    for (let i = 0; i < oldKeys.length; i++) {
        // Object.is - the comparison to note
        if (!oldProps.hasOwnProperty(newKeys[i]) ||
            !Object.is(oldProps[newKeys[i]], newProps[newKeys[i]]\
)) {
            return false;
        }
    }

    return true;
}
```

`oldProps` and `newProps` are objects containing the previous render's props and the current render's props. Much of the function is just boilerplate to ensure the prop objects are the same type and shape. The important part is noted in the loop where we use JavaScript's `Object.is` method to compare each prop object's values.

If you're not familiar with `Object.is`, it is nearly the same as the identity operator `===` except it treats `-0` and `+0` as equal but not does not treat `Number.NaN` as equal to `NaN`.

It is important to notice that if a prop value is an object then we are *not* testing its contents, only whether the objects themselves are the same object or not. For example, if we have two props `a` and `b` set to the following objects that look the same they will cause `defaultAreEqual` to return false.

```
const a = { x: 1 };
const b = { x: 1 };
Object.is(a, b); // false
```

Even though `a` and `b` look like the same object they are in fact instances of two different objects and will therefore cause `memo` to not find a match and your component will re-render. Using object literals, like in the example above, as prop values is a very common pattern that will “break” memoization of a component.

This is also a potential pitfall in another way:

```
const a = { x: 1 };
const b = a;
Object.is(a, b); // true
a.x = 2;
Object.is(a, b); // true
```

In this example it may be obvious that `a` still equals `b` at the end but in React applications this is often less clear because the object being used as a prop is coming from somewhere else. The lesson to watch out for is that if you pass the same object to a memoized component while changing that object’s

contents between renders the memoized component won't know that the contents have changed and will instead return a cached render instead of doing what you probably are expecting: re-rendering. So the overall lesson when using objects in props with memoized components is that objects with the same contents should be the same object and objects with different contents should be different objects. If managing this is a problem in your application there are immutability libraries that you can use that can help out.

As you can see there is a cost to memoizing a component both in computer resources and programmer effort so it is important to only apply memoization when a component needs it and will benefit from it.

If a component only renders a few times or infrequently it is not a good candidate for memoization since it is unlikely that a memoized render result will get returned and even if it does it is unlikely to make up for the cost of implementing and using it unless its rendering process is unusually computationally intense.

Another case when memoization is not a good idea is when the props for a component are not often the same as a previous render. Like take, for example, a component that renders the current hours, minutes, and seconds and receives those inputs as props. Unless you're rendering that component multiple times per second the props will never be the same as a

previous render. So if you were to memoize that component you would be using CPU cycles for the memoization process and filling up memory with render results without ever being able to re-use a render.

Here some rules for working with memoized components:

- Don’t use object literals
- Don’t modify objects
- Objects with the same contents should be the same instance
- Use memoization: on components that get called frequently
- Use memoization: when props will often be the same for multiple renders in succession
- Use memoization: to prevent part of the component tree from re-rendering

And finally we have the `memo` implementation:

```
function memo(component, areEqual = defaultAreEqual) {
    let oldProps = [];
    let lastResult = false;
    return (props) => {
        const newProps = propsToArray(props);
        if (lastResult && areEqual(oldProps, newProps) {
            return lastResult;
        } else {
            lastResult = component(props);
            oldProps = newProps;
            return lastResult;
        }
    };
}
```

`memo` is quite straightforward. We just store the previous props and result and if the new props match the old props we return the last result. In React this is also connected to the `useState` and `useContext` hooks so that whenever state is changed a re-render is forced and the result stored.

Of course, you can provide your own `areEqual` implementation instead of using the default shallow comparison version. When might this make sense and are there any performance considerations in doing so?

The default shallow comparison method is relatively fast so by itself it is unlikely to be a performance bottleneck so the only reason to implement your own version is if you want `areEqual` to do a deeper comparison of the props, like comparing the contents of objects passed as props or the contents of arrays. You could just write your own implementation that does more involved comparisons on the props that you want it

to but that is also a potential pitfall. Like if another developer adds a new prop to the component but doesn't realize there is a custom `areEqual` implementation the component will break since it won't detect when the new prop has a new value and therefore won't trigger a new render. A better approach is to use a generic deep comparison procedure that does a deep comparison on all props but this can easily become a performance bottleneck so use it with care (and is likely the reason React doesn't use it by default).

TODO useCallback

Identifying & Diagnosing Bottlenecks

When your application is not performing as you would like or expect, what can you do? What follows is my general approach to solving performance bottlenecks with a focus on React specific tools. There is an element of creativity to the process though so you should take this more as a guide and not something set in stone. You can start here but also try to find what works best for you.

These are the six main steps I use to solve performance bottlenecks:

- Describing the performance bottleneck
- Measuring the bottleneck
- Identifying the source
- Diagnosing the cause
- Generating possible solutions
- Selecting and implementing a solution

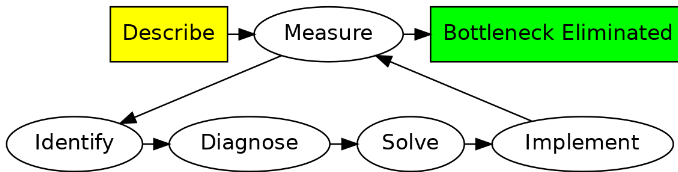


Figure 1: Solving Bottlenecks Cycle

As you can see in Figure 1, I use them in a cycle centered around measuring the bottleneck so that I'll be able to track my progress and know when I have eliminated the bottleneck.

Describing Performance Issues

The first step is to identify and qualify the bottleneck. You can start by asking "what are the symptoms?" and "what triggers them?". It's very important to dig in as much as possible at this stage and gather as much information as possible. Here are the questions I generally use to get you started:

- What specifically is the issue?
  - A lack of responsiveness?
  - Temporary jankiness?
  - What am I/the user being prevented from doing?
- When does the issue start?
- When does it finish?
- Is the intensity and/or duration variable or constant?
- Is it predictable? Does it always happen?

- Does it seem like anything triggers it?

You can start with trying to answer these questions or you can come up with your own. The only way to get good at it is practice as it's more an art than a science, at this stage.

This stage might not seem that important or the answers might seem obvious but being thorough here can actually save you a lot of time later on. It's very easy to misunderstand a bottleneck and then begin your investigation in the wrong place, wasting valuable time, or even worse, crafting a solution to the wrong problem.

Measuring

Once you've described the performance issue in as much detail as you can it's time to move on to the next stage: measuring the issue. This stage is vital to the process. Do not skip this stage. The only way later on to ensure you've actually fixed the problem is to measure the problem to the best of your ability. The better you can quantify the issue the easier the rest of the process will be. This can be very challenging, especially with things that seem immeasurable, like UI jankiness but if you work at it there is generally a way to do. We will discuss a few techniques coming up.

One technique I use is very simple: timing a section of code and logging the results to the console. Most modern browsers

now support the PerformanceNavigationTiming API which includes many tools to make this process easier but doing it without the API works too. I often start logging an expansive amount of my code and then moving my measurement locations in closer to each other until they have either isolated a section of code or eliminated a section of code from the possibilities.

While the logging times can be very useful for some bottlenecks, they can also be more cumbersome for a lot of React bottlenecks because it can be a lot of work to insert your timing calls in the right location when the bottleneck appears to be coming from somewhere in a tree of React components. React provides a tool for this though: a profiler. A profiler is a tool, usually in conjunction with a compiler, that effectively inserts many timers all over your code and then compiles the results in to charts so it's easier to pinpoint issues.

Profilers don't usually insert "timers" instead using sampling or other techniques but the important part for us here is just knowing that they can provide insight into the performance of your program and that they can have an effect on performance themselves.

To use the React profiler you will first need to ensure that you have installed the React developer tools plugin for the browser you are using. After that you must ensure your build is `in-`

`strumented` (meaning it is setup for use with the profiler). With create-react-app this will be when in development mode. If you are unsure check out the documentation for the React profiler as well as your build tools, like webpack.

Reducing Renders

Improving DOM Merge Performance

Reducing Number of Components

higher-order components

Windowing

Performance Tools

trace from scheduler/tracing/profiler component

JS Performance Tools

Code Splitting

React.lazy, suspense

use on routes

how to handle updates of assets that have new names?

Server Side Rendering

Concurrent Rendering

UX

JS Service Workers

Keys

Reconciliation

- diffing algorithm based on heuristics. generic algorithm is $O(n^3)$
- "Fiber" algorithm notes
  - lists reordering without key means full list output/update
  - type changes cause full re-render
  - keys should be stable, predictable, unique